UNIVERSITÀ DI PISA

DIPARTIMENTO DI INFORMATICA
DOTTORATO DI RICERCA IN INFORMATICA

PH.D. THESIS

A Control-Theoretic Methodology for Adaptive Structured Parallel Computations

Gabriele Mencagli

SUPERVISOR

Marco Vanneschi

May 28, 2012

Dipartimento di Informatica, Largo B. Pontecorvo, 3, I-56127 Pisa, Italy

Contents

List of Figures

List of Tables

To the memory of my Grandfather,
who lived in honesty and simplicity
giving me an example to follow.

Abstract

ADAPTIVITY for distributed parallel applications is an essential feature whose importance has been assessed in many research fields (e.g. scientific computations, large-scale real-time simulation systems and emergency management applications). Especially for high-performance computing, this feature is of special interest in order to properly and promptly respond to time-varying QoS requirements, to react to uncontrollable environmental effects influencing the underlying execution platform and to efficiently deal with highly irregular parallel problems. In this scenario the *Structured Parallel Programming* paradigm is a cornerstone for expressing adaptive parallel programs: the high-degree of composability of parallelization schemes, their *QoS predictability* formally expressed by *performance models*, are basic tools in order to introduce dynamic reconfiguration processes of adaptive applications. These reconfigurations are not only limited to implementation aspects (e.g. parallelism degree modifications), but also parallel versions with different structures can be expressed for the same computation, featuring different levels of performance, memory utilization, energy consumption, and exploitation of the memory hierarchies.

Over the last decade several programming models and research frameworks have been developed aimed at the definition of tools and strategies for expressing adaptive parallel applications. Notwithstanding this notable research effort, properties like the optimality of the application execution and the stability of control decisions are not sufficiently studied in the existing work. For this reason this thesis exploits a pioneer research in the context of providing formal theoretical tools founded on *Control Theory* and *Game Theory* techniques. Based on these approaches, we introduce a formal model for controlling distributed parallel applications represented by computational graphs of structured parallelism schemes (also called skeleton-based parallelism).

Starting out from the performance predictability of structured parallelism schemes, in this thesis we provide a formalization of the concept of adaptive parallel module performing structured parallel computations. The module behavior is described in terms of a *Hybrid System* abstraction and reconfigurations are driven by a *Predictive Control* approach. Experimental results show the effectiveness of this work, in terms of execution cost reduction as well as the stability degree of a system reconfiguration: i.e. how long a

reconfiguration choice is useful for targeting the required QoS levels.

This thesis also faces with the issue of controlling large-scale distributed applications composed of several interacting adaptive components. After a panoramic view of the existing control-theoretic approaches (e.g. based on decentralized, distributed or hierarchical structures of controllers), we introduce a methodology for the distributed predictive control. For controlling computational graphs, the overall control problem consists in a set of coupled control sub-problems for each application module. The decomposition issue has a twofold nature: first of all we need to model the *coupling relationships* between control sub-problems, furthermore we need to introduce proper notions of negotiation and convergence in the control decisions collectively taken by the parallel modules of the application graph. This thesis provides a formalization through basic concepts of *Non-cooperative Games* and *Cooperative Optimization*. In the notable context of the distributed control of performance and resource utilization, we exploit a formal description of the control problem providing results for equilibrium point existence and the comparison of the control optimality with different adaptation strategies and interaction protocols. Discussions and a first validation of the proposed techniques are exploited through experiments performed in a simulation environment.

Chapter 1

Introduction

Distributed parallel applications are complex sets of specialized software components which cooperate in such a way as to perform a global goal in a distributed manner. Cooperation can be exploited by transmitting and receiving data or by accessing shared data structures according to the used cooperation model [1] (e.g. message-passing or shared memory). Based on the frequency of cooperation activities, the coupling degree between distributed components can be quite different, spreading from loosely-coupled systems, like in Grid [2, 3] and Cloud Computing [4] applications, to tightly-coupled computations as in the case of high-performance computing (e.g. scientific and engineering applications) and real-time systems as multimedia streaming.

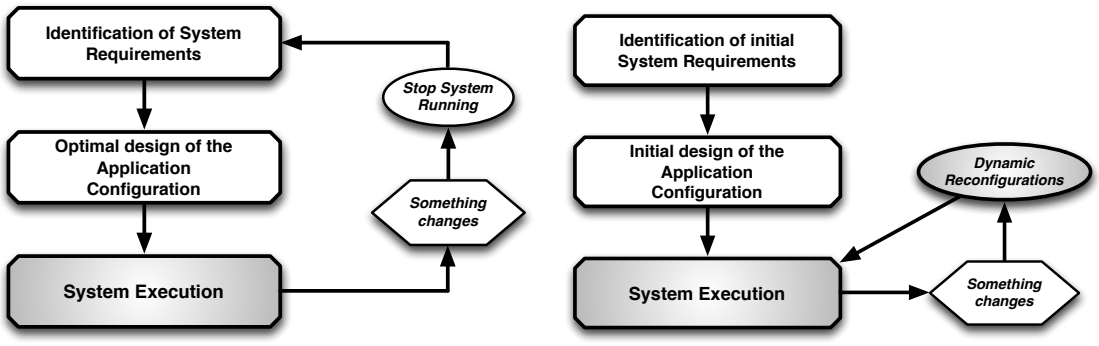
Over the last decades the scientific community has been involved in facing with the programming issues of distributed and parallel systems, aimed at the definition of increasingly complicated applications featuring a high complexity in the number of distributed components, in the spatial distribution and cooperation between interested parties and in their degree of heterogeneity. Well-studied issues are *security* of component identification and information transmission, and *fault-tolerance*, which is especially useful for designing robust systems as in the case of highly distributed and mobile architectures as next-generation Grids [5] and financial processing platforms. Hardware and software *heterogeneity* further complicates the existing problems in distributed systems. For instance component placement and interconnection may depend on processor speed and type, the set of operating system services that are available on different computing resources and the bandwidth and latency provided by interconnection networks.

In addition to the previous issues, recently the research and developmental trend in distributed and parallel computing has been focused on a further critical objective. The wide-ranging composition of distributed platforms in terms of different classes of computing nodes and network technologies, the strong diffusion of mission-critical applications which require real-time elaborations and online compute-intensive processing as in the case of Emergency Management Systems, Homeland Security and I-Transportation, lead to the emerging of systems featuring properties like self-managing, self-organization, self-controlling and strictly speaking ***adaptivity***. Adaptivity is an essential property that

modern distributed parallel systems should provide, and how to render this feature in different environments and for several classes of applications (e.g. parallel programs, mobile and pervasive applications) is an open research problem that still requires further research efforts. In the last years the study of these issues has laid the groundwork for a new model of computing called *Autonomic Computing* [6].

The adaptive management of the distributed system behavior is often a crucial prerequisite for an execution that accomplishes desirable results. The adaptive behavior is often driven by the user expectation of the system behavior. Although with different meanings, this concept is known as *Quality of Service* (QoS): i.e. a set of metrics reflecting the experienced behavior of the system, as quantitative metrics describing the resource consumption e.g. in terms of memory and power usage, and the performance level achieved by the application (e.g. the system response time to user requests), as well as qualitative metrics reflecting abstract properties like the user's degree of satisfaction or the precision of the computed results.

Making completely static assumptions, in which we suppose the existence of QoS requirements fixed by the users, the presence of reliable and dedicated execution platforms and fully predictable computations (e.g. featuring constant or regular workloads), we are usually able to define at design time the best application *configuration* [7, 8] that achieves the required QoS levels. For configuration we mean a specific identification of the application components, their mapping onto the available computing resources, the specification of their internal behavior (e.g. if they exploit a sequential or a parallel elaboration) and the way in which the computations are performed (i.e. the algorithms for sequential components or the parallelization approach for a parallel computation). If something of the initial requirements changes, the application is stopped, a new optimal configuration identified, and the application is deployed and restarted again (this behavior is schematized in Figure 1.1a).



(a) Static approach. (b) Dynamic approach.

Figure 1.1: Static and Dynamic approaches to changes in the application QoS requirements.

In distributed computational environments like Grids and Clouds the previous situation is no longer acceptable. Such environments are not necessarily dedicated to a single

parallel or distributed application, but the number and features of nodes and network facilities may vary significantly and in unpredictable ways. Moreover the application behavior can vary over time changing processing loads and communication demands as for irregular parallel problems. In these cases achieving the desired QoS requires applications with appropriate support for *reconfiguration* and adaptability to the dynamic nature of computations and execution environments. Moreover the applications (e.g. the runtime support of a high-level programming model) need to be able to identify violations of QoS constraints, potentially dangerous events or unsatisfactory execution conditions and dynamically re-design and re-deploy the optimal configuration without stopping the execution (see Figure 1.1b).

In this chapter we introduce the main issues concerning adaptivity for distributed parallel applications, we discuss the features of existing research work and we give a brief overview of the methodology provided in this thesis.

1.1 Adaptivity issues in Distributed Parallel Applications

Over the years the research effort has focused on the central issues of providing adaptivity and reconfiguration support for distributed parallel applications. A first question concerns *how a distributed application can change its behavior during the execution*. This aspect implies the research in general classes of dynamic reconfigurations that can be applied to the system at run-time. Moreover, another more theoretical issue investigates the logics and the methodological tools behind the definition of adaptation strategies: i.e. *how reconfigurations are selected in response to critical and unexpected execution conditions*.

Based on existing reviews on adaptivity for distributed parallel computations [9, 10, 11, 12, 13, 14], we can classify the dynamic reconfiguration processes in four broad categories:

- **Geometrical Changes** affect the mapping between the internal structure of an application component and the system resources on which it is currently executed. Such class of changes consist in migrating a component or some of its internal implementation processes in response to specific conditions, e.g. when a system resource fails or a new node is included in the system. These reconfigurations can be useful for load-balancing or for improving the service reliability;

- **Structural Changes** affect the internal structure of an application component. A notable case is when an internally parallel component changes the number of its processes or threads. This occurs when the component actually provides poor performance or if a task scheduler changes the workload of a specific component;

- **Implementation Changes** are modifications of the behavior of an application component, which changes completely its internal algorithm and/or the parallelization pattern but preserving the elaboration semantics and its input and output interfaces;

- **Interface Changes** are intensive modifications of the component behavior, which changes during the execution the set of its external operations or services offered to the other components or to the users.

Dynamic reconfigurations are intrusive actions which may induce semantic inconsistencies [13, 15, 16] as well as performance degradations if not properly dealt with. To prevent the occurrence of these problems, reconfigurations must be executed according to specific optimized protocols in order to minimize the reconfiguration cost, e.g. the downtime due to change the application configuration, and not compromise the correctness and the consistency of the computation.

A solution consists in putting everything concerning the adaptive behavior inside the run-time support of a high-level programming model, in such a way as to completely hide these aspects from the programmer's viewpoint. This *transparent approach* to reconfigurations requires a deep knowledge of the application structure, the communication patterns and the distributed data layout in such a way that the reconfiguration code can be automatically extracted by a static process of compilation of the program. In this scenario the programmer is freed from directly programming the reconfiguration phase, but it is involved in defining, by means of proper directives or programming constructs, the reconfigurations that are admissible. On the other hand an opposite approach consists in programmers directly involved in defining and implementing the reconfiguration activities. For each possible reconfiguration the programmer must provide the implementation that correctly performs the dynamic reconfiguration. In this case the programming problem is likely to explode and in many cases reconfigurations can only be expressed for limited and well-defined situations.

A reconfiguration should be taken only if the new resulting system configuration improves the achievement of the QoS requirements. The correspondence between reconfigurations and QoS effects can be determined by the programmer's experience. This is the case when a control logic is provided through policy rules, that specify a precise mapping between unexpected situations and corresponding sequences of reconfigurations on the application components. In this case it is difficult to ensure the *stability of control decisions*, i.e. taking a control action with the reasonable expectation that this choice will be the best solution to target the required QoS for a sufficiently long temporal horizon. Ineffective adaptation strategies can make the system to oscillate performing a large sequence of reconfigurations in a relatively short time period, that implies the system to spend more time in modifying itself instead of accomplishing its goals. Moreover, only with policy rules it may be hard to cover all the possible combinations of event-action pairs, especially when simultaneous critical conditions imply to come to an agreement in the control decisions taken by different application components.

Improving these aspects requires a more effective and rigorous *model-driven approach* to the control logic design, in which the complete behavior of the system is modeled through mathematical and formal tools able to abstractly reproduce the effects of sequence of reconfigurations over the quality of the system execution. Control-theoretic techniques, as optimal and predictive control, have these potentialities and, though their exploitation

is more complex than policy rules, their parametric definition can allow the system to reach a better degree of optimality and stability of its control decisions. Therefore in this thesis we have investigated these techniques in order to design a novel methodology for adaptive distributed parallel applications.

1.2 Existing Approaches and the proposed Methodology

In this section we start by giving a brief overview of the existing techniques and frameworks for developing adaptive distributed applications, then we provide a description of the motivations that have led to the approach proposed in this thesis.

1.2.1 Brief review of the literature

Adaptivity and dynamic reconfigurations have been studied in different research areas, from mobile and pervasive computing, in which the main objective is to provide seamless services to the users in face of changes in the execution platform and guessing user intentions, to mission-critical applications that require the satisfaction of performance constraints in highly heterogeneous scenarios (e.g. emergency management systems).

A general issue is to take advantage of the knowledge about the underlying execution platform and the QoS requirements in order to design and deploy the best application configuration. The definition of applications as a composition of well-known parallel patterns [17] makes it possible to study formal performance models and thus the development of optimized configurations for each specific situation. Consequently many languages [18, 19] and tools [20, 21] exist, able to analyze the high-level description of a parallel application and to produce an optimized implementation for specific platforms and QoS requirements.

For dynamic environments as Grids [2] and when parallel computations feature statically unpredictable workloads and load unbalancing, optimizations based on a static knowledge are no more effective: applications must be able to change their structure at run-time [22, 6]. In this scenario the exploitation of structured parallelism schemes makes it possible to define adaptive applications in which reconfiguration activities are completely transparent to the application programmer [13, 23, 24]. This approach is completely different from non-structured parallel programming models [25, 26], in which reconfiguration mechanisms are developed entirely by the programmer, who is also involved in preserving the computation consistency during the reconfiguration processes. Moreover the adaptive logic considers the application as a unstructured group of processes without any knowledge about their communication pattern, making impossible the definition of performance models. For this reason reconfiguration decisions must be taken without any precise knowledge about their impact in the overall performance.

Although structured parallelism schemes can be analyzed through performance models, this feature is only a precondition to apply effective reconfiguration strategies. In existing programming models for structured parallel computations [24, 27, 28, 25, 26]

adaptation strategies are rather simple, as in the case of policy-based rules (e.g. event-condition-action rules). The design of such adaptation strategies is usually an ad-hoc solution feasible only for relatively simple systems (in terms of number of components and their interconnections). Moreover policy-based strategies are based on a deep experience of the programmer, which is not always a practical situation. This research work does not address the achievement of properties like the *application optimality*, i.e. assuring the satisfaction of the QoS constraints in the long-term execution, and the *reconfiguration stability*, i.e. avoiding oscillating behaviors and minimizing the number of reconfigurations. To focus on these issues the research community has tried to apply different methodologies from heterogeneous research areas. In particular interesting approaches are based on Control Theory [29, 30], Artificial Intelligence [31] and Agent-based techniques [32].

Control theory approaches are considered difficult to be applied to computing systems, especially because they require a formal mathematical model of the controlled system. Notable examples are described in [29] for controlling the IBM Lotus Domino and the Apache Web Server. This work exploits statistical models for representing the system dynamics. The exploitation of a dynamical or static model of the target system is extremely important to predict future violations of execution thresholds and to decide, in a proactive way, reconfigurations that guarantee the QoS requirements. However existing work does not assume any knowledge about the computing structure, thus empirical or statistical approaches are needed to instantiate system models through initial experimental data.

In this thesis we propose a novel approach for controlling distributed parallel computations featuring well-known structures in terms of parallel computations and their composition in computation graphs of distributed components. Starting out from this knowledge, we apply control strategies inherited from control theory in order to provide powerful adaptation strategies and discuss their effectiveness. In the next section we list the basic features of this methodology.

1.2.2 Basic features of the Methodology proposed in the thesis

In this section we summarize the basic features characterizing the methodology proposed in this thesis. In the first part we introduce the general context in which we have formulated our approach and we present a performance modeling technique for studying the behavior of parallel computations. This first part, which provides a set of preliminary notions for the rest of the thesis, is composed of two main aspects:

Requirements identification: in our vision a distributed parallel application is composed of application components interconnected in generic acyclic or cyclic (request-reply) graph structures. While the structure of the application graph is initially studied without special constraints, we suppose that each component executes a parallel computation expressed through well-known structured parallelism schemes (e.g. task-farm and data-parallel paradigms) for which the structure of the computations and the interaction pattern between processes are fully identified and known. Moreover we introduce two classes of dynamic reconfigurations for structured parallel

computations and we discuss existing adaptation strategies and their limitations.

Performance modeling: we investigate formal performance models for structured parallelism schemes and for evaluating the performance of graph structures, in which components interact according to a blocking communication semantics. The performance models and algorithms developed in this thesis are based on Queueing Networks theory, but they focus on modeling issues that are not clearly addressed in existing research work in performance evaluation. Starting from the initial information about the ideal performance behavior of each component, we provide algorithms and properties for understanding how the graph stabilized to a steady-state behavior, and what is the real performance behavior of each component.

The second part of the thesis focuses on the design of effective adaptation strategies exploiting techniques inherited from control theory and optimal control. The effort is to adapt existing approaches presented in different scenarios and formalizing the control of a generic adaptive parallel module, that represents the basic block of our approach. To instantiate such adaptation techniques, we exploit the performance modeling presented in the previous part of the thesis. This part can be summarized in the following aspects:

ParMod control strategies: we formalize adaptation strategies for a single adaptive parallel module (ParMod), organized in terms of an Operating Part and a Control Part interconnected in a closed-loop scheme. We describe the Operating Part in terms of a hybrid system featuring a multi-configuration behavior: i.e. each configuration corresponds to a specific model that describes the QoS reached by the configuration over the execution. The definition of these models is given using the performance modeling tools presented in the first part of the thesis. We propose two distinct adaptation strategies. The first one inherits from discrete event controllers and consists in a *reactive approach*, in which reconfigurations are driven by the presence of specific events generated by evaluating actual monitored data from the observed computation. The second approach exploits in a better way the performance models of structured parallel computations, in order to provide a *predictive strategy* that selects an optimal sequence of reconfigurations optimizing an objective function. In this case we discuss the exploitation of control-theoretic techniques as the *Model-based Predictive Control* (shortly MPC), in which control decisions are taken solving a receding-horizon optimization scheme.

Experimental validation: we develop experimental scenarios in which the adaptive behavior of a ParMod is exploited through the reactive and the predictive adaptation strategies. The examples, based on real-world parallel applications, provide an experimental validation of these techniques in terms of application optimality (e.g. minimization of the long-term operating cost) as well as stability degree of reconfigurations (minimizing the number of reconfigurations performed during the execution).

The last part of the thesis studies the problem of multiple adaptive ParMods interconnected in computation graph structures. In this case, besides control-theoretic techniques, we investigate the exploitation of mathematical frameworks as Game Theory and Distributed Optimization. This part can be described by the following points:

Distributed control schemes: we present the problem of controlling multiple adaptive components of the same application. Centralized schemes are initially studied in order to highlight their infeasibility for large-scale systems and, thus, the necessity of decomposing the control problem into multiple interconnected controllers. At this regard we discuss different implementations, as single-layer or multi-layer controller organizations. We provide an extension of the ParMod predictive strategy, in which controllers communicate exchanging interconnecting variables that represent coupling relationships between distinct control sub-problems. The modeling is presented in the context of a distributed MPC strategy, formalizing controller interactions through definitions of Game Theory and providing the structure of a generic interaction protocol between control parts.

Performance and Resource control example: we concretize the distributed control model to the case in which components of an application graph need to optimize their effective performance level and resource utilization cost. We completely formalize this problem using our methodology and we provide results about the existence and the identification of equilibrium strategies among controllers, formalized as Nash equilibrium points. Protocols for reaching such equilibria are discussed with different assumptions in terms of controller interactions and number of iterations. Limitations of Nash equilibria, and thus of a non-cooperative approach, are addressed recalling concepts of Pareto optimality and Price of Stability. For this reason we study an alternative approach that implements a better cooperation between controllers. Due to the mathematical properties of this problem, we adopt an existing optimization technique based on the distributed subgradient method.

Experimental validation: we provide an experimental validation of the distributed adaptation strategies described in the thesis. Examples of distributed control are defined for controlling the parallelism degree of multiple ParMods of the same application graph. We study two alternative situations: the first one in which we optimize the performance behavior of the application with the strictly necessary number of used resources, and a second example in which different ParMods make distinct trade-offs between performance and resource utilization. We compare the global execution cost with different adaptation strategies, proving the feasibility and the effectiveness of these techniques.

1.3 List of Contributions of the Thesis

In this section we clearly summarize the research contributions of the thesis. The fundamental contributions are the following:

- we present one of the first attempt to integrate control theory techniques, as the model-based predictive control strategy, with the problem of controlling an adaptive parallel module, exploiting the receding-horizon optimization scheme with the performance models of structured parallel computations and their composition in application graphs;

- the thesis presents a methodology for controlling computation graphs which is a merger of different theories: Control theory for the distributed model-based predictive control technique, Hybrid Systems for the formalization of the Operating Part of a ParMod, and Game Theory for formalizing interactions between controllers that need to find an agreement in their control decisions;

- the thesis presents a relevant example in which controllers optimize their performance and resource utilization cost. Results and properties for equilibrium existence and interaction protocols are presented for this example. Moreover, in order to achieve a better cooperation level between controllers, we adapt an existing distributed optimization technique (the distributed subgradient method) to the problem of cooperatively controlling the performance and the resource utilization cost of ParMod graphs.

Other correlated results are highlighted in the following points:

- the thesis presents a performance modeling of parallel computations through notions of Queueing Networks Theory and provides a novel algorithmic procedure for evaluating the steady-state behavior of acyclic computation graphs with single-source modules;

- the thesis reviews different existing distributed control models from the control theory literature. Several structures for multiple-controllers schemes are presented in a clear way discussing their advantages and disadvantages.

1.4 Outline of the Thesis

The outline of the thesis is the following:

- Chapter 2 reviews existing research work concerning programming environments and models for developing adaptive distributed parallel applications. Firstly we analyze existing work in the area of mobile and pervasive computing. Although these research fields do not consider the execution of compute-intensive processing, they are oriented towards the satisfaction of user's desires through advanced techniques in monitored data interpretation for designing proactive adaptation strategies. Then we also present existing work in the area of parallel computing, especially programming models for unstructured and structured adaptive parallel computations;

- Chapter 3 states the assumptions and the requirements addressed in this thesis for the definition of a novel approach. We present the structured parallel programming paradigm and different classes of reconfigurations for modifying implementation aspects of the computations (e.g. parallelism degree), and the parallelization schemes. Then, we compare different adaptation strategies based on policy rules and we introduce existing work in applying control-theoretic techniques for controlling computing systems;

- Chapter 4 provides a performance modeling of computation graphs based on Queueing Networks. We initially analyze the case of acyclic graph structures, providing an algorithm for evaluating the steady-state behavior for a very significant and large class of graphs: acyclic computations with a single source module. For cyclic graphs modeling client-server computations, we describe an approximated performance evaluation approach based on Queueing Theory. With respect to the main thesis contribution (a control-theoretic approach to adaptive parallel computations), this chapter represents a stand alone contribution in the area of performance modeling of graph-structured parallel computations. Though it is necessary for the exploitation of adaptation strategies, the details of the graph analysis techniques could be skipped during a first reading;

- Chapter 5 introduces the control-theoretic strategies developed for controlling an adaptive parallel module (ParMod). We describe the ParMod organization in terms of an operating part and a control part, we provide a description of the operating part as a Hybrid System and we develop two strategies based on reactive and predictive control techniques;

- Chapter 6 presents a set of experiments of a single ParMod. Different adaptation strategies are compared and their results analyzed in terms of application optimality and reconfiguration stability;

- In Chapter 7 we introduce distributed control schemes for controlling large-scale systems. According to the existing literature, distributed control can be expressed through decentralized, distributed or hierarchical schemes. We formalize the distributed control of ParMod graphs extending the predictive control strategy with a single-layer organization of controllers;

- Chapter 8 presents a concretization of the model presented in the previous chapter for controlling the performance and the resource utilization of ParMod graphs. We provide interesting results concerning equilibrium existence and the exploitation of interaction protocols based on Non-cooperative and Cooperative games;

- Chapter 9 describes two experiments of distributed control of computation graphs performed through a simulation environment developed by extending the OMNeT++ simulator. Experimental results compare and evaluate different interaction protocols and adaptation strategies in terms of optimality of the adaptation process.

1.5 Current publications by the author

The following represents the publications that I worked on during my Ph.D research:

Conference and Workshop Papers:

- G. Mencagli and M. Vanneschi. QoS-Control of Structured Parallel Computations: a Predictive Control Approach. Proceedings of the 3rd IEEE International Conference on Cloud Computing Technology and Science, 296-303, Athens, Greece, 2011. isbn: 978-1-4673-0090-2, doi: 10.1109/CloudCom.2011.47. (I attended the conference - Slides).

- C. Bertolli, G. Mencagli and M. Vanneschi. Consistent Rollback Protocols for Autonomic ASSISTANT Applications. CoreGRID/ERCIM Workshop on Grid, Clouds and P2P Computing, CGWS 2011, in conjunction with EUROPAR 2011, Bordeaux, France, 2011.

- C. Bertolli, G. Mencagli and M. Vanneschi. Consistent Reconfiguration Protocols for Adaptive High-Performance Applications. Proceedings of the 7th International Wireless Communications and Mobile Computing Conference, 4-8, Istanbul, Turkey, 2011. doi: 10.1109/IWCMC.2011.5982862. (I attended the conference - Slides).

- C. Bertolli, G. Mencagli and M. Vanneschi. A Cost Model for Autonomic Reconfigurations in High-Performance Pervasive Applications. Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems, 20-29, Copenhagen, Denmark, 2010. isbn: 978-1-4503-0213-5, doi: http://doi.acm.org /10.1145/1858367.1858370. (I attended the workshop - Slides).

- C. Bertolli, D. Buono, G. Mencagli, M. Mordacchini, F. M. Nardini, M. Torquati and M. Vanneschi. Resource Discovery Support for Time-Critical Adaptive Applications. Proceedings of the 6th International Wireless Communications and Mobile Computing Conference, 504-508, Caen, France, 2010. isbn: 978-1-4503-0062-9, doi: http://doi.acm.org/10.1145/1815396.1815513. (I attended the conference - Slides).

- C. Bertolli, G. Mencagli and M. Vanneschi. Analyzing Memory Requirements for Pervasive Grid Applications. Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, 297-301, Pisa, Italy, 2010. issn: 1066-6192, doi: 10.1109/PDP.2010.71. (I attended the conference - Slides).

- C. Bertolli, G. Mencagli and M. Vanneschi. Adaptivity in Risk and Emergency Management Applications on Pervasive Grids. Proceedings of the 10th International Symposium on Pervasive Systems, Algorithms and Networks, 550-555, Kaoshiung, Taiwan, 2009. isbn: 978-0-7695-3908-9/09, doi: 10.1109/I-SPAN.2009.92.

- C. Bertolli, D. Buono, S. Lametti, G. Mencagli, M. Meneghin, A. Pascucci and M. Vanneschi. A Programming Model for High-Performance Adaptive Applications on Pervasive Mobile Grids. Proceedings of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems, Cambridge, USA, 2009. isbn: 978-0-88986-838-0 (Print) 978-0-88986-811-3 (CD).

- C. Bertolli, D. Buono, G. Mencagli and M. Vanneschi. Expressing Adaptivity and Context Awareness in the ASSISTANT Programming Model. Proceedings of the third International ICST Conference on Autonomic Computing and Communication Systems, 32-47, Limassol, Cyprus, 2009. isbn: 978-3-642-11481-6 (Print) 978-3-642-11482-3 (CD), doi: 10.1007/978-3-642-11482-3_3.

Journal Papers:

- C. Bertolli, R. Fantacci, G. Mencagli, D. Tarchi and M. Vanneschi. Next Generation Grids and Wireless Communication Networks: Towards a Novel Integrated Approach. Wireless Communications and Mobile Computing, 445-467, 9(4), 2008, John Wiley & Sons. issn: 1530-8669, doi: http://dx.doi.org/10.1002/wcm.v9:4.

Book Chapters:

- G. Mencagli and M. Vanneschi. Developing Real-time Emergency Management Applications: Methodology for a novel Programming Model Approach. Emergency Management, 2012, InTech. isbn: 979-953-307-417-6.

- C. Bertolli, G. Mencagli and M. Vanneschi. High-Performance Pervasive Computing. Pervasive Computing, series of Computer Science, Technology and Applications, 2011, Nova Science Publisher. isbn: 978-1-61122-615-7.

- C. Bertolli, D. Buono, G. Mencagli and M. Vanneschi. An approach to Mobile Grid platforms for the development and support of Complex Ubiquitous Applications. Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts, 2011, IGI Global. isbn: 978-1-60960-042-6, doi: 10.4018/978-1-60960-042-6.

Chapter 2

Literature Review

In this chapter we describe research work in the context of programming models and frameworks for adaptive distributed parallel applications. The first part focuses on frameworks for distributed systems in pervasive and mobile environments. Although these contexts do not usually address compute-intensive applications, they are extremely interesting because adaptivity and application reconfigurations play a key role to provide seamless services to the users. The second part focuses on high-level programming models and tools for developing adaptive parallel computations on high-performance platforms. This work is characterized by run-time supports that provide dynamic reconfiguration processes of the computation structure and dynamic changes of its mapping onto the underlying computing resources. This chapter clearly describes the interesting features and the limitations of the existing approaches, which are a starting point for identifying the requirements of a novel methodology that will be described in the next chapter.

2.1 Expressing Adaptivity in Mobile and Pervasive environments

Over the last decade, much research work has been conducted concerning novel programming techniques, models and in general frameworks for defining adaptive applications in contexts of high-mobility and pervasiveness of computing devices. In this section we start the description by discussing four different research experiences: Odyssey, Aura, Cortex and Cobra. Of course this description is not a comprehensive review of the scientific literature but these existing approaches are useful in order to understand the developing issues of distributed applications for mobile and pervasive environments.

Odyssey [33] is a research framework for the definition of *mobile* applications capable to adapt their behavior, and especially their resource utilization, according to the actual state of the underlying execution environment. The framework features run-time reconfigurations which are noticed by the final users as a change in the application execution quality. In Odyssey this QoS concept is called fidelity: a fidelity decrease leads to a lower

utilization of system resources (e.g. memory usage and battery consumption). The framework periodically controls these system resources, and interacts with applications raising or lowering the corresponding fidelity levels. In the case of Odyssey, all these reconfigurations are automatically activated by the run-time system without any user intervention. For instance, in a media player application the fidelity can be the available compression level of the played audio file, which can be dynamically selected according to the actual network bandwidth.

In Odyssey applications are composed of two distinct parts: the first one produces input data according to a certain fidelity level, and the second one executes the visualization activities on the previous data. The first part of each application is managed by a set of specific components called Warden. Each warden produces data with the predefined fidelity level, and they are coordinated by a unique entity called Viceroy. The viceroy is responsible for centralized resource management, for monitoring the resource utilization level and it handles in-coming application requests routing them to the proper warden. If the actual resource level is outside a defined range (i.e. window of tolerance), applications are notified via up-calls. Applications respond to these notifications by changing their fidelity level and using different wardens. The communications and interactions between the two application parts are managed by a kernel module (Interceptor), which extends the operation system features providing resource monitoring activities.

In Odyssey adaptivity is performed by a collaborative interaction between the run-time system (i.e. operating system) and the individual applications. This approach encourages a coordinated adaptivity between different applications which is not completely subsumed by the run-time system. As a counterpart, the fidelity concept (which is a key-point of this approach) is application-dependent: it is not possible to define generic fidelity variation strategies which can be parametrically configured for every applications. Another relevant consideration is the narrow relationship between the fidelity level and the quality of visualized data: the mobile parts of Odyssey applications exploit only visualization activities. This assumption can be restrictive when we consider more complex applications involving an intensive cooperation between computation, communication and visualization. In this case adaptivity must concern not only the quality of the visualized data, but also optimized algorithms, protocols and the performance of critical computations.

In **Aura** [34] the potential *heterogeneity* of mobile platforms is the main issue that has been faced. For each resource type proper application components exist, which make it possible to fully exploit the underlying device features. As an example a word-editor for a smart-phone has probably less features than a standard one, but it is optimized for using the device touch-pad. In Aura adaptivity is expressed introducing the abstract concept of Task: a specific work that a user has submitted to the system (e.g. writing a document). A task can be completed by many applications (called services or Suppliers), and the framework dynamically utilizes the most-suitable service executing all the support activities to automatically migrate a task from an application to another. Consider the following situation: a user must prepare his presentation for a meeting and he uses the personal computer localized in his office. Then the user is late, so he must leave the office and complete his presentation by using a mobile device (e.g. his PDA device). Aura framework takes care

of all the necessary reconfiguration and adaptation processes. So, the users partial work is automatically transferred to his PDA and transformed for the target mobile application.

Aura framework is composed of a set of different layers. The task manager (called Prism) analyzes context information (e.g. user location and motion) guessing the user intentions. Context data are obtained by means of a Context Observer (i.e. a set of sensor devices and the corresponding raw data interpretation activities). Service Suppliers represent all the services that are able to execute a specific submitted task. They are implemented by wrapping existing applications providing the predefined Aura interfaces that make it possible to extract all the useful data from the actual utilized service, and use them as partially computed tasks which can be completed by a different supplier.

In this framework application adaptivity is expressed by selecting the most proper service, according to environmental data (e.g. the user location) obtained from sensor devices. It is an example of adaptivity mainly expressed at the run-time system level: each service supplier is a standard application not aware of any adaptation process. The run-time support decides the service selection strategies by using interpreted context data, but this is not directly part of the application semantics. In particular Aura essentially considers very simple applications (e.g. write a presentation). On the other hand, if we consider more complex elaborations as compute-intensive processing, transferring a partial computed task to a different supplier can be a critical activity. In this case the knowledge of the structure of the computation can be of great importance for providing specific transformations to efficiently complete the task migration (e.g. suspending the executed task, changing the sequential algorithm, the parallelism pattern or the execution platform, and restoring the task without loosing the partially completed work).

Cortex [35] is a programming model for adaptive context-aware applications, focusing on *time-critical* distributed scenarios (e.g. automatic car control systems and air traffic control avoiding collisions). For these applications it is very critical to properly manage the system response time without any centralization point in the underlying system architecture and adapting the application components to lead the system into a safe state, even in case of unexpected environmental changes. As an example, an air traffic system controls thousands of airplanes during their taking-off and landing phases, preserving the safe distances and avoiding traffic congestion.

In Cortex an application is composed of a set of Sentient Objects. Each object is a small context-aware system which can cooperate with the other objects by means of asynchronous events. A sentient object has a set of sensors to obtain context data and a set of actuators (i.e. physical devices capable of a real-world actuation). Sensor data can pre-processed through data-fusion techniques and interpreted by using a specific hierarchical Context Model. The most important part is the Inference Engine: interpreted context data are utilized to infer new facts and situations by using a set of rules which the programmer can express in CLISP (C Language Integrated Production System).

Cortex is a very interesting approach to context-aware adaptive systems, especially in the case of developing applications capable of perceiving the state of the surrounding environment, operating independently of human control, and being proactive (i.e. being anticipatory and taking own decisions without the user intervention). This research work

presents many positive features, though it is mainly an ad-hoc solution for mobile control systems. Programming the inference engine by means of CLISP rules and using the corresponding context model can be a hard task, critical for the adaptive behavior of applications. It requires very skilled programmers and the management code could be very difficult to be reused for other applications.

Cobra [36] (Context Broker Architecture) is a research framework for the development of smart-space pervasive applications. Smart-spaces are specific instances of pervasive environments characterized by a community of software agents that can coordinate and cooperate with each other to provide services to human users. *Pro-activity* and *context awareness* are necessary properties of such kind of applications: a smart system must be able to identify the user situational conditions in order to minimize their intervention and to anticipate their future intentions. An interesting use case is the EasyMeeting smart-space, an intelligent computing infrastructure for assisting speakers and audiences during presentations. A meeting room is equipped with a large set of pervasive devices: for speech recognition, for displaying the presentations and for controlling the lighting conditions in the room. The system provides a context shared model for all the software agents and maintains consistent information about the scheduled presentations and the location of the participants. If the speaker enters the meeting room and its presentation is scheduled for that time, the system automatically transfers the presentation file from the user's PDA to the projector and starts the presentation program. The system is also able to identify the missing participants which can be automatically notified via a phone message or an email.

Cobra is a broker-centric architecture characterized by the presence of a centralized component for supporting context-aware applications. This component is the Context Broker, which maintains a shared model of context on behalf of a community of software agents and computing devices. The broker is composed of four main functional parts: the Knowledge Base is a persistent storage space of context information, the Reasoning Engine is responsible for reasoning over the stored context data, possibly extracting an implicit and useful knowledge. Acquisition Module performs all the acquisition activities by using proper context providers (e.g. sensors, monitoring and localization services). Finally the Policy Management Module exploits a set of policy rules for controlling the right device permissions to share context information with other resources in the smart-space. A key-issue in Cobra is how to manage context information in such a way as to define a formal reasoning process. Cobra proposes a solution based on an ontology model expressed by using the OWL language [37]: i.e. a set of classes with their corresponding properties and organized in a hierarchical manner.

Cobra is a very interesting approach to context awareness for pervasive systems. The context knowledge allows the system to perform proper actions that are anticipatory of the future user intentions. This approach tries to overcame the classic reactive behavior of adaptive systems, in which the main aim is to identify predefined situations (e.g. violations of some QoS constraints) which can lead to execute specific application reconfigurations or reactions. Cobra supports proactive applications, in which the user intentions are the main requirements that the system should be able to anticipate. An important limita-

tion of Cobra framework is the centralized view of the context model. By using a unique Context Broker we can simplify the information acquisition and management processes. Unfortunately this approach is characterized by a limited scalability of the overall system architecture, especially for very large distributed systems.

In this section we have introduced some interesting research studies concerning adaptivity in mobile and pervasive environments. In these approaches adaptivity is exploited by performing at run-time proper application reconfigurations driven by the users. In mobile environments like Odyssey the user expectation of the application execution, that is the so-called fidelity level, is automatically adjusted in such a way as to properly react to platform changes (e.g. different network behaviors or resource availability) maintaining the desired quality of service and the continuity of the service. Reconfigurations consist in the selection of different behaviors for specific application components (e.g. performed algorithms, data layout and optimizations) that can be considered implementation changes of the involved components. On the other hand in programming frameworks for pervasive environments, the main objective of the adaptive behavior is to be pro-active and anticipate the user intentions and preferences. This pro-activity is triggered by a pre-processing and an interpretation of environmental information (also known as *Context Awareness* [38, 39]), and run-time reconfigurations are mainly geometrical changes which consist of the selection and the deployment of application components on specifically selected computing resources.

2.2 Programming models for Adaptive High-Performance applications

Parallel applications are particular classes of tightly coupled distributed computations composed of several cooperating processes. Since the last years we have seen a significant diffusion of parallel programming in different research areas as scientific computation (e.g. numerical algorithms, computational chemistry and earth observation), data-mining (e.g. searching and knowledge discovery), image processing and multimedia elaborations. According to the current trend in computer technology more and more platforms are currently equipped with very powerful parallel computing facilities, such as multi-/many-core components and GPUs, provided also for mobile devices rendering the embedding of compute-intensive functions quite feasible at low power consumption. In this complex scenario adaptivity for parallel applications is an unavoidable requirement, especially for real-time systems in which the presence of strong real-time deadlines and performance constraints requires the capability of automatically adapting the application behavior for achieving the required QoS and quickly responding to platform changes. Emergency management systems (e.g. for disaster prediction and management, risk mitigation of floods and earthquakes) are notable situations in which real-time deadlines and an adaptive behavior of the system are strictly required features. In this section we will introduce some research work concerning programming environments for adaptive

high-performance computations for several classes of computing architectures, i.e. from traditional HPC platforms to heterogeneous mobile infrastructures.

The most widespread programming model for parallel applications, which is a *de-facto* standard, is based on the **MPI** [40] (Message Passing Interface) communication library. In this approach parallel programs are expressed in an unstructured way by describing the behavior of a set of distributed processes cooperating by using communication channels. Last implementations of the MPI library provide some form of support to dynamic reconfigurations of parallel computations. MPI2 [41] provides a mechanism for instantiating new processes at run-time: this feature can be exploited to perform structural changes of a parallel program, for instance it is possible to increase the parallelism degree (e.g. the number of executed processes) achieving in this way a potentially better performance level. In this approach the management of dynamicity and adaptivity aspects is completely left to the programmer, which must be aware of the adaptive behavior and it is heavily involved in ensuring and maintaining the consistency and the correctness of the computation during the reconfiguration phase. Implementing complex adaptation logics in this way usually requires a huge effort, dealing with many low-level details which lead to a complexity explosion for programming large parallel adaptive applications.

In order to partially solve the complexity issues of unstructured parallel programming approaches, some notable research work has been proposed, as the **ASSIST** [18] programming environment for distributed parallel elaborations. ASSIST is a parallel programming framework for several classes of computing architectures, from shared-memory platforms as SMP and NUMA multi-processors to general distributed-memory multicomputers as cluster of workstations and large-scale Grids [42]. The most important novelty of this approach is the structured methodology for expressing parallel computations, which are instances of well-known parallelism schemes (e.g. task-parallel programs as task-farm or pipeline and data-parallel programs as map, reduce or communication stencils). This approach is known to the scientific community as *Structured Parallel Programming* [17].

In ASSIST a run-time support to dynamic reconfigurations [13, 43] is rendered by exploiting a transparent approach to adaptivity. A dynamic reconfiguration is a run-time change that involves a specific application component by modifying: (i) the mapping between execution processes and the underlying computing resources (i.e. geometrical changes); (ii) by increasing or decreasing the number of execution processes of a component (i.e. structural changes). As said before, transparent approach means that reconfiguration activities are not directly defined by the application programmer, but their implementation can be automatically derived by exploiting the well-known behavior of structured parallel computations. Some papers [44, 45] describe how adaptivity is exploited in the ASSIST framework. Reconfigurations are *performance-oriented* [23]: dynamic changes of application components are triggered by the run-time support in presence of QoS violations of a pre-defined performance contract. A typical scenario is: *"the mean throughput of a parallel component is lower than a minimum acceptable threshold that has been granted to the final user"*. In response to this QoS violation the run-time support may decide to execute a parallelism degree reconfiguration.

The ASSIST framework is a interesting research work and its experience can be use-

ful for identifying the requirements of a more general approach to adaptivity for parallel programs, in which some existing limitations can be overcome: (i) ASSIST does not face with the programming issues of highly heterogeneous architectures as mobile platforms, which suggest the introduction of proper classes of application reconfigurations and also the run-time control of several non-functional aspects of a computation besides performance metrics (e.g. as in the case of energy- and memory-aware elaborations); (ii) though the research activity in the ASSIST framework is specifically focused on reconfiguration optimizations for several parallelism schemes, in ASSIST there is not a strong and well-defined methodology for controlling the performance and in general the behavior of parallel programs ensuring properties like stability of an application configuration and for providing some assurances that the desired QoS goals will be attained during the execution.

Within this research line an interesting work is the **Behavioural Skeleton** [24] approach. Adaptivity for distributed high-performance computations is exploited by means of the Grid Component Model [46, 47] (GCM) and the structured parallel programming paradigm (which the authors also call skeleton-based programming). In GCM an adaptive component is composed of two main parts: the Membrane which is an abstract unit responsible for controlling the adaptive behavior of the component, and the Content composed of a set of processes which perform the corresponding functional logic of the computation. These entities can also be other GCM components (i.e. inner components): therefore the GCM model makes it possible a natural hierarchical nesting between several adaptive parallel components.

In more detail a Behavioural Skeleton is a component which exploits a structured parallel computation according to a well-known parallelism scheme. The adaptive behavior of the component is performed by means of two membrane elements. The first one is the Autonomic Manager, which is responsible for receiving the desired QoS level from users or from other application components (e.g. a minimum average throughput), and it can also trigger QoS violation events to other managers in the hierarchy. The second control entity is the Autonomic Behavioural Controller, which is responsible for monitoring specific execution parameters (e.g. the mean service time) and for executing component reconfigurations (e.g. structural changes as parallelism degree modifications). The adaptation policy, that is the strategy which is followed by an adaptive parallel component, is expressed inside the Autonomic Manager by using JBOSS [48] rules having the usual Event-Condition-Action shape: if a specific event is identified (i.e. a QoS violation notified by the controller), the manager triggers corresponding reconfigurations by using the non-functional interfaces of the component's controller.

This approach is also characterized by very interesting run-time mechanisms for controlling multiple non-functional concerns of a parallel computation (e.g. it is possible to simultaneously control different parameters as performance and security objectives). In this case the solution proposed in [49] provides multiple autonomic managers for a single component, each one controlling a specific non-functional concern by using a set of policy rules. Different policies can lead to conflicting decisions: in this case the authors propose a distributed consensus-based solution. If a reconfiguration action decided by a manager

is in conflict with the policy of other managers, the first one should possibly find another equivalent action which takes into account the other controlled non-functional concerns of the computation.

In the previously presented work adaptivity for parallel programs is usually considered in a traditional scenario in which compute-intensive elaborations are performed on classic high-performance architectures as server clusters and mainframes interconnected by fixed and high-speed networks (e.g. InfiniBand or Mirynet technologies). The diffusion of on-chip energy-aware parallel architectures equipped also on mobile platforms requires to study adaptivity also in the context of mobile environments featuring beowulf network facilities based on commodity wireless technologies (i.e. Bluetooth, IEEE 802.11n/g). There are few research studies that have faced with the programming issues of parallel elaborations for these next-generation mobile platforms. A first attempt to extend pervasive programming models considering also parallel computations is the **MB++** [50] framework.

MB++ is a framework for developing distributed applications for mobile pervasive environments. Such applications are pervasive (i.e. designed for small mobile devices) and require also the execution of high-performance computations performed by HPC resources (e.g. a cluster architecture). Typical examples are transformations on data streams (e.g. data-fusion, format conversion, feature extraction and classifications). These applications are described as data-flow graphs whose nodes are transformations on data streams and the results are visualized by mobile nodes. An example of MB++ application is a metropolitan-area emergency response infrastructure. A large set of input data are obtained from pervasive and sensor devices: e.g. traffic cameras, mobile devices of local police and alarms located in controlled buildings. These data are not only available for monitoring activities, but they are also useful for executing complex real-time analysis (e.g. forecasting models and decision support systems) by using HPC centralized resources.

MB++ system architecture is composed of some clients, which are mobile devices producing or consuming information, and a set of HPC resources which execute the main system components: the Type Server, the Stream Server and a set of Transformation Engines. Type server dynamically manages data type definitions for each stream and all the transformation requests received from the clients. Stream server is responsible for executing data-flow graphs submitted by clients. A Scheduler, inside the stream server, enqueues the received graphs in specific command queues for each transformation engine. A transformation engine is executed on each HPC resource available in the system. The stream server allocates data-flow graphs onto a set of transformation engines, whereas the corresponding source code are provided by the type server component.

As said above, MB++ is one of the first research work focusing on high-performance computations in mobile and pervasive scenarios. The data-flow graph assignment is performed statically by the stream server when the graphs are allocated for the first time. Therefore, in specific situations we are not able to assure a load-balanced execution by means of run-time reconfigurations and reorganizations of the data-flow graphs. In MB++ adaptivity and context awareness are expressed in a limited fashion and there are no in-

teractions between mobile devices (except those with the stream server). In particular, mobile devices execute only pre-processing or post-processing activities, whereas data-flow graphs can be executed on HPC resources only. In many critical applications, such as Emergency Management Systems, these shortcomings are a crucial point since it may be useful to dynamically execute real-time intensive computations also on a distributed set of localized mobile resources featuring a sufficient computational power.

2.3 Summary

In this section we have presented a limited review of the actual state of the art concerning adaptive distributed systems. There is not a unified approach for programming time-critical high-performance elaborations especially for highly heterogeneous infrastructures. Some research work focuses on HPC computations in real-time environments, but in these approaches the pervasive and mobile part of application definition is essentially missing. It means that these programming models or frameworks are not properly defined for considering the execution of parallel computations on mobile architectures, in which concerns like memory and energy utilization are of special interest in addition to classic performance-related requirements. On the other hand other research work achieves the necessary expressiveness for defining pervasive and adaptive applications, but it does not face on intensive real-time computations neither performed by HPC centralized resources nor by systems of mobile devices. In the next chapter we will establish the requirements and the main features of a novel approach for adaptive parallel computations.

Chapter 3

Requirements of a Novel Approach to Adaptive Parallel Applications

The development of adaptive applications for highly distributed computing platforms requires a novel approach which has not been completely faced in the research work introduced in the previous chapter. This approach must be characterized by a strong synergy between two different research fields: Pervasive and Mobile Computing [51, 52] and Grid Computing [2, 3]. Both of them consist of a set of methodologies to define applications and systems for heterogeneous distributed execution environments, but this common objective is faced by adopting very different points of view. Pervasive and Mobile Computing is centered upon the creation of systems characterized by a multitude of different computing and communication resources, whose integration aims at offering seamless services to the users according to their current and time-varying needs and intentions. In this scenario the main issue is to provide a complete and automatic integration between the final users and an evolving execution platform. On the other hand Grid Computing focuses on the efficient execution of compute-intensive processes by using geographically distributed sets of computing resources. In this field, methodologies to deal with the heterogeneity and the dynamicity of network facilities and computing nodes (e.g. scheduling, load balancing, data management techniques) are more oriented towards the achievement of given levels of performance, efficiency and security.

To merge these two research areas, since the last years next-generation Grid [5] platforms and novel distributed programming environments have been introduced, but the research effort is still at the beginning: adaptation techniques and models still require intensive theoretic and experimental research. This integration must provide a proper combination of high-performance programming models and mobile computing frameworks in such a way as to express a QoS-driven adaptive behavior. The main goal of this thesis is to present a novel structured approach for adaptive distributed parallel computations. In order to introduce and formalize this approach, we will describe the guidelines that have been followed for expressing the proposed solutions. The most crucial issues can be summarized as follows:

High-Performance computations: first of all we require a structured methodology for expressing parallel programs in such a way as to make the programming effort less costly and less time-consuming. This means that we need sufficient high-level abstractions featuring a high-degree of programmability, compositionality and performance portability. A relevant starting point is the Structured Parallel Programming methodology initially introduced in [17]. A significant property of this approach is the existence of proper ***Performance Models*** able to predict and quantify the behavior (in general the Quality of Service) of a parallel computation in several execution and environmental conditions. These models, for instance based on Queueing Networks Theory [53], are analytical formulations of the expected performance (e.g. the completion time or the throughput) in function of different model inputs: e.g. the calculation time of specific sequential computations and the time spent in communication between different processes. Performance models have been used in existing structured parallel programming frameworks [18] to statically decide (i.e. at compile or at loading time) the optimal parallelism degree of a parallel computation, in order to optimize application metrics as the throughput or the application completion time.

Dynamic application reconfigurations: an approach to the run-time adaptation of distributed systems requires the introduction of a set of dynamic reconfiguration activities able to modify the current system operation. By exploiting performance models of structured parallel programs we can evaluate the key-parameters that mainly influence the behavior of our applications. Some of them may not be directly controllable by the system itself, but instead they may be considered as environmental information or disturbances that affect the application behavior. As an example if we consider a client-server application, the average number or user requests received in a time period contributes to the congestion degree of the server, which certainly influences its performance. On the other hand other model parameters can be considered as system design aspects that, in a dynamic scenario, we can modify at run-time. In the previous example controllable system features can be: the server service rate, the sequential algorithm exploited for serving each request, the parallelization used for improving the server performance and, finally but not less important, the actual deployment decisions, i.e. the corresponding computing resources on which the application or some of its sub-parts are currently executed. Changing one of these control parameters induces the execution of dynamic reconfiguration activities. In this chapter we will introduce some specific classes of reconfigurations for structured parallel computations.

Adaptation strategies and properties of the control solutions: reconfigurations are the basic actions for restructuring a distributed parallel application. Although they are mainly activities which involve a single specific application component, we may be required to modify the entire application graph by means of a sequence of these reconfiguration actions, in such a way as to perform a global adaptation

of multiple distributed components. In every case the decision to execute a set of reconfigurations, changing the current application configuration, is taken by the control logic exploiting a proper *adaptation strategy*, which can be implemented according to different schemes: centralized approaches in which a unique *controller* (called manager in other approaches as in [13, 43]) is responsible for the entire application management, or more realistically by using proper organization of controllers (e.g. hierarchical controllers as in [24]).

Currently it is essentially missing a methodological approach to adaptation strategies for distributed parallel applications which ensures execution properties such as the *stability* of a system configuration and the *optimality* of the adaptation process. For these reasons the main objective of this thesis is to define a control-theoretic model for controlling structured parallel computations.

In the rest of this chapter we will describe in greater detail these three main requirements. In the next section we will introduce an existing methodology for expressing structured parallel programs featuring proper performance models, which is a starting point for the approach studied in this thesis. Next, we will present a novel classification of dynamic reconfigurations, and finally, in the last section of this chapter, we will describe the actual state of the art concerning adaptation strategies for distributed parallel applications.

3.1 A Structured methodology for expressing Parallel programs

Over the last decade a significant research effort has been invested in studying and developing new programming models and frameworks for parallel computations. A big challenge has been the definition of approaches which render parallel programming *ease to use*, improving the *reuse* of existing components to create different and more complex systems and providing *performance portability* without requiring intensive interventions of programmer to tune the performance of each application. Portability for parallel programs exhibits a twofold nature: portable parallel applications should be able to be used on different computing platforms without modifying the program source code, and the porting phase should also be able to exploit in the best way as possible the physical aspects of the underlying architecture (e.g. by tuning proper application parameters during the compiling phase, as the parallelism degree, task granularity and process/data mapping on corresponding processing elements).

As it is well established by the scientific literature [54, 55, 56], a high-level approach is the only solution to performance portability of parallel applications: in other terms defining parallel programs having a reasonable expectation about their performance, and in general about their behavior when they are executed on heterogeneous architectures (e.g. multiprocessors, workstation clusters and multi-/many-core components). Whilst directly programming with message-passing or shared-memory libraries is the most used

approach, it does not exhibit sufficient expressive power to support high-level development of complex applications and thus performance portability.

For these reasons the structured parallel programming [17] (SPP) has been proposed as an effective and attractive approach to parallel programming featuring interesting properties like high-level programmability and performance portability. According to the SPP methodology a parallel computation is expressed by using well-known *parallelism schemes*, for which parametric implementations of communication and computation patterns are clearly identified. In fact the experience in parallel programming suggests that parallel programs make use of a limited number of parallelism patterns exhibiting regular structures, both concerning data organization and partitioning or replication of functions. In this way we can identify several parallelism paradigms as *data parallelism* schemes (e.g. map, reduce, parallel prefix and communication stencils) and *task parallelism* structures (e.g. pipeline, task-farm and data-flow). Furthermore the ***QoS predictability*** of these parallelism schemes has been studied by exploiting formal analyzes, thus rendering the performance modeling of this class of computations quite usable also by automatic tools as compilers (e.g. for statically deciding the best application configuration on a specific target execution platform) and from run-time supports (e.g. for providing efficient fault-tolerance [57] mechanisms but also dynamic reconfigurations [13]).

Therefore the SPP approach is a starting point for the methodology proposed in this thesis. In particular this methodology inherits from the past experiences in structured parallel programming of the Computer Science Department of the University of Pisa [18, 58], providing a formal modeling of adaptivity for parallel computations in dynamic and heterogeneous execution environments.

3.1.1 Structured parallelism schemes and their composition

This section provides a brief overview about the structured parallel programming approach, discussing notable parallelism schemes, their properties and utilization in different application scenarios.

The structured parallel programming methodology is based on the concept of parallelism scheme, also called *skeleton*. As stated in the previous section they are schemes of parallel computations that recur in the realization of many real-life algorithms and applications. They exhibit the following features:

- they are characterized by constrains in the parallel computation structure;
- they have a precise semantics;
- their behavior can be predicted through a proper performance model;
- they can be composed to form complex graph computations.

First of all we can characterize two broad categories of parallelization paradigms:

- **Stream-parallel paradigm**: parallelism schemes belonging to this class are able to improve the *throughput* of a computation in the case in which a large sequence (possibly of unlimited length) of input elements is defined (i.e. *stream-based computation*), e.g. a stream of images or video frames represented as matrices, each one requiring the execution of a specific computation. The existence of a large sequence of input elements is a necessary precondition in order to apply these parallelization techniques, on the contrary no performance enhancements can be obtained if we consider a single or a limited set of input elements. Parallelism schemes that follow this assumption are the task-farm and pipeline;

- **Data-parallel paradigm**: sometimes it is possible that the application operates on a single or on a limited set of data values instead of on a large sequence of input elements. This case occurs also when, though in a stream-based computation, the temporal distance between the reception of two consecutive stream elements is much greater than the processing time of each element. In these situations parallelism schemes also able to improve the *computation latency* are of great importance. Examples are data parallelism schemes as map, reduce, parallel-prefix and communication stencils.

Another important source of differences between different schemes is the way in which data and functions are replicated and/or partitioned. Task-farm scheme exploits replication only, applied to functions and to non-modifiable data; thus, only stateless computations (pure functions) are candidate for this parallelization. Pipeline exploits a partitioning of the sequential elaboration into a sequence of successive phases, each one using a set of replicated (non-modifiable) or partitioned (modifiable) data. On the opposite direction data-parallel schemes correspond to the replication of the same functionality (also a computation with state is acceptable) and to the partitioning of data, so that distinct parallel units are able to apply the same operations to different data partitions in parallel. Important performance measures are:

- **Computation latency**: the average time needed to execute the computation on just one stream element;

- **Throughput**: the average number of stream elements which can be completed in a time unit. Alternatively we can consider the inverse of the throughput, a.k.a the *service time*, which consists in the average time interval between the beginning of the executions on two consecutive stream elements.

Parallelism schemes can have different impacts on these two performance parameters. Task-farm and pipeline, though able to increase the throughput of the computation, tend also to increase the latency compared to the sequential implementation, due to communication overhead, while data-parallel and data-flow tend to decrease both the computation latency and the service time of the computation.

In the rest of this thesis we will make use of structured parallelism schemes and we will evaluate their impact on application QoS. For a clear understanding of the following

parts, in the sequel we will introduce in more detail the structure of a generic structured parallel computation (see Figure 3.1) discussing its implementation and properties. Without loss of generality we will assume that cooperation between parallel entities (processes or threads) are exploited through a message-passing model.

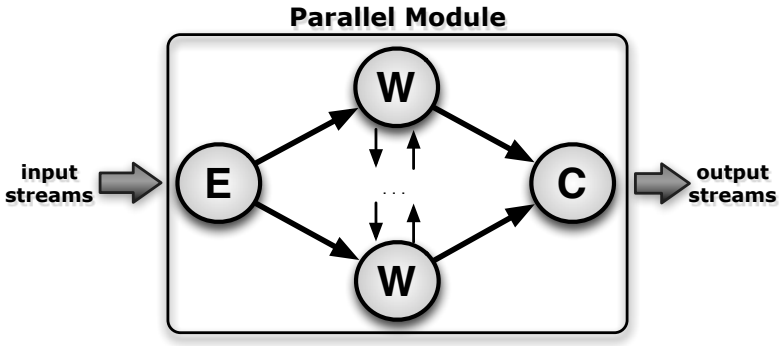


Figure 3.1: General overview of a Structured Parallel Computation.

The behavior of a structured parallel computation is described by a set of parallel units cooperating by executing communication primitives (i.e. *send* and *receive* operations) on proper channel data structures. The computation is activated by the data reception on a set of input streams. The activation can adopt different semantics:

- *Data-Flow semantics*: the beginning of the computation requires the presence of a corresponding element in all the input streams, i.e. we have to wait for the reception of an input element from all the input streams;
- *Non-Deterministic semantics*: the elaboration is activated when an input element is present in any of the possible input streams selected in a non-deterministic fashion.

We can identify three special parallel units: the *emitter*, activated in a data-flow or in a non-deterministic manner which is responsible for distributing, according to a specific strategy, the input data structures among a set of concurrent units executing the parallel computation; the *collector* that receives the computed results from workers and transmits the final results onto the output streams of the parallel module. Finally we identify a set of concurrent units called *workers*, which are in charge of performing a partitioned or replicated sequential computation on the received data.

In the following part we describe a notable set of parallelism schemes that will be broadly used in rest of this thesis:

Task-Farm scheme: task-farm is a stream-parallel scheme based on the replication of a pure function among a set of workers, without knowing the internal structure of the function itself. In this scheme emitter schedules each input stream value to a worker. The general objective is to balance the workers load, i.e. in such a way that their processing capabilities are exploited at best. This is the most fundamental

feature characterizing this parallelism approach. A possible scheduling strategy is the *round-robin* one, i.e. circular distribution. However, this strategy is not able to assure load balancing if the calculation time of a worker has a significantly high variance, e.g. if it depends on the input data values. For this case an *on-demand* approach is much more effective: its implementation is based on the availability of workers to accept a new input task. Collector is essentially an output interface which is responsible for the reception of worker results and for transmitting them onto the output streams. From the performance viewpoint a task-farm scheme has the main advantage of reducing the mean service time of the computation. Notwithstanding this scheme is completely useless if we are interested in the computation latency for a single stream element.

Pipeline scheme: pipeline scheme is another stream-parallel pattern. It is a very simple and effective solution to some parallelization problems. The application of this scheme requires the knowledge of the sequential computation, that needs to be expressed as a composition of functions. In this case a possible straightforward parallelization is a linear graph of parallel units, also called pipeline stages, each one providing the execution of a specific function. This parallelism scheme can be adopted in order to increase the throughput. Unfortunately this solution has two important drawbacks: (i) it may be hard to define a well-balanced pipeline structure, but usually the computation of certain stages may be more intensive that the other ones; (ii) the computation latency is proportional to the number of stages due to communication overhead.

Data-Parallel scheme: a data-parallel computation, on streams and/or on single data values, is characterized by partitioning (replication) of data structures and function replication. The emitter provides the distribution of each input element among the set of workers according to proper collective communications (e.g. a *scatter* or a *multicast*). Collection of the worker results is achieved by the collector exploiting a *gather* operation (i.e. collector receives worker results and builds a unique data structure, e.g. a vector or a matrix of elements). In a data-parallel scheme each worker applies a sequential elaboration on its own data. In order to apply this function, a worker may require to access data contained in other worker partitions, according to the particular data dependencies imposed by the computation semantics. In this case we speak about *stencil-based computations*, where a stencil is a data dependence pattern implemented by information exchanges between different workers. A very special, but sometimes possible, data-parallel scheme is the so-called *map*, in which workers are fully independent i.e. each of them operates on its own local data only, without any communication during the execution. As said above data-parallel paradigm is able to reduce the computation latency for a single input element but, in the case of a large sequence of input tasks, it can also improve the throughput of the computation by reducing the mean service time.

As far as composability is concerned, parallelism schemes can be composed in com-

plex computation graphs which describe distributed parallel applications. Each module can exploit a sequential or a parallel computation, in this last case based on a specific structured parallelism scheme. Stream-parallel and data-parallel approaches can be composable in stream-based computations and the semantics and the performance of the entire computation are obtained as proper compositions of the individual semantics and performance models of the used parallelism schemes. In conclusion in our approach a parallel computation can be expressed as a directed graph of cooperating modules (i.e. expressing an ***inter-module parallelism***) each one featuring a well-known internal parallel structure (i.e. structured parallelism schemes are used for expressing ***intra-module parallelism***).

3.2 Dynamic reconfigurations of Parallel programs

If we consider a parallel application defined as a directed graph of parallel components, an application ***configuration*** can be expressed by the definition of:

- a precise choice of the parallelism scheme adopted by each parallel component (e.g. both stream-parallel and data-parallel schemes can be used);
- the current parallelism degree of each parallel component;
- a complete mapping between application components and platforms on which they are currently executed.

Any run-time change which modifies some of these three aspects involves a dynamic reconfiguration activity. Although there are specific situations in which the ”best” configuration can be statically selected during the system design phase (e.g. optimizing the trade-off between performance and resource utilization cost), in dynamic execution conditions the best configuration: (1) may not be identifiable without run-time information; (2) it can change over the execution, since it depends on the current execution conditions. This last case represents a common situation because:

- the actual condition of the execution platform can be highly variable and the degree of availability of computing and network resources can dynamically vary in unexpected ways. Non-dedicated execution environments represent a relevant example along this line. In this case multiple applications compete in the utilization of system resources, and their degree of availability can heavily influence the expected quality of the execution of our parallel programs;
- in many real-time systems it is required to execute complex computations respecting precise QoS requirements. In mission-critical systems these constraints may change unpredictably e.g. due to dynamic user intentions;
- there are particularly irregular parallel problems for which a static application configuration is not sufficient to achieve a highly efficient execution. These problems

are characterized by a computation whose size, distribution and complexity depends on the properties of the input data that must be currently processed. For this class of parallel programs a run-time support to dynamic reconfigurations is an unavoidable feature to provide efficient implementations.

In order to effectively deal with dynamic execution conditions, we need a proper support to adaptivity for parallel programs. As we have seen adaptivity consists in a twofold concern:

- the definition and the implementation of dynamic reconfiguration activities for changing the current application configuration;
- the definition of proper adaptation strategies that drive the reconfiguration decisions.

For the first issue the exploitation of the SPP methodology renders feasible the development of programming models in which application reconfigurations are automatically provided by the run-time support system, without any programmer's intervention. In this case we speak about a *transparent approach* to application reconfigurations (as in [18, 13, 23]).

For structured parallel computations we can classify the set of adaptation processes in two categories namely non-functional and functional reconfigurations. Both of them concern a single application module (i.e. *local reconfigurations*). ***Non-functional Reconfigurations*** are adaptation processes involving the run-time modification of some implementation aspects of a parallel module. For implementation aspects we intend:

- it is possible to modify the current parallelism degree exploited by the parallel module, e.g. increasing the number of parallel workers in such a way as to expect a better performance (e.g. service time and/or computation latency per task) or, otherwise, decrease the parallelism degree for releasing under-utilized computing resources. Following the terminology introduced in the first chapter, such kind of modifications are structural changes of a parallel module;
- the run-time support can modify the mapping (i.e. geometrical changes) between implementation processes of a parallel module and computing resources on which they are executed;
- another important class of geometrical changes involves the run-time *data re- distribution* among the set of worker processes of a parallel module. Such changes are an effective approach for solving load-balancing issues in data-parallel parallelizations of highly irregular parallel problems.

The common aspect of the previous reconfigurations is that they do not modify the sequential algorithm performed by the parallel module, neither the exploited parallelism scheme. Hence, they are exclusively geometrical or structural changes which do not influence the internal parallelization pattern of the module.

As said, structured parallel programs are written exploiting specific parallelization paradigms characterized by well-known structures and properties in terms of performance (i.e. throughput and computation latency), data distribution and function replication. Another important feature of structured parallel programming is that the same problem can often be solved in a parallel fashion by exploiting different parallelism schemes. Consider the following example.

Example. Let us suppose that we need to parallelize a sequential computation in which we receive from other application components two input streams: the first one consists in a sequence of input square matrices A (composed of M^2 numerical elements), the second one is a sequence of vectors B of M elements. The sequential computation is activated according to a data-flow semantics: i.e. whenever a pair of an input matrix and a vector are received from the two input streams. For each pair the computation is a generalized matrix-vector product described by the following pseudo-code.

Procedure Generalized Matrix-Vector Product(A, B)

Data: an input matrix A and a vector B.
Result: a result vector C.

```
1 begin
2     for i ← 1 to M do
3         C[i] ← 0;
4         for j ← 1 to M do
5             C[i] ← F(C[i], A[i, j], B[j]);
6     return C
```

According to the way in which we perform input data distribution among parallel workers, we can parallelize the problem in several ways:

- the emitter can schedule each received pair to an available worker (i.e. on-demand strategy) which performs the generalized matrix-vector product completely on the received pair. This parallelization is based on the task-farm structure. As for any stream-parallel scheme, this solution improves the throughput of the computation in terms of number of pairs computed in a time unit that increases proportionally with the current parallelism degree;

- for each received pair the emitter can send to each worker a partition (i.e. scatter strategy) of the input matrix (e.g. a set of contiguous rows) and a copy (i.e. multicast strategy) of the input vector B. According to this distribution, each worker applies the generalized matrix-vector product independently on its partition and produces a partition of the output vector C. This parallel structure is a map scheme belonging the data-parallel paradigm. Therefore this solution is able to improve the throughput but, differently to the task-farm case, also the computation latency for completing a single pair;

- for each received pair the emitter can send to each worker a partition (e.g. a scatter of a set of contiguous rows) both of the input matrix and of the input vector. Each worker is able to apply the generalized product on its partition but, according to the existing data dependencies, it now needs to receive vector partitions from other workers. We can also note that at each iteration of the inner loop (see the algorithm pseudo-code), each worker interacts with a different worker according to a precise communication stencil. As in the map case this solution improves the throughput and the computation latency for a single input pair, but it requires a lower memory occupation since the vector B is partitioned instead of being replicated.

The previous scenario suggests that we can use this important feature of structured parallel programming in such a way as to effectively deal with highly dynamic and heterogeneous execution environments. Suppose to have a compute-intensive application which is executed on a HPC cluster architecture. Due to some events related to the state of the surrounding execution platform, we could require the migration of this computation onto a set of mobile intelligent devices (e.g. this scenario is described in [59, 60] for an emergency management system). This migration is a complex operation, concerning not only implementation issues (e.g. migrating the computation state efficiently and consistently), but also the relevant differences of new available resources and their efficient exploitation. A parallel computation for a cluster architecture could not be efficiently executed on a set of mobile nodes, due to their possible limitations such as memory and processing capacity or the performance offered by their interconnection networks. For effectively dealing with these scenarios, we can think to change the parallelism scheme exploited by a parallel module during its execution. This modification consists in an implementation change: a parallel module can modify its internal structure and the behavior of its implementation processes maintaining its input and output interfaces, in such a way that these modifications are completely transparent to other application modules. For these reasons we introduce a further class of adaptation processes, namely, ***Functional Reconfigurations***.

Functional reconfigurations are implementation changes that consist in providing a set of alternative *versions* of the same module. Alternative versions have a different but compatible semantics: they can exploit different sequential algorithms, parallelism schemes or optimizations, but preserving the module interfaces in such a way that the selection of a specific version does not modify the global application behavior. Each version of the same parallel module is characterized by a certain computing (in terms of calculation time) and communication (in terms of frequency and size of exchanged messages) footprint and memory hierarchy utilization. Therefore they are optimized for being executed whenever certain execution and platform conditions are satisfied (e.g. based on the presence of certain levels of communication bandwidth, available computational power, memory, and specific classes of computing resources).

In this section we have described two different classes of dynamic reconfigurations for structured parallel computations. Non-functional reconfigurations are mainly geometrical changes (where parallel computations are executed) or structural changes (how many

processes or threads compose a parallel computation). On the other hand functional reconfigurations intensively exploit the properties of the structured parallel programming for dealing with heterogeneous execution environments. These reconfigurations modify the way in which a computation is actually exploited, both in terms of executed algorithms and parallelism schemes.

Reconfigurations are building blocks for developing adaptive systems, but a further and interesting aspect that will be investigated in the following section concerns the adaptation logic that drives the selection of such reconfiguration activities.

3.3 Methodologies for expressing Adaptation Strategies

An adaptive application is a system which evolves over time changing its initial configuration in response to external events (e.g. environmental modifications), variable QoS requirements and for dealing with irregular computations. Providing a run-time support to dynamic reconfiguration activities is an essential requirement in these contexts. Although research studies as in [24, 13, 43, 23, 42] focus on specific implementation issues describing several optimizations for reducing reconfiguration costs in terms of performance degradations, do not pay sufficient attention to the decision process that triggers the execution of these reconfigurations. We refer to this process as an ***Adaptation Strategy***, that is a plan, a set of rules or generally speaking a method used by the system for attaining the desired execution goals.

Especially in the case of parallel computations, an adaptation strategy is aimed at meeting the required performance constraints. The system must offer its functionalities to the users according to a certain notion of *execution quality*. A general classification of this concept has been proposed in [61]:

- in many adaptive systems [24, 23] we need to control the execution progress preserving observed metrics within user-defined ranges. Classic examples are: e.g. maintaining the mean response time of the system within a *window of tolerance*, i.e. between a maximum and a minimum acceptable threshold. In this case we refer to this objective as a **threshold specification** problem;

- alternatively we can require to maintain some execution metrics as closer as possible to a set of desired reference values. For instance it could be necessary to maintain the mean service time of a computation equal to a specified target value desired by the final user. According to a control-theoretic terminology this objective is called a **set-point regulation** problem;

- more often we need to control several system measurements as the performance, energy and memory requirements, number of utilized computing and network resources. In this situation we need to find a control law such that a certain optimality criterion is achieved. This problem can be casted into a mathematical fashion introducing an objective function that should be minimized or maximized. The function

can depend on several parameters (e.g. performance, memory usage and resource utilization costs). The adaptation strategy is aimed at optimizing this multi-variable function during the system execution. This formulation is known as an **optimal control** problem.

We have introduced two fundamental issues of an adaptive system: the presence of specific adaptation mechanisms able to modify the system behavior, and a strategy to select control actions as and when necessary. A general control model that comprises these two aspects can be structured in two distinct interconnected parts:

- a target computing system (namely ***plant***) that we want to control. It takes *functional inputs* and generates *functional outputs* according to the semantics of its computation. In computing systems input-output relationships reflect how work is done and/or data are transformed. However, for evaluating and controlling the current system operation, also the input-output relationship among QoS variables is of great importance. The system behavior can be influenced by receiving specific *reconfiguration commands*, whose values influence the way in which the system execution is exploited. Moreover, the quality of the system operation is evaluated by measuring non-functional metrics also called *monitored data*, e.g. the actual level of performance parameters;

- a ***controller***, which is an independent entity able to monitor and affect the system operational conditions by taking and analyzing the monitored data and deciding a corresponding set of reconfiguration commands.

Interactions between these two parts occur in two directions: (i) system operation is observed by the controller exploiting monitoring activities; (ii) based on the evaluation of a specific strategy, a set of reconfiguration commands can be issued by the controller. Then reconfiguration commands trigger the execution of dynamic reconfiguration activities.

System execution evolves along three successive phases that are continuously exploited by the two parts of the control architecture:

Monitoring phase involves capturing current properties and measurements of the system which are effective for identifying when the execution of a dynamic reconfiguration can be useful for achieving the desired QoS. This information can be acquired from different providers: sensors can be used to obtain environmental information, profiling services can measure several execution parameters such as the performance (e.g. the actual service time of a certain application component) but also the resource utilization levels (e.g. memory usage or number of nodes).

Planning phase consists in a set of concrete actions aimed to select a new set of reconfigurations that are the best response to the current monitored data characterizing the system execution.

Reconfiguration phase applies the decided reconfigurations to the controlled system by exploiting functional or non-functional reconfiguration activities.

These three phases and their periodical execution identify a *closed-loop interaction* scheme (see Figure 3.2) between the plant and its controller, which is a general and well-known structure for adaptive systems followed by different research fields (i.e. Control Theory, Autonomic Computing and Theory of Agents).

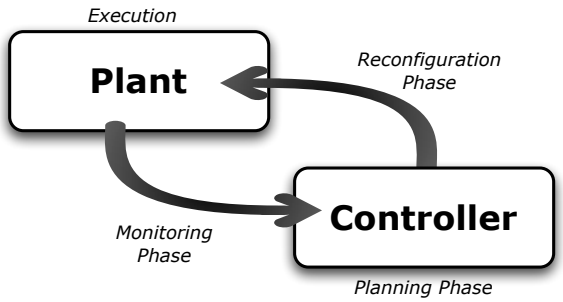


Figure 3.2: Control-loop scheme of an Adaptive system.

In adaptive computing systems the plant is a software application. More precisely in the case of a distributed parallel computation the plant can be considered a generic graph of cooperating parallel components. System adaptation strategy is performed by a set of proper software entities (i.e. controllers) aimed at the execution of the three previous phases. In many existing techniques these activities are exploited by a single software element (i.e. in a centralized fashion) or, for a better scalability, by a multitude of control units (i.e. in [24, 43] called managers) which distributedly evaluate the system adaptation strategy. As a first classification the logic behind these strategies can be distinguished based on the degree of far-seeing of control decisions:

- **Reactive logic**: a reactive system [62] is a system that changes its status in response to external stimuli. A reactive logic is able to analyze system monitored data and to identify QoS violations performing corresponding reactions in a way that enforces or enables the desired system behavior;

- **Pro-active** or **Predictive Logic**: pro-activity is a feature characterized by a higher level of complexity than reactivity. Instead of merely react to stimuli, being proactive means that system can consciously involve acting in advance of a future situation, and controlling the effects of these actions. As an example a proactive system can avoid the violation of specific QoS constraints by performing in advance proper modifications of its behavior.

Since the last years several methodologies for developing adaptive systems and their adaptation strategies have been proposed. In the following part of this section we introduce two distinct approaches to adaptation strategies: (1) a declarative approach based on policy rules; (2) a model-driven approach based on control theory foundations for designing and developing controllers.

3.3.1 Adaptation strategies expressed by policy rules

In this section we briefly introduce a very general and flexible methodology for expressing adaptation strategies for adaptive systems, which has been intensively used in pervasive and mobile contexts due to its simplicity and programmability. Reconfiguration activities can be viewed as reactions to pre-defined system situations. As an example in mobile applications, if the available energy level of a mobile device is lower than a specified threshold, the execution can switch to a lower-consumption version which preserves the battery duration exploiting a limited display utilization. Therefore a possible solution for expressing adaptation strategies consists in providing a mapping between execution events and corresponding reconfigurations (i.e. situation-action pairs) as a finite set of imperative policy rules. For these reasons this methodology is known as a ***policy-based approach*** [31] to adaptation strategy specification.

Policy rules are a form of guidance used to determine decisions and corresponding actions on the system execution. They have originally been introduced in the field of intelligent agents [63], which are abstract entities able to perceive their environmental conditions and react on the basis of this information in order to optimize their objectives. An adaptive system is in a specific internal state at every given moment of time. A set of policy rules may cause an action to be taken an therefore a transition to a different internal state of the system. Policy rules can be expressed according to several paradigms. In the scientific literature we can distinguish between three main classes of rules: Event-Condition-Action, Goal-Oriented and Utility-Function rules.

Event-Condition-Action policies (ECA) represent a well-known paradigm which has originally been defined for expert systems [64] and distributed active databases [65]. An ECA rule has the following syntax:

when event if condition then action;

Informally its abstract semantics is: the occurrence of the event triggers the rule evaluation, the condition is checked in order to ensure if the system is in a specific internal state. If this condition holds, the corresponding action is enforced. We can observe that, though the state that will be reached by applying the rule is not explicitly expressed, the policy programmer has to know the desired effect of the selected action. In this approach a set of ECA rules is responsible for monitoring the system execution and responding to environmental changes. When an event occurs, the system controller determines which rules in the policy set have to be evaluated. Two or more rules can be based on the same event-condition pair, thus they can be triggered on the same situation.

ECA rules are a straightforward way to express a reactive behavior. On the other hand some limitations characterize this methodology. First of all policies may have a set of rules that trigger each other continuously. This situation namely *Policy Cycle* [66] yields to possibly un-terminating sequences of rules which are continuously executed without reaching a convergence condition.

Another source of problems for policy-based systems are conflict issues between different rules. As stated in [67] ECA rules conflict if: (1) their actions are in contrast, and

(2), they may be triggered at the same time (i.e. the event-condition pairs overlap). As an example if a rule requires to increase the number of used computing nodes and another rule imposes the system to reduce its power consumption, these two rules are in conflict. The presence of conflicts and their efficient resolution is a fundamental issue in policy based approaches. Conflict resolution techniques increase the complexity of the reconfiguration decision process. In some situations these conflicts can be detected by a static analysis of the policy rule set. For instance the programmer can add meta-policies [31, 67] to disable some of the conflicting rules. Another method is to define the resolution strategy by applying different priority levels: among those rules whose conditions are simultaneously satisfied, only the one with the highest priority will be executed. Unfortunately there are many situations in which a static analysis can only suggest the possibility of a conflict, but its effective presence can be detected only at run-time. This is especially true when conflicts occur due to side-effects of the rule actions or in the case of long-running actions (i.e. a rule can conflict with rules that will be applied in the future).

Another alternative approach consists in using a different shape of logic expressions called **Goal-Oriented** rules. A goal rule is a specific condition which characterizes a desired state of the system. E.g. we can identify a performance requirement as:

$$Response\_Time \leq THRESHOLD$$

In this case we desire to maintain the mean response time provided by a component lower than a given threshold. We can observe that, in contrast to the ECA paradigm, the goal rules do not explicit the actions that will be necessary to meet the desired state, but the system must exploit other mechanisms in order to decide them (e.g. actions are inferred by an automatic learning engine).

One of the main drawbacks of a goal-oriented approach is the limited solutions that can be applied when some conflicts between different goals are detected [67]. If more goal rules are defined, the system will take a set of actions in order to ensure that all of them will be satisfied. If the system can satisfy only one goal rule (e.g. due to constraints or conflict goals), we do not know which among the desired states is preferred. Of course some solutions are straightforward, as applying proper priority levels between different goals, but they do not solve the problem in general.

Utility-Function policy rules generalize the approach of goal-oriented policies by introducing a utility function. This function is applied to define a desirability level for each state of the system, instead of performing a limited binary classification between desirable and undesirable states. A utility function returns an output value which is a relative measurement of the utility degree. Therefore, the adaptive behavior aims to place the system into the state with an utility level as higher as possible. Another advantage of this approach is that it is completely conflict-free. In fact the utility function makes it possible to map each system state onto a different real value, which is the only parameter used for deciding if a reconfiguration must be hold.

Utility-function policies are a very flexible approach and they are also conflict-free by definition. But, on the other hand, the main drawback of this solution is the high-degree of complexity concerning the utility function definition. Specifying a numeric value for each

internal state of the system could be extremely hard. The system administrator or the designer has to decide the most suitable shape for this function and all the input parameters need to be clearly identified.

The previous methodologies are characterized by some advantages and drawbacks. Hence, several existing approaches focus on the definition of mixed policy rules, by using a composition of the existing solutions. Some approaches can be mixed very well: as an example goal-oriented rules and utility-function rules have the common feature to be expressed in terms of the space of desired system states. Instead, in the case of ECA rules, the policies are expressed by considering the current system states and providing the corresponding actions in a straightforward fashion.

The design of adaptation strategies by using a set of policy rules is usually an ad-hoc work viable only for simple systems, in which only a limited number of event-action pairs are necessary. Moreover policy-based adaptation strategies are usually tightly coupled with the specific controlled system, and it is hard to quantify their effectiveness in terms of the control optimality for the long-term execution and the stability of the reconfiguration decisions. In existing frameworks [24], rule-based approaches have been used for controlling structured parallel computations, but the knowledge of the structure of the computation, that we consider a first-place assumption in our approach, can be used in a better way in order to define more powerful adaptation strategies. In the last section of this chapter we will present different controlling techniques that can be a starting point for a more advanced controlling of parallel and distributed applications.

3.3.2 Control-theoretic strategies for Computing Systems adaptation

Besides artificial intelligence, the concept of automated operations is an important aspect also in engineering disciplines for developing autonomous and adaptive systems able to target their functional requirements. The methodologies based on Control Theory [29] foundations have been intensively exploited for designing controllers and feedback systems in many industrial plants and mechanical infrastructures. These solutions provide powerful mechanisms for dealing with dynamic changes, uncertainties, and system disturbances.

The control-loop scheme depicted in Figure 3.2 provides a general blue print for developing self-adaptive computing systems, which is also a starting point for control-theoretic techniques as feedback configurations [29, 30]. Recently, control-based approaches have been a solid solution to solve network problems like congestion [68] and flow [69] control, rate adaptation of queueing networks [70]. This leads to an increasingly important research area in which the adaptive system is the computing network itself (i.e. Autonomic Networking [71]), whose interconnection facilities are able to automatically detect, diagnose and repair failures, as well as to adapt their underlying configuration and optimize the performance and quality of service.

Nevertheless, the exploitation of these techniques for computing systems has been rarely used actually. In this context system controllers are instead specialized for limited applicative domains (e.g. as we have seen by expressing policy rules), without us-

ing a general methodology for dominating the complexity of adaptive systems definition. Therefore in existing approaches controlled parameters are regulated in an ad-hoc fashion, exploiting the intuition that the behavior of a computing system can be modified changing some of its implementation features (e.g. buffer sizes, number of processes and data structure memory layout). The most crucial challenge in applying control-theoretic foundations in this case, is to find a rigorous modeling of controlled plants, which renders possible and realistic the formal control of the system behavior. Along this line several research work has attempted to study such modeling effort for computing systems. In [72] the problem of managing resource utilization for web servers has been studied providing a combination of queueing models and feedback schemes used to regulate the system response time. In [73] adjusting multiple QoS parameters has been introduced for the IBM Lotus Domino Server: both CPU consumption and memory usage have been simultaneously controlled by using a statistical model defined observing the system behavior in realistic and expected workload conditions. In [74] the authors propose the concept of adaptive component, composed of multiple versions each one featuring a QoS behavior modeled through empirical techniques. These models have been used in order to tune proper version switching decisions able to avoid QoS violations of pre-defined thresholds.

The systematic design of controllers requires the ability to quantify the reconfiguration effects on the system execution. Hence a controller must be equipped with a mathematical representation of the system behavior. This model should be expressed in an *input-output form*: i.e. it describes the relationship between input and output variables of the model. Note that they may have nothing to do with traditional functional inputs and the outputs of a computation. System models can be classified into two broad categories:

- **First-Principle models** [75] are based on the actual physics laws that govern the dynamics of the system being modeled;
- **Empirical or Statistical models** are based on observing system execution and inferring relationships between inputs and outputs by applying statistical techniques. Such *System Identification* [76] techniques are of great importance especially in situations in which the knowledge of the internal structure of the controlled system is not known a-priori. In this case modeling phase consists of four steps: (i) the mathematical structure of the system model is initially decided (e.g. in terms of differential or difference equations with specific orders and parameters); (ii) a set of experiments are designed to collect data representing the system execution in expected workload conditions; (iii) statistical approaches, as least-square techniques, are applied in order to estimate model parameters based on the previous experimental results; (iv) the quality of the system model is evaluated using statistical metrics (e.g. correlation coefficient and root-mean square error). Design a proper set of experiments is of particular importance for applying statistical techniques, for this reason this approach is also known as *data-driven modeling*.

First-principle models are in general unavailable for computing systems except for special cases. *We claim that a relevant example in which first-principle models can be used,*

consists in the performance modeling of structured parallel computations. For structured parallelism schemes and their composition in computation graphs, performance models offer a sufficient guideline to define powerful adaptation strategies for distributed parallel applications.

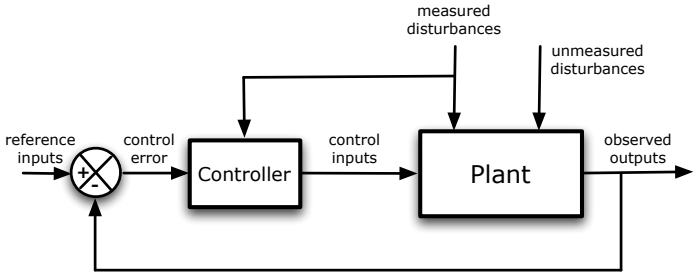


Figure 3.3: Feedback Control Scheme.

Figure 3.2 depicts a closed-loop configuration of a plant and a system controller. In control theory this configuration is also known as a *Feedback Control System* [77], which is exploited in a wide variety of situations in everyday life. Feedback configurations consist in a closed-loop between a plant and a controller as depicted in Figure 3.3. In addition to control inputs and observed outputs, a feedback scheme provides other essential elements: (i) reference inputs are used in set-point regulation problems since they represent target values of observed variables; (ii) disturbances are non-controllable parameters that modify the way in which control inputs affect the system observed outputs; (iii) control error represents the difference between target values and actually measured results. This scheme is characterized by the utilization of monitored information from the plant to decide control inputs and achieve the desired execution goals.

Other control techniques avoid using the observed outputs to adjust system control inputs. This is the situation of classical open-loop schemes as *Feedforward Control Systems* [77] depicted in Figure 3.4. In this approach control inputs only depend on current

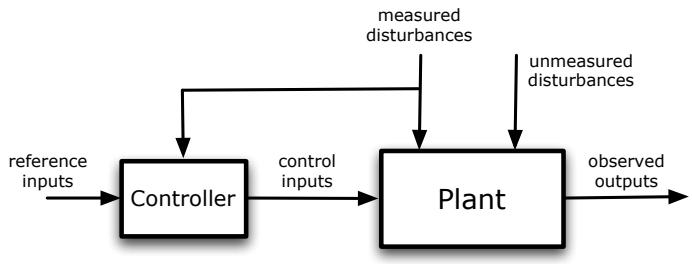


Figure 3.4: Feedforward Control Scheme.

reference inputs, and the definition of an accurate and robust system model is even more critical for the applicability of this control scheme.

Control theory techniques represent a valuable research direction for improving adaptation strategies for computing systems and for structured parallel computations. These methodologies, compared to policy-based approaches, make it possible a more rigorous analysis of the controlled system and the enhancement of properties like:

- ***Optimality*** of the adaptation strategy. How a set of control decisions makes it possible the achievement of the desired QoS objectives during the entire system execution: e.g. observed values are close to reference values (set-point regulation), or they are inside an acceptable region of values (threshold specification) or a cost functional is minimized (optimal control);
- ***Stability*** of reconfigurations. How long a system configuration represents a "good choice" for targeting the required QoS objectives.

In the following chapters we will explore existing control-theoretic techniques and we will discuss their exploitation in the context of structured parallel computations.

3.4 Summary

In this section we have provided an overview about the general requirements for a novel approach to adaptive distributed parallel applications. As we have seen the structured parallel programming methodology is a cornerstone for the control techniques described in this thesis. We consider parallel computations that are not general and ad-hoc parallelizations, but they follow specific and well-defined paradigms and schemes. As we will demonstrate, this deep knowledge of the computation structure and of the communication patterns makes it possible the definition of well-structured reconfigurations and performance models. We argue that such methodology has the potentiality to being controlled through control-theoretic techniques. Before starting with the presentation of these approaches, in the next chapter we will investigate how QoS predictability for structured parallel computations can be formulated based on performance models of structured parallelism schemes.

Chapter 4

Performance Modeling of Parallel Computations

IN order to apply a formal approach to adaptivity, the *QoS predictability* of distributed parallel computations is a fundamental feature that needs to be addressed. The QoS behavior of a parallel component (or module) needs to be evaluated by using models that describe the relationship between QoS measurements, e.g. performance parameters as the mean throughput, computation latency, completion time of a parallel computation, in function of its actual configuration as the used parallelism scheme and the parallelism degree. To this end in this chapter we will introduce analytical methods for evaluating the performance behavior of parallel applications expressed as composition of modules cooperating in generic computation graph structures.

4.1 Performance modeling of computation graph structures

From a general point of view a distributed parallel application can be represented as a directed application graph (*work-flow*) of modules cooperating by exchanging typed messages[1]. Modules can exchange single values or, as in stream-based computations, a sequence of messages by means of communication channels. As we have hinted in Chapter 3, we can distinguish between two different levels of parallelism:

- ***Intra-module parallelism***: the computation of a module is activated by receiving messages (i.e. tasks) from a set of source modules according to a non-deterministic or data-flow semantics. For each activation the module starts either a sequential or a parallel computation. In the latter case the internal parallelization follows a

[1] In the rest of this chapter we will assume that parallel computations are expressed by a classic message-passing cooperation model.

specific structured paradigm: i.e. *intra-module parallelism is expressed by instantiating well-known structured parallelism schemes* (see Chapter 3), for which the parallel organization and the properties of the parallelization are known and clearly identified. This aspect represents a fundamental feature of our methodology;

- ***Inter-module parallelism***: *modules can be composed in computation graphs with a general structure*. Modules can represent different subjects taking part of the whole computation (e.g. as in a client-server system), or can correspond to different application phases involving distinct and time-consuming computations.

The methodology that we are introducing is aimed to completely model the performance at any level, formalizing the internal behavior of a single module and the performance of the entire computation graph.

In this chapter we provide a performance modeling approach for steam-based distributed parallel computations expressed in terms of fundamental results in the area of Queueing Theory and Queueing Networks. In this way we will be able to formalize important issues related to:

- how to evaluate the performance of a graph computation starting from the knowledge of the performance of each module;
- how to evaluate the effective performance of a module based on the ideal performance behavior of all the modules of the computation graph;
- how to detect bottlenecks in a computation graph, that is modules that seriously limit the performance of the entire application.

The basic idea consists in modeling the performance of a module M (e.g either sequential or internally parallel) by abstracting its behavior as a *queueing system*, as shown in Figure 4.1. This scheme is a logical one, not necessarily corresponding to the real im-

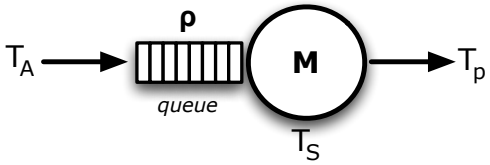


Figure 4.1: A computation module modeled as a queueing system.

plementation. However, it is aimed at capturing the essential elements of the problem at hand. For example, in same real cases there exist distinct input communication channels of a module: a unique queue in front of M could not exist physically, however it is emulated by a set of channel buffers in the source and destination nodes. The behavior of a queueing system is characterized by expressing five different parameters:

1. the *service discipline*: if not explicitly defined the FIFO policy is assumed;

2. the *queue size*, that is the number of buffer positions available for storing the incoming requests to the module;

3. the probability distribution of a random variable *inter-arrival time* t_a (i.e. the time interval between two consecutive arrivals of requests), with average value T_A and (optionally) variance σ_A;

4. the probability distribution of a random variable *(ideal) service time* t_s, which represents the ideal time passing between the beginning of the executions on two consecutive stream elements. The term ideal indicates that this parameter depends only on the internal features of the module *in isolation*, not considering the interactions with other modules of the computation graph. We denote with T_S and σ_S the average value and (optionally) the variance of this random variable;

5. the probability distribution of a random variable *inter-departure time* t_p (with average value T_p and optionally variance σ_P), which indicates the time between two successive result departures from the module. It is worth noting that this parameter can be higher than the ideal service time, due to the interactions with other modules as we will discuss in more detail later in this chapter.

A central parameter for our performance evaluation is the ***utilization factor*** ρ of a computation module, defined by the following ratio:

$$\rho = \frac{T_S}{T_A} \tag{4.1}$$

It expresses a global, average measure of the congestion degree, or traffic intensity, of requests to the queueing system. Large values represent high congestion degrees whereas small utilization factors consist in a more limited workload condition to the node.

Each computation module can be abstracted as a queueing system and the computation graph can be described as a network of queues [53], where the departures of some nodes form the arrivals of others. From the network topology viewpoint queueing networks can be categorized into two broad classes namely **open queueing networks** and **closed queueing networks**. In an open queueing network a possibly infinite number of requests are generated by source nodes, go through several nodes or even revisit a particular node more than once and finally leave the system. On the other hand, in a closed queueing network requests neither arrive at nor depart from the system, but a fixed number of requests continuously circulate through the nodes of the network. Open and closed networks are powerful modeling tools that have been applied for formalizing the performance behavior of different classes of systems. Open networks have been used for modeling flows as in traffic models and notably in data networks. Closed networks are considered a valuable tools for modeling systems where there exists a finite task population. CPU scheduling [78], supply-chain manufacturing systems [79] and window-type network flow protocols [80] are typical examples.

Stream-based parallel computations can be modeled using queueing networks models. In this section we consider two general computation graph structures:

Acyclic computation graphs describe complex distributed applications involving several computing phases. A large set of tasks is generated by source modules. Each task passes through the modules following a certain routing: each module performs a specific computation for each received task. This graph can be modeled by *acyclic open queueing networks*: i.e. a request in the network can pass through any particular node at most once.

Cyclic computation graphs describe parallel computations exhibiting a request-reply behavior. Notable examples are *client-server* computations, in which client modules transmit requests and then wait for the corresponding results from a server module. These graphs can be viewed as *cyclic closed queueing networks*.

Based on the previous distinction in the rest of this section we will describe a performance modeling methodology for these two classes of computation graphs. The following considerations will be completely independent w.r.t the internal behavior of each computation module, i.e. if it is sequential or internally parallel. In fact, the only essential parameter is the average value of the ideal service time of each module in isolation and, in some cases, its probability distribution. The behavior of intra-module parallelism will be described for structured parallelism schemes in Section 4.2 and, as we will see, it will be based on the very same results provided in the following sections for general computation graphs.

4.1.1 Acyclic Computation Graphs: analytical treatment

For acyclic graphs, Queueing Networks theory is a sufficiently powerful methodology for our modeling purposes. It does not utilize an explicit analytical treatment in terms of probability distributions, instead the performance modeling is expressed in terms of some basic results about the information flow in the network, the presence of bottlenecks and the average values of inter-arrival time, service time and inter-departure time variables [2]. In the rest of the discussion we will assume that each computation module produces exactly one output stream value for each received task. This assumption simplifies the model construction without limiting its scope, since many stream-based computations can be described as a graph following this behavior (or reducible to it).

In order to evaluate acyclic graphs of computation modules, we consider two interrelated phases of the performance analysis:

- ***Transient analysis*** consists in a formal study of the network behavior in the initial *transient phase* of the execution. For transient phase we intend the initial situation in which the performance behavior of each node can significantly change at relatively short time periods due to the starting conditions of the network (e.g. due to the size of the queues). This analysis is aimed to evaluate for each node its utilization factor and to discover the presence of ***bottlenecks***: *when* $\rho > 1$ *a node represents*

[2] Unless otherwise noted, in this section we will implicitly intend ideal service time, inter-arrival time and inter-departure time as average measurements.

a bottleneck, since it is not able to process the in-coming requests at their arrival rate but the information flow is delayed by the node presence;

- ***Steady-state analysis*** provides results for evaluating the effective performance (i.e. the mean inter-departure times[3]) of each node in the network during the *steady-state phase*. For steady-state phase we intend when the performance behavior of each module is completely stabilized and it is no longer influenced by the initial conditions.

During the transient phase the main parameter influencing our performance analysis is the ideal service time of each node in the network. As said before, this parameter indicates the average time passing between the beginning of the computations on two consecutive stream elements, in the ideal case in which the computation module is analyzed in isolation i.e. without considering its interactions with the other modules of the computation graph. In this ideal case the inter-departure time from a module coincides with its service time. This is no longer true if the module is considered as part of a computation graph structure. The speed of the output flow of results from the module can be delayed by the interactions with other modules in the network. This means that the mean inter-departure time T_p from a node can really be higher than its ideal service time T_S. In particular $T_p = T_S + \Delta$, where $\Delta \geq 0$ is a delay induced by two possible conditions:

- a node M may receive service requests with an inter-arrival time higher than its ideal service time. This means that after the completion of the service on a stream element, the node must wait (it is blocked) for the reception of the next one before starting the successive service request;

- in real systems queues have a fixed maximum size in terms of in-coming tasks received by other nodes. If a task attempts to enter a full capacity destination queue upon completion of a service at node M, it is forced to wait in this node until the destination node has a free position in its queue. During this phase the source module M stops processing tasks (it is blocked) until destination node completes a task service. This behavior is known as *blocking-after-service* [81] and it is the most common semantics in many concurrent systems in which computing entities exchange asynchronously messages onto channels with a limited buffer size.

Let us suppose that during our performance analysis we study the behavior of a computation module M:

- if the inter-arrival time to M is greater than its ideal service time (i.e. its utilization factor is less than 1), the inter-departure time from M equals its inter-arrival time and the node is not a bottleneck (at steady-state its utilization factor ρ keeps to be lower than 1);

[3]The effective performance level achieved by a module is represented by its mean inter-departure time assumed at steady-state.

- if the inter-arrival time is lower than its ideal service time, M is a bottleneck and during the transient phase its utilization factor results greater than 1. When its input queue becomes full, upstream nodes start to be blocked and the effect is that at steady-state the effective inter-arrival time to M will increase in such a way as to coincide with its ideal service time. *This means that the condition $\rho > 1$ is only a transient one*.

In order to discover bottlenecks, we are interested in studying both the transient behavior and the steady-state behavior of a computation graph. During the transient analysis of a module, we can consider two possible situations:

- if $\rho \leq 1$, the module is not a bottleneck and, on the average, its transient behavior coincides with the steady-state behavior;

- if $\rho > 1$, the module is a bottleneck and a non-null transient phase exists before reaching the steady-state behavior. Once the behavior is stable, the inter-arrival time to the module becomes equal to its service time (if there are no greater bottlenecks in the graph).

These conditions can be summarized in the following proposition, verified by flow conservation:

Proposition 4.1.1 (Steady-state behavior of a node)**.** *At steady-state the effective inter-arrival time of each node is equal to its inter-departure time. If that inter-arrival time also coincides with the ideal service time of the node, the node is a bottleneck and its utilization factor stabilizes to* 1. *Otherwise the node is not a bottleneck and its utilization factor stabilizes to a value less than* 1.

In the rest of this section we will provide the basic results for studying acyclic computation graphs. In particular we are interested in analytical results that allow us to study the graph behavior during the transient phase, identify the bottleneck nodes and their blocking effects on the other modules in order to determine the long-term, steady-state behavior of the graph.

In the following discussion we will start by considering deterministic service times: i.e. initially we will suppose constant service times (with zero variance) for each node of the network. Later in this chapter we will describe the impact of randomness on the results of this analysis.

4.1.1.1 Analysis of Tandem Queueing Systems

We start from a first situation in which two nodes are joined in series as depicted in Figure 4.2. Requests are generated by the first node S_1 and will join the second one S_2. In other words the departing requests from the first module form the arrivals to the second one. Let us suppose that these two nodes are characterized by ideal service times T_{S1} and

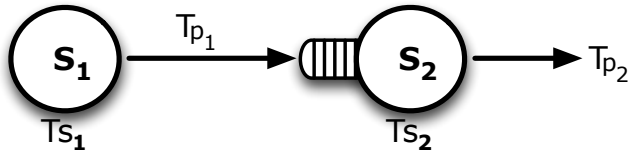


Figure 4.2: Two-queue tandem system.

T_{S2} respectively. In the initial transient phase of the execution, the utilization factor of the second node is determined by the following ratio:

$$\rho_{S_2} = \frac{T_{S2}}{T_A} = \frac{T_{S2}}{T_{S1}}$$

i.e. at the beginning of the system execution the inter-arrival time T_A to the second node corresponds to the service time of the first node in the network. At this point we can evaluate what nodes are bottlenecks and the effective behavior of each node at steady-state, that is the inter-departure times T_{p1} and T_{p2} from the two modules.

The second module is the bottleneck iff its utilization factor is greater than 1 (i.e. if $T_{S2} > T_{S1}$). Let us consider the case in which this is not true, so the service time of the first node is not smaller than the second one: $T_{S1} \geq T_{S2}$. This scenario is schematized in Figure 4.3. In this case the bottleneck node of the graph is the first one. If a node

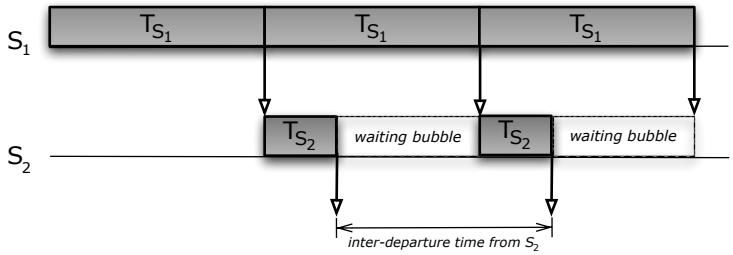


Figure 4.3: Two-queue tandem analysis: first node is the bottleneck.

is a bottleneck, at steady-state its inter-departure time coincides with its ideal service time because in the average case the node is not blocked due to communications with other nodes in the network. Therefore we have that $T_{p1} = T_{S1}$. For the second node its utilization factor is less than 1 (the node is under-utilized) and it is periodically blocked for receiving a request from S_1. From Figure 4.3 we can see that S_2 is delayed by a waiting bubble Δ equal to $T_{S1} - T_{S2}$. Thus the inter-departure time from S_2 is given by:

$$T_{p2} = T_{S1} + \Delta = T_{S2} + (T_{S1} - T_{S2}) = T_{S1}$$

This means that the inter-departure time from the second node equals the ideal service time of the first one.

The opposite case considers the situation in which the utilization factor of the second node is greater than 1, thus $T_{S2} > T_{S1}$. After an initial transient phase the queue of the second node becomes full and the steady-state behavior is depicted in Figure 4.4. Upon

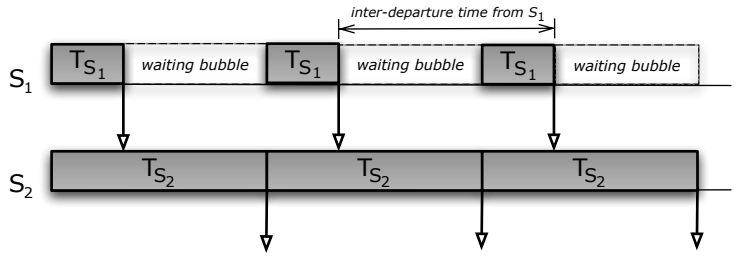


Figure 4.4: Two-queue tandem analysis: second node is the bottleneck (steady-state behavior).

the completion of the current service, S_1 is not able to transmit the next request to the second node S_2 until this node frees a position in its queue. This means that the first node is delayed by the remaining service time in the second node, which is equal to $\Delta = (T_{S2} - T_{S1})$. Hence at steady-state the inter-departure time from the first node becomes:

$$T_{p1} = T_{S1} + \Delta = T_{S1} + (T_{S2} - T_{S1}) = T_{S2}$$

Therefore the inter-departure time from the first node equals the ideal service time of the second node. Moreover, since the second node is the bottleneck of the graph, also its inter-departure time equals its service time: i.e. $T_{p2} = T_{S2}$.

We can summarize the previous results:

$$T_{p1} = T_{p2} = \max\{T_{S1}, T_{S2}\} \tag{4.2}$$

At steady-state the effective behavior of the two nodes is equal to the maximum ideal service time of the two modules of the network.

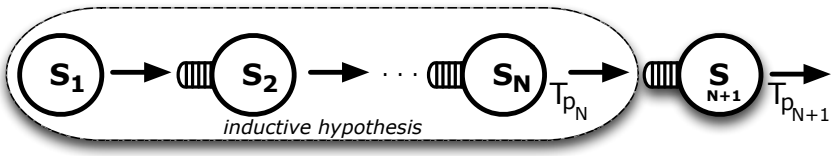


Figure 4.5: Performance analysis of a pipeline graph.

The previous results can be generalized to a tandem system of an arbitrary number of nodes (see Figure 4.5), also known as the *pipeline graph*. In this case the following proposition holds:

Proposition 4.1.2 (Pipeline)**.** *In a pipeline graph the bottleneck is the node with the highest ideal service time. Moreover, at steady-state the inter-departure times from each node of the graph stabilize to that ideal service time.*

Proof. The proposition can be proved by induction on the pipeline length. The two-queue tandem system is the base. Hence we can directly consider the inductive case: we have a pipeline of N nodes and we know from the inductive hypothesis that the inter-departure time T_{p_i} from each node is equal to the maximum ideal service time of the N nodes in the graph: i.e. $\forall i = 1,2,\dots N\ T_{p_i} = T_{S_Z} = \max_{j=1}^{N}\{T_{S_j}\}$, where S_Z is the bottleneck node. Now, in order to complete the inductive reasoning, we consider the presence of a further node S_{N+1} with ideal service time $T_{S_{N+1}}$, which is added at the end of the graph (see Figure 4.5). We have two possible situations:

- if $T_{S_{N+1}} \leq T_{S_Z}$ the inter-arrival time to S_{N+1} (which is equal to the inter-departure time from S_N i.e. $T_{p_N} = T_{S_Z}$) is greater than the service time of the new node. We are in the very same scenario as the one depicted in Figure 4.3. The new node is periodically blocked to wait for the reception of the next request from node S_N. This means that the inter-departure time from S_{N+1} equals the inter-departure time from S_N, that is $T_{p_{N+1}} = T_{p_N} = T_{S_Z}$. Therefore the proposition is verified;

- let us consider the case $T_{S_{N+1}} > T_{S_Z}$. During the transient phase the inter-arrival time to the new node is equal to the inter-departure time from S_N (i.e. $T_{p_N} = T_{S_Z}$) and it is lower than the ideal service time $T_{S_{N+1}}$ of the new node. This means that we are in the case depicted in Figure 4.4. Since $T_{S_{N+1}} > T_{S_Z} = \max_{j=1}^{N}\{T_{S_j}\}$, S_{N+1} becomes the new bottleneck of the graph and its inter-departure time equals its service time, i.e. $T_{p_{N+1}} = T_{S_{N+1}}$. At steady-state the node S_N is periodically blocked for transmitting a new request to S_{N+1}, and its inter-departure time adapts to the service time of the new bottleneck node $T_{p_N} = T_{S_{N+1}}$. The reasoning can be repeated up to the first node S_1, proving that the inter-departure time from each node of the pipeline equals the service time of S_{N+1}. The proposition is demonstrated also in this case.

□

4.1.1.2 Analysis of a Queueing Node with multiple destinations

Let us suppose to have a graph composed of a source node S and a set of destination nodes $D_1, D_2, \dots, D_N$ (Figure 4.6). A typical situation is when each destination is able to provide a specific service to the source node: each request from S is routed to a destination node according to a certain probability. Let p_i the probability that a request from S is directed to D_i, where:

$$\sum_{i=1}^{N} p_i = 1$$

If we indicate with T_{D_i} the ideal service time of each D_i and with T_S the ideal service time of the source, a crucial point is to determine the initial inter-arrival time T_{A_i} to each destination and thus if they are bottlenecks or not. Initially the inter-departure time from the source coincides with its ideal service time, i.e. $T_{p_S} = T_S$. The following result allows us to determine the inter-arrival time to each destination during the initial transient phase:

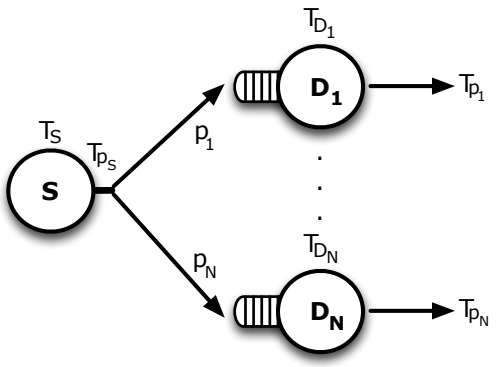


Figure 4.6: Example of a queueing node with multiple destinations.

Proposition 4.1.3 (Inter-arrival time during the transient phase)**.** *During the initial transient phase, the inter-arrival time T_{A_i} to each destination D_i is given by:*

$$T_{A_i} = \frac{T_{pS}}{p_i} \tag{4.3}$$

Proof. Node S transmits requests to a particular destination node D_i with probability p_i and to the set of the other destinations with probability $1 - p_i$. For this reason the inter-arrival time t_{A_i} is a random discrete variable with the following distribution:

t_{A_i}	Probability
T_{pS}	p_i
$2T_{pS}$	$p_i(1-p_i)$
$\dots$	$\dots$
NT_{pS}	$p_i(1-p_i)^{N-1}$

Thus the mean inter-arrival time T_{A_i} is given by[4]:

$$T_{A_i} = \sum_{n=0}^{\infty} n\,T_{pS}\,p_i\,(1-p_i)^{n-1} = \frac{p_i T_{pS}}{(1-p_i)} \sum_{n=0}^{\infty} n\,(1-p_i)^n = \frac{p_i T_{pS}}{(1-p_i)} \cdot \frac{(1-p_i)}{p_i^2} = \frac{T_{pS}}{p_i}$$

□

The previous result allows us to determine the utilization factor of each destination node, and thus to establish if some of the destinations are bottlenecks or not. If no destination node is a bottleneck, its inter-arrival time is greater than its ideal service time,

[4]We use the general property: $\sum_{n=0}^{\infty} n x^n = \frac{x}{(1-x)^2}$ for $x < 1$.

i.e. $T_{A_i} \geq T_{D_i}\ \forall i = 1,2,\ldots,N$. In this case the result of Proposition 4.1.3 is also valid at steady-state, and the effective inter-departure time T_{p_i} from each destination node equals the inter-arrival time: i.e. $T_{p_i} = T_{A_i}$.

On the other hand, if at least one destination node is a bottleneck, the steady-state inter-arrival times can no more be derived independently each other: i.e. the bottleneck influences the effective inter-arrival times to the other destination nodes of the network. Fortunately we can prove that only the worst bottleneck node, that is the one with the highest utilization factor, influences the effective inter-arrival times. In fact the following proposition holds:

Proposition 4.1.4 (Steady-state behavior of a multiple-destination queue)**.** *If at least one destination node is a bottleneck ($\exists i : \rho_i > 1$), let us denote D_Z the node with the highest utilization factor: i.e. $\rho_z = \max_{i=1}^{N} \rho_i$. The steady-state inter-arrival time T'_{A_i} to each destination node D_i can be expressed in function of the service time of D_Z:*

$$T'_{A_i} = T_{D_Z} \cdot \frac{p_z}{p_i} \tag{4.4}$$

Proof. The bottleneck existence introduces a delay in S with probability p_z, i.e. the steady-state inter-departure time T'_{p_S} from S is increased w.r.t the transient one T_{p_S}. From Proposition 4.1.1 we know that the new inter-departure time from S is a value such that:

$$\frac{T'_{p_S}}{p_z} = T_{D_z} \text{ thus } T'_{p_S} = T_{D_z} \cdot p_z$$

After this correction no destination node can have a utilization factor higher than 1. This can be proved by absurd. Suppose that there exists a destination node D_j different from D_Z which remains a bottleneck. This means that the following inequality is verified:

$$\frac{T_{D_j}}{T'_{A_j}} > 1$$

where T'_{A_j} is the corrected inter-arrival time to D_j which is equal to T'_{p_S}/p_j. By expanding the expression we have:

$$\frac{T_{D_j}}{T'_{p_S}} \cdot p_j = \frac{T_{D_j}}{T_{D_Z} \cdot p_z} \cdot p_j > 1$$

That can be transformed into:

$$T_{D_j} \cdot p_j > T_{D_Z} \cdot p_z \rightarrow \frac{T_{D_j} \cdot p_j}{T_{p_S}} > \frac{T_{D_Z} \cdot p_z}{T_{p_S}} \rightarrow \rho_j > \rho_z$$

which is absurd, since by initial hypothesis D_Z was the destination node with the highest utilization factor. The consequence of this fact is that, as steady-state, no further destination node is a bottleneck anymore. Therefore the effective inter-arrival time to the

destination nodes is given by:

$$T_{A_i}^{'} = \frac{T_{pS}^{'}}{p_i} = T_{D_z} \cdot \frac{p_z}{p_i}$$

□

With the previous result only D_z is the real bottleneck of the graph, since it influences the steady-state behavior of all the other nodes of the network. This means that its ideal service time coincides with its inter-departure time (and also with its corrected inter-arrival time). For all the other destinations instead, their effective inter-arrival times are higher than their ideal service times (i.e. they are under-utilized) so $T_{p_i} = T_{A_i}^{'}$.

4.1.1.3 Analysis of a Queueing Node with multiple sources

Let us consider a node D that accepts service requests from a set of sources (clients) $C_1, C_2, \ldots, C_N$ (see Figure 4.7). Let us denote with T_D the ideal service time of D and with T_{p_i} the inter-departure time from each client C_i initially equal to their ideal service times $T_{p_i} = T_{S_i}$. We can note that this graph does not contain cycles: the node D starts a

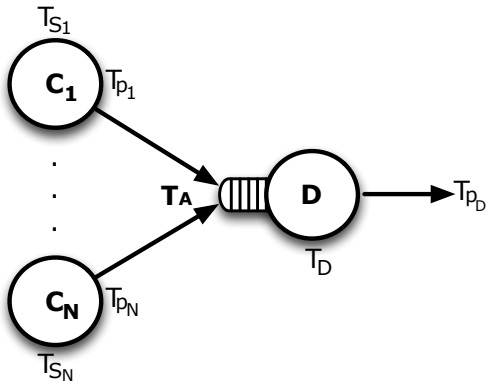


Figure 4.7: Example of a queueing node with multiple sources.

service whenever a request is present in its queue and the results are transmitted outside the depicted network, e.g. to further destination nodes. The total inter-arrival time T_A to D during the transient phase can be determined by applying the following proposition:

Proposition 4.1.5 (Total inter-arrival time during the transient phase)**.** *If a node D has multiple sources each one with an initial inter-departure time T_{p_i} to D, during the transient phase the total inter-arrival time to D is given by:*

$$T_A = \frac{1}{\sum_{i=1}^{N} \frac{1}{T_{p_i}}} \tag{4.5}$$

Proof. The inverse of the inter-arrival time is the arrival rate (or frequency), that is the number of requests received in a time unit. The arrival rate from each client is equal to:

$$\lambda_i = \frac{1}{T_{p_i}}$$

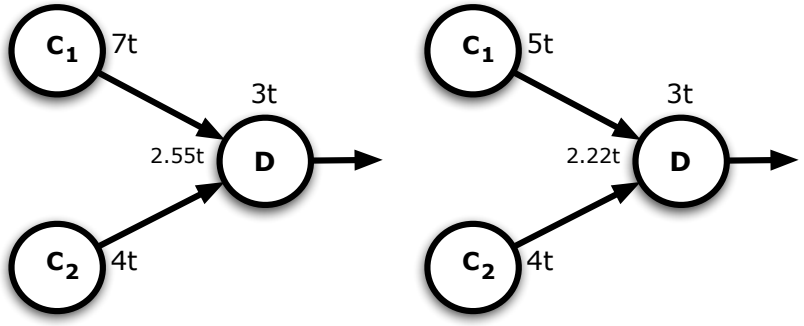
The total number of requests received by D in a time unit can be determined by summing the individual arrival rates from each clients: i.e. $\lambda_{tot} = \sum_{i=1}^{N} \lambda_i$. Thus we have:

$$T_A = \frac{1}{\lambda_{tot}} = \frac{1}{\sum\limits_{i=1}^{N} \lambda_i} = \frac{1}{\sum\limits_{i=1}^{N} \frac{1}{T_{p_i}}}$$

□

Once the total inter-arrival time to D has been determined, we can calculate its utilization factor. If $\rho_D \leq 1$ the inter-departure time from D equals its inter-arrival time (the node is periodically blocked for receiving the tasks from clients), i.e. $T_{p_D} = T_A$ and the client inter-departure times continue to be equal to their ideal service times also at steady-state. On the other hand if $\rho_D > 1$, D is the bottleneck and its inter-departure time equals its ideal service time: i.e. $T_{p_D} = T_D$. In this case we need to determine the steady-state inter-departure times from clients, that now will be greater than their ideal service times due to blocking events. This problem can be exemplified with the following example.

Example. Let us consider the following graphs in which the queueing node D is interconnected with two distinct sources C_1 and C_2.



(a) First example. (b) Second example.

Figure 4.8: Two examples of queueing nodes with multiple sources.

In the first case (Figure 4.8a) the initial inter-departure times from the two sources are equal to their ideal service times $7t$ and $4t$ and the server has an ideal service time equal to $3t$ (where t is a standardized time unit). In this scenario D has a utilization factor greater than one, since its total inter-arrival time T_A is equal to $\sim 2.55t$. Therefore the

steady-state inter-departure time from the two sources will be greater than their initial values. The steady-state behavior of the network has been evaluated through a Queueing Network simulator (*Java Modelling Tool* - JMT [82]). JMT is an open-source portable simulation environment written in Java, for performance evaluation, capacity planning and modeling of computer and communication systems. The suite implements state-of-the-art techniques for asymptotic and simulative analysis of queueing networks, supporting extended features as blocking semantics. Simulation results (with $D/D/1/K$ queues) are shown in Table 4.1a.

Node	Initial T_{p_i}	Effective T_{p_i}
C_1	$7t$	$7.5t$
C_2	$4t$	$5t$

(a) First example.

Node	Initial T_{p_i}	Effective T_{p_i}
C_1	$5t$	$6t$
C_2	$4t$	$6t$

(b) Second example.

Table 4.1: Steady-state behavior of the two sources C_1 and C_2.

In the first case the steady-state inter-departure time from the two sources become $7.5t$ and $5t$. This means that C_1 and C_2 will be delayed by a mean waiting bubble of length $0.5t$ and $1t$ respectively. Moreover we can observe that, as said in Proposition 4.1.1, with the new inter-departure times the effective inter-arrival time to D equals its ideal service time (and its inter-departure time since it is the bottleneck). In Figure 4.8b is depicted another example in which the initial inter-departure time from the first source is set to $5t$ instead of $7t$, resulting in a higher utilization factor of the destination node w.r.t the first situation. The simulation results (Table 4.1b) give an effective inter-departure time equal to $6t$ for both the sources.

This problem is a critical issue in performance analysis of acyclic computation graphs. Determining the effective behavior of a set of source nodes when the destination is a bottleneck is a hard problem that requires to reason about the relative speeds of the involved nodes. For instance in the network shown in Figure 4.8b, the two sources are so fast that the destination in the average case alternates a service of the first source and a service of the second one. In the first example of Figure 4.8a, the first source is slow enough that the destination node is able to process a certain number of requests of the second source before serving a request of the first one. This results in a smaller waiting bubble for the two nodes w.r.t the second example. Although analytical models that provide the mean waiting bubble length in function of the relative speeds of the nodes can be studied, in the next section we will present a different approach to face with this problem that makes it possible an elegant performance modeling of acyclic graphs with particular structures.

4.1.1.4 An algorithm for Performance Analysis of Single-Source graphs

With the previous results we have the basic tools for evaluating the performance of complex application work-flows. An important problem is how we can apply the previous

results in order to define an automatic procedure for exploiting the performance analysis of acyclic graphs. In particular we need an algorithm designed for solving the following problem:

Problem 4.1.6 (Steady-state analysis of acyclic computation graphs)**.** *Given an acyclic computation graph $G = (V, E)$ in which nodes represent computation modules and edges data streams, we need a procedure that determines the steady-state inter-departure times from each node in the graph.*

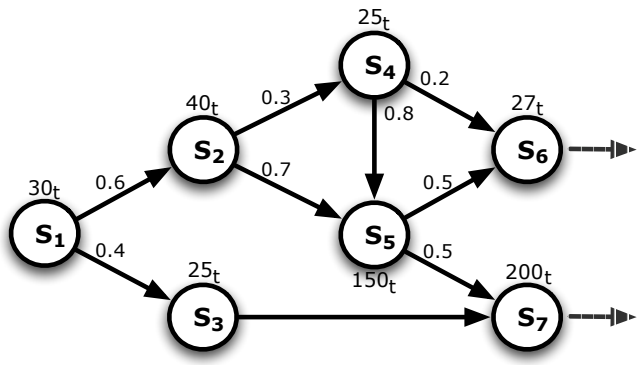


Figure 4.9: An acyclic computation graph labeled with the ideal service times of each node and the routing probabilities.

Given an *ideal graph*, i.e. a graph labeled with the ideal service times of each node, there corresponds a unique and well-identified *steady-state graph*, i.e. a graph labeled with the steady-state inter-departure times from each node. However a potentially infinite number of ideal graphs may have the same steady-state behavior. In this context we are looking for an algorithmic procedure for establishing the univocal correspondence between ideal graphs and steady-state graphs.

In Figure 4.9 is depicted an example of an acyclic computation graph of seven modules working asynchronously and cooperating by exchanging messages. In the graph each node is labeled with its ideal service time and, when a node has multiple out-going edges, arcs are labeled with the corresponding transmission probability. We need an algorithm with the following features:

- it performs an ordered graph traversal: to correctly establish for each node its inter-arrival time and thus its utilization factor, each node should be visited only when all its in-coming neighbors have been visited and their inter-departure times correctly determined;

- for each node of the graph it is necessary to calculate its inter-arrival time and its utilization factor in order to discover if it is a bottleneck or not. We have two possible situations: (1) the currently visited node is not a bottleneck, i.e. its ideal service

time is equal or less than its inter-arrival time and consequently the node influences neither the inter-departure times of the already visited nodes nor of the nodes that are still to be explored; (2) the current node is a bottleneck and it influences the inter-departure times of the previously explored nodes that need to be properly corrected.

The first requirement implies that the nodes of the graph should be visited according to a specific ordering. As it is known from basic notions in Graph Theory, every directed acyclic graph has at least one *topological ordering*, i.e. an ordering of its nodes such that the starting vertex of every edge occurs earlier in the ordering than the ending vertex. It can also be shown that an acyclic graph can have multiple topological orderings. Therefore let us suppose to have one of these topological orderings. We can observe that if we visit the nodes of the graph following this ordering, the first requirement will be achieved: each node will be visited iff all its in-coming neighbors have already been explored. An example of a topological ordering of the graph in Figure 4.9 is depicted in Figure 4.10.

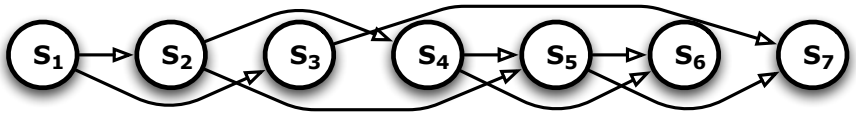


Figure 4.10: A topological ordering of the graph depicted in Figure 4.9.

We need a data-structure that represents a generic node of the graph. Each node n has four numerical attributes that describe: (1) its inter-arrival time; (2) its ideal service time; (3) its inter-departure time; (4) its utilization factor. Moreover the node maintains a list **OUT** of references to out-going neighbors and a list **IN** of pairs (n', p), where n' is a reference to one of its in-coming neighbors which transmits to n with probability p.

Class definition of the node data-structure

```
1 Class Node {
2     double T_A;
3     double T_S;
4     double T_p;
5     double ρ;
6     ListNode OUT;
7     ListPair IN;
8 }
```

These data-structures are properly initialized at the beginning of the algorithm execution. For each node the service time variable and the IN and OUT lists are properly initialized according to the structure of the input graph. The inter-arrival and the inter-departure times, and the utilization factors will be calculated by the algorithm. For each sink node

we assume the presence of a fictitious out-going edge such that the inter-departure time can always be defined. For each source node (although in-coming edges do not exist), the inter-arrival time is kept to be equal to the inter-departure time from that node (and furthermore at the beginning of the execution it coincides with the ideal service time of the node too). The algorithm evolves as follows:

1. the inputs are a directed graph $G = (V, E)$ and one of its topological ordering $\mathscr{S}$ (represented as an array of $|V|$ nodes);

2. the algorithm performs the graph traversal by visiting each node following the ordering $\mathscr{S}$. For each explored node its inter-arrival time and its actual utilization factor are determined. Therefore we are able to identify if the node is a bottleneck or not;

3. if the currently explored node is not a bottleneck ($\rho \leq 1$), its inter-departure time equals its inter-arrival time and the graph traversal continues with the next node in the topological ordering $\mathscr{S}$;

4. if the explored node is a bottleneck ($\rho > 1$), its inter-departure time coincides with its ideal service time. At this point the algorithm updates the inter-departure times of the nodes already visited in the graph ordering;

5. the algorithm ends when all the nodes have been explored and no bottleneck has been discovered (i.e. nodes have a utilization factor less or equal to 1).

The fourth point is the most critical one. In this section we introduce an algorithmic procedure for acyclic graphs in which ***there is exactly one source node***. In this case we are able to define an algorithm whose correctness can be proved by introducing the following invariant property:

Invariant 4.1.7. *When the i-th node in the input topological ordering $\mathscr{S}$ is visited, all the previously explored nodes (i.e. from the first one to the $(i-1)$-th of the ordering) have a utilization factor less or at most equal to* 1.

The invariant is satisfied at the beginning of the execution. Every topological ordering of a single source graph starts with the source node. As stated before, initially the (fictitious) inter-arrival time of the source is initialized to its ideal service time, that also coincides with its initial mean inter-departure time; thus the source utilization factor is initially equal to 1. If, during the graph traversal, no bottleneck node is discovered, the algorithm will end when the last node is visited. For each node its inter-departure time equals its inter-arrival time and the graph analysis is trivially completed.

On the other hand let us suppose that when the i-th node of ordering is visited, its utilization factor is greater than 1. This situation is depicted in Figure 4.11. We denote the currently discovered bottleneck the node B at position i of the topological ordering. Its ideal service time T_B is greater than its actual inter-arrival time T_A (i.e. $\rho_B > 1$). By the

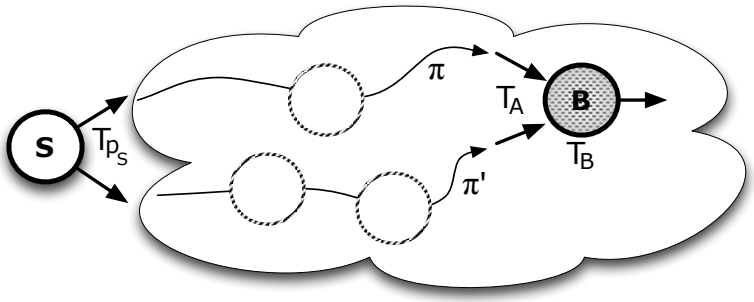


Figure 4.11: Bottleneck discovery.

invariant every previous node in the ordering has already been visited and its utilization factor is less (or equal) to 1. Therefore the actual inter-arrival time to B can be expressed in function of the actual inter-departure time T_{p_S} from the unique source node S of the graph. To this end we take the set $\mathscr{P}(S \to B)$ of all the paths in the graph starting from the source node S and ending to the current bottleneck node B. A path π is an ordered sequence of edges such that the origin of each is equal to the destination of its predecessor edge. E.g:

$$\pi = \Big\langle (N_1, p_1, N_2), (N_2, p_2, N_3), \ldots, (N_{k-1}, p_{k-1}, N_k) \Big\rangle$$

Where each directed edge is represented as a triple $e = (N, p, N^{'})$ where the first and the third element are the two end-point vertices of the edge and the second element is the probability that the first node transmits to the second one. For brevity we indicate with $e.p$ the probability corresponding to the edge e. The inter-arrival time to the bottleneck node is given by:

$$T_A = \left(\sum_{\forall \pi \in \mathscr{P}(S \to B)} \left(\frac{\prod_{\forall e \in \pi} e.p}{T_{p_S}} \right) \right)^{-1} \tag{4.6}$$

As we have seen the presence of the new discovered bottleneck node B influences the inter-departure times of all the previously explored nodes: i.e. they must be properly corrected. We know that after this correction, the new inter-arrival time $T_A^{'}$ to B must be equal to its ideal service time T_B (see Proposition 4.1.1), because B is the bottleneck. Thus, similarly to the previous case, we can express the new inter-arrival time to B in function of the corrected inter-departure time from the source node $T_{p_S}^{'}$:

$$T_A^{'} = \left(\sum_{\forall \pi \in \mathscr{P}(S \to B)} \left(\frac{\prod_{\forall e \in \pi} e.p}{T_{p_S}^{'}} \right) \right)^{-1} = T_B \tag{4.7}$$

In order to understand how we can correct the inter-departure time from the source, we can express the following relation: $T_{p_S}^{'} = T_{p_S} \cdot \alpha$ where α is a multiplicative factor. At this

point we need to find an α such that:

$$\left(\sum_{\forall \pi \in \mathscr{P}(S \to B)} \left(\frac{\prod_{\forall e \in \pi} e.p}{\alpha T_{pS}} \right) \right)^{-1} = T_B$$

We can rewrite the left-hand side of the previous equation in the following way:

$$\frac{\alpha T_{pS}}{\sum_{\forall \pi \in \mathscr{P}(S \to B)} \left(\prod_{\forall e \in \pi} e.p \right)} = T_B$$

Moving α in the right-hand side we obtain:

$$\frac{T_{pS}}{\sum_{\forall \pi \in \mathscr{P}(S \to B)} \left(\prod_{\forall e \in \pi} e.p \right)} = \frac{T_B}{\alpha}$$

We can observe that the left-hand side of the equation is now the original inter-arrival time T_A, and we can find the unique value of α such that the steady-state conditions are satisfied:

$$T_A = \frac{T_B}{\alpha} \text{ thus } \alpha = \frac{T_B}{T_A} = \rho_B$$

Therefore, when a new bottleneck node is discovered, we can correct the inter-departure time from the source node by multiplying its old inter-departure time by the utilization factor of the bottleneck node that has been discovered.

Proposition 4.1.8 (Invariant preservation)**.** *During the algorithm execution, if the currently visited node is the i-th of the topological ordering and it is a bottleneck ($\rho_i > 1$), we correct the inter-departure time from the source by multiplying this value by the utilization factor ρ_i. Then the algorithm is re-started from the beginning and, this time, when the i-th node is reached, its utilization factor will be equal to* 1 *and all the previous nodes in the ordering will continue to have a utilization factor less than* 1.

Proof. This proposition proves the correctness of the algorithm. Multiplying the inter-departure time of the source by the utilization factor of the discovered bottleneck, is the only way to achieve a new corrected inter-arrival time $T_A^{'}$ to B equal to its service time T_B. Since $\rho_B > 1$, this means that the corrected inter-departure time $T_{pS}^{'}$ will be greater than the original one T_{pS}, and thus the nodes preceding B in the ordering will continue to have a utilization factor less than 1. □

Based on this result Algorithm 1 presents an automatic procedure for single source acyclic graph analysis. The algorithm proceeds in the following fashion. All the nodes are visited according to an input topological ordering. For each node is calculated its

inter-arrival time by accessing its IN neighbor list (row 4). After that the utilization factor of the node is determined (row 5) and the bottleneck and non-bottleneck cases are examined. The most simply situation is the non-bottleneck case (from row 9 to 11): the inter-departure time of the current node is equal to the calculated inter-arrival time. Otherwise, in the bottleneck case (from row 6 to 8), the inter-departure time of the source (first node in the ordering) is corrected and the visit re-starts from the beginning.

Algorithm 1: Steady-state Analysis(G, S)

Data: a single-source acyclic graph $G = (V, E)$ and a topological ordering $\mathscr{S}$.
Result: at the end of the execution the attribute T_p of each node corresponds to the effective inter-departure time at steady-state.

```
1  begin
2      i ← 1;
3      while i ≤ |V| do
4          S[i].T_A = ( Σ_{(u,p)∈S[i].IN} p / u.T_p )^{-1};
5          S[i].ρ = S[i].T_S / S[i].T_A;
6          if S[i].ρ > 1 then bottleneck case
7              S[1].T_p = S[1].T_p · S[i].ρ;
8              i ← 1;
9          else not bottleneck case
10             S[i].T_p = S[i].T_A;
11             i ← i + 1;
```

Proposition 4.1.9 (Time complexity of steady-state analysis)**.** *At the worst case the time complexity of steady-state analysis is $O(|V|^2)$ for sparse graphs and $O(|V|^3)$ for dense graphs.*

Proof. The cost in terms of time complexity of a graph traversal (without any restart) is $O(|V| + |E|)$, since for each node its list *IN* is visited once (see row 4). The traversal of the graph will be re-started whenever a bottleneck node is discovered. Let us consider b the number of bottleneck nodes that are discovered during the algorithm execution, where $0 \leq b \leq |V|$. The complexity of steady-state analysis is $O(b \cdot (|V| + |E|))$ where at the worst case $b = |V|$ (i.e. whenever a node is explored for the first time it is a bottleneck). Therefore for sparse graphs (where $|E| = O(|V|)$) the time complexity is $O(|V|^2)$ whereas for dense graphs (where $|E| = O(|V|^2)$) is $O(|V|^3)$. We can also note that if no bottleneck node is discovered (i.e. $b = 0$), the time complexity of the algorithm is the same of a simple graph visit. □

For completeness we can observe that the algorithm takes as inputs the acyclic graph G but also one of its topological ordering $\mathscr{S}$. As it is well-known the time complexity for finding a topological ordering is the same of a DFS (*depth-first search*) traversal [83] of the graph, i.e. $O(|V|+|E|)$. Thus the cost of Algorithm 1 dominates the overall time complexity for the steady-state analysis.

Example. In Figure 4.12 is provided an example of steady-state analysis of an acyclic graph. Let us consider the input graph depicted in Figure 4.9 labeled with the ideal service times of each node and the routing probabilities. Figure 4.12 depicts the different phases of the algorithm execution following the topological ordering shown in Figure 4.10. Gray nodes represent explored vertices, white nodes correspond to vertices that are still to be explored whereas a point-based black node is the currently discovered bottleneck. The final results are summarized in Table 4.2.

Node	Service time	Inter-departure time	Utilization factor
S_1	$30t$	$136t$	.2205
S_2	$40t$	$227t$	.1762
S_3	$25t$	$341t$	.0733
S_4	$25t$	$758t$	.0330
S_5	$150t$	$242t$	.6198
S_6	$27t$	$429t$	.0629
S_7	$200t$	$200t$	1

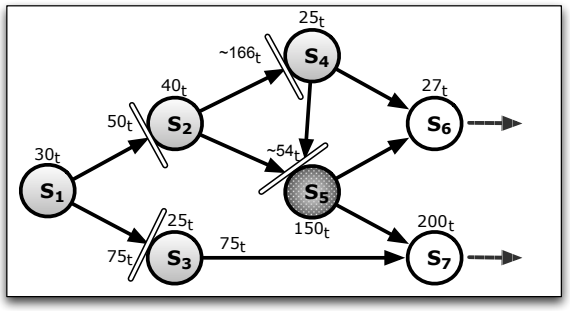
Table 4.2: Results of steady-state analysis.

We can note an important property of the final steady-state graph obtained at the end of the algorithm execution. The mean inter-departure time from the unique source node corresponds to the total inter-departure time from the sinks of the graph, i.e.:

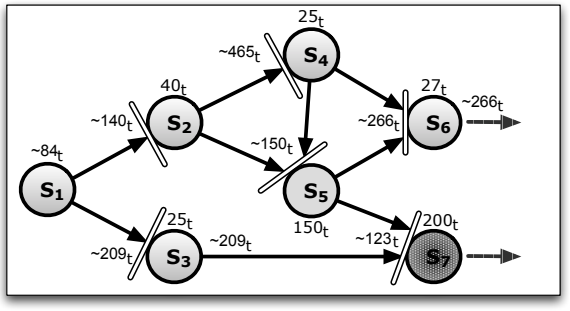
$$\frac{1}{\frac{1}{429t}+\frac{1}{200t}} \simeq 136t$$

where the total inter-departure time from the sinks can be calculated by taking the sum of the individual departure rates from each sink in a similar way to what we have done in Proposition 4.1.5. In light of this observation we can formulate the following general property characterizing the steady-state behavior of acyclic graphs:

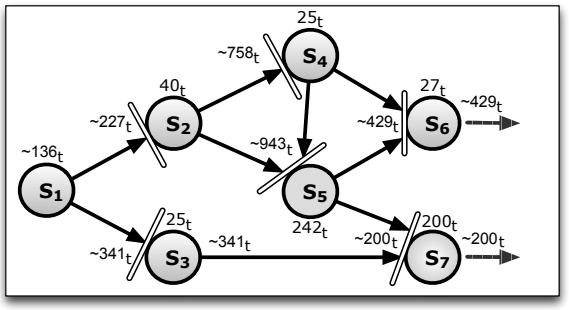
Proposition 4.1.10 (Steady-state property of the source and sink nodes)**.** *Given the final (single-source) graph after the execution of the steady-state analysis, the effective inter-departure time from the unique source always coincides with the total inter-departure time from the sink nodes.*



(a) The graph traversal starts from the node S_1. It is the unique source so its inter-arrival time is initially equal to its service time. Next, node S_2 is explored: the node is not a bottleneck since its inter-arrival time ($50t$) is greater than its service time. The same thing happens for nodes S_3 (with inter-arrival time $75t$) and for S_4 with inter-arrival time $166t$. When S_5 is discovered, its is a **bottleneck**: its inter-arrival time $54t$ is less than its service time $150t$ and its utilization factor is $\rho_5 = 2.79$. Therefore we update the inter-departure time of the source node that passes from $30t$ to $30t \cdot \rho_5 = 84t$.



(b) At this point the graph traversal re-starts from node 1. When S_5 is reached, it is not a bottleneck anymore (i.e. $\rho_5 = 1$). Now the node S_6 is explored and it is not a bottleneck (its inter-arrival time is $266t$). Then the last node S_7 is visited and its inter-arrival time $123t$ is less than its service time $200t$. So this node is a **bottleneck** and its utilization factor is $\rho_7 = 1.62$. Hence we update the inter-departure time of the source node that passes from $84t$ to $84t \cdot \rho_7 = 136t$.



(c) The graph traversal re-starts from node 1. At this point no bottleneck node is identified: i.e. for every node in the graph its utilization factor is now lower or equal to 1. The algorithm terminates correctly providing the steady-state behavior of the acyclic graph.

Figure 4.12: An example of steady-state analysis of a single-source acyclic graph.

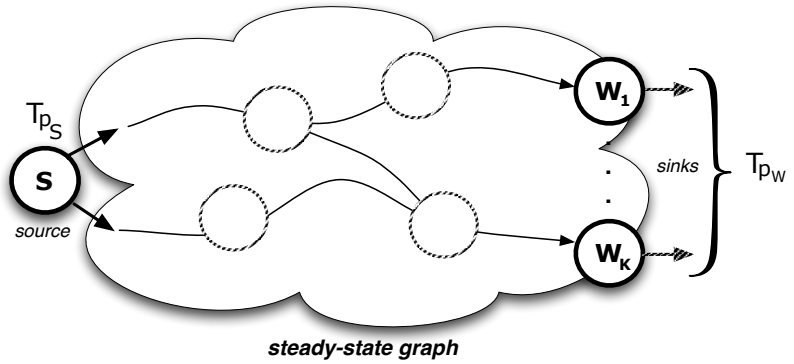


Figure 4.13: Steady-state property between the source node and the sinks.

Proof. The proof is based on a similar argument to the one used for demonstrating the invariant preservation (Proposition 4.1.8) of the algorithm. We know that at steady-state the utilization factors of all the graph nodes will be smaller or at most equal to 1. This means that the effective inter-arrival time to each sink node can be determined in function of the steady-state inter-departure time from the unique source, by taking all the paths from the source to that sink. Let us consider Figure 4.13, where S is the unique source with inter-departure time T_{pS} and $W_1, \ldots, W_k$ are the sink nodes. The steady-state inter-arrival time to the i-th sink is given by:

$$\left(\sum_{\forall \pi \in \mathscr{P}(S \to W_i)} \left(\frac{\prod\limits_{\forall e \in \pi} e.p}{T_{pS}} \right) \right)^{-1}$$

Since at steady-state each node has a utilization factor smaller or equal to 1, the inter-departure time $T_{p_{W_i}}$ from each W_i is equal to its inter-arrival time. Therefore the total inter-departure time T_{pW} from the sinks is given by:

$$T_{pW} = \frac{1}{\sum\limits_{i=1}^{k} \left(\sum\limits_{\forall \pi \in \mathscr{P}(S \to W_i)} \left(\frac{\prod\limits_{\forall e \in \pi} e.p}{T_{pS}} \right) \right)} = \frac{1}{\underbrace{\frac{1}{T_{pS}} \sum\limits_{i=1}^{k} \left(\sum\limits_{\forall \pi \in \mathscr{P}(S \to W_i)} \left(\prod\limits_{\forall e \in \pi} e.p \right) \right)}_{=1}} = T_{pS}$$

which is equal to the steady-state inter-departure time from the source node. It is important to observe that the expression inside the horizontal brace is the sum of the probabilities for the different paths from the source to any sink node. Since we consider all the possible paths in the graph this probability is equal to 1. □

Finally we can express the following corollary of the previous proposition, that can be applied for acyclic graphs with a particular structure:

Proposition 4.1.11. *Given an acyclic graph with a single source and a single sink, at steady-state the inter-departure time from the source equals the inter-departure time from the sink.*

In other words, if the source and the sink are unique, we can state that at steady-state they behave in the same manner.

4.1.1.5 Impact of the randomness on the Performance Analysis of acyclic computation graphs

We conclude this section by providing a discussion about the impact of the randomness on the result accuracy of the algorithm. In the previous sections we have assumed deterministic service time distributions for each module of the graph: i.e. for each node its ideal service time assumes a fixed constant value throughout the execution. With these conditions the algorithm precision has been evaluated on several examples through the JMT simulator. The results demonstrate a high level of accuracy of the algorithm, which is able to accurately quantify the steady-state behavior of single-source acyclic graphs with a relative error w.r.t the simulations less than 1%. For the example of Figure 4.12 this behavior is summarized in Table 4.3.

Module	**Algorithm T_p**	**Simulation T_p**	**Error %**
S_1	136.40	136.90	.36450
S_2	227.33	228.56	.538152
S_3	341.00	341.49	.143489
S_4	757.78	761.07	.432286
S_5	241.84	243.04	.495232
S_6	428.93	430.54	.375662
S_7	200.00	200.00	.001764

Table 4.3: Accuracy of steady-state analysis with deterministic service time distributions.

Things become different if we introduce randomness. In this case the service times assume stochastic values following a known probability density function and a specific average value. A first interesting modeling consists in assuming that the service times of each node follow an *exponential distribution*. With this assumption each node in the graph is modeled as a $M/M/1/K$ queue (instead of $D/D/1/K$ queues as in the previous discussion). The main property of this distribution is the memoryless one: if we assume that each service request (task) is independent from the others, the service time of a task does not depend on the service times spent for the previous tasks calculated by the node. For stream-based parallel computations the independence among tasks is a reasonable assumption in many real cases.

All the introduced propositions for pipeline graphs and for multiple- destination and multiple-source queues still remain valid assuming that, instead of having fixed service times, we have proper average values. On the other hand, compared to the deterministic case, in the exponential case the size of each queue plays an important role for attenuating the randomness impact. In fact we expect that the results of the steady-state algorithm approximate well the behavior of a $M/M/1/K$ network if, for each node, the queue size

is large enough (but still bounded). For this reason we have simulated the behavior of the computation graph shown in Figure 4.9, in which the ideal service times are assumed to be the average values of corresponding exponential random variables. The simulations have been performed using the JMT simulator with different sizes of each queue length (denoted with K). Tests for 20, 10, 5, 3, 2 buffer positions are depicted in Table 4.4.

In the table are reported for each queue node the percentage errors between the simulation results and the inter-departure times obtained by the algorithm execution. As we can expect if we decrease the size of each queue the error increases. For this example we can note that for large enough queue sizes (e.g. 10 buffer positions), the errors are less than 2% for each node. For very small queue sizes (e.g. 3 and 2 buffer positions), the errors increase but they are still limited: i.e. $6 \div 8\%$ and $10 \div 11\%$ for each node respectively.

In order to have a more complete set of experiments, we have compared the results of the algorithm execution with different simulations in which we exploit other service time distributions with different parameters. Table 4.5 shows the mean inter-departure times achieved with service times following a *uniform distribution* with the same average values of the previous example. Similarly to the exponential case, also with uniform distributions the relative error increases with a smaller size of each queue. However in this case the increase is much slighter than with exponential service times. With very limited buffer sizes (e.g. 2 positions for each queue), the error is $2 \div 3\%$.

Tables 4.6, 4.7, 4.8, 4.9 and 4.10 show the mean inter-departure times T_p from each node of the graph in which the service times follow a *normal distribution* with the same average values of the previous examples. In this case, in addition to the queue size K, another critical parameter is represented by the *variance* of these random variables: i.e. the average deviation of the variable values around its mean. A high variance indicates that the measurements of the random variable are spread out over a large range of values. Therefore we have studied the simulation results varying the queue size K and the variance of each service time variable. Instead of using directly the variance, we have used the *standard deviation* (denoted with stddev) which is the square root of the variance and, thus, has the same unit of measurement of the service times (i.e. in this case the standardized time unit t). The tables show the results with different queue sizes ($K = 20, 10, 5, 3, 2$) and standard deviations ($stddev = 5t, 8t, 15t, 30t, 50t$). The results show that, especially with sufficiently large buffer sizes, the impact of the variance is extremely limited since the relative error of the simulations w.r.t the algorithm results is less than $1 \div 2$ % and it slightly increases with very limited queue sizes (e.g. 2 positions), but not exceeding $3 \div 4\%$ at the worst case.

These results suggest that the steady-state analysis algorithm is a useful and precise approach to measure the long-term, steady-state behavior of single-source acyclic graphs, independently of the service time distributions and their parameters, and also with a limited and practical queue size for each buffer.

	$K = 20$		$K = 10$		$K = 5$		$K = 3$		$K = 2$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	136.2	**0.14**	137.24	**0.61**	141.06	**3.42**	145.07	**6.36**	151.17	**10.83**
S_2	227.79	**0.20**	228.37	**0.46**	235.53	**3.61**	241.24	**6.12**	251.57	**10.66**
S_3	339.09	**0.56**	344.15	**0.92**	351.76	**3.16**	364.00	**6.75**	378.80	**11.09**
S_4	749.42	**1.10**	769.92	**1.60**	784.40	**3.51**	820.99	**8.34**	844.37	**11.43**
S_5	242.35	**0.21**	243.13	**0.53**	250.61	**3.63**	256.26	**5.96**	267.95	**10.80**
S_6	430.67	**0.41**	431.32	**0.56**	445.64	**3.90**	456.77	**6.49**	474.85	**10.71**
S_7	199.49	**0.25**	201.47	**0.73**	206.45	**3.23**	212.67	**6.33**	221.82	**10.91**

Table 4.4: Results with exponential service time distributions and different sizes of each queue (K).

	$K = 20$		$K = 10$		$K = 5$		$K = 3$		$K = 2$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	136.29	**0.08**	136.45	**0.04**	137.34	**0.69**	138.29	**1.38**	139.59	**2.34**
S_2	228.64	**0.57**	229.09	**0.77**	228.40	**0.47**	230.55	**1.42**	232.59	**2.32**
S_3	337.78	**0.94**	337.58	**1.00**	344.53	**1.03**	345.60	**1.35**	349.08	**2.37**
S_4	748.21	**1.26**	761.78	**0.53**	762.74	**0.65**	761.73	**0.52**	769.50	**1.55**
S_5	244.14	**0.95**	244.02	**0.90**	242.87	**0.42**	245.17	**1.38**	247.34	**2.27**
S_6	428.70	**0.05**	432.22	**0.77**	433.88	**1.15**	438.10	**2.14**	438.96	**2.34**
S_7	200.12	**0.06**	199.55	**0.23**	201.02	**0.51**	202.12	**1.06**	204.70	**2.35**

Table 4.5: Results with uniform service time distributions and different buffer sizes (K).

	$stddev = 5t$		$stddev = 8t$		$stddev = 15t$		$stddev = 30t$		$stddev = 50t$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	136.41	**0.01**	136.07	**0.24**	136.28	**0.09**	136.56	**0,12**	136.22	**0.13**
S_2	227.65	**0.14**	227.61	**0.12**	226.83	**0.22**	227.38	**0.02**	226.94	**0.17**
S_3	340.63	**0.11**	338.54	**0.72**	341.63	**0.19**	342.15	**0.34**	341.11	**0.03**
S_4	760.26	**0.33**	758.78	**0.13**	751.97	**0.77**	763.92	**0.81**	759.46	**0.22**
S_5	242.70	**0.36**	242.53	**0.29**	241.97	**0.06**	241.88	**0.02**	241.47	**0.15**
S_6	430.30	**0.32**	426.82	**0.49**	428.79	**0.03**	431.83	**0.68**	426.68	**0.53**
S_7	200.01	**0.00**	200.01	**0.01**	200.03	**0.01**	199.99	**0.01**	200.47	**0.23**

Table 4.6: Results with normal service time distributions and buffer size $K = 20$.

	$stddev = 5t$		$stddev = 8t$		$stddev = 15t$		$stddev = 30t$		$stddev = 50t$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	137.15	**0.55**	136.25	**0.11**	136.11	**0.21**	136.60	**0.14**	136.39	**0.01**
S_2	230.27	**1.29**	227.54	**0.09**	226.46	**0.38**	228.61	**0.57**	226.19	**0.50**
S_3	339.28	**0.51**	339.75	**0.37**	341.42	**0.12**	339.50	**0.44**	343.71	**0.80**
S_4	774.97	**2.27**	760.67	**0.38**	759.20	**0.19**	767.16	**1.24**	754.70	**0.41**
S_5	245.07	**1.33**	242.02	**0.08**	241.21	**0.26**	242.86	**0.42**	240.72	**0.47**
S_6	437.07	**1.90**	428.34	**0.14**	425.96	**0.69**	430.57	**0.38**	428.80	**0.03**
S_7	200.01	**0.01**	199.95	**0.03**	200.20	**0.10**	200.20	**0.11**	200.19	**0.09**

Table 4.7: Results with normal service time distributions and buffer size $K = 10$.

	$stddev = 5t$		$stddev = 8t$		$stddev = 15t$		$stddev = 30t$		$stddev = 50t$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	136.64	**0.18**	136.91	**0.37**	136.22	**0.13**	136.50	**0.08**	137.60	**0.88**
S_2	227.86	**0.23**	227.99	**0.29**	227.12	**0.09**	226.58	**0.33**	231.67	**1.91**
S_3	341.38	**0.11**	342.76	**0.52**	340.43	**0.17**	343.44	**0.72**	338.98	**0.59**
S_4	750.20	**1.00**	763.87	**0.80**	757.85	**0.01**	760.12	**0.31**	772.86	**1.98**
S_5	242.30	**0.19**	242.72	**0.36**	241.42	**0.17**	241.10	**0.30**	246.07	**1.75**
S_6	431.19	**0.53**	433.31	**1.02**	426.98	**0.46**	429.31	**0.09**	440.25	**2.64**
S_7	200.10	**0.05**	200.21	**0.10**	200.13	**0.06**	200.20	**0.10**	200.29	**0.15**

Table 4.8: Results with normal service time distributions and buffer size $K = 5$.

	$stddev = 5t$		$stddev = 8t$		$stddev = 15t$		$stddev = 30t$		$stddev = 50t$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	137.15	**0.55**	137.24	**0.62**	137.74	**0.98**	137.03	**0.46**	137.65	**0.92**
S_2	229.15	**0.80**	229.16	**0.81**	230.57	**1.42**	228.08	**0.33**	229.01	**0.74**
S_3	341.65	**0.19**	342.22	**0.36**	342.15	**0.34**	343.28	**0.67**	345.09	**1.20**
S_4	767.11	**1.23**	758.51	**0.10**	769.97	**1.61**	755.42	**0.31**	762.66	**0.64**
S_5	243.66	**0.75**	243.70	**0.77**	244.99	**1.30**	243.27	**0.59**	243.79	**0.81**
S_6	431.37	**0.57**	431.63	**0.63**	437.24	**1.94**	428.99	**0.01**	432.46	**0.82**
S_7	201.13	**0.56**	201.28	**0.64**	201.13	**0.57**	201.39	**0.70**	201.97	**0.99**

Table 4.9: Results with normal service time distributions and buffer size $K = 3$.

	$stddev = 5t$		$stddev = 8t$		$stddev = 15t$		$stddev = 30t$		$stddev = 50t$	
	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %	T_p	Err. %
Module										
S_1	138.50	**1.54**	139.13	**2.00**	138.87	**1.81**	139.69	**2.42**	140.20	**2.78**
S_2	231.72	**1.93**	231.03	**1.63**	231.32	**1.76**	233.27	**2.61**	233.19	**2.58**
S_3	344.33	**0.98**	349.81	**2.58**	347.47	**1.90**	348.27	**2.13**	351.59	**3.10**
S_4	767.77	**1.32**	777.03	**2.54**	759.56	**0.23**	787.47	**3.92**	778.38	**2.72**
S_5	246.35	**1.86**	245.27	**1.42**	246.55	**1.95**	247.67	**2.41**	247.73	**2.44**
S_6	433.76	**1.13**	440.18	**2.62**	437.33	**1.96**	445.05	**3.76**	442.70	**3.21**
S_7	203.52	**1.76**	203.47	**1.73**	203.50	**1.75**	203.64	**1.82**	205.21	**2.60**

Table 4.10: Results with normal service time distributions and buffer size $K = 2$.

4.1.2 Cyclic Computation Graphs: analytical treatment

In the previous section we saw that Queueing Networks are a sufficiently powerful methodology in order to determine the performance modeling of acyclic computation graphs. On the other hand the analytical treatment of Queueing Systems, in terms of probability distributions of inter-arrival and service times, is mainly necessary for cyclic graph computations exhibiting a request-reply behavior.

Let us consider a system in which a set of client modules $C_1, C_2, \ldots, C_N$ transmit to a server module S a set of requests and wait for an explicit reply in order to continue their elaboration. An example of these graphs is shown in Figure 4.14. As we can see this interaction yields to cycles in the communication pattern. The parameters of interest in evaluating the server performance are:

- the *mean queue length* L_q: the average number of client requests in the waiting queue of the server node;
- the *mean request number* N_q in the system: with respect to the queue length, it includes the number of requests currently in execution in the server computation;
- the *mean waiting time* W_q in queue: the average time spent by a request in the waiting queue of the server;
- the *mean response time* R_q: with respect to W_q includes the time spent on the currently served request(s). It is also called response time, and consists in the average time that a client waits before receiving the result of the requested service.

These parameters are related with each other in several ways. One of the most useful results is the ***Little's law*** [84]:

$$L_q = \frac{W_q}{T_A} \quad N_q = \frac{R_q}{T_A} \tag{4.8}$$

This law is a fundamental long-term relationship which ties together the concept of waiting and response time and the concept of population size of a queue. T_A is the total inter-arrival time to the server S, so its inverse represents the average frequency of arrivals. Other important relations are the following:

$$\begin{aligned} N_q &= L_q + \rho_s \\ R_q &= W_q + L_s \end{aligned} \tag{4.9}$$

The former holds since the average number of requests currently in the service phase is equal to the utilization factor of the server $\rho_s = T_S/T_A$, where T_S is the mean service time of S. The latter means that we add, to the average time spent in the waiting queue, the mean computation latency L_s of a service phase. This aspect is very important especially if the server module is internally parallel. As we know, several structured parallelizations of the server can be adopted, with different impacts on the mean service time and on

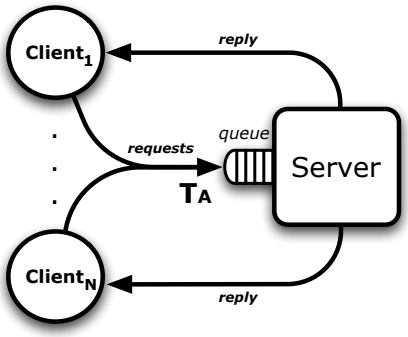


Figure 4.14: A Client-Server computation graph.

the mean computation latency. Therefore, in the response time expression, is important to consider the computation latency per request, which may be different from the mean service time in general.

The graph depicted in Figure 4.14 needs to be properly modeled from the performance viewpoint in order to evaluate the performance parameters of the server. Two important considerations emerge from the semantics of the request-reply behavior:

- each client generates the next request only when the result of the previous one has been received. This means that from a modelistic point of view it is equivalent to consider a finite population of tasks, as many as the global number of clients N, that circulate continuously and never leave the network;

- the network has a *self-stabilizing behavior*: i.e. a temporary increase in the inter-arrival time has the effect of a decrease in the server response time that tends to lower the inter-arrival time itself. In such kind of systems we cannot speak of bottlenecks, or at least with the same meaning exposed for acyclic graphs. However high values of ρ_s (tending to 1) cause a greater mean response time of the server.

In the following part we will provide the basic formulation inherited from [85] for modeling the performance of cyclic computation graphs of parallel modules. Notable cases will be client-server parallel applications.

4.1.2.1 Performance modeling of Client-Server parallel computations

For simplicity we assume that all clients have an identical behavior. Let T_C and T_S the ideal service times of a generic client and of the server. We need a model able to quantify the inter-departure time T_{p_i} from each client at steady-state. This inter-departure time will be certainly greater than the client ideal service time. In fact the request-reply behavior requires that, after the transmission of a request (transmitted by each client with a theoretical rate $1/T_C$), the client waits for an explicit reply from the server. Only after the reception of a reply the client can generate a further request. Therefore we expect that

T_{p_i} will be increased w.r.t T_C by the mean response time R_q of the server. The performance behavior of a client-server graph can be modeled through the following system of equations:

$$\begin{cases} T_{p_i} = T_C + R_q \\ \rho_s = \dfrac{T_S}{T_A} \\ R_q = W_q(\rho_s, T_S, T_A) + L_s \\ T_A = \dfrac{T_{p_i}}{N} \end{cases} \tag{4.10}$$

where the last equality is derived by applying the expression 4.5 in which N indicates the number of clients. The solution of the system is subject to constraint $\rho_s < 1$, due to the self-stabilizing behavior of the cyclic interaction. Proper mean waiting time (W_q) expressions, of second or higher order in ρ_s, can be derived by applying well-known results of Queueing Theory [84]. These expressions depend on the probability distributions of the random variables modeling the server service time and the inter-arrival times from clients. Notable cases are the following:

M/M/1 queue considers an exponential distribution of the inter-arrival time from clients and of the server service time. In this case the mean waiting time in queue can be approximated by:

$$W_q = \frac{T_S^2}{T_A - T_S} \tag{4.11}$$

M/G/1 queue considers an exponential distribution of the inter-arrival time from clients and a generic distribution for the server service time. A valuable approximation of the mean waiting time is given by the Pollaczek-Khinchnine formula:

$$W_q = \frac{\sigma_s + T_S^2}{2\,T_A - 2\,T_S} \tag{4.12}$$

Where σ_s in the M/G/1 expression indicates the variance of the random variable of server service time. A notable case is when the server service time distribution is deterministic (the server provides a constant service time). In this case (namely M/D/1 queue) the mean waiting time can be obtained from (4.12) with a zero variance, i.e. $\sigma_s = 0$.

The previous system of equations provides a simple way to study the performance behavior of client-server cyclic computation graphs from a qualitative point of view. In fact it is able to describe the request-reply interaction between computation modules and model the congestion degree of the server. Moreover it is able to describe how the mean response time is influenced by the relative speeds of the clients and of the server itself. On the other hand from a quantitative point of view the accuracy of results depends on the used W_q expression. In the case of the M/M/1 formula, which assumes an infinite queue and an unlimited population, the system reduces to a second order equation in ρ_s that can be trivially solved analytically. When this approximation is not sufficient, other

more precise and complex waiting time expressions can be adopted (e.g. considering a limited population and a finite queue as with the $M/M/1/K//N$ queue) resulting to a higher order system that can be solved by exploiting numerical techniques.

4.2 Performance modeling of Structured parallelism schemes

In the previous section we have presented a performance modeling for computation graphs of cooperating modules. In this approach we suppose to know the ideal service times of each computation module, without no knowledge about their internal structure (i.e. if they are sequential or parallel computations and what kind of parallelism has been used inside a module definition). However, in Chapter 3 we have seen that a central aspect characterizing our approach is the exploitation of structured parallel programming for defining the so-called intra-module parallelism. In this paradigm a parallel module exploits a limited set of well-known structured parallelism patterns whose behavior, in terms of data partitioning or replication and function replication, is well-defined.

As usual our performance analysis starts from the notion of mean service time of a computation module. In a message-passing programming model the behavior of an execution unit (e.g. a process or a thread) alternates computation phase and communication phase (in which messages are transmitted onto communication channels). In order to clearly take into account the communication aspects, two possibilities can happen:

- in the most classic situation the order of calculation and communication phases is strictly sequential. Therefore, the communication latency is entirely paid. In this case the ideal service time of a sequential module is composed by two terms: the calculation time T_{calc} for serving each input request, and a term L_{com} corresponding to the time spent in communication by the module, both for receiving requests and for transmitting the results. Therefore we have: $T_S = T_{calc} + L_{com}$. The sequential order of calculation and communication phases can be forced by semantics reasons (e.g. utilization of synchronous communications), or by the absence of proper architectural supports (e.g. processors dedicated to the execution of communication primitives);

- if proper architectural facilities are provided (e.g. every node of the underlying parallel architecture is equipped with an independent communication processor) and if the communication form is asynchronous, in a stream-based computation the communication latency can be partially or totally masked by the calculation phase. The ideal service time of a module is now equal to $T_S = \max\{T_{calc}, L_{com}\}$: i.e. in the case of a full overlapping the module ideal service time is equal to its calculation time whereas, if the length of calculation phase is lower than the communication latency, communications have a non-null impact on the service time.

Despite the previous differences, when we refer to the term ideal service time we will intend a proper composition of calculation and communication time. In the last part of this chapter we will consider the performance modeling of two notable cases of structured parallelism schemes (i.e. intra-module parallelism): the task-farm and the data-parallel schemes. This modeling allows us to define the ideal service time of a structured parallel computation, and thus the ideal service time of a parallel module which is a starting information for applying the computation graph analysis described in the sections before.

4.2.1 Task-Farm performance modeling

When a computation module performs a task-farm scheme, its internal parallelism is exploited by three classes of parallel entities (i.e. processes or threads): an emitter (which schedules input data), a collector (which collects output results) and a set of replicated workers which simultaneously execute a pure function on different stream elements. Let us evaluate the task-farm ideal service time T_{farm} by formally solving the internal acyclic graph (See Figure 4.15).

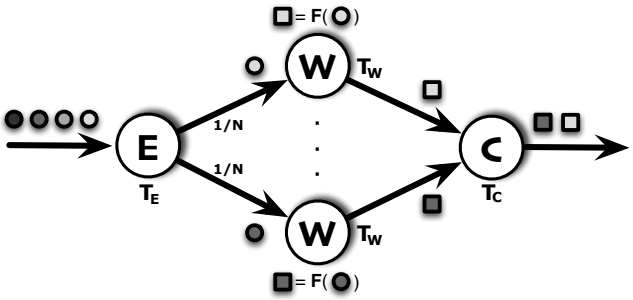


Figure 4.15: Task-Farm parallelism scheme.

Assuming a load-balanced scheduling strategy (e.g. emitter performs an on-demand distribution), the probability that an input stream element is sent to any worker is uniformly distributed and equal to $1/N$, where N is the parallelism degree in terms of number of workers. Thus, if T_E is the ideal service time of the emitter, the transient inter-arrival time to any worker is given by Proposition 4.1.3: i.e. T_E/N. Let us observe that workers are identical with a ideal service time T_W. Therefore we have two possibilities:

- if the workers are bottlenecks (their utilization factor is greater than one), their inter-departure time is equal to their ideal service time and the total inter-arrival time to the collector is given by Proposition 4.1.5: i.e. T_W/N;

- if the workers are not bottlenecks, their inter-departure time is equal to their inter-arrival time T_E/N and the total inter-arrival time to the collector is equal to T_E by Proposition 4.1.5.

The performance model of the task-farm in isolation is given by the following expression:

$$T_{farm} = \max\left\{T_E, \frac{T_W}{N}, T_C\right\} \tag{4.13}$$

Where the ideal service time of the task-farm T_{farm} is the inter-departure time from the collector. In the expression we have also account for the case in which the collector is the bottleneck. If T_C is its ideal service time, if C is the bottleneck the service time of the task-farm coincides with T_C. This analysis considers the task-farm in isolation. If the module coexists with other modules interconnected in computation graph structures, then the steady-state inter-departure time from the collector can be different from the ideal one, and it should be calculated by applying the methodology described in the previous sections of this chapter.

Although the service time is often the most important parameter for stream-based applications, also the computation latency may be of special importance especially if the task-farm is part of a closed graph exploiting a request-reply behavior. The latency expresses the time needed for completing a single task elaboration, thus it considers the sum of the delays of each phase of the task-farm scheme (without any overlapping between calculation and communication):

$$L_{farm} = T_E + T_W + T_C \tag{4.14}$$

Since each worker applies a pure function to each received task sequentially, we remark that in a task-farm structure no computation latency improvement is achieved compared to a sequential implementation, instead the ideal service time decreases if we increase the parallelism degree (until the emitter or the collector become bottlenecks).

4.2.2 Data-Parallel performance modeling

More complex w.r.t the task-farm case is the definition of performance models for data-parallel schemes. As we have seen in the example provided in Section 3.2, for data-parallel computations many forms exist and, for each form, some variants are possible, e.g. with or without replicated data, with or without data communications between workers, and so on. In this section we provide a general description of a performance modeling for data-parallel programs, which needs to be instantiated to real cases as it will be exemplified in Chapter 6.

We consider data-parallel programs in which a composite input state (vectors and/or matrices) is partitioned/replicated among a set of workers which apply a function F on each element of its assigned partition for a certain number of iterations. The function evaluation is a sequential computation that can feature statically known data dependencies: for instance the evaluation of F on the i-th element of an array can depend on the values of the nearest neighbors $i-1$ and $i+1$. Such dependencies can vary between different iterations of the a data-parallel program (*variable stencil*) or they can be the same for all iterations (*fixed stencil*). In our model at each iteration i, all data dependencies are

related to the element values computed at the end of the previous iteration $i-1$. At the implementation level, we map state partitions onto workers of a parallel module. At this level dependencies spanning across different state partitions are solved by means explicit asynchronous communications between workers (see Figure 4.16). Next, the data-parallel

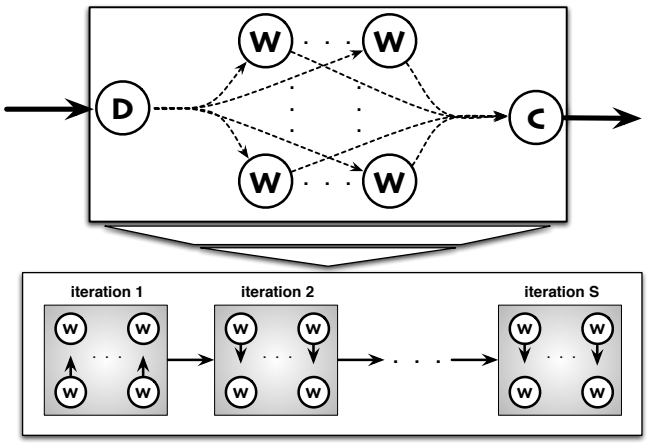


Figure 4.16: Data-Parallel parallelism scheme.

computation starts and the workers continue in iterating it for a fixed number of iterations or until some convergence condition becomes true. In some cases, at the end of the computation, a dedicated process performs the gathering of the local results of each worker, filling an output data-structure.

In the above description emerge three different phases of a data-parallel program: (i) the distribution of the input data-structures; (ii) the execution phase composed of a set of iterations performed by each worker in parallel; (iii) the collection of worker results. The performance model of data-parallel programs is defined in terms of the service time and the computation latency of these three phases. Let us start by studying the computation latency, which indicates the time from the reception of an input task until the corresponding result is produced by the collector process. This time depends on the latency of the three different phases, in general we have:

$$L_{dp} = L_{Distribution} + \sum_{i=1}^{S} T_{iter}(i) + L_{Collection} \tag{4.15}$$

where $L_{Distribution}$ is the computation latency for completing the distribution of the input data-structures of a task, and $L_{Collection}$ is the latency for completing the results collection. The middle term of the equation indicates the computation time of each worker. Data-parallel programs usually iterate a sequential computation for each element of a large input data structure (e.g. a vector or a matrix) in parallel, for a fixed number of iterations or until a convergence condition is satisfied. In expression (4.15) we have considered for simplicity a fixed number of iterations S. For each worker, the computation time per iteration consists of a calculation phase, in which the sequential computation is applied to

all the elements of its partition, and in a communication phase, in which a portion of the local data is transmitted to other workers according to the data-dependencies imposed by the computation semantics. Therefore we have:

$$T_{iter}(i) = T_F \frac{M}{N} + T_{comm}(i)$$

where M and N are the size of the input data structure and the parallelism degree of the computation, T_F indicates the calculation time per element and T_{comm} indicates the communication time required for exchanging data with other workers of the computation. In the case of a variable stencil, the communication time depends on the actual index of the iteration, i.e. the communication pattern (in term of involved processes and amount of data transmitted) can vary at each iteration of the data-parallel program. A notable case is when no communication between workers is required at each iteration: in this case we speak about a *map* data-parallel program and $T_{comm}(i) = 0$ for each iteration i.

If a data-parallel scheme operates in a stream-based computation, its ideal service time is given by the following expression:

$$T_{dp} = \max \left\{ T_{Distribution}, \sum_{i=1}^{S} T_{iter}(i), T_{Collection} \right\} \tag{4.16}$$

Now, the three phases operate in a three-staged pipeline structure: i.e. distribution, calculation and collection phases. Therefore, the ideal service time of the entire program (given by the inter-departure time from the last stage) is the maximum between the service time of each stage separately. Let us note that the service time of distribution and collection phases (i.e. $T_{Distribution}$ and $T_{Collection}$ in the above formula), can be different from their corresponding computation latency (i.e. $L_{Distribution}$ and $L_{Collection}$). Although this is not always true, for instance if the distribution and collection processes are exploited sequentially by a linear set of send and receive operations, this can become possible if we consider parallel implementations of distribution/collection activities (e.g. tree structured schemes with logarithmic latency).

4.3 Summary

This chapter provides performance modeling tools for distributed parallel computations based on basic results on Queueing Theory and Queueing Networks. Our standpoint considers complex graph-based applications in which each module may be internally parallel. Two classes of parallelism have been identified: (i) intra-module parallelism, which is expressed by well-known parallelism schemes as task-farm and data-parallel ones; (ii) modules can cooperate (inter-module parallelism) in computation graphs. In this chapter we have described a unified performance modeling approach: we are able to evaluate the ideal service times of notable parallelism schemes, that can be encapsulated inside parallel application modules, and we have provided tools for evaluating the steady-state

performance behavior of different classes of acyclic and cyclic computation graphs. In the next chapter these results will be a key point for applying control methodologies for the run-time adaptation of parallel applications.

Chapter 5

Definition, Formalization and Control of an Adaptive Parallel Module

THis chapter lays the foundations for the approach to adaptive distributed parallel applications that has been developed in this thesis. The main focus is on the definition of control-theoretic adaptation strategies for controlling a generic adaptive parallel module. In the first part of this chapter the module organization, in terms of a controlled system (i.e. a parallel computation expressed according to the SPP methodology) and a controller (exploiting proper adaptation strategies) will be presented discussing the specific choices and modeling directions that have been taken. In the second part the modeling of the QoS behavior of parallel modules will be described in terms of techniques inherited from Control Theory, discussing how the performance analysis presented in the previous chapter is a starting point for the model construction. The third part focuses on proper adaptation strategies for parallel modules based on control-theoretic foundations. In particular the formulation of reactive and predictive adaptation logics will be presented discussing their feasibility and effectiveness.

5.1 Parallel Module definition

The core element is the concept of adaptive parallel module (i.e. ***ParMod***), an independent and active unit featuring a parallel computation and an adaptation strategy for responding to different causes of dynamicity. The ParMod concept presented in this chapter is inherited from our past research work in the context of the ASSIST programming model [18] and its extension ASSISTANT [27, 28]. This section recalls the basic concepts and focuses on the novel modeling aspects for developing adaptation strategies for ParMod control.

In the sequel we will assume a heterogeneous distributed execution platform composed of several classes of resources:

- shared-memory architectures as symmetric (SMP) or non-symmetric (NUMA) multi-

processors and/or multi-/many-core platforms featuring a set of computing nodes;

- distributed-memory multicomputers, as cluster of workstations.

Following the next-generation grid paradigm [5], such set of computing resources can be interconnected by heterogeneous infrastructures as mobile networks. Therefore we suppose a reference execution environment Env composed of several parallel computing platforms, i.e. $Env = \{Pf_1, Pf_2, \ldots, Pf_g\}$, each one with a collection of homogeneous processing elements. A function $Nodes : Env \rightarrow \mathbb{N}^+$ returns the number of processing nodes for each computing resource available in the execution environment.

5.1.1 Operating Part and Control Part structuring

As described in [28, 61], an adaptive parallel module is structured in two interconnected parts following the closed-loop interaction scheme introduced in Chapter 3:

- **Operating Part**: this part is responsible for performing the so-called *functional logic* of a parallel module, that is a parallel computation expressed according to a certain structured parallelism scheme (e.g. task-farm and data-parallel). Without loss of generality we assume a stream-based computation in which a large set of input tasks from other parallel modules are periodically received by the operating part. The parallel computation of a ParMod can be activated either through a non-deterministic selection of tasks from different source modules, or, based on a data-flow semantics, waiting for the reception of an element from each input interface of the module. If it is necessary, computed results are transmitted onto output data streams (output interfaces) to other parallel modules of the application graph. From a control-theoretic standpoint the operating part represents the observed plant of the closed-loop architecture;

- **Control Part**: this part represents the controller, an autonomous entity able to observe the operating part execution and modify its behavior exploiting reconfiguration activities. The *control logic* that drives the selection of these reconfiguration choices depends on the adopted adaptation strategy.

As stated above, the operating part is in charge of executing the functional logic of the module by exploiting a parallel computation. This behavior can be described in terms of a set of cooperating execution units called in abstract *virtual resources*. They can be processes cooperating by transmitting and receiving messages or threads acceding shared data structures. As we have seen in Chapter 3, structured parallelism schemes can be described in terms of a limited class of distinct parallel functionalities: an *emitter* that receives input streams from the external world (e.g. from other ParMods), a *collector* for results collection and their transmission onto the output streams of the module, and a set of *worker* resources that perform a sequential calculation on the received data according to the implemented parallelization pattern. The control part of a ParMod can conceptually

be composed of a single management entity (i.e. a manager), that can be replicated or distributed for reliability and scalability reasons.

During the execution, the operating part computation can be re-structured by applying run-time reconfiguration activities. Based on the SPP methodology, for each ParMod we suppose the presence of multiple alternative parallel versions of the operating part computation. Such versions can be based on the same sequential algorithm, but parallelized according to different parallelism schemes, or they can feature a different sequential algorithm too. As stated in Section 3.2 the only constraint that we impose is that every versions must respect the same input and output interfaces of the ParMod, in such a way that a local reconfiguration involving a single parallel module does not modify the interfaces provided to other application modules.

In the rest of this thesis instead of the general term version we will use the more precise notion of ParMod ***operation*** [61, 28]. An operation identifies a parallelized sequential algorithm and a parallelism scheme that a ParMod can exploit during the execution. Therefore the operating part definition of a ParMod M can be provided with different operations $M.OP = \{op_1, op_2, \ldots, op_z\}$ compatible in terms of input and output interfaces. *Obviously at each time instant only one operation can be currently in execution*. Different ParMod operations can be suitable or optimized for specific execution architectures on which the computation can be deployed, depending on the energy, processing and memory capability and the impact of a parallel computation on the memory hierarchy of such architectures. This suitability degree can be represented by a function $Platforms : M.OP \rightarrow PowerSet(Env)$, which maps each ParMod operation onto a specific set of suitable computing architectures.

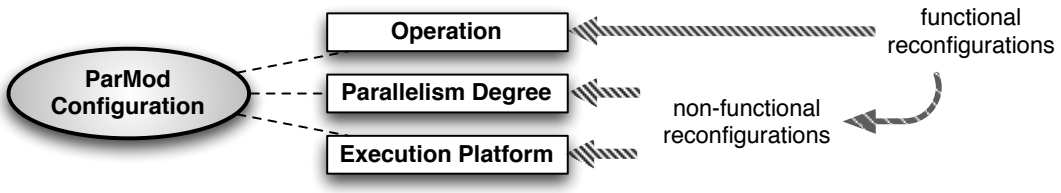


Figure 5.1: Non-functional reconfigurations exclusively modify the parallelism degree and/or the current execution platform of the operating part computation. Functional reconfigurations also change functional aspects as the parallelized sequential algorithm and/or the parallelization scheme.

In other words the operating part execution can be exploited based on multiple *configurations*. For configuration (Figure 5.1) we intend a specific instantiation of all the parameters that characterize the ParMod execution: (i) the current operation (i.e. parallelism scheme and parallelized sequential algorithm), (ii) the actual parallelism degree and (iii) the execution platform. During the execution this configuration can be modified by reconfiguration activities:

- functional reconfigurations consist in changing the current operation performed by

the operating part. Moreover these activities usually involve the modification of the current execution platform on which the computation is currently mapped and the parallelism degree. Hence we require the presence of run-time support mechanisms that: (i) instantiate a properly sized set of virtual resources (i.e. an emitter, a collector and a specified number of workers) on a target execution platform; (ii) operating part virtual resources are able to exploit each operation that has been provided with the ParMod definition;

- non-functional reconfigurations consist in changing the parallelism degree and/or the currently exploited execution platform of a parallel module. W.r.t functional reconfigurations, in this case we are not able to modify the ParMod operation: i.e. these reconfiguration processes do not modify the parallelism scheme and the parallelized sequential algorithm.

Figure 5.2 depicts the ParMod structure. As stated in the previous chapter closed-loop interaction between the plant and the controller is performed through a periodical information exchange based on system monitored data and reconfiguration commands:

- *Monitored data*: for monitored data we intend all the interesting measurements that describe the operating part computation in terms of qualitative or quantitative metrics. For instance QoS metrics as memory occupation, energy consumption, the mean throughput or the current computation latency represent typical information periodically transmitted to the control part;
- *Reconfiguration commands*: they are proper actions that modify the current operating part configuration exploiting non-functional or functional reconfiguration processes.

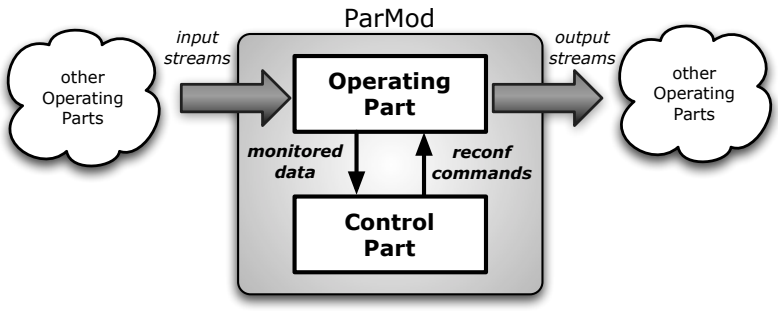


Figure 5.2: Operating Part and Control Part structuring of a ParMod.

In the literature controllers are classified into several families according to their temporal evolution and the frequency of the control decisions:

- **Time-driven Controllers** (synchronous controllers). In this approach adaptation strategy evaluation is performed periodically, at equally spaced time points. The

time between two subsequent decision points represents the concept of **control step**; in other words a time-driven controller takes control decisions every step. These controllers are broadly used in digital systems [86], where continuous time signal from a controlled process are represented by their sampled values at a fixed sampling rate (e.g. by an Analog-to-Digital converter);

- **Event-driven Controllers** (asynchronous controllers). On the opposite direction, in an event-driven controller it is the occurrence of an event rather than passing of time that decides when the next sampled value is produced. In this case adaptation strategy is evaluated at unexpected and not equally spaced time instants.

The main advantage of asynchronous controllers is the possibility to reduce the resource utilization of the controlling task: in fact the adaptation strategy evaluation is performed only when it is strictly necessary (e.g. when a certain critical event occurs). Nevertheless application of event-driven controllers is hampered by a lack of a system theory, which is due to the fact that a formal analysis is much more complicated than for time-driven controllers (see [87] about this point). *For this reason the adaptation strategies introduced in this thesis will be based on time-driven controllers, that perform a time discretization based on the control step concept*. This is a fundamental difference with the previous ParMod definitions described in [13] and in [61].

According to the previous considerations the ParMod control model is based on a time-driven controller, the control part, that takes reconfiguration decisions every control step of duration τ. At the beginning of each step the control part obtains updated monitored data from the operating part, and executes a specific control algorithm in such a way as to decide a corresponding set of reconfiguration commands that will be communicated to the operating part. These inputs identify the new ParMod configuration that the operating part should use for the entire duration of the current control step. After that, the control-loop interaction will be repeated at the beginning of the next control interval. The pseudo-code of this synchronous behavior is described in the algorithm 2.

Algorithm 2: Control Part synchronous behavior.

```
1 begin
2     foreach control step τ do
3         Acquisition of monitored data from the Operating Part;
4         Evaluation of the adaptation strategy for deciding the new ParMod
          configuration;
5         Transmission of corresponding reconfiguration commands to the Operating
          Part;
```

We remark that control part decides at the beginning of each control step the current ParMod configuration (i.e. operation, execution platform and parallelism degree) that will be used throughout the current step. Hence, we can observe that in this control model

it is possible to change the current ParMod configuration only at the beginning of each control step of the execution. For this reason the control step length is a critical parameter. Shorter steps make it possible a fast response of the ParMod to unexpected changes in its execution quality but, on the other hand, a high-frequency evaluation of the control algorithm could be extremely resource expensive and useless. Longer control steps render the resource utilization more limited, but at the cost of larger-granularity reconfiguration decisions that could be ineffective for responding to fast system dynamics.

5.1.2 General interaction scheme and reconfiguration metrics

In this section we will present a general interaction scheme between operating and control part of a ParMod. The following description will be completely independent w.r.t the possible forms of cooperation between operating part and control part (e.g. by using message passing or shared variables), which will be discussed in Section 5.1.3.

As stated in the previous section, operating part and control part exchange monitored data and reconfiguration commands at each step. In Figure 5.3 is depicted the interaction pattern for a specific step.

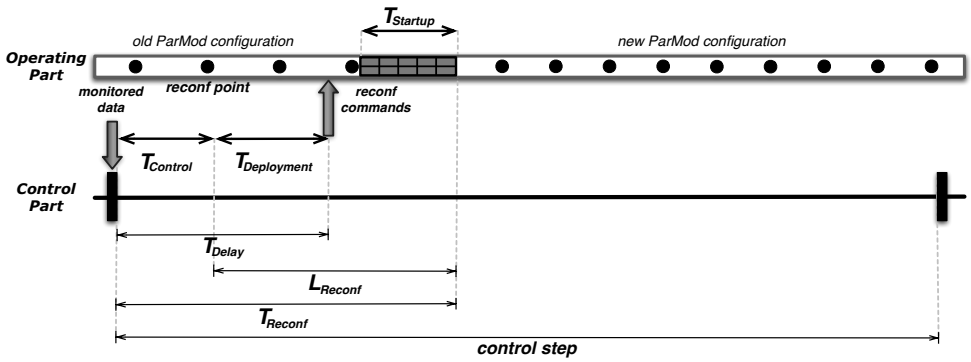


Figure 5.3: Abstract interaction scheme between Operating Part and Control Part.

This interaction scheme will be characterized by the following interesting features, that will be discussed in this section:

- ***Synchronous execution of the control logic***: inheriting from a classical control-theoretic design, control part evaluates the adaptation strategy at the beginning of each sampling interval. This step notion is only inside the controller, without any impact on the parallel computation of the operating part (which is completely asynchronous w.r.t the control logic);

- ***Control logic overlapping***: this scheme tries to generate a minimum overhead on the operating part execution, by overlapping several control phases with the computation (e.g. the control logic execution and deployment activities).

Let us consider this interaction in more detail. First of all control part acquires monitored data from the operating part according to the specific way in which cooperation is exploited (e.g. for instance by reading shared data structures in which updated QoS parameters are periodically stored). At this point control part evaluates its adaptation strategy for selecting the new ParMod configuration that will be used during the current control step. We use the term $T_{Control}$ for indicating the mean execution time of the control algorithm (see Figure 5.3). The result is the identification of a new ParMod configuration which may consist of: (i) the operation that will be used; (ii) a corresponding execution platform; (iii) a new parallelism degree. A set of run-time support activities should be executed by the control part for preparing the utilization of the new ParMod configuration. These actions can be *deployment activities* involving the instantiation of a set of worker resources on the current platform (e.g. in the case of a parallelism degree variation), or the instantiation of the entire virtual resource set on a new target architecture (e.g. if a platform change is required). We denote with $T_{Deployment}$ the average time spent on this phase. On the other hand, if no reconfiguration is executed at that control step (i.e. the old and the new ParMod configuration coincide), no deployment activity is necessary thus $T_{Deployment} = 0$.

After the deployment actions, reconfiguration commands are transmitted from the control part to the operating part in order to notify the parallel computation that a reconfiguration must be completed. We can observe that the operating part elaboration evolves asynchronously w.r.t the control part synchronous behavior (Algorithm 2). Reconfiguration commands can be received and processed by the operating part virtual resources at certain *reconfiguration points* during their elaborations. These points are specific computing instants during the operating part elaboration flows in which the virtual resources test if the control part has issued a new set of reconfiguration commands. In this section we limit our description by considering two general classes of reconfiguration points:

- in a stream-based computation a straightforward way to define reconfiguration points is at each activation of operating part execution. This means that the emitter virtual resource is able to check the presence of reconfiguration commands while it is waiting for a successive (non-deterministic or data-flow) activation from its input streams;

- the previous reconfiguration point concept is not applicable in parallel computations with a single or very limited set of input elements. In this situation further points need to be defined. An approach described in [60, 61] considers programmer-transparent points as consistency lines obtained by means of coordination protocols and synchronizations between virtual resources (e.g. at the end of each iteration of a data-parallel program). Such finer points can also be useful for stream-based computations, in order to reduce the average time between two successive reconfiguration points and thus the responsiveness of the operating part to receive reconfiguration commands.

After the reception of reconfiguration commands, the operating part must complete

the decided reconfiguration activities. At this point a main issue consists in assuring that the computation semantics is not violated during this phase. Consider the following examples:

- let us suppose that we exploit a non-functional reconfiguration by increasing the parallelism degree of a ParMod, thus without modifying its currently executed operation. For state-less parallelizations, as a task-farm structure, this reconfiguration can be performed in a very efficient way [13] by linking the new instantiated workers with the emitter and the collector. This is not the same case if we consider state-ful parallel computations, in which an internal state has to be maintained during successive elaborations on different input stream elements. In this case a parallelism degree variation requires the execution of specific protocols between virtual resources in order to consistently maintain and distribute the internal state;

- let us suppose that we perform a functional reconfiguration activity: e.g. we decide to modify the ParMod operation changing the parallelism scheme and the platform on which the computation is executed. In this case the de-activation of ParMod virtual resources on the old platform and the activation of the new instances on the target architecture could require to preserve specific computation properties: e.g. without loosing input stream element computations or, more strictly, preserving the ordering of input element elaborations in the case of a state-ful processing. In both cases virtual resources of the operating part must cooperate according to specific reconfiguration protocols for preserving the computation semantics (see [88]).

Therefore proper *reconfiguration protocols* have to assure that the ParMod semantics is not violated during a reconfiguration phase. Their specific description in terms of well-known fault-tolerance methodologies as roll-back and roll-forward techniques has been studied in our previous research work [88, 60] on adaptivity for structured parallel computations. In this thesis we do not focus on their definition and optimization, since we are more interested in describing and formalizing control-theoretic adaptation strategies. Therefore, for the time being, we will abstract reconfiguration protocols details by consider their mean completion time with the term $T_{Startup}$ (see Figure 5.3).

We can introduce some relevant metrics that provide useful indicators of the reconfiguration impact on the operating part execution:

- **Total reconfiguration time**: it is the average time T_{Reconf} from completing the reconfiguration process;

- **Reconfiguration Latency**: after the adaptation strategy evaluation a new ParMod configuration has been decided. For reconfiguration latency L_{Reconf} we intend the average time required to instantiate the new configuration after the control decision has been taken;

- **Control Delay**: in an ideal model the controller is able to decide and transmit reconfiguration commands to the system instantaneously at the beginning of each

control step. This is not possible in a real model in which the decision process and the transmission phase require a non-null time. In our interaction model the time spent from the control part to decide the new configuration and to eventually complete deployment activities represents a control delay: i.e. the average time T_{Delay} from the beginning of the control step until the set of reconfiguration commands are ready to be transmitted to the operating part.

We can express the values of these measurements in terms of delays $T_{Control}$, $T_{Deployment}$ e $T_{Startup}$ introduced above.

$$\begin{aligned} T_{Delay} &= T_{Control} + T_{Deployment} \\ L_{Reconf} &\simeq T_{Deployment} + T_{Startup} \\ T_{Reconf} &\simeq T_{Delay} + T_{Startup} = T_{Control} + L_{Reconf} \end{aligned} \tag{5.1}$$

The total reconfiguration time can have a notable impact for properly sizing the length of a control step for our applications. As stated above a ParMod can change its configuration only once per control step. Thus the step duration must be sized in such a way as to complete any possible reconfiguration activity until the end of the step.

Proposition 5.1.1 (Correctness of control step length)**.** *For a correct interaction between operating and control part of a ParMod, the length of the control step should be fixed to a specific value τ such that $\tau \geq T^{max}_{Reconf}$, where T^{max}_{Reconf} is the maximum time spent for completing a ParMod reconfiguration in the worst case.*

In the majority of cases a formal assurance of the previous property is not feasible. In fact it is very difficult or even impossible to have an exact assurance about the T^{max}_{Reconf} parameter and its estimation may depend on several factors especially in a distributed environment. Therefore we will assume a more flexible interpretation of the interaction scheme in which the correct behavior is exploited in most cases, and the run-time support is able to detect possible erroneous interactions (e.g. a new set of reconfiguration commands can be transmitted to the operating part only if the previous reconfiguration activities have been fully completed, also if in some cases they can span between successive control steps).

We conclude this section by considering another property of this interaction model: it is possible a partial overlapping between the total reconfiguration time and the "normal" execution of the operating part computation. For instance we can observe from Figure 5.3 that $T_{Control}$ and $T_{Deployment}$ delays are always overlapped with the execution of the old ParMod configuration. Control part transmits reconfiguration commands only when: (i) a new ParMod configuration has been decided, and (2) after the completion of all the necessary deployment actions. Thus, $T_{Startup}$ is a very meaningful parameter because it is not overlapped with the ParMod computation and may represent a potential source of performance degradation of the operating part execution. Moreover, if the adaptation strategy is

effective, we expect that in many control steps no reconfiguration will be exploited. This means that the current configuration of the ParMod is still a ”good” choice for attaining the control goals. In this case the overhead induced by this control model is very limited: i.e. $T_{Control}$ is completely overlapped with the operating part computation and we have no deployment and reconfiguration protocol overhead.

5.1.3 Issues of Run-time support implementation

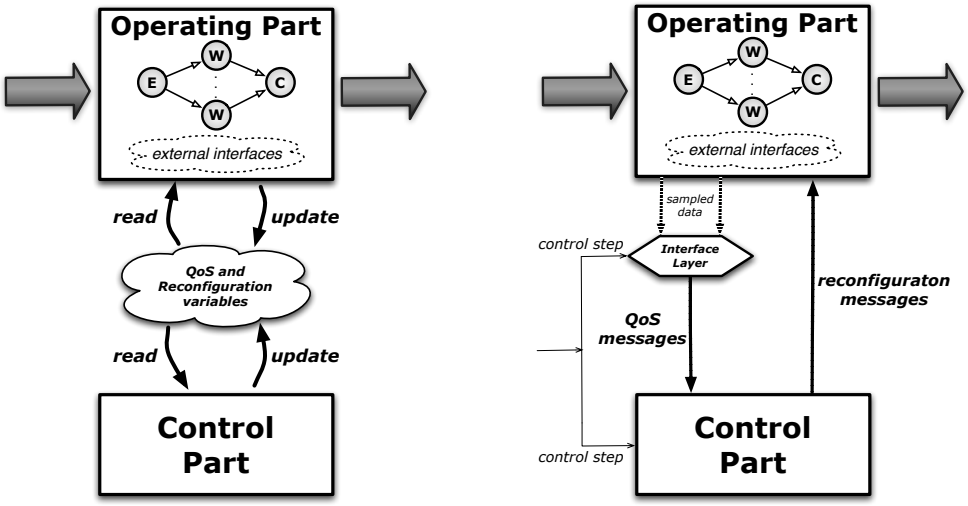
In order to conclude the general description of the ParMod concept, in this section we will introduce some considerations about the way that operating part and control part exploit for interacting with each other. As it is known, concurrent/parallel activities can adopt two different approaches to cooperation. A first approach consists in providing a message-passing, also called *local environment* cooperation, in which cooperation is exploited by receiving and transmitting messages onto proper communication channels. On the opposite direction the other solution exploits a shared-memory model, also called *global environment* cooperation, in which concurrent entities share a set of variables that represent a global state of the computation, and cooperation is exploited by consistently modifying the values of these shared data structures.

As stated above the ParMod interaction scheme comprises two different information exchanges:

- control part must be able to access an updated view of operating part monitored data;
- operating part must be able to receive reconfiguration commands from the control part, which identify the new ParMod configuration that will be used.

The implementation of these information exchanges can be exploited according to the two different cooperation approaches. First of all let us consider a global environment implementation of the ParMod run-time support (see Figure 5.4a). In this case a set of *global variables* shared by operating part and control part virtual resources is defined:

- **QoS variables** describe the actual values of monitored data of the operating part execution. They can be updated by the operating part in two possible ways:
  1. some of them are directly updated by the operating part virtual resources. This is the case of variables describing QoS information strictly concerning the actual behavior of the parallel computation, notably its performance (e.g. in a stream-based computation the emitter virtual resource can update a variable describing the number of received tasks while the collector can keep track of the number of produced results). In this situation, the operating part virtual resources directly perform *internal monitoring* activities;
  2. other variables, albeit they also represent QoS metrics of the computation, are more difficult to be updated directly by operating part virtual resources. This



(a) Shared-Variables implementation. (b) Message-Passing implementation.

Figure 5.4: Alternative run-time support implementations of a ParMod.

is the case of resource availability information as the actual energy level of the current execution platform (notably a mobile node) or the current memory occupation. In this case such information can be generated by *external monitoring* entities called ***external interfaces***, which are conceptually part of the ParMod operating part but they are dedicated to the exploitation of system resource analysis and monitoring activities;

- **Reconfiguration variables** correspond to the actual reconfiguration commands issued to the operating part. As an example the values of these variables identify the actual data-distribution strategy that the emitter should exploit (e.g. an on-demand distribution or a scatter functionality), and the version of the sequential code that will be executed by the worker resources. If, due to a reconfiguration process, the entire set of operating part virtual resources should be executed on a different execution platform, the deployment phase is responsible for correctly and consistently migrating the shared reconfiguration variables on the new target architecture, and properly initialize them with the information for performing the new ParMod operation with the new parallelism degree.

At the beginning of each control step the control part reads the actual values of QoS variables while the operating part virtual resources access the set of reconfiguration variables at each reconfiguration point.

An alternative design of a ParMod adopts a message-passing cooperation between operating part and control part (as depicted in Figure 5.4b). In this approach at each step, reconfiguration commands are transmitted as messages onto specific reconfiguration channels from control part to operating part virtual resources. More interesting is the cooperation for monitored data. In this case the control part expects to receive updated

monitored data synchronously at the beginning of each control step. For this reason we introduce the presence of an ***interface layer***: an abstract entity (concretely implemented through a proper set of dedicated processes or threads) able to collect observed measurements (directly from the operating part virtual resources or rather generated by external interfaces) for the whole duration of each control step, and to transmit their sampled values as QoS messages at the beginning of each control step.

In the rest of this chapter we will not provide other implementation aspects concerning the ParMod run-time support, but we will focus on adaptation strategies and their properties for controlling the parallel computation of a single module. Other implementation details will be given in Chapter 6.

5.2 Formal modeling of the Operating Part behavior

As we have seen in Chapter 3, in Control Theory methodologies a controller exploits an internal model able to predict the future QoS evolution of the observed system. One of the most widespread approach to system modeling consists in introducing a set of mathematical equations that approximate the true behavior of the system. To carry out this process we introduce a set of variables of the system model. The actual QoS of a parallel computation, denoted by performance, power and memory measurements, can be described by *observed outputs* of the model. Other variables have a completely different meaning: they represent *control inputs* that can be varied over time. The final goal of a system model is to determine a mathematical relationship between these observed outputs and control inputs.

As we have hinted in the Chapter 3, we remark that there not exists a unique method for determining the input-output relationship of a model, but this critical task is influenced by the specific objectives and the degree of precision required. For instance first-principle models are based on a deep knowledge of the system behavior such that physical laws or ad-hoc relations can be specifically derived for the target system. On the opposite direction are empirical models, which assume a limited (gray-box) or fully absent (black-box) understanding of the system internal behavior, and the input-output relationship is extracted by applying statistical techniques over the results obtained from a significant and properly designed set of experiments.

Many classifications exist for comparing different modeling techniques. As said statistical and physical models are a first classification which highlights the methodology applied for extracting the input-output equations. Another distinction concerns how past system observations are related to future measurements. If future system outputs only depend on the current control inputs and they are completely independent of their past values, in this case we speak about a ***static model***, which is expressible through algebraic equations. On the other hand, if the outputs of a system also depend on their past history, we speak about a ***dynamical model***. In the case of dynamical models the input-output relationship (also called the system *evolution rule*) can be expressed either as a set of differential equations or as a set of finite-difference equations, depending if the time domain

representation is continuous or discrete. Moreover, dynamical models can be enhanced by expressing a notion of *internal state*, that describes the actual condition of the system at a given time. In this case we speak about a dynamical model in a *state-space* form.

In our approach the existence of a system model is one of the most important preconditions for applying advanced strategies for controlling the adaptive behavior of a parallel module. *We claim that the exploitation of the structured parallel programming methodology plays a decisive role in order to establish static or dynamical models of a parallel module, whose expressions governing the QoS evolution can be expressed exploiting first-principle laws*. Relevant examples are:

- the predictability of steady-state performance measurements of a parallel computation, as the mean service time, the mean queue length and the computation latency, can be predicted and formally analyzed according to analytical models based on queueing theory and queueing networks. At this regard *the concepts and the procedures exposed in Chapter 4 constitute a central point for developing ParMod adaptation strategies*;

- memory utilization models can also be derived for well-known parallelism schemes by exploiting the well-defined behavior of parallelism patterns in terms of function and data replication or partitioning. A first attempt in this sense has been given in [89];

- structured parallelism schemes have a precise semantics in terms of computations performed by each parallel unit (e.g. emitter, worker, collector), the size of exchanged messages and the frequency of activities as calculation and message transmission. Such knowledge can be a starting point in order to define models for measuring the power consumption and the resource utilization cost of a parallel module, especially when its execution is mapped onto limited resources as mobile devices.

In the previous chapter we have focused on queueing theory and queueing networks for predicting the steady-state performance behavior of computation graphs and structured parallelism schemes. In order to apply them we focus on a basic issue: how can we can quantify the effects of reconfigurations on observed QoS variables of a ParMod? What kind of transitions characterize the evolution of these variables? This section answers to these questions by providing a formal modeling of the ParMod operating part.

Let us suppose to have a mathematical model $\mathscr{L}$ that describes the relation between a performance parameter as the mean service time T_S of a parallel computation in function of input parameters as the mean calculation time T_{Calc} of a certain sequential function and the average time L_{com} spent in communications (i.e. the *communication latency*) between cooperating parallel units (e.g. processes).

$$T_S = \mathscr{L}(T_{Calc}, L_{com}, \ldots)$$

Of course such model can be extended to other parameters but, in general, it expresses the relationship between inputs and outputs taking values in a continuous domain (i.e. the

positive reals in this case). Its mathematical formulation can depend on several factors as the currently active operation of the ParMod, the parallelism degree and the execution platform on which the computation is actually executed. These configuration parameters affect the model definition by changing some of its constant terms (e.g. the parallelism degree and architecture-dependent parameters) or the complete mathematical structure of the model can be modified (e.g. when we change the parallelism scheme). For these reasons we can view the operating part as a multi-modal system.

Definition 5.2.1. (Multi-modal behavior of ParMod Operating Part). At each point of time the operating part can behave according to a certain active configuration belonging to a specific set C of alternatives:

$$C = \{C_0, C_1, \ldots, C_{\nu-1}\} \tag{5.2}$$

This set represents a ***finite and discrete set of statically known alternative configurations*** (C has a finite cardinality ν), corresponding to a precise choice of: ParMod operation (i.e. parallelism scheme and parallelized sequential algorithm), parallelism degree (i.e. number of worker resources) and execution platform. For each of these configurations the dynamics of QoS measurements are governed by a specific mathematical model.

According to this multi-modal structure we can identify two classes of transitions that characterized the system execution:

- *continuous transitions*: when a configuration has been fixed, the evolution of continuous QoS parameters of the ParMod execution can be predicted by applying a specific model corresponding to the currently used configuration;
- *discrete transitions*: due to the adaptive behavior of a ParMod, the current active configuration can be changed passing from a configuration C_i to a different alternative configuration C_j, shortly $C_i \rightarrow C_j$ with $i \neq j$.

The presence of continuous transitions (of continuous variables) and discrete transitions (of alternative configurations) suggests the possibility to abstractly model the operating part as a particular class of hybrid systems in which these two dynamics are formally modeled in a unique mathematical structure.

5.2.1 Hybrid model of ParMod Operating Part

Hybrid systems [90] are a class of systems that expose a heterogeneous nature, where their degree of heterogeneity refers to a proper combination and composition of continuous and discrete parts that are formalized using a unique formal model. The concept of discrete and continuous parts has a different meaning in the scientific literature:

- often continuous and discrete terms refer to the time-domain. In this case hybrid systems are systems with continuous time dynamics (modeled by differential equations) and a discrete logic (modeled with if-then-else rules, finite-state machines

or propositional and temporal logics). Model state variables evolve continuously following the rules of a set of differential equations. When certain conditions concerning the state variables are fired, they jump to specific values in response to discrete transitions;

- in other modeling approaches continuous and discrete terms refer to the domain in which model variables take their values, where the time domain can be either continuous or discrete. In this case a hybrid model consists in a set of differential or difference equations in which some of the model variables take continuous values (e.g. real numbers) whereas other variables are intrinsically discrete (e.g. a finite set of discrete values).

For modeling the operating part behavior as a hybrid system we adopt the second view in which the hybrid nature of a ParMod is due to the presence of variables featuring both continuous and discrete domains, whereas the ParMod time evolution is fixed to be discrete assuming a discretization based on the control step concept. With this assumption the operating part model is expressed as a set of equations describing the evolution of the observed measurements. The temporal interpretation of the control step concept is shown in Figure 5.5. Each control step has a fixed duration τ and it is uniquely identified

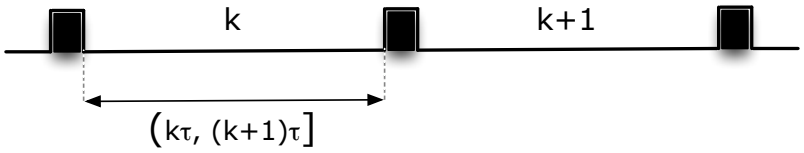


Figure 5.5: Temporal interpretation of a control step.

with a sequence identifier $k = 0, 1, \ldots, N-1$. Control step k refers to the time interval $(k\tau, (k+1)\tau]$.

According to the previous distinction a hybrid model uses variables that are mixed signals, consisting in combinations of continuous or discrete variables that take their values at equally spaced time intervals. Each signal can be a sequence of single continuous values (i.e. *scalar signals*) or, more useful, a sequence of n-component vectors (i.e. *n-vector signals*) composed of n variables. In our model description we consider the following sets of signals:

Internal state s-vector signal: we refer with the term $\mathbf{x}(k)$[1] as the current value of the internal state of the operating part model at the beginning of the control step k. This state vector is composed of s components:

$$\mathbf{x}(\mathbf{k}) = \begin{bmatrix} x_1(k) \\ x_2(k) \\ \vdots \\ x_s(k) \end{bmatrix}$$

[1]Hereafter we use a lowercase bold letter to indicate a vector.

with $\mathbf{x}(k) \in \mathbb{R}^s$. State variables represent system information that is useful to maintain during successive control steps. As an example the number of actually queued tasks and the total number of completed input stream elements. State variables are especially important in dynamical models.

Observed output n-vector signal: the term $\mathbf{y}(k)$ indicates the value assumed by the model outputs at the beginning of the control step k. The output vector is composed of n components:

$$\mathbf{y}(\mathbf{k}) = \begin{bmatrix} y_1(k) \\ y_2(k) \\ \vdots \\ y_n(k) \end{bmatrix}$$

with $\mathbf{y}(k) \in \mathbb{R}^n$. As we will see in the following part of this section, outputs can be defined as proper transformations of state variables or, in particular cases, they can also be directly the values of some state variables or even the whole state. In this case we refer to a *completely observable state* of the model.

Disturbance input m-vector signal: a very important notion for modeling the behavior of complex systems is the concept of ***disturbance***. Disturbance inputs are uncontrolled exogenous signals that can affect the relationship between control inputs and the observed plant outputs. Uncontrolled means that the controller (i.e. the control part in our case) is not able to decide and fix their values, which are instead determined by environmental or external decisions outside of ParMod control. Disturbance inputs can be classified as *measured* or *unmeasured*. For measured disturbances the way in which they can affect the system observed outputs is known. On the other hand unmeasured disturbances are completely unknown to the controller, which is not able to quantify their exact values and their effects. Based on this distinction in our model we assume the presence of measured disturbances which are identified by a m-vector signal $\mathbf{d}(k)$ that indicates the values of disturbances throughout the k-th control step (e.g. in many cases we are interested in their average values). The vector has m components:

$$\mathbf{d}(\mathbf{k}) = \begin{bmatrix} d_1(k) \\ d_2(k) \\ \vdots \\ d_m(k) \end{bmatrix}$$

with $\mathbf{d}(k) \in \mathbb{R}^m$. For our purposes typical disturbances will be: the mean calculation time of a sequential function executed on a target computing architecture, the communication latency and the mean inter-arrival time of requests to the system. These parameters can be highly variable due to platform conditions (e.g. the current state of the communication networks and if the computing resources are dedicated

or shared with other applications) or due to application-dependent conditions (e.g. a sudden increase in the size of received tasks).

Depending on the particular class of system model that we are defining, some sets of variables can be omitted. For instance in a dynamical model state variables play a central role. In practical cases the entire state is often observable by the control part, thus observed outputs coincide with state variables. Otherwise, in static system models, in which we express a direct relationship between inputs and outputs, state variables can be omitted. In the sequel, unless otherwise noted, the notion of system model will correspond to the most general meaning of a dynamical model with state, output, input and disturbance variables. Particular cases in which we consider a completely observable state or rather static models will be highlighted when necessary.

For each configuration $C_i \in C$ of the operating part the temporal evolution of observed outputs and of the next state variables can be expressed by a model in a state-space form as shown in (5.3).

$$\begin{cases} \mathbf{x}(k+1) = \phi_i\big(\mathbf{x}(k), \mathbf{d}(k)\big) \\ \mathbf{y}(k) = H\big(\mathbf{x}(k)\big) \end{cases} \tag{5.3}$$

This state-space modeling comprises two finite difference equations that describe the state and output dynamics. The next state expression applies a function ϕ_i that calculates the values assumed by state variables at the beginning of the next control step, in function of the actual value of the state, disturbance and control inputs. For each configuration, *the definition of this function is based on the exploitation of performance models of structured parallel computations*. This aspect will be investigated in more detail in the next chapter, with the description of real-world experiments in which these concepts will be rigorously applied. Observed outputs at the beginning of step k are obtained by applying a generic transformation on current state variables at the same step. If the state is fully observable the function H is the identity and observed outputs coincide with the current model state.

Two considerations are important at this point. First of all functions ϕ_i and H can be obtained by applying statistical methods or by using a higher level knowledge of the operating part behavior to extract proper first-principle relations. The second point is that in this modeling, observed outputs are directly obtained as functions of the current state without any relation with the actual operating part configuration (i.e. we only have a unique function H). On the other hand, for next state expression, the function ϕ_i is strictly coupled with a specific operating part configuration. Therefore, since operating part features a multi-modal behavior (see Definition 5.2.1), we have multiple models $\phi_0, \phi_2, \ldots, \phi_{\nu-1}$ that describe the internal state evolution according the the operating part configuration which is currently active.

This variable-structure behavior is typical of a large class of hybrid systems featuring a limited set of alternative operating modes. Such class consists in the so-called ***Switched Hybrid Systems*** [91] described by the following general discrete-time model:

$$\mathbf{x}(k+1) = \phi_i\left(\mathbf{x}(k)\ldots\right) \; i = 0, 1, \ldots, \nu - 1$$

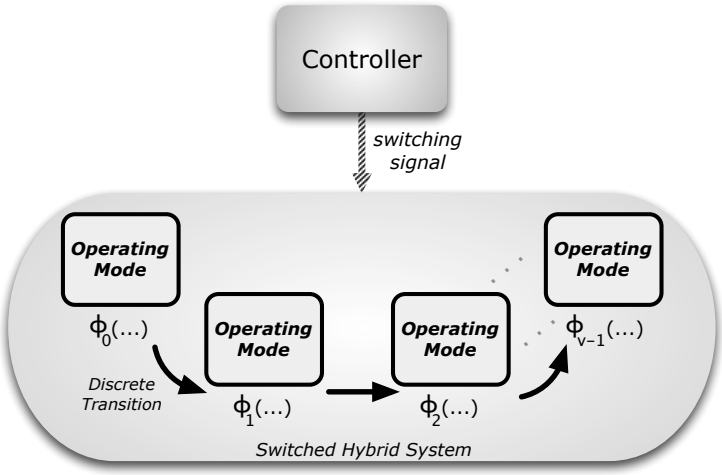


Figure 5.6: Switched Hybrid System with controlled switching law.

The system model for each different operating mode is expressed by a set of finite difference equations of continuous input, output and internal state values. The current operating mode can be selected according to a specific *switching law* that may follow two different approaches:

- ***autonomous switching***: in this approach the switched hybrid system may autonomously decide to switch to a different operating mode. We can distinguish between two different switching techniques: *state-dependent switching* and *time-dependent switching*. The first one is the phenomenon where the vector field f_i changes abruptly when the continuous internal state $\mathbf{x}(k)$ hits certain pre-defined boundaries. This switching law assumes a partitioning of the state-space in a finite number of *operating regions*, each one corresponding to a different model. Therefore a switching of the current operating mode is autonomously exploited by the system whenever its internal state changes the operating region in which it is localized. The time-dependent switching law provides a temporal evolution of the system which triggers the operating mode transitions at pre-defined controls steps. A switching sequence can be defined as:

  $$\Gamma = \Big\{(i_0,0),(i_1,1),\ldots,(i_{N-1},N-1)|i_n \in \{0,1,\ldots,\nu-1\},n=0,1,\ldots,N-1\Big\}$$

  where each pair identifies an operating mode and a corresponding control step. Hence we have a mapping between control steps and corresponding system configurations;

- ***controlled switching***: the operating mode at any control step is chosen by an external entity, i.e. a controller (see Figure 5.6).

On the basis of the previous classification *we will model the behavior of the ParMod operating part as a switched hybrid system with controlled switching law*. Therefore it

is the control part which is responsible for deciding at the beginning of each control step the current configuration that should be used for the whole duration of the current control step. For this reason we introduce an **endogenous control input signal** $\mathbf{u}(k)$ to our hybrid model. The main difference w.r.t the other previously introduced signals, is that control inputs assume a finite and discrete number of possible values that correspond with the operating part alternative configurations. Let us consider the following vector:

$$\mathbf{u}(\mathbf{k}) = \begin{bmatrix} op(k) \\ n(k) \\ p(k) \end{bmatrix}$$

The vector components fully identify the operating part configuration for the whole duration of control step k: i.e. the current operation $op(k)$, parallelism degree $n(k)$ and execution platform $p(k)$. The set of the admissible control input vectors is defined as follows[2]:

$$\mathbf{U} = \left\{ \mathbf{u}(k) \middle| op(k) \in M.OP \wedge p(k) \in Platforms\big(op(k)\big) \wedge 1 \leq n(k) \leq Nodes\big(p(k)\big) \right\} \tag{5.4}$$

The constraints are: (i) $op(k)$ corresponds to an existing operation identifier of the ParMod M; (ii) $p(k)$ identifies an execution platform on which the selected operation is executable; (iii) $n(k)$ is limited to the number of platform execution nodes. We can note that the set of admissible control inputs $\mathbf{U}$ is isomorphic to set of possible operating part configurations C, that is we can introduce a biunivocal function $\pi : \mathbf{U} \rightarrow C$ that maps each possible admissible control input vector onto a specific configuration identifier.

Let us consider now the finite difference equations of the operating part state-space model. As stated above the next state expression is coupled with the current configuration. Hence a global hybrid model of the operating part behavior can be expressed as shown in (5.5):

$$\begin{cases} \mathbf{x}(k+1) = \Phi\big(\mathbf{x}(k), \mathbf{d}(k), \mathbf{u}(k)\big) \\ \mathbf{y}(k) = H\big(\mathbf{x}(k)\big) \end{cases} \tag{5.5}$$

in which function Φ is a piecewise-defined function whose definition is the following:

$$\begin{aligned} &\mathbf{x}(k+1) = \text{ if } \Big(\pi\big(\mathbf{u}(k)\big) = C_i\Big) \text{ then } \phi_i\big(\mathbf{x}(k), \mathbf{d}(k)\big) \\ &i = 0, 1, \ldots, \nu - 1 \end{aligned}$$

At this point we can provide the complete description of operating part in terms of a hybrid model.

Definition 5.2.2 (Operating Part Model)**.** From an abstract point of view the operating part behavior is modeled as a switched hybrid system with controlled switching law defined by the tuple $\langle \mathbf{U}, \mathbf{X}, \mathbf{D}, \mathbf{Y}, \Phi, H \rangle$ where: $\mathbf{U}$ is the finite and discrete set of admissible control inputs corresponding to the possible operating part configurations, $\mathbf{X} \subseteq \mathbb{R}^s$ is the

[2]We use a bold uppercase letter to indicate a set of vectors.

continuous-valued space of the internal states, $\mathbf{D} \subseteq \mathbb{R}^m$ is the continuous-valued set of disturbance inputs and $\mathbf{Y} \subseteq \mathbb{R}^n$ is the continuous-valued set of observed outputs. The model provides two functions: $\Phi : \mathbf{U} \times \mathbf{X} \times \mathbf{D} \to \mathbb{R}^n$ maps a specific discrete-time model $\mathbf{x}(k+1) = \phi_i(\mathbf{x}(k), \mathbf{d}(k))$ onto a configuration $C_i \in C$ such that $\pi(\mathbf{u}(k)) = C_i$; $H : \mathbf{X} \to \mathbb{R}^n$ maps each internal state vector onto a corresponding observed output vector.

We conclude the operating part modeling with a final consideration. A typical and usually undesired property that can be featured by hybrid systems is the so-called *Zeno behavior* [92]. A Zeno behavior occurs when there are an infinite number of discrete transitions in a finite amount of time. Detecting the presence of this situation can be important in many practical cases, thus sufficient and necessary conditions [93] for the existence of this phenomenon are provided for several classes of hybrid systems. We can observe that this behavior is completely avoided in our operating part model by construction: i.e. the operating part is able to perform at most one configuration switching per control step.

5.3 Control Part strategies for ParMod dynamic adaptation

In this last part of the chapter we will introduce the problem of defining different adaptation strategies for a parallel module. We are interested in studying two different classes of control techniques:

- ***Reactive Control Strategy***: the ParMod control part decides the set of reconfiguration commands evaluating the current monitored data received from the operating part at the beginning of each control step. The main idea of this approach is based on a specific assumption: the set of ParMod reconfigurations are decided on the basis of the actual results of the module monitoring phase, hoping that the decisions that have been taken at the current time will be effective also for future execution conditions;

- ***Optimal and Predictive Strategy***: an alternative and more control-oriented approach provides the control part with the possibility to define a trajectory of reconfiguration decisions for a contiguous sequence of control steps, in such a way as to optimize a properly defined cost function or performance index of the system. W.r.t a reactive approach, this solution is aimed at providing a more solid solution to ParMod control, which tries to give sufficient assurances of the adaptation process optimality, the stability degree of a ParMod configuration and the achievement of the required execution constraints.

In the rest of this section we will introduce these two adaptation strategies in more detail.

5.3.1 An adaptation strategy based on a Reactive approach

The utilization of classical control-theoretic techniques as feedback controllers are not feasible for controlling a parallel module that, as we have seen, exhibits a hybrid behavior and is uniquely controlled by deciding the values of a certain set of discrete control inputs. The reactive control technique that we propose in this section is inspired to the theory of *reactive systems* [94] and particularly of ***Discrete Event Controllers*** [95]. Reactive systems are discrete processes which continuously react on conditions that can be satisfied at arbitrary points in time. Such conditions can refer to the system internal behavior (e.g. by reflexive information) or to external environmental observations. One of the most important features is that control decisions (i.e. reconfigurations) are engaged only as needed, on a "just-in-time" basis rather than pro-actively and in advance of critical events.

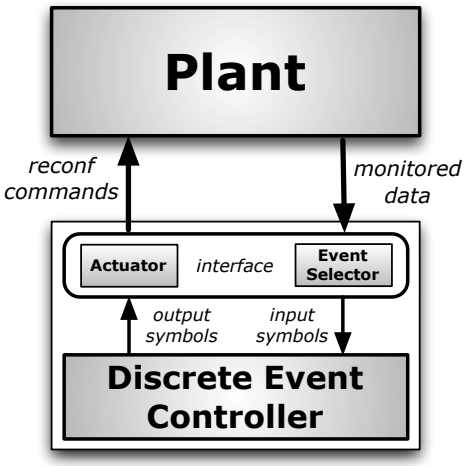


Figure 5.7: Discrete Event Controller: structure and interface with a plant.

In many existing industrial and engineering scenarios a common way to implement a reactive control logic is by introducing the concept of discrete event controller (DEC). In this approach a plant is interconnected with a controller managing input and output events expressed in terms of *symbols* from a finite alphabet. Figure 5.7 shows the main concepts of a DEC.

The controller communicates exclusively via discrete events. At any point of time the current input symbol to the controller identifies an input event which is typically generated by analyzing monitored data of the system execution. Each output symbol from the controller must be properly translated into a corresponding set of reconfiguration commands injected into the plant, which modify its behavior correspondently. These translation activities are performed by a proper interface [95] between the plant and its discrete event controller, composed of two interfacing structures:

- ***Event Selector***: an entity able to translate the actual values of monitored data from

the plant into a corresponding set of input symbols to the controller;

- ***Actuator***: an entity which translates output symbols generated from the controller into a corresponding set of reconfiguration signals transmitted to the plant.

Such translation activities can be executed in different ways:

- at equidistant instants of time (i.e. *synchronous discrete event controller*): e.g. a pair of input and output symbols are generated to/from the controller only at the beginning of each control step;
- controller symbols can be generated at time instants that are not predictable (i.e. *asynchronous discrete event controller*): e.g. only when certain observed measurements cross specific thresholds, corresponding symbols are generated from the event selector to the controller.

The previous description can be mapped onto our ParMod definition. The operating part is modeled as a switched hybrid system featuring a discrete set of alternative operating modes. The switching between different modes is regulated by the control part. In terms of input and output variables of the model, and the measurements effectively exchanged between operating part and control part, we have:

- control inputs $\mathbf{u}(k)$ of the operating part model have a direct correspondence with reconfiguration commands that will be effectively transmitted from the control part to the operating part;
- monitored data received from the operating part at the beginning of each sampling interval have a double aim: (i) they indicate the values of observed outputs $\mathbf{y}(k)$ of the operating part model at the beginning of the current control step; (ii) they identify the values assumed by measured disturbances that affected the operating part execution during the previous control step: i.e. $\mathbf{d}(k-1)$ where k is the current control step of the execution.

Based on this correspondence the reactive adaptation strategy of a ParMod can be formulated as follows:

- the operating part is a discrete-time switched hybrid system with controlled switching law. The plant behavior is described by the hybrid model (5.5);
- at the beginning of each control step, monitored data from the operating part allow the controller to have an updated view of the values assumed by observed outputs $\mathbf{y}(k)$ and past disturbances $\mathbf{d}(k-1)$ affecting the plant execution. The event generator translates these values into an input symbol $\xi(k)$ to the controller;
- the control part is a synchronous discrete event controller that receives the input symbol $\xi(k)$ and, after the evaluation of its internal logic, generates an output symbol $c(k)$ which identifies the current configuration that should be used for the whole duration of control step k;

- the output symbol $c(k)$ is translated by the actuator into a corresponding control input vector $\mathbf{u}(k)$ which is transmitted to the operating part.

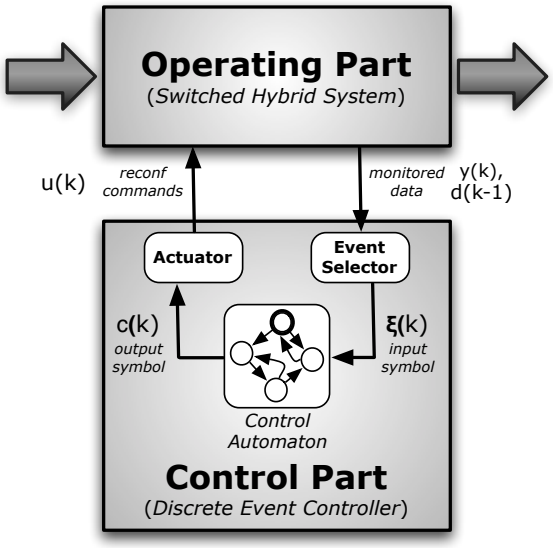


Figure 5.8: Reactive adaptation strategy of a ParMod.

We provide a formal definition of the reactive control by describing in more detail two main aspects of the control part (see Figure 5.8):

- an interfacing structure composed of an event selector and an actuator;
- a finite state machine which formally describes the reactive logic used by the discrete event controller.

The event selector, also called generator in similar research work [96], is responsible for analyzing the observed measurements received from the operating part. In our approach the generation of the input symbols is based on the evaluation of a fixed set of logical predicates over the values of the variables received from the operating part. For this reason we introduce the notion of ***conditional variables*** (shortly *CVs*):

Definition 5.3.1 (Conditional Variables)**.** A conditional variable is a propositional logic variable. Its value is determined by the event selector through the evaluation of a corresponding logical predicate over the numerical values received form the operating part. The propositional calculus is the formal system used for expressing these logical predicates. A conditional variable can be defined according to the syntax provided by the following grammar:

```
VAR ::= M | C
COND_VAR ::= VAR = VAR | VAR < VAR | VAR > VAR | VAR <= VAR |
```

```
VAR >= VAR | VAR != VAR | COND_VAR and COND_VAR |
COND_VAR or COND_VAR | not COND_VAR
```

where M indicates any monitored variable from the operating part (i.e. the value of an observed output or a disturbance), whereas C is any constant value.

Therefore the event selector generates the truth values of the set of conditional variables $CV_0, CV_1, \ldots, CV_{e-1}$ at each control step. The input symbol $\xi(k)$ transmitted to the control part is a specific configuration of a boolean value for every conditional variables. Hence the input symbol $\xi(k)$ is selected from a finite alphabet Ξ of 2^e possible alternative symbols.

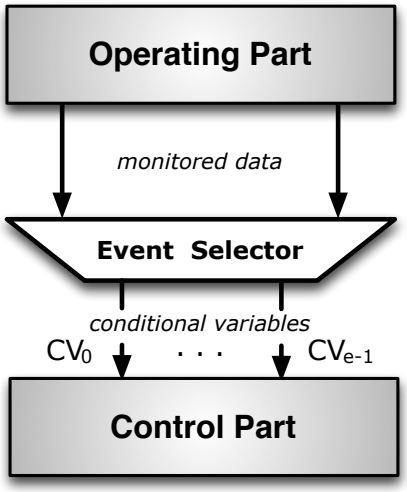


Figure 5.9: Event Selector and generation of conditional variables.

The behavior of the event selector is schematized in Figure 5.9 and a more precise definition is given below:

Definition 5.3.2 (Event Selector)**.** The Event Selector implements a pure function that maps current monitored data from the operating part onto a combination of boolean values for every conditional variable $CV_0, CV_1, \ldots, CV_{e-1}$. This mapping is established by evaluating a predefined set of logical predicates.

We denote with Ξ the finite set of all the possible combinations of boolean values corresponding to the conditional variables (Ξ consists in 2^e possible combinations).

The actuator is characterized by a dual behavior. At each control step k the output symbol from the controller indicates the ParMod configuration which has been chosen for being executed during the current control step. Thus the output symbol domain is the finite alphabet of possible configuration identifiers C introduced in (5.2). As stated in Section 5.2.1 this set is isomorphic to the admissible control input vector set $\mathbf{U}$ described in (5.4). Therefore the actuator is formally defined as follows:

Definition 5.3.3 (Actuator)**.** The actuator is a pure function that maps each output symbol from the controller (which corresponds to a configuration identifier), onto a corresponding set of reconfiguration commands (which correspond to a control input vector $\mathbf{u}(k)$) that will be transmitted to the operating part of the parallel module.

In addition to the interface units, the reactive logic of the control part is defined in terms of a finite state machine, namely ***Control Automaton***, featuring a finite alphabet of input, output and internal state symbols (i.e. it is an Input/Output (I/O) Automaton).

Definition 5.3.4 (Control Automaton)**.** A Control Automaton is defined as a tuple $\langle \Xi, S, C, \omega, \sigma \rangle$ where: Ξ is the finite alphabet of input symbols generated from the event selector, S is the finite alphabet of internal state symbols and C is the finite set of output symbols introduced in (5.2). Furthermore $\omega : \Xi \times S \to C$ and $\sigma : \Xi \times S \to S$ identify the internal state transition and the output generation functions defined as follows:

$$\begin{cases} s(k+1) = \sigma\big(\xi(k), s(k)\big) \\ c(k) = \omega\big(\xi(k), s(k)\big) \end{cases} \tag{5.6}$$

We can observe that the control automaton is a I/O finite state machine in which both the next internal state and the output symbol directly depend on the current internal state and input symbol. This model of I/O automaton is known as *Mealy machine* [97].

We summarize the reactive adaptation strategy. At the beginning of control step k the actual operating part behavior is described by an updated set of monitored data. In the interface layer the event selector maps this information onto a corresponding symbol $\xi(k)$ transmitted to the finite state machine modeling the control part of the ParMod. Based on the evaluation of the two transition functions (5.6), the next control part internal state and a new operating part configuration $c(k)$ for the current step are selected. Finally, the actuator is responsible for translating the output symbol $c(k)$ from the automaton into a proper set of reconfiguration commands transmitted to the operating part.

5.3.2 Towards the Optimal Control of Parallel Modules

Reactive adaptation strategies are control methods where system reacts to well- identified circumstances in pre-programmed way. Reactive control emphasizes the interaction with environment through perception-action pairs expressed in terms of a finite state machine (as in our approach) or exploiting logic rules (as described in Section 3.3.1).

In general reactive solutions are easy to design but they have also some critical shortfalls. In a purely reactive methodology the mapping between execution conditions and reconfiguration actions must be appropriately designed by the system manager who has developed the adaptation logic. Of course some approaches rely on an automatic learning of situation-action pairs but, in general, the utilization of a model that predicts the QoS future behavior can be a critical advantage that needs to be exploited in a better way.

For this reason we use control techniques able to improve the foresight of the ParMod adaptation strategy. In general terms predictive approaches are control methods where the system controller tries to estimate the future in some way, thinking ahead of corrective actions such that certain undesired conditions can be prevented. Although predictive methods are aimed at being one step further towards better overall system control than reactive ones, their applicability to real-world applications is constrained to:

- the presence of an empirical or first-principle model used to estimate the future system behavior in function of a planned sequence of control inputs and a future prediction of measured disturbances;
- the presence of an objective function describing the control aims which drive the selection of the optimal control input sequence;
- the specification of boundary conditions on state and control input variables.

This formulation of the predictive control is known to control theorists as ***Optimal Control*** [98]. Optimal control is the process of determining control and state trajectories of the system over a certain period of time in order to optimize a properly defined objective function. A typical representation of an objective function is given below:

$$J(\mathbf{x}(0),N) = \Theta\big(\mathbf{x}(N)\big) + \sum_{i=0}^{N-1} L\big(\mathbf{x}(i),\mathbf{u}(i)\big) \tag{5.7}$$

The function represents and aggregate cost which depends on the desirability level of future system internal states and the cost of control inputs taken for a horizon of N successive control steps (e.g. ideally the whole execution duration). Θ is the *final cost* of the resulting system internal state at the end of the horizon and L is the *stage cost* for each intermediate step. The resolution of the optimization problem yields to the identification of an optimal control input trajectory (i.e. ***a reconfiguration plan***) which can be applied step by step by the controller in an open-loop way.

In our case the basic requirement for the application of the optimal control approach is the existence of a model of the ParMod behavior. Future QoS measurements can be predicted exploiting the hybrid model (5.5) of the ParMod operating part. Suppose to be at the beginning of the control step k, and to know the exact value of the current model state $\mathbf{x}(k)$. If we are able to make a prediction of disturbance input values for the entire duration of the current control step we can predict the state value at the beginning of the next control step:

$$\hat{\mathbf{x}}(k+1) = \Phi\big(\mathbf{x}(k),\hat{\mathbf{d}}(k),\mathbf{u}(k)\big)$$

where the symbols $\hat{\mathbf{x}}$ and $\hat{\mathbf{d}}$ indicate future estimations of internal state and disturbance input variables.

If we suppose to know the exact trajectory assumed by disturbance inputs for the whole execution duration, and if the plant model is sufficiently accurate and precise, we can statically define an optimal reconfiguration plan that optimizes the ParMod execution. In practise this assumption is often unfeasible for two main reasons:

- disturbance inputs are variables whose behavior (e.g. average values) can not be statically known a priori but they may depend on uncontrollable factors (as the actual conditions of the underlying execution platform);
- the plant model can be effected by perturbations and unmodeled dynamics (e.g. due to unmeasured disturbances) that limit the quality of the future QoS estimations.

For the previous reasons open-loop optimal control techniques are difficult to be exploited directly, but on the other hand several suboptimal approaches are available in order to iteratively apply optimization problem solutions in a closed-loop fashion. In the next section one of this technique, namely ***Model-based Predictive Control***, will be introduced.

5.3.3 Model-based Predictive Control

Model-based predictive control [99, 100] (shortly **MPC**) is a repetitive procedure that combines the advantages of a long-term planning (feedforward control based on system predictions over a future horizon) with the advantages of reactive control (feedback using actual monitored information from the system). At the beginning of each control step k the monitored data from the plant are acquired (e.g. the value of current observed output variables and past measured disturbances). Without loss of generality in this description we assume a completely observable model state, thus the observed output vector coincides with the internal state vector, i.e. function H (5.5) is the identity. At this point a future time horizon (i.e. called *prediction horizon*) of h consecutive control steps is considered, and a prediction of disturbance inputs for this time interval is exploited through appropriate statistical techniques. So doing we are able to estimate the following trajectory:

$$D_k^{k+h-1} = \{\hat{\mathbf{d}}(k|k), \hat{\mathbf{d}}(k+1|k), \ldots, \hat{\mathbf{d}}(k+h-1|k)\} \tag{5.8}$$

where the syntax $\hat{\mathbf{d}}(k+i|k)$ means that the estimated value of disturbances for the step $k+i$ is predicted using the knowledge available at control step k. For simplifying the notation we adopt the syntax $\overline{D}(k)$ for identifying the trajectory of disturbance inputs from the current step k.

These predictions are used to plan a sequence of reconfiguration decisions for each control step of the prediction horizon. An optimization problem similar to (5.7) is solved for a limited time horizon and an optimal control input trajectory is determined:

$$\overline{U}(k) = \{\mathbf{u}(k|k), \mathbf{u}(k+1|k), \ldots, \mathbf{u}(k+h-1|k)\} \tag{5.9}$$

where $\mathbf{u}(k+i|k)$ indicates the control input vector selected by the control part for the step $k+i$ based on the decisions taken at the current step k. This trajectory is chosen in such a way as to optimize the system objective function for the whole prediction horizon.

At this point one can think to apply this reconfiguration plan step by step in an open-loop fashion and calculate a new optimal control trajectory only at the end of the prediction horizon. In practical scenarios this is not an effective approach for the same motivations expressed at the end of the previous section. The uncertainty of disturbance input

estimations (which increases going deeper in the prediction horizon) and the potential inaccuracy of the modeled dynamics suggest a more effective approach based on an iterative procedure. Only the first control decision of the optimal trajectory will be transmitted from the control part to the operating part at the beginning of the current step k. When this control step ends, the entire procedure is repeated at the next control step exploiting the new measurements from the system. Therefore instead of applying the reconfiguration plan in an open-loop way, this plan is continuously checked and updated at each control step. The effect is to move the prediction horizon towards the future following the so-called ***receding or rolling horizon*** technique (see Figure 5.10).

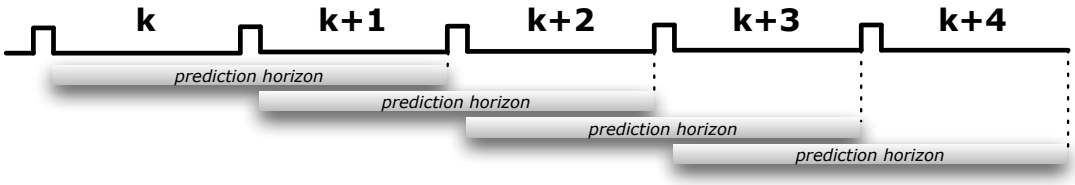


Figure 5.10: Example of the receding horizon principle with a prediction horizon length of two control steps.

If we assume the existence of a sufficiently accurate model of the operating part behavior the benefits of the predictive control are evident. Suppose to have a ParMod control objective formulated as a threshold specification problem (see Section 3.3): we need to maintain the observed mean throughput of a ParMod within an acceptable region of values (e.g. between a maximum and a minimum threshold established by the user). Although the best initial configuration can be identified during the design phase, future modifications of the initial conditions modeled as disturbance inputs can prevent this configuration to be no longer acceptable. In this case the utilization of a predictive strategy can be a interesting approach for two main reasons:

- selecting in advance reconfiguration actions can be crucial in order to anticipate undesired behaviors and promptly mitigate the future variability of disturbance inputs;

- reconfigurations usually involve a cost on the execution. After a reconfiguration decision a ParMod can incur some dead-time, due to the control delay for performing deployment actives, for completing reconfiguration protocols (see Section 5.1.3) or rather the cost for acquiring computational resources.

In practical applications the horizon length h can not be an unconstrained parameter, and the MPC technique usually exploits relatively short horizons (e.g. $h = 2,3,4$) due to the inaccuracy of the operating part model, the prediction errors in estimating disturbance inputs, and, as we will discuss in Section 5.3.3.2, also for computational complexity reasons.

In the rest of this chapter we will explain in more detail the theoretical aspects concerning the applicability issues of predictive control for controlling ParMods. A description of real-world experiments of structured parallel computations will be discussed in Chapters 6.

5.3.3.1 Formulation of the MPC optimization problem

The MPC approach consists in finding and optimal control trajectory for a limited horizon that optimizes a cost functional of the system. By applying this reconfiguration plan the ParMod QoS is driven towards a trajectory of future internal states:

$$\overline{X}(k) = \{\hat{\mathbf{x}}(k+1|k), \hat{\mathbf{x}}(k+2|k), \ldots, \hat{\mathbf{x}}(k+h|k)\} \tag{5.10}$$

In a minimization context (the maximization formulation is completely dual) the optimization problem can be stated as follows:

$$\begin{aligned}
&\underset{\overline{U}(k)}{\operatorname{argmin}}\, J\big(\overline{X}(k), \overline{U}(k)\big) \\
&\text{subject to:} \\
&\mathbf{u}(i|k) \in \mathbf{U}\ i = k, k+1, \ldots, k+h-1 \\
&\hat{\mathbf{x}}(i+1|k) = \Phi\Big(\hat{\mathbf{x}}(i|k), \hat{\mathbf{d}}(i|k), \mathbf{u}(i|k))\Big)\ i = k, k+1, \ldots, k+h-1 \\
&\hat{\mathbf{d}}(i|k) = \overline{D}(k)[i]\ i = k, k+1, \ldots, k+h-1 \\
&\hat{\mathbf{x}}(k|k) = \mathbf{x}(k)
\end{aligned} \tag{5.11}$$

where the constraints indicate respectively: (1) control vectors must belong to the admissible set $\mathbf{U}$, see (5.4); (2) future internal states are estimated by means of the operating part model Φ (5.5); (3) disturbance input vectors are part of the predicted trajectory $\overline{D}(k)$ (5.8), where the notation $[i]$ indicates the i-th element of the trajectory; (4) the first internal state of the trajectory $\mathbf{x}(k)$ is obtained through the current monitored data received by the operating part at the beginning of the current control step k.

The behavior of the predictive control part is shown in Figure 5.10. The controller, received monitored data from the operating part, computes the disturbance predictions and solves the optimization problem in order to find the best control trajectory. Then the first action of the optimal sequence will be issued to the operating part.

One of the most crucial point of the predictive control approach is the definition of the objective function which establishes the aim of the adaptation strategy. For instance the controller can:

- optimize the performance behavior of a parallel computation (e.g. in a stream-based computation the number of completed tasks);
- express multiple trade-offs between reconfiguration costs and the desirability level of future QoS measurements as the performance, energy consumption and memory occupation of the controlled parallel computation.

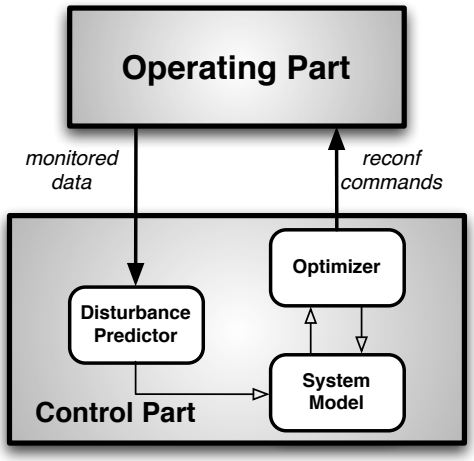


Figure 5.11: Abstract view of the ParMod control part with MPC strategy.

Moreover objective functions can also be modeled in such a way as to account the model inaccuracies and the quality of the disturbance input predictions. As an example in some research experiences (i.e. as in [101]) specific techniques have been provided in order to encode in the objective functions the risk of the decision process (e.g. considering an uncertainty band) and exploiting discounting factors for measurements that become inaccurate as we go deeper into the prediction horizon.

5.3.3.2 Predictive control of Hybrid Systems: complexity issues and approaches to tractability

Over the last decades optimal control techniques and suboptimal methods as MPC have been applied for controlling several chemical, mechanical and industrial plants. In most cases the optimization problems involve a continuous set of control inputs. In this context a linear approximation of the system behavior in a limited operation region (e.g. by applying system identification techniques) and the utilization of convex cost functions, lead to optimization problems that are easy to solve in general. Although linear MPC controllers are widely used in several real-world scenarios, there exist many situations in which they are not sufficiently accurate due to the presence of process nonlinearities that can not be neglected. In this case the optimization problem is usually harder (i.e. non-convex optimization), and the MPC feasibility is a challenging problem especially in real-time scenarios.

The previous limitations are even more drastic for switched hybrid systems, where the admissible control inputs belong to a finite and discrete set of choices. This is exactly our case. We face with a combinatorial optimization problem that theoretically implies an exhaustive search of the best solution by testing all the possible feasible combinations of reconfiguration decisions, with a consequent intractability of the resulting optimization process. In this section we will discuss proper techniques in order to render this approach

feasible for our typical scenarios.

Before starting with the description, it is important to note that predictive control techniques have already been applied to the problem of controlling hybrid systems in many different research contexts. In [102] a formulation of the MPC approach is applied to a supply chain management. In [103] a modified depth-first search algorithm is exploited to search the optimal solutions for controlling a pneumatic hopping robot. In [104] the MPC tractability is faced by investigating Neural Network techniques. A novel modeling of hybrid systems based on fuzzy models has been applied in [105, 106]. An interesting application of the MPC technique to hybrid systems has been described in [101, 107, 108, 109]. In this research work a predictive controller (called Limited Look-ahead Predictor) is defined and used for exploiting performance-power control of a CPU by adjusting its clock rate among a finite set of alternative frequencies. This research work applies optimal control techniques by exploiting statistical forecasting of measured disturbances (in this work called environmental inputs), which represents a similar approach to the one proposed in this section: also the behavior of structured parallel computations is significantly influenced by un-controllable exogenous inputs whose values can be predicted over limited prediction horizons.

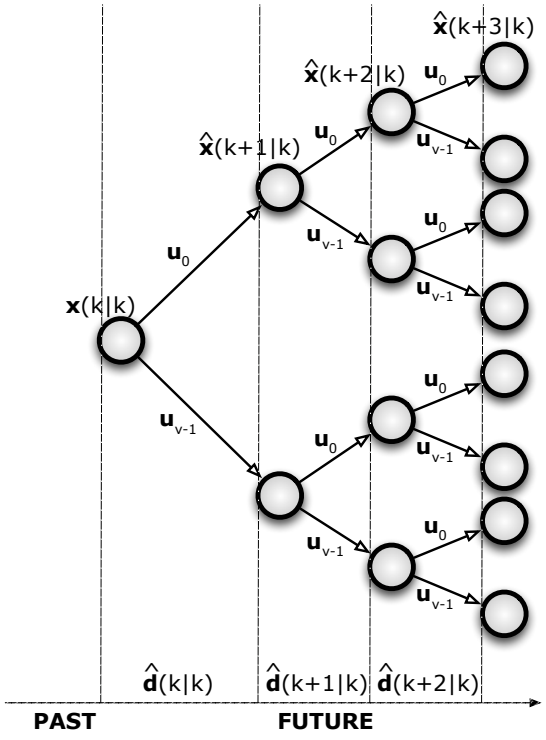


Figure 5.12: Example of a ParMod Evolution Tree: prediction horizon length equal to three control steps.

We can describe the future evolution of a ParMod (in terms of its QoS measurements) as a tree structure called ***evolution tree***.

Definition 5.3.5 (ParMod Evolution Tree)**.** At each control step k the evolution tree of a ParMod is a complete tree in which its arity corresponds to the finite number ν of possible alternative configurations of the ParMod (see (5.2)), and the height is equal to the prediction horizon length h. Each node at a certain level $l = 0, 1, \ldots, h$ indicates the future state that the operating part model will assume at control step $k + l$, where the radix node corresponds to the actually measured state. Each edge describes a possible reconfiguration decision: each node (except the leaves) has exactly ν out-going arcs for each admissible control input vector. The tree is built by exploiting the operating part hybrid model and taking the prediction of measured disturbances for each level of the evolution tree (that correspond to the length of the prediction horizon).

Figure 5.12 describes the evolution tree of a ParMod with a prediction horizon of three control steps. As we can see, finding the optimal trajectory of reconfigurations consists in searching a path from the radix to any leaf such that the aggregate sum of costs for each step is the best (minimum) one. Then, according to the MPC iterative procedure, only the first control input of the optimal trajectory will be applied to the operating part.

Since we have a finite number of control inputs and a fixed prediction horizon of h steps, then also the number of possible reconfiguration trajectories is finite and is given by ν^h, thus exponential in the length of the prediction horizon. Due to the finite (even if potentially enormous) size of the feasible solution set, the problem can be viewed as a search process in a tree data-structure, where the theoretical number N_{space} of explored states is given by the following expression:

$$N_{space} = \sum_{i=0}^{h} \nu^i \tag{5.12}$$

Hence the optimization problem is *NP-hard* and the computational effort needed to solve it grows exponentially with the size of the problem. A naive approach, consisting in an exhaustive search, is computationally prohibitive except for short prediction horizons and for control problems with few control inputs.

Therefore the MPC exploitation for hybrid systems requires to develop proper techniques for dominating the complexity. Although a broad description is out of the scope of this thesis (interested readers can read [110] for further details), in this part of the chapter we focus on a general technique based on a ***Branch and Bound*** (B&B) method. Similar approaches have been used in [111, 112] for controlling hybrid systems with discrete and mixed inputs.

B&B methods are enumerative schemes based on the following consideration: only a limited portion of the search space will contain the optimal solutions, while many other parts can be eliminated if proper assumptions hold. Let us introduce two operations:

- *Branching* is done by partitioning the set of feasible solutions into smaller partitions. This operation is applied to the current node of the evolution tree and consists in applying one of the possible ν alternative reconfiguration decisions, which generate ν smaller search sub-spaces;

- *Bounding* is done by checking specific criteria in order to decide whether a branch needs to be examined or not.

This approach can be applied to the predictive control of parallel modules in order to reduce the computational effort and make the running time of the control algorithm ($T_{Control}$) feasible w.r.t the control step length. A general bounding strategy is applicable if the MPC optimization problem retains the following property:

Assumption 5.3.1. *The objective function used by the predictive strategy must represent a cost which is monotonically increasing with the steps of the prediction horizon.*

In other words going deeper in the prediction horizon the value of the objective function can not decrease. If this condition holds, we can apply the following bounding procedure (depicted in Figure 5.13):

- each node i of the tree is labeled with the aggregate cost C^i from the radix of the tree to that node. The cost of the radix is zero;

- for each node i we have ν potentially branches for each control input (reconfiguration). The branch corresponding to the control input $\mathbf{u}'$ is followed iff the following inequality holds:
$$C^i + L_w(i, \mathbf{u}') < C^{max}$$
where $L_w(i, \mathbf{u}')$ is a lower bound for the cumulative cost of all the paths starting from j and reaching a leaf. Node j is the one which is reached from node i after applying control input $\mathbf{u}'$. C^{max} is equal to an upper bound for the global optimal cost.

The previous conditions require to estimate a lower and an upper bound in order to make an effective reduction of the search space. If such estimations are not available an effective procedure can still be applied, as discussed for a similar problem in [113]. In this case we can consider a temporary variable C^{opt} that contains the minimum cost of all radix-to-leaf paths already explored during the search process (C^{opt} is initialized to ∞). If the current node i is a leaf of the evolution tree, if $C^i < C^{opt}$ then this path becomes the best one actually found during the search algorithm (thus $C^{opt} = C^i$). If i is not a leaf and the following condition holds, i.e. $C^i \geq C^{opt}$, than for the monotonicity of the cost function we can avoid to search in the entire sub-tree rooted at i. The application of this idea to the ParMod predictive control is described in Algorithm 3.

The algorithm follows a depth-first strategy to build the evolution tree and branches the portions of the search space that certainly do not yield to the optimal solution. The algorithm inputs are the initial value of the operating part model at the beginning of the current control step, i.e. x_{init}, and the predicted trajectory of disturbance inputs $\overline{D}$ for every control steps of the prediction horizon. The basic data structure used by this algorithm is a tree *Node N* composed of four internal variables: (i) the predicted value $N.x$ of the internal state of the operating part model; (ii) the node level $N.level$ in the evolution tree;

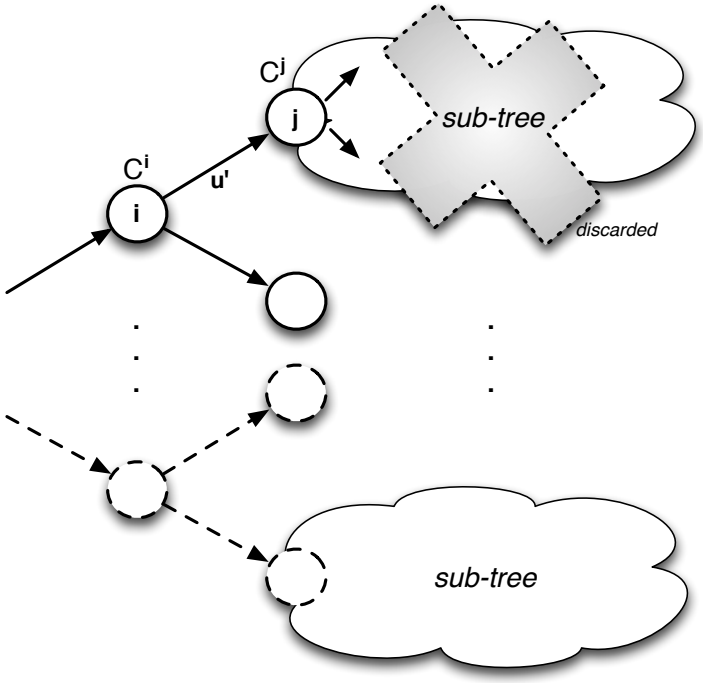


Figure 5.13: Representation of the Branch & Bound approach.

(iii) the cumulative cost $N.cost$ from the radix to that node N; (iv) the trajectory $N.\overline{U}$ of reconfigurations up to this node. From line 8 to line 10 the algorithm checks if the current node is a leaf and if its cumulative cost is the best found so far: in this case variables N_{opt} and C^{opt}, storing a reference to the best node and its cumulative cost, are updated correspondently. From line 11 to line 17 the algorithm checks the non-leaf case: if a node is not a leaf and if its cumulative cost is smaller than the best cost found so far, in this case a new node is instantiated and added into a stack data structure S. Otherwise the subtree is not explored. The algorithm concludes when no node is present in the stack. At line 16, the operation "$\circ$" consists in appending a new control input at the end of the reconfiguration trajectory (e.g. implemented as a list).

As we will show in Chapter 6, the utilization of this B&B method will be of great importance for improving the computational feasibility of predictive control. In fact B&B techniques have important advantages: (i) a global optimum solution is always found; (ii) the algorithm does not suffer of poor initialization conditions; (iii) very few restrictions on the objective function structure are required; (iv) B&B implicitly deals with constraints. The last point is especially useful if we use B&B in conjunction with other techniques as *blocking strategies*. For blocking strategy we intend that the control part is designed in such a way that if a specific configuration for the operating part has been decided, this must be fixed for a certain set of consecutive control steps (block duration). In this case

Algorithm 3: Branch&Bound Search(x_{init}, $\overline{D}$)

Data: the initial value of model state x_{init} and the predicted trajectory $\overline{D}$ of disturbance inputs.
Result: the optimal trajectory $N_{opt}.\overline{U}$ of reconfiguration decisions.

```
1  begin
2      J_opt ← ∞;
3      N_opt ← nil;
4      //First node with value x_init, null cost, level 0 and
       empty reconfiguration trajectory.
5      S ← {Node(x_init, 0, 0, [])};
6      while S ≠ ∅ do
7          N ← pop(S);
8          if (N.level == h) and (N.cost < J_opt) then
9              J_opt ← N.cost;
10             N_opt ← N;
11         else if N.level < h then
12             foreach u ∈ U do
13                 x' ← Φ(N.x, D̄[N.level], u);
14                 c ← Cost(x', u);
15                 if N.cost + c < J_opt then
16                     N' ← Node(x', N.level + 1, N.cost + c, N.Ū ∘ u);
17                     S ← Push(S, N');
18     return N_opt.Ū;
```

such constraints require further bounding conditions that improve the search space reduction. Other techniques imply the utilization of *suboptimal termination conditions*, e.g. as soon as a radix-to-leaf path with a cumulative cost lower than an acceptable threshold is found, the corresponding control input trajectory is used as to optimal one. In this case an effective approach consists in using a *priority list* at the place of a stack S, in which the extraction operation returns the node in the list with the minimum cost (i.e. the next node is selected according to a *greedy* strategy).

5.3.3.3 Forecasting disturbance input trajectories

As described in the previous section the predictive control strategy for structured parallel computations is based on the assumption that, having a deep knowledge of the behavior of the computation, we are able to define a proper model of the operating part behavior such that the future QoS levels can be predicted with a sufficient accuracy. However, we have also seen that the expected QoS can also be heavily influenced by exogenous disturbance inputs which are not controllable, but they reflect environmental

execution conditions. As an example let us consider a distributed parallel application executed on a large-scale distributed-memory architecture (e.g. a cluster of workstations or a geographical computational Grid), in which the interconnection network features a time-varying availability in terms of network bandwidth and latency (see Figure 5.14). A parallel computation usually requires to exchange several data between parallel entities: in this scenario the communication latency between processes, that is the average time spent in transmitting and receiving messages, plays a crucial role for the efficiency of a parallel execution. Especially when the network workload features important fluctuations and congestions, the time spent in communications may change significantly over the execution.

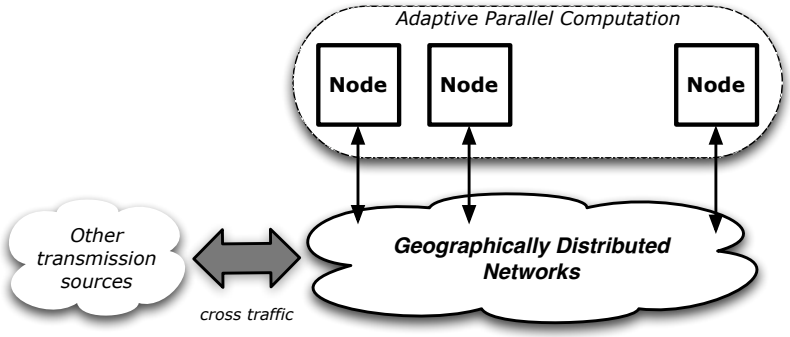


Figure 5.14: Time-varying network availability scenario.

The above scenario can be generalized in this way: in order to define the most convenient reconfiguration plan for a parallel module, we need to characterize the predicted trajectory of disturbances. Some of them exhibit either a *resource-dependent* behavior (e.g. due to resource availability fluctuations) or an *application-dependent* behavior (e.g. dynamics due to the behavioral patterns of the users or of the applications themselves). At this regard disturbance variables can be:

- the mean communication latency between cooperating parallel entities implementing the ParMod operating part. As shown in the previous example this parameter can be influenced by exogenous causes as the actual behavior of the interconnection network;
- in a stream-based computation the mean inter-arrival time (i.e average time between the reception of two consecutive input tasks from the users) can exhibit time-varying and periodical fluctuations. This can be due to exogenous factors, as the network behavior, or to endogenous aspects of our applications, as the users' behavioral patterns;
- the mean calculation time per task of a stream-based computation can be another source of dynamicity. This can be due to exogenous factors, as the resource availability level (e.g. actual CPU load) but also to endogenous causes as in the case of

irregular algorithms [114] in which the weight of input tasks can depend on specific properties of the received data structures.

In general the problem of selecting the most effective predictive technique depends on two main factors: (i) the length of the prediction horizon and of the sampling interval; e.g. if we are interested in long-term predictions (e.g. in the range of hours) or in short-term estimations (e.g. in the range of few minutes); (ii) the type of the predicted information. As one can expect this suggests that it is not possible to provide a general technique for statistically estimate disturbance input trajectories.

Statistical techniques have been studied over the last decades [115]. Detecting failures, threshold violations, promptly detect critical execution conditions and predict the user behavior is as difficult as important. A wide-spread approach is based on historical data (i.e. ***time series***), represented as discrete observations taken at equal time intervals (e.g. at each control step). Suppose to have a windows of T samples of a disturbance variable. We can apply statistical models as follows:

$$\hat{\mathbf{d}}(k|k) = \Psi\big(a(k), \mathbf{d}[k,T]\big)$$

where $\mathbf{d}[k,t]$ indicates the sequence of the last T historical samples of a disturbance input, i.e. $\mathbf{d}[k,T] = \{\mathbf{d}(k-1), \mathbf{d}(k-2), \ldots, \mathbf{d}(k-T)\}$ where k is the current control step and $a(k)$ are a set of numerical coefficients. The previous expression denotes a general model capable to predict the next value of a disturbance input (i.e. one-step ahead), but the same approach can be iteratively repeated for h successive steps of a longer prediction horizon by using the last estimation as the new historical measurement.

5.4 Summary

In this chapter we have introduced the formalization of the adaptive parallel module (ParMod) concept. After the description of the operating and a control part structuring, in this chapter we have described a formal modeling of the operating part behavior based on switched hybrid systems: several operating modes (configurations) can be identified, and for each one of them a proper model can be expressed in order to predict the QoS future behavior of the computation. Next, proper control techniques have been presented concerning the adaptation strategy adopted by a single parallel module. A first attempt consists in a reactive approach based on discrete event controllers: i.e. control part behaves as a formal automaton. In order to provide a more powerful class of reconfiguration strategies, in this chapter we have proposed an approach based on the optimal control of a ParMod. At this regard we have discussed the exploitation of the model-based predictive control technique, which is an interesting approach for controlling a systems with a discrete set of control inputs. In the next chapter we will complete this description evaluating the effectiveness of this control approaches with two real-world distributed parallel applications.

Chapter 6

Experiments on Controlling single Adaptive Parallel Modules

In this chapter we apply the control strategies introduced in Chapter 5 in two real-world experiments. For each of these examples we consider the presence of a unique adaptive parallel module whose behavior will be adapted by executing functional and non-functional reconfiguration activities. In this chapter we avoid to discuss the problem of coordinating the adaptation logics of multiple ParMods of the same computation graph. This problem, known as distributed control, requires an extension of our control model that will be addressed in the next chapters of the thesis.

The organization of the chapter is the following. In the first part we describe a real-time system for managing environmental emergency scenarios like floods, in which different QoS objectives involving the execution of a compute-intensive fluid dynamics computation need to be optimized throughout the system execution. In this example we evaluate and compare the exploitation of the reactive and the predictive MPC control strategies, discussing their control outcome and the effectiveness of these two adaptation techniques.

In the second part we introduce an image-processing application based on a client-server architecture, in which the parallel server component tries to maintain its response time under a maximum acceptable threshold established with the users and minimize the number of reconfigurations. In this context we evaluate the predictive control technique with different lengths of the prediction horizon, discussing the impact of proper search-space reduction techniques for improving the feasibility of this approach in real-time contexts.

6.1 Adaptivity on a System for Flood Detection

The last years have been characterized by a strong evolution/revolution of several innovative distributed applications thanks to the development of efficient wired/wireless communication technologies, the wide-spreading of multi-/many-core processors, and,

consequently, to their integration into complex and heterogeneous computing platforms (e.g. Computational Grids and Clouds) [2, 4]. The provision of computational resources on-demand, through efficient computer networks, is a critical requirement of a large set of next-generation distributed systems, that require the execution of time-consuming computations featuring tight real-time requirements. Usually such computations have to be executed on sufficiently powerful back-end centralized architectures (e.g. as traditional Grid or Cloud platforms) or, sometimes, by a cooperative utilization of mobile platforms (e.g. as Mobile Grids).

A notable example of such applications are Emergency Management Systems (shortly EMS), developed for addressing the critical demand of computation and communication during the different phases of a natural or man-made disaster (e.g. as in flood and earthquake contexts). In order to facilitate the decision making process of civil protection personnel, EMS provide the periodical execution of computationally intensive simulations (e.g. flood forecasting models) and the prompt and capillary diffusion of the results to the interested users, equipped with heterogeneous endpoint devices (e.g. pervasive mobile platforms as Personal Digital Assistants and smart-phones). The complexity of this scenario is further complicated by fluctuating demand of such computations/services, due to unexpected environmental conditions (e.g. the detection of an emergency can lead to stricter computational requirements) or due to the dynamic availability of the underlying computing and communication facilities. This last situation is especially true in large-scale platforms in which the co-existence of several applications and users renders the quality of the resource utilization time-varying with extremely high fluctuations.

This context is a typical example in which a methodology for designing and developing adaptive parallel applications plays a decisive role. In fact such scenario depicts the necessity to keep updated the configuration of a parallel computation, in order to maintain the necessary performance levels and, at the same time, optimizing the resource utilization level or in general the operational cost of the service delivery. This last aspect is of special interest in Cloud Computing environments, in which the computational power of large server architectures is sold as a service [4] to the interested users (following a *pay-per-use* model).

The application that we are presenting in this section is an example of the emergency management system for flood emergencies previously described in [60, 27, 61]. It is represented as an acyclic computation graph sketched in Figure 6.1. We can identify the

Figure 6.1: Computation graph of the first experiment. In this example the Solver ParMod is the unique parallel module with an adaptation logic.

presence of three application components exploiting sequential or parallel computations:

- a third-party wireless sensor network includes a set of sensors deployed along and around an observed environmental scenario (e.g. a river basin), with the aim of monitoring punctual precipitations, water depth and water speed. Data collected from sensors are stored in a database or in a Geographical Information System and are made available to a pre-processing sequential component (namely ***Generator***), able to produce stream of data to the other application components;

- the main core of this application is the flood forecasting component (i.e. ***Solver***), which takes as input environmental data from the generator and returns the near or far future forecasts for a specific requested area;

- endpoint application components (***Clients***) are available to final users equipped with mobile devices or dedicated workstations and laptop computers. Users can be civil protection personnel in an institutional center or even mobile operators directly involved in the emergency management activities. They operate by observing flood forecasting results in real-time and by applying the results of the further application components for decision support or for temporal-spatial analysis.

In this application context, flood forecasting activities and the real-time monitoring of the interested area are critical phases. The first one is mostly related to risk prevention, and it is characterized by harder performance requirements than the second case. In practice, this activity gives detailed and updated information on the (near) future situation of the whole monitored area in temporal terms of few minutes or hours. In typical settings, a complex hydrodynamic modeling software is executed on sufficiently powerful computational resources, acquiring input data from all available monitors (i.e. from the generator component) and delivering results to the monitoring board of the civil protection central division (post-elaborated and visualized by a client component).

The adaptation of the underlying available resources to the transient demand plays a central role for such class of systems. A promising paradigm is Cloud Computing [4], in which the provision of resources (*Infrastructure-as-a-Service*) as CPU, storage, network bandwidth is provided on-demand via computer networks by a service provider with a high-degree of flexibility and virtualization to the final users. For instance to support a dynamic workload it is reasonable to adapt the number of real or virtualized nodes available for executing a compute-intensive elaboration by a proper interaction with a Cloud Management Infrastructure (CMI).

In this example we will assume a concrete mapping between computation modules and the distributed execution resources. Generator and clients are application components executed on specific workstations whereas, the core element of the system i.e. the solver module, is hosted on a remote cloud architecture featuring a large set of computing elements and storing capability.

6.1.1 Flood forecasting system and Parallelization

In this section we focus on the solver parallel module giving a brief overview of the mathematical formulation and the computational requirements of this phase. Interested readers can obtain more detailed information about this application in [60]. The main core of the flood forecasting component can be expressed in terms of a bi-dimensional hydrodynamic model (see [116] for the original formulation). To do so, we define a 2D discretization of the environmental scenario (i.e. we suppose a river basin) and, for each space point, we solve a system of partial differential equations modeling the conservation of mass and momentum with the following parameters: water surface elevation; depth averaged velocity components in X and Y directions; depth of water; distance in X and Y directions; horizontal diffusion of momentum coefficient; Coriolis coefficient; Chczy coefficient; logical time at which variable values are collected. The result is represented by the sum of the components of external forces in the X and Y directions.

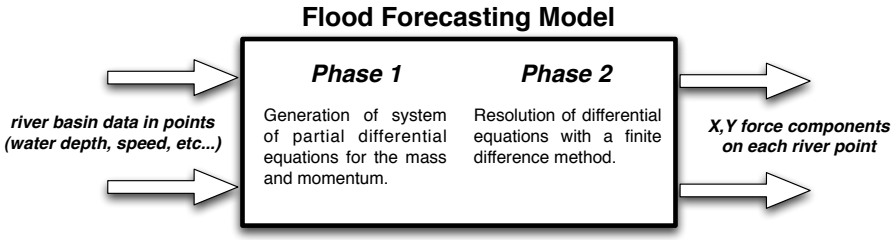


Figure 6.2: Abstract scheme representing input data, computation and output data of an implementation of the flood forecasting model.

We can solve these systems of differential equations according to finite difference methods, broadly obtaining a task consisting of the resolution of a set of tri-diagonal systems of linear equations [116]. Therefore, the computing core of this component can be defined as follows (see Figure 6.2): it takes as input the information from GIS and/or the WSN through the generator; it instantiates the system of partial differential equations and solves the related tri-diagonal systems returning the numerical values of the external force components. We model this computing core as a parallel module where:

- input stream elements include all the environmental information for each discretization point described above. The input data-structures are large collections of parameters represented as double precision floating-point numbers. Typical size of each stream element depends on the discretization degree of the environmental scenario (from 4 to 32 MB in our experiments);
- the computation quality can be configured by selecting finer or coarser grains of the time-discretization, impacting on the size of tri-diagonal systems;
- the output stream elements transmitted to clients are vectors expressing the force components whose size depends on the size of the solved systems.

There exist multiple optimized techniques for solving tri-diagonal systems [117]. Particularly interesting are direct methods, i.e. methods which attempt to find an exact solution in a fixed number of steps. Examples are twisted factorization and cyclic reduction [117]. Especially for the latter one very optimized variants exist, that minimize the necessary number of floating-point operations.

Therefore the application behavior can be schematized as follows. The generator module periodically receives environmental information from sensors and GIS services. This component generates a large set of independent requests (tasks) for each point of the two-dimensional discretization of the river basin. Each request is represented as an input data-structure containing the required parameters of the model. For each received task, the solver ParMod computes the two phases of the hydrodynamic model depicted in Figure 6.2. The most computationally intensive step is the re-solution of a set of tri-diagonal systems by applying an efficient variant of the cyclic reduction algorithm. Due to the fact that each input point can be computed independently from the others, this computation is amenable of being parallelized by a ***task-farm*** scheme (see Section 3.1.1), in which the resolution of different input tasks is applied in parallel by a proper set of parallel worker functionalities hosted on a specific number of processing elements on the centralized server. The results are transmitted with the corresponding identifiers to a client component, that exploits post-processing activities and results visualization.

6.1.2 Operating Part model and Control Strategies

In this section we define a model that describes the QoS behavior of the solver ParMod. We can identify two distinct aims of the control action:

1. ***Performance objective***: in order to provide fresh and updated information to the clients, *we need to complete the forecasting computation in a completion time as minimum as possible*;

2. ***Operational Cost objective***: in the technological scenario described above, we have supposed the presence of a centralized cloud server architecture hosting the solver ParMod elaboration. In the cloud computing paradigm customers do not own the physical infrastructure, but they pay a cost proportional to the amount of time and the amount of resources they use. In this application a certain number of computing nodes (CPUs) are dynamically allocated to the users. Customers pay a monetary cost for each reserved CPU and for the effective time of their utilization. Additional homogeneous processing elements can be allocated in the cloud architecture by adding new CPUs to existing virtual machines, and/or by switching on real computing nodes on-demand. As suggested in [118], in order to discourage too many resource re-organizations in response to fluctuating run-time conditions, we suppose a fixed cost that should be paid each time a new resource request is submitted to the cloud management infrastructure. In this scenario *it is of great importance to optimize the operational cost*: that is to adapt the resource utilization to the current

application workload and, at the same time, minimizing the interaction with the cloud architecture for changing the current configuration.

In order to address these two requirements the solver ParMod is provided with an adaptation logic able to express non-functional reconfigurations (see Section 3.2), i.e. modifications of the current utilized parallelism degree. The number of real (or virtualized) processing elements on which worker functionalities are hosted can be adapted to the expected workload expressed in terms of number of tasks per second received from the generator. We will define different adaptation strategies for the solver ParMod. First of all the main requirement that we have to meet is the performance-related objective: we need to dynamically select the parallelism degree in order to optimize the number of computed tasks throughout the execution. Furthermore, among the set of strategies able to target this requirement, we adopt a strategy that produces the lowest operational cost.

Following the methodology presented in Chapter 5, in this section we will describe a modeling of the QoS parameters of the solver ParMod computation. We will use these specific classes of input and state variables:

- *disturbance input variables* $\mathbf{d}(k)$: in order to model a time-varying workload we are interested in the average number of tasks received from the generator per control step. Thus we consider a variable mean inter-arrival time per step $T_A(k)$. In Section 6.1.3 we will be consider a case in which the value of this variable changes due to an dynamic availability of the underlying network environment;

- *state variables* $\mathbf{x}(k)$: in order to model the QoS of the ParMod execution, we introduce a state variable representing the actual length of the input task queue of the solver ParMod. This variable is denoted with $q(k)$, and indicates the actual queue length at the beginning of control step k;

- *control input variables* $\mathbf{u}(k)$: in this example the configuration of the solver ParMod is uniquely identified by the parallelism degree parameter. Therefore a manipulated variable $n(k)$ indicates the parallelism degree that will be used for control step k. $n(k)$ takes integer values from 1 to a maximum number N_{max} of available nodes on the remote server.

The model definition for the solver operating part follows the hybrid modeling described in Chapter 5. This is an interesting case to understand *how performance models of structured parallel computations can be composed with the hybrid model of the operating part*. We know from Chapter 4 that the ideal service time of a task-farm can be expressed by the following static performance model:

$$T_{farm}(k) = \max\left\{T_E, \frac{T_W}{n(k)}, T_C\right\}$$

where the ideal service time $T_{farm}(k)$ at control step k is the maximum between the mean service time of the emitter, the collector and the service time of a generic worker divided

by the actual parallelism degree. The worker service time T_W is equal to the mean calculation time T_{calc} per task of the forecasting computation, which we suppose fixed throughout the execution. This static performance model can be casted into a dynamical model of the QoS. The following model (already used in existing work as [101]) expresses the future evolution of the state variable of the operating part model:

$$q(k+1) = \max\left\{0, q(k) + \left(\frac{\tau}{T_A(k)} - \frac{\tau}{T_{farm}(k)}\right)\right\} \tag{6.1}$$

where the length of the queue at the beginning of the next control step $k+1$ is obtained by the last queue length $q(k)$ adding the difference between the number of (predicted) arrivals during the k-th control step and the potential number of served tasks in the same step (τ is the control step length). The max function constrains the queue length to assume only positive or null values.

6.1.2.1 Applying the Reactive Control Strategy

The first strategy is based on a purely reactive approach (see Section 5.3.1). In this case the ParMod model is not used for predicting future system behaviors, but reconfiguration decisions are taken when specific conditions are verified. In particular in this example we consider the utilization factor of the solver calculated with the actual measurements obtained by its operating part. The utilization factor at the beginning of control step $k+1$ is given by:

$$\rho(k+1) = \frac{T_{farm}(k)}{T_A(k)}$$

The utilization factor is the ratio between the mean service time of the task-farm and its mean inter-arrival time, which expresses an average measure of the congestion degree of tasks to the ParMod. Higher values (i.e. greater than 1) of the utilization factor imply a more intensive congestion degree of the solver: i.e. its assigned resources are fully utilized, but the parallel module is not able to produce the results at the same frequency of arrivals from the generator. In this situation the control part can try to increase the parallelism degree (acquiring new computing resources) in order to improve its effective service rate (and also the system operating cost). The opposite case is the situation in which ρ is less than 1, which implies an under-utilization of the assigned resources. In fact the ParMod would be able to process the input tasks at a frequency which would theoretically be higher than the arrival rate. Therefore the control logic can decide to reduce the parallelism degree (releasing computing resources), also reducing the system operating cost. The reactive adaptation strategy behaves as follows:

- at the beginning of each control step k the control part receives the updated value of the utilization factor from the operating part. This value indicates the resource utilization degree during the previous step $k-1$;
- the event generator is responsible for evaluating a fixed set of logical predicates over the current value of the utilization factor. For this example we introduce three

condition variables (see Section 5.3.1) that represent three mutually exclusive conditions:

$$\begin{aligned} \xi_0: &\quad \rho(k) &< T_1 \\ \xi_1: T_1 \leq &\quad \rho(k) &\leq T_2 \\ \xi_2: &\quad \rho(k) &> T_2 \end{aligned}$$

The first condition variable is true iff in the last observed control step the ParMod utilization factor was smaller than a fixed threshold T_1. The second one is true iff the utilization factor was inside the interval $[T_1, T_2]$ and the third one considers the situation in which the utilization factor was greater than a second threshold T_2, where $T_1 < T_2$;

- the values of the condition variables indicate a specific input symbol of a finite-state automaton controlling the ParMod behavior. This input symbol triggers a transition in the automaton, which is translated into a corresponding modification of the current parallelism degree for the control step k.

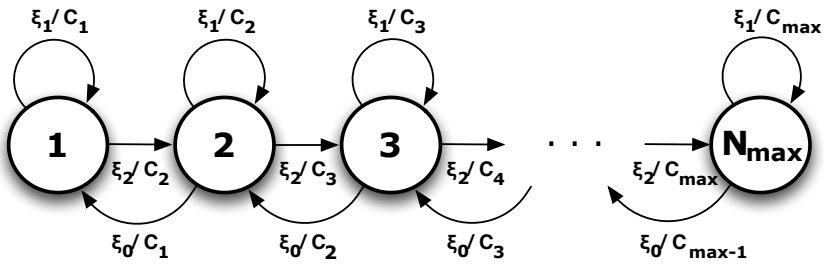


Figure 6.3: Control Automaton of the reactive strategy. Transitions are labeled with E/R, where E is an input symbol and R is an output symbol. C_i indicates the ParMod configuration with parallelism degree equal to i.

In Figure 6.3 is depicted the control automaton which describes the reactive behavior of the control part. We have a distinct internal state for each admissible parallelism degree (N_{max} different internal states). The initial control state corresponds to the starting ParMod configuration, that is the parallelism degree used at the beginning of the execution.

As we can note each control state (except the first and the last one) has three outgoing transitions, triggered by the three condition variables. If ξ_0 is true, the control part reduces of one unit the parallelism degree since the actual utilization factor is smaller than a pre-defined threshold (some resources are under-utilized). The opposite case is when ξ_2 is true: in this case the parallelism degree is increased by one unit since the utilization factor is greater than the second threshold. When ξ_1 is true the eventual under- or over-utilization degree of computing resources is considered acceptable, and the Control Part does not change the parallelism degree used in the previous control step.

6.1.2.2 Applying the Model-based Predictive Control Strategy

The second approach is based on the MPC strategy described in Section 5.3.3. In this formulation we consider a prediction horizon of h control steps, for which a predicted trajectory of disturbance variables is calculated through statistical techniques. At the beginning of the current step k, the control part receives updated measurements from the operating part. Such measurements are related to:

- the actual number of tasks received by the emitter that are waiting to be scheduled (it is the queue length);
- the mean inter-arrival time of tasks during the last control step.

The operating part model represents a central point for applying the MPC strategy. Starting out from the current model state, the control part finds the optimal trajectory of reconfiguration decisions (i.e. parallelism degrees) such that a utility/cost function is properly maximized/minimized. Let us introduce the concept of operational cost for a control step:

$$c(k) = C_{node}\, n(k) + C_{fix}\, \Delta(k) \tag{6.2}$$

C_{node} and C_{fix} are two coefficients representing the cost for using a single node throughout a control step and the fixed cost for each parallelism degree variation. At this regard the variable $\Delta(k)$ indicates if at control step k a non-functional reconfiguration has been executed by the control part:

$$\Delta(k) = \begin{cases} 1 \text{ if } n(k) \neq n(k-1) \\ 0 \text{ otherwise} \end{cases}$$

Therefore the following *cost function* can be introduced, exhibiting proper trade-offs between performance and operational cost objectives:

$$\min J(k) = \sum_{i=k+1}^{k+h} \Big[w_1\, q(i) + w_2\, c(i-1) \Big] \tag{6.3}$$

The control part explores the prediction horizon and selects a trajectory of h parallelism degrees such that the cost function is minimized. The two coefficients w_1 and w_2 indicate the current preference among the two QoS objectives. Assuming that $w_1 >> w_2$ we select a reconfiguration plan such that at the end of the prediction horizon the number of completed tasks is maximized (we try to keep the input queue as empty as possible). However, if multiple plans exist that satisfy this objective, the control part selects the reconfiguration plan with the minimum operational cost, which depends on the number of used computing nodes and of reconfigurations for which a fixed cost should be paid.

Moreover, only the first reconfiguration (for the step k) of the optimal sequence is provided to the operating part and the rest is discarded. Then, the optimization procedure will be repeated at the next control step using the updated results of the ParMod monitoring phase.

Given an accurate prediction of disturbance variables for the whole length of the prediction horizon, the predictive strategy can be an effective approach for several reasons:

- if a persistent drop in the mean inter-arrival time is predicted, the number of computing resources used by the solver ParMod needs to be increased in order to respond to harder workload conditions. In this case the predictive strategy can modify in advance the ParMod configuration to face with lower inter-arrival times and, thus, minimizing the number of required reconfigurations (i.e. the parallelism degree can be directly adapted to the optimal value);

- if the task inter-arrival time is expected to be higher in the future steps, the control part may decide to release a proper amount of computing nodes, thus reducing the parallelism degree of the computation and also its operating cost. If we are able to estimate how long the inter-arrival time will be greater than a specific threshold, the control part can evaluate if the resources release is effectively useful;

- in the case of temporary fluctuations of the mean inter-arrival time, the solver ParMod can avoid to acquire or release computing resources repeatedly, that implying higher operating costs without a significant utility from the performance viewpoint.

6.1.3 Workload prediction: exploiting time-series analysis

In this section we face with the problem of predicting the future trajectory of disturbance variables. As said, in our case we need to predict the mean inter-arrival time of tasks from the generator module. We can experience a time-varying inter-arrival time due to several reasons. For this example we have represented a common situation in which the inter-arrival time may change significantly due to dynamic network availability conditions. In particular we suppose a non-dedicated interconnection network among the endpoint modules, i.e. the generator, the client and the solver ParMod executed on a remote centralized server (see Figure 6.4). Such interconnection network is composed of several heterogeneous links, routers and other network facilities shared with different users, applications, providers and institutions. In this context it is impracticable to have specific

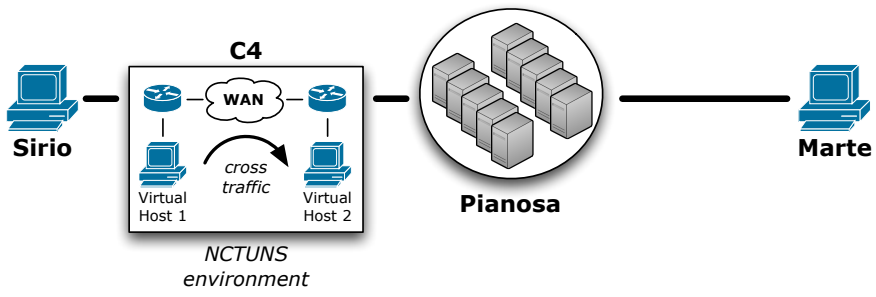


Figure 6.4: Execution platform of the first experiment.

assurances about the reliability of the network, and the provided throughput level of connections among application modules can be highly dynamic. Over the last decades many research activities have been aimed at the definition of accurate techniques for predicting the behavior of a TCP connection between two endpoints. Several promising approaches have been developed based on the historical knowledge of the past network behavior, in order to predict its future behavior. *History-based* predictions exploit methods similar to traditional time series forecasting, where past measurements of an unknown random process are used to predict the future value of the process in the future.

In our example we suppose that the connection between the generator module and the solver ParMod hosted on a remote server, is exploited over an interconnection network featuring dynamic conditions, as the available bandwidth level, packet loss probability (due to router congestion) and delays. Supposing that the generator is able to produce a new task at a fixed rate, the only source of dynamicity of the mean inter-arrival time is due to the network behavior. The average time needed to transfer a large data-structure representing a task (from 4 to 32 MB in our experiments) is influenced by the availability of network resources along the path through the two platforms hosting the application modules. In this context, inter-arrival time historical data for each step of the execution can be viewed as a time-series of past observations featuring several classes of non-stationarities. As stated in [119], three non-stationarities may be observed in the behavior of a TCP connection:

- *Level shifts*: a level shift is an event that affects a time series at a particular time point, causing a significant and typically sudden change in the mean of the observed values. In real network situations, a level shift of the actual network throughput can occur due to load variations or route changes;

- *Outliers*: an outlier is a measurement that is significantly different compared to the typical level of statistical variations relative to nearby measurements;

- *Trends*: a trend is a relatively slow long-term movement in a time series, which changes its current average level along a specific direction (e.g. *upward* or *downward* direction). Trends can be modeled according to different mathematical formulations: e.g. a *linear* trend is the simplest case described as a straight line along several points of the time series values. When different gradients of a trend phase can be appreciated, a *non-linear* trend (e.g. quadratic or cubic) can usually be a more suitable modeling. During network congestion phases, upward and downward trends are typical non-stationary processes of the available TCP throughput: for instance when the congestion level on a path is aggravating, the throughput of each flow along the path will be reduced correspondently.

In order to deal with such non-stationarity processes, we have emulated stochastic network conditions by using proper tools in our experimental architecture. For these reasons the generator and the client modules are executed on two dedicated workstations (hostnames *Sirio* and *Marte*) and the solver ParMod is mapped onto a cluster (namely *Pianosa*)

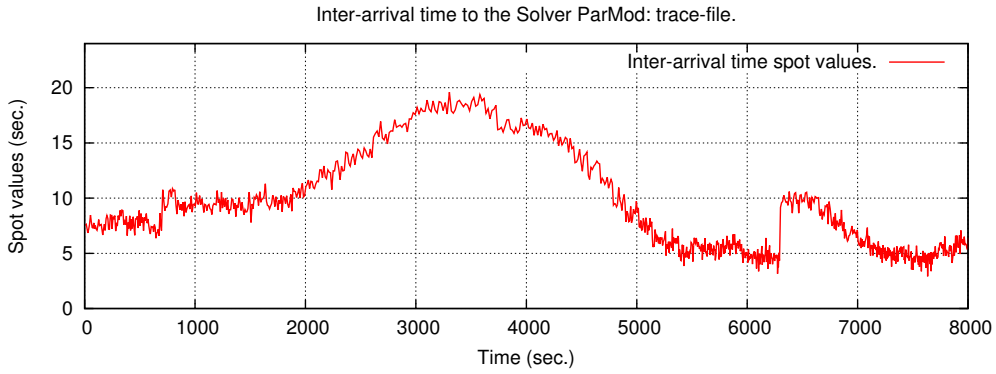


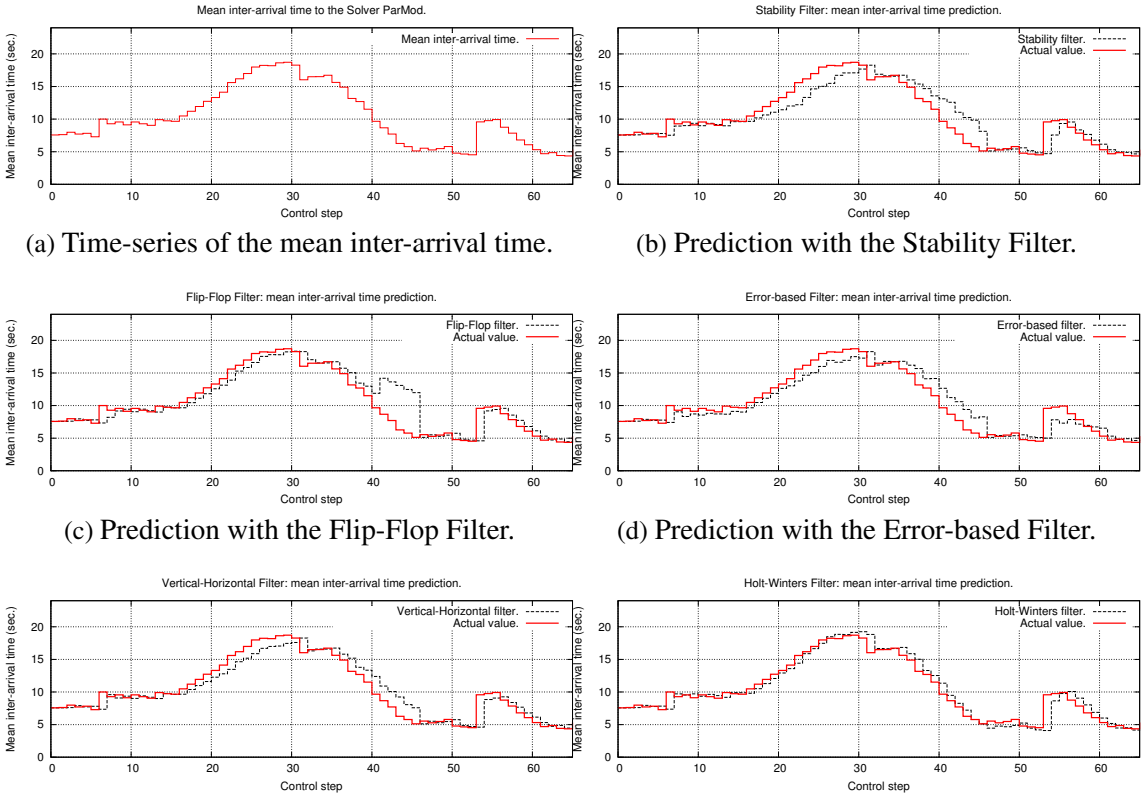
Figure 6.5: Trace-file example of the inter-arrival time, considering tasks of a fixed size of 32 MB.

of production computers interconnected through a dedicated Fast Ethernet network. In order to simulate a dynamic network, we have exploited a network emulator/simulator namely NCTUNS (see [120]), which allows the integration of simulated network topologies with real hosts running application components (see Figure 6.4). NCTUNS is executed on a specific workstation (C_4) and the entire network traffic from Sirio (hosting the generator) to Pianosa (hosting the solver ParMod) is redirected to the NCTUNS node. Inside the the host C_4 a network topology is simulated, composed of two routers and a WAN object reproducing classic wide-area network delays, packet loss and re-ordering probabilities. In order to evaluate the execution scenario under different network conditions, two virtual hosts have been used in order to generate cross network traffic.

Realistic network traffic is generated by using the D-ITG [121] traffic generator, which has been executed among the two virtual hosts. This traffic generator is able to produce realistic TCP and UDP traffic flows, alternating unloaded phases, in which the network is reliable and performs well, and loaded phases in which we experience an increasing of the inter-arrival time measurements. We have generated many different inter-arrival time traces, each one made up of ~ 900 samples during a total execution of 130 minutes. A notable trace-file is depicted in Figure 6.5 for task of size 32 MB.

In this trace-file we experience several non-stationarities of the inter-arrival time process. Three significant level shifts can be detected at time $t \simeq 800$ sec., $t \simeq 3700$ sec. and $t \simeq 6300$ seconds. Moreover the time-series is also characterized by different upward and downward trends. For instance during the time period $t \in (1900\,sec. - 3000\,sec.)$ an upward trend is experienced, corresponding to an intensive generation of cross traffic between the two virtual hosts, whereas during the time interval $t \in (3600\,sec. - 5400\,sec.)$ a downward trend corresponds to a slow network re-start. Based on the previous trace-file, in Figure 6.6a is depicted the average level assumed by the inter-arrival time of tasks during the whole duration of each control step, of a fixed size τ equal to 120 seconds.

In order to predict the mean inter-arrival time for a limited prediction horizon of few control steps, we have to apply proper statistical filtering techniques to our time-series data. In the following we describe the prediction results of five different filters applied



(a) Time-series of the mean inter-arrival time.

(b) Prediction with the Stability Filter.

(c) Prediction with the Flip-Flop Filter.

(d) Prediction with the Error-based Filter.

(e) Prediction with the Vertical-Horizontal Filter.

(f) Prediction with the Holt-Winters Filter.

Figure 6.6: Forecasting techniques applied to the mean inter-arrival time of tasks.

to the historical time-series of the mean inter-arrival time shown in Figure 6.6a. The first four filters are specific variants of the *Auto-Regressive Moving Average* (ARMA) approach, defined by the following general structure:

$$\hat{T}_A(k) = a \cdot \frac{1}{z} \sum_{i=k-z}^{k-1} \Big[T_A(i) \Big] + (1-a) \cdot T_A(k-1) \tag{6.4}$$

The prediction at control step k is exploited by using the average value assumed by the inter-arrival time in a history window of the past z control steps and the observed mean inter-arrival time during the last measured control step $k-1$. This two components, that is the auto-regressive and the moving-average part of the filter (of order z and 1 respectively), are properly weighed through a positive smooth factor (*gain*) $a \in [0,1]$. The gain definition is a central point for tuning the accuracy of these predictive techniques. Greater values correspond to a more "stable" filter in which old estimates dominate the predicted values. Smaller gains, by contrast, induce very "agile" filters that promptly adapt to the last values assumed by the observed measurement. Better results are usually achieved by exploiting a dynamic adaptation of the gain. Some interesting techniques [122], especially fruitful for network predictions, have been formulated for *exponentially-weighted moving*

average (EWMA) filters. In our case we have applied these techniques for dynamically tuning the gain a of the ARMA model presented in (6.4).

Stability Filter: the gain is adapted to the variance of the time-series values. If a high variance is experienced in the last measurements, the gain is increased in order to obtain a better stability, otherwise if the measured variance is relatively small, the gain is properly decreased. For estimating the variance of past measurements, a further ARMA filter (with a static gain $b = 0.6$) is used for estimating the difference between two subsequent measurements, i.e. $U(k) = T_A(k) - T_A(k-1)$. The gain $a(k)$ at control step k is dynamically calculated by taking the following ratio: $a(k) = \hat{U}(k)/U_{max}$, where U_{max} is the maximum difference between two subsequent measurements experienced in a limited window of past observed values.

Error-based Filter: the gain is adapted w.r.t the quality of the predictions calculated so far. A further ARMA filter (with a static gain $b = 0.6$) is used for predicting the absolute error between the predicted measurement at time k and the corresponding real value: i.e. $E(k) = |\hat{T}_A(k) - T_A(k)|$. The gain $a(k)$ at control step k is calculated by taking the following expression: $a(k) = 1 - (E(k)/E_{max})$ where E_{max} is the maximum error in a limited window of past measurements.

Flip-Flop Filter: this technique uses two ARMA filters with different static gains. The first one is agile, with a gain of 0.1, the other is stable with a gain of 0.9. A control law selects between the two filters by using the agile one when it is possible, but falling back to the stable filter when observations fluctuate intensively. The control law is based on a *control chart*. A control chart plots the sample mean of a controlled quantity against the desired population mean. The chart contains a center line that represents the mean value and two horizontal lines, called the upper control limit (UCL) and the lower control limit (LCL). These control limits are adjusted in such a way that almost all of the measurements will fall within them during the process. The limits are usually defined by using a specific rule concerning the standard deviation: e.g. $\mu \pm 3\sigma$ (a.k.a 3σ rule), where σ is the sample standard deviation and μ is the population mean. The standard deviation is not known in advance, thus it is approximated by the *mean range* i.e. the average difference between two consecutive measurements. If the process is inside the current upper and lower control limits, the stable filter will be exploited, otherwise we will use the agile one.

Vertical-Horizontal Filter: this filter is an adaptive ARMA filter in which the gain a is dynamically adjusted during the prediction process according to a fixed law given below:

$$a(k) = \frac{0.3\,\Delta_{max}}{\sum_{i=k-M}^{k} |T_A(k) - T_A(k-1)|}$$

where Δ_{max} is the difference between the maximum and the minimum values in the M most recent observations (i.e in our experiments $M = 10$ is the value giving the

best results). This law has been originally proposed in [123] for an EWMA filter, and in our case we have adapted it to an ARMA filter.

Non-Seasonal Holt-Winters Filter: the last exploited filter uses a different prediction approach based on a simple EWMA model that attempts to capture the trend in the underlying time series. Two different EWMA filters are used, the first one for estimating the smooth component of the predicted value, and the second one for predicting the trend component.

$$\begin{aligned} \hat{T}_A(k) &= \hat{s}(k) + \hat{t}(k) \\ \hat{s}(k+1) &= a \cdot T_A(k) + (1-a) \cdot \hat{T}_A(k) \\ \hat{t}(k+1) &= b \cdot \left[\hat{l}(k) - \hat{l}(k-1)\right] + (1-b) \cdot \hat{t}(k-1) \end{aligned} \tag{6.5}$$

For this experiment the static gains a and b have been fixed to 0.9 and 0.2 respectively, that give the best prediction results with the time-series depicted in Figure 6.14a.

In Figures 6.6b, 6.6d, 6.6c, 6.6e and 6.6f are shown the predictions of the mean inter-arrival time for each controls step by applying the five different filtering techniques. In the figures the dashed lines are the predicted values, whereas the solid red line is the real average value during each control step.

The different techniques can be compared according to a notion of accuracy, e.g. the *root mean square relative error* (RMSRE) defined as:

$$RMSRE = \sqrt{\frac{1}{n}\sum_{i=1}^{n} E(i)^2} \tag{6.6}$$

where $E(k)$ is the relative error between the predicted and the real measurement at control step k and n is the total number of samples in the time-series. The accuracy of the previous filters is depicted in Figure 6.7. As we can seen the Holt-Winters approach is the best

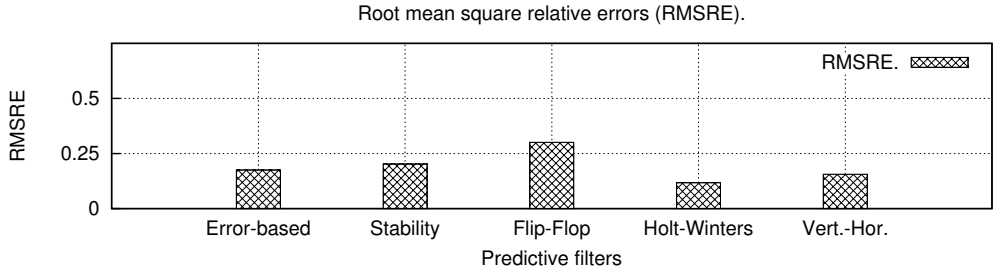


Figure 6.7: RMSRE of the different filtering techniques.

method, providing an average relative error of about 10%. This is due to the fact that this filter is the only one that tries to predict a trend component in the predicted values. These filters can be used for exploiting a multiple-step ahead prediction. In this case the last predicted value is used as the new measured value, and the filters are iteratively applied for all the steps of the prediction horizon.

6.1.4 Implementation of non-functional reconfigurations

In this section we briefly introduce some realization details about the experiment implementation. The whole computation graph depicted in Figure 6.1 has been implemented by a set of distributed processes executing the specific application phases. Each computation module consists in a MPI (Message Passing Interface [40]) program executed on the underlying computing architectures. Communications between generator, solver and client components have been implemented by using the standard POSIX socket library exploiting TCP connections. The whole network traffic between the generator and the solver ParMod has been redirected to the NCTUNS node through a proper configuration of the routing tables of Sirio (hosting the generator) and of the cluster front-end node.

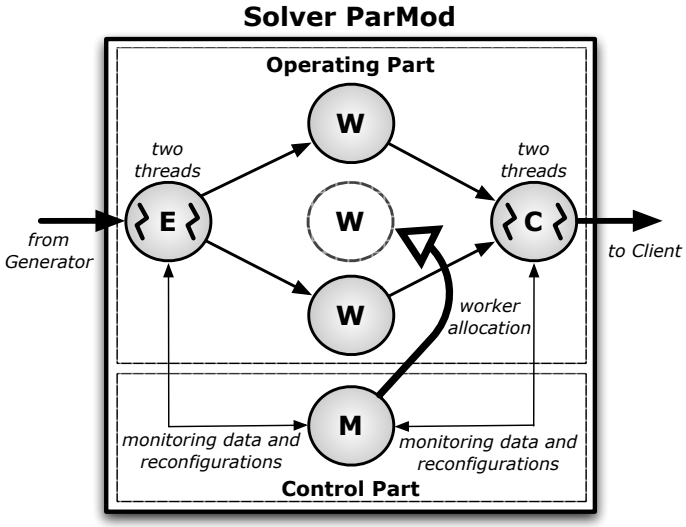


Figure 6.8: Distributed processes implementing the Solver ParMod.

However the most important implementation issues have been addressed for developing the solver ParMod. In Figure 6.8 is depicted the network of processes implementing the adaptive parallel module. The solver ParMod is a MPI program featuring a proper set of processes communicating through MPI send/receive primitives. The emitter is responsible for scheduling each in-coming task to an available worker performing an on-demand distribution. The emitter is composed of two threads: the first one receives tasks from the generator and collects them into a queue buffer; the second one consumes from the queue buffer and schedules each task to an available worker. The first thread is also responsible to periodically transmits the queue length to the control part (at each control step through a timer), and the actual mean inter-arrival time (by measuring the time spent in receiving each tasks from the generator).

The reactive and the predictive adaptation strategies are executed by a proper centralized manager process (*M* in Figure 6.8) that constitutes the ParMod control part. The parallelism degree variation is exploited by a manager process through the MPI library

function `MPI_COMM_SPAWN`, that instantiates a new set of processes executing the worker functionality (a main function identical between the workers). The new worker identifiers are finally passed to the emitter and the collector by the manager process.

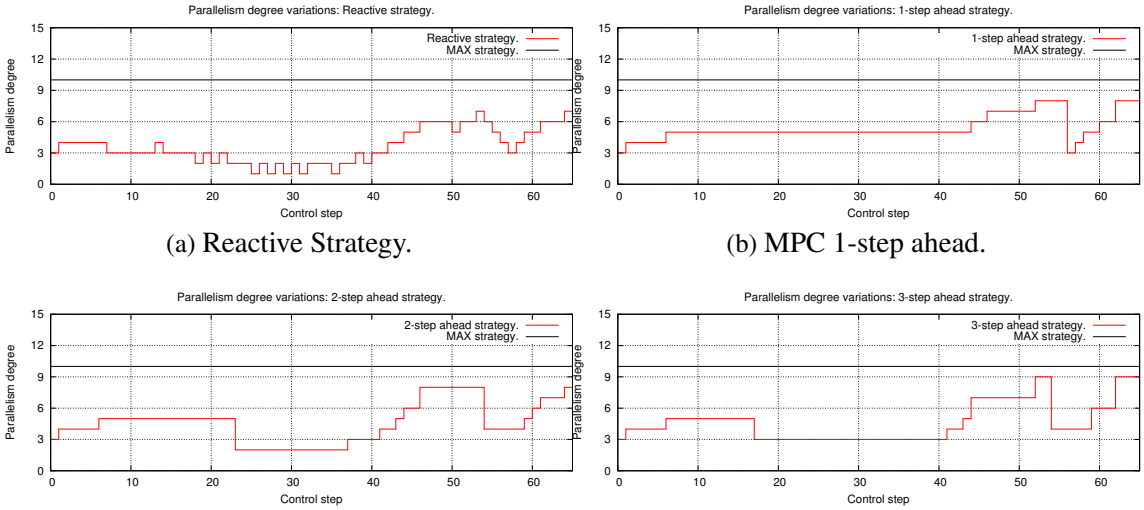
6.1.5 Experimental results and comparison between Control Strategies

We conclude this first example by providing experimental results and the comparison between the two different adaptation strategies. The experiment has been developed by assuming a fixed task size equal to 32 MB, that requires a completion time per task of ~ 28 seconds on our cluster composed of 15 homogeneous workstations. The reference execution scenario consists in a computation of 67 control steps each of length 120 seconds that characterizes a total execution time equal to ~ 130 minutes.

Thresholds	Reconfigurations	Completed Tasks
$T_1 = 0.9$ $T_2 = 1.2$	38	874
$T_1 = 0.9$ $T_2 = 1.3$	24	845
$T_1 = 0.9$ $T_2 = 1.4$	19	817
$T_1 = 0.9$ $T_2 = 1.5$	19	789
$T_1 = 0.9$ $T_2 = 1.6$	19	769
$T_1 = 0.9$ $T_2 = 1.7$	11	751

Table 6.1: Parameter configuration of the Reactive Strategy.

The reactive and the MPC approaches will be compared with a strategy in which, for each control step of the execution, we fix the parallelism degree of the solver ParMod to the maximum admissible value. This static solution (that we call *MAX* strategy) optimizes the performance, allowing the solver ParMod to complete the maximum number of tasks during the whole execution time, but it induces an unacceptable operational cost (many resources are often under-utilized). In our experimental case this strategy makes it possible to complete 946 tasks, corresponding to distinct points of the flood model. Ideally we require a control approach that reaches a performance level as similar as possible to the one of the MAX strategy but, at the same time, also optimizing the operational cost of the execution. The first adaptation strategy that we have presented is based on a reactive logic. In this approach the degree of reactivity of the solver ParMod depends on the definition of the two thresholds T_1 and T_2 that specify the conditional variables. In Table 6.1 are presented the results of different configurations of these parameters. From Section 6.1.2.1 we know that if the utilization factor ρ is greater than the second threshold T_2, the control part is encouraged to acquire new resources by increasing the parallelism degree. From the previous table is highlighted that increasing the threshold T_2 entails a smaller number of parallelism degree variations but also a smaller number of completed tasks since the ParMod is less prompt to react to a decrease of the mean inter-arrival



(a) Reactive Strategy. (b) MPC 1-step ahead.

(c) MPC 2-step ahead. (d) MPC 3-step ahead.

Figure 6.9: Sequence of reconfigurations of the distinct adaptation strategies compared to the MAX strategy.

time. In the table the best reactive configuration for maximizing the number of completed tasks is with $T_1 = 0.9$ and $T_2 = 1.2$, that results in 874 completed tasks (72 less than the MAX strategy). Furthermore, with this adaptation strategy the control part performs 38 non-functional reconfigurations during the entire execution (as shown in Figure 6.9a).

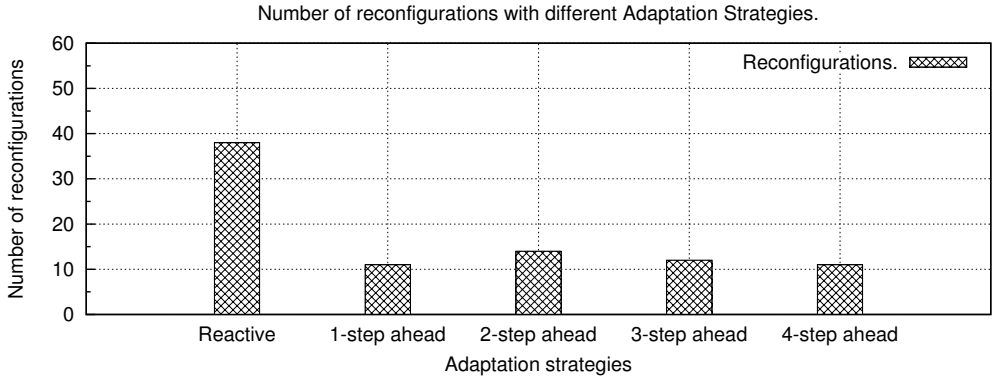


Figure 6.10: Number of parallelism degree variations.

The second reconfiguration strategy is based on the MPC control scheme. The controller predicts the behavior of the mean inter-arrival time for a limited prediction horizon and selects the best sequence of reconfiguration decisions such that the cost function (6.3) is minimized. As stated in the sections before, our objective is to optimize the number of tasks completed during the entire execution and trying at the same time to minimize the operational cost. For this reason we compare the results of the MPC approach to the reactive strategy with $T_1 = 0.9$ and $T_2 = 1.2$. For the predictive strategy we have considered

four different lengths of the prediction horizon equal to 1, 2, 3 and 4 control steps. The parallelism degree variations are shown in Figure 6.9b, 6.9c and 6.9d, whereas the number of completed tasks is described in the Table 6.2. As we can see the predictive approach

Adaptation Strategy	Completed Tasks
MAX	946
Reactive(0.9, 1.2)	874
MPC 1-step ahead	944
MPC 2-step ahead	945
MPC 3-step ahead	944
MPC 4-step ahead	944

Table 6.2: Number of completed tasks with different adaptation strategies.

is able to complete more tasks than the reactive one during the same execution time and with an identical inter-arrival time fluctuation. In this case the number of completed tasks is similar to the one obtained by fixing the parallelism degree to the maximum number of available computing nodes (we lose one or two tasks which is a negligible loss). Moreover, the predictive approach has also a positive impact in the number of reconfigurations and on the operating cost of the execution. In Figure 6.10 is depicted the number of reconfigurations with the different adaptation strategies. For every prediction horizon length, the MPC strategy always features a lower number of reconfigurations than the reactive approach.

It is important to describe the results from the operational cost viewpoint. For this experiment we have supposed a fixed cost for each parallelism degree variation (i.e. C_{fix}) which is five times greater than the unitary cost for using a computing node for a control step (i.e. C_{node}). We consider the total operating cost of the entire application, given by the sum of the cost described in expression (6.2) for each control step of the execution. The total cost is a valuable metric to understand when, during the execution, a certain adaptation strategy starts to be more or less effective than the others. The actual value of the total cost at each control step is depicted in Figure 6.11.

In this figure the importance of having a dynamic adaptation of the solver ParMod is clearly highlighted. In fact, compared to the MAX strategy (the red line in Figure 6.11), any adaptation strategy makes it possible a significant reduction of the total operating cost. The percentage reduction w.r.t the MAX strategy of the total operating cost at the end of the execution is shown in Figure 6.12. From the experimental results it emerges that the predictive strategies are able reduce the cost of even the 60% compared to the MAX strategy. Moreover they produce a further improvement w.r.t the reactive adaptation of the 20% with the MPC 3-step ahead. On the other hand in this example having a horizon length greater than three steps is not useful, since the accuracy of the prediction filter degrades with longer prediction horizons. In fact with a prediction horizon of four control steps (not shown in the figure for representation clarity), the operational cost is similar than with a three-step horizon without any additional completed task.

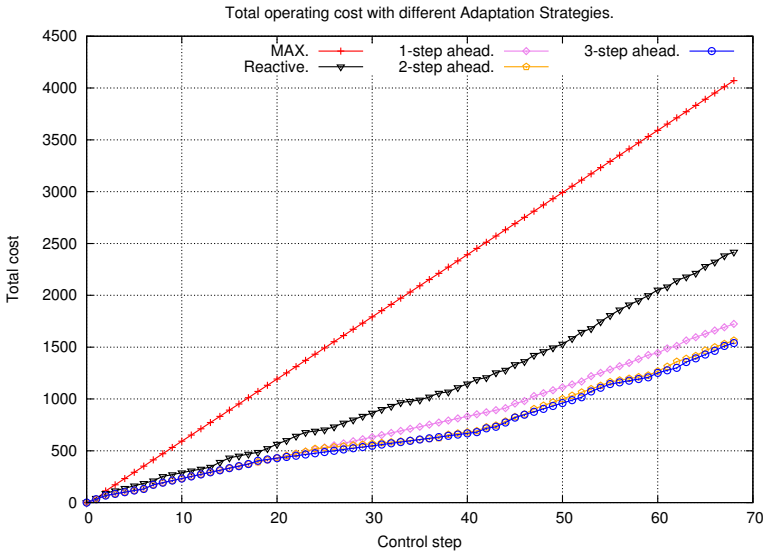


Figure 6.11: Total operational cost with different adaptation strategies.

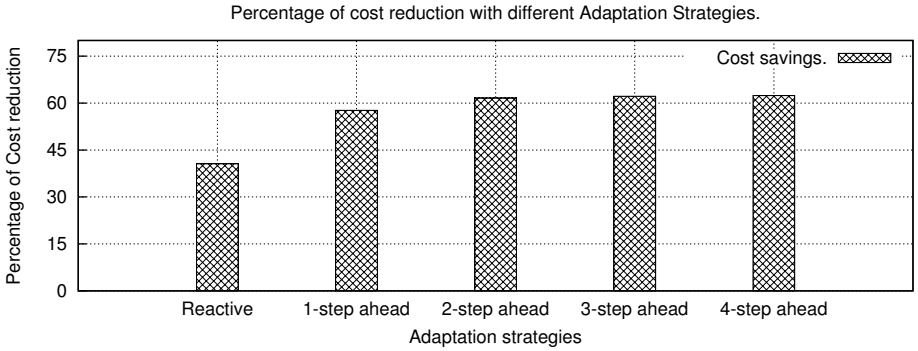


Figure 6.12: Operational cost reduction w.r.t MAX strategy.

In conclusion this first example shows the importance of having a dynamic adaptation of a parallel module for meeting performance and operational cost constraints. Moreover in this example we have shown that the MPC approach is a valuable solution that significantly reduces the operational cost of the execution maintaining the number of completed tasks at a very similar level to the one of the optimal MAX strategy.

6.2 Dynamic adaptation of a Digital Image-Processing application

In this section we introduce a second experiment concerning the exploitation of the model-based predictive control strategy for an image-processing application featuring a client-server interaction among several computation modules. In this scenario we consider the presence of a ***Server*** ParMod which receives input tasks from a set of ***Clients***.

The interaction pattern between clients and the server follows a request-reply behavior: i.e. each client generates a task which is transmitted to the server and then waits for the explicit reception of the corresponding result.

A task contains a variable size high-resolution digital image (e.g. acquired by CCD cameras of a surveillance system) that can be contaminated by noise effects during image acquisition and transmission. Noise is a random variation of brightness or color information that can severely degrade the image quality and cause the loss of important graphic details. Noise elimination is a preliminary activity in many image processing applications: noise reduction filters need to be applied before more complex phases as edge detection, image segmentation and object recognition.

For the scope of this experiment we suppose that the server component performs on each received image a noise reduction algorithm. Our application needs to be configured in order to achieve the following QoS constraints:

- in applications like surveillance systems, a large sequence of input images are processed in a real-time fashion, applying complex statistical algorithms in order to identify potentially dangerous events and send alert messages to the users. For this reason we suppose that the mean response time of the server component (i.e. the average time from the transmission of a task from a client to the reception of the corresponding filtered result) needs to be maintained within a maximum and minimum acceptable threshold;

- in order to minimize the probability of system failure and malfunctions, it is further requested to optimize the "stability degree" of the server behavior. In other words this component performs the strictly necessary number of functional and non-functional reconfigurations throughout the execution.

Due to the time-varying size of the received images, the server ParMod should perform a proper adaptation strategy in order to meet the QoS objectives. In this example we will discuss the exploitation of the MPC strategy.

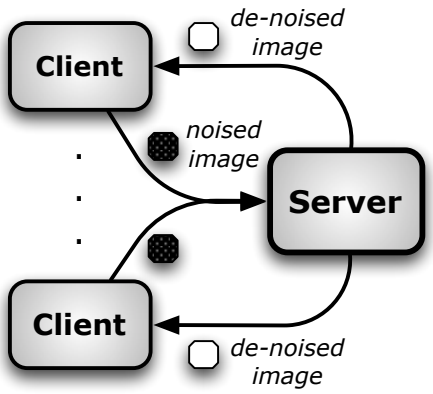


Figure 6.13: Digital Image-Processing application: client-server computation graph.

The computation graph of this second experiment is shown in Figure 6.13. As we can see each client transmits tasks containing a noisy image to be filtered. Clients are computation modules executed on a heterogeneous set of computers, whereas the server, due to the response time constraint, needs to be executed on a parallel architecture. For this experiment we have used a cluster of 15 homogeneous production workstations.

6.2.1 Image Filtering techniques and Parallelization

Various filtering techniques have been proposed for removing impulse noise. In this experiment we will exploit two well-known basic filters:

- **Mean filter**: the mean filter is a simple type of low-pass linear filter for smoothed images. It consists in replacing each pixel with the weighted average value of all the pixels in a square window surrounding the target pixel;
- **Median filter**: the median filter is a nonlinear filtering technique. The main idea is to replace each pixel of a 2D image with the median of its neighboring entries. The median is calculated by sorting all the pixel values from the surrounding neighborhood (i.e. a square window centered in the examined pixel) into numerical order (e.g. based on the calculated brightness) and then replacing the pixel being considered with the middle pixel value.

These two simple filters have some pros and cons. The mean filter is able to suppress isolated out-of-range noise, but it blurs sudden changes of the source image like sharp edges. In contrast the median filter preserves the image edges while removing noise, but at the cost of a more computationally intensive calculation. In fact for each pixel we have to order a set of neighbor pixels (proportional to the radius of the window) and to select the median one. Even if the median filter is preferable in general, the server ParMod can switch to the simplest mean filter whenever this choice is essential to respect the response time constraints. Therefore we consider two different alternative operations for the server ParMod:

- a ***task-farm*** parallelism scheme of the mean filter. In this case each received task is scheduled according to a load-balanced strategy (on-demand) to an available worker that performs the filtering on the whole image. A collector receives the filtered images from workers, and it is responsible for transmitting the results to the corresponding clients;
- a ***map*** data-parallel scheme of the median filter. Each received image is sliced in different partitions (with proper overlapping regions for border pixels) which are independently filtered by a set of workers. The filtered image is reconstructed by a gather process that transmits the result to the corresponding client.

During the execution the server control part can change the current operation (task-farm of the mean filter or the map of the median filter), and also the parallelism degree of the

used version. Thus in this experiment we consider both functional and non-functional reconfigurations of the adaptive parallel module.

6.2.2 Operating Part model and Control Strategy

In order to properly select the right reconfigurations, we define a model able to predict the QoS behavior of the server component. We can identify the following set of model variables:

- the average size $M(k)$ of tasks (images) received from clients during a control step k, represents an uncontrollable disturbance that can significantly influence the response time of the server;
- in this example the interesting QoS variable is the mean response time $R_q(k)$ experienced at the beginning of control step k. For the entire execution this measure has to be maintained between two thresholds: i.e. Rq_{max} and Rq_{min};
- two control variables indicate the functional or non-functional reconfigurations exploited during control step k. The parallelism degree is indicated by a variable $n(k)$ assuming integer values from 1 to N_{max}, whereas the adopted operation, i.e. the task-farm of the mean filter (0) or the data-parallel of the median filter (1), is selected through a variable $op(k) \in \{0,1\}$.

As stated in the Section 4.1.2, the performance behavior of a client-server parallel computation can be predicted through a static performance model consisting in the following system of equations:

$$\begin{cases} T_C(k) = T_G + R_q(k+1) \\ R_q(k+1) = W_q(k) + L_s(k) \\ \rho_s(k) = \dfrac{T_S(k)}{T_A(k)} \\ T_A(k) = \dfrac{T_C(k)}{N_{client}} \end{cases} \tag{6.7}$$

T_G indicates the average time for generating a new request to the server (client ideal service time), whereas $T_C(k)$ corresponds to the effective client performance obtained by adding to T_G the current mean response time of the server. For simplicity in this experiment we will consider a fixed average value throughout the execution for T_G and for the number of connected clients N_{client}.

This model allows the server control part to predict its mean response time. This prediction requires to know the mean service time $T_S(k)$ and the mean computation latency $L_s(k)$ of the server during the k-th step. These two performance parameters can be quantified through the performance models of the different ParMod configurations. First of all we need to evaluate the communication latency (or communication time also denoted

by L_{com}) on the underlying distributed-memory architecture (i.e. the server cluster and the workstations of client modules). The time spent for transmitting/receiving a message depends on the message size and on the architectural features of the interconnection network. For a first approximation we can use the following linear relationship:

$$L_{com}(size) = t_{startup} + size \cdot t_{trasm} \tag{6.8}$$

The communication latency is given by two terms: $t_{startup}$ is the time spent to initiate the communication, t_{trasm} is the transfer time to send one data word (e.g. of one byte) and *size* is the number of data words that compose a message. As we will discuss in the sequel, this model will give a reasonably accurate approximation of the communication cost particularly for large messages.

Task-Farm version	**Data-Parallel version**
$Tcalc(k) = T_H \cdot \frac{M(k)}{3}$	$Tcalc(k) = T_F \cdot \frac{M(k)}{3}$
$T_S(k) = \max\left\{2L_{com}\big(M(k)\big), \frac{Tcalc(k)}{n(k)}\right\}$	$T_S(k) = \max\left\{L_{com}\big(M(k)\big) + T_{scatter}(k), \frac{Tcalc(k)}{n(k)}\right\}$
$L_S(k) = 4L_{com}\big(M(k)\big) + T_{calc}(k)$	$L_S(k) = 2L_{com}\big(M(k)\big) + 2T_{scatter}(k) + \frac{T_{calc}(k)}{n(k)}$
	$T_{scatter}(k) = n(k) \cdot L_{com}\left(\frac{M(k)}{n(k)}\right)$

Table 6.3: Service time and computation latency of alternative Server configurations.

At this point we can describe in more detail the service time $T_S(k)$ and the computation latency $L_S(k)$ of the alternative server configurations. The results are summarized in Table 6.3. The mean calculation time is shown in the first row of the table. Given the mean task size $M(k)$ (expressed in bytes) for the control step k, the calculation time is obtained by multiplying the number of image pixels (each one coded in three bytes) for the unitary calculation time per pixel which is:

- for the task-farm version T_H corresponds to the time needed for the calculation of the mean value of the neighbor pixels (assuming a fixed window size);
- for the data-parallel version T_F corresponds to the time needed for sorting the neighbor pixels and selecting the median one (assuming also in this case a fixed window size).

The service time and the computation latency of the task-farm and of the map version are expressed by instantiating the performance models described in Section 4.2.1 and 4.2.2. As it is known the map version of the median filter, though starting from a higher calculation time, is able to reduce both the service time and the computation latency. On the other hand the task-farm version is able to reduce only the service time of the server ParMod, but starting from a lighter filtering algorithm. We can observe that in this model

we have exploited a set of assumptions that can be pessimistic in a real context. We have assumed the same communication time for the scatter and the gather phases of the data-parallel version and we have considered the same communication latency for intra-cluster communications (between processes inside the server operating part) and inter-cluster communications (between clients nodes and the server cluster). This last assumption will not induce relevant inaccuracies, since in our test-bed platform client workstations and cluster nodes are part of the same LAN interconnected through a Fast Ethernet network technology.

The previous model is used for selecting a trajectory of reconfigurations (operations and parallelism degrees) that optimizes a proper cost function. In this example we are interested in maintaining the mean response time of the server between two provided thresholds and minimizing the number of exploited reconfigurations. In this scenario reconfigurations are necessary operations due to the time-varying average size of received tasks.

The formulation of the optimal control problem is stated as follows: we introduce two variables $G(k)$ and $\Delta u(k)$. The further is a measure of how much the response time constraint is satisfied. If the predicted mean response time is greater than Rq_{max} (or less than Rq_{min}), the variable is equal to the corresponding relative error between the actual predicted value and the maximum (minimum) threshold. Otherwise, it is zero for any response times between the two threshold values.

$$G(k) = \begin{cases} 0 \quad \text{if } Rq_{min} \leq R_q(k) \leq Rq_{max} \\ \dfrac{R_q(k) - Rq_{max}}{Rq_{max}} \quad R_q(k) > Rq_{max} \\ \dfrac{Rq_{min} - R_q(k)}{Rq_{min}} \quad R_q(k) < Rq_{min} \end{cases}$$

The second variable indicates if at the beginning of the control step k a reconfiguration has been taken:

$$\Delta u(k) = \begin{cases} 0 \quad \text{if } \mathbf{u}(k) = \mathbf{u}(k-1) \\ 1 \quad \text{otherwise} \end{cases}$$

At this point the cost function is defined as follows:

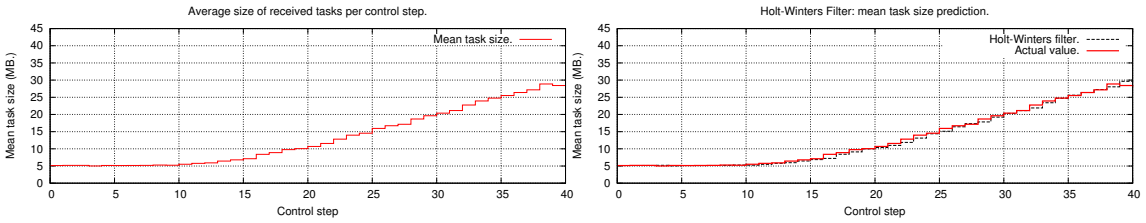
$$\min J(k) = w_1 \cdot \sum_{i=k+1}^{k+h} G(i) + w_2 \cdot \sum_{i=k}^{k+h-1} \Delta u(k) \tag{6.9}$$

Where w_1 and w_2 are two relative weights. If $w_1 >> w_2$, the control part explores the prediction horizon in order to find the best sequence of reconfigurations such that the mean response time is maintained between the two thresholds. If multiple reconfiguration plans

satisfy this requirement, the controller selects the trajectory with the minimum number of control input changes. Based on the predictive control approach, only the first input of the optimal trajectory is applied to the operating part and the rest is discarded. Then the optimization process is repeated at the next control step exploiting the new measurements of disturbance input.

6.2.3 Task size prediction

In order to apply the predictive strategy introduced in the previous section, we estimate the trajectory of disturbance inputs for the prediction horizon. In this experiment the disturbance input is the average size of received tasks from clients. We have simulated an increasing workload to the server ParMod. That is, throughout an execution time of 5 hours, clients transmit images with a size that, in the average case, increases during the execution. In Figure 6.14a is depicted the mean task size $M(k)$ experienced during each control step k of length 300 seconds.



(a) Increasing workload. (b) Holt-Winters prediction.

Figure 6.14: Average size of the images received by the Server ParMod: workload trend and statistical prediction.

Similarly to the first experiment, also in this case we exploit statistical predictive filters in order to estimate the disturbance input trajectory. Since an increasing trend can be clearly identified in Figure 6.14a, we can try to predict the mean task size by using an exponential smoothing filter with a linear trend. For this purpose the non-seasonal Holt-Winters filtering technique is applied to this input time-series. The results are depicted in Figure 6.14b. In this scenario the filtering technique is quite accurate, providing a mean prediction error lower than 5%.

6.2.4 Implementation of functional reconfigurations

The server ParMod has been implemented as a MPI program executed on a cluster architecture of 15 production workstations. The operating part resources, i.e. the emitter, the collector and a set of workers, are able to execute the two distinct operations: the task-farm and the data-parallel version of the mean and the median filter respectively. The implementation scheme is depicted in Figure 6.15.

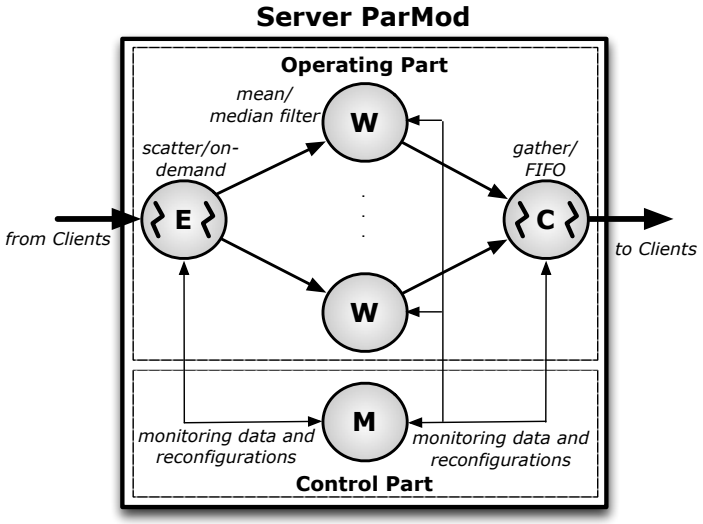


Figure 6.15: Distributed processes implementing the Server ParMod.

Since MPI provides a message-passing model, operating part and control part processes cooperate exchanging messages on proper communication channels. The emitter and the collector resources are composed of two internal threads. The first one performs the the distribution and the collection activities, while the second is responsible for the interconnection with the control part (composed of a unique centralized manager process). At the beginning of each control step, monitored data (in this case the mean task size experienced in the previous step) are transmitted to the controller, which notifies through reconfiguration commands the current version that will be executed. In the case of the task-farm, the emitter schedules each received task according to an on-demand policy to an available worker, otherwise if the current version is the map, the received image is partitioned (scattered) among the worker set. Each worker receives the current task (or a partition of a task) from the emitter and applies the mean or the median filter based on the version notified by the control part. The collector logic is much more complicated. It needs to perform a simple FIFO collection, if the task-farm is executed, or a gather functionality in the case of the data-parallel version. In order to ensure that the switching from the two collection strategies is performed correctly, the collector resource receives an explicit notification from the workers that they are ready to change the operation. After that the collector can change the collection strategy being sure that, e.g. in the case of a gather, all the partial results from the workers have been collected correctly before switching to the FIFO collection.

6.2.5 Results and discussion

First of all we need to evaluate how the performance model for client-server computations behaves quantitatively for approximating a real execution. For this reason we have

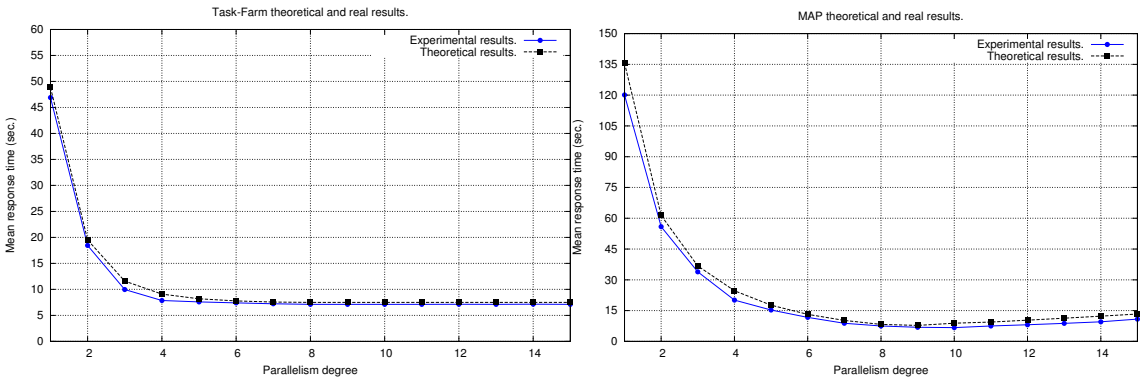
performed an experiment in which the average size of input matrices has been fixed to 6.5 MB, i.e. for each client the task size is a random variable with mean 6.5 MB. The mean generation time T_G for each client is also a random variable with mean 15 seconds, where the client number has been fixed to 10 for the entire duration of the execution. Supposing that:

- the size of a received task at a given time instant is independent of the previously received tasks;
- the time between two subsequent tasks generation from the same client is independent from the previous requests.

we can approximate the average waiting time in queue W_q by using the $M/M/1$ model defined in Chapter 4:

$$W_q(k) = \frac{T_S(k)^2}{T_A(k) - T_S(k)}$$

The resolution of the system (6.7) gives a second order equation in $\rho(k)$ that admits one and only one real positive solution satisfying $\rho(k) < 1$ (which is a necessary steady-state constraint). On each homogeneous computing node of the cluster, the calculation of each pixel has been estimated in $T_H = 2.5 \times 10^{-6}$ sec. for the task-farm version, and $T_F = 6.02 \times 10^{-6}$ sec. for the data-parallel one. In our test-bed network the parameters of the communication cost model (6.8) have been estimated through a linear regression model in $t_{startup} = 1.1 \times 10^{-1}$ and $t_{trasm} = 4.13 \times 10^{-2}$, that provide an acceptable approximation in our cluster network (a Fast Ethernet 100 Mbit/s) for large enough transfered data (more than 256 KB).

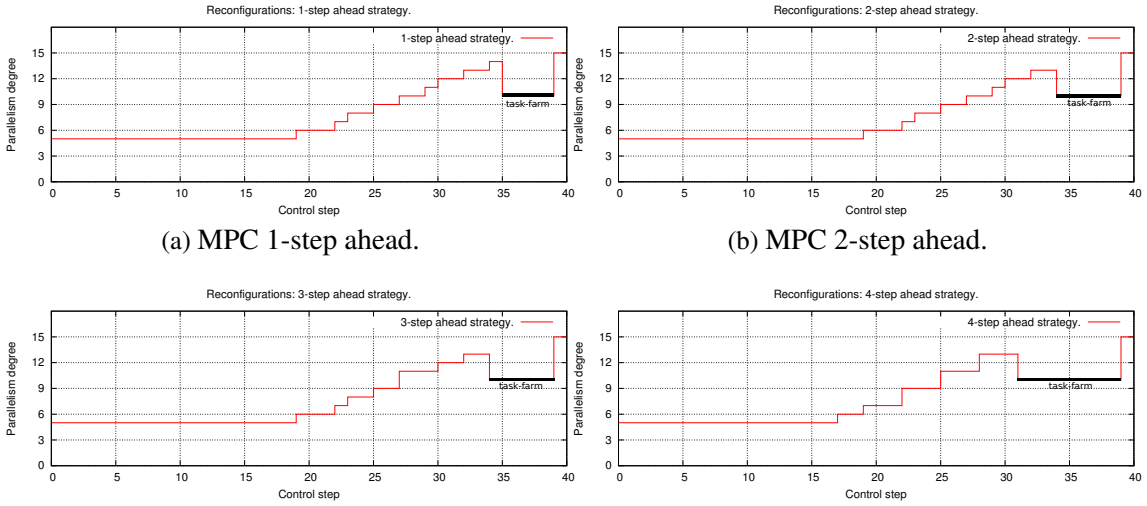


(a) Task-Farm operation. (b) Map operation.

Figure 6.16: Comparison between the mean response time of the model and the real one obtained by experiments.

In Figure 6.16 is depicted the mean response time of the server in function of the parallelism degree (up to 15 nodes, which is the maximum number of available resources in our cluster architecture). The theoretical results obtained with the operating part model

have been compared with the real experienced values. For the task-farm operation (Figure 6.16a), both the experimental results and the theoretical values exhibit a monotonically decreasing behavior. For parallelism degrees greater than the number of connected clients (10), the task-farm operation gives no performance improvement (in this case whenever a request from any client arrives at the server, it can certainly be served by an available worker). As shown in Figure 6.16a and 6.16b the $M/M/1$ queueing model results are an over-estimation of the real experienced ones, since the exponential distribution though its mathematical soundness is not always the best modeling for realistic cases. However the maximum relative error is less than 10% in this experiment and this model will be sufficient for driving an effective server adaptation.



(a) MPC 1-step ahead.

(b) MPC 2-step ahead.

(c) MPC 3-step ahead.

(d) MPC 4-step ahead.

Figure 6.17: Sequence of reconfigurations with different prediction horizon lengths.

The map operation features a different behavior. In this case, changing the server parallelism degree, we expect a non-monotonic behavior of the mean response time. In fact, an increasing in the parallelism degree induces a smaller computation latency of the server (a typical effect of a data-parallel parallelization w.r.t a task-farm one) with a consequently improvement in the mean response time. On the other hand large parallelism degrees cause a greater cost for the scatter process (more transmissions should be paid) which negatively influences the server service time (see Section 4.2.2) and thus the mean waiting time in queue (which can be approximated by $W_q = L_q \cdot T_S$ where L_q is the mean queue length). With parallelism degrees greater than 9 with the theoretical model (10 with the real values), the negative effect on the service time dominates the computation latency reduction, giving slightly worse response times. The real behavior of the experiment if well predicted by the performance model, giving a maximum relative error less that 15% in this experiment.

We have studied the behavior of the predictive control strategy with the time-varying mean task size behavior depicted in Figure 6.14a. The maximum and the minimum thresh-

olds have been fixed to 30 sec. and 10 sec. respectively, and the best control trajectories have been selected optimizing the cost function (6.9). Without loss of generality the initial configuration of the server is supposed to be the MAP version with parallelism degree equal to 5 workers. At the beginning of each control step, the control part predicts the trajectory assumed by the mean task size for the prediction horizon (through the Holt-Winters filter), and selects the optimal reconfiguration plan. Then only the first element of the trajectory is applied for the current step (operation-parallelism degree pair). We have executed this application for 40 control steps (each one of 300 seconds) and the exploited reconfigurations are depicted in Figure 6.17 for different prediction horizon lengths.

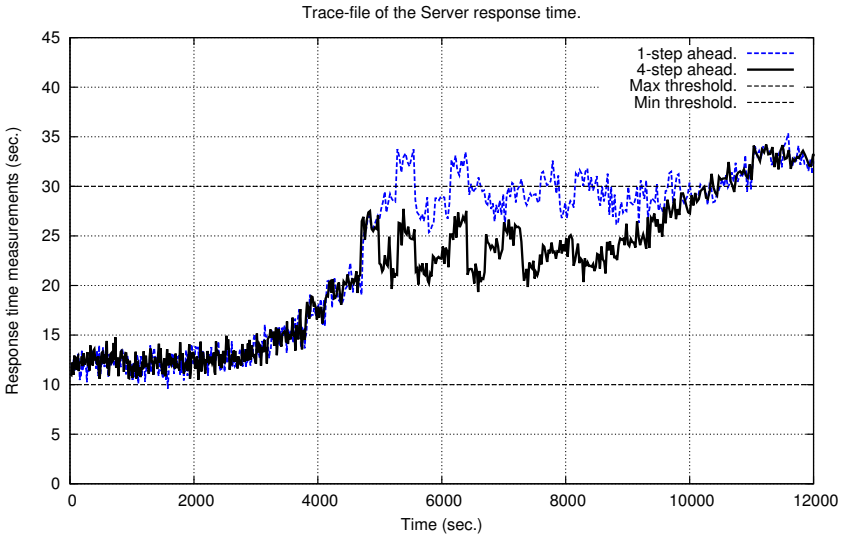


Figure 6.18: Response time for each request experienced by the Server during the entire execution.

Control steps in which the task-farm version has been used are highlighted in black in the figures, otherwise the data-parallel version has been adopted. In this example a sufficiently long prediction horizon gives a positive degree of foresight of the controller, which is able to determine the minimum set of reconfigurations that allow the mean response time to be within the provided thresholds for the whole prediction horizon. The effect on the execution is clear: by adopting sufficiently long prediction horizons, the control part takes a lower number of corrective actions and in advance to future violations of the QoS constraints. In fact by adopting a prediction horizon of 4 steps we have a ~ 33 % reduction of reconfigurations w.r.t the 1-step ahead case. Unfortunately the length of the prediction horizon can not be an unbounded parameter, because long prediction horizons may compromise the accuracy of disturbance input predictions and because the length of the horizon has a notable impact on the complexity of the optimization process (see Section 6.2.6 for further considerations about this).

Finally the performance behavior of the server is depicted in Figure 6.18. In this case we present the response time for each computed task by the server, which is interesting for highlighting the real violations of the threshold constraints. The dashed line represents

the achieved response time for each request with a 1-step prediction horizon. The black solid line represents the response time with the 4-step ahead strategy (results for 2 and 3-step ahead are not shown for brevity, but they are between the dashed and the solid lines depicted above). As we can see with a prediction horizon of 4 steps the threshold violations are significantly less than than the 1-step ahead case, except in the last part of the execution in which, due to a too large mean task size, no server configuration exists that satisfies the required QoS constraints.

6.2.6 Considerations about the computational complexity of the control strategy

Finally we conclude the description of this experiment by providing a discussion about the computational complexity of this strategy. As it has been described in Chapter 5, the predictive control strategy requires the exhaustive exploration of future QoS states, achieved considering all the different sequence of reconfigurations over the prediction horizon. At each control step the optimization process finds the best reconfiguration plan exploring an evolution tree whose size grows exponentially in the length of the prediction horizon.

In order to apply the predictive control in real-time, the optimization process must be efficiently completed at each control step. Especially when we have a high number of possible ParMod configurations (e.g. in this example 30 configurations), techniques able to reduce the research space are a central point for applying this approach in real-time. For this example we have adopted the Branch & Bound technique described in Section 5.3.3.2, that performs a significant reduction of the number of explored states. As said, this approach can be applied if the cost function is monotonically increasing with the prediction horizon. This condition is clearly satisfied by function (6.9), where the total cost is expressed as the sum of the non-negative partial costs for each step of the prediction horizon. In Table 6.4 two results are compared: the theoretical number of

Strategy	Theoretical n. of States	Average Explored States
1-step ahead	31	31
2-step ahead	931	189
3-step ahead	27931	824
4-step ahead	837931	3690

Table 6.4: Number of explored states adopting the Branch & Bound approach.

states that compose the evolution tree, and the average number of explored states with the Branch & Bound approach. As we can see the reduction in terms of explored nodes is of one or more orders of magnitude, and increases considerably with the length of the prediction horizon. Due to this fact, in this experiment the average control delay

for completing the optimization process at each control step is less than 3 seconds, thus negligible w.r.t the size of a control step.

6.3 Summary

In this chapter we have presented two examples of parallel applications in which the capability of dynamically reconfigure parallel modules is of extreme importance. For the first experiment we have dealt with a varying inter-arrival time of tasks due to dynamic network conditions. In this approach we have studied a reactive and a predictive control strategy for reducing the operating cost of the execution (in terms of utilized nodes and reconfiguration number) trying to maximize the number of completed tasks. In this experiment the adoption of a predictive approach makes it possible a further reduction of the operating cost w.r.t a purely reactive strategy, with also a higher number of completed tasks. In the second experiment we have exploited a client-server parallel computation modeled with the results described in Chapter 4. In this scenario we have shown how a sufficiently long prediction horizon can be of special interest for reducing the number of QoS violations and the number of necessary reconfigurations performed during the entire execution, improving in this way the stability degree of the control decisions.

Chapter 7

Models for Distributed Control of ParMod Computation Graphs

In this chapter we face with the crucial issue of controlling distributed graphs of adaptive parallel modules. In Chapter 5 we have presented a formalization of the concept of adaptive parallel module and we have introduced specific control-theoretic techniques for controlling its QoS. In Chapter 6 such techniques have been instantiated to real-world parallel applications, providing results about the viability of our approach. Nevertheless, so far we have considered relatively simplified examples in which the adaptation process involves only one ParMod. In other words we have studied examples in which a single adaptive parallel module has been controlled to meet user requirements and preferences such as performance and operational cost, maximum and minimum thresholds and number of reconfigurations.

In this chapter we extend the previous work providing theoretical foundations and methodological tools for controlling complex compositions of adaptive ParMods. In this case communication and cooperation among modules is exploited for two main reasons. Operating parts of different ParMods interact in order to exploit the distributed application logic. In fact a distributed computation is composed of separate phases, each one that may require an internal parallelization, that are executed on different classes of execution platforms. The control of the entire application often deals with different objectives: modules can represent users (e.g. clients) requiring different notions of QoS, or computing phases featuring distinct capabilities in terms of performance, memory usage and resource utilization cost. Secondly in many cases the QoS of a module is not independent with the behavior of the other modules. For these reasons in distributed scenarios multiple control parts communicate to find an agreement in their control actions. This chapter is aimed at describing an extension of the ParMod model in order to deal with the general issues of distributed control.

7.1 Centralized control of Parallel applications

The most straightforward solution for controlling a computation graph of parallel modules consists in exploiting a unique application controller, which is responsible for observing the entire application behavior and taking reconfiguration actions for each of its component.

This centralized control model is a reformulation of the ParMod model described in Chapter 5. In fact now the operating part corresponds to the entire computation graph that needs to be observed and controlled by a control part, as depicted in Figure 7.1. Following this approach we can identify for each module a proper set of corresponding

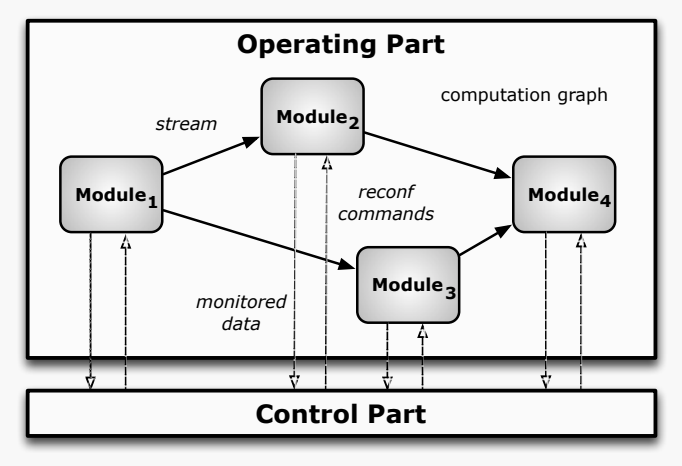


Figure 7.1: Centralized control of a distributed parallel application.

model variables:

- a continuous-valued vector $\mathbf{x}_i(k)$ that describes the current value assumed at the beginning of control step k by state variables belonging to the i-th ParMod;
- a continuous-valued vector $\mathbf{d}_i(k)$ that describes the values assumed by disturbance inputs influencing the QoS behavior of the i-th ParMod;
- a discrete-valued vector $\mathbf{u}_i(k)$ that identifies, for the k-th control step, the configuration (i.e. parallelism degree, operation and execution platform) of $ParMod_i$.

We can model the application behavior by defining a set of global state, disturbance and control input variables for the entire computation graph, through a proper composition of the single-module vectors:

$$\mathbf{x}_g(k) = \left[\mathbf{x}_1(k)^T, \mathbf{x}_2(k)^T, \ldots, \mathbf{x}_N(k)^T\right]^T$$
$$\mathbf{u}_g(k) = \left[\mathbf{u}_1(k)^T, \mathbf{u}_2(k)^T, \ldots, \mathbf{u}_N(k)^T\right]^T$$
$$\mathbf{d}_g(k) = \left[\mathbf{d}_1(k)^T, \mathbf{d}_2(k)^T, \ldots, \mathbf{d}_N(k)^T\right]^T$$

where the subscript "g" indicates the attribute "global", that refers to the entire application.

In order to apply formal control-theoretic methodologies as the model-based predictive control (shortly **MPC**) introduced in Section 5.3.3, we need a system model that can be used for predicting the future values of QoS variables in function of reconfiguration actions and proper predictions of disturbance inputs. The previous hybrid modeling of the ParMod operating part described in Section 5.2.1 can be extended to the entire application graph: in this case an application configuration consists of a specific choice of parallelism degree, operation and execution platform for each parallel module of the computation graph. Theoretically, this means that we need a *centralized system model* expressed as:

$$\mathbf{x}_g(k+1) = \Phi_g\Big(\mathbf{x}_g(k), \mathbf{d}_g(k), \mathbf{u}_g(k)\Big) \tag{7.1}$$

With this model we can apply the adaptation strategies already discussed in the previous chapters. For the MPC approach, a centralized control of the computation graph involves the following phases:

- the centralized control part periodically receives monitored data (state variables and measured disturbances) from each module of the operating part;
- the control part exploits statistical techniques for predicting the future trajectory assumed by disturbance inputs over a prediction horizon of h control steps. To be more precise the control part predicts the future trajectory of global disturbances, i.e. $\overline{D}_g(k) = \{\hat{\mathbf{d}}_g(k|k), \hat{\mathbf{d}}_g(k+1|k), \ldots, \hat{\mathbf{d}}_g(k+h-1|k)\}$;
- based on the centralized system model, the control part finds a sequence of reconfigurations $\overline{U}_g(k) = \{\mathbf{u}_g(k|k), \mathbf{u}_g(k+1|k), \ldots, \mathbf{u}_g(k+h-1|k)\}$ such that a *global objective function* J_g is optimized over the prediction horizon;
- of the optimal control trajectory only the first element $\mathbf{u}_g(k|k)$ is applied to the plant. In other words a proper set of reconfiguration commands are transmitted to each ParMod in order to specify the global configuration that will be executed for the entire duration of the k-th control step;
- the entire procedure will be repeated at the beginning of the next control step $k+1$, by exploiting the new updated measurements of the application behavior.

Over the years centralized control schemes have been intensively applied for the control of complex systems. The main advantage of this approach is the presence of a centralized model of the entire system, which is able to capture all the internal dynamics of the plant and represent them as accurately as possible. Typically model-based predictive controllers are implemented following the centralized scheme, in which the system is completely modeled and all the control inputs are calculated by solving a global optimization problem. For this reason in the literature this control structure is often referred as

an ideal solution, since in principle it can determine actions that give the optimal *control quality*.

Nevertheless centralized control is not likely to be a feasible method for large-scale systems as power, water distribution, traffic and manufacturing systems as well as complex distributed computing systems executed on heterogeneous and highly distributed computing platforms. In this case a purely centralized approach may be difficult to be applied due to robustness and reliability problems (a single controller is also a single point of failure). Another fundamental limitation of centralized approaches, especially critical for hybrid systems, is the required computational complexity. In fact we can observe that this approach explodes in complexity when the number of control input combinations is large enough to render the online optimization unfeasible in real-time contexts.

Solutions follow the general concept of decomposing a large-scale system into multiple interacting sub-systems controlled by a complex organization of local controllers. In the following section general decomposition techniques will be discussed, presenting their formalization, the feasibility of the resulting control schemes and the implications on the control quality compared to the centralized solution.

7.2 Control schemes for large-scale distributed systems

The main objective of a control scheme is to assure that, while the system is working, it is capable to reach its goals and assure that no requirement is violated. As we have seen in a control technique two important aspects can be pointed out: (i) the behavior of the system has to be modeled to predict how it will behave given a certain set of control decisions; (ii) the underlying control logic needs to be designed in order to take proper sets of corrective actions and apply them at each decision point. Therefore we can introduce the following concept:

Definition 7.2.1 (Control Problem)**.** A ***control problem*** is composed of a formal model of the system, that describes in a mathematical fashion its behavior in response to control inputs and disturbances, and an adaptation strategy i.e. a rule that specifies how control actions are decided at each decision point.

If we consider the MPC approach, a predictive control problem is completely described by a static or dynamical system model and an objective function used to find the sequence of actions such that the system goals are achieved optimally. Solving a control problem is done through the use of controllers that have access to the monitored data of the system and have the necessary computational resources for applying the adaptation strategy.

In this section we will describe how control problems can be solved by a single or, more interesting, by a collection of properly structured controllers.

7.2.1 Centralized control schemes

The first approach, already hinted in the MPC context in Section 7.1, is the centralized control scheme defined as follows:

Definition 7.2.2 (Centralized Control Scheme)**.** A centralized control scheme is defined as a tuple $\langle P, C, \tau \rangle$ composed of: (i) the observed system (plant) P; (ii) a unique controller C that observes the entire system execution and takes reconfiguration decisions; (iii) a control step length τ that indicates how often control actions are exploited by the system controller.

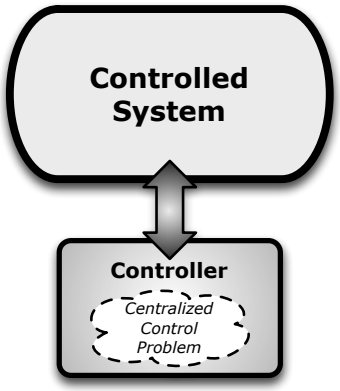


Figure 7.2: Centralized control scheme.

This control scheme (Figure 7.2) is defined as a *centralized approach*, since a unique control entity is responsible for observing the entire set of QoS variables and taking control inputs that influence the behavior of the entire system. In other words the controller solves a unique centralized control problem. This means that the controller has a complete vision of the system behavior (i.e. it receives monitored data from all the system), it owns a complete plant model in order to predict the system response to disturbances and control inputs, and it applies a centralized adaptation strategy.

The centralized approach is characterized by specific advantages but also disadvantages that affect the feasibility and the effectiveness of this solution. As stated in [124], the main advantage is due to the centralized and complete vision that the controller has of the system, that in principle leads to theoretically optimal reconfiguration strategies. As we have hinted in Section 7.1, in the literature the centralized approach is often considered the best solution from the control quality viewpoint. Nevertheless centralized schemes have crucial limitations that may prevent their application:

- some control problems have a distributed nature (e.g. they are composed of multiple sub-parts interacting with each other). In this case a centralized control scheme may suffer from robustness, reliability and scalability issues. As an example in case

of a system extension (e.g. in terms of additional features or by adding new components or entities) it may be hard to update the centralized controller adequately, since a unique system model, though useful for reaching the best control actions, can be difficult to be extended since all the system dynamics need to be clearly modeled. Moreover, when the number of model variables is extremely high, it may be impossible to extend the model fitting all the interesting dynamics in a sufficiently accurate way;

- in practical scenarios, due to the potentially large size of data transmitted by the system to the controller, an efficient and reliable interconnection network is an essential requirement for centralized control schemes. Moreover the control delay induced by the network behavior has to be considered in the system model (increasing its complexity even more);

- depending on the adaptation strategy adopted by the controller, a centralized solution can be hampered by computational complexity problems, that may limit the feasibility of this control approach. As described in Section 7.1, model-based predictive controllers are notable examples in which a centralized solution for large systems is often impracticable, especially for systems in which reconfiguration actions correspond to a precise choice among a discrete set of control inputs.

For these reasons the centralized approach often scales poorly with the size of the problem and it is not usually a viable solution in practise. We remark that this is especially true for ***complex control problems*** in which the system model is is highly complicated, representing several dynamics involving a large number of observed, disturbance and control variables. Therefore, to efficiently deal with such complex control problems, more scalable solutions than centralized control schemes have been proposed and applied in the literature. As stated in [125], two different research lines are:

- the formulation of suboptimal centralized control schemes, in which the complexity of the control problem is properly reduced in order to render the problem tractable for a centralized control approach. As an example suboptimal techniques can consist in reducing the number of control variables that can be modified by the controller at each sampling interval;

- other approaches consist in modeling techniques in which the entire control problem is decomposed into multiple sub-problems, each one solved by a corresponding local controller.

The second solution is a more general approach that leads to a well-identified set of control schemes of large applicability, that can be easily adapted to different control objectives and scenarios.

7.2.2 Control schemes based on a Single-Layer organization

Alternative approaches w.r.t centralized schemes are based on a more complex structure which is no longer composed of a unique entity in charge of observing and controlling the entire system, but the original control problem is decomposed into multiple smaller sub-problems than the original one, each one involving a fewer number of variables and constraints. This solution is based on a *divide-and-conquer* approach in which:

- the original control problem is *decomposed* into a set of well-identified and structured control sub-problems, each one with a sub-model that represents the dynamics involving a limited amount of observed, disturbance and control variables;
- each sub-problem is *solved* by a dedicated control entity that selects the values for a limited set of system control inputs;
- finally controllers may *interact* to find a sort of agreement on their control decisions, e.g. accounting for the effects on their part of the system originated by the control actions taken by other controllers.

Such decomposition can be exploited following two alternative directions:

- in some cases an initial centralized control problem is fully identified and, in particular, it is well-defined the original system model that describes the entire controlled plant. In this case, starting from this initial knowledge, a set of sub-problems and their corresponding sub-models can be identified. In this case we speak about a ***top-down*** decomposition approach;
- in other situations the centralized system model is not known a-priori, but it is only considered implicitly. In this case we try to decompose a system without considering any original centralized version of the control problem. In this case we speak about a ***bottom-up*** decomposition approach.

A complex control problem approached in this way is also known as a ***compound control problem***, since it is characterized into multiple and distinct sub-problems. The process of transforming a complex control problem into a corresponding compound control problem can be exploited following different principles. A feasible decomposition can be applied based on the identification of *distinguishable physical components*. In this case each component can be clearly identified and controlled by a local controller exploiting a model and a proper adaptation strategy for this part of the system. Another decomposition principle is based on *distinguishable control goals*, in which the aim of the entire system control can be represented as a multi-objective control problem in which multiple performance indices are optimized. In this case each local controller can be designed to optimize a specific objective of the system, requiring proper controller interactions if two objectives are coupled and inter-dependent with each other.

A decomposition approach is aimed at overcoming the limitations of a purely centralized control. In fact it improves the robustness and reliability of the control structure and

it reduces the communication delay, since each local controller is responsible and receives data only from a specific sub-part of the system. However, the solution of a compound problem may have a lower quality than the control solution achieved by a centralized controller that with a complete vision of the system. Therefore, a crucial issue concerns how the solutions of smaller local problems can be integrated in order to find an effective solution of the starting problem.

For these reasons different sub-problems are usually not independent, but that can interact with each other because their corresponding sub-systems are strongly coupled. This means that, even if each controller finds the solution of each sub-problem separately, the control actions selected by one of the local controllers influence the behavior (e.g. the control quality) of other sub-systems and, as a consequence, the decisions taken by the other controllers. This class of relationships (also called *indirect dependencies* in [126]), is of great importance for our modeling purpose and can be defined as follows:

Definition 7.2.3 (Coupling Relationships)**.** A ***coupling relationship*** between two sub-problems of the same original control problem is a specific law that describes how the control actions applied for the first sub-problem influences the control actions that will be selected for solving the second sub-problem, and vice-versa.

When deciding how to solve a compound control problem, several strategies can be adopted for organizing an effective and efficient control structure. Five different aspects drive this design:

- how subdividing the original complex control problem into multiple sub-problems and how they can be mapped onto a corresponding set of local local controllers. For doing this several trade-offs can be made, for instance between the number of sub-problems and their complexity. In general the larger the number of sub-problems is, the simpler the single sub-problem will be. However, in the case of a high number of sub-problems, also the complexity of existing coupling relationships will increase;

- how the entire controlled plant exchanges information with the control structure and its local controllers: monitored data can be transmitted to a set of interesting controllers simultaneously (i.e. multicast of data) or through single point-to-point interconnections;

- how coupling relationships between sub-problems can be implemented in a real scheme: e.g. what class of interconnections exist between controllers;

- what is the ***interaction protocol*** that the controllers follow for exchanging data and deciding the control actions on their corresponding sub-systems;

- what is the possible degree of authority that a controller has over the others.

In order to answer to these points we will present a first class of control structures based on a ***single-layer decomposition*** of the original complex control problem. These

schemes consist in a partitioning of the entire observed plant P into a set of distinct sub-systems $P_1, P_2, \ldots, P_N$. A typical property of single-layer organizations consists in having multiple controllers with the same level of authority, i.e. there is no hierarchy among local controllers, but each one of them is responsible for observing and controlling a well-identified sub-system. Therefore we can identify a collection of local controllers $C_1, C_2, \ldots, C_N$ each one autonomously responsible for observing and taking decisions involving a specific sub-system. In this control organization each controller C_i is able to:

1. acquire monitoring information exclusively from its corresponding part P_i of the system;

2. understand and predict the behavior of its controlled sub-system P_i through a local model Φ_i;

3. in order to decide control input values, the local controller C_i can be interconnected with other local controllers and exchange control information following a specific interaction protocol;

4. based on the results of the adaptation strategy of the controller, control input modifications are only transmitted to the corresponding observed part of the system.

As we will see later in this section the third point is especially critical, and significant classifications among single-layer control structures will be characterized by the existence of information exchanges among different controllers.

Single-layer control schemes can be represented through two different *interaction structures*, that describe the existing interactions between sub-systems and between controllers. The interaction structure among sub-systems I_p can abstractly be represented by a matrix $\{0, 1\}^{N \times N}$ defined as follows:

$$I_p[i, j] = \begin{cases} 1 \text{ if sub-system } P_i \text{ directly interacts with } P_j \\ 0 \text{ otherwise} \end{cases}$$

In a similar way it is possible to describe an interaction structure I_c that represents the existing interactions among local controllers, that is a matrix $\{0, 1\}^{N \times N}$ that indicates if two controllers are directly interconnected for exchanging control information. We can introduce the following definition:

Definition 7.2.4 (Single-Layer Control Scheme)**.** A single-layer control scheme can be defined by a tuple $\langle \{P_i\}, \{C_i\}, I_p, I_c, \tau \rangle$ where the first element is the set of sub-systems in which the original plant has been decomposed, the second element is a set of controllers exactly of the same number of the identified sub-systems (one controller per sub-system). Then the single-layer control structure is characterized by two interaction structures that describe the interactions between sub-systems and between controllers, and finally a unique concept of control step τ, common to all the controllers.

A first classification of single-layer control schemes is based on the existence of interactions between local controllers. In many real-world cases the original plant is decomposed into sub-systems controlled by a set of controllers that are designed to operate in a completely independent fashion. In this case we speak about ***decentralized single-layer control schemes*** (see Figure 7.3), in which no controller interaction exists (i.e. the structure I_c has all zero elements). We remark that in this control schemes the sub-systems can be coupled with each other, but these coupling relationships are completely neglected and controllers operate independently.

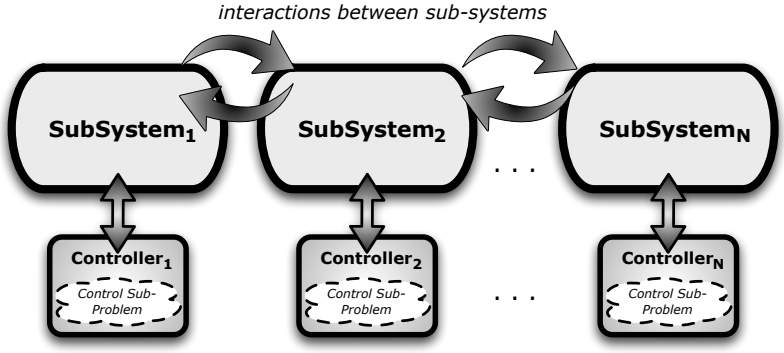


Figure 7.3: Decentralized single-layer control scheme: no existence of controller interactions.

Generally the main advantage of a decentralized scheme is its high scalability with the number of sub-systems. Each controller evaluates its control strategy without any interaction and synchronization with other controllers, thus operating in parallel minimizing the overhead for calculating the control inputs for each sub-system. However these advantages are at the price of a potentially limited quality of the control decisions, which can be far from the ideal one of a centralized approach. In fact a decomposition of a complex system hardly produces completely independent sub-systems, but in many cases significant dependencies and relationships among sub-systems exist. As said before, in a decentralized control scheme all these dependencies are completely neglected, and the control actions of a sub-system are selected in a completely independent fashion w.r.t the actions taken by the other controllers. As a consequence a decentralized approach is a viable solution only for loosely-coupled sub-systems, in which the dependencies among sub-systems can be completely neglected without incurring in a significant loss of the control quality.

When the decomposition identifies strongly-coupled sub-systems, a completely decentralized approach may give inaccurate results and may not be a satisfactory solution. In this situation, in order to achieve a better control quality, controllers of single-layer organizations usually exchange control information at each control step so that they have some knowledge of the behavior of the others. In this case we speak about ***distributed single-layer control schemes*** (Figure 7.4), in which the interaction structure I_c has non-zero elements. In a distributed scheme each controller, in order to correctly establish

the control inputs that will be supplied to its sub-system, takes into account not only disturbance predictions and local measurements, but also control information from other controllers.

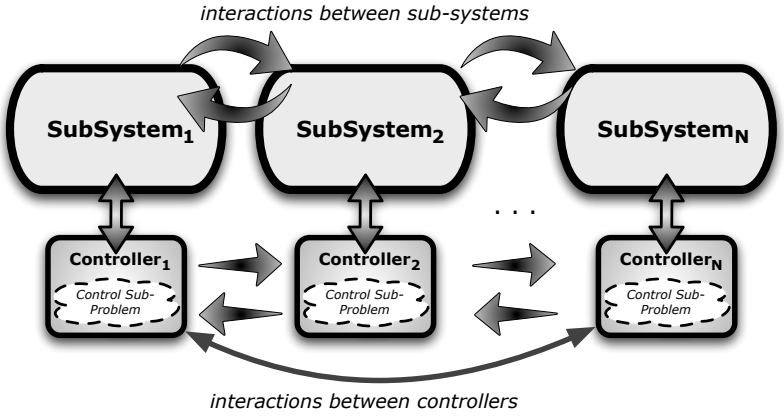


Figure 7.4: Distributed single-layer control scheme: existence of controller interactions.

As reported in [124] distributed control schemes can be classified into several categories based on specific properties of the controller interconnections. A first distinction depends on the interaction pattern. We have:

- ***fully-interconnected schemes*** in which each controller communicates with all the other local controllers at each control step;
- ***partially-interconnected schemes*** in which each controller is directly interconnected with a limited set of neighbor controllers (i.e. those of which the interactions between corresponding sub-systems can not be neglected).

Partially-interconnected schemes are convenient when a small number of coupling relationships between control sub-problems is sufficient to correctly model with good accuracy the dominant interactions between sub-systems. Another important classification is related to how many times control information is exchanged among controllers at each control step before correctly establish the set of control inputs. In this case we have two main possibilities:

- ***non-iterative interaction protocols*** (Figure 7.5) in which controllers exchange control information only one time within the control step. This means that, at the beginning of each control step, controllers exchange control data and then based on local disturbance predictions and actual monitoring data from their sub-systems, they directly establish their control inputs for the current control step;
- ***iterative interaction protocols*** (Figure 7.6) in which the information exchange occurs many times within each control step, until a convergence condition is satisfied or for a fixed number of iterations.

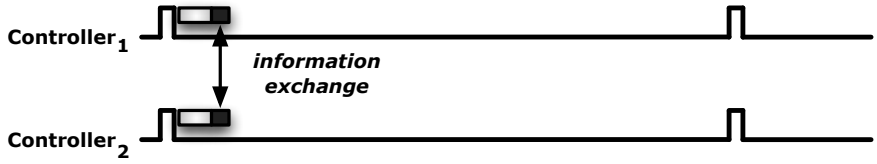


Figure 7.5: Non-iterative interaction protocol between local controllers.

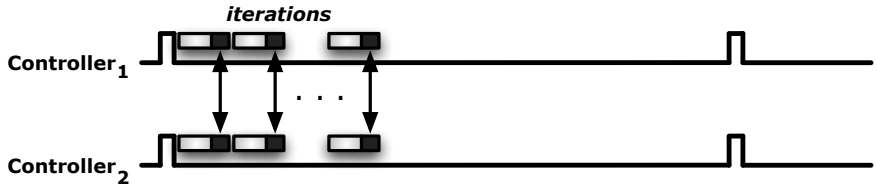


Figure 7.6: Iterative interaction protocol between local controllers.

Typically non-iterative protocols require a lower number of communications between controllers and thus the time for completing the interaction protocol and deciding control inputs is in general smaller than with iterative schemes. We can also note that the time for completing the interaction protocol corresponds to the parameter $T_{Control}$ of the interaction model between operating and control part introduced in Section 5.1.2.

Finally a last classification among distributed control schemes is based on the behavior assumed by each controller w.r.t the others:

- ***self-interest interaction***: each controller establishes the control inputs that will be supplied to its sub-system trying to optimize a local performance index, without taking into account the effects of its actions on the behavior of the other controllers. It is worth noting to not confuse this class of distributed schemes with purely decentralized approaches. As said, in the latter case local controllers are completely independent and they do not communicate for exchanging control information. On the other hand in a self-interest interaction protocol local controllers communicate with each other, i.e. control actions depend on the received control information but each controller always applies the best strategy for itself;

- ***cooperative interaction***: each controller selects the current control inputs for its sub-system trying to optimize a global performance index of the entire plant (i.e. for all of its sub-systems). This kind of interaction among controllers is aimed at reaching a control quality as near as possible to the ideal quality achievable by a centralized scheme.

In summary so far we have introduced different classes of control schemes. The centralized approach is characterized by the possibility to achieve the best control quality, since a unique controller has a complete view of the system dynamics, but at a price of a limited applicability to real cases due to robustness, scalability and computational

tractability issues. On the other hand decentralized schemes are an extremely scalable solution featuring smaller control overheads. Nevertheless, due to the absence of controller interactions, decentralized schemes are often an effective solution only for loosely-coupled sub-systems, in which their interactions can be completely neglected. When this is not viable, decentralized schemes produce a control quality far from the corresponding centralized approach. In order to overcome such limitations and leverage the strengths of these solutions, distributed control schemes can be used. The presence of controller interactions leads to a higher control overhead and a more limited scalability compared to decentralized schemes, but these communications are aimed to move closer to the ideal quality of a centralized approach (and in some cases to reach it).

7.2.3 Hierarchical control schemes

A different approach to system decomposition consists in structuring the distributed control scheme as a set of local controllers organized in a hierarchical manner. So far we have described single-layer control schemes in which all the controllers are at the same authority level: i.e. each one of them is responsible for a specifically identified sub-part of the system. In this section we will briefly introduce a general ***multi-layer organization*** [127] of local controllers based on different authority degrees.

The authority level among local controllers can be represented as a tree structure, in which the leaves represent controllers of the lowest level, that are directly responsible for observing a sub-part of the system and for transmitting control inputs to it. Higher level controllers are not directly responsible for controlling a specific sub-system, but they coordinate the behavior of local controllers of their sub-tree. In Figure 7.7 is depicted an example of hierarchical control of a system composed of four different parts controlled by a hierarchy of seven controllers. In this figure the tree representing the controller hierarchy is a binary tree, but other structures could be possible (e.g. with a different arity and a different number of controllers). A definition of multi-layer control scheme can be provided as follows:

Definition 7.2.5 (Multi-Layer Control Scheme)**.** A multi-layer control scheme can be defined as a tuple $\langle \{P_i\}, T, \{\tau_j\} \rangle$ where the first element is the set of N sub-systems in which the original plant has been decomposed, the second element is a tree that has exactly the same number of leaves as the number of sub-systems. The third element is a set of control step lengths one for each level of the tree.

As we have seen internal controllers are not directly connected with any sub-system, but they are interconnected with a set of controllers of the lower level of the hierarchy. For instance in Figure 7.7 we can observe that controller 5 directly interacts with controllers 1 and 2 which are connected to the first and the second sub-part of the system. Controllers that represent leaves are connected to a sub-system through a classic closed-loop interaction, but information exchanges also exist between controllers at different levels of the hierarchy. At this regard we can identify:

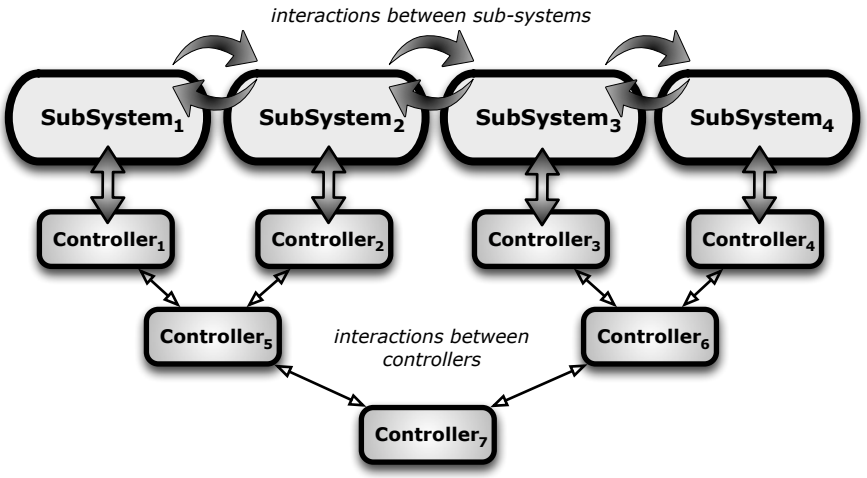


Figure 7.7: Multi-layer control scheme.

- *top-down information exchange* between different layers: as an example higher-layer controllers can determine the objectives of the control actions exploited by lower-layer controllers. For instance controller 5 can establish the control objectives of controllers 1 and 2;

- *bottom-up information exchange* between different layers: e.g. control objectives decided by higher-layer controllers need to be feasible for lower-layer controllers. For instance this means that lower-layer controllers have to inform their supervisors of the actual quality of their control actions.

The hierarchical organization that we have described is the most classic one, in which controller interactions can be represented by regular tree structures with different arities. In general, especially for large plants exploiting complex interactions, several degrees of freedom can be considered, for instance the case in which controllers of the same layer can also be interconnected with each other. In this case instead of a classic tree structure, controller hierarchy can be represented as graphs of any form.

Multi-layer control schemes are often a flexible approach for controlling real-world systems, and thus they are widely studied in many contexts. Their main advantages can be summarized in two distinct points:

- hierarchical control structures are often applied when a complex system is characterized by dynamics with different speeds. In this case it can be effective to control the faster dynamics through lower-layer controllers with shorter control steps, and regulate slower dynamics by higher-layer controllers with longer sampling intervals. In this case multiple-layer approaches are more flexible w.r.t single-layer schemes in which the entire set of controllers is characterized by a unique notion of control step;

- a hierarchical structuring of controllers can be useful in order to apply a heterogeneous set of control strategies. As an example it is possible to exploit a reactive approach for the leaves directly connected with the sub-systems, and a predictive control technique for controlling long-term system dynamics through proper high-level controllers.

Nevertheless, as reported in [127], an important issue of multi-layer control schemes is how to provide accurate system models for higher-layer controllers. In this case a high-level system model needs to represent the behavior of several sub-systems and also of their local controllers. In many cases this makes the formulation of control strategies more complex to be defined.

7.2.4 Summary of different control structures

In this section we have described several control schemes that can be applied in many real-world situations, especially for controlling large-scale distributed systems. The classifications that we have provided are summarized in Figure 7.8. As said before, centralized control schemes are rarely a feasible solution, both for complexity and robustness reasons. To this end a set of local controllers can be developed for controlling large-scale systems starting from a decomposition of the original control problem. The main classification within single-layer approaches depends on the existence of interactions among controllers (decentralized vs. distributed schemes). The presence of such interactions is

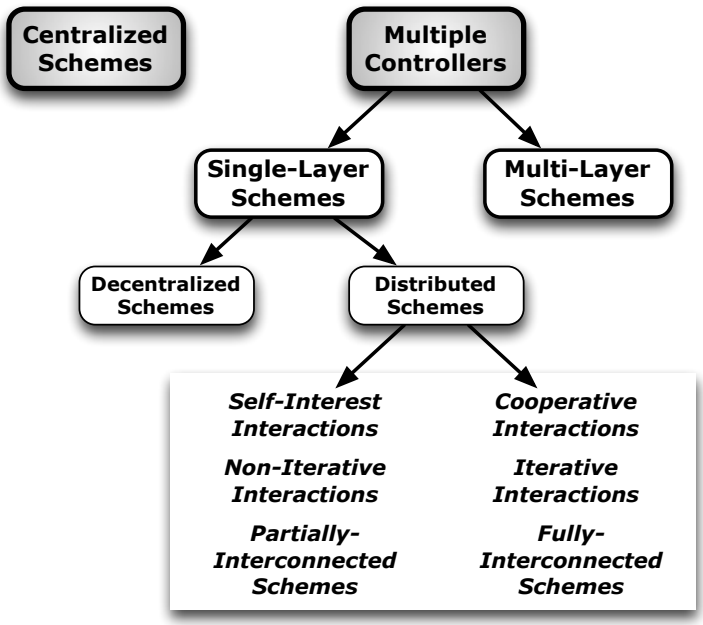


Figure 7.8: Taxonomy of control schemes.

aimed at achieving a final control quality as near as possible to the ideal quality of a fully centralized approach, especially when strongly coupled sub-systems have been identified. Multi-layer schemes are a different modeling structure in which controllers are organized in a hierarchical manner. It is worth noting that there not exists a unique best solution for every case, but the most convenient structure depends on the particular behavior of the controlled system and how its sub-parts are organized and interconnected. Based on the control schemes described before, in the following part of this chapter we will describe a modeling strategy for controlling distributed parallel applications composed of multiple adaptive parallel modules.

7.3 A distributed control model for Parallel Applications

In this section we will propose a solution to the problems introduced in Section 7.1 for controlling distributed parallel computations featuring a set of modules interconnected in graph structures. In particular we will present an extension of the ParMod model introduced in Chapter 5 and we will discuss how control parts of different modules can exchange information in order to exploit a distributed adaptation strategy.

As we have seen a distributed parallel application can be represented as a graph of multiple modules or components, that cooperate to achieve a global goal in a distributed manner. Each module is an independent unit that executes a possibly parallel computation (exploiting well-known structured parallelism schemes). The entire application is usually executed respecting several QoS requirements, like performance constraints, that need to be maintained by reconfiguration activities.

In this context a centralized control logic can be decomposed considering each parallel module as a sub-system. This means that each ParMod will be equipped with sufficient resources and computational capabilities to exploit both its functional logic (a structured parallel computation) and its control logic that selects local functional or non-functional reconfigurations when it is required. This approach requires that global QoS objectives of the entire application are translated in terms of local objectives of each ParMod:

- some QoS objectives can be easily decomposed into a set of local objectives. As an example if we want to minimize the total memory usage of a distributed application, a possible strategy consists in minimizing the memory utilization of each module through a proper selection of the more memory-saving configuration;

- for other QoS measurements the decomposition can be harder. For instance consider the case in which we have to optimize the efficiency of each module of the graph, expressed by the ratio between the ideal performance of the current configuration respect to the effective performance achieved by the module. In this case the local control logic of the module needs to be able to predict how reconfigurations of other ParMods influence its effective performance behavior. This means that some QoS objectives impose strong coupling relationships among distinct control sub-problems.

The approach that we are proposing in this section consists in a *single-layer control scheme* that can be described by the following points:

- a distributed parallel application is represented as a directed graph G of ParMods $P_1, P_2, \ldots, P_N$;
- each ParMod P_i is composed of an operating part PO_i and a control part PC_i;
- the entire application is decomposed in terms of a set of sub-systems corresponding to the operating parts $PO_1, PO_2, \ldots, PO_N$ (parallel or sequential sub-computations) interconnected through streams of data implementing the distributed functional logic. The interaction structure I_p describing sub-systems interactions is a $N \times N$ matrix assuming value 1 at position (i, j) iff PO_i is directly interconnected with PO_j through a data stream, zero otherwise;
- each control part PC_i is interconnected with its corresponding operating part PO_i following the classic closed-loop interaction scheme described in Chapter 5;
- every control parts exploit the same control step length τ, which is the sampling interval common to all the application components;
- for each ParMod the behavior of local QoS parameters is described by a *local model* involving local state variables $\mathbf{x}_i(k)$, local disturbances $\mathbf{d}_i(k)$ and local control inputs $\mathbf{u}_i(k)$ that specify the alternative configurations of the module.

QoS predictions can be exploited through a local model of the operating part behavior: i.e. $\mathbf{x}_i(k+1) = \Phi_i\big(\mathbf{x}_i(k), \mathbf{d}_i(k), \mathbf{u}_i(k)\big)$. It is worth noting that this local model, compared to a global model of the entire application as discussed in Section 7.1, is much more amenable to be managed, since the number of involved variables is significantly smaller.

This modeling is sufficient to define a decentralized single-layer control scheme for distributed parallel applications, in which the entire control problem is decomposed into multiple parallel modules each one equipped with a proper adaptation strategy. In this case each control part operates in a completely independent fashion w.r.t the other control parts. As said a decentralized approach is not sufficiently accurate to achieve effective control decisions in the case of strongly coupled sub-systems. As we have seen in Chapter 4, this is the case of the performance modeling of structured parallel computations and their compositions in graph structures. Therefore, in order to deal with these coupling relationships, we extend the ParMod model taking into account possible information exchanges among control parts.

In our model, coupling relationships are implemented by means of proper ***interconnecting variables*** among control parts (depicted in Figure 7.9). Besides classic state, disturbance and control variables, the operating part model considers further sets of variables:

- ***input interconnecting variables*** model the interaction between control sub- problems of other ParMods and the local problem of $ParMod_i$. Their values are indicated

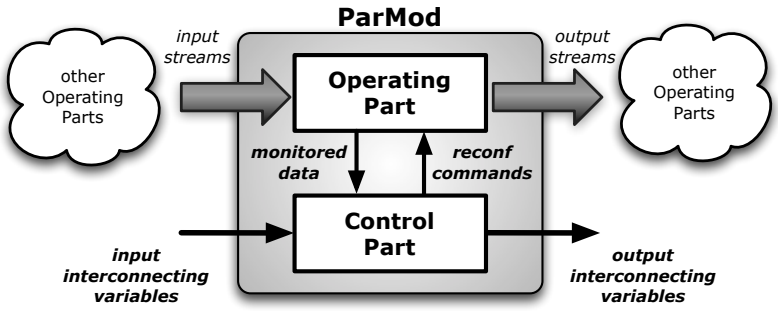


Figure 7.9: Controllers of different ParMods exchange interconnecting variables.

by a continuous-valued vector $\mathbf{v}_{in-i}(k)$ that represents the values assumed by input interconnecting variables throughout the k-th control step of the execution;

- ***output interconnecting variables*** represent information of the local problem of $ParMod_i$ that are transmitted to a set of other control parts of the application graph. Their values for the k-th control step are denoted by a continuous-valued vector $\mathbf{v}_{out-i}(k)$.

The model of $ParMod_i$ is extended considering the presence of such interconnecting variables. Input interconnecting variables directly influence the state variables evolution: i.e. the future values of state variables depend on disturbance variables, control inputs and also on input variables received by other control parts of the distributed control structure:

$$\mathbf{x}_i(k+1) = \Phi_i\big(\mathbf{x}_i(k), \mathbf{d}_i(k), \mathbf{u}_i(k), \mathbf{v}_{in-i}(k)\big) \tag{7.2}$$

A different aspect concerns how output interconnecting variables are determined. In this case a flexible approach consists in considering output interconnecting variables as possible values generated in function of the actual state variables, disturbances and control inputs:

$$\mathbf{v}_{out-i}(k) = Z_i\big(\mathbf{x}_i(k), \mathbf{d}_i(k), \mathbf{u}_i(k)\big) \tag{7.3}$$

in which Z_i is the *output generation function*. As special cases some output interconnecting variables can exactly correspond to the values currently assumed by control inputs, disturbances or state variables, i.e. for some variables the generation function can be the identity.

Each controller is responsible for solving its sub-problem at each control step. Interconnecting variables are in charge of modeling coupling relationships between two sub-problems. We denote with $\mathcal{V}_{in-j,i}$ the set of input interconnecting variables that $ParMod_j$ receives from $ParMod_i$ and with $\mathcal{V}_{out-i,j}$ the set of output interconnecting variables that $ParMod_i$ transmits to $ParMod_j$. Since the values of interconnecting variables transmitted by $ParMod_i$ to $ParMod_j$ are equal to the variables received by $ParMod_j$ from $ParMod_i$,

the can introduce the following *interconnecting constraints*:

$$\begin{cases} \mathscr{V}_{in-j,i} = \mathscr{V}_{out-i,j} \\ \mathscr{V}_{in-i,j} = \mathscr{V}_{out-j,i} \end{cases} \quad \forall i,j = 1,2,\ldots,N \tag{7.4}$$

Based on these definitions we can formally indicate when two ParMod control sub-problems are directly coupled with each other (see Figure 7.10):

Definition 7.3.1 (Directly coupled control sub-problems)**.** The control sub-problem of $ParMod_i$ is directly coupled with the sub-problem of $ParMod_j$ iff $\mathscr{V}_{in-i,j} \neq \emptyset$ and $\mathscr{V}_{in-j,i} \neq \emptyset$. In other words if PC_i receives some interconnecting variables from PC_j and vice-versa.

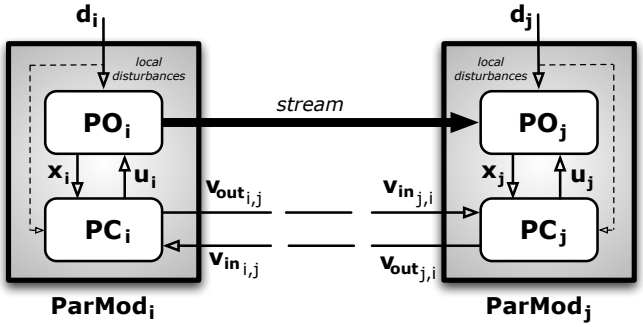


Figure 7.10: Directly coupled ParMod control sub-problems.

The set of input (output) interconnecting variables of $ParMod_i$ can be obtained taking the union of all the input (output) interconnecting variable sets received (transmitted) from (to) the other modules of the application graph:

$$\begin{cases} \mathscr{V}_{in-i} = \bigcup_{j=1}^{N} \mathscr{V}_{in-i,j} \\ \mathscr{V}_{out-i} = \bigcup_{j=1}^{N} \mathscr{V}_{out-i,j} \end{cases} \tag{7.5}$$

7.3.1 Distributed MPC Strategy for Parallel applications

In this section we will present a distributed formulation of the model-based predictive control strategy. Our goal is to describe how the extended ParMod model with interconnecting variables can be used for implementing a predictive control strategy of parallel applications composed of a proper interconnection of adaptive ParMods.

For a generic $ParMod_i$ the distributed MPC strategy can be expressed by the following sequence of actions that will be exploited at the beginning of each control step k:

- the control part of $ParMod_i$ acquires monitoring information from its operating part. As usual sensed data concern: (i) the actual values of state variables of the operating part model; (ii) the values assumed by disturbance inputs during the previous control step;

- based on the information acquired at the previous point, PC_i executes a proper statistical prediction of the values assumed by disturbance inputs over a limited prediction horizon of h control steps (i.e. the trajectory $\overline{D}_i(k)$). As said, we assume that the length of the horizon is the same for all the control parts involved in the distributed adaptation strategy;

- PC_i exchanges trajectories of interconnecting variables with other control parts to which it is directly interconnected, e.g. the input interconnecting variables trajectory received by PC_i is given by $\overline{V}_{in-i}(k) = \{\mathbf{v}_{in-i}(k|k), \mathbf{v}_{in-i}(k+1|k), \ldots, \mathbf{v}_{in-i}(k+h-1|k)\}$. Such information exchange can be exploited once per control step, or for a fixed number of iterations, or until a convergence condition is satisfied, depending on the interaction protocol (this aspect will be considered later in this chapter);

- on the basis of the currently received values, the control part of the i-th ParMod solves a *local optimization problem* composed of a local objective function:

$$
\begin{aligned}
&\underset{\overline{U}_i(k)}{\operatorname{argmin}}\ J_i\big(\overline{X}_i(k), \overline{U}_i(k), \overline{V}_{in-i}(k)\big)\\
&\text{subject to:}\\
&\mathbf{u}_i(j|k) \in \mathbf{U}_i \ \ j = k, k+1, \ldots, k+h-1\\
&\hat{\mathbf{x}}_i(j+1|k) = \Phi_i\Big(\hat{\mathbf{x}}_i(j|k), \hat{\mathbf{d}}_i(j|k), \mathbf{u}_i(j|k)), \mathbf{v}_{in-i}(j|k))\Big) \ \ j = k, k+1, \ldots, k+h-1\\
&\hat{\mathbf{d}}_i(j|k) = \overline{D}_i(k)[j] \ \ j = k, k+1, \ldots, k+h-1\\
&\hat{\mathbf{x}}_i(k|k) = \mathbf{x}_i(k)\\
&\mathbf{v}_{out-i}(j|k) = Z_i\Big(\hat{\mathbf{x}}_i(j|k), \hat{\mathbf{d}}_i(j|k), \mathbf{u}_i(j|k)\Big) \ j = k, k+1, \ldots, k+h-1
\end{aligned}
\tag{7.6}
$$

  The optimization problem requires to find the optimal trajectory of reconfigurations for $ParMod_i$ such that a local cost function J_i is minimized. It is worth noting that, respect to a centralized MPC formulation (see Section 7.1), each control part solves a local optimization problem featuring a specific local objective function J_i and a local model of the QoS dynamics Φ_i. So doing we expect that each sub-problem is simpler to be solved than the centralized one, since it involves a smaller number of state, control and disturbance variables;

- of the optimal trajectory $\overline{U}_i(k)$ found at the previous point, PC_i applies only the first control input of the sequence to establish the ParMod configuration that will be applied throughout the current control step. The rest of the sequence is discarded and the distributed MPC procedure will restart at the beginning of the next control step $k+1$.

In the approach described before, each controller selects its current control action based on the resolution of a local optimization problem. Local problems are coupled through input/output interconnecting variables, that are properly exchanged between different control parts at each control step. At this point two aspects need to be studied. The first one concerns the properties retained by the final result of the distributed control scheme. At each sampling interval the control parts of the application graph exchange (also several times) trajectories of interconnecting variables until a final agreement is achieved: i.e. what is this final set of reconfiguration trajectories and what are their properties?

The second aspect concerns how interconnecting variables are exchanged between controllers. Alternative techniques differ in the way in which control parts are interconnected, how many times interconnecting variables are exchanged and how long a protocol is executed before converging to a final agreement. For this reason interaction protocols are usually evaluated by comparing the required number of exchanged messages and the number of iterations before reaching a convergence in the control part decisions. The following two sections address these two points describing a theoretical framework for studying the concept of agreement between control part decisions.

7.3.2 Control Part interactions modeled by a Game-Theoretic approach

In this section we will introduce the basic concepts of a mathematical framework for studying the problem of coordination between controllers of a distributed parallel application. The distributed MPC strategy proposed in the previous sections can be presented as a *game* between a set of distributed agents.

Definition 7.3.2 (Game-based view of the distributed control of parallel applications)**.** At each control step the problem of selecting a set of reconfiguration trajectories for each ParMod can be represented as a *finite strategic game* defined by the tuple $\Gamma = \langle C, \{\Omega_i\}, \{J_i\}\rangle$:

- the set of players corresponds to the control parts of the application graph, i.e. $C = \{PC_1, PC_2, \ldots, PC_N\}$;
- each player PC_i is able to play a finite set of possible *strategies* that, in our case, correspond to the set of possible trajectories of reconfigurations that a control part can select. Each parallel module has a finite set of possible configurations, thus the number of reconfiguration sequences $\overline{U}_i$ over a limited horizon constitutes a finite number of possible alternatives;
- the set of control trajectories chosen by each player is called *strategy profile* and it is denoted by $S = \{\overline{U}_1, \overline{U}_2, \ldots, \overline{U}_N\}$;
- for each control part PC_i the quality of the control action is measured through a local cost function. Since control sub-problems are usually coupled in our problems, the

optimal strategy of a player depends on the interconnecting variables transmitted by other players of the game. Thus each J_i is expressed in terms of the future state (QoS) variables trajectory $\overline{X}_i$, the trajectory of reconfigurations (strategy) $\overline{U}_i$ and the trajectory of input interconnecting variables $\overline{V}_{in-i}$. Moreover we assume that each controller is a *rational agent*, i.e. it always operates in order to optimize its objective function J_i.

The distributed MPC approach requires that control parts will reach a convergence in their decisions at the end of the interconnection protocol. In this vision the problem of reaching this convergence can be mapped onto a theoretic formulation of a finite strategic game. Therefore the following definition is of great importance:

Definition 7.3.3 (Best response to the trajectory of received interconnecting variables)**.** Let us consider a generic control part PC_i of the graph. Given an input trajectory of interconnecting variables $\overline{V}_{in-i}$ from the other directly coupled control parts, we say that the strategy U_i^* is a ***best response*** to the received interconnecting variables iff the following condition is satisfied:

$$J_i\big(\overline{X}_i, \overline{U}_i^*, \overline{V}_{in-i}\big) \leq J_i\big(\overline{X}_i, \overline{U}_i, \overline{V}_{in-i}\big) \ \forall \text{ admissible } \overline{U}_i \tag{7.7}$$

The previous definition means that $\overline{U}_i^*$ is the best strategy that PC_i can take to optimize J_i given the current trajectory of input interconnecting variables $\overline{V}_{in-i}$. With the previous definition we are able to formalize a first notion of agreement among the decisions of different control parts of a graph, i.e. the notion of ***Nash Equilibrium***.

Definition 7.3.4 (Nash Equilibrium)**.** The strategy profile $S^e = \{\overline{U}_1^e, \overline{U}_2^e, \ldots, \overline{U}_N^e\}$ is a Nash Equilibrium iff for each control part PC_i, the selected strategy $\overline{U}_i^e$ is a best response to the input trajectory of interconnecting variables received by that control part.

Since the trajectory of output interconnecting variables from a control part depends on the currently selected strategy by that controller, this means that at the equilibrium no player has a unilateral incentive to deviate from its currently selected strategy if the other players do not deviate from the yours. For completeness in this discussion a Nash equilibrium is a strategy profile in which each player deterministically selects a trajectory of reconfigurations (a.k.a a *pure strategy*). In a more general discussion to which we are not interested here, the notion of Nash equilibrium can be generalized to mixed strategies expressed by a distribution of probability over the set of admissible pure strategies. In this thesis we always refer to pure strategies.

At this point two central issues have to be investigated. The first aspect is the ***existence*** of a strategy profile that represents a Nash equilibrium of the game. The second one is related to the ***convergence*** to a Nash equilibrium: i.e. what conditions need to be satisfied in order to ensure that an interaction protocol reaches an equilibrium strategy profile.

First of all we start our analysis from the first problem. As it is known in Game Theory, pure equilibrium points based on deterministic strategies *not alway exist for any finite*

strategic game. In order to understand this aspect we can model the set of strategy profiles and their best-response relations by a representation called ***Nash Dynamics Graph*** (NDG).

Definition 7.3.5 (Nash Dynamics Graph (NDG))**.** For each finite strategic game G we can define a corresponding directed graph in which:

- each node of the graph corresponds to a precise strategy profile for each control part;
- if the i-th control part responds to the strategy profile $S = \{\overline{U}_1, \overline{U}_2, \ldots, \overline{U}_i, \ldots, \overline{U}_N\}$ with the strategy profile $S' = \{\overline{U}_1, \overline{U}_2, \ldots, \overline{U}_i^{'}, \ldots, \overline{U}_N\}$ such that its local action is changed from $\overline{U}_i$ to $\overline{U}_i^{'}$, then in the graph there exists an arc labeled with i that starts from the node corresponding to S and ends in the node that maps $S^{'}$.

In this representation arcs depict single-player best responses to the strategy profiles of the game. The existence of a Nash equilibrium can be easily verified through the following condition:

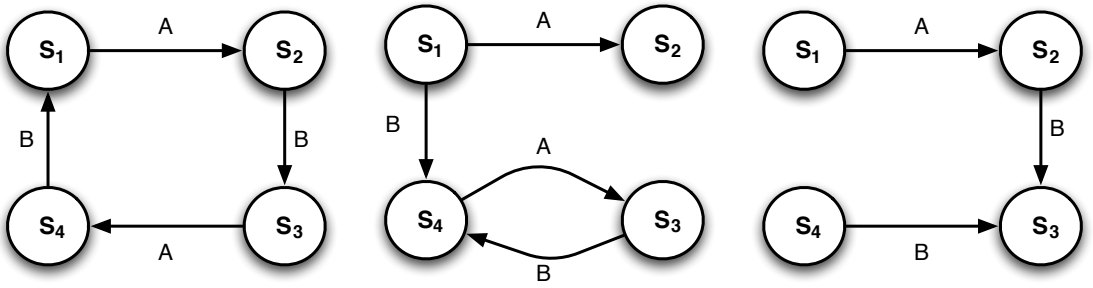
Proposition 7.3.1 (Sink nodes of NDG)**.** *Given a finite strategic game G, a strategy profile S is a Nash equilibrium of the game iff the corresponding node of the Nash dynamics graph is a sink node (i.e. it has no outgoing edges).*

The previous proposition can be simply proved, since given a Nash equilibrium profile no player has an incentive to unilaterally change its action, thus no edge can start from the corresponding node in the NDG. Therefore we have a simple condition for checking if a finite strategic game has at least one Nash equilibrium: we can build the corresponding Nash dynamics graph and check if there exists at least one sink node.

An explicit building of the NDG is not feasible in practise, since its size is exponential in the number of players. Fortunately in many cases, taken proper relaxations of the original finite strategic game, other results help us to be sure that at least one Nash equilibrium exists. This is the case of *continuous games*, in which the set of pure strategies, instead of being finite, may be uncountably infinite. A continuous game will not necessarily have a Nash equilibrium strategy profile. However, if proper conditions on the problem formulation are satisfied, the existence of Nash equilibria can be guaranteed. At this regard an interesting example are continuous games with compact strategy sets and continuous cost functions. The following theorem (sometimes called Rosen's theorem [128] in the literature) states more precise conditions by a proper generalization of the Kakutani fixed point theorem:

Theorem 7.3.2 (Nash equilibrium existence)**.** *Let us consider a continuous game* Γ *that satisfies these properties: (i) the set of possible admissible strategies for each player is a compact and convex Euclidean subspace of* $\mathbb{R}^n$*; (ii) each cost function* J_i *is continuous in the strategy of each player; (iii) the cost function* J_i *is quasi-convex in the strategy of player i. If all the previous conditions are satisfied, the game G has at least one strategy profile which is a (pure) Nash equilibrium of the game.*

Of course this theorem proves the existence of at least one Nash equilibrium, not its uniqueness (i.e. multiple strategy profiles can simultaneously satisfy the Nash optimality conditions).



(a) Case 1. (b) Case 2. (c) Case 3.

Figure 7.11: Nash equilibria existence and convergence of Nash dynamics: nodes are strategy profiles and arcs are labeled with the player that performs an action.

A distributed control approach modeled by a game-theoretic framework is based on a precise optimality criterion: the interaction protocol converges to a Nash equilibrium of the game. This convergence condition needs to be satisfied by a proper construction of the game (especially depending on the cost functions definitions). Figure 7.11 summarizes different situations that can characterize a game in terms of equilibria existence and convergence. For instance in Figure 7.11a is depicted a Nash dynamics graph involving four different strategy profiles and two players (A and B). In this case we can observe that no strategy profile is a Nash equilibrium, since no sink node exists. This means that no interaction protocol based on best responses is able to converge to a stable result: i.e. for each strategy profile there always exists at least one player that has a unilaterally incentive to deviate from the actual profile changing its current strategy. A different situation is depicted in Figure 7.11b. Although in this case a *best response cycle* exists, the convergence to an equilibrium profile can be reached by a proper choice of the initial conditions of the game: i.e. if the initial strategy profile is S_1 and the first player that makes an action is A, we reach the Nash equilibrium profile S_2. In fact we can observe that the best response cycle only involves the profiles S_3 and S_4. Finally Figure 7.11c depicts an ideal situation in which independently from the initial conditions the best response dynamics always reach the equilibrium profile S_3.

7.3.3 A generic interaction protocol between Control Parts

Once the notion of agreement has been established a question arises: how do control parts communicate in order to achieve an agreement? In this section we will propose a generic distributed interaction protocol described through the flowchart of Figure 7.12.

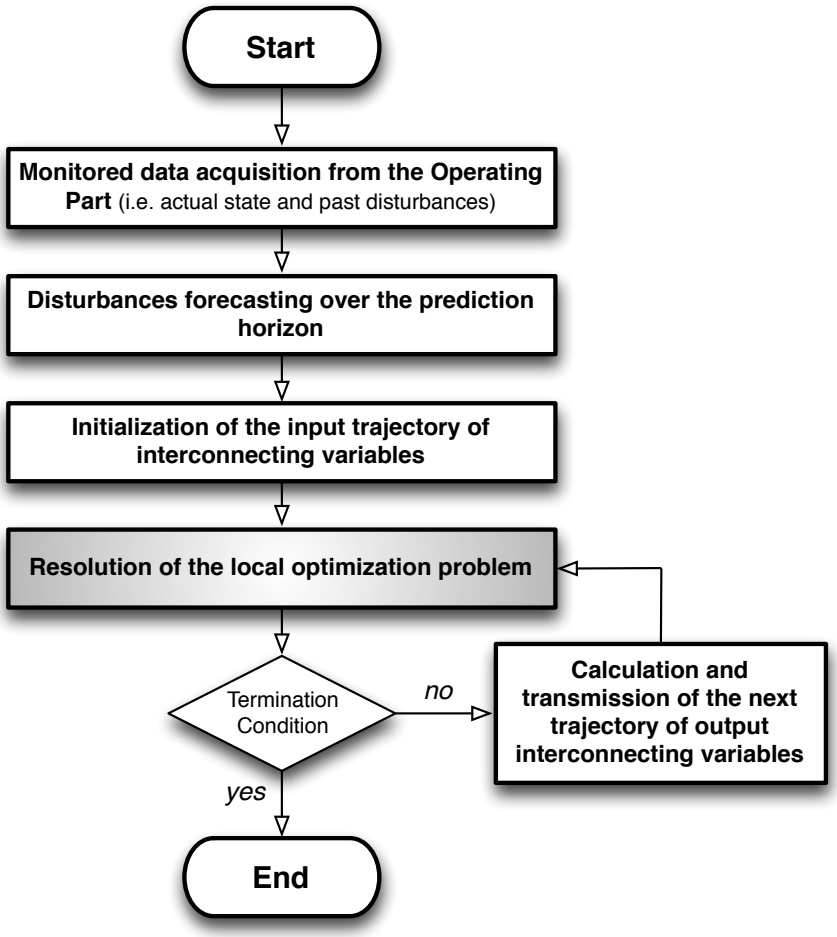


Figure 7.12: Phases of a generic interconnection protocol between control parts.

The basic steps of the protocol, executed in parallel by all the controllers, are described below:

1. PC_i acquires current state variables and past disturbances at the beginning of current control step k;

2. PC_i predicts the future behavior of local disturbances over the prediction horizon. For this phase statistical filtering techniques are applied;

3. PC_i assumes a fixed set of initial values assumed by the trajectory of its input interconnecting variables. In general this trajectory is denoted with $\overline{V}_{in-i}^{(q)}(k)$, where the superscript q corresponds to the current iteration of the protocol (initially $q = 0$);

4. PC_i solves its local optimization problem (7.6) composed of a cost function J_i and a local system model Φ_i. In this way the best trajectory of reconfigurations $\overline{U}_i^{(q)}(k)$

can be identified (also in this case the superscript q corresponds to the iteration number);

5. PC_i evaluates a ***global termination condition*** that needs to be satisfied by all the control parts of the application graph. If this condition holds, the procedure terminates and the first element of the optimal control trajectory will be applied throughout the current step k. Otherwise the procedure goes to the following point;

6. PC_i starts a new iteration $q+1$ and calculates the new trajectory of its output interconnecting variables $\overline{V}_{out-i}^{(q+1)}(k)$. Due to interconnecting constraints (see (7.4)), this means that the input variables to other control parts will change. Then each control part re-evaluates the optimal solution of its local control problem (i.e. the procedure returns to point 4).

We can observe that each iteration of this protocol consists of a *calculation phase*, in which controllers simultaneously solve their local optimization problems, and a *communication phase*, in which each control part transmits the trajectory of its output interconnecting variables to the interested control parts.

The previous distributed algorithm represents a general interaction protocol between controllers. Termination condition states when the interaction protocol can be considered concluded. As an example the protocol can be concluded in just one information exchange among controllers, or in a fixed number of iterations statically known. In some cases the convergence of control actions can be achieved after a number of iterations which is not statically known. For instance a possible termination condition is given below:

$$\overline{U}_i^{(q)}(k) = \overline{U}_i^{(q+1)}(k) \ \forall i = 1,2,\ldots,N \tag{7.8}$$

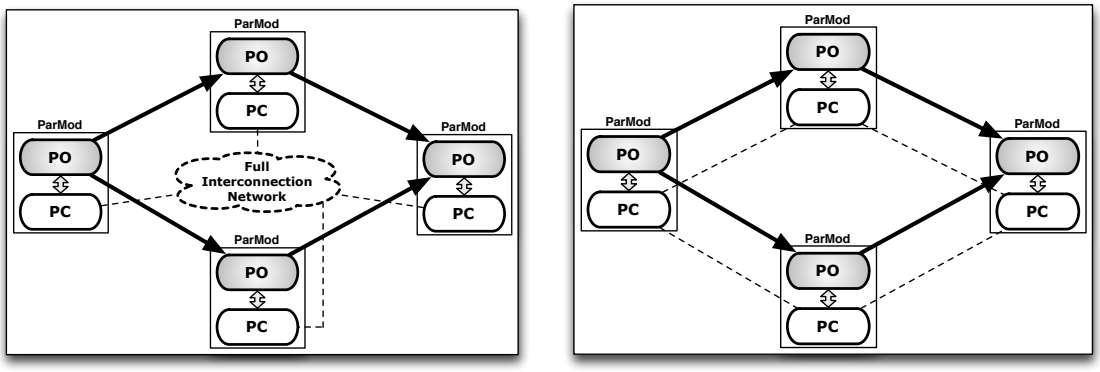
Protocol terminates if for all the controllers the control trajectory selected at a step is the same of the previous step. If the protocol converges to a strategy profile S, this profile is a Nash equilibrium of the game. In fact no ParMod has an incentive to change its trajectory of control inputs given its actual trajectory of input interconnecting variables (so fixed the control actions of the other controllers).

The distributed MPC approach presented so far, based on communications of interconnecting variables and best response strategy, can be defined as follows:

Proposition 7.3.3 (Communication-based Distributed MPC Strategy)**.** *The approach that we have introduced is a* ***Communication-based Distributed MPC****. Controllers of different ParMods communicate exchanging trajectories of interconnecting variables and the notion of agreement of control decisions follows the formal foundation of Nash optimality.*

Moreover this protocol can exploit different interconnections between control parts:

- we can consider a fully-interconnected scheme, in which the interaction protocol is applied considering each control part interconnected with all the other control parts



(a) Fully-interconnected scheme. (b) Partially-interconnected scheme.

Figure 7.13: Possible interconnections among control parts of an application graph.

of the graph. This means that operating parts (sub-systems) are interconnected following the application graph structure characterized by streams of data between parallel modules, whereas each control part is interconnected to every other controller. More formally $I_c[i, j] = 1 \; \forall i, j = 1, 2, \ldots, N \; i \neq j$;

- a second choice considers a partially-interconnected scheme, in which two controllers are directly interconnected iff a data stream exists between the corresponding operating parts: i.e. $I_p[i, j] = 1 \iff \big(I_c[i, j] = I_c[j, i] = 1\big)$.

Figure 7.13 shows the different interconnections for the same application graph composed of four ParMods.

At this point of the description some issues remain opened. Although we have described a useful result (Theorem 7.3.2) for proving the existence of a Nash equilibrium profile of a game, we need to provide a game formulation, in terms of particular properties of cost functions and local models, such that the interaction protocol described in this section converges to a Nash equilibrium. These issues will be addressed in the next chapter, when a specific example of distributed control of distributed parallel applications will be described.

7.3.4 Self-interest and Cooperation: different perspectives

So far the interactions between control parts have been described in terms of a framework based on the concept of best response to a set of interconnecting variables. This solution is based on a underlying assumption which is the *selfish behavior* of the controllers. This can be summarized in the following points:

- in our scheme each controller of the computation graph is a rational agent that optimizes its local objective function. Control parts only know their local functions and models, and not the ones of the other controllers;

- each controller, received the trajectory of input interconnecting variables (that depends on the control decisions selected by the other control parts), takes the best response optimizing its local objective, without taking into account the effects of its control actions (reconfigurations) on the other controllers.

A critical aspect of this approach is the global (plantwide) optimality reached with this control scheme. The problem of taking reconfiguration actions can be represented as a *multi-objective optimization problem* in which the system is aimed at optimizing a set of local objectives $J_1, J_2, \ldots, J_N$ at each control step of the execution, through proper information exchanges between a set of localized controllers. The *utopia point* is obtained by optimizing each local objective without accounting for other objective functions. It is a rare case that all the objectives are minimized by the same solution point, so in general the utopia solution only exists ideally and it is not usually admissible. However there exists a set of optimal solutions that correspond the the best trade-offs between the different objectives. In order to explain better this concept we introduce a partial-order relation between strategy profiles:

Definition 7.3.6 (Pareto domination)**.** Given two strategy profiles S and S', we say that S dominates S' iff with the strategy profile S at least one local objective is better off and no one is worse off than in the strategy profile S'.

This means that in a dominated strategy profile S' there exists at least one control part that has the incentive to change its reconfiguration trajectory without making the other local objectives worse off. Therefore in a multi-objective optimization problem there is not a unique optimal solution, but a set of non-comparable optimal solutions:

Definition 7.3.7 (Pareto optimality)**.** A strategy profile S is ***Pareto optimal*** iff there not exists any strategy profile that dominates it.

The set of optimal solutions of a multi-objective problem is also called *Pareto set* or frontier. The concept of Pareto optimality is extremely important for our purposes. Ideally we desire a distributed control scheme such that the strategy profile selected at the end of the interaction protocol is a Pareto optimum, which indicates that an optimal trade-off between local objectives of different parallel modules has been achieved. Moreover, in many situations, we can be interested in reaching a specific Pareto optimum that retains certain properties. A relevant example is when we translate a multi-objective optimization problem into a unique global objective function of the whole system (i.e. a ***plantwide objective function*** J_G), e.g. obtained by a linear combination of the local objectives of each control part:

$$J_G = \sum_{i=1}^{N} w_i J_i \tag{7.9}$$

The parameters w_i indicate proper weights applied to local objectives such that $\sum_{i=1}^{N} w_i = 1$. The construction of this function usually requires a proper knowledge of the system, in order to correctly establish preferences among local objectives (e.g. otherwise weights can be uniformly distributed among controllers). An important concept is given below:

Definition 7.3.8 (Social optimum)**.** A strategy profile $S^{(soc)}$ is a ***social optimum*** iff it optimizes the value of the provided weighted sum J_G of local objectives.

We can note that this special strategy profile belongs to the Pareto set, i.e. it is a Pareto optimum. Let us suppose by absurd that there exists a strategy profile S that dominates a social optimum strategy profile $S^{(soc)}$. This means that with S it is possible to improve the value of a local objective without making the others worse off, thus the global function J_G with S assumes a better value than with $S^{(soc)}$, which is absurd since $S^{(soc)}$ is the social optimum.

It is a well-known result in Game Theory that a Nash equilibrium of a game may not be a Pareto optimum and, thus, it may be different from the social optimum strategy profile. This inefficiency is due to the selfish behavior of the controllers: control parts can globally improve the plantwide objective function if they select control actions taking into account not only the local outcome of their decisions, but also the effects of their actions on the other controllers. A notion of efficiency measures how far is the outcome of the best Nash equilibrium profile $S^{(e)}$ (that gives the best value of J_G among the Nash equilibria) compared to the social optimum $S^{(soc)}$:

$$PoS = \frac{J_G(S^{soc})}{J_G(S^{(e)})} \tag{7.10}$$

which is called ***Price of Stability*** (PoS). Now a question is: how is it possible to modify the distributed MPC scheme in order to reach such social optimum profile?

Several research work [124] has faced with the problem of developing MPC approaches able to reach the social optimum of the problem, such that the quality of the distributed control strategy is equal to the one of a centralized controller. Techniques for reaching this objective are based on theoretical foundations of *distributed optimization*, in which a centralized optimization problem is decomposed into a set of sub-problems of a manageable size and complexity. Each sub-problem is assigned to a distinct controller that solves it in a cooperative way: i.e. having in mind the global objective function of the system. The common principle consists in an information exchange between controllers, that compromise their individual choices towards the global benefit instead of solving each sub-problem in a selfish way. This phase requires to achieve a global consensus between controllers, assuring that the reached solution is the globally optimal one.

Different cooperation schemes can lead cooperative optimization algorithms of completely different computational behaviors and amenable to solve problems with specific structures (e.g. decomposable optimization problems in which the global cost function separates on the different control variables) or mathematical features (e.g. techniques for smooth convex optimization w.r.t other approaches for non-differentiable objective functions). Despite the large set of distributed optimization methods, we can identify a general concept:

Proposition 7.3.4 (Cooperative Distributed MPC Strategy)**.** *In a **Cooperative Distributed MPC** approach controllers exchange control information and perform local computations*

until they collectively reach an approximation of the social optimum strategy profile: i.e. a set of control decisions that are an approximation of the plantwide optimal solution of the equivalent centralized control problem.

Due to the presence of several distributed optimization techniques, we postpone the presentation of a precise cooperative approach in the next chapter, when an example of cooperative control for distributed parallel applications will be presented.

7.4 Summary

The problem of controlling graphs of distributed adaptive parallel modules consists in decomposing the centralized control of the entire application in a set of controllers (one per ParMod), that applying their adaptation strategies to optimize a local objective function based on a local knowledge of the sub-system behavior. Due to coupling relationships, control actions selected by a controller can influence the control quality of the other controllers of the graph. This means that controllers communicate in order to establish an agreement in their control decisions. In this chapter we have described a set of theoretical concepts to model this notion of agreement. Some basic results in Game Theory, as the notion of Nash equilibrium and its existence, have been provided mapping a classic game-theoretic formulation onto the extended ParMod model with interconnecting variables. In the next chapter these concepts will be concretized applying the distributed control model to a real situation in which we optimize the performance behavior and the resource utilization cost of distributed parallel computations.

Chapter 8

Distributed Control of Performance and Resource Utilization of ParMod Graphs

In this chapter we will present different distributed control solutions applied to the problem of regulating the performance behavior and the resource utilization of distributed parallel applications. As the first case we will apply the communication-based distributed MPC strategy in which control parts interact pursuing their self-interest. At this regard we will show the existence of equilibrium strategy profiles and how to reach the best equilibrium. We will also discuss the main advantages of a selfish behavior, especially related to the simplicity of control part interaction and the quickness to find an equilibrium profile. On the other hand this modeling is characterized by a potentially limited optimization of the global application behavior, as the adaptation logic of each ParMod is individually rational. To this end we will discuss an alternative cooperative distributed MPC approach that, though more expensive in terms of number of controller interactions and convergence speed, is able to achieve a better optimal control of the entire application.

8.1 Problem description and QoS modeling

In this chapter we will consider an interesting case in which we control a distributed parallel application regulating the achieved performance level and the actual cost in terms of resource utilization. Performance and resource optimization is an important research problem in distributed contexts as Grids and Clouds [129, 130, 131]. As we have seen in the examples described in Chapter 6, the resource requirements of an application are inherently varying during the execution. This may be due to application-dependent reasons (e.g. the computational weight of received tasks can increase or decrease significantly due to causes related to the application semantics) or due to dynamic platform conditions (e.g. the arrival rate of tasks may be conditioned by the actual behavior of the interconnection network between application components). These scenarios require resource management and runtime adaptation techniques that find proper trade-offs between performance level

and resource utilization.

All these issues become increasingly challenging if we consider adaptive applications that are not composed of a single centralized component, as for the examples of Chapter 6 in which the adaptation logic was concentrated into a single adaptive ParMod. In the previous chapter we have seen that the control logic of a distributed application has to be distributed over the set of application modules. In this case the difficulty to organize proper trade-offs between performance level and resource usage is much more complicated, as we also need to account for the coordination between control logics of different sub-systems instead of limiting the reconfiguration decisions to the local knowledge of each sub-system behavior.

As hinted in the previous chapter, for our control scenarios the main coupling relationships between control sub-problems are the ones imposed by the performance behavior of computation graph structures. In Chapter 4 we have formalized an important concept which is reported below:

Remark 8.1.1. *The steady-state performance behavior of a computation module depends on its internal configuration (e.g. its actual degree of parallelism) but also on the performance behavior of the other parallel modules of the graph structure in which the ParMod resides.*

We know that these coupling effects are noticed as performance degradations: i.e. due to the interconnections with other application modules and the blocking semantics of communications, the performance behavior of a parallel module at steady-state can be worse off w.r.t the ideal behavior of the ParMod considered in isolation. This means that the control logics communicate and coordinate with each other in order to determine the effective performance level of each module.

In this chapter we will study the exploitation of the distributed model-based predictive control strategy characterized by the following points:

- we will consider the control of acyclic graphs starting from particular structures (e.g. pipeline and functional-partitioning structures);
- ParMods of the application graph will communicate exchanging sequence of tasks and results (i.e. stream-based applications);
- for each ParMod we will suppose that the QoS objective consists in regulating the effective performance level and the resource utilization in terms of number of nodes reserved for the parallel computation;
- we will suppose that the only class of reconfigurations that each ParMod is able to exploit consists in changing the number of nodes currently involved in the computation (i.e. non-functional reconfigurations);
- the predictive formulation of the model-based predictive control strategy will adopt the shortest prediction horizon length, equal to 1 control step;

- the control structure follows a single-layer distributed scheme in which each ParMod is composed of an operating part and a control part and the control parts of different ParMods communicate exchanging interconnecting variables;
- due to the distributed nature of the application, we will suppose that each ParMod is executed on a dedicated execution environment featuring a set of homogeneous computing nodes.

In the next section we will describe the ParMod modeling for this control problem.

8.1.1 Application of the Distributed MPC Strategy

As stated above in this problem we are exclusively interested in controlling the parallelism degree of each ParMod through the execution of non-functional reconfiguration processes. For each $ParMod_i$ we introduce proper sets of model variables. A control input vector $\mathbf{u}_i(k)$ is uniquely identified by a single scalar value $n_i(k)$ that indicates the parallelism degree adopted by $ParMod_i$ throughout the k-th control step of the execution. This control input takes the following set of admissible values:

$$\mathbf{U}_i = \left\{ n_i(k) \middle| n_i(k) \in \mathbb{N} \wedge 1 \leq n_i(k) \leq n_i^{max} \right\}$$

where n_i^{max} is the maximum parallelism degree (it depends on the number of available nodes in the $ParMod_i$ execution architecture).

The disturbance input vector $\mathbf{d}_i(k)$ is identified by a single real-valued parameter $T_{calc-i}(k)$ that denotes the average value of the calculation time per task for the k-th control step. As we have seen disturbance prediction is exploited by the control part through statistical techniques based on past historical observations. Therefore the actual calculation time experienced during each control step needs to be sensed by the control part in order to keep updated the history window used for making the predictions. We can now ask how this information can be obtained from the resources (emitter, collector and workers) that are currently applying a structured parallel computation. Just to clarify things let us considering the following cases:

- in a task-farm scheme each worker applies the same computation sequentially on each received task. Parallelism is achieved by executing different tasks on distinct worker entities simultaneously. In this case each worker is able to monitor the average value of its calculation time and provide this information as monitored data to the control part at the beginning of each control step;
- in a data-parallel scheme, both with stencil patterns or in the map case, parallelism is achieved by partitioning the current input task and assigning each partition to an available worker. In this case each worker can calculate the average time required to apply the computation on its partition or, alternatively, the average size of received partitions if the task size is variable during the execution (as for the example

described in Section 6.2). With this information for the control part is relatively simple to quantify the mean calculation time per task.

State variables describe the actual performance behavior of $ParMod_i$. For a stream-based computation that processes a large set of input tasks, it is extremely important to remark the differences between the following two measurements, as explained in Chapter 4:

- the performance behavior of a parallel module can be measured in isolation, i.e. without considering its interaction with other modules of the graph. In this case the performance achieved depends exclusively on its internal configuration (e.g. in this case the parallelism degree). This ideal behavior is modeled by the concept of ***mean service time*** of a parallel module;
- the effective performance behavior of a ParMod can be measured in terms of its ***mean inter-departure time***: i.e. the steady-state average time between the transmission of two subsequent results to other modules of the application graph. This average measurement takes into account the ideal behavior of the module (its service time) but also the performance degradation due to the blocking semantics of inter-module communications. This means that *the mean inter-departure time of a module is always greater or at most equal to its mean service time*[1].

In this problem we are interested in controlling the performance level and the resource utilization cost for each parallel module. Therefore we consider a QoS variable that models the actual effective performance level of a ParMod computation: i.e. a real-valued variable $T_{p_i}(k)$ that denotes the mean inter-departure time of results from $ParMod_i$ at the beginning of control step k.

In order to exploit the MPC strategy we define an objective function that the control part optimizes at each sampling interval based on its model and disturbance predictions. In this example we want to address the general problem of making proper trade-offs between the achieved performance level and the resource utilization cost. For modeling the resource utilization, we exploit a simple cost model in which the resource cost proportionally depends on the number of nodes settled for the ParMod computation. In a similar way we also consider a cost proportional to the performance measurement: i.e. a higher inter-departure time (e.g. corresponding to a slower performance behavior), is associated with a higher cost compared to lower inter-departure times. Therefore the ***local cost function*** for each $ParMod_i$ has the following structure:

$$J_i(k) = \alpha_i T_{p_i}(k+1) + \beta_i n_i(k) \tag{8.1}$$

where α_i and β_i are two positive proportional coefficients for the performance level and resource utilization cost. It is worth noting that these two coefficients can be different

[1] In the rest of the description we will often omit the adjective "mean", that should always be considered implicitly.

between ParMods. As an example it is possible that some computing environments (e.g. as Clouds with pay-per-use billing models) feature a higher resource cost than other distributed computing platforms.

8.2 Analytical description of a Self-interest interaction between Controllers

In this section we will apply the communication-based distributed MPC approach (presented in Section 7.3.4) in which each control part selects the best parallelism degree optimizing its local cost function (selfish behavior). Since this problem formulation is feasible for a one-step ahead predictive control, control input, state and disturbance trajectories are reduced to be single measurements for the current control step. This means that at each control step each control part chooses an ***optimal parallelism degree*** which is the best response to the currently received set of interconnecting variables from the other controllers. Let us analyze the performance coupling relationships in the simple case of pipeline graph structures.

8.2.1 The case of pipeline graphs: performance coupling relationships

In Section 4.1.1.1 we have seen that a pipeline graph is an array of N stages with a source node and a sink node at the beginning and at the end of the chain. The pipeline structure parallelizes the computations of different tasks by operating on them concurrently in different stages, thus improving the application throughput in terms of number of completed tasks per time unit w.r.t a monolithic implementation consisting in a single sequential module.

The decomposition of a complex computation in a sequence of well-identified phases is a technique that emerges in many real-world scenarios. Data streaming applications such as graphic computations, multimedia and image-processing applications are notable examples in which a decomposition into multiple computational phases is a straightforward way to improve the throughput especially in real-time contexts. Other interesting examples are signal processing systems (as parallel implementations of the Space-time Adaptive Processing) in which, due to tight real-time constraints, they require an internal parallelization of each phase through other parallelism paradigms, as functional replications (e.g. task-farm) or data parallelism schemes.

Therefore in this example we consider the general case of a pipeline graph in which each stage is an adaptive parallel module that exploits parallelism in the most appropriate way. Let us consider the i-th stage of the pipeline. As said the disturbance variable $T_{calc-i}(k)$ measures the mean calculation time throughout the k-th control step, that will be predicted step-by-step by using history-based techniques based on time-series analysis. The service time of a parallel computation is determined by considering the module

in isolation, i.e. neglecting its interactions with other application ParMods. Independently from the used parallelism scheme, we assume that the mean service time $T_{S_i}(k)$ of $ParMod_i$ can be determined by the following relation:

$$T_{S_i}(k) = \frac{T_{calc-i}(k)}{n_i(k)} \tag{8.2}$$

where $n_i(k)$ is the parallelism degree that the parallel module will exploit for the k-th control step. In other words for the sake of simplicity we will make a *perfect scalability* assumption: i.e. we will assume that the structured parallel computation performed by a ParMod scales perfectly. Certainly this ideal behavior is not always possible in practise, and a parallel computation may not scale ideally due to workload unbalance, synchronizations and architectural limits of the underlying execution platform. For this reason later in this chapter (in Section 8.4.2) we will discuss possible non-ideal modelings of the service time, that however will not significantly modify the description presented in this chapter.

As it has been described in Chapter 4, the service time of a parallel module coincides with its effective performance achieved at steady-state if and only if the module is a bottleneck in the graph structure. Otherwise its effective behavior, i.e. the inter-departure time, will be higher than the service time, and it depends on the particular structure of the graph that we are analyzing. In the case of pipeline graphs, as demonstrated in Proposition 4.1.2, the inter-departure time from any stage coincides with the maximum service time of all the ParMods. Therefore for pipeline graphs the static performance model is the following:

$$T_{p_i}(k+1) = \max_{j=1,\ldots,N} \left\{ T_{S_j}(k) \right\} \tag{8.3}$$

where $T_{S_j}(k)\ \forall j = 1,2,\ldots,N$ and $j \neq i$ are the service times of all the other stages of the pipeline graph. Since these variables refer to external information w.r.t $ParMod_i$, they are input interconnecting variables of the i-th control part model (i.e. $\mathbf{v}_{in-i}(k) = [T_{S_1}(k),\ldots,T_{S_{i-1}}(k),T_{S_{i+1}}(k),\ldots,T_{S_N}(k)]^T$), whereas $T_{S_i}(k)$ is the output interconnecting variable whose generation function is given by the expression (8.2). Therefore the local cost function J_i can be rewritten as follows:

$$J_i(k) = \alpha_i \max_{j=1,\ldots,N} \left\{ T_{S_j}(k) \right\} + \beta_i n_i(k)$$

Although very simple this model has a set of interesting features. Coupling relationships between control problems are limited to the performance modeling, since resource cost only depends on the local parallelism degree. The inter-departure time from each ParMod depends on the local parallelism degree but it is also influenced by the service times of all the other ParMods. In this first formulation of the problem we will assume that at each control step each controller exchanges interconnecting variables with all the other control parts. In other words:

Assumption 8.2.1. *We will assume a fully-interconnected structure between control parts: i.e. every control part is directly interconnected with all the other control parts of the computation graph.*

An example of a pipeline graph with a fully-interconnected network between control parts is depicted in Figure 8.1.

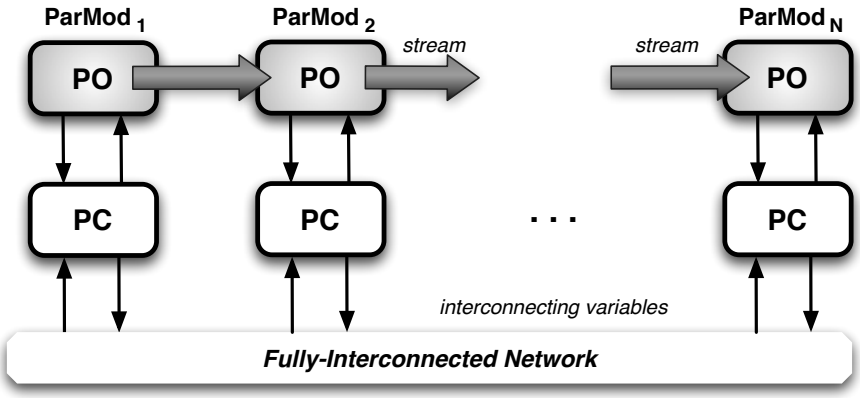


Figure 8.1: Pipeline distributed control with a complete interconnection network between control parts.

8.2.1.1 Existence and identification of Equilibrium strategy profiles

Let us consider this problem formulation in more detail. The local cost function (8.1) expresses a linear resource utilization cost proportional to the parallelism degree, and a cost that depends on the effective performance level achieved by the ParMod. First of all it is useful to understand the cost behavior of the ParMod in isolation, i.e. neglecting all the coupling relationships between control sub-problems.

In isolation the mean inter-departure time from a ParMod coincides with its mean service time. Therefore we can rewrite the cost function in the following way:

$$J_i(k) = \alpha_i T_{S_i}(k) + \beta_i n_i(k) = \alpha_i \frac{T_{calc-i}(k)}{n_i(k)} + \beta_i n_i(k)$$

The first term indicates a cost related to the current service time of the module. Higher service times, that correspond to lower parallelism degrees, will be associated to a higher cost. Thus between the first cost and the parallelism degree of $ParMod_i$ there exists an inverse proportional relationship. On the other hand the second cost is proportional to the number of currently used nodes.

In order to make our analysis tractable, we treat the parallelism degree $n_i(k)$ as a positive real-valued number. Of course this *continuous relaxation* is not admissible in practise: the parallelism degree strictly indicates an integer quantity. We will discuss

the impact of the integrality constraint on our problem formulation later in this chapter. With this assumption the finite strategic game between control parts becomes a continuous game (see Section 7.3.2) in which the behavior of the cost function J_i in isolation is shown in Figure 8.2.

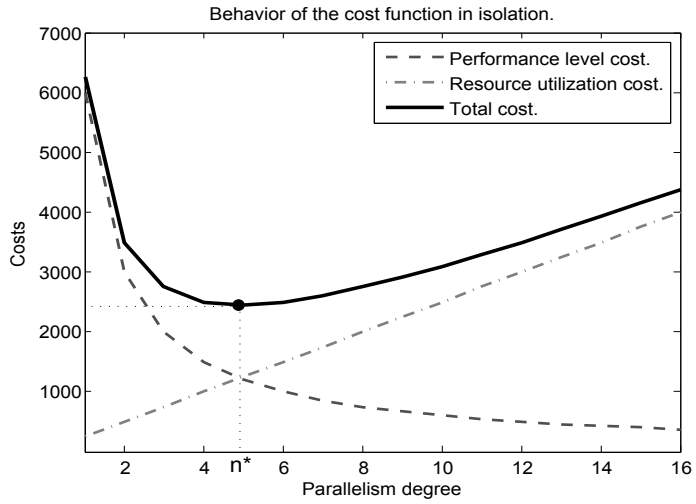


Figure 8.2: Local cost function J_i of a ParMod: behavior of the total cost in function of the parallelism degree.

First of all we consider the second order derivative of the local cost function:

$$\frac{d^2 J_i}{d\, n_i} = \frac{2\,\alpha_i\, T_{calc-i}}{n_i^3} > 0$$

for clarity of presentation sometimes we will omit to indicate the current control step index k from disturbance, state and control variables.

We can observe that the second order derivative is always positive, since the unitary cost α_i and the calculation time T_{calc-i} are positive quantities and the domain of the parallelism degree is the closed interval $[1, n_i^{max}]$. Therefore the function J_i is *convex* and the optimization problem can be simply solved finding the parallelism degree value such that the first order derivative of the cost function is nullified.

$$\frac{d\, J_i}{d\, n_i} = -\frac{\alpha_i\, T_{calc-i}}{n_i^2} + \beta_i = 0$$

We are interested in the positive solution of this second order equation, since only positive values for the parallelism degree are meaningful. Therefore the optimal parallelism degree in isolation is given by:

$$n_i^*(k) = +\sqrt{\frac{\alpha_i\, T_{calc-i}(k)}{\beta_i}} \tag{8.4}$$

This parallelism degree is especially important for our purposes: it expresses the optimal trade-off between performance level and resource cost when the ParMod has no knowledge about the performance behavior of the other modules of the graph. This concept is further clarified by the following definition:

Definition 8.2.1 (Ideal Parallelism Degree)**.** The ***ideal parallelism degree*** of $ParMod_i$ is a value $n_i^*(k)$ such that the local cost function of the module is minimized in isolation, i.e. completely neglecting the performance coupling relationships with the other modules of the graph structure in which the ParMod resides.

The value denoted by expression (8.4) can be higher than the maximum number n_i^{max} of nodes available for $ParMod_i$ execution. In this case, due to the convexity of the cost function J_i, the first order derivative is negative in the interval $[1, n_i^{max}]$, thus the cost is monotonically decreasing and the ideal parallelism degree is given by the maximum number of available nodes, i.e. $n_i^* = n_i^{max}$. Without loss of generality in the rest of the description we will assume that for each module the expression (8.4) always gives an admissible value.

So far we have avoided to consider the presence of coupling relationships among control sub-problems. As stated before the effective performance level achieved by a parallel module in a pipeline structure is given by the maximum between the service times of all the stages of the graph. This means that the control decisions taken by the other controllers influence the optimal parallelism degree that a control part will select.

In this example it is relatively simple to understand what will be the optimal parallelism degree of a module in function of the current values of the received interconnecting variables. We have seen that, at the beginning of each control step, the controllers exchange their actual service times with each other. For $ParMod_i$ let us denote the following variable:

$$T_{S_{max,i}}(k) = \max_{j=1,\ldots,N \; j \neq i} \left\{ T_{S_j}(k) \right\}$$

that indicates the maximum value of the input interconnecting variables currently received by the control part of $ParMod_i$. In a pipeline graph this value has a crucial meaning: due to the current performance behavior of the other modules, it is not possible for $ParMod_i$ to achieve a mean inter-departure time lower than $T_{S_{max,i}}(k)$. Therefore this control part has to account for this constraint when it decides its optimal parallelism degree.

Another important parameter is given by the following parallelism degree:

$$\tilde{n}_i(k) = \frac{T_{calc-i}(k)}{T_{S_{max,i}}(k)} \tag{8.5}$$

that indicates the parallelism degree that allows the $ParMod_i$ to reach a service time equal to the maximum service time currently advertised by the other control parts.

At this point let us analyze the possible best responses to the actual received interconnecting variables by a generic PC_i. We can identify two possible situations:

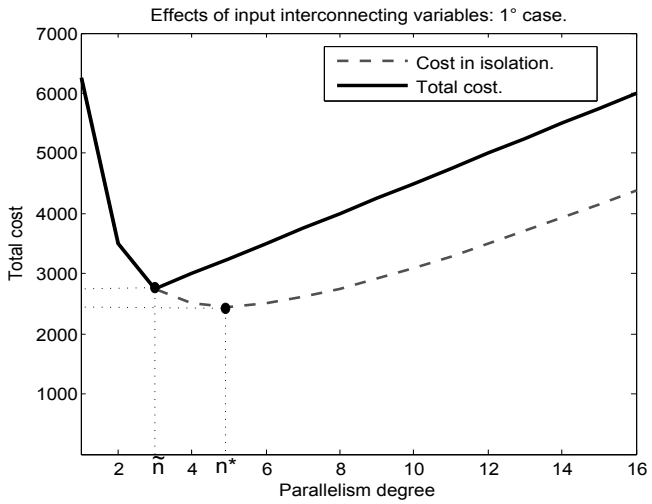


Figure 8.3: Effects of interconnecting variables: first case.

First case $\tilde{\mathbf{n}}_\mathbf{i} < \mathbf{n}_\mathbf{i}^*$: due to the presence of input interconnecting variables, the local cost function J_i becomes a piecewise-defined function which is continuous in the interval $[1, n_i^{max}]$. Its definition is given by:

$$J_i = \begin{cases} \alpha_i \dfrac{T_{calc-i}}{n_i} + \beta_i n_i & \text{if } 1 \leq n_i \leq \tilde{n}_i \\ \\ \alpha_i T_{S_{max,i}} + \beta_i n_i & \text{if } n_i > \tilde{n}_i \end{cases} \tag{8.6}$$

The first piece of the function corresponds to a the situation in which $ParMod_i$ is a bottleneck: i.e. its service time is greater than the maximum service time of all the other pipeline stages. In this case increasing the parallelism degree n_i up to the value $\tilde{n}_i$ will result in a lower cost (i.e. the cost function is monotonically decreasing in this first part of the domain). In the second piece we have a different behavior: $ParMod_i$ is not a bottleneck anymore because its service time is lower than the maximum service time of the other modules of the graph. This implies that increasing the parallelism degree more than $\tilde{n}_i$ is completely useless: we would increase the resource utilization cost without achieving any performance improvement at steady-state. This behavior is depicted in Figure 8.3. From a game-theoretic viewpoint this case implies that the parallelism degree $\tilde{n}_i$ is the best response to the currently received interconnecting variables, i.e.:

$$J_i\Big(\tilde{n}_i(k), \mathbf{v}_{in-i}(k)\Big) \leq J_i\Big(n_i(k), \mathbf{v}_{in-i}(k)\Big) \quad \forall n_i(k) \in \mathbf{U}_i$$

Second case $\tilde{\mathbf{n}}_\mathbf{i} \geq \mathbf{n}_\mathbf{i}^*$: in the second case the cost function is always the piecewise-defined function denoted by equations (8.6). As before the second piece of the function is monotonically increasing. On the other hand the cost behavior in the first piece

is different w.r.t the first case. In fact now J_i is non-monotone in the sub-domain $[1, \tilde{n}_i]$, since the ideal parallelism degree n_i^* is smaller than the parallelism degree $\tilde{n}_i$ (Figure 8.4 shows this behavior). Therefore in this case the best response to the received interconnecting variables is represented by ideal parallelism degree n_i^*, i.e.:

$$J_i\Big(n_i^*(k), \mathbf{v}_{in-i}(k)\Big) \leq J_i\Big(n_i(k), \mathbf{v}_{in-i}(k)\Big) \quad \forall n_i(k) \in \mathbf{U}_i$$

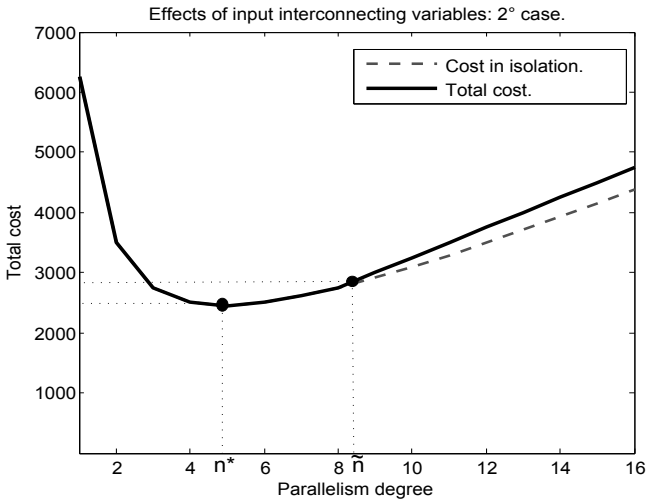


Figure 8.4: Effects of interconnecting variables: second case.

As explained in Section 7.3.2 we can model the interaction between control parts of the pipeline stages through a game-theoretic approach in which equilibrium solutions correspond to the concept of Nash optimality. We can prove that:

Proposition 8.2.2. *In the problem of distributed control of pipeline graphs there exists at least one strategy profile which is a Nash equilibrium of the corresponding game between control parts.*

Proof. This result can be proven using Theorem 7.3.2. This theorem requires to prove the fulfilment of a certain set of properties of the game. The first property concerns the admissible set of strategies of each player (control part), which has to be a compact and convex Euclidean sub-space. Since the strategy set of a controller is the closed interval $[1, n_i^{max}]$ of real numbers, we can observe that:

- it is a sub-space of $\mathbb{R}$;
- a single interval of real values is a convex set;
- the interval is also compact since it is closed (it contains its boundary elements) and it is limited.

The second property concerns the cost function J_i of each player. Theorem 7.3.2 requires that each function is continuous in the strategies of the players. Since in our model we have represented the controller relationship with interconnecting variables, we can expand the service time expressions in order to directly express the cost functions in terms of the parallelism degrees of each control part. For the generic J_i we have:

$$J_i(k) = \alpha_i \max_{j=1,\ldots,N} \left\{ \frac{T_{calc-j}(k)}{n_j(k)} \right\} + \beta_i n_i(k)$$

We recall that the domain of each variable $n_i(k)$ is the closed interval $[1, n_i^{max}]$. Therefore the critical point 0 in not admissible (the parallelism degree is meaningful only for values greater or equal to 1) and thus each term of the maximum function is continuous in its admissible domain. Furthermore, since the pointwise maximum of continuous functions is also continuous, and the sum of two continuous functions is continuous too, we can conclude that the cost function J_i is continuous in the strategy of each player.

Finally, theorem 7.3.2 requires to prove another property of the cost function. Fixed the parallelism degrees of all the other control parts, we need to verify that the function J_i is quasi-convex in the parallelism degree $n_i(k)$. A single-variable function is quasi-convex in the domain $[1, n_i^{max}]$ iff there exists a value n_0 in its domain such that:

- the function is not increasing for each $n_i \in [1, n_i^{max}]$ and $n_i < n_0$;
- the function is not decreasing for each $n_i \in [1, n_i^{max}]$ and $n_i > n_0$.

As we have seen this property is verified by our cost functions (see the two cases described before). The value n_0 is the parallelism degree $\tilde{n}_i$ in the first case or the ideal parallelism degree n_i^* in the second one. Therefore we can conclude that our distributed control problem admits at least one Nash equilibrium, i.e. a set of parallelism degrees in which no controller can choose a better parallelism degree given the control decisions of the other control parts of the graph. □

The previous result shows the existence of Nash equilibria in our game. At this point we can effectively identify what are these equilibria and their properties.

Proposition 8.2.3. *In the pipeline distributed control game a vector of parallelism degrees* $\mathbf{s}^{(e)} = [n_1^{(e)}, n_2^{(e)}, \ldots, n_N^{(e)}]^T$ *such that the following conditions are satisfied:*

$$\begin{cases} \dfrac{T_{calc-i}}{n_i^{(e)}} = \dfrac{T_{calc-j}}{n_j^{(e)}} \ \forall i, j = 1, 2, \ldots, N \\ \\ n_i^{(e)} \leq n_i^* \ \forall i = 1, 2, \ldots, N \end{cases}$$

is a (pure) Nash equilibrium of the game.

Proof. The previous proposition denotes that a strategy profile such that: (1) all the stages of the pipeline are balanced with each other (i.e. they have the same service time) and (2) no ParMod exploits a parallelism degree greater than its ideal parallelism degree, is a Nash equilibrium of the game. The first condition implies that each ParMod has not a unilateral incentive to increase its parallelism degree, since its effective performance can not be made better off if the other ParMods do not change their current parallelism degrees. The second condition implies that each ParMod can not be in the situation described in Figure 8.4. In other words if $n_i^{(e)} \leq n_i^*$ no control part has a unilateral incentive to decrease its current parallelism degree. Therefore we can conclude that a strategy profile with these properties is certainly a Nash equilibrium of the game. □

It is worth noting that multiple strategy profiles can retain the previous properties, therefore multiple Nash equilibrium profiles exist.

In Chapter 7 we have described the Pareto domination which is an important relation to compare different strategy profiles. As we known Pareto efficiency captures the idea that a strategy profile is inefficient iff it is possible to achieve an improvement of some cost functions without making the others worse off. It is well-know that a Nash equilibrium profile can be Pareto inefficient. This means that albeit we have found an equilibrium profile (so no controller has a local benefit to change its currently selected parallelism degree), if controllers coalize with each other they can reach a non-worse outcome for each of their local cost functions.

In our example we are interested in identifying among the set of Nash equilibria the ones that are also Pareto efficient. We recall that a Nash equilibrium is efficient iff it is Pareto optimal. In our example the efficiency of a Nash equilibrium is given by the following condition:

Proposition 8.2.4 (Efficient Nash Equilibrium)**.** *In the game modeling the distributed control of pipeline graphs, an equilibrium strategy profile* $\mathbf{s}^{(e)} = [n_1^{(e)}, n_2^{(e)}, \ldots, n_N^{(e)}]^T$ *such that the conditions of Proposition 8.2.3 are satisfied is also Pareto efficient iff there exists at least one control part that it choosing a parallelism degree equal to its ideal parallelism degree, i.e:*

$$\exists i = 1, 2, \ldots, N \text{ such that } n_i^{(e)} = n_i^*$$

Proof. The proposition can be easily proved. As we have seen if all the stages are balanced and no control part PC_i is choosing a parallelism degree greater that n_i^*, this means that no player has a unilaterally incentive to change its current control choice. If, besides these conditions, there is also a controller that is choosing exactly its ideal parallelism degree, this means that any deviation from that value will certainly increase the corresponding local cost function. Therefore this equilibrium strategy profile can not be dominated by any other strategy profile, thus it is Pareto efficient. □

Proposition 8.2.5 (Uniqueness of the Efficient Nash Equilibrium)**.** *In the pipeline game there exists a unique Nash equilibrium profile which is also Pareto efficient.*

Proof. Let us suppose that $ParMod_p$ is the module such that its service time with the ideal parallelism degree is the largest among the ones of the other modules (i.e. it is the bottleneck module): i.e.

$$\forall i = 1,2,\ldots,N \; \frac{Tcalc_i}{n_i^*} \leq \frac{Tcalc_p}{n_p^*}$$

A set of parallelism degrees in which all the stages are balanced and $ParMod_p$ is choosing its ideal parallelism degree is the only case of an efficient Nash equilibrium. In fact we can observe that: (1) the strategy profile is an equilibrium, since the other controllers are certainly choosing a parallelism degree smaller than their ideal parallelism degree (due to the fact that $ParMod_p$ is the bottleneck); (2) if the p-th control part deviates from n_p^*, the local cost function J_p will certainly increase. Hence the unique efficient Nash equilibrium of the game is given by the following strategy profile:

$$\mathbf{s}^{(e)} = \left\{\tilde{n}_1,\ldots,\tilde{n}_{p-1},n_p^*,\tilde{n}_{p+1},\ldots,\tilde{n}_N\right\} \tag{8.7}$$

in which $ParMod_p$ is the bottleneck and all the other controllers select a parallelism degree $\tilde{n}_i$ (determined by expression (8.5)) such that their service time equals the service time of the bottleneck module. □

Example (Distributed control of a two-modules tandem graph)**.** To clarify the previous concepts we will describe a fist example in which we control a pipeline of two parallel modules $ParMod_1$ and $ParMod_2$. The pipeline graph is depicted in Figure 8.5. The two application modules communicate through a data stream: the first module periodically produces a sequence of tasks to the second one. Each module exploits a structured parallel computation and their control parts are involved in optimizing the performance level and the resource utilization cost induced by their execution. We suppose that the two ParMods are executed on two remote parallel architectures composed of 16 nodes each other, on which the resource cost depends on the number of used nodes during the execution.

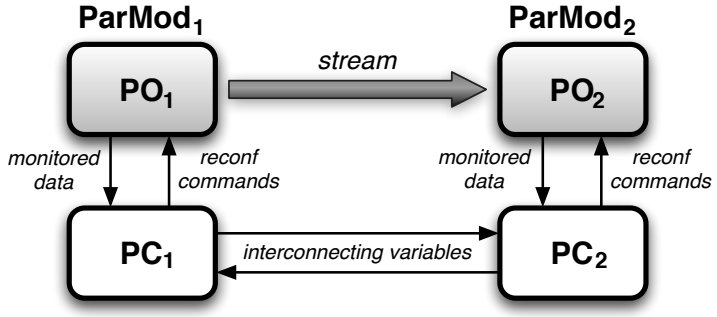


Figure 8.5: Example of distributed control: a tandem graph of two adaptive parallel modules.

Let us suppose that we are at the beginning of a control step k. We recall that in a distributed control scheme each ParMod shares the same control step concept with the other

modules of the application graph. We consider the following coefficients and parameters for the two control sub-problems:

- each control part performs a statistical prediction of it disturbance input. In this example the disturbance input is represented by the mean calculation time of a ParMod. Let us suppose that for the first ParMod the predicted calculation time is $1500t$ whereas for the second one is $2000t$, where t is a standardized time unit;
- the two modules give a different importance to their performance behavior achieved at steady-state: the coefficients α_1 and α_2 are 4 and 12 respectively;
- the two sub-problems consider different costs for the resource utilization: the first module considers a cost coefficient equal to 250 and the second one a lower cost equal to 140 (i.e. in the first execution platform the resource utilization is more expensive than in the second one).

So doing the two control sub-problems can be described as follows: the interconnecting variables between control parts are represented by the service times given by the following expressions:

$$T_{S_1}(k) = \frac{T_{calc-1}(k)}{n_1(k)}$$
$$T_{S_2}(k) = \frac{T_{calc-2}(k)}{n_2(k)}$$

The effective performance level is represented by the mean inter-departure time from the two parallel modules, that in a pipeline graph can be determined as the maximum between the two service times:

$$T_{p_1}(k+1) = \max\left\{\frac{T_{calc-1}(k)}{n_1(k)}, T_{S_2}(k)\right\}$$
$$T_{p_2}(k+1) = \max\left\{\frac{T_{calc-2}(k)}{n_2(k)}, T_{S_1}(k)\right\}$$

Therefore the two local cost functions are given by:

$$J_1(n_1,n_2) = \alpha_1 T_{p_1}(k+1) + \beta_1 n_1(k) = 4 \cdot \max\left\{\frac{T_{calc-1}(k)}{n_1(k)}, T_{S_2}(k)\right\} + 250 \cdot n_1(k)$$
$$J_2(n_1,n_2) = \alpha_2 T_{p_2}(k+1) + \beta_2 n_2(k) = 12 \cdot \max\left\{\frac{T_{calc-2}(k)}{n_2(k)}, T_{S_1}(k)\right\} + 140 \cdot n_2(k)$$

According to expression (8.4) we can now calculate the two ideal parallelism degrees that optimize the corresponding local cost function of each ParMod in isolation. We obtain:

$$n_1^*(k) = \sqrt{\frac{\alpha_1 T_{calc-1}(k)}{\beta_1}} = \sqrt{\frac{4 \cdot 1500t}{250}} \simeq 4.89$$
$$n_2^*(k) = \sqrt{\frac{\alpha_2 T_{calc-2}(k)}{\beta_2}} = \sqrt{\frac{12 \cdot 2000t}{140}} \simeq 13.09$$

As we know from Proposition 8.2.3 and 8.2.4 a strategy profile is the efficient Nash equilibrium of this example if and only if each ParMod selects a parallelism degree such that: (i) the two pipeline stages are balanced, i.e. they have the same service time; (ii) one ParMod selects its ideal parallelism degree while the other one chooses a number of nodes for the execution lower than its ideal parallelism degree. We can observe that in this example the unique efficient Nash equilibrium between the two control parts is given by the following pair of parallelism degrees:

$$\mathbf{s}^{(e)} = \left[n_1^*(k), \tilde{n}_2(k)\right]^T = \left[4.89, 6.53\right]^T$$

With these parallelism degrees the two pipeline stages are balanced (they have a service time equal to $307t$) and each parallelism degree is the best response to the actually received interconnecting variables. Moreover, since the first ParMod is choosing its ideal parallelism degree, there not exists a strategy profile that makes it possible to improve the value of the first cost function. Therefore this resource allocation is: (i) a Nash equilibrium; (ii) it is Pareto optimal. The total value of the two cost functions with this allocation is 2449 for the first cost function and 4590 for the second one for a total cost of $J_G = J_1 + J_2 = 2449 + 4590 = 7039$.

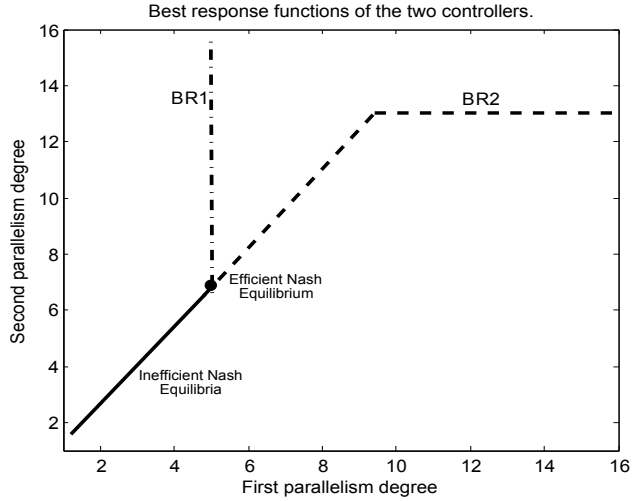


Figure 8.6: Best response functions of the two controllers.

For a game with two players it is useful to depict the best response functions and discuss the properties of Nash equilibria graphically. For our example the two best response functions, for the first and the second controller, are given by the following relations:

$$BR_1(n_1) = \text{argmin}_{n_1 \in \mathbf{U}_1} J_1(n_1, n_2) = \min\left\{\frac{T_{calc-1}}{T_{calc-2}} n_2, n_1^*\right\} = \min\left\{\frac{1500t}{2000t} n_2, 4.89\right\}$$

$$BR_2(n_2) = \text{argmin}_{n_2 \in \mathbf{U}_2} J_2(n_1, n_2) = \min\left\{\frac{T_{calc-2}}{T_{calc-1}} n_1, n_2^*\right\} = \min\left\{\frac{2000t}{1500t} n_1, 13.09\right\}$$

In Figure 8.6 is shown the behavior of the two best response functions. The Nash equilibria of the game are given by the intersections of the two functions. In fact at those points the response of each controller is the best response to the other controllers' strategies. As said before, in this game there are an infinite number of strategy profiles that are Nash equilibria (all the points belonging to the black solid line in the Figure). As stated in Proposition 8.2.3 they are all the pairs of parallelism degrees in which the two ParMods have the same service time and they are choosing a parallelism degree smaller than their ideal parallelism degree n_i^* determined in isolation. We can note that the intersection point in Figure 8.6 corresponds to the unique efficient Nash equilibrium.

8.2.2 Considerations about the integrality constraint

In the previous sections we have treat the strategy set of each controller as the real interval $[1, n_i^{max}]$. In practise the parallelism degree n_i of a ParMod always takes positive integer values for each control step of the execution. This continuous relaxation has been extremely important to develop a tractable analysis of our problem and to analyze the existence of Nash equilibria and their properties through classical theorems for continuous games.

In the previous example of a two-modules tandem graph we have exploited the continuous relaxation in order to identify the efficient Nash equilibrium. As a special case if this strategy profile $\mathbf{s}^{(e)} = [n_1^{(e)}, n_2^{(e)}, \ldots, n_N^{(e)}]^T$ is also an integer vector (i.e. $n_i^{(e)} \in \mathbb{N}\ \forall i = 1, \ldots, N$), then it will certainly be a Nash equilibrium of the original finite strategic game with integer strategies. Otherwise, at the end of the interaction protocol, control parts select an integer approximation of the real-valued strategy profile that has been found. We have 2^N possible approximations, in which each component:

$$n_i^{(e)} \text{ can be approximated by } \begin{cases} \lfloor n_i^{(e)} \rfloor \\ \lceil n_i^{(e)} \rceil \end{cases} \tag{8.8}$$

We call these integer vectors the *approximate Nash equilibria* of the game. Among this set of possible approximations (whose size is exponential in the number of graph modules), two of them are of special interest:

- ***performance-conservative approximation***: it consists in the integer strategy profile $\mathbf{s}^{(e)} = [\lceil n_1^{(e)} \rceil, \lceil n_2^{(e)} \rceil, \ldots, \lceil n_N^{(e)} \rceil]^T$. In other words, after the identification of the efficient Nash equilibrium, each controller rounds up its real-valued parallelism degree $n_i^{(e)}$. With this integer strategy profile the effective performance achieved by each ParMod is not worst w.r.t the other approximate Nash equilibria, at the cost of a higher resource utilization cost;

- ***resource-conservative approximation***: it consists in the integer strategy profile $\mathbf{s}^{(e)} = [\lfloor n_1^{(e)} \rfloor, \lfloor n_2^{(e)} \rfloor, \ldots, \lfloor n_N^{(e)} \rfloor]^T$ in which each controller rounds down its continuous parallelism degree $n_i^{(e)}$. This approximate Nash equilibrium, though the worst

from the performance point of view (each ParMod has a lower service time), has the best resource utilization cost among the other possible approximations.

Therefore in the rest of this chapter we will assume that the continuous formulation of the problem is used for performing the interaction protocols and orchestrating the best response dynamics between control parts. After the conclusion of the interaction protocol we will apply a proper rounding on the resulting strategy profile. Especially for parallelism degrees taking large values (which is a reasonable assumption with the actual trend of parallel computing technologies) this continuous relaxation, though suboptimal, produces acceptable approximations with the advantage of being based on well-founded theoretical results.

8.2.3 Approaching more general acyclic graph structures

So far we have studied the distributed control of computation graphs having a pipeline structure: i.e. a linear sequence of adaptive parallel modules that cooperate transmitting and receiving sequences of tasks. As we have seen several distributed applications can be described in this way but, in general, the structure of a distributed computation can be extremely varying. In this section we will extend our analysis in order to cover applications represented by more general acyclic graph structures.

Queueing networks performance analysis states that the steady-state behavior of a computation module depends on its internal configuration, e.g. its actual degree of parallelism, but it is also influenced by the behavior of the other modules of the graph. This second aspect is extremely critical. The steady-state behavior of a graph can be studied exploiting the results of Chapter 4. So far we have discussed the distributed control of pipeline graph structures. In this special case the mean inter-departure time from each module can be calculated in a simple way by taking the maximum among the service times of all the stages of the network. Another notable example is represented by ***functional-partitioning graphs*** (as the one shown in Figure 8.7).

A functional-partitioning scheme consists in a computation module IN that generates a stream of input tasks that are alternatively transmitted to one of the modules $M_1, \ldots, M_N$ according to a certain probability distribution (e.g. a class-based routing). P_i denotes the probability that a request generated by IN is transmitted to M_i, where $\sum_{i=1}^{N} p_i = 1$. The results are received non-deterministically by a module OUT, that executes a final post-processing computation before transmitting the results out of the network (e.g. to other destination modules).

The previous distributed computation can have different applications in practise. The set of parallel modules $M_1, \ldots, M_N$ can be used in different ways:

- they can be "general-purpose" modules able to process all the tasks generated by IN. In this case we have a "macro task-farm" scheme in which the elaboration capability is replicated onto a specific set of distributed instances. In this case the objective of the IN module is to balance the workload to each ParMod M_i, in order

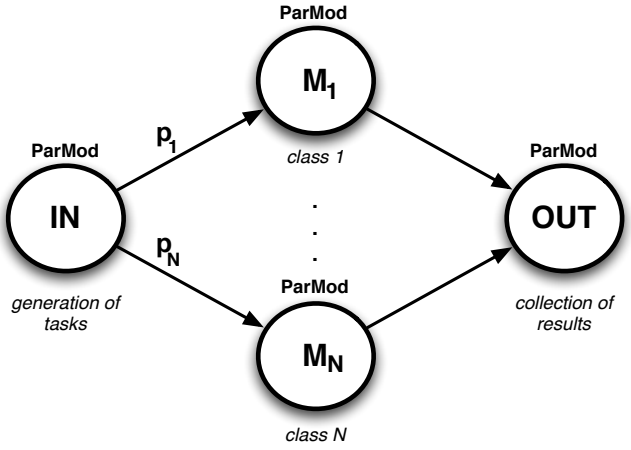


Figure 8.7: Parallel modules organized in a functional-partitioning scheme.

to exploit their capabilities at best. For instance with the on-demand scheduling strategy we obtain a scheduling probability with a uniform distribution (i.e. $\forall i = 1,2,\ldots,N\ p_i = 1/N$);

- parallel modules $M_1,\ldots,M_N$ can be specialized for performing the parallel computation only for a specific class of tasks. The *IN* module is responsible for scheduling each task to the most appropriate module M_i according to its class. In this case the transmission frequencies depend on the probabilities $p_1, \ldots, p_N$.

As for pipeline structures, for this type of acyclic graphs we are able to provide a simple closed-form expression for the steady-state inter-departure time from each module. In a functional-partitioning scheme we have three possible situations: (i) the bottleneck is the module *IN* or (ii) OUT; (iii) the bottleneck is one of the module M_i. In fact we know from Proposition 4.1.4 that at most one module M_i can be the bottleneck of the graph. Let us analytically describe the mean inter-departure times from the different modules:

OUT **module:** for calculating the steady-state behavior of the OUT module we consider three cases: (1) OUT is the bottleneck, in this case its effective behavior coincides with its service time $T_{S_{OUT}}$. (2) *IN* is the bottleneck of the graph, i.e. the inter-arrival times to all the M_i modules are greater than the corresponding service times. This means that the effective behavior of OUT is equal to its inter-arrival time that can be calculated in function of the service time of *IN* i.e. $T_{S_{IN}}$:

$$T_{A-OUT} = \left(\sum_{i=1}^{N} \frac{p_i}{T_{S_{IN}}}\right)^{-1} = \left(\frac{1}{T_{S_{IN}}} \sum_{i=1}^{N} p_i\right)^{-1} = T_{S_{IN}}$$

Therefore the inter-departure time of OUT is equal to the service time of *IN*. The last case is when M_i is the bottleneck. Also in this case the effective behavior of

OUT is given by its inter-arrival time, that can be determined in function of the service time of the bottleneck module (M_i) i.e. T_{S_i}:

$$T_{A-OUT} = \left(\sum_{j=1}^{N} \frac{p_j}{T_{S_i} p_i} \right)^{-1} = \left(\frac{1}{T_{S_i} p_i} \sum_{j=1}^{N} p_j \right)^{-1} = T_{S_i} p_i$$

In summary the inter-departure time from OUT can be expressed taking the maximum between the three previous conditions:

$$T_{p\_OUT}(k+1) = \max\left\{ T_{S_{OUT}}(k), T_{S_{IN}}(k), T_{S_i}(k) \cdot p_i \ \forall i = 1,2,\ldots,N \right\} \tag{8.9}$$

IN **module:** as we have demonstrated in Proposition 4.1.11, at steady-state the performance behavior of IN and OUT coincides, since they are the unique source and sink respectively. Therefore the mean inter-departure time from the source IN is given by:

$$T_{p\_IN}(k+1) = \max\left\{ T_{S_{IN}}(k), T_{S_{OUT}}(k), T_{S_i}(k) \cdot p_i \ \forall i = 1,2,\ldots,N \right\} \tag{8.10}$$

If IN is the bottleneck, its inter-departure time is equal to its service time. If OUT is the bottleneck the inter-departure time from IN coincides with the service time of OUT. Finally, if a generic M_i is the bottleneck, the inter-departure time from IN is equal to the service time of M_i multiplied by the probability p_i.

Generic M_i module: let us consider the inter-departure time from a generic module M_i. If the bottleneck of the graph is the module M_i itself, its inter-departure time equals its service time T_{S_i}. If the bottleneck of the graph is IN or OUT, in both the cases the inter-departure time from M_i is equal to the bottleneck service time divided by the probability p_i. The last case is when a module M_j with $j \neq i$ is the bottleneck. In this situation the inter-departure time from IN is equal to the service time of M_j multiplied by the probability p_j. Hence the inter-departure time from M_i can be calculated by dividing this value by the probability p_i:

$$T_{p_i}(k+1) = \max\left\{ T_{S_i}(k), \frac{T_{S_{IN}}(k)}{p_i}, \frac{T_{S_{OUT}}(k)}{p_i}, \frac{T_{S_j}(k)\, p_j}{p_i} \ \forall j = 1,\ldots,N \wedge j \neq i \right\} \tag{8.11}$$

As we can note the result of this analysis makes it possible to model the inter-departure time from any module of a functional-partitioning graph in a similar way to pipeline graphs. We have the pointwise maximum of $N+2$ terms each one involving the service time of exactly one module of the graph.

This result can be extended to a general class of acyclic graphs that we have already introduced in Chapter 4: i.e. *acyclic computation graphs with a unique source module*. The next proposition is extremely important for exploiting the distributed control of such class of graphs, since it provides an alternative performance modeling method w.r.t the algorithmic procedure described in Chapter 4.

Proposition 8.2.6 (Closed-form performance modeling of single-source acyclic graphs)**.** *Given a single-source acyclic graph G of N nodes, the mean inter-departure time T_{p_i} from each node i of the graph can be represented by the following closed-form expression:*

$$T_{p_i} = \max\Big\{ f_{i,1}(n_1),\, f_{i,2}(n_2), \ldots, f_{i,N}(n_N) \Big\} \tag{8.12}$$

The inter-departure time from node i can be calculated as the pointwise maximum between the values of a set of N functions $f_{i,j}$ with $j = 1,2,\ldots,N$, each one addressing the case in which the node j of the graph is the bottleneck module. Function $f_{i,j}$ can be defined as follows:

$$f_{i,j}(n_j) = T_{S_j}(k) \frac{\sum\limits_{\forall \pi \in \mathscr{P}(S \to j)} \left(\prod\limits_{\forall e \in \pi} e.p \right)}{\sum\limits_{\forall \pi \in \mathscr{P}(S \to i)} \left(\prod\limits_{\forall e \in \pi} e.p \right)} = \frac{T_{calc-j}(k)}{n_j(k)} \frac{\sum\limits_{\forall \pi \in \mathscr{P}(S \to j)} \left(\prod\limits_{\forall e \in \pi} e.p \right)}{\sum\limits_{\forall \pi \in \mathscr{P}(S \to i)} \left(\prod\limits_{\forall e \in \pi} e.p \right)} \tag{8.13}$$

where S is the unique source node of the graph G and $\mathscr{P}(S \to i)$ is the set of all the paths in the graph from node S to i.

Proof. The proposition is based on the results described in Chapter 4. The mean inter-departure time T_{p_i} from a generic node i can be calculated by addressing the possible alternative cases of bottleneck identification. The first case is when the node i itself is

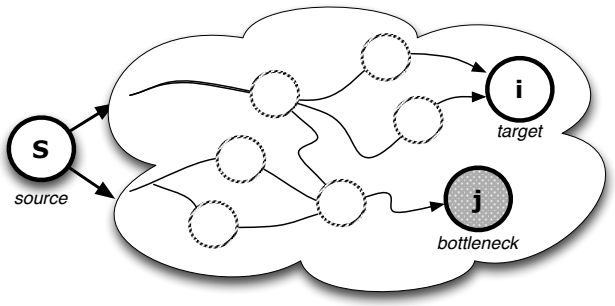


Figure 8.8: Single-source acyclic computation graph: S is the unique source, j is the bottleneck node and i is the target node for which we want to determine the mean inter-departure time T_{p_i}.

the bottleneck. In this case we know that its mean inter-departure time coincides with its service time, i.e. $T_{p_i} = T_{S_i}$ and the utilization factor of the module at steady-state is equal to 1. This result is correctly established by expression (8.13). In fact the second term of this expression is equal to 1 since $i = j$ and thus function $f_{i,i}$ coincides with the service time expression $T_{S_i}(k) = T_{calc-i}(k)/n_i(k)$.

In the general case the steady-state behavior of node i depends on the performance behavior of another node j of the graph which is currently the bottleneck (as in Figure 8.8).

In order to prove the correctness of expression (8.13) we can observe that, if j is the bottleneck, its inter-arrival time at steady-state is equal to its service time. This can by obtained by correcting the inter-departure time from the source S in the following way:

$$T_{p_s} = T_{S_j}(k) \cdot \sum_{\substack{\forall \pi \in \\ \mathscr{P}(S \to j)}} \left(\prod_{\forall e \in \pi} e.p \right)$$

If j is the bottleneck, the mean inter-departure time from the source is equal to the service time of j multiplied by the total probability of all the paths in the graph from the source S to the bottleneck. At this point the mean inter-arrival time from a generic node i can be calculated by dividing the previous expression by the total probability of all the paths in the graph from the source S to the node i obtaining the expression (8.13). Since we do not know a priori which is the bottleneck of the graph, we calculate the the mean inter-departure time from the generic node i as the maximum value assumed by functions $f_{i,j}$ for $j = 1, \ldots, N$. □

Example. Let us consider the graph of Figure 8.9. As an example the mean inter-departure time from module 4 (denoted as"target" in the figure) can be determined by reasoning on the graph structure:

$$T_{p_4} = \max \left\{ T_{S_4}, \frac{T_{S_1}}{p\,p'}, \frac{T_{S_2}}{p}, T_{S_3} \frac{(1-p)}{p\,p'}, T_{S_5} \frac{(1-p')}{p'}, \frac{T_{S_6}}{p'}, T_{S_7} \frac{(1-p)}{p\,p'}, \frac{T_{S_8}}{p\,p'} \right\}$$

where $T_{S_i} = T_{calc-i}/n_i$ is the service time of the i-th ParMod of the graph. In a similar way we can proceed to determine the inter-departure times from all the other modules.

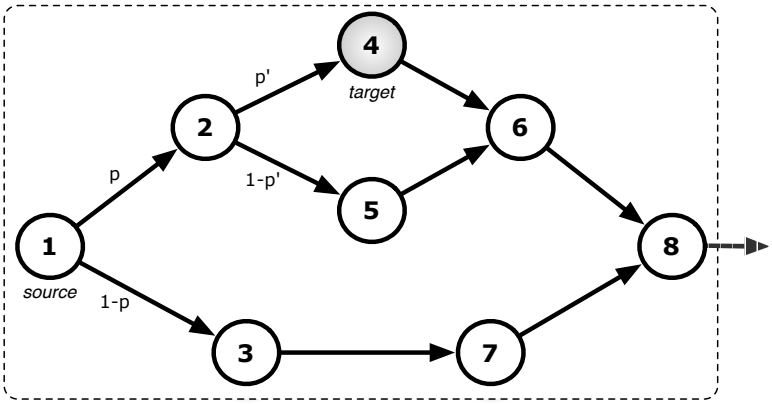


Figure 8.9: Example of closed-form performance modeling of a single-source acyclic graph.

At this point our aim is to study if it is possible to apply to single-source acyclic graphs the results of the self-interest interaction that we have described before for pipeline graphs. In this case performance coupling relationships are a generalization of the pipeline case:

instead of having the maximum between the service times of the modules, we have the maximum of a set of terms involving the service times properly multiplied by positive multiplicative factors that depend on the routing probabilities. This means that the previous description of Section 8.2 is still valid, since, given a generic module $ParMod_i$:

- we can calculate its ideal parallelism degree in isolation in the same manner as for a module in a pipeline graph;
- the existence of Nash equilibria can be proved by applying Theorem 7.3.2 as we have done with Proposition 8.2.2;
- the best response to the set of received interconnecting variables (i.e. the current service times advertised by the other modules) is still: (i) the ideal parallelism degree $n_i^*(k)$ of the module; (ii) or a parallelism degree $\tilde{n}_i(k)$ smaller than the ideal one, given by:

$$\tilde{n}_i(k) = \frac{T_{calc-i}(k)}{f_i^{max}(k)} \tag{8.14}$$

where $f_i^{max}(k) = \max_{j \neq i} f_{i,j}\left(n_j(k)\right)$ denotes the best performance (smaller inter-departure time) achievable by $ParMod_i$ due to the control decisions actually taken by the other controllers at control step k. It is worth noting that expression (8.14) is a generalization of (8.5) for single-source acyclic graphs.

Let us exemplify this concept with a numerical example.

Example (Example of distributed control of a functional-partitioning structure)**.** Let us consider four parallel modules: IN, OUT, M_1 and M_2 organized in a functional-partitioning interaction pattern. The probability of transmission to M_1 is $p_1 = 0.6$ and to M_2 is $p_2 = 0.4$. The local objective functions of each ParMod have the same structure of the ones described in Section 8.1.1. In Table 8.1 are listed the cost function parameters and

Parameters	**IN**	$\mathbf{M_1}$	$\mathbf{M_2}$	**OUT**
T_{calc}	$1500t$	$2000t$	$3500t$	$2750t$
α	4	12	6	3
β	250	140	90	110
n^*	4.9	13.09	15.27	8.66

Table 8.1: Parameters of the functional-partitioning example.

the ideal parallelism degree of each module evaluated in isolation. With these parallelism degrees the bottleneck module is OUT, with a real parallelism degree 8.66 that corresponds to a theoretical service time of $318t$. The unique efficient Nash equilibrium is given by the following vector of real-valued parallelism degrees:

$$\mathbf{s}^{(e)} = \left[\tilde{n}_{in}(k), \tilde{n}_1(k), \tilde{n}_2(k), n^*_{out}(k)\right]^T = \left[4.72, 3.78, 4.40, 8.66\right]^T$$

that corresponds to the following service times for each node of the graph:

$$T_{S_{in}} = 318t \;\; T_{S_1} = 529t \;\; T_{S_2} = 799t \;\; T_{S_{out}} = 318t$$

With this set of parallelism degrees the sum of the local cost functions is 16403. Concretely, since real parallelism degrees are integer values, we exploit a performance-conservative rounding (i.e. to the upper integer). The final strategy profile is given by the vector $[5,4,5,9]^T$.

In the previous example we can note a general property of all the equilibrium solutions: *with the parallelism degrees of a Nash equilibrium all the modules of the graph have a service time (in isolation) equal to their steady-state inter-arrival time.* ***In other words at equilibrium the utilization factor of each module is equal to*** 1. In fact, if a ParMod selects a parallelism degree such that its service time results lower than its inter-arrival time, this choice can not be the best response to the control actions of the other controllers, because the module is unnecessary fast. This condition, though its simplicity, is extremely important for identifying equilibrium strategy profiles for acyclic graph structures. The following general proposition can be stated:

Proposition 8.2.7. *In the distributed control of single-source acyclic graphs, let us consider a set of parallelism degrees* $\mathbf{s}^{(e)} = [n_1^{(e)}, n_2^{(e)}, \ldots, n_N^{(e)}]^T$ *such that:*

$$\begin{cases} \dfrac{T_{calc-i}}{n_i^{(e)}} = T_{p_i} \; \forall i = 1,2,\ldots,N \\ \\ n_i^{(e)} \leq n_i^* \; \forall i = 1,2,\ldots,N \end{cases}$$

where T_{p_i} *is the steady-state inter-departure time of the i-th module obtained with expression (8.12). This strategy profile is a (pure) Nash equilibrium of the game.*

We can note that for pipeline graphs this proposition coincides with Proposition 8.2.3. A set of parallelism degrees that satisfies these conditions is a Nash equilibrium since, due to the formulation of local cost functions (8.1): having a service time lower than the inter-arrival time is completely useless from the performance viewpoint and it is also a waste of resources, while higher service times (i.e. lower parallelism degrees) are not cost effective since parallelism degrees at equilibrium can not be higher than the corresponding ideal parallelism degrees of the modules in isolation. In a similar way to Proposition 8.2.4 we can also provide the conditions for identifying the unique Pareto optimal Nash equilibrium.

Proposition 8.2.8. *In the distributed control of single-source acyclic graphs, an equilibrium strategy profile* $\mathbf{s}^{(e)} = [n_1^{(e)}, n_2^{(e)}, \ldots, n_N^{(e)}]^T$ *such that the conditions of Proposition 8.2.7 are satisfied is the unique Pareto efficient Nash equilibrium of the game iff there*

exists at least one control part that it choosing a parallelism degree equal to its ideal parallelism degree, i.e:

$$\exists i = 1, 2, \ldots, N \text{ such that } n_i^{(e)} = n_i^*$$

The proof is similar to the one of Proposition 8.2.4. No other strategy profile can dominate a strategy profile that satisfies the previous conditions, because the local cost of the module that is choosing its ideal parallelism degree will increase if that control part performs any change that deviates from its ideal parallelism degree.

8.2.4 Interaction protocols for reaching the efficient Nash equilibrium

In this section we will discuss how control parts interact in order to reach an agreement in their control decisions. In the communication-based distributed MPC strategy the control logics of different adaptive parallel modules interact through a non-cooperative game framework, in which the notion of agreement corresponds to the concept of Nash equilibrium.

In Section 7.3.3 we have introduced a general interaction protocol based on an iterative information exchange between controllers: i.e. control parts exchange trajectories of interconnecting variables that describe the coupling relationships between control subproblems. In the next sections we will describe specific formulations of this interaction protocol for the control problem presented in this chapter.

8.2.4.1 Interaction protocol based on a fully-interconnected scheme

In this section we describe a first interaction protocol for single-source acyclic computation graphs. This protocol is characterized by the following points:

- each controller transmits the current value of its service time variable directly to all the other controllers and receives the service times from the others (controllers are interconnected in a complete fashion, i.e. Assumption 8.2.1);
- every controller knows the complete structure of the graph and also all the routing probabilities, i.e. *complete knowledge* assumption.

In this case we are able to reach the efficient Nash equilibrium in only one information exchange between controllers: i.e. control parts simultaneously exchange interconnecting variables only once per control step. The pseudocode of this protocol is provided below:

As a first step each PC_i applies a statistical estimation (e.g. based on the past history) of its mean calculation time per task expected for the next control step k (row 2). Next,

Algorithm 4: Selfish interaction protocol for single-source acyclic graphs with a complete network between Control Parts.

1 **foreach** *control step k each* PC_i **do**
2 $\quad T_{calc-i}(k) = \text{Predictive\_Filter}(...)$;
3 $\quad$ **foreach** $j = 1,\ldots,N \wedge j \neq i$ **do**
4 $\quad\quad T_{S_j}(k) = 0$;
5 $\quad n_i^*(k) = \text{optimize}(J_i, \{T_{S_j}(k), \forall j \neq i\})$;
6 $\quad T_{S_i}(k) = T_{calc-i}(k)/n_i^*(k)$;
7 $\quad \text{send\_to\_all}(T_{S_i}(k))$;
8 $\quad \text{receive\_from\_all}(\{T_{S_j}(k), \forall j \neq i\})$;
9 $\quad n_i(k) = \text{optimize}(J_i, \{T_{S_j}(k), \forall j \neq i\})$;
10 $\quad$ use a rounding of $n_i(k)$ as the new parallelism degree for step k;

at the beginning of the interaction protocol, each control part assumes that all the input interconnecting variables are zero-initialized (row 4). At this point PC_i optimizes its local cost function J_i considering its performance behavior in isolation (interconnecting variables are zero) and calculates its ideal parallelism degree n_i^* (row 5). After that the control parts exchange (rows 7 and 8) their mean service times with the other controllers of the graph. Finally, after this information exchange, PC_i is able to calculate the parallelism degree that it will use for the k-th control step of the execution (row 9), i.e. applying a proper integer rounding of the final parallelism degree (row 10).

The interaction protocol exploits a complete interconnection network between controllers for reaching the final result in a single information exchange. Before exchanging the set of interconnecting variables, each controller calculates its ideal parallelism degree. In other words the initial strategy profile consists in the following vector of control inputs, in which the performance coupling relationships have been completely neglected.

$$\mathbf{s}^{(init)} = \left[n_1^*(k), n_2^*(k), \ldots, n_N^*(k)\right]^T \tag{8.15}$$

Then each control part PC_i transmits to all the other controllers its service time $T_{S_i}(k)$ and receives the input interconnecting variables $\{T_{S_1}(k),\ldots,T_{S_{i-1}}(k),T_{S_{i+1}}(k),\ldots,T_{S_N}(k)\}$ from the other control parts. After this information exchange each control part has a complete view of the ideal performance behavior of all the ParMods. For single-source acyclic graphs we are able to exploit a closed-form expression of the mean inter-departure time from each module of the graph. Therefore each controller PC_i can calculate the parallelism degree (best response) that optimizes its local cost function with the actually received interconnected variables. Given $f_i^{max} = \max_{j\neq i} f_{i,j}(n_j)$, that denotes the minimum inter-departure time that $ParMod_i$ can assume due to the performance behavior of the other modules, we have two possible situations:

- ***the new optimal parallelism degree is still the ideal parallelism degree*** $\mathbf{n_i^*}$. In this situation the ParMod is the bottleneck of the graph and we are in the second case

described with Figure 8.4: the best response to the received interconnecting variables remains the ideal parallelism degree. Greater or smaller parallelism degrees will produce a higher value of the local cost function J_i compared with the value achieved with the ideal parallelism degree;

- ***the new optimal parallelism degree is lower than the ideal one***, and it is the parallelism degree such that the service time of $ParMod_i$ coincides with the minimum inter-departure time f_i^{max} induced by the service times advertised by the other parallel modules (we are in the first situation described with Figure 8.3). This parallelism degree is denoted by $\tilde{n}_i$ and it is calculated by expression (8.14). Higher parallelism degrees for $ParMod_i$ would be completely useless, because they do not produce a further performance improvement in the steady-state behavior of the computation at the price of a higher resource utilization cost.

The final result of the interaction protocol is the unique efficient Nash equilibrium of the game, given by:

$$\mathbf{s}^{(e)} = \left\{\tilde{n}_1(k), \ldots, \tilde{n}_{p-1}(k), n_p^*(k), \tilde{n}_{p+1}(k), \ldots, \tilde{n}_N(k)\right\}$$

where $ParMod_p$ is the module which represents the bottleneck of the graph.

The behavior of this interaction protocol can be evaluated from different points of view. In terms of number of iterations, this protocol concludes its execution in a single information exchange. In terms of number of exchanged messages, each controller exchanges the value of its service time with all the other control parts. This means that each controller transmits $N-1$ messages where N is the total number of nodes of the graph. Totally we have a number of messages which is quadratic in the number of ParMods: $N(N-1) = N^2 - N \simeq O(N^2)$. The total execution time of the protocol can be expressed as follows:

$$T_{Control} = 2\,T_{opt} + (N-1)\,T_{snd\_var} + (N-1)\,T_{rcv\_var} \tag{8.16}$$

where T_{opt} is the time for optimizing the local objective function and calculate the actual best response, T_{snd_var} is the time spent in transmitting one output interconnecting variable to a destination and T_{rcv_var} is the time for receiving an input interconnecting variable from another controller.

8.2.4.2 Interaction protocol based on a partially-interconnected scheme

In this section we will introduce an alternative and interesting protocol characterized by the following assumptions:

- each control part is directly interconnected only with a limited set of *neighbor controllers*. This means that each controller transmits and receives service time variables only with its neighbors;

- each controller has a partial view of the computation graph. It only knows its neighbor nodes and the strictly necessary routing probabilities, i.e. *partial knowledge* assumption.

About the first point we will assume that control parts of different ParMods are interconnected according to the following property:

Assumption 8.2.9. *We will assume a partially-interconnected structure between control parts: i.e. control part PC_i is directly interconnected with control part PC_j and vice versa iff, at the application level, there exists a data stream between the two corresponding operating parts (parallel computations).*

An example of partial interconnection for a pipeline graph is shown in Figure 8.10. We denote with $Nh(i)$ the set of neighbor controllers of the i-th control part (PC_i). In a pipeline graph this set is defined by:

$$Nh(i) = \begin{cases} \{PC_{i+1}\} \text{ if } i = 1 \\ \{PC_{i-1}\} \text{ if } i = N \\ \{PC_{i-1}, PC_{i+1}\} \text{ if } 2 \leq i \leq N-1 \end{cases} \tag{8.17}$$

In this protocol the mean inter-departure time model needs to be modified accounting

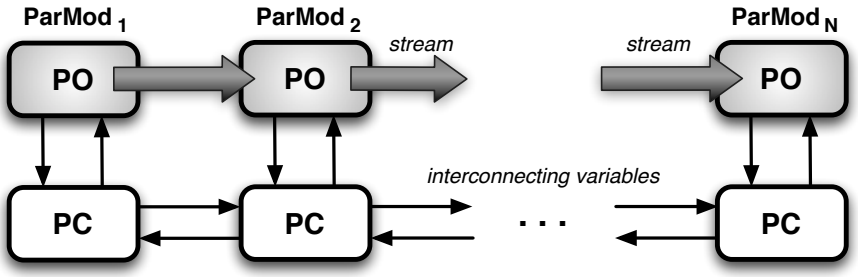


Figure 8.10: Pipeline distributed control with a partial interconnection network between control parts.

only for the service time variables received by the neighbors:

$$T_{p_i}(k+1) = \max_{\forall\, j \in Nh(i)} \left\{ \frac{T_{calc-i}(k)}{n_i(k)}, f_{i,j}(n_j) \right\} \tag{8.18}$$

We recall that each term $f_{i,j}(n_j)$ involves the service time of the j-th neighbor multiplied/divided by proper routing probabilities. We will exemplify this concept with the next example.

Example. Let us consider the graph of Figure 8.9. In the protocol that we are describing in this section the mean inter-departure time from a module is expressed only in function

of the set of service time variables received from its neighbors. As an example for module 4 we have:

$$T_{p_4} = \max\left\{T_{S_4}, \frac{T_{S_2}}{p'}, \frac{T_{S_6}}{p'}\right\}$$

since module 4 is directly interconnected only with module 2 and 6.

As we know from the steady-state performance of a module depends on the performance behavior of *all* the other modules that reside in the same computation graph structure. This means that, though interconnecting variables are exchanged only between neighbor controllers, their effects need to propagate inside the network in such a way that each controller accounts for the control decisions taken by all the other controllers (also the ones not directly interconnected with it). For this reason the interaction protocol will be modified w.r.t the fully-interconnected case. The main difference is that the efficient Nash equilibrium strategy profile will be reached after a fixed number of iterations (and not only once as in the previous case). The pseudocode is presented in Algorithm 5.

Algorithm 5: Selfish interaction protocol for single-source acyclic graphs with a partial interconnection network between Control Parts.

1 **foreach** *control step k each* PC_i **do**
2 $\quad T_{calc-i}(k) = \text{Predictive\_Filter}(...)$;
3 $\quad$ **foreach** $j \in Nh(i)$ **do**
4 $\quad\quad T_{S_j}^{(0)}(k) = 0$;
5 $\quad n_i^{(0)}(k) = \text{optimize}(J_i, \{T_{S_j}^{(0)}(k), \forall j \in Nh(i)\})$;
6 $\quad T_{S_i}^{(0)}(k) = T_{calc-i}(k)/n_i^{(0)}(k)$;
7 $\quad$ **for** q=*1* **to** D **do**
8 $\quad\quad \text{send\_to\_neighbors}(T_{S_i}^{(q-1)}(k))$;
9 $\quad\quad \text{receive\_from\_neighbors}(\{T_{S_j}^{(q)}(k), \forall j \in Nh(i)\})$;
10 $\quad\quad n_i^{(q)}(k) = \text{optimize}(J_i, \{T_{S_j}^{(q)}(k), \forall j \in Nh(i)\})$;
11 $\quad\quad T_{S_i}^{(q)}(k) = T_{calc-i}(k)/n_i^{(q)}(k)$;
12 $\quad$ use a rounding of $n_i^{(q)}(k)$ as the new parallelism degree for step k;

The superscript on the service time and the parallelism degree variables indicates the current iteration of the protocol denoted by q. The algorithm proceeds in a similar way to the fully-interconnected case. At each control step the i-th controller starts by estimating its future mean calculation time and initializes all the input interconnecting variables from its neighbors to zero (rows 2 and 4). Then it calculates its ideal parallelism degree $n_i^{(0)}(k)$ and the corresponding service time (rows 5 and 6). At each iteration q, the controller transmits (row 8) the current value of its service time and receives the interconnecting

variables from the neighbor controllers (row 9). Then it optimizes its local cost function J_i based on the current values of the received interconnecting variables. The algorithm proceeds for a fixed number of iterations that depend on the graph structure.

In our methodology a directed edge between two modules corresponds to a data stream between the parallel computations performed by the corresponding operating parts. Based on Assumption 8.2.9 two control parts are interconnected (they exchange information in both the directions) iff there exists a data stream between the two corresponding operating parts. Therefore the control part graph can be obtained by the computation graph by replacing each directed edge (stream) by a corresponding undirected edge (bidirectional link between control parts). At this point we can introduce the following concept:

Definition 8.2.2 (Diameter of a network)**.** Given an undirected acyclic graph, the diameter D is the maximum of the lengths of the shortest paths between all pairs of nodes.

In order to correctly execute the interaction protocol, the effects of interconnecting variables from each module must propagate in the graph reaching all the other modules. Therefore the protocol will be executed for a number of iterations equal to the diameter of the undirected graph of control parts.

Proposition 8.2.10. *Given a single-source acyclic graph the interaction protocol described with Algorithm 5 ends reaching the efficient Nash equilibrium of the game.*

Proof. The proof is straightforward given the problem formulation. We can observe that at a generic iteration q of the protocol, the service time variable $T_{S_i}^{(q)}(k)$ is higher or at least equal to the same variable calculated at the previous iteration of the protocol: i.e. $T_{S_i}^{(q-1)}(k)$. This means that the following sequence of inequalities is verified:

$$T_{S_i}^{(0)}(k) \leq T_{S_i}^{(1)}(k) \leq \ldots \leq T_{S_i}^{(D)}(k)$$

Therefore the sequence of parallelism degrees selected by PC_i is monotonically decreasing starting from the ideal parallelism degree $n_i^*(k)$ determined in isolation, i.e.:

$$n_i^*(k) = n_i^{(0)}(k) \geq n_i^{(1)}(k) \geq \ldots \geq n_i^{(D)}(k)$$

A monotonically decreasing sequence converges if and only if it is bounded below. This is our case since the parallelism degree can not be smaller than 1. Since this monotonically behavior is true for each control part, the algorithm will converge to a fixed-point strategy profile.

Let us consider $ParMod_p$ the module whose best response is always its ideal parallelism degree (i.e. the bottleneck module of the graph). Since for each control part the sequence of control decisions starts with the ideal parallelism degree and it is monotonically decreasing, $ParMod_p$ will choose n_p^* at each iteration. In fact the best response of the bottleneck ParMod to the service times advertised by the other controllers will always be its ideal parallelism degree. The other $ParMods_i$, with $i \neq p$, are characterized by

a monotonically decreasing sequence of parallelism degrees, that starts from their ideal parallelism degree n_i^* and ends up with a smaller parallelism degree $\tilde{n}_i$ that allows the module to have a service time equal to the minimum inter-departure time f_i^{max} induced by the presence of the bottleneck module $ParMod_p$. Therefore the final strategy profile reached at the end of the protocol is the vector of parallelism degree (8.7), i.e. the unique efficient Nash equilibrium. □

As for the previous case, this formulation of the interaction protocol can be evaluated by studying the number of exchanged messages and the completion time in function of the diameter of the graph. For the number of exchanged messages we can observe that each controller transmits the interconnection variable describing its actual service time to its neighbors and receives the output interconnecting variables from them at each iteration of the protocol. Therefore the completion time can be approximated as follows:

$$T_{Control} \simeq D\,T_{iter} = D\left(T_{opt} + |Nh(i)| \cdot T_{snd\_var} + |Nh(i)| \cdot T_{rcv\_var}\right) \tag{8.19}$$

while the number of exchanged messages is $D \cdot \sum_{i=1}^{N} |Nh(i)|$, since at each iteration every controller communicates only with its neighbors.

8.3 A cooperative interaction among Control Parts

In Chapter 7 we have discussed the potential limitations of a self-interest interaction between controllers. They can be summarized in the following points:

- depending on the problem formulation (e.g. in terms of cost function definitions), all the Nash equilibria of a game may be Pareto inefficient. As we have seen this is not the case of the example described in this chapter, in which a unique efficient Nash equilibrium always exists;
- the social optimum, i.e. the strategy profile that optimizes the (potentially weighted) sum of the local cost functions of each ParMod, though always Pareto optimal, may not be an efficient Nash equilibrium and thus it can not be reached by any of the interaction protocols described in the previous sections.

The last point is especially critical. At this regard let us analyze the following example of a functional-partitioning graph.

Example (Social optimum and Efficient Nash equilibrium profile)**.** Let us consider the example of distributed control of the functional-partitioning graph introduced in the previous section. As we have seen the efficient Nash equilibrium consists in the vector of real-valued parallelism degrees $\mathbf{s}^{(e)} = [4.72, 3.78, 4.40, 8.66]^T$ which corresponds to a total cost of the sum of local objectives equal to 16403. Although this strategy profile is the

best Nash equilibrium, in the sense that it is the only Pareto optimal equilibrium of our game, it is not the social optimum. The social optimum is given by:

$$\mathbf{s}^{soc} = [8.38, 6.70, 7.91, 15.37]^T$$

which gives a total cost of 12925, lower than the one achieved with the efficient Nash equilibrium. We can compute the price of stability which measures how the efficiency of the distributed control degrades due to selfish behavior of the controllers:

$$\frac{J_G(\mathbf{s}^{soc})}{J_G(\mathbf{s}^{(e)})} = \frac{12925}{16403} = 0.79$$

We can observe that with the social optimum the modules IN and OUT have a unilateral incentive to decrease their parallelism degrees, because in this way they can made their local cost functions better off. On the other hand this selfish action has a negative impact on the local costs of the other controllers that results in a higher total cost w.r.t the social optimum profile. This behavior explains why the social optimum profile may not be reached by exploiting a selfish interaction among control parts.

Even if the selfish interaction can be useful in some applicative contexts, due to its simplicity for reaching the best Nash equilibrium with a relatively small information exchange between controllers, in many cases we are interested in optimizing the entire application behavior expressed as the sum of the local objectives of each ParMod. This means that control parts have to exploit a better cooperation instead of a pure information exchange driven by their individual self-interest. At this point one question arises: how can we define a distributed MPC strategy able to reach the social optimum profile?

As discussed in Section 7.3.4 there are several ways to define a more cooperative interaction among controllers. For the example that we have introduced so far we have specific constraints that drive the selection of a proper cooperation technique:

- the local cost function and the performance model of a parallel module is known only to its controller. For this reason control parts exchange information even several times inside each control step of the execution;
- control parts can be completely interconnected or the interconnections can exist only between neighbor controllers;
- performance modeling of parallel computations induces the presence of *non-smooth local cost functions*. For single-source acyclic graphs we are able to express the mean inter-departure time from a module as the pointwise maximum function among a set of terms that depend on the graph structure. Due to the non-differentiability of the max, the resulting optimization problem is non-smooth.

In the rest of this chapter we will define a more cooperative approach between control parts which is not based on the best response strategy, but when a controller takes a control decision it takes care of the effects of its actions on the objectives of the other controllers.

8.3.1 Cooperative MPC strategy based on the Subgradient Method

In this section we will define a cooperative distributed MPC approach for controlling the performance behavior and the resource utilization cost of single-source acyclic computation graphs. For these graphs we know that the mean inter-departure time from any module i can be expressed as the maximum of N functions $f_{i,j}$, each one involving the parallelism degree of the j-th module of the graph (where N is the total number of modules), i.e.:

$$T_{p_i}(\mathbf{s}) = \max\Big\{ f_{i,1}(n_1), f_{i,2}(n_2), \ldots, f_{i,N}(n_N) \Big\}$$

where the term $\mathbf{s}$ denotes the vector (strategy profile) $\mathbf{s} = [n_1, n_2, \ldots, n_N]^T$ of parallelism degrees.

In this section we will propose a distributed cooperative optimization approach for solving the global constrained optimization problem with the following properties:

$$\begin{aligned} &\min \sum_{i=1}^{N} J_i(\mathbf{s}) \\ &\text{subject to: } \mathbf{s} \in \mathbf{U} \end{aligned} \tag{8.20}$$

where the N functions $J_i : \mathbf{U} \to \mathbb{R}$ are convex local cost functions for each controller and $\mathbf{U}$ is a closed and convex set of admissible solutions. These two properties are retained by our control problem. The general structure of a local cost function of a ParMod is the sum of a cost proportional to the steady-state inter-departure time from that module and a cost proportional to the number of used nodes. J_i is given by:

$$J_i(\mathbf{s}) = \alpha_i \max_{j=1}^{N} \Big\{ f_{i,j}(n_j) \Big\} + \beta_i n_i$$

From the definition (8.13) we know that each $f_{i,j}$ is a convex function in n_j. Therefore the local cost J_i is convex since: (i) the pointwise maximum of convex functions is a convex function; (ii) the product of a convex function by a positive constant α_i is also convex; (iii) $\beta_i n_i$ is convex in n_i; (iv) the sum of convex functions is convex. As we have seen the domain of the functions J_i (exploiting the continuous relaxation of the parallelism degrees) is given by the following set:

$$\mathbf{U} = \Big\{ \mathbf{s} \;\Big|\; \mathbf{s} = [n_1, \ldots, n_N]^T, \forall i = 1, \ldots, N \, n_i \in \mathbb{R} \wedge 1 \leq n_i \leq n_i^{max} \Big\}$$

which is a closed and convex subset of $\mathbb{R}^N$ by definition.

According to the model-based predictive control strategy this global optimization problem must be solved at the beginning of each control step based on the actual prediction of disturbance inputs. In this section we will often omit to indicate the control step index k from the expressions and variable identifiers for a clearer representation. Also in this case for tractability and feasibility reasons we apply a suboptimal continuous relaxation of the original control problem, since parallelism degrees are real values. This means that the final set of integer parallelism degrees will be selected according to a

proper rounding strategy of the socially optimum set of parallelism degrees calculated by solving the previous global optimization problem.

A direct way to solve this optimization problem is through a centralized approach. All the system measurements from the application graph can be collected by a centralized controller, able to identify the better trade-off between local objectives in such a way to optimize their sum or optionally a weighted sum if some objectives are more important than the others. As discussed in Section 7.1 the situation in which the control logic is concentrated to one point is not desiderable in many practical scenarios, because a unique controller may not have a direct connection with all the application components and for robustness reasons (the control system fails if the centralized controller collapses or it becomes unavailable). For this reason we are interested in proper techniques in which the control problem is solved in a distributed fashion among the set of local controllers. We can note that such class of optimization problems is different from well-studied distributed optimization frameworks [132] that assume local cost functions that depend only on the control action of the corresponding controller, and consider global constraints on the entire set of control variables.

Distributed optimization techniques are based on a partitioning of a large optimization problem into a number of smaller subproblems. A coordination method is usually used to drive the individual solutions of the subproblems towards a solution that is globally optimal. In this way global optimization is performed through local computations of each controller and communications following an interaction protocol. In contrast to a selfish approach also based on local computations and communications, in this case the control action of a controller takes into account its effects on the local objectives of the other controllers.

Due to the non-differentiability of the local costs J_i, in this chapter we will propose the exploitation of a ***distributed subgradient method*** introduced in general contexts in [133]. This distributed optimization model is based on an iterative approach aimed at optimizing the sum of convex and potentially nonsmooth cost functions in a multi-agent environment. In our distributed control problem we remark the following features:

- each control part (agent) knows its local cost function and the model of the mean inter-departure time from its operating part;

- each pair of control parts can communicate directly (if we assume a complete network between them) or they can only communicate with their immediate neighbors (if we consider a partially interconnected network);

- besides taking into account their local objective function, in a cooperative approach controllers exploit a coordination mechanism that ensures they are optimizing the whole global problem. At this regard in this cooperative modeling the input/output interconnecting variables are estimates of the optimal strategy profile instead of single service time variables as in the self-interest approach. This means that controllers maintain an own estimate of the whole optimal strategy profile $\mathbf{s}$ (vector

of parallelism degrees). Controllers exchange their estimates several times until a consensus is reached.

In the following we describe the formulation of the distributed subgradient method as it was presented in [133]. This method is based on the following rules:

- a ***connectivity rule*** about how control agents are interconnected with each other;
- a ***weight rule*** about the weights that will be used for combining the estimates received by a controller with its own local estimate of the optimal strategy profile. We denote with the term $\mathbf{s}_{[i]}^{(q)}$ the estimate of the optimal strategy profile at iteration q from the point of view of the i-th controller.

In [133, 134] the authors have studied the distributed subgradient method for networks with time-varying topologies (e.g. *random networks*) and affected by disconnection events. This method exploits the consensus model presented in [135] based on the following minimal assumption required for the connectivity among agents: *the output interconnecting variables of each controller need to reach each and every controller j directly (if they are directly interconnected), or indirectly (through a path in the graph).* In other words the effects of the control decisions taken by a controller need to propagate in the graph reaching all the other controllers after a finite number of information exchanges. This behavior is similar to the one exploited by the selfish interaction protocol with a partial interconnection between control parts (see Section 8.2.4.2).

In our cases we will always assume a fixed communication topology whereby control part interconnections can not change throughout the execution. In this way the connectivity rule can be satisfied, since:

- if controllers are fully interconnected, for each pair of control parts (i, j) there exists a direct interconnection (in both the directions) between the two controllers;
- if controllers are partially interconnected, we will apply assumption 8.2.9: i.e. two control parts are directly interconnected iff a data stream exists between the corresponding operating parts. In this way if a directed path exists between two parallel computations, a path in both the directions exists between the two corresponding control parts.

In the distributed subgradient method each controller maintains an estimate of the optimal strategy profile from its point of view, starting from some initial point. At each iteration of the protocol each controller updates its current estimate with the estimates received by its neighbors. The current estimate of controller PC_i is updated through the following fundamental relation:

$$\mathbf{s}_{[i]}^{(q+1)} = P_{\mathbf{U}} \left[\sum_{j=1}^{N} \left(W[i,j]\, \mathbf{s}_{[j]}^{(q)} \right) - a^{(q)}\, g_i \right] \tag{8.21}$$

where $a^{(q)} > 0$ is a scalar value called the *stepsize* used by each controller at iteration q and g_i is ***a subgradient*** of the cost function J_i at $\mathbf{s}_{[i]}^{(q)}$. The parameter $W[i,j]$ indicates the weight that controller i associates to the estimate received by controller j and $P_{\mathbf{U}}$ denotes the Euclidean projection onto the admissible convex set $\mathbf{U}$. The meaning of the previous relation is the following. Each controller maintains its own vision of the optimal decision vector of parallelism degrees, which is updated in two phases at each iteration of the method. In the first phase we apply a distributed consensus protocol among the local estimate and the estimates communicated by the neighbors. In the second phase the result of the consensus phase is updated in the negative direction of a subgradient of the local cost function J_i.

The set of weights reflects the structure of the control part interconnections. If the network topology is fixed (as in our cases), the set of weights is statically determined at the beginning of the execution. As demonstrated in [133], in order to assure that each local estimate converges to the social optimum strategy profile, the set of weights must be properly selected according to specific rules. We have denoted with W the square matrix of size $N \times N$ in which each row i is the weight vector used by the i-th controller of the graph. As reported in [133] the first precondition for assuring the convergence and the optimality of the subgradient method is that W is a *doubly stochastic* matrix:

Definition 8.3.1 (Doubly stochastic matrices)**.** A matrix $W \in \mathbb{R}^{N\times N}$ is doubly stochastic iff it is symmetric and each row of the matrix is a stochastic vector. A stochastic vector is one whose entries are from the interval $[0,1]$ and whose entries sum is 1.

Based on the existing interconnections between control parts, we can apply two possible weight rules. The first one can be applied if a complete network between control parts exists:

Definition 8.3.2 (Uniform weights)**.** If control parts are fully interconnected, each controller can use the same weight for its local estimate and for the estimates of the other controllers: i.e. for each PC_i $W[i,j] = 1/N$ for $j = 1,2,\ldots,N$. In this case the weight matrix W is obviously doubly stochastic.

If controllers are partially interconnected we have to exploit a different approach. We can apply the following rule:

Definition 8.3.3 (Symmetric weights)**.** If control parts are partially interconnected, for each controller PC_i we can apply the following weights:

$$\begin{cases} W[i,j] = \min\left\{\dfrac{1}{|Nh(i)|+1}, \dfrac{1}{|Nh(j)|+1}\right\} \text{ if } j \in Nh(i) \\ W[i,j] = 0 \text{ if } j \notin Nh(i) \\ W[i,i] = 1 - \displaystyle\sum_{j\neq i} W[i,j] \end{cases}$$

Example. Let us consider a pipeline graph with four stages M_1, M_2, M_3, M_4. If the corresponding control parts are completely interconnected with each other, we can use the following uniform weight matrix:

$$W = \begin{pmatrix} 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \\ 1/4 & 1/4 & 1/4 & 1/4 \end{pmatrix}$$

which is obviously doubly stochastic. If control parts are partially interconnected, we can apply the symmetric weight rule resulting in the following doubly stochastic matrix:

$$W = \begin{pmatrix} 2/3 & 1/3 & 0 & 0 \\ 1/3 & 1/3 & 1/3 & 0 \\ 0 & 1/3 & 1/3 & 1/3 \\ 0 & 0 & 1/3 & 2/3 \end{pmatrix}$$

As demonstrated in [133] the estimates of all the controllers converge to an approximation of the social optimum strategy profile if the following general conditions are satisfied:

- matrix W is doubly stochastic and is constructed using the uniform weight rule or the symmetric weight rule depending on the two possible interconnection topologies between control parts;
- the information of each controller reaches every other controller (directly or indirectly);
- each local cost function J_i is convex.

In other terms the controllers converge to a vector such that:

$$\lim_{q \to \infty} \mathbf{s}_{[i]}^{(q)} \simeq \mathbf{s}^{(soc)}$$

where $\mathbf{s}^{(soc)}$ is the social optimum. For the stepsize two different rules can be exploited:

- ***constant stepsize rule***: the stepsize $a^{(q)}$ is maintained the same for all the controllers and for all the iterations of the protocol: i.e. $a^{(q)} = a$;
- ***diminishing stepsize rule***: the stepsize is dynamically modified at each iteration of the protocol. The sequence of stepsizes $\{a^{(q)}\}$ has to be a non-increasing sequence that satisfies the following conditions:

$$\sum_{q=0}^{\infty} a^{(q)} = \infty \quad \sum_{q=0}^{\infty} \left(a^{(q)}\right)^2 < A < \infty$$

In [136] the authors proved that with the diminishing stepsize rule the averaged estimates converges to the social optimum strategy profile. With a constant stepsize, the averaged estimates converges to a neighborhood of the optimal point. In this case the error is bounded by a constant term proportional to the stepsize. A smaller stepsize induces a smaller error between the limit point and the social optimum profile at the cost of a higher number of iterations.

8.3.1.1 Subgradient calculus and the Distributed Cooperative algorithm

In order to apply the distributed subgradient method to our example, we need to be able to calculate the subgradient of each local objective. At each iteration every control part PC_i calculates a subgradient $g_{[i]}$ of its local objective J_i. A subgradient of J_i at point $\mathbf{x} \in \mathbf{U}$ is a vector $g_{[i]} \in \mathbb{R}^N$ such that:

$$J_i(\mathbf{z}) \geq J_i(\mathbf{x}) + g_{[i]}^T (\mathbf{z} - \mathbf{x}) \ \forall \mathbf{z} \in \mathbf{U}$$

The subgradient of a non-differentiable function at a given point may not be unique. The *subdifferential* of a function J_i at point $\mathbf{s} \in \mathbf{U}$ is denoted by the symbol $\partial J_i(\mathbf{s})$. It consists in the set of all the subgradients of J_i at that point. The problem of completely identifying the subdifferential of a function is considered a hard problem in general. Fortunately in many convex optimization approaches, as the distributed subgradient method, we are usually required to know a subgradient of the target function at a given point.

The ***weak subgradient calculus*** states the basic rules for finding one subgradient of a target function. The basic rules that we will use are given below:

Differentiable functions: if a function $h(x)$ is differentiable at point x, the unique subgradient of that function at that point is given by its gradient: i.e.

$$\partial h(x) = \{\bigtriangledown h(x)\}$$

Scaling rule: given a function $h(x)$ and $a > 0$ the subdifferential of $ah(x)$ at point x is given by:

$$\partial \left(ah(x)\right) = a \left(\partial h(x)\right)$$

therefore the subdifferential of $ah(x)$ at point x is a set obtained by taking all the subgradients of $h(x)$ multiplied by the scalar value a.

Sum of functions: given two functions $h_1(x)$ and $h_2(x)$ the subdifferential of $h(x) = h_1(x) + h_2(x)$ at point x is given by:

$$\partial h(x) = \partial h_1(x) + \partial h_2(x)$$

where the symbol $+$ corresponds to the Minkowski addition of two sets, formed by the addition of vectors element-wise from the summand-sets. Thus a subgradient of $h(x)$ at x is obtained by summing a subgradient of the first function with a subgradient of the second one.

Pointwise maximum: given the pointwise maximum $h(x) = \max\{h_1(x), \ldots, h_m(x)\}$, the subdifferential of $h(x)$ at point x is given by:

$$\partial h(x) = \mathbf{Co} \bigcup \{\partial h_i(x) \,|\, h_i(x) = h(x)\}$$

the subdifferential of $h(x)$ at point x is given by the convex hull of the union of subdifferentials of the functions h_i that are active at that point. For active we mean the functions that give the maximum value at that point. In other words a subgradient of $h(x)$ at x is given by a subgradient of one of the active functions at that point.

These rules are the basic building blocks for finding a subgradient of the local cost function J_i. The subdifferential of J_i at $\mathbf{s}$ is given by:

$$\partial J_i(\mathbf{s}) = \alpha_i \partial \left(\max_{j=1}^{N} \{ f_{i,j}(n_j) \} \right) + \partial(\beta_i n_i) \tag{8.22}$$

The subdifferential of the second part $\partial(\beta_i n_i)$ is a set composed of a unique vector since this function is differentiable. For the first part we exploit the properties of the subgradient calculus for the subdifferential of a pointwise maximum. To find a subgradient of the maximum of functions $f_{i,j}$ at a given point, we choose any subgradient of the function $f_{i,j}$ that achieves the maximum at that point. Since functions $f_{i,j}$ are differentiable, their subgradients coincide with their gradient.

Now we can introduce the cooperative algorithm performed by each control part. At

Algorithm 6: Cooperative interaction protocol for single-source acyclic graphs based on the Distributed Subgradient Method.

```
foreach control step k each PC_i do
    T_calc−i(k) = Predictive_Filter(...);
    Send/Receive preliminary information from/to other PCs;
    s_[i]^(0)(k) = initial_point;
    q = 0;
    while termination_condition = false do
        transmit the current estimate to neighbors;
        receive estimates from neighbors;
        calculate the subgradient g_[i] of J_i at point s_[i]^(q)(k);
        calculate the new local estimate s_[i]^(q+1)(k);
        q = q + 1;
    use a rounding of the i-th component of s_[i]^(q)(k) as the new parallelism degree
    for step k;
```

the beginning of each control step every PC_i performs a statistical prediction of its mean

calculation time (disturbance input) for the current control step k. Then control parts exchange possible preliminary data with each other. As an example the new predicted calculation times can be disseminated into the computation graph. If control parts are fully interconnected with each other, this can be performed in a straightforward fashion. However the distributed subgradient method can alternatively be executed with partial interconnections among control parts. In this case different solutions can be exploited:

- before starting the distributed subgradient protocol, controllers execute a dissemination algorithm to update the disturbance input predictions of all the nodes of the graph;
- alternatively each controller can transmits its predicted mean calculation time only if it is significantly different from the estimated value at the previous step.

The core of the cooperative algorithm consists in a sequence of iterations starting from an initial estimate that can be:

- a ***fixed initial point***: the initial estimate can be a pre-defined set of parallelism degrees for each ParMod;
- a ***warm start***: i.e. the initial estimate for each controller is the final one calculated at the previous control step of the execution. This warm start approach is motivated by the fact that, if the difference of the values assumed by disturbances between two consecutive control steps is limited (as it is in many practical scenarios), the optimal strategy profile determined at the previous step is an initial point near to the new social optimum for the current control step (and in general better than any fixed initial point).

At each iteration the local estimate is updated according to the distributed subgradient relation (8.21). The algorithm terminates when a stopping criterion is satisfied (e.g. a pre-defined maximum number of iterations has been reached).

8.4 Concluding remarks and final discussion

In this last section we will give the concluding remarks of this chapter. First of all we will make a final overview about the presented interaction protocols for controlling distributed parallel computations. Then we will discuss a final aspect about the service time model of a ParMod.

8.4.1 Overview of the presented interaction protocols

We can summarize the protocols for controlling the behavior of single-source acyclic computation graphs. The results are depicted in Figure 8.11. For the communication-based MPC strategy, in which controllers interact pursuing their self-interest, we have

provided two distinct interaction protocols for reaching the best Nash equilibrium. The first one, based on the complete knowledge assumption and featuring a fully interconnected scheme, reaches the efficient Nash equilibrium in a single information exchange between control parts; the second one, based on the partial knowledge assumption with a fully-interconnected scheme, reaches the same equilibrium profile in a number of iterations that depends on the diameter of the graph, which is a design property of the application. This protocol is especially useful for large computation graphs in which a complete interconnection among controllers is not a viable solution.

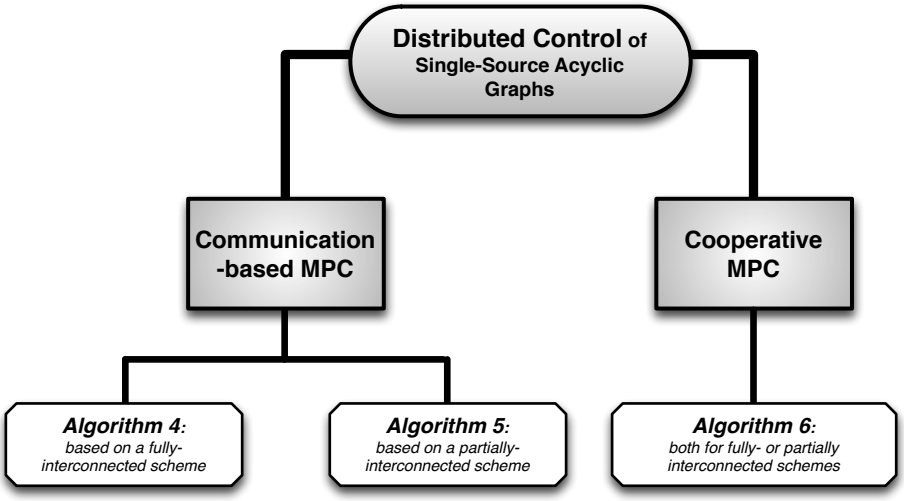


Figure 8.11: Summary of interaction protocols for the distributed control of single-source acyclic computation graphs of ParMods.

For the cooperative MPC strategy we have proposed an interaction protocol for single-source acyclic graphs that can work both with completely interconnected controllers, and when controllers are only partially interconnected with each other. Under proper assumptions this protocol reaches a socially optimal agreement in the control decisions of the controllers.

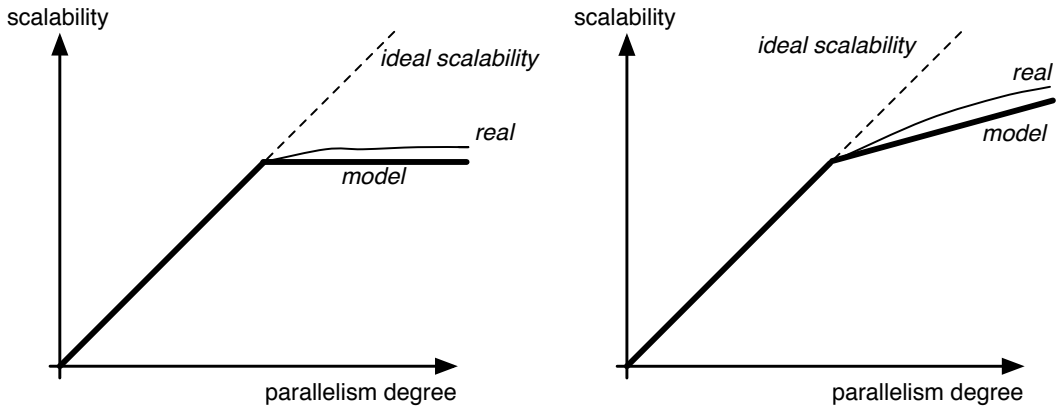
8.4.2 Final considerations about the service time model

So far we have supposed that the internal performance behavior of each parallel module of a computation graph scales perfectly by increasing its parallelism degree. This simple model covers the ideal behavior of a parallel computation: how good a parallelization is can be measured through the *scalability* metric, that indicates how much a parallel implementation accelerates w.r.t the sequential version in function of the parallelism degree. In our approach the internal parallelism is expressed through structured parallelism schemes as task-farm and data-parallel patterns.

In this chapter we have considered parallel modules that feature a perfect scalability: i.e. the service time of a ParMod is expressed as the ratio between the sequential

calculation time of the computation divided by the current parallelism degree. Possible extensions that address non-ideal modelings can be simply defined for the specific cases in which a parallelization does not scale perfectly. For instance classical situations are:

- the scalability of a structured parallel computation on a target architecture is near the ideal one until a specific parallelism degree value is reached. After that the scalability stops to increase or it variates negligibly (this behavior is schematized in Figure 8.12a);
- the scalability of a parallel computation is near the ideal one until a certain parallelism degree value and then it slightly increases for greater values (this behavior is schematized in Figure 8.12b).



(a) Case 1. (b) Case 2.

Figure 8.12: Different non-ideal behaviors of the scalability metric of a structured parallel computation.

In these situations it is possible to modify the service time model for accounting non-ideal behaviors as the ones described before. For instance in the first case we can modify the service time model in order to consider the sudden stop in the scalability:

$$T_{S_i}(k) = \max\left\{\frac{T_{calc-i}(k)}{n_i(k)}, T_i^{min}\right\}$$

For the results and the propositions presented in this chapter the only property that we require for a different non-ideal model of the service time of a ParMod, is that it is a convex function on the parallelism degree of that module.

8.5 Summary

In this chapter we have exemplify the theoretical concepts introduced in Chapter 7 providing a description of a communication-based distributed MPC and a cooperative

distributed MPC strategy for performance and resource utilization control of acyclic computation graphs. For this example we have described the existence of Nash equilibria and how to reach such equilibrium profiles through interaction protocols with different properties. The limits of a selfish interaction have been described in terms of classical metrics as the price of stability. The problem of improving the control quality towards the social optimum strategy profile has been addressed through an existing approach for distributed optimization of nonsmooth convex functions based on the subgradient method.

Chapter 9

Examples of Controlling Acyclic Graphs with a Simulation Environment

In this last chapter of the thesis we will introduce some examples about the problem of controlling acyclic computation graphs of adaptive parallel modules applying models and protocols introduced in the previous chapter. In order to practically study generic graphs with different configurations of controller interactions and control objectives, we will introduce a simulation environment written extending the OMNeT++ discrete event simulator. The first experiment consists in controlling a computation graph in which the main objective is to reach the best performance level with the minimal configuration in terms of number of used nodes for each ParMod. In this example we will compare a communication-based MPC adaptation strategy w.r.t a non-adaptive approach in which the parallelism degree of each ParMod is fixed to the maximum value throughout the execution. This example will show that this adaptation strategy will be able to greatly reduce the average number of used nodes with a minimal loss in the number of completed tasks over the execution. The second experiment will consider a different example in which a large sequence of tasks are scheduled to two different application components, that optimize their behavior applying different trade-offs between performance and resource utilization cost. For this example the quality of two different control strategies, i.e. a communication-based MPC and a cooperative MPC based on the distributed subgradient method, will be compared providing a discussion about the results of these two strategies.

9.1 A simulation environment for ParMod graphs

In this section we will describe the simulation environment[1] that has been used for the experiments described in this chapter. OMNeT++ is a discrete event simulator for modeling communication networks, multiprocessor architectures and also generic distributed parallel systems. The most common use of OMNeT++ is for simulation of communica-

[1] I have to thank Daniele Buono for his crucial help in the development of the simulation environment.

tion networks and IT systems, but it is also used for queuing network simulations. The simulator has a component-oriented approach in which the programmer is encouraged in defining modules and complex hierarchy of modules that can be instantiated multiple times with different parameters in network structures. This environment has been extended for developing a simulation module that re-produces the behavior of a generic ParMod, i.e. the basic building block of our approach for composing adaptive distributed applications.

9.1.1 An OMNeT++ module simulating an Adaptive ParMod

As said our ParMod model is composed of two interconnected entities, an operating part performing a reconfigurable structured parallel computation and a control part that executes a proper adaptation strategy and interacts with other control parts of the graph. At a first point we need to face with the problem of simulating a structured parallel computation, and reproduce its behavior with different configurations. For simplicity and in order to apply the control modeling introduced in the previous chapter, we have considered only non-functional reconfigurations based on dynamic changes of the current parallelism degree. A schematized description of the OMNeT module simulating a ParMod is given in Figure 9.1.

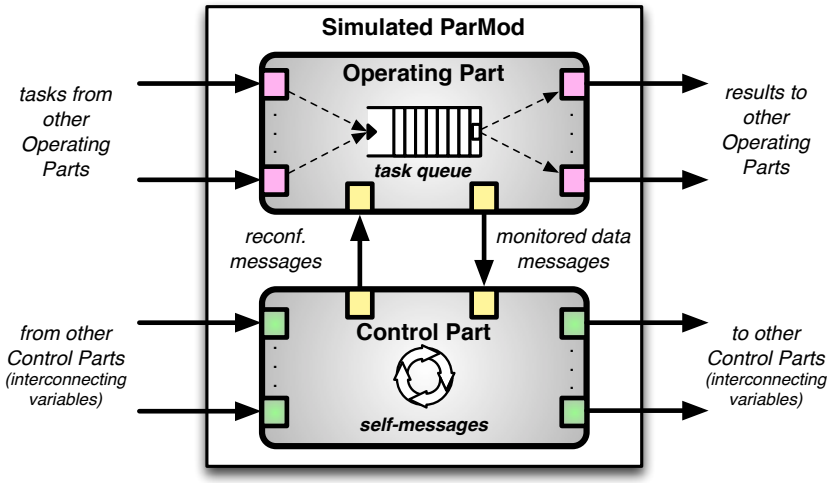


Figure 9.1: Adaptive ParMod simulated through an OMNeT module. Small square boxes represent ports used for communications between OMNeT modules. Pink ports are used to interconnect operating part sub-modules of different ParMods, green ports interconnect distinct control parts and yellow ports are used for the internal closed-loop interaction between the operating part and the control part of a ParMod.

The ParMod object is composed of two sub-modules: a simple module implementing the operating part and a simple module implementing the control part. The behavior of a generic simulation module can be programmed following an event-driven programming

style. A module can receive different classes of messages from other modules. Each time a new message is received, the `handlemessage()` routine is called automatically. Inside the definition of this routine the programmer can specify different handlers based on the type of the received message, and also generate new messages that will be transmitted to other modules. Communications are exploited through the definition of *ports*: each port is binded with a port of another module in such a way that each message transmitted using a local port will be delivered to a well-identified destination.

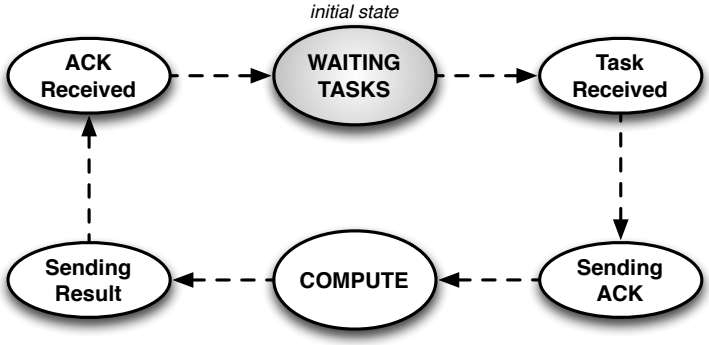


Figure 9.2: Abstract behavior of the interaction between different simulation modules implementing operating parts of ParMods.

In order to simulate the behavior of a structured parallel computation operating on a stream of input tasks, the operating part simulation module implements a queue logic with a blocking-after-service semantics depicts in abstract terms in Figure 9.2. In other words the operating part is simulated through a queueing station. Tasks from other operating parts are received by the input ports and are buffered into a queue. To reproduce a blocking semantics we have implemented a communication protocol based on the transmission of `SEND` and `ACK` messages. If the received task can be queued (i.e. there is a free position in the buffer), the operating part module transmits an ACK message to the sender. Otherwise, if there is no free space in the queue actually, the received task and other information (e.g. the sender identifier) are stored in a special data-structure and the ACK transmission will be delayed until a position in the buffer is freed. The sender implements a simple protocol in which when it transmits a task message to a destination, it has to wait for an explicit ACK message before continuing the execution.

When a new task has been extracted from the queue of the operating part module, the execution of a structured parallel computation with a parametric parallelism degree will be simulated. For this reason we have implemented two different working logics:

- after the extraction of a task from the input queue, the operating part module waits a time equal to the ratio between its calculation time and its actual parallelism degree. After that a corresponding result message will be delivered to one of the destination port of the module, selected with a specific probability among the set of output ports. This behavior reproduces a generic *data-parallel* computation, in which by

increasing the parallelism degree it is possible to improve the performance both in terms of service time and computation latency (see Section 3.1);

- other structured parallelism schemes, as the *task-farm*, are able to improve only the service time by increasing the parallelism degree of the computation, without any improvement of the computation latency per task. In order to reproduce this behavior, the operating part module is able to process multiple input tasks in parallel, with a maximum number given by the actual parallelism degree. In this case each task is processed by the operating part module waiting a time equal to its calculation time and producing a corresponding result that will be transmitted to an output port of the module selected with a given probability.

The parameters of the operating part module are given below:

- an integer value that specifies the actual parallelism degree used by the operating part;
- the calculation time is a random variable with a specific distribution and two parameters indicating the mean value and the variance;
- the queue size, i.e. the total number of buffer positions in the queue, which is a design parameter of the module that can not be changed during the execution;
- a set of probabilities for transmission of results to the output ports. The set of probabilities is a discrete probability distribution, i.e. their sum must be equal to 1.

In this simulation environment we have not modeled the size of tasks and thus a variable communication latency for communications between operating parts. However the cost for performing communications can be considered implicitly by increasing the mean calculation time per task correspondently.

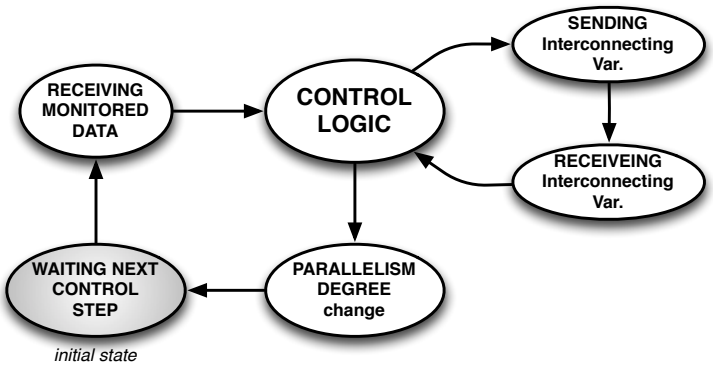


Figure 9.3: Abstract behavior of the simulation module implementing the ParMod control part.

A second OMNeT module has been developed for simulating the control part behavior. Control part is responsible for analyzing monitored data received by the operating

part at each control step, and applying the adaptation strategy exchanging information with other control parts of the application graph. This module transmits and receives four different classes of messages:

- *interconnecting variables messages* from other control parts: these messages contain information depending on the adaptation strategy executed by the control part. For instance in the case of the communication-based MPC strategy, each message contains the unique identifier of the sender and the actual value of the service time variable. Otherwise in the case of the cooperative MPC approach based on the distributed subgradient method, the message contains the actual estimate of the strategy profile advertised by the sender control part. Moreover this class of messages will also be used for exchanging other useful information between controllers, as we will see later in this chapter;

- *monitored data messages* from the operating part of the ParMod: these messages are received by the control part at the beginning of each control step of the execution, in order to obtain useful monitored data of the operating part execution (e.g. the average value of the calculation time per task of the last control step);

- *self-messages* auto-generated by the control part module: in the OMNeT simulation environment it is possible for modules to generate self-messages that can be handled in the same way as the other received messages. This behavior can be achieved with the `scheduleAt()` call, that schedules a self-message at a given simulation time instant. This functionality makes it possible a simple implementation of the control part discrete-time behavior. Self-messages are generated every control step, and the control logic is started each time a self-message is received;

- after the evaluation of the control logic, the control part module transmits the new calculated parallelism degree to the operating part through proper *reconfiguration messages*. The operating part module, received the new parallelism degree, updates its internal parameter correspondently.

The OMNeT++ simulator provides useful tools for collecting several statistics during the execution of an application. In our case for each ParMod we will collect the following parameters:

- the mean service time of a ParMod for each control step of the execution, given by the ratio between the mean calculation time and the used parallelism degree;

- the mean calculation time and the probabilities of transmission of results to different destinations for each control step of the execution;

- the mean inter-departure time from a ParMod for each control step, given by the average time between the transmission of two consecutive results;

- the utilization factor of a ParMod for each control step of the execution;

- the total number of tasks completed by a ParMod over the execution;
- the parallelism degree used by a ParMod for each control step;
- for some experiments it is also useful to collect the local cost of the ParMod (i.e. the value of function J_i) during each control step of the execution.

9.2 First example: optimizing the performance with the lowest number of used resources

In the first experiment of this chapter we will describe the utilization of the simulation environment for controlling the performance and the resource utilization cost of a simulated distributed parallel computation composed of eight ParMods, interconnected in the acyclic graph depicted in Figure 9.4. The graph is composed of eight modules whose con-

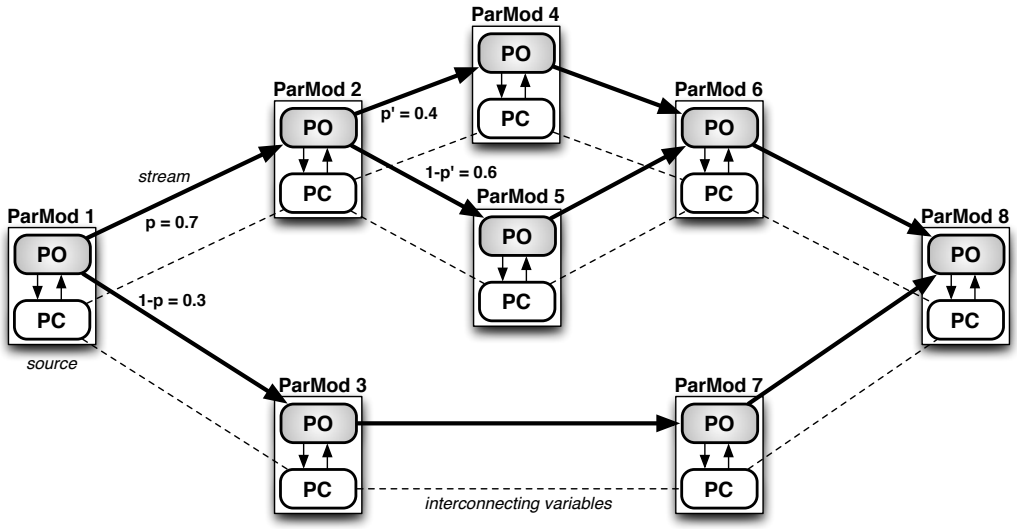


Figure 9.4: First simulated application: an acyclic graph of adaptive ParMods.

trol parts are partially interconnected with each other, i.e. two control parts are directly interconnected (in both the directions) iff there exists a stream of tasks between the two corresponding operating parts. For this example we will apply the modeling introduced in Chapter 8, in which each ParMod executes non-functional reconfigurations (i.e. changes in the parallelism degree of the module) in order to optimize its local cost function over the execution. The cost function is exactly the one introduced in the previous chapter:

$$J_i(k) = \alpha_i T_{p_i}(k+1) + \beta_i n_i(k)$$

in which the local cost consists in a first term which depends on the effective performance behavior of the module (i.e. its mean inter-departure time), and a second term which is a

cost proportional to the used parallelism degree. In this applicative scenario we suppose the following trade-off between performance and resource cost:

Assumption 9.2.1. *In this example we will suppose that the first control objective of each ParMod is to reach the best performance level (i.e. the lowest mean inter-departure time). Then, in order to do this, the ParMod tries to use strictly necessary number of resources.*

This special trade-off is mapped onto the local cost function by choosing $\alpha_i >> \beta_i$. In this way the mean inter-departure time has the highest impact in the cost function, so it is firstly important to minimize it. Then, if distinct configurations make it possible to achieve that minimum inter-departure time, we will select the one with the smaller number of used nodes. This control objective has a notable consequence in our modeling, as explained below:

Proposition 9.2.2. *In this control problem the efficient Nash equilibrium corresponds to the social optimum strategy profile of the game.*

By selecting $\alpha_i >> \beta_i$ the ideal parallelism degree of a generic $ParMod_i$ is equal to the maximum parallelism degree for that module, i.e. n_i^{max}. This means that, if we do no consider the performance coupling relationship between modules, the best parallelism degree for achieving the control objective corresponds to the maximum one. On the other hand due to the coupling relationships, the steady-state performance of a ParMod can be limited by the other modules of the graph. In this case a ParMod can be using a over-sized parallelism degree, and it can reduce it maintaining the same performance level. At this point we can provide a proof of the previous proposition:

Proof. The social optimum strategy profile is a set of (real-valued) parallelism degrees such that the sum of the local cost functions is minimized. With this strategy profile the utilization factor of each ParMod has to be equal to 1. This can be proved by absurd. If we suppose that with the social optimum strategy profile one ParMod has a utilization factor lower than 1 (greater than 1 is not possible due to steady-state conditions, see Proposition 4.1.1), it can slightly reduce its parallelism degree without modifying its effective performance. In this way that module can improve the value of its local cost function without making the local costs of the other ParMods worse off. Therefore the social optimum strategy profile is not Pareto efficient, which is absurd. Moreover we can observe that each ParMod: (1) or it is the bottleneck so it selects the highest parallelism degree n_i^{max}, which is also its ideal parallelism degree in this problem, or (2) it selects a smaller parallelism degree than the ideal one. Therefore the social optimum profile is a set of parallelism degrees such that: (i) all the ParMods have a utilization factor equal to 1; (ii) the bottleneck ParMod is selecting its ideal parallelism degree (which is its maximum one); (iii) all the other ParMods select a parallelism degree lower than their ideal (maximum) one. We can observe that these conditions exactly coincide with the ones for identifying the unique efficient Nash equilibrium of the game (see Proposition 8.2.7), so that it corresponds to the social optimum strategy profile. □

Therefore the control parts of the graph need to exchange messages in order to calculate the efficient Nash equilibrium at each control step of the execution, and select a proper integer approximation of their corresponding parallelism degree. In the previous chapter we have seen that the communication-based MPC approach is able to reach the efficient Nash equilibrium. Due to the partial interconnection between control parts we will exploit Algorithm 5 as the interaction protocol between controllers. The definition and the features of this adaptation strategy will be discussed in the next section in more detail.

9.2.1 Applying the Communication-based MPC strategy

For controlling this experimental application we will exploit a communication-based MPC approach based on a partial interconnection between control parts. In order to reproduce a dynamic execution condition and evaluate the effectiveness of this adaptation strategy, we will simulate dynamic changes in the optimal application configuration.

Assumption 9.2.3. *We will suppose a varying mean calculation time of the source module (i.e. $ParMod_1$). In other words we will assume that the mean generation rate of tasks from the unique source of the graph can change significantly during the execution. Therefore the social optimum set of parallelism degrees can change at each control step.*

For the other ParMods of the graph we will suppose that their calculation times maintain the same average value and variance over the execution (they are random variables with normal distribution and a small standard deviation). Due to the dynamicity of the source module behavior, e.g. there are phases of the execution in which tasks are generated faster than during other periods, the optimal set of parallelism degrees can change during the execution. In order to find the best configuration at each control step, the application modules interact according to the communication-based distributed MPC strategy. At the beginning of each control step k the generic $ParMod_i$ will execute the following sequence of actions:

- at the beginning of each control step k the controller of the i-th ParMod receives monitored data messages from its operating part. Monitored information consists in the past value assumed by the mean calculation time of the ParMod (it is the disturbance input of the model) during the last control step $k-1$;

- based on this current measurement and on the history of past observations, the controller is able to exploit a statistical prediction of its mean calculation time. In this problem *the prediction horizon is limited to one single control step*, therefore the controller exploits a prediction of the value that the mean calculation time will assume during the k-th control step;

- for a fixed number of iterations, equal to the diameter of the graph (in this example the diameter is 4), the controller calculates its actual optimal parallelism degree in

function of the received interconnecting variables and transmits its service time to its neighbor controllers (see Algorithm 5);

- after the last iteration, the set of parallelism degrees selected by each ParMod is the efficient Nash equilibrium, which in this problem corresponds to the social optimum strategy profile. Next each controller applies to its operating part an integer approximation of its equilibrium parallelism degree.

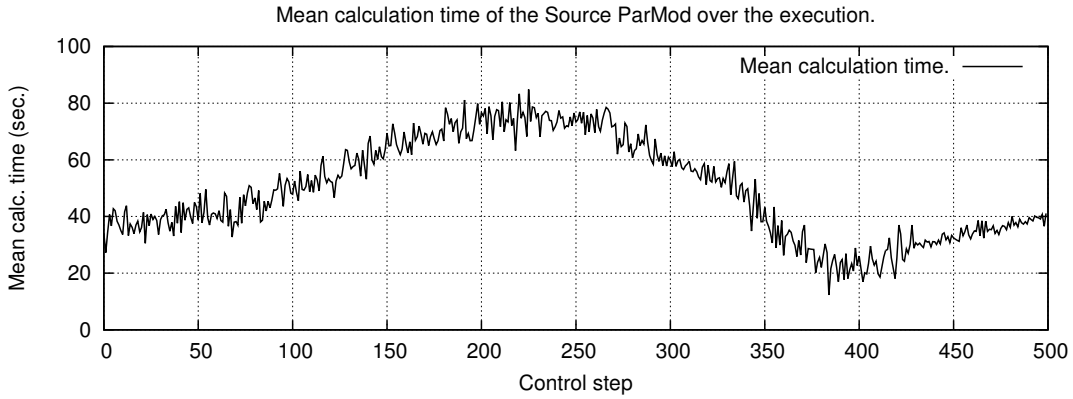


Figure 9.5: Trace-file of the variability of the mean calculation time of the Source ParMod.

As said before, while the other modules maintain the same mean calculation time over the execution, we will suppose that this parameter is highly variable for the source module. For this experiment we have considered a control step length of 120 seconds[2] and and execution of 500 control steps, which is equal to more than 16 hours of the simulated execution.

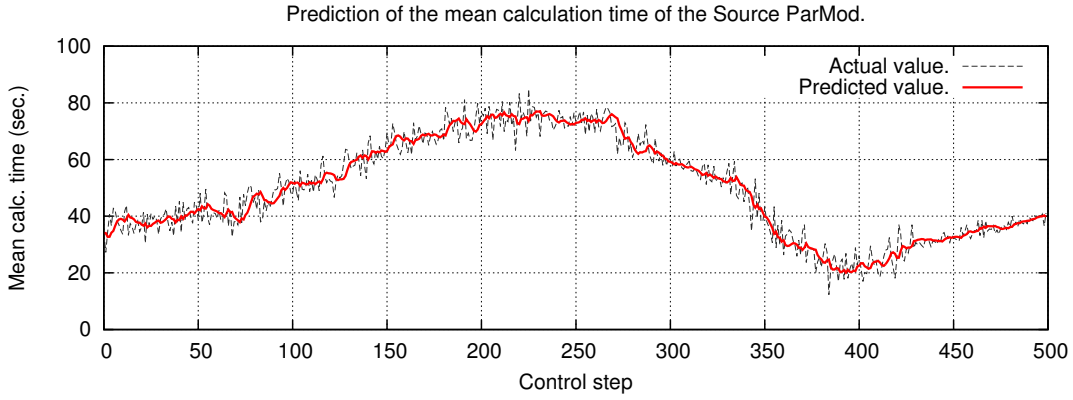


Figure 9.6: Prediction of the mean calculation time per task of the source ParMod with the Holt-Winters filtering technique.

[2]Seconds will be adopted as our time unit for this experiment. It is important to note that this time concept is abstract in the OMNeT++ simulator, which gives the possibility to consider also the same execution with a different time-scale, e.g. milliseconds instead of seconds, depending on the application scenario.

Our experiment starts by considering a dynamic behavior of the mean calculation time per task of the source ParMod, which generates tasks with a varying generation time. This behavior is typical of many real-world applications in which the input task rate of a system changes due to the user behavior or network conditions. We have considered a trace-file example depicted in Figure 9.5. The figure shows the average values assumed by this parameter for each control step of the execution. We can observe that the average value is initially near to 40 seconds per tasks and it slowly increase during the execution reaching a peak of about 80 seconds after more than 200 control steps, after that it slowly decreases reaching values near to the initial conditions at the end of the execution. In this experiment this parameter is the main source of dynamicity to which the modules of the application graph need to adapt. As said for the other ParMods we suppose that their mean calculation times do not change significantly. Their values are summarized in Table 9.1.

ParMod	**Mean** T_{calc}
S_2	64 sec.
S_3	40 sec.
S_4	40 sec.
S_5	120 sec.
S_6	26 sec.
S_7	38 sec.
S_8	24 sec.

Table 9.1: Mean calculation times for the other ParMods of the application graph.

In order to apply the MPC strategy, each control part predicts the future value assumed by its mean calculation time for the current control step. In a similar way w.r.t the experiments of Chapter 6, we have applied the Holt-Winters statistical filter for exploiting the one-step ahead prediction of the mean calculation time for each ParMod. This prediction is especially important for the source ParMod. In Figure 9.6 it is shown the same time-series of Figure 9.5 (with a black dashed line this time) and the predicted values with the Holt-Winters filter (solid red line) that produces a root mean square relative error (RMSRE) less than 8% over the execution.

9.2.2 Effectiveness of the Adaptation Strategy and results discussion

In this experiment each controller optimizes its local objective function in order to reach the best performance with the strictly necessary number of used resources. According to the interaction protocol described in Section 8.2.4.2, expressions (8.1) provide the inter-departure time model of each ParMod in function of its service time and of the service times of its neighbors. In these expressions $T_{calc-i}(k)$ is the mean calculation time of $ParMod_i$ during the k-th control step, whereas $n_i(k)$ is its parallelism degree. $T_{S_i}(k)$ is the actual service time of $ParMod_i$ which is also its output interconnecting variable, while

$T_{S_j}(k)\ \forall\, j \in Nh(i)$ are the interconnecting variables received by the neighbors. Moreover p is the routing probability from $ParMod_1$ to $ParMod_2$ and p' from $ParMod_2$ to $ParMod_4$, which are fixed over the execution to 0.7 and 0.4 respectively (see Figure 9.4).

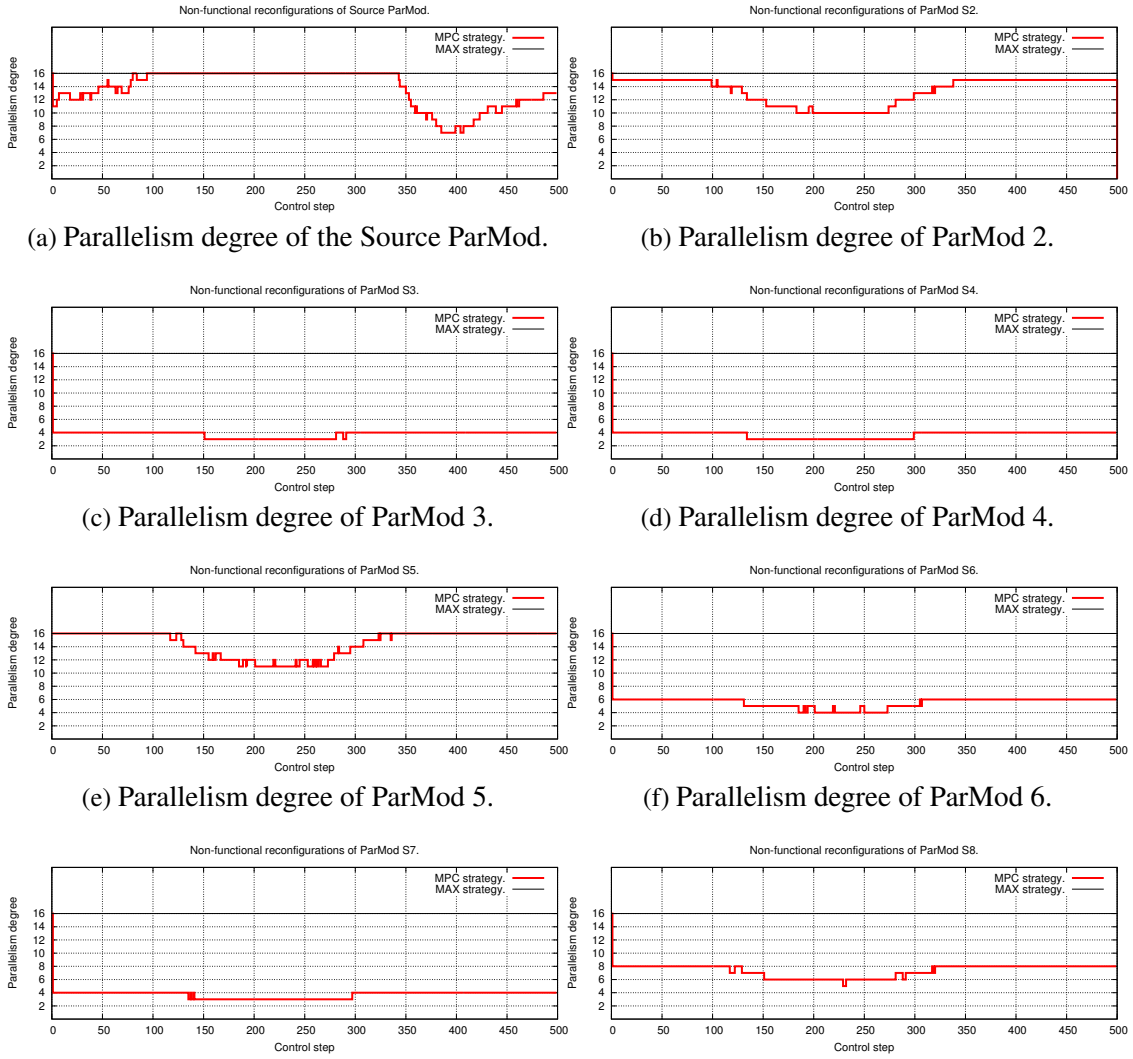
$$
\begin{aligned}
T_{p_1}(k+1) &= \max\left\{\frac{T_{calc-1}(k)}{n_1(k)},\, p\,T_{S_2}(k),\, (1-p)\,T_{S_3}(k)\right\}\\
T_{p_2}(k+1) &= \max\left\{\frac{T_{calc-2}(k)}{n_2(k)},\, \frac{T_{S_1}(k)}{p},\, p'\,T_{S_4}(k),\, (1-p')\,T_{S_5}(k)\right\}\\
T_{p_3}(k+1) &= \max\left\{\frac{T_{calc-3}(k)}{n_3(k)},\, \frac{T_{S_1}(k)}{(1-p)},\, T_{S_7}(k)\right\}\\
T_{p_4}(k+1) &= \max\left\{\frac{T_{calc-4}(k)}{n_4(k)},\, \frac{T_{S_2}(k)}{p'},\, \frac{T_{S_6}(k)}{p'}\right\}\\
T_{p_5}(k+1) &= \max\left\{\frac{T_{calc-5}(k)}{n_5(k)},\, \frac{T_{S_2}(k)}{(1-p')},\, \frac{T_{S_6}(k)}{(1-p')}\right\}\\
T_{p_6}(k+1) &= \max\left\{\frac{T_{calc-6}(k)}{n_6(k)},\, \frac{T_{S_8}(k)}{p},\, p'\,T_{S_4}(k)\,,\, (1-p')\,T_{S_5}(k)\right\}\\
T_{p_7}(k+1) &= \max\left\{\frac{T_{calc-7}(k)}{n_7(k)},\, T_{S_3}(k)\,,\, \frac{T_{S_8}(k)}{(1-p)}\right\}\\
T_{p_8}(k+1) &= \max\left\{\frac{T_{calc-8}(k)}{n_8(k)},\, p\,T_{S_6}(k)\,,\, (1-p)\,T_{S_7}(k)\right\}
\end{aligned}
\tag{9.1}
$$

Each of the expressions indicates how the mean inter-departure time from an application module depends on the ideal service time of the module and of its neighbors. Based on Algorithm 5 control parts interact exchanging the service time variables. Received the interconnecting variables from the neighbors, a ParMod calculates the parallelism degree that optimizes its local objective function and transmits the new service time variable to its neighbors. In order to correctly establish the steady-state behavior of a module, this information exchange needs to be executed a number of times equal to the diameter of the graph (i.e. *four iterations* in this case). In this way the effects of coupling relationships completely propagate in the network, and each controller is able to evaluate its effective inter-departure time accounting the behavior of all the modules of the graph.

Due to the variability of the source mean calculation time, the other ParMods adapt to an increase or a decrease of this value by modifying their parallelism degree, in order to achieve the lowest inter-departure time with the minimum number of used resources. We have performed the simulation comparing the Communication-based MPC strategy with the following strategy:

Definition 9.2.1 (MAX Strategy)**.** In the MAX strategy we maintain the parallelism degree of each ParMod equal to the maximum number of nodes, i.e. n_i^{max} throughout the execution.

This comparison represents an important experiment. Since we have assumed an ideal scalability of each parallel module, by selecting the maximum parallelism degree for each



(a) Parallelism degree of the Source ParMod.

(b) Parallelism degree of ParMod 2.

(c) Parallelism degree of ParMod 3.

(d) Parallelism degree of ParMod 4.

(e) Parallelism degree of ParMod 5.

(f) Parallelism degree of ParMod 6.

(g) Parallelism degree of ParMod 7.

(h) Parallelism degree of ParMod 8.

Figure 9.7: Parallelism degree variations over the execution.

ParMod we are sure that the best performance will be achieved. In other words with this configuration the application graph is able to complete the highest number of tasks during the simulated execution time. Of course in this non-adaptive scenario certain modules will use an oversized number of nodes. Ideally the objective of an adaptation strategy is to adapt the ParMod parallelism degrees to the current (or predicted) mean generation rate of the source module, in order to properly decrease the number of used nodes keeping the effective performance of the graph as similar as possible to the one of the MAX strategy. In reference to the variability of the source behavior depicted in Figure 9.6, the parallelism degree variations with the communication-based MPC strategy are depicted in Figure 9.7.

At the beginning of the execution each module uses its maximum parallelism degree

that for simplicity we assume it is equal to 16 nodes for each ParMod. In the MAX strategy this parallelism degree is fixed throughout the execution.

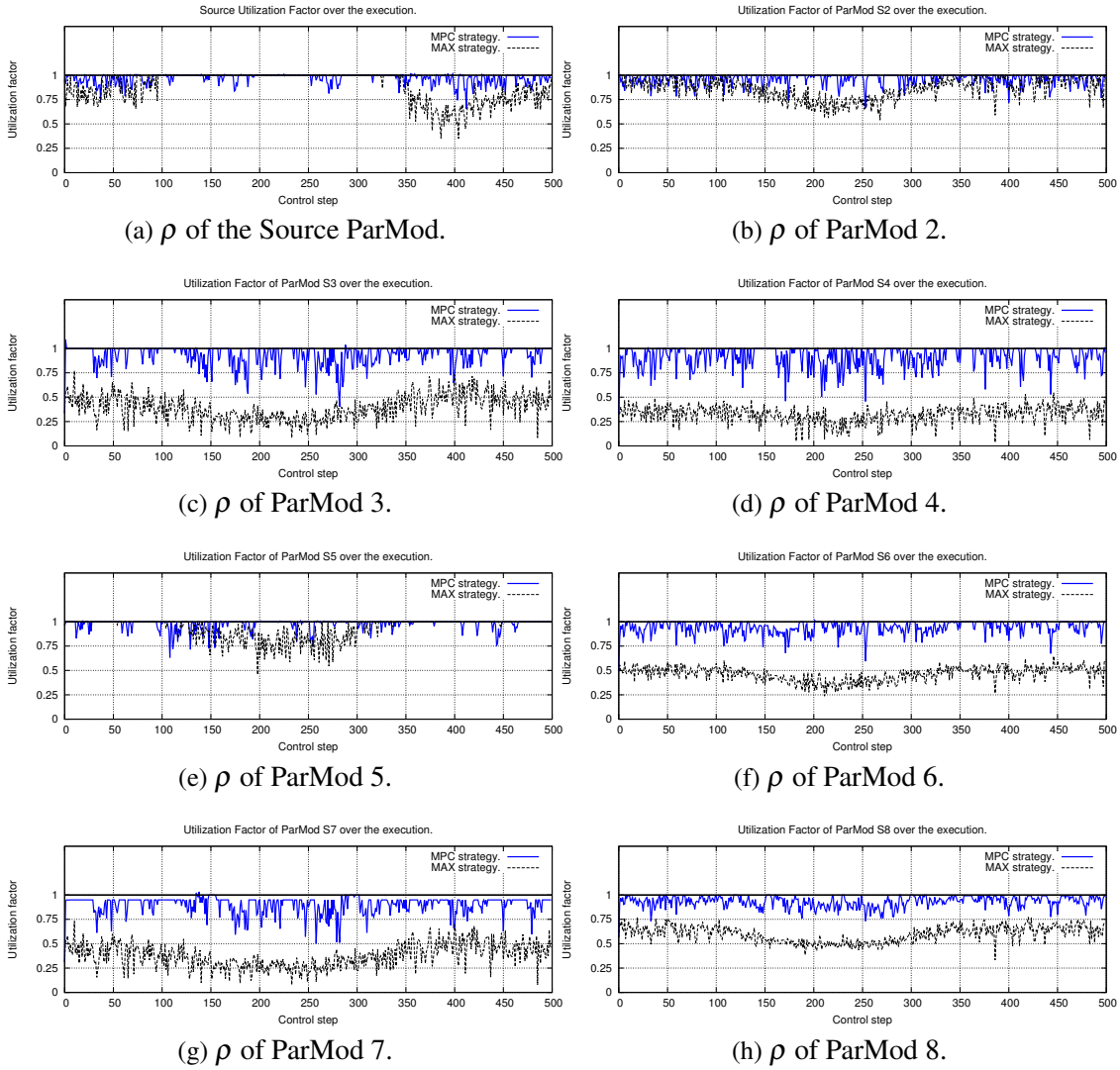
Let us discuss the results of Figure 9.7. At each control step the controllers of the graph interact in order to find the efficient Nash equilibrium, that changes over the execution due to the variability of the source mean calculation time. Then controllers select an integer approximation of the final result of the interaction protocol. We can observe that from the beginning of the execution until about control step 100, the mean calculation time of the source increases but $ParMod_1$ is not yet the bottleneck of the graph. In fact we can note that during this period the parallelism degree chosen by the source increases passing from 11 to 16 nodes. In this time period the application bottleneck is represented by $ParMod_5$, which always uses its maximum parallelism degree. In this phase $ParMod_1$ is the only one that performs reconfigurations: i.e. due to an increase of its mean calculation time, the source module increases its parallelism degree until it reaches the maximum value.

A different behavior happens from about control step 100 to 340. The source ParMod becomes the bottleneck of the graph. This means that the mean inter-departure time from each module now depends on the current source service time. In this phase the mean calculation time of the source increases reaching a peak of 80 sec. and then starts to slowly decrease. Due to the bottleneck condition, the source maintains the maximum parallelism degree throughout this period (Figure 9.7a), while the other ParMods adapt their parallelism degree by firstly decreasing it (to adapt to the growth of the source calculation time) and then increasing it. As an example the second module (Figure 9.7b) passes from 15 nodes to 10 nodes while $ParMod_5$ (Figure 9.7e) from 16 to 11.

After about control step 340 the application graph returns to behave in a similar way as during the initial phase of the execution. Due to the decrease of the source mean calculation time, that reaches a minimum value of about 20 seconds near control step 380, $ParMod_1$ stops to be the bottleneck of the graph and the inter-departure times of the modules restart to be influenced by the service time of $ParMod_5$. This behavior is clearly highlighted by the fact that after control step 340, $ParMod_5$ uses the maximum parallelism degree. In this last phase only $ParMod_1$ adapts its parallelism degree passing from the maximum one to a minimum of 7 nodes in correspondence to the minimum peak of its mean calculation time (near control step 380).

Another important set of experiments are described in Figure 9.8. In this figure is shown the utilization factor ρ of each ParMod at each control step of the execution. We recall that the utilization factor of a computation module is the ratio between its mean service time, that describes the theoretical performance level of the module given its actual parallelism degree, and the mean inter-arrival time to that module. We known that at steady-state the inter-arrival time to a module coincides with its inter-departure time (see Proposition 4.1.1), therefore we can calculate the utilization factor as the ratio between the mean service time of the module (theoretical behavior) and its mean inter-departure time (effective behavior) during the control step. Therefore the utilization factor represents an efficiency measurement that indicates how the resources the module are well exploited. Values near to 1 represent the ideal situation in which the resources are well-utilized,

otherwise lower values correspond to an under-utilization of the parallelism degree of the module.



(a) ρ of the Source ParMod.

(b) ρ of ParMod 2.

(c) ρ of ParMod 3.

(d) ρ of ParMod 4.

(e) ρ of ParMod 5.

(f) ρ of ParMod 6.

(g) ρ of ParMod 7.

(h) ρ of ParMod 8.

Figure 9.8: Utilization factor of each ParMod: comparison between MPC and MAX strategies.

As we have seen in the previous chapter a property of Nash equilibrium strategy profiles is that the utilization factor of each module is equal to 1. In practise a utilization factor exactly equal to 1 can not be reached due to two main reasons:

- because the efficient Nash equilibrium is determined based on actual predictions of the mean calculation times of each ParMod. In general these predictions are not the exact values;
- each controller will concretely select an integer approximation of its equilibrium parallelism degree. As we have seen different approximations can be exploited

(e.g. to the upper integer in order to optimize the performance with a slightly higher resource utilization cost).

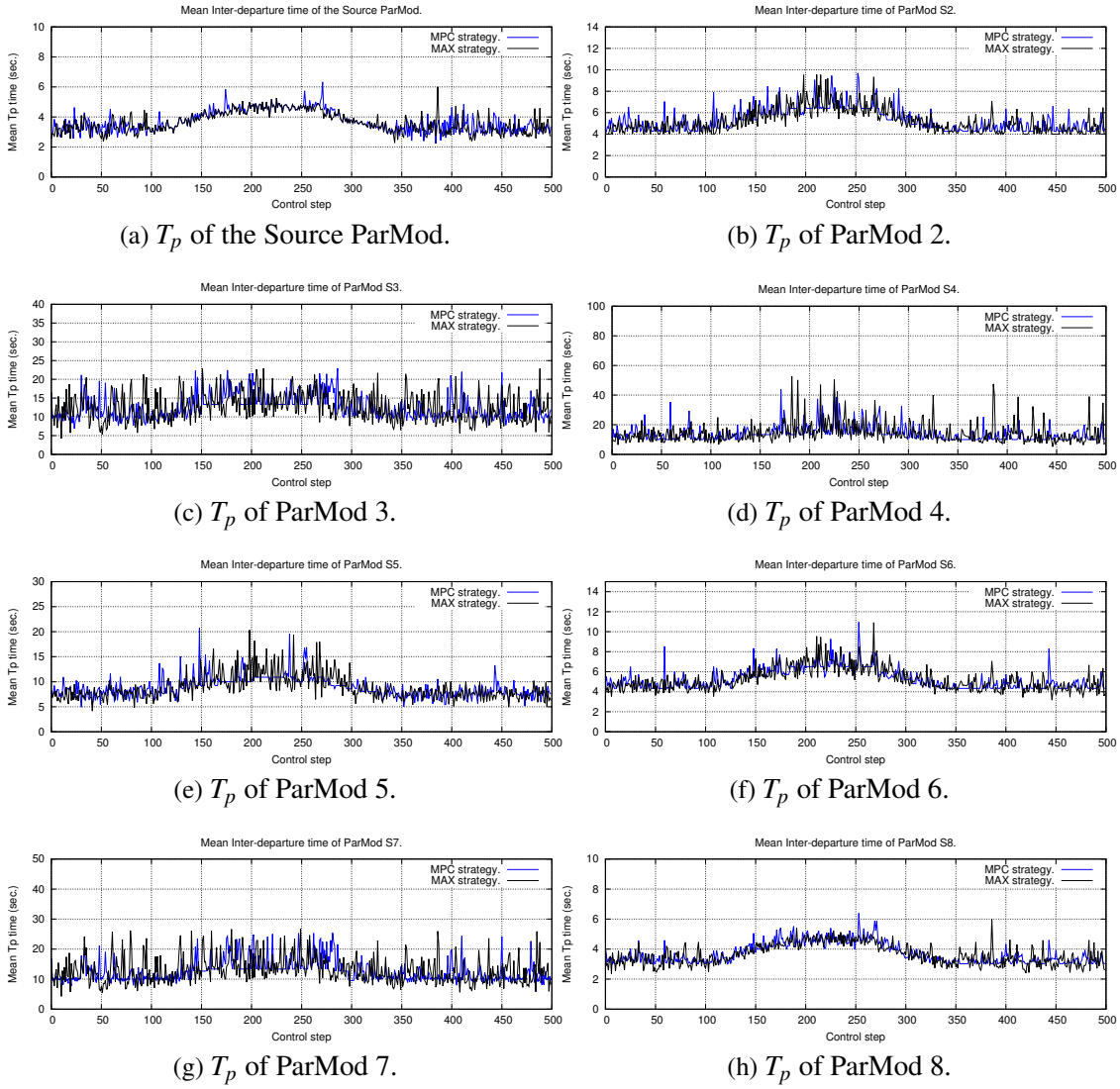
With the communication-based MPC approach we are able to achieve a better utilization of the system resources w.r.t the MAX strategy (Figure 9.8). For all the ParMods their utilization factors with the distributed MPC are higher and nearer to 1 than with the MAX strategy. This is particularly evident for ParMod 2, 3, 4, 6, 7 and 8. For $ParMod_1$ and $ParMod_5$ the behavior is extremely interesting. We can note that the source ParMod, from about control step 100 to 340 in which it is the bottleneck, has a utilization factor near to 1 both with the MPC and the MAX strategy. The same thing happens for $ParMod_5$ before control step 100 and after control step 340. In fact during these execution intervals, those modules are the application bottleneck and they select the maximum parallelism degree both with the MAX strategy and the MPC strategy.

Figure 9.9 shows the mean inter-departure time T_p per control step of each ParMod during the execution. The black line indicates the inter-departure time with the MAX strategy and the blue line with the distributed MPC strategy. As we can observe the performance behavior is extremely similar. For each ParMod the black line and the blue line behave in a similar way. This fact has a clear meaning: the performance level achieved by each module with the MAX strategy, that exploits the maximum number of nodes, is similar to the one achieved with the MPC strategy in which the parallelism degree is properly regulated in order to avoid to waste resources unnecessarily. Moreover in correspondence to the execution phase in which the source module is the bottleneck (from about control step 100 to 340), the mean inter-departure time from each ParMod is higher than in the other phases of the execution.

This concept is further highlighted in Table 9.2, in which it is reported the number of completed tasks for each ParMod throughout the execution. The table compare the MAX strategy, which ensures the best performance level by over-sizing the parallelism degrees, with the MPC approach with different integer rounding techniques. Herein we discuss three rounding techniques: (i) a performance-conservative rounding in which the real parallelism degrees are rounded to the smaller upper integers ($MPC \uparrow$); (ii) a resource-conservative approach in which the real parallelism degrees are rounded to the highest lower integer ($MPC \downarrow$); (iii) a classic approximation in which each parallelism degree is rounded up or down to the nearest integer ($MPC \sim$).

From the table we evince that, especially with the MPC $\uparrow$ rounding, the number of completed tasks of each ParMod is slightly lower than the best case of the MAX strategy. In Table 9.3 this result is more evident by considering the percentage of task loss (Δ) compared to the MAX strategy. As we can see with the MPC $\uparrow$ approach each ParMod completes a number of tasks which is 2% smaller than with the MAX strategy. This percentage increases if we exploit other rounding techniques: it is near to 8% with MPC $\sim$ and to 22% with MPC $\downarrow$.

Besides a similar performance behavior compared to the MAX strategy (especially with MPC $\uparrow$), the MPC is able to save a great amount of used nodes during the execution. In Table 9.4 it is shown the mean number of nodes used by each ParMod with the MPC $\uparrow$



(a) T_p of the Source ParMod.

(b) T_p of ParMod 2.

(c) T_p of ParMod 3.

(d) T_p of ParMod 4.

(e) T_p of ParMod 5.

(f) T_p of ParMod 6.

(g) T_p of ParMod 7.

(h) T_p of ParMod 8.

Figure 9.9: Mean inter-departure time from each ParMod: comparison between MPC and MAX strategies.

strategy and the mean percentage of saved nodes during the execution. With this adaptation technique the application execution requires total average number of nodes equal to 65.74, which is about half the total number of nodes that would be used with the MAX strategy (i.e. 128 nodes for the entire application execution).

In conclusion this first experiment shows that by adopting the communication-based MPC approach each ParMod of the application graph is able to control its resource utilization maximizing its effective performance behavior. These results demonstrate that, proper predictions of the generation rate of the source module and the exploitation of the distributed MPC strategy, make it possible to reduce in a significant way the average number of used resources maintaining a very similar performance level than the more

ParMod	**(MAX)**	**(MPC ↑)**	**(MPC ↓)**	**(MPC ∼)**
S_1	17058	16756	13266	15778
S_2	11951	11732	9306	11020
S_3	5096	5009	3949	4746
S_4	4826	4701	3706	4485
S_5	7122	7021	5590	6526
S_6	11947	11721	9292	11010
S_7	5096	5008	3948	4745
S_8	17043	16728	13238	15754

Table 9.2: Number of completed tasks with the different rounding techniques.

ParMod	**Δ % (MPC ↑)**	**Δ % (MPC ↓)**	**Δ % (MPC ∼)**
S_1	1.77	22.23	7.50
S_2	1.83	22.33	7.79
S_3	1.70	22.51	6.81
S_4	2.59	23.21	7.07
S_5	1.42	21.51	8.37
S_6	1.89	22.22	7.84
S_7	1.73	22.53	6.89
S_8	1.85	22.33	7.56

Table 9.3: Percentage of task loss with the different strategies.

expensive MAX strategy.

ParMod	**Mean n. of nodes**	**Mean n. of saved nodes**
S_1	13.76	2.64
S_2	13.37	2.63
S_3	3.76	12.24
S_4	3.69	12.31
S_5	14.57	1.43
S_6	5.52	10.48
S_7	3.70	12.30
S_8	7.37	8.63

Table 9.4: Number of saved nodes over the execution.

9.3 Second example: applying proper trade-offs between performance and resource utilization cost

In the previous experiment we have considered the notable case in which we desire to optimize the performance of each module with the minimum number of used resources. We can consider more general cases in which each application module performs different trade-offs between these two control objectives. In fact we can consider distributed applications characterized by specific parts of the computation graph that are executed on back-end HPC computing architectures whose resources, e.g. in terms of number of nodes for instance, are leased to the application execution directly of through a virtualized environment. As we have hinted in Chapter 6, this approach is typical of the emerging computational paradigms as Cloud Computing, in which the flexible on-demand access to a shared pool of configurable resources needs to be provisioned and released with the minimal management effort to interested users and their applications. Cloud computing enables the dynamic utilization of resources according to a *pay-per-use* basis, in which different pricing strategies and billing models can be applied by Cloud providers. For instance the users can pay from the moment the provision a set of resources to the moment they de-provision them. In this case it is irrelevant how much these resources are effectively used, but the users pay for the duration of their reservation.

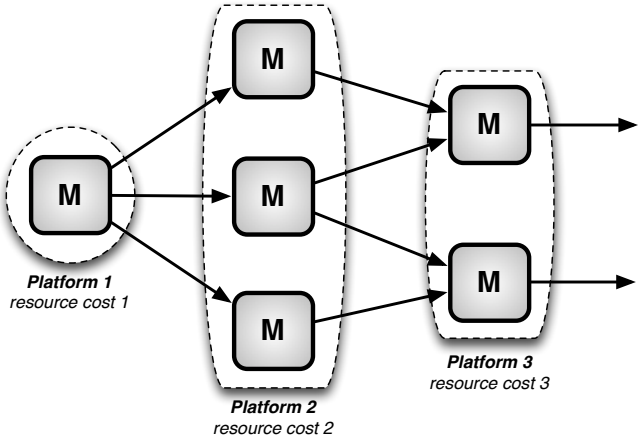


Figure 9.10: Example of application graph executed on remote computing architectures applying different resource costs.

In the above context it is extremely important to perform a proper analysis between resource usage and application performance. As depicted in Figure 9.10, a distributed parallel application can be executed on several distributed architectures managed by different providers applying distinct resource costs. Therefore it is important for the distributed control logic to be aware of the differences between the execution platforms, and explore the admissible application configurations in order to find the best trade-off between performance level and the resource usage.

In this section we present a second example in which we exploit different distributed control strategies applying proper trade-offs between control requirements. We will consider the very schematized application graph depicted in Figure 9.11, composed of three adaptive parallel modules executed on different distributed HPC architectures. This application is aimed at processing a large sequence of tasks generated by the first module (that applies a pre-processing computation), which transmits each task to one of the two modules $ParMod_2$ and $ParMod_3$ after a particular analysis of the task features. Parametrically each task is transmitted to $ParMod_2$ with probability p and to $ParMod_3$ with probability $1-p$.

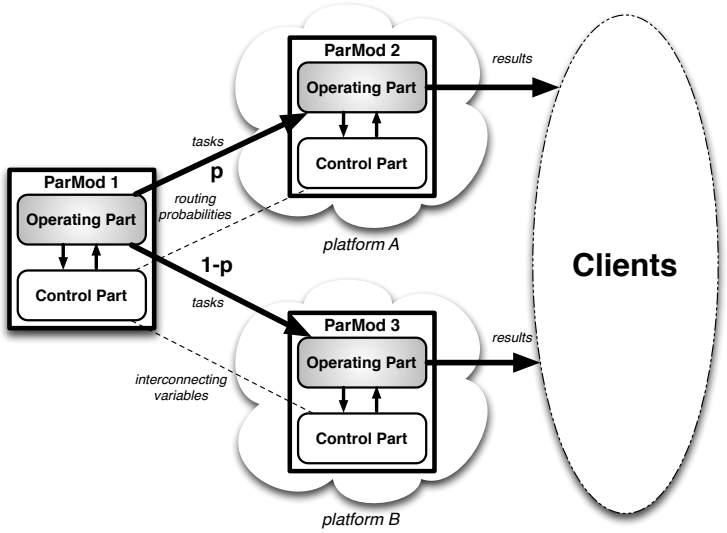


Figure 9.11: Computation graph of the second simulated application.

In this example we will suppose a different source of dynamicity of the application behavior than the previous example:

Assumption 9.3.1. *In this second example we will suppose that the probability p of transmission of tasks from $ParMod_1$ to $ParMod_2$ can change significantly during the execution.*

This means that, instead of being a constant term of the application behavior, there are execution phases in which a large amount of tasks are processed by $ParMod_2$ and other phases in which this module is rarely used. Therefore the probability p represents the disturbance input of the model. A growth in the probability p means that the mean inter-arrival time of tasks to $ParMod_2$ decreases, and that module, in order to effectively process this new arrival rate, may require to change its configuration (i.e. increasing its parallelism degree). In a similar way a decrease in the probability p consists in a higher inter-arrival time to the second module, that can release resources that are under-utilized at that moment. Of course the performance behavior of the three ParMods is strictly coupled: a growth in the probability p corresponds to a decrease of the arrival rate of tasks to $ParMod_3$. Moreover the effective inter-departure time of tasks from the source

module depends on the behavior of $ParMod_2$ and $ParMod_3$ (and in particular to what module is currently the application bottleneck, as we have seen in Section 4.1.1.2). In this example we suppose that the mean calculation time per task of each ParMod is a random variable with a low variance and a fixed mean value throughout the execution. Therefore the probability p is the only source of dynamicity in this experiment.

9.3.1 Selfish and Cooperative MPC strategies

Also in this experiment we consider a local cost function for each module defined as follows:

$$J_i(k) = \alpha_i T_{p_i}(k+1) + \beta_i n_i(k)$$

Differently from the previous example, each ParMod applies a different trade-off between performance and resource cost. In our simulation we have applied specific values for the model parameters summarized in Table 9.5. As said the three modules have a fixed mean

	ParMod 1	**ParMod 2**	**ParMod 3**
T_{calc}	15 sec.	30 sec.	40 sec.
α	4	4	4
β	0.6	2.5	1.5
n_i^*	10	6.93	10.33

Table 9.5: Configuration of model parameters for the second example.

calculation time over the execution. The three ParMods are executed on computing platforms featuring a maximum number of computing nodes equal to 10 for the first module and 20 for the other two. Although the parameter α is the same for all the application modules (thus they give the same importance to their effective performance level), each ParMod applies a different β term. In this experiment we suppose that the resource utilization is more expensive in the platform on which $ParMod_2$ is executed than on the other platforms. Moreover the first module $ParMod_1$ is responsible for transmitting tasks to the other two modules as fast as possible with the minimum number of necessary resources.

In this example we have studied the differences between two distributed adaptation strategies:

- a *communication-based MPC strategy*, in which each controller decides the parallelism degree based on its self-interest, i.e. optimizing its local cost function without taking into account the effects of this decision on the other modules;

- a *cooperative MPC strategy*, in which the application controllers cooperate in order to find a set of parallelism degrees such that the sum of the local cost functions is minimized. W.r.t the selfish approach, a controller can select a parallelism degree which is not optimal from its local viewpoint (by deviating from it the controller

decreases the value of its local cost function), but this choice can be beneficial for the other modules such that the sum of their objectives is optimized.

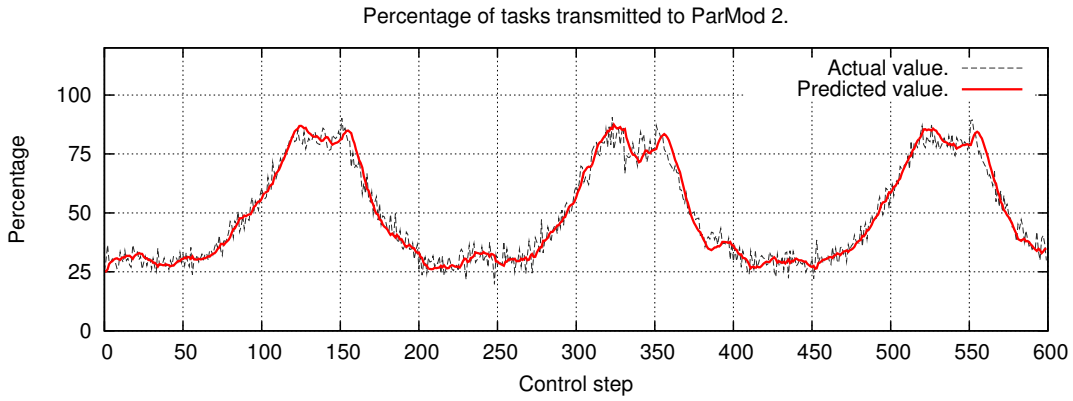


Figure 9.12: Probability $p(k)$ of transmission to ParMod 2 over the execution.

As we have seen in Chapter 8, the communication-based MPC strategy consists in selecting at each control step a set of parallelism degrees (a strategy profile) that corresponds to the efficient Nash equilibrium. Differently from the previous example, in which the efficient Nash equilibrium also corresponds to the social optimum profile, in this example the equilibrium profile can be far from the social optimum. For this reason in this experiment it is interesting to study the exploitation of a more cooperative approach based on the distributed subgradient method (see Section 8.3.1).

During the execution both the efficient Nash equilibrium and the social optimum strategy profile change due to a time-varying probability of task transmission to $ParMod_2$ and $ParMod_3$. We denote with $p(k)$ the probability measured during the k-th control step of the execution. In our simulation environment the values of this disturbance input are a transmitted from the operating part to the controller of $ParMod_1$ at the beginning of each control step. For this experiment we have reproduced a situation in which this probability (expressed in percentage) periodically fluctuates reaching peaks of over the 75%. We have considered a simulated execution of 600 control steps of length 240 sec. equal to an execution of 40 hours. In Figure 9.12 are depicted the values assumed by $p(k)$ for each control step. The probability is measured as the ratio between the number of tasks transmitted to $ParMod_2$ during a control step and the total number of tasks transmitted during that step.

Both for the communication-based MPC approach and for the cooperative MPC strategy, at the beginning of each control step $ParMod_1$ predicts the value assumed by the probability $p(k)$ for the current control step k. Also for this experiment we consider a prediction horizon of one single control step, and the predictions are exploited through the Holt-Winters predictive filter based on the history of past observations. In Figure 9.12 the measured values of the probability are indicated with a black dashed line, whereas the solid red line corresponds to the predicted values with the filter, that give a RMSRE error less than 10% over the execution. We can observe three execution phases: from control

step 50 to 200, from step 250 to 400 and from 450 to 600. In these phases the probability starts from a value near to 25% and grows reaching a peak higher than 75% and than slowly returns to the initial condition. This behavior is typical of seasonal workload variations in which periodically (e.g. during specific hours of the day) the number of requests to a specific part of the system increases or decreases due to the user behavior.

In this example we have compared the execution of the selfish strategy and of the cooperative strategy. In the selfish approach the three controllers perform Algorithm 5 that requires a partial interconnection between them: i.e. the first controller is directly interconnected with the second and the third controller while these two controllers are not directly interconnected. In order to propagate the effects of interconnecting variables, the protocol requires *two iterations* (which is also the diameter of the graph of Figure 9.11). The interconnecting variables exchanged by controllers are their mean service times $T_{S_i}(k)$ and in this case also the probability $p(k)$. The performance coupling relationships are summarized with expressions (8.2). These expressions model the direct coupling relationships between control sub-problems from the performance viewpoint. Through a proper number of information exchange (two iterations), the effects of the control decision taken by a controller influences all the other controllers of the graph and the steady-state mean inter-departure times can be correctly determined.

$$
\begin{aligned}
T_{p_1}(k+1) &= \max\left\{\frac{T_{calc-1}}{n_1(k)},\, p(k)\,T_{S_2}(k),\, \big(1-p(k)\big)\,T_{S_3}(k)\right\} \\
T_{p_2}(k+1) &= \max\left\{\frac{T_{calc-2}}{n_2(k)},\, \frac{T_{S_1}(k)}{p(k)}\right\} \\
T_{p_3}(k+1) &= \max\left\{\frac{T_{calc-3}}{n_3(k)},\, \frac{T_{S_1}(k)}{\big(1-p(k)\big)}\right\}
\end{aligned}
\tag{9.2}
$$

For the cooperative MPC strategy, each controller performs Algorithm 6 for a fixed number of iterations (their number will be discusses in the next section). At the beginning of the protocol $ParMod_1$ transmits to the other ParMods the prediction of the disturbance input $p(k)$. Then at each iteration the controllers exchange their estimates of the optimal strategy profile (a vector of parallelism degrees for each ParMod). Since the controllers are not completely interconnected ($ParMod_2$ and $ParMod_3$ are not directly interconnected), the weight matrix follows the symmetric rule (see Section 8.3.1) and is given by:

$$
W = \begin{pmatrix} 1/3 & 1/3 & 1/3 \\ 1/3 & 2/3 & 0 \\ 1/3 & 0 & 2/3 \end{pmatrix}
$$

We can note that $ParMod_2$ gives a zero weight to the estimate of $ParMod3$ and viceversa, because the two controllers are not directly interconnected thus they never receive the estimate of the other's optimal strategy profile. The distributed subgradient method is aimed at optimizing in a distributed fashion the sum of a set of convex objective functions.

In our case the three objectives are expressed as follows:

$$
\begin{aligned}
J_1(k) &= \alpha_1 \max\left\{\frac{T_{calc-1}}{n_1(k)}, p(k)\,\frac{T_{calc-2}}{n_2(k)}, \big(1-p(k)\big)\,\frac{T_{calc-3}}{n_3(k)}\right\} + \beta_1\, n_1(k) \\
J_2(k) &= \alpha_2 \max\left\{\frac{T_{calc-2}}{n_2(k)}, \frac{T_{calc-1}}{n_1(k)\,p(k)}, \frac{T_{calc-3}}{n_3(k)\,\big(1-p(k)\big)}\,p(k)\right\} + \beta_2\, n_2(k) \\
J_3(k) &= \alpha_3 \max\left\{\frac{T_{calc-3}}{n_3(k)}, \frac{T_{calc-1}}{n_1(k)\,\big(1-p(k)\big)}, \frac{T_{calc-2}}{n_2(k)\,p(k)}\,\big(1-p(k)\big)\right\} + \beta_3\, n_3(k)
\end{aligned}
\tag{9.3}
$$

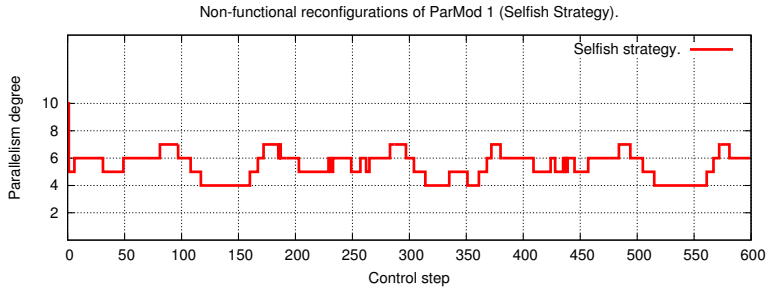
In this experiment the mean values of the calculation times of the three ParMods (i.e. T_{calc-1}, T_{calc-2} and T_{calc-3}) are assumed constant during the execution. The distributed subgradient method converges to an approximation of the social optimum strategy profile, which in this case corresponds to the set of parallelism degrees that minimizes the equally weighted sum of the three local cost functions, i.e. a function $J_G = J_1 + J_2 + J_3$, *which represents a measure of the global execution cost of the entire application graph*. As usual this modeling (both for the selfish and the cooperative case) considers a suboptimal continuous relaxation of the problem, thus the final parallelism degree is a real value that will be rounded to an integer value and passed to the corresponding operating part.

9.3.2 Results presentation and discussion

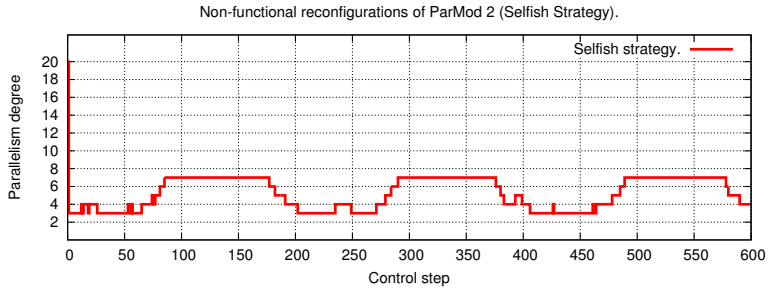
Let us consider the simulation results for this second example. In Figure 9.13 are depicted the sequence of parallelism degree variations of the three ParMods with the communication-based MPC strategy characterized by a selfish interaction between controllers. Each control part finds the parallelism degree that, in function of the currently received interconnecting variables, minimizes its local cost. Then, once the efficient Nash equilibrium has been found, each controller selects an integer approximation of its parallelism degree (in this example we suppose a performance-conservative rounding to the lowest upper integer).

From the figure we can note several interesting aspects. Each adaptive module starts the execution with the maximum parallelism degree. We can observe that the execution phases in which the probability $p(k)$ is higher correspond to a growth in the parallelism degree of $ParMod_2$, because it receives tasks with a lower inter-arrival time. These phases also correspond to a decrease in the parallelism degree of the third module, which receives tasks slowly and thus it can release under-utilized computing resources. The opposite behavior can be appreciated when the probability falls reaching minimum values near to 25%.

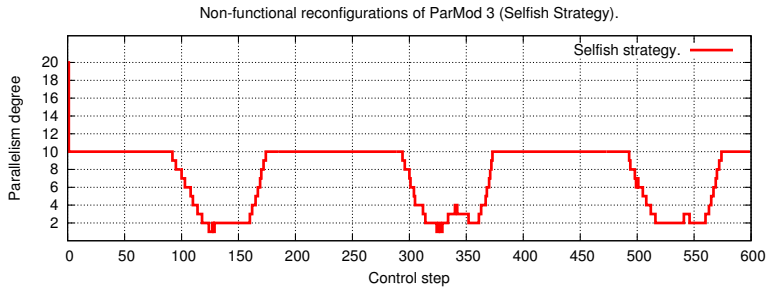
It is important to observe another peculiarity of the selfish approach. In Chapter 8 we have seen that the best response of a controller to the input interconnecting variables received by its neighbors, is a parallelism degree lower or at most equal to its ideal parallelism degree in isolation. *This means that in a selfish approach a controller never uses a parallelism degree greater than its ideal parallelism degree*. This fact is clearly evident



(a) Reconfigurations of ParMod 1.



(b) Reconfigurations of ParMod 2.



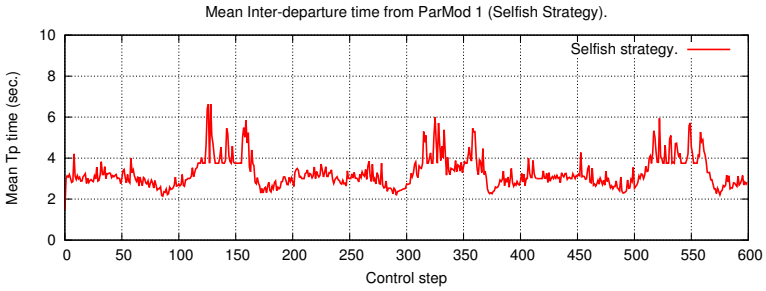
(c) Reconfigurations of ParMod 3.

Figure 9.13: Parallelism degree variations for each ParMod: Selfish strategy.

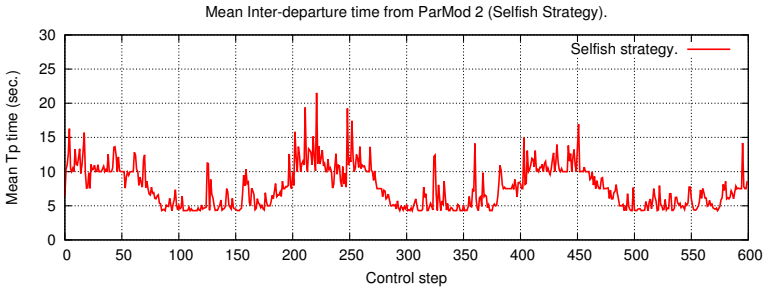
in Figure 9.13b and 9.13c. Controllers of ParMods 2 and 3 never use a parallelism degree greater than 7 and 11 respectively, which are the rounded up integer values of their ideal parallelism degrees (see Table 9.5).

With these parallelism degree variations, in Figure 9.14 is depicted the mean inter-departure time from the three modules for each control step of the execution. We can note that the effective performance behavior of the three modules adapts to the probability $p(k)$. Phases in which the second module is more stressed (p is higher as from control step 100 to 150) correspond to a lower inter-departure time from that module. In fact in these time intervals the mean inter-arrival time of tasks to $ParMod_2$ decreases and its controller responds increasing the parallelism degree. On the contrary, phases as from control step 200 to 250 correspond to a lower probability of transmission to the second ParMod, that responds by increasing its ideal service time and thus releasing computational resources.

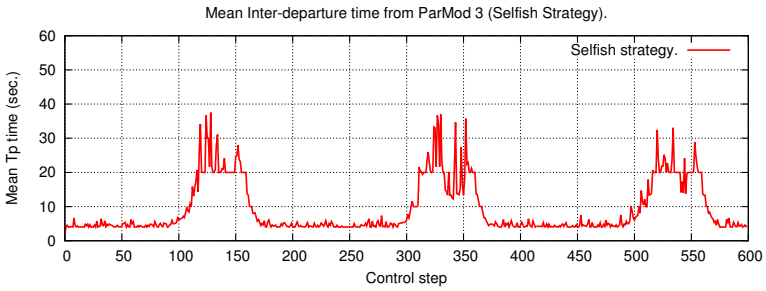
For the cooperative MPC strategy the controllers interact several times at each control



(a) Mean inter-departure time from ParMod 1.



(b) Mean inter-departure time from ParMod 2.



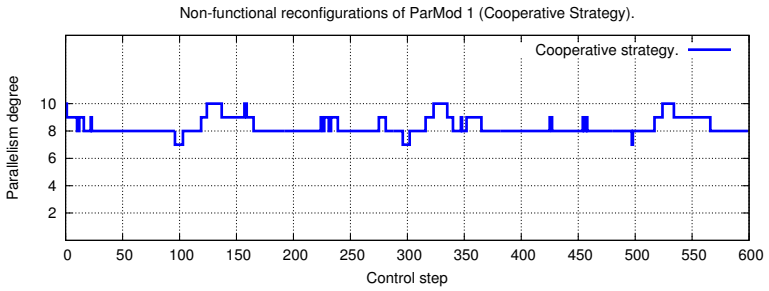
(c) Mean inter-departure time from ParMod 3.

Figure 9.14: Mean inter-departure time from each ParMod: Selfish strategy.

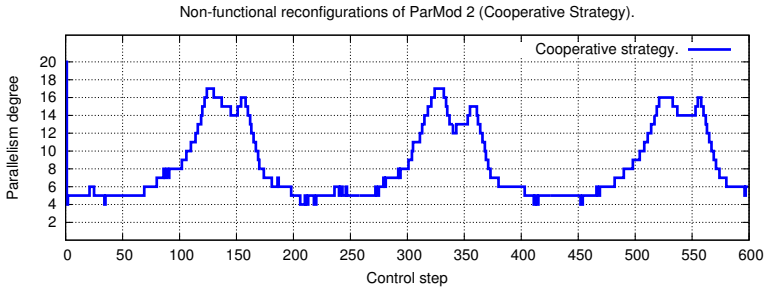
step in order to find an approximation of the social optimum strategy profile, and then they apply and integer rounding of the final value. Differently from the selfish strategy, each controller is no longer individually rational: i.e. it can take control decisions that are non-optimal for its local objective function, but they can be useful for the objectives of the other modules of the graph. This behavior is depicted in Figure 9.15, in which are shown the parallelism degree variations performed by the ParMods with the cooperative approach (also in this case exploiting a performance-conservative rounding).

The main difference w.r.t the selfish approach is that now the parallelism degree selected by each ParMod is no longer limited by its ideal parallelism degree n_i^*, but a controller can take a reconfiguration which is non-optimal from its local viewpoint. In fact we observe that $ParMod_2$ and $ParMod_3$ select parallelism degrees greater that 7 and 11 nodes respectively. Also in this case the execution phases in which the probability $p(k)$ assumes high values correspond to higher parallelism degrees of $ParMod_2$ and lower parallelism

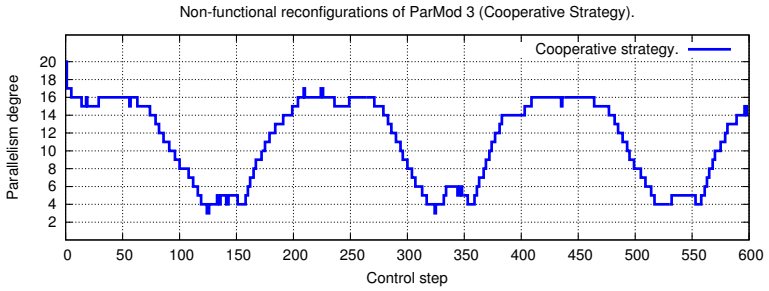
degrees for $ParMod_3$ and vice-versa. With the cooperative MPC strategy the mean inter-departure time of each ParMod is described in Figure 9.16, showing a fluctuating behavior that adapts to the actual value of the probability $p(k)$.



(a) Reconfigurations of ParMod 1.



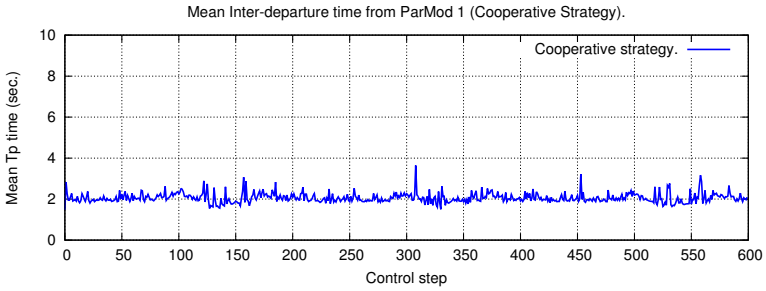
(b) Reconfigurations of ParMod 2.



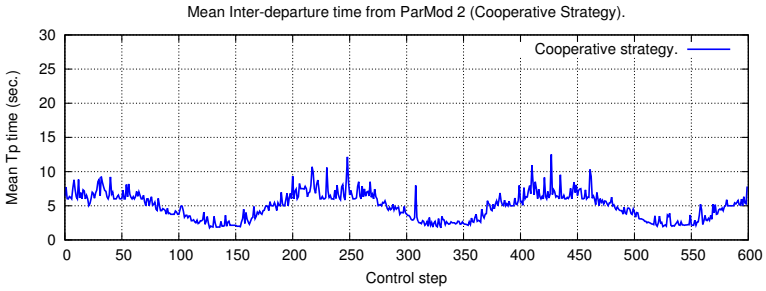
(c) Reconfigurations of ParMod 3.

Figure 9.15: Parallelism degree variations for each ParMod: Cooperative strategy.

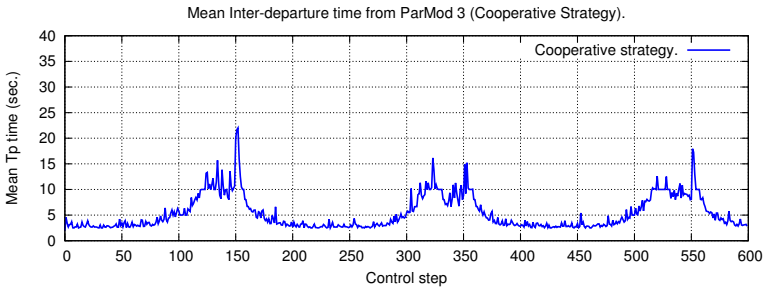
In is extremely useful to observe another general property of the two strategies. The first approach consists in finding at each control step the efficient Nash equilibrium and then select an integer-approximation of this strategy profile. In the second approach we exploit an iterative method such that, after a sufficient number of iterations, every controller reaches an approximation of the social optimum strategy profile, and then they select an integer approximation of their corresponding parallelism degree. In both the situations the efficient Nash equilibrium and the social optimum have a common property: *with these sets of real-valued parallelism degrees the utilization factor of each ParMod is equal to* 1 *(its resources are ideally utilized at best)*. In other words, if with a strategy profile some ParMods have a utilization factor less than 1, this profile can not be Pareto



(a) Mean inter-departure time from ParMod 1.



(b) Mean inter-departure time from ParMod 2.



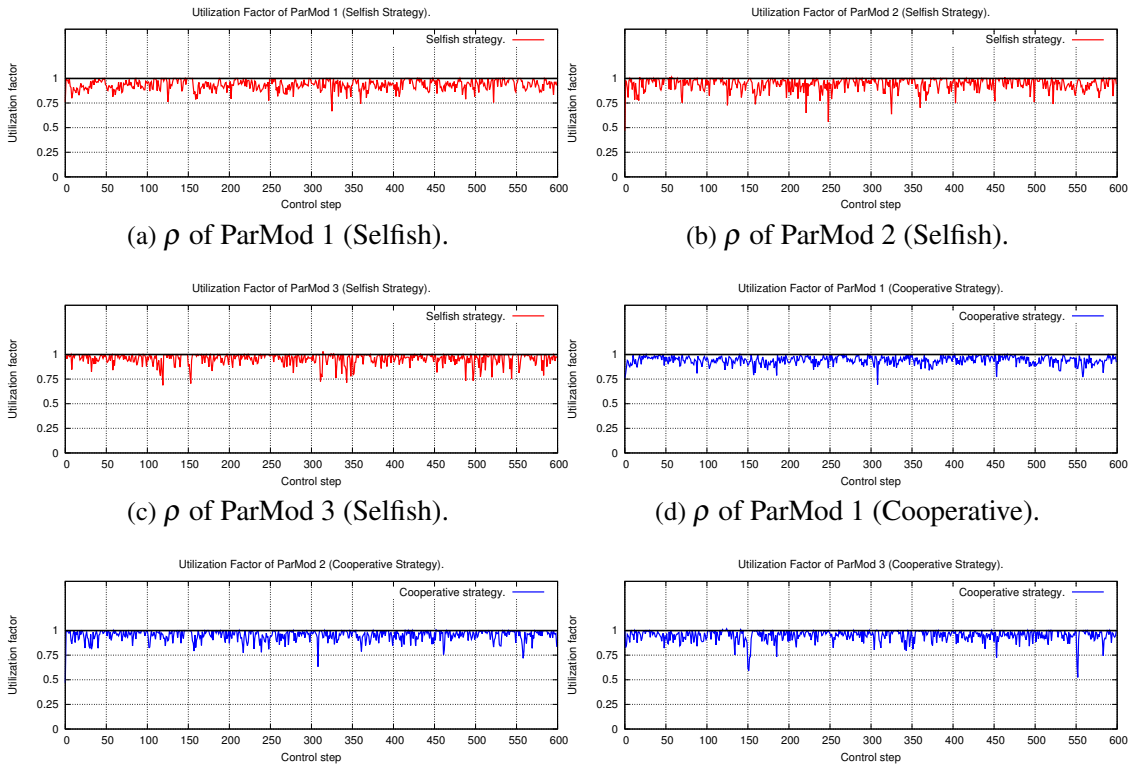
(c) Mean inter-departure time from ParMod 3.

Figure 9.16: Mean inter-departure time from each ParMod: Cooperative strategy.

optimal and thus it can not be neither the efficient Nash equilibrium nor the social optimum point. This concept is evident in Figure 9.17, in which are depicted the utilization factors assumed by the three ParMods. Figures 9.17a, 9.17b and 9.17c correspond to the communication-based MPC (red lines) while the other three figures (blue lines) are the utilization factors with the cooperative MPC. The utilization factors are always near the ideal one. As discussed for the previous example, it is not possible to reach exactly a utilization factor equal to 1 due to rounding and prediction errors.

The final set of experimental results is important for understanding the differences between the selfish and the cooperative approaches. In Figure 9.18 are represented the values assumed by the local cost functions J_1, J_2 and J_3 for each control step of the execution. The red lines correspond to the selfish strategy while the blue lines are the values of the local costs with the cooperative MPC approach.

For $ParMod_1$ the local cost is better with the cooperative approach. A similar behavior



(a) ρ of ParMod 1 (Selfish). (b) ρ of ParMod 2 (Selfish).

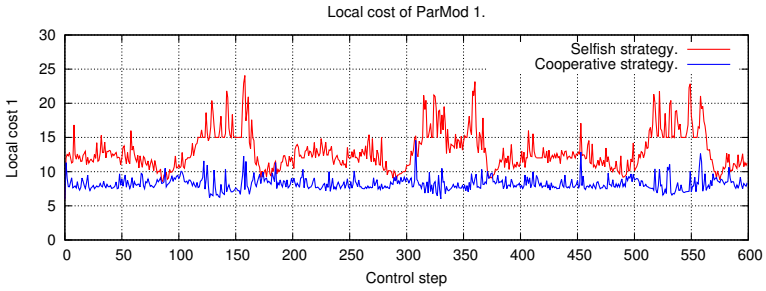
(c) ρ of ParMod 3 (Selfish). (d) ρ of ParMod 1 (Cooperative).

(e) ρ of ParMod 2 (Cooperative). (f) ρ of ParMod 3 (Cooperative).

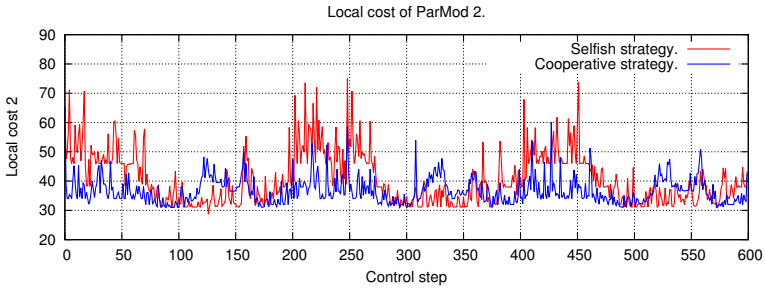
Figure 9.17: Utilization factor ρ for each ParMod with the different adaptation strategies.

is exhibited by the local cost of $ParMod_3$. For $ParMod_2$ the behavior of the local cost deserves a more careful examination. Also in this case there are execution phases in which the local cost assumes better values with the cooperative approach compared to the selfish strategy. For instance this behavior is evident from control step 200 to 250 and from 400 to 450. On the other hand for this ParMod the local cost with the cooperative approach is not always better than with the selfish strategy. In fact we can note execution phases in which with the selfish approach the local objective J_2 is better than with the cooperative technique. For instance this is true from control step 100 to 150 or from 300 to 350. *This aspect is the essence of cooperation*: with the cooperative approach there are execution phases in which $ParMod_2$ selects control decisions which are not driven by its self-interest (i.e. they are not locally optimal), but these decisions are taken because in this way the local objectives of the other modules can be improved such that the global cost can be globally optimized.

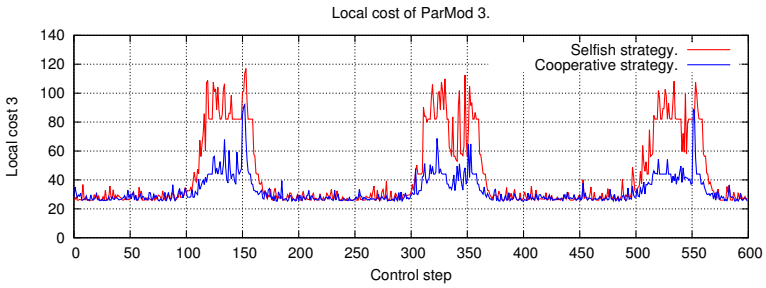
The behavior of the global cost function J_G is depicted in Figure 9.19. From the global application viewpoint the cooperative MPC is able to reach better results, minimizing the whole execution cost for the application graph. As said this objective requires to overcome the limitation of a pure self-interest interaction between controllers. The red line represents the global cost at each control step with the selfish strategy. As we can



(a) Local cost of ParMod 1.



(b) Local cost of ParMod 2.



(c) Local cost of ParMod 3.

Figure 9.18: Local costs for each module: comparison between Selfish and Cooperative strategies.

observe this cost is higher than the other lines corresponding to the cooperative strategy.

Figure 9.19 gives also further useful information about the feasibility of the cooperative approach. As we have seen in Chapter 8, this cooperative strategy is based on the distributed subgradient method. This approach ensures that the local estimates of each controller converges to an approximation of the social optimum strategy profile after a sufficiently large number of iterations. Although a large number of iterations is useful for improving the result accuracy, it may hamper the feasibility of this approach since the interaction protocol has to be completed at each sampling interval. In the literature the subgradient method, despite its flexibility to be applicable also for non-smooth cost functions, suffers from low convergence speed to reach a sufficiently accurate approximation of the optimal point. Since this method is executed at each control step, it is extremely important to complete the protocol with a reasonable number of iterations, in order to

maintain its completion time limited compared to the control step length.

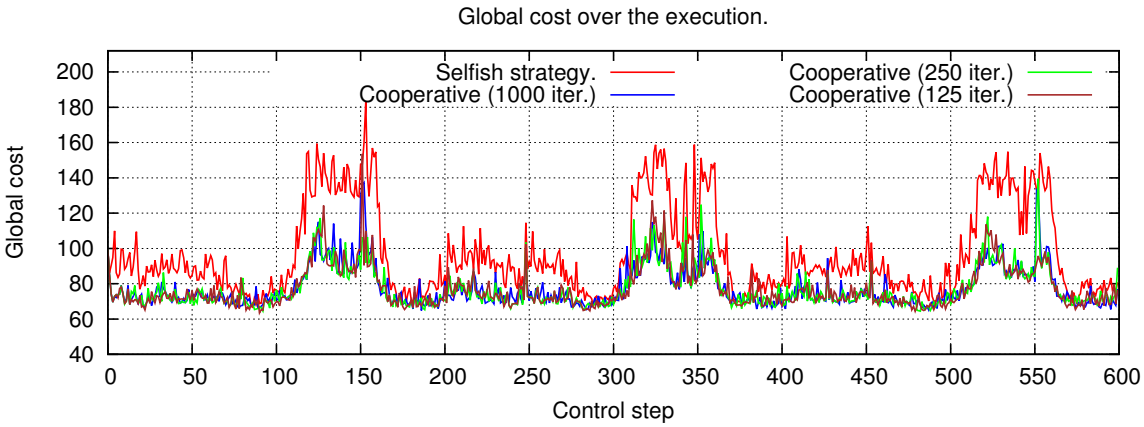


Figure 9.19: Global cost over the execution: comparison between Selfish and Cooperative strategies.

At this regard we have studied the behavior of the global cost performing the cooperative strategy with a constant stepsize equal to 0.005 and different maximum thresholds to the total number of iterations. In Figure 9.19 we have considered three thresholds. The blue line corresponds to the cooperative MPC in which controllers exchanges their local estimates 1000 times before stopping the interaction protocol. The green line considers 250 iterations and the brown line 125 iterations. From the figure we can observe that the global cost is nearly the same with the three different number of iterations. This behavior is confirmed by the results shown in Table 9.6. The table summarizes the sum of local costs and the global cost for each control step of the execution. We can observe that passing from 1000 to only 125 iterations does not make worse the total cost significantly. This is due to two different reasons:

	Selfish	**Cooperative(1000)**	**Cooperative(250)**	**Cooperative(125)**
Total J_1	7778	4939	4943	4928
Total J_2	24375	21990	22000	21977
Total J_3	25994	19490	19424	19566
Total J_G	58167	46419	46366	46471

Table 9.6: Operating costs (local and global) with selfish and cooperative strategies.

- the final control decision taken by each ParMod at the end of the interaction protocol is an integer approximation of the final parallelism degree. This means that a high level of precision in finding the real values of the parallelism degrees at the social optimum point is not required in this problem. This is a good case for the applicability of the subgradient method. As known in the literature, both for its slow convergence speed and for the difficulty to find accurate stopping criteria, the

subgradient approach is a valuable candidate in problems that do not require a great accuracy;

- the accuracy of the distributed subgradient method also depends on the initial conditions, i.e. the initial strategy profile that we use for starting the iterative protocol. In our modeling, at each control step of the execution some disturbance input variables can change their value. Due to this fact the social optimum strategy profile changes during the execution, due to the variability of the probability $p(k)$. For this reason in our approach we need to re-evaluate the social optimum strategy profile at each control step. Instead of starting the distributed subgradient method from a fixed initial point (that can become far from the social optimum point), a more effective approach consists in using as the starting point the final strategy profile determined at the last iteration of the previous control step (this approach is also called *warm-start* as hinted in Section 8.3.1.1). Since disturbance inputs change gradually step by step, the previous optimal point is a good starting point for reaching a good approximation of the new social optimum with a relatively limited number of iterations.

With the warm-start technique and a properly selected starting point for the first control step (chosen at design-time), this experiment shows that passing from 1000 to 125 iterations (thus reducing the iterations of one order of magnitude) does not substantially change the quality of the cooperative control. These considerations suggest the feasibility of this approach, since few tens of iterations are a reasonable number in many practical scenarios.

Finally, from Table 9.6 we can also analyze the effective improvement of the total cost over the execution. We can note that the total value of the three local costs J_1, J_2 and J_3 is lower with the cooperative approach than with the selfish protocol. In terms of total cost (i.e. the sum of the values assumed by J_G at each control step), we can note that the cooperative strategy (independently from the considered number of iterations) makes it possible to reduce this cost of about the 20% w.r.t the communication-based MPC approach, which implies an average Price of Stability of about 0.80 in this example. Moreover these results highlight the very slight difference between passing from 1000 iterations to 125 iterations, confirming the feasibility of the distributed subgradient method for this distributed control problem.

9.4 Summary

In this chapter we have concretely applied the distributed control modeling presented in Chapter 8 for controlling the performance and the resource utilization cost of distributed parallel applications. This chapter is extremely important because in these experiments we have applied the main concepts described in the thesis: the performance modeling of acyclic graphs introduced in Chapter 4, the operating part/control part structuring of

a ParMod presented in Chapter 5, and the distributed control model described in general terms in Chapter 7 and then applied in Chapter 8 to an example of performance and resource utilization control. The experiments show the importance of accounting the coupling relationships between control sub-problems, in order to apply an effective adaptation technique that formalizes controller interaction, negotiation and convergence of the control decisions. Finally we have discussed in detail the advantages and disadvantages of the self-interest and the cooperative strategies in terms of complexity as well as quality of the control.

Chapter 10

Conclusion

THis thesis faced with the problem of defining a methodology for controlling adaptive distributed parallel applications represented by generic graphs of computation modules, that cooperate exchanging sequence of tasks as in stream-based applications. The presented approach is based on two levels of parallelism of applications: (i) intra-module parallelism, i.e. how parallelism is expressed inside a single parallel module; (ii) inter-module parallelism: i.e. how computation modules are interconnected with each other. In this methodology we exploited the so-called *Structured Parallel Programming paradigm* for expressing intra-module parallelism through well-known parallelism schemes, as task-farm and data-parallel patterns that recur in the parallelization of many real-life algorithms and applications as clearly demonstrated by the existing scientific literature. For inter-module parallelism we considered the strictly necessary number of constraints to the structure of application graphs, that can feature acyclic interaction patterns or cyclic behavior and in classic client-server computations.

The first aim of this thesis was to develop a performance modeling methodology for determining the steady-state performance behavior of a computation graph starting from initial measurements of the ideal performance of each computation module. The methodology described in Chapter 4 consists in very interesting results from modeling the performance behavior of a quite general class of acyclic graphs, evaluating the steady-state behavior of each module (i.e. its mean inter-departure time) through algorithmic procedures and Queueing Networks theorems. For cyclic graphs we provided an approximated computation model for measuring the mean response time of a module acting as a server component w.r.t a set of clients transmitting requests and waiting for the corresponding results.

In Chapter 5 we presented the basic concept of *ParMod*, that is a parallel module performing structured parallel computations with an adaptive behavior. It behavior was formally described in terms of an Operating Part, that executes the computation, and a Control Part which is in charge of observing and controlling the computation behavior. According to the structured parallel programming approach, two classes of reconfigurations activities were described: (i) non-functional reconfigurations consist in geometrical

and structural changes of the component behavior; (ii) functional reconfigurations are implementation changes of the operating part behavior that modify the currently parallelized algorithm and the performed parallelization scheme.

In this thesis we developed different control techniques for a generic ParMod. A reactive approach was presented in Chapter 5, in which the control logic was formalized as a finite-state machine which interacts with a *Hybrid System* (i.e. the Operating Part) that exploits different, alternative operating modes (configurations) that can be changed during the execution. The main limitations of a purely reactive technique, especially in terms of the degree of stability of a system operating mode, were discussed providing the motivations for studying alternative approaches. The thesis presented the application of a control-theoretic methodology for controlling an adaptive parallel module. The presented approach inherits from the so-called *Model-based Predictive Control* technique (MPC), known to control theorists as a valid though suboptimal control approach that provides good results in practical situations. This strategy consists in a finite-horizon optimization of a system objective function, taking proper statistical predictions of measurements considered as disturbance inputs, and exploiting the QoS predictability of structured parallelism schemes through the definition of a *system model*. In Chapter 6 these control strategies were tested and compared on two real applications.

The second part of the thesis addressed the problem of controlling multiple ParMods interconnected in generic computation graph structures. This problem is critical, and its formalization and the development of effective distributed control techniques is a valuable contribution of this thesis. In the existing literature of *Control Theory*, the problem of distributing the control logic of complex systems has been intensively studied over the last years, providing general modeling techniques and precise solutions to specific problems and applications. A main contribution of this thesis was to clearly organize the different modelings of distributed control (e.g. centralized schemes versus distributed or decentralized solutions as well as multi-layer approaches). In Chapter 7 this knowledge was a starting point to define a distributed MPC strategy for controlling graphs of ParMods whose extended model accounts for controller interactions and coupling relationships between control sub-problems, i.e. when control decisions taken by a controller influence the local objectives of others.

In Chapter 8 this model was applied to a notable case in which we need to optimize the performance behavior and the resource utilization cost of the execution of a distributed parallel application. This problem recurs in many scientific research areas, from Grids to Clouds and in general in multi-agent optimizations of distributed computation environments as Mobile Grids and Wireless Sensor Networks. In this chapter we formalized this control problem providing different methodology for orchestrating the interactions between controllers. A first approach is based on controllers pursuing their self-interest. In this case we described this interaction through basic notions of *Game Theory*, in which the MPC strategy consists in controllers that optimize their local objective functions exchanging control information and reaching proper agreements in their decisions that correspond to the notion of Nash equilibrium. A second MPC strategy starts from a different standpoint in which controllers need to perform a more extensive cooperation in order

to improve the global quality of the control strategy. In Chapter 9 we analyzed these different techniques through a simulation environment in order to present in a clear way the limitations and advantages of a self-interest and a cooperative interaction between ParMod controllers of a computation graph.

Bibliography

[1] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 123–169, 1998.

[2] F. Berman, G. Fox, and A. J. G. Hey, *Grid Computing: Making the Global Infrastructure a Reality*. New York, NY, USA: John Wiley & Sons, Inc., 2003.

[3] I. Foster and C. Kesselman, *The Grid 2: Blueprint for a New Computing Infrastructure*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003.

[4] B. Hayes, "Cloud computing," *Commun. ACM*, vol. 51, no. 7, pp. 9–11, 2008.

[5] T. Priol and M. Vanneschi, *Towards Next Generation Grids: Proceedings of the CoreGRID Symposium 2007*. Springer Publishing Company, Incorporated, 2007.

[6] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[7] J. Cao, A. Chan, Y. Sun, and K. Zhang, "Dynamic configuration management in a graph-oriented distributed programming environment," *Sci. Comput. Program.*, vol. 48, no. 1, pp. 43–65, 2003.

[8] J. Kramer and J. Magee, "Dynamic configuration for distributed systems," *IEEE Trans. Softw. Eng.*, vol. 11, no. 4, pp. 424–436, 1985.

[9] N. Arshad, D. Heimbigner, and A. L. Wolf, "Deployment and dynamic reconfiguration planning for distributed software systems," *Software Quality Control*, vol. 15, no. 3, pp. 265–281, 2007.

[10] J. Hillman and I. Warren, "An open framework for dynamic reconfiguration," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, (Washington, DC, USA), pp. 594–603, IEEE Computer Society, 2004.

[11] A. T. A. Gomes, T. V. Batista, A. Joolia, and G. Coulson, "Architecting dynamic reconfiguration in dependable systems," pp. 237–261, 2007.

[12] W. T. Tsai, W. Song, Y. Chen, and R. Paul, "Dynamic system reconfiguration via service composition for dependable computing," in *Proceedings of the 12th Monterey conference on Reliable systems on unreliable networked platforms*, (Berlin, Heidelberg), pp. 203–224, Springer-Verlag, 2007.

[13] M. Vanneschi and L. Veraldi, "Dynamicity in distributed applications: issues, problems and the assist approach," *Parallel Comput.*, vol. 33, no. 12, pp. 822–845, 2007.

[14] M.-C. Pellegrini and M. Riveill, "Component management in a dynamic architecture," *J. Supercomput.*, vol. 24, no. 2, pp. 151–159, 2003.

[15] K. Moazami-Goudarzi and J. Kramer, "Maintaining node consistency in the face of dynamic change," in *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, (Washington, DC, USA), p. 62, IEEE Computer Society, 1996.

[16] I. Warren and I. Sommerville, "A model for dynamic configuration which preserves application integrity," in *ICCDS '96: Proceedings of the 3rd International Conference on Configurable Distributed Systems*, (Washington, DC, USA), p. 81, IEEE Computer Society, 1996.

[17] M. Cole, "Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming," *Parallel Comput.*, vol. 30, no. 3, pp. 389–406, 2004.

[18] M. Vanneschi, "The programming model of assist, an environment for parallel and distributed portable applications," *Parallel Comput.*, vol. 28, no. 12, pp. 1709–1732, 2002.

[19] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi, "P3l: A structured high-level parallel language, and its structured support," *Concurrency: Practice and Experience*, vol. 7, no. 3, pp. 225–255, 1995.

[20] Y. Karasawa and H. Iwasaki, "A parallel skeleton library for multi-core clusters," in *Proceedings of the 2009 International Conference on Parallel Processing*, ICPP '09, (Washington, DC, USA), pp. 84–91, IEEE Computer Society, 2009.

[21] H. Kuchen and J. Striegnitz, "Features from functional programming for a c++ skeleton library: Research articles," *Concurr. Comput. : Pract. Exper.*, vol. 17, pp. 739–756, June 2005.

[22] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, 2008.

[23] M. Aldinucci, M. Danelutto, and M. Vanneschi, "Autonomic qos in assist grid-aware components," in *PDP '06: Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, (Washington, DC, USA), pp. 221–230, IEEE Computer Society, 2006.

[24] M. Aldinucci, S. Campa, M. Danelutto, and M. Vanneschi, "Behavioural skeletons in gcm: Autonomic management of grid components," in *Parallel, Distributed and Network-Based Processing, 2008. PDP 2008. 16th Euromicro Conference on*, pp. 54–63, Feb. 2008.

[25] M. Parashar, H. Liu, Z. Li, V. Matossian, C. Schmidt, G. Zhang, and S. Hariri, "Automate: Enabling autonomic applications on the grid," *Cluster Computing*, vol. 9, pp. 161–174, April 2006.

[26] G. Wrzesinska, J. Maassen, and H. E. Bal, "Self-adaptive applications on the grid," in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, (New York, NY, USA), pp. 121–129, ACM, 2007.

[27] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi, "Expressing adaptivity and context-awareness in the assistant programming model," in *Proceedings of the Third International ICST Conference on Autonomic Computing and Communication Systems*, pp. 38–54, September 2009.

[28] C. Bertolli, D. Buono, S. Lametti, G. Mencagli, M. Meneghin, A. Pascucci, and M. Vanneschi, "A programming model for high-performance adaptive applications on pervasive mobile grids," in *Proceeding of the 21st IASTED International Conference on Parallel and Distributed Computing and Systems*, pp. 38–54, November 2009.

[29] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury, *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[30] M. M. Kokar, K. Baclawski, and Y. A. Eracar, "Control theory-based foundations of self-controlling software," *IEEE Intelligent Systems*, vol. 14, pp. 37–45, May 1999.

[31] J. Kephart and W. Walsh, "An artificial intelligence perspective on autonomic computing policies," in *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pp. 3–12, June 2004.

[32] M. Morandini, L. Penserini, and A. Perini, "Towards goal-oriented development of self-adaptive systems," in *Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems*, SEAMS '08, (New York, NY, USA), pp. 9–16, ACM, 2008.

[33] B. Noble, "System support for mobile, adaptive applications," *Personal Communications, IEEE*, vol. 7, pp. 44–49, Feb 2000.

[34] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project aura: Toward distraction-free pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22–31, 2002.

[35] J.-W. Chang, J. H. Kim, and Y.-K. Kim, "Design of overall architecture supporting context-aware application services in pervasive computing," in *ESA* (H. R. Arabnia and L. T. Yang, eds.), pp. 216–221, CSREA Press, 2007.

[36] H. Chen, T. Finin, and A. Joshi, "An intelligent broker for context-aware systems," *Adjunct Proceedings of Ubicomp 2003*, pp. 183–184, October 2003. poster paper.

[37] I. Horrocks, P. Patel-Schneider, and F. van Harmelen, "From shiq and rdf to owl: The making of a web ontology language," *Journal of Web Semantics*, vol. 1, no. 1, pp. 7–26, 2003.

[38] A. K. Dey, "Understanding and using context," *Personal Ubiquitous Comput.*, vol. 5, no. 1, pp. 4–7, 2001.

[39] E. J. Y. Wei and A. T. S. Chan, "Towards context-awareness in ubiquitous computing," in *EUC'07: Proceedings of the 2007 international conference on Embedded and ubiquitous computing*, (Berlin, Heidelberg), pp. 706–717, Springer-Verlag, 2007.

[40] M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference*. Cambridge, MA, USA: MIT Press, 1995.

[41] E. Lusk, "Mpi in 2002: has it been ten years already?," p. 435, 2002.

[42] M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo, "Execution support of high performance heterogeneous component-based applications on the grid," in *Euro-Par'06: Proceedings of the CoreGRID 2006, UNICORE Summit 2006, Petascale Computational Biology and Bioinformatics conference on Parallel processing*, (Berlin, Heidelberg), pp. 171–185, Springer-Verlag, 2007.

[43] M. Aldinucci, M. Coppola, M. Danelutto, M. Vanneschi, and C. Zoccolo, "Assist as a research framework for high-performance grid programming environments," in *Grid Computing: Software environments and Tools*, pp. 230–256, Springer, 2005.

[44] M. Coppola, M. Danelutto, N. Tonellotto, M. Vanneschi, and C. Zoccolo, "Execution support of high performance heterogeneous component-based applications on the grid," in *Euro-Par'06: Proceedings of the CoreGRID 2006, UNICORE Summit 2006, Petascale Computational Biology and Bioinformatics conference on Parallel processing*, (Berlin, Heidelberg), pp. 171–185, Springer-Verlag, 2007.

[45] M. Danelutto, M. Vanneschi, and C. Zoccolo, "A performance model for stream-based computations," in *PDP '07: Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, (Washington, DC, USA), pp. 91–96, IEEE Computer Society, 2007.

[46] S. Ahumada, L. Apvrille, T. Barros, A. Cansado, E. Madelaine, and E. Salageanu, "Specifying fractal and gcm components with uml," in *SCCC '07: Proceedings of the XXVI International Conference of the Chilean Society of Computer Science*, (Washington, DC, USA), pp. 53–62, IEEE Computer Society, 2007.

[47] E. Mathias, F. Baude, and V. Cave, "A gcm-based runtime support for parallel grid applications," in *CBHPC '08: Proceedings of the 2008 compFrame/HPC-GECO workshop on Component based high performance*, (New York, NY, USA), pp. 1–10, ACM, 2008.

[48] P. Browne, *JBoss Drools Business Rules*. Packt Publishing, 2009.

[49] M. Aldinucci, M. Danelutto, and P. Kilpatrick, "Autonomic management of non-functional concerns in distributed & parallel application programming," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, (Washington, DC, USA), pp. 1–12, IEEE Computer Society, 2009.

[50] D. J. Lillethun, D. Hilley, S. Horrigan, and U. Ramachandran, "Mb++: An integrated architecture for pervasive computing and high-performance computing," in *RTCSA '07: Proceedings of the 13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, (Washington, DC, USA), pp. 241–248, IEEE Computer Society, 2007.

[51] U. Hansmann, L. Merk, M. S. Nicklous, and T. Stober, *Pervasive Computing : The Mobile World (Springer Professional Computing)*. Springer, August 2003.

[52] T. Priol and M. Vanneschi, *From Grids To Service and Pervasive Computing*. Springer Publishing Company, Incorporated, 2008.

[53] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative system performance: computer system analysis using queueing network models*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1984.

[54] J. Darlington, Y.-k. Guo, H. W. To, and J. Yang, "Parallel skeletons for structured composition," in *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 19–28, ACM, 1995.

[55] M. Vanneschi, "Heterogeneous hpc environments," in *Euro-Par '98: Proceedings of the 4th International Euro-Par Conference on Parallel Processing*, (London, UK), pp. 21–34, Springer-Verlag, 1998.

[56] D. B. Skillicorn and D. Talia, "Models and languages for parallel computation," *ACM Comput. Surv.*, vol. 30, no. 2, pp. 123–169, 1998.

[57] C. Bertolli, *Fault tolerance for High-Performance applications using structured parallelism models*. Saarbrcken, Germany: VDM Verlag, 2009.

[58] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi, "A heterogeneous environment for hpc applications," *Parallel Computing*, vol. 25, no. 13-14, pp. 1827–1852, 1999.

[59] C. Bertolli, D. Buono, G. Mencagli, and M. Vanneschi, *An approach to Mobile Grid platforms for the development and support of complex ubiquitous applications*, vol. 1 of *Handbook of Research on Mobility and Computing: Evolving Technologies and Ubiquitous Impacts*. Portugal: IGI Global Publisher.

[60] C. Bertolli, G. Mencagli, and M. Vanneschi, *High-Performance Pervasive Computing*, vol. 1 of *Computer Science, Technology and Applications*. Nova Publisher.

[61] C. Bertolli, G. Mencagli, and M. Vanneschi, "A cost model for autonomic reconfigurations in high-performance pervasive applications," in *Proceedings of the 4th ACM International Workshop on Context-Awareness for Self-Managing Systems*, CASEMANS '10, (New York, NY, USA), pp. 3:20–3:29, ACM, 2010.

[62] E. Rutten, "Reactive control of adaptive embedded systems: a position paper," in *Proceedings of the 7th workshop on Reflective and adaptive middleware*, ARM '08, (New York, NY, USA), pp. 47–48, ACM, 2008.

[63] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003.

[64] N. Bassiliades, I. Vlahavas, and A. K. Elmagarmid, "E-device: An extensible active knowledge base system with multiple rule type support," *IEEE Trans. on Knowl. and Data Eng.*, vol. 12, no. 5, pp. 824–844, 2000.

[65] T. Heimrich and G. Specht, "Enhancing eca rules for distributed active database systems," in *Revised Papers from the NODe 2002 Web and Database-Related Workshops on Web, Web-Services, and Database Systems*, (London, UK), pp. 199–205, Springer-Verlag, 2003.

[66] C. Shankar and R. Campbell, "A policy-based management framework for pervasive systems using axiomatized rule-actions," in *Proceedings of the Fourth IEEE International Symposium on Network Computing and Applications*, (Washington, DC, USA), pp. 255–258, IEEE Computer Society, 2005.

[67] S. Reiff-Marganiec and K. J. Turner, "Feature interaction in policies," *Comput. Netw.*, vol. 45, pp. 569–584, August 2004.

[68] R. Srikant, *The Mathematics of Internet Congestion Control (Systems and Control: Foundations and Applications)*. SpringerVerlag, 2004.

[69] S. Keshav, "A control-theoretic approach to flow control," *SIGCOMM Comput. Commun. Rev.*, vol. 21, pp. 3–15, August 1991.

[70] Y. Mitrophanov and V. Dolgov, "Dynamic control of service rates in queueing networks," *Automatic Control and Computer Sciences*, vol. 42, pp. 311–319, 2008. 10.3103/S0146411608060060.

[71] R. Mortier and E. Kiciman, "Autonomic network management: some pragmatic considerations," in *Proceedings of the 2006 SIGCOMM workshop on Internet network management*, INM '06, (New York, NY, USA), pp. 89–93, ACM, 2006.

[72] Y. Diao, N. Gandhi, J. Hellerstein, S. Parekh, and D. Tilbury, "Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server," in *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pp. 219 – 234, 2002.

[73] S. Parekh, N. Gandhi, J. Hellerstein, D. Tilbury, T. Jayram, and J. Bigus, "Using control theory to achieve service level objectives in performance management," *Real-Time Syst.*, vol. 23, pp. 127–141, July 2002.

[74] J. W. Cangussu, K. Cooper, and C. Li, "A control theory based framework for dynamic adaptable systems," in *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, (New York, NY, USA), pp. 1546–1553, ACM, 2004.

[75] J. Love, *Process Automation Handbook: A Guide to Theory and Practice*. Springer Publishing Company, Incorporated, 1st ed., 2007.

[76] L. Ljung, *System identification: theory for the user*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1986.

[77] G. F. Franklin, *Feedback Control of Dynamic Systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2nd ed., 1993.

[78] J. Mincer-Daszkiewicz, "Program i/o behavior: Models and their applications," *Journal of Systems and Software*, vol. 14, no. 1, pp. 51–62, 1991.

[79] L. Kerbache and J. M. Smith, "Queueing networks and the topological design of supply chain systems," *International Journal of Production Economics*, vol. 91, no. 3, pp. 251–272, 2004.

[80] M. Hassan, R. Egudo, and J. Breen, "A closed queueing network model for retransmission based transport protocols over cell-switching networks," in *Communications, Computers, and Signal Processing, 1995. Proceedings. IEEE Pacific Rim Conference on*, pp. 86 –89, may 1995.

[81] H. G. Perros, *Queueing networks with blocking*. New York, NY, USA: Oxford University Press, Inc., 1994.

[82] D. of electronics and computer science polytechnic of Milan, "Java Modelling Tools." `http://jmt.sourceforge.net/`, 2011.

[83] *Introduction to algorithms*. Cambridge, MA, USA: MIT Press, 2nd ed., 2001.

[84] L. Kleinrock, *Theory, Volume 1, Queueing Systems*. Wiley-Interscience, 1975.

[85] M. Vanneschi, "High performance computing systems and enabling platforms." Course Notes of the Master Program in Computer Science and Networking, University of Pisa, jan 2011.

[86] I. D. Landau and G. Zito, *Digital Control Systems: Design, Identification and Implementation (Communications and Control Engineering)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[87] J. H. Sandee, W. P. M. H. Heemels, and P. P. J. Van Den Bosch, "Case studies in event-driven control," in *Proceedings of the 10th international conference on Hybrid systems: computation and control*, HSCC'07, (Berlin, Heidelberg), pp. 762–765, Springer-Verlag, 2007.

[88] C. Bertolli, G. Mencagli, and M. Vanneschi, "Consistent reconfiguration protocols for adaptive high-performance applications," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2011 7th International*, pp. 2121 –2126, july 2011.

[89] C. Bertolli, G. Mencagli, and M. Vanneschi, "Analyzing memory requirements for pervasive grid applications," *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, vol. 0, pp. 297–301, 2010.

[90] S. A. J. v. der and J. M. Schumacher, *Introduction to Hybrid Dynamical Systems*. London, UK: Springer-Verlag, 1999.

[91] D. Liberzon, *Switching in Systems and Control*. Boston, USA: Birkhuser Boston Production, 2003.

[92] A. Ames and S. Sastry, "Characterization of zeno behavior in hybrid systems using homological methods," in *American Control Conference, 2005. Proceedings of the 2005*, pp. 1160 – 1165 vol. 2, June 2005.

[93] A. Lamperski and A. D. Ames, "Sufficient conditions for zeno behavior in lagrangian hybrid systems," in *Proceedings of the 11th international workshop on Hybrid Systems: Computation and Control*, HSCC '08, (Berlin, Heidelberg), pp. 622–625, Springer-Verlag, 2008.

[94] L. Aceto, A. Inglfsdttir, K. Guldstrand, and J. Srba, *Reactive Systems: Modelling, Specification and Verification*. Cambridge, UK: Cambidge University Press, 2007.

[95] C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[96] S. J. A., A. P. J., and L. M. D., "A logical des approach to the design of hybrid control systems," *Mathematical and Computer Modelling*, vol. 23, no. 11-12, pp. 55 – 76, 1996.

[97] J. E. Hopcroft and J. D. Ullman, *Introduction To Automata Theory, Languages, And Computation*. USA: Instrument Society of America, 1st ed., 1990.

[98] D. P. Bertsekas, *Dynamic Programming and Optimal Control, Two Volume Set*. Athena Scientific, 2nd ed., 1995.

[99] C. E. Garcia, D. M. Prett, and M. Morari, "Model predictive control: theory and practice a survey," *Automatica*, vol. 25, pp. 335–348, May 1989.

[100] J. A. Rossiter, *Model-based predictive control: a practical approach*. Control series, pub-CRC:adr: CRC Press, 2003.

[101] D. Kusic and N. Kandasamy, "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems," in *Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, (Washington, DC, USA), pp. 74–83, IEEE Computer Society, 2006.

[102] E. Mestan, M. Türkay, and Y. Arkun, "Optimization of operations in supply chain systems using hybrid systems approach and model predictive control," *Industrial & Engineering Chemistry Research*, vol. 45, no. 19, pp. 6493–6503, 2006.

[103] J. Wu and S. Abdelwahed, "Discrete-input receding horizon control applied to pneumatic hopping robot energy regulation," in *Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on*, pp. 1567 –1572, Oct. 2007.

[104] L. Zhang and S. Quan, "Model predictive control of nonlinear hybrid system based on neural network optimization," in *Asian Control Conference, 2009. ASCC 2009. 7th*, pp. 1097 –1102, Aug. 2009.

[105] M. Mahfouf, S. Kandiah, and D. A. Linkens, "Fuzzy model-based predictive control using an arx structure with feedforward," *Fuzzy Sets Syst.*, vol. 125, pp. 39–59, January 2002.

[106] N. Li, S.-Y. Li, and Y.-G. Xi, "Multi-model predictive control based on the takagi-sugeno fuzzy models: a case study," *Inf. Sci. Inf. Comput. Sci.*, vol. 165, pp. 247–263, October 2004.

[107] S. Abdelwahed, N. Kandasamy, and S. Neema, "Online control for self-management in computing systems," in *Real-Time and Embedded Technology and Applications Symposium, 2004. Proceedings. RTAS 2004. 10th IEEE*, pp. 368 – 375, may 2004.

[108] N. Kandasamy, S. Abdelwahed, and J. Hayes, "Self-optimization in computer systems via on-line control: application to power management," in *Autonomic Computing, 2004. Proceedings. International Conference on*, pp. 54 – 61, may 2004.

[109] V. Bhat, M. Parashar, H. Liu, M. Khandekar, N. Kandasamy, and S. Abdelwahed, "Enabling self-managing applications using model-based online control strategies," in *Autonomic Computing, 2006. ICAC '06. IEEE International Conference on*, pp. 15 – 24, June 2006.

[110] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization: algorithms and complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1982.

[111] J. M. Sousa, R. Babuska, and H. B. Verbruggen, "Fuzzy predictive control applied to an air-conditioning system," *Control Engineering Practice*, vol. 5, no. 10, pp. 1395–1400, 1997.

[112] A. Ichtev, J. Hellendoom, R. Babuska, and S. Mollov, "Fault-tolerant model-based predictive control using multiple takagi-sugeno fuzzy models," in *Fuzzy Systems, 2002. FUZZ-IEEE'02. Proceedings of the 2002 IEEE International Conference on*, vol. 1, pp. 346 –351, 2002.

[113] B. Potočnik, G. Mušič, I. Škrjanc, and B. Zupančič, "Model-based predictive control of hybrid systems: A probabilistic neural-network approach to real-time control," *J. Intell. Robotics Syst.*, vol. 51, pp. 45–63, January 2008.

[114] S. Lucco, "A dynamic scheduling method for irregular parallel programs," in *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, (New York, NY, USA), pp. 200–211, ACM, 1992.

[115] R. Vilalta, C. V. Apte, J. L. Hellerstein, S. Ma, and S. M. Weiss, "Predictive algorithms in the management of computer systems," *IBM Syst. J.*, vol. 41, pp. 461–474, July 2002.

[116] A. Charteris, W. Syme, and W. Walden, "Urban flood modelling and mapping 2d or not 2d," in *Proceedings of the 6th Conference on Hydraulics in Civil Engineering: The State of Hydraulics*, (Barton, Australia), pp. 355–363, Barton A.C.T: Institution of Engineers, 2001.

[117] I. Duff and H. van der Vorst, "Developments and trends in the parallel solution of linear systems," *Par. Comp.*, vol. 25, no. 13-14, pp. 1931–1970, 1999.

[118] O. Kremien, "Buying and selling computational power over the network," in *Computer Communications and Networks, 1995. Proceedings., Fourth International Conference on*, pp. 616 –619, sep 1995.

[119] Q. He, C. Dovrolis, and M. Ammar, "On the predictability of large transfer tcp throughput," *Comput. Netw.*, vol. 51, pp. 3959–3977, October 2007.

[120] NCTUNS, "Network Simulator and Emulator." `http://nsl.csie.nctu.edu.tw/nctuns.html/`, 2011.

[121] D-ITG, "Distributed Internet Traffic Generator." `http://www.grid.unina.it/software/ITG/papers.php`.

[122] M. Kim and B. Noble, "Mobile network estimation," in *Proceedings of the 7th annual international conference on Mobile computing and networking*, MobiCom '01, (New York, NY, USA), pp. 298–309, ACM, 2001.

[123] E. Goldoni and G. Rossi, "Improving available bandwidth estimation using averaging filtering techniques," 2008.

[124] R. Scattolini, "Architectures for distributed and hierarchical model predictive control - a review," *Journal of Process Control*, vol. 19, no. 5, pp. 723–731, 2009.

[125] E. Camponogara, D. Jia, B. Krogh, and S. Talukdar, "Distributed model predictive control," *Control Systems, IEEE*, vol. 22, pp. 44 –52, feb 2002.

[126] A. J. N. van Breemen Breemen van, *Agent-Based Multi-Controller Systems - A design framework for complex control problems*. PhD thesis, Enschede, The Netherlands, May 2001.

[127] R. R. Negenborn, B. D. Schutter, J. Hellendoorn, R. R. Negenborn, B. D. Schutter, and J. Hellendoorn, "Multiagent model predictive control: A survey," tech. rep., 2004.

[128] J. B. Rosen, "Existence and uniqueness of equilibrium points for concave n-person games," *Econometrica*, vol. 33, no. 3, pp. pp. 520–534, 1965.

[129] D. Meilander, A. Ploss, F. Glinka, and S. Gorlatch, "A dynamic resource management system for real-time online applications on clouds," in *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *Lecture Notes in Computer Science*, Springer, 2011.

[130] D. Meilander, A. Bucchiarone, C. Cappiello, E. D. Nitto, and S. Gorlatch, "Using a lifecycle model for developing and executing real-time online applications on clouds," vol. 7221 of *Lecture Notes in Computer Science*, Springer, 2011.

[131] A. Ploss, D. Meilander, F. Glinka, and S. Gorlatch, *Towards the Scalability of Real-Time Online Interactive Applications on Multiple Servers and Clouds*, vol. 20 of *Advances in Parallel Computing*. IOS Press, 2011.

[132] B. Johansson and M. Johansson, "Distributed non-smooth resource allocation over a network," in *Decision and Control, 2009 held jointly with the 2009 28th Chinese Control Conference. CDC/CCC 2009. Proceedings of the 48th IEEE Conference on*, pp. 1678 –1683, dec. 2009.

[133] A. Nedic and A. Ozdaglar, "Distributed subgradient methods for multi-agent optimization," *Automatic Control, IEEE Transactions on*, vol. 54, pp. 48 –61, jan. 2009.

[134] S. S. Ram, A. Nedic, and V. V. Veeravalli, "Distributed subgradient projection algorithm for convex optimization," in *Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, ICASSP '09, (Washington, DC, USA), pp. 3653–3656, IEEE Computer Society, 2009.

[135] R. Olfati-Saber and R. Murray, "Consensus problems in networks of agents with switching topology and time-delays," *Automatic Control, IEEE Transactions on*, vol. 49, pp. 1520 – 1533, sept. 2004.

[136] I. Lobel and A. Ozdaglar, "Distributed subgradient methods for convex optimization over random networks," *Automatic Control, IEEE Transactions on*, vol. 56, pp. 1291 –1306, june 2011.