University of Pisa and Scuola Superiore Sant’Anna

Master Degree in Computer Science and Networking

Master Thesis

Improving stability in Adaptive Distributed Parallel applications: a cooperative predictive approach

Candidate                        Supervisor
Emanuele Vespa                   Prof. Marco Vanneschi

Academic Year 2011/2012
## Contents

1 Introduction 1

2 Overview of Structured Parallel Computations 7
   2.1 Structured Parallelism 8
      2.1.1 Dynamic reconfigurations 10
   2.2 Performance modeling 12
      2.2.1 Performance modeling of Acyclic Computational Graphs 15

3 Adaptive Parallel Modules 21
   3.1 Parallel Module definition 21
      3.1.1 Control Part behavior 23
      3.1.2 The Operating Part as a Hybrid System 23
      3.1.3 Adaptation strategy based on the MPC approach 25
   3.2 Controlling graphs of Parallel Modules 27
      3.2.1 Control scheme and distributed model definition 27
      3.2.2 A distributed formulation of the MPC approach 29

4 Accounting for reconfiguration stability: a new distributed control model 33
   4.1 Control problem instance and QoS modeling 34
      4.1.1 Introducing a metric for reconfiguration overhead 36
   4.2 Distributed optimization based on the sub-gradient method 38
      4.2.1 Weak sub-gradient calculus and its application 42
      4.2.2 Considerations on model assumptions 48
   4.3 The Holt-Winters forecasting method 48

5 Evaluating the impact of the switching cost on reconfiguration stability 51
   5.1 A simulation environment for ParMod graphs 52
      5.1.1 An OMNeT++ module simulating an Adaptive ParMod 52
Contents

5.2 An example of functional partitioning with three parallel modules

5.2.1 Time series forecasting errors

5.2.2 Effectiveness of a multiple-step ahead approach: performance versus stability

5.2.3 On the approach feasibility

5.2.4 Conclusions

5.3 Stability in a heterogeneous scenario: a second example

5.3.1 Considerations on performances and stability

6 Conclusions
## List of Figures

1.1 Number of reconfigurations versus tasks completed  
2.1 Task-farm scheme.  
2.2 Generic Computational Graph.  
2.3 Queuing system  
2.4 Network of queues  
2.5 Two-module pipeline system  
2.6 n-stages pipeline system  
2.7 Queuing network with single source and multiple destinations.  
2.8 Queuing network with multiple sources and single destination.  
2.9 Example of multiple paths from source to destination.  
3.1 Example of multiple paths from source to destination.  
3.2 Time discretization in control steps.  
3.3 Receding Horizon technique with horizon length of two control steps.  
3.4 Partially interconnected control scheme  
3.5 Interconnection scheme between two ParMods  
4.1 Pipeline of $n$ parallel modules  
4.2 Example of a graph composed by five ParMods.  
4.3 Example of time-series forecasting.  
5.1 Adaptive ParMod simulated through an OMNeT module.  
5.2 Abstract behavior of the interaction between different simulation modules implementing operating parts of ParMods.  
5.3 Abstract behavior of the simulation module implementing the Par-Mod control part.  
5.4 Experiment 1 application graph.  
5.5 Probability evolution for 600 time steps.  
5.6 Mean-Absolute Percentage Error per step - Horizon 1 and 2  
5.6 Mean-Absolute Percentage Error per step - Horizon 3 and 4
List of Figures

5.7 Reconfigurations of the three parallel modules without considering the switching cost ........................................ 63
5.8 Reconfigurations of ParMod$_1$ considering the switching cost ................................................................. 64
5.9 Reconfigurations of ParMod$_2$ considering the switching cost ................................................................. 65
5.10 Reconfigurations of ParMod$_3$ considering the switching cost ................................................................. 66
5.11 Reconfiguration magnitude histogram ........................................................................................................ 67
5.11 Reconfiguration magnitude histogram ........................................................................................................ 68
5.12 ParMod$_1$ utilization factor histograms .......................................................................................................... 70
5.13 ParMod$_2$ utilization factor histograms .......................................................................................................... 70
5.14 ParMod$_3$ utilization factor histograms .......................................................................................................... 70
5.15 Experiment 1 application graph..................................................................................................................... 74
5.16 Mean calculation time of the Source ............................................................................................................... 76
5.17 Mean-Absolute Percentage Error per step - Horizon 1 and 2 ................................................................. 76
5.17 Mean-Absolute Percentage Error per step ..................................................................................................... 77
5.18 Reconfigurations of ParMod$_1$ and ParMod$_2$ with the switching cost ................................................................. 78
5.19 Reconfigurations of ParMod$_3$ and ParMod$_4$ with the switching cost ................................................................. 79
5.20 Efficiency of the Source node ............................................................................................................................ 80
5.21 Efficiency of the parallel modules when they exploit the maximum parallelism degree. ................................................................. 80
Chapter 1

Introduction

In last years, several efforts have been done in studying adaptiveness in distributed parallel computations. Adaptiveness is the general property of a system to react to computation dynamics, i.e. those factors that affect the system behavior and that during the computation execution could assume statically unpredictable values. In a dynamic context, the initial execution conditions could rapidly change, resulting in a possibly inefficient system configuration. The main feature of an adaptive system is the ability of self-adjusting its behavior in order to face the various dynamicity sources, and to fulfill the desired Quality of Service (QoS), i.e. a series of quantitative and qualitative metrics that reflect what we expect from the distributed computation behavior (e.g. performances in terms of tasks computed per unit of time, power consumption or network bandwidth exploited). The entire adaptation process, lays its foundation on two main aspects: the ability to perform dynamic reconfigurations and the ability to take reconfiguration decisions. The first is the capability of changing at run time several characteristics of the components. Distributed parallel applications are expressed by means of proper compositions of computational components (namely parallel modules [22, 1]). In this context, an application configuration is completely specified by: (i) the parallelism scheme adopted by each parallel module, (ii) the parallelism degree exploited by each parallel module, (iii) the mapping between application components and the actual platform in which they are executed. Each of these aspects can be subject of dynamic reconfiguration, which are distinguished according to two broad categories:

Functional reconfiguration : they involve the semantics of the parallel computation. In this case the reconfiguration can involve algorithms, parallelism schemes, and even sequential computation exploited by a component. A necessary condition is that the newly chosen configu-
ration is semantically compatible with the previous one: the module reconfiguration must not change the whole application behavior.

**Non-functional reconfiguration**: they involve non-functional aspects of the parallel computation. Parallelism degree and execution platform are notable examples.

In general, performing a reconfiguration could induce a cost, which in our view is an abstract concept that could have several concrete characterizations. In switching from a configuration to another, it should be taken into account several factors, such as the cost of the newly selected configuration (e.g. in a pay-per-use execution environment proportional to the resource utilized). A reconfiguration cost could also be proportional to the magnitude of the switch. Changing the parallelism degree of a module by a significant factor could be harder w.r.t to changing it of one unit, or as an example changing the parallelism scheme is obviously more difficult than adjusting the parallelism degree. After all these considerations, it is clear that a reconfiguration should be executed when it brings effective benefits in achieving the QoS requirements.

In this direction, proper *control logics* must be adopted. In [16] has been proposed a novel control framework based on *control theoretic* techniques. It is shown as, given the performance models of structured parallel computation, it is possible to exploit adaptation strategies that try to optimize reconfiguration decision from the QoS point of view. A well-known control approach is the so called *Model-based Predictive Control* (MPC) [20, 15]. It is a form of *optimal control* and it is based on two assumptions:

- the existence of a system model. By exploiting a formal model, it is possible to predict the behavior that the controlled system will exhibit in the future. In our view, the parallel behavior of each component of the computational graph is expressed by means of few, well-known parallelism schemes. Each of these schemes has a clear and well-defined structure, with associated a precise semantics. In this way, it is possible to formulate analytical models that capture their behavior, and helps in predicting the module performance [23, 24]. Furthermore, by exploiting their *composability* property, the computation performance can be expressed as a function of the individual module performances;

- the possibility of express the computation QoS goals by means a proper set of objective functions, each one associated to each graph component. In this way, each module has a local knowledge of the computation QoS goal.
The main advantage of this control approach resides in the fact that, predictions of the application future behavior are exploited in order to take reconfiguration decisions for a finite subsequent time interval (namely \textit{prediction horizon}). By making use of statistical forecasts of the values that the external factors will take \cite{10, 25, 5}, it is possible to predict and ideally anticipate incoming QoS violations.

In order to exploit an MPC control strategy, parallel modules may need to interact in order to optimize their local cost function. At this regard, two possible approach to the interaction are possible: modules can expose a \textit{competitive} or a \textit{cooperative} behavior. In a competitive approach, modules optimize their local cost function without considering what impacts their reconfiguration decisions will have on other modules. On the contrary, a cooperative approach consists on minimize each local objective function in a way that the global objective (i.e. the sum of the local cost functions) is minimized \cite{18}. Obviously this process must be completely automated. To face this problem, each parallel module has been provided of an autonomic controller (also called \textit{control part}), which is capable of observing the behavior of the module functional part (namely \textit{operating part}) and to take consequently control decisions by proper interacting with others interconnected controllers.

\textbf{Taking a step toward reconfiguration stability}

As we have already hinted, a reconfiguration process possibly induces costs, both in terms of overhead or monetary charges due to the dynamic provisioning of resources. Taking into account these aspects is of crucial importance. \textit{Grid} platforms \cite{8} where very popular in the last decade and \textit{Cloud} infrastructures are gaining more and more attention by the scientific computing community (see \cite{13} for a survey on this topic). It is clear that, given the nature of these execution environments, reconfigurations do not come for free. Even if, from the application viewpoint, resources (that can be of any kind, from computing power to storage volumes) are presented as homogeneous, dedicated and possibly infinite, they are not. They are instead distributed and shared, meaning that, under sustained load situations, finding appropriate additional resources could be tricky. Regarding \textit{elasticity} (i.e. the ability of dynamically allocate/release resources \cite{12}) cloud infrastructures are still not mature enough for supporting adaptiveness in a proper way. As an example, \textit{public clouds} infrastructures such as Amazon EC2\footnote{http://aws.amazon.com/ec2/}, support elasticity but in terms of deploying/undeploying virtual
machines. Clearly these kind of operations may take minutes to complete, inducing both performance and monetary cost.

In [16] it has been proposed a QoS modeling aimed at controlling the performance and the resource utilization of distributed parallel applications. In this way it was possible to express proper trade-offs between the actual number of tasks computed and the resources utilized during the execution. However, that formulation was not completely satisfying from the reconfiguration stability point of view. In this thesis stability is expressed in two terms. The first is the total number of reconfigurations enacted for the entire execution duration. The second is the average time for which a module configuration remains stable. Both of these aspects are of crucial importance, since reflect the quality of control decisions. Our goal is to maximize the execution performance, but at the same time, exploit the less number of reconfiguration possible, guaranteeing general system stability. If a reconfiguration plan lasts for a reasonably long amount of time, it implies that control decisions where robust and durable.

**Thesis contributions**

Given the previous considerations, we claim that a model that takes into account the costs induced by a reconfiguration process is needed. The main goal, is to provide a control strategy that induces stable reconfiguration decisions. In this direction, we give the following contributions

- we will extend the model proposed in [16] in order to exploit multiple-step ahead predictions of the modules behavior. By doing so, reconfiguration plan will take into account not only what is going to happen in the immediate future, but also what is likely to happen further in the prediction horizon.

- we introduce a metric, namely a switching cost, that takes into account the costs that a reconfiguration phase could bring. This will lead to a consistent stabilization of the system, especially in execution environments that expose rapidly time-varying execution conditions

The switching cost is a quantitative parameter that each parallel module explicitly consider in its local cost function. We show experimentally that with a sufficiently long horizons, the cooperative MPC approach can significantly increase the stability of reconfigurations, with little impact on the performances. This is an important point, since we would like to have a more stable system without decreasing the number of tasks computed.
In order to prove the effectiveness of our approach, we perform an extensive testing in a simulation environment. The objective of analyze from a qualitative and quantitative point of view the computation properties of stability, performance and efficiency. As an example, we consider a computational graphs with a single sequential source, in which its service time continuously change in time, causing consistent variation on the workload to be sustained by the rest of the system. As a consequence, the other parallel modules need to adjust their parallelism degree in order to face the increasing/decreasing arrival of tasks. We compare two adaptation strategy: the first tries to optimize performances using the less number of resources possible. A second formulation instead, maintain the same goal, but at the same time consider a cost proportional to the magnitude of the switch (i.e. bigger parallelism degree variations imply a bigger penalty). For the observed portion of the computation, the second formulation, associated to a sufficiently long prediction horizon, is capable of decreasing of a 40 % the total number of reconfigurations (i.e. the sum of the number of reconfiguration performed individually by parallel modules), but computing only a 0.5% less of tasks with respect to the first strategy. We have shown as, by considering different prediction horizon length, it is possible to obtain different trade-offs between stability and performances. In Histogram 1.1 is represented the total number of tasks computed by the system in the observed computation interval with the corresponding number of reconfigurations. The leftmost values are relative to the strategy which does not consider any switching cost (labeled No Switching Cost), whether labeled with Horizon i are the values relative
Chapter 1. Introduction

to the formulation with a switching costs, for prediction horizon of length $i = 1, 2, 3, 4$. It is clear that, by considering a switching cost and exploiting a longer prediction horizon for the MPC strategy, the number of reconfiguration is consistently reduced, while the number of tasks completed is slightly less.
Chapter 2
Overview of Structured Parallel Computations

Distributed parallel computations have been subject of extensive investigations throughout the past decade. Research focused on defining methodologies to make parallel programming simpler and effective from a performance and efficiency point of view. Performance portability is a fundamental issue: structured parallel computations should be executable on different hardware-software platforms with no modification at the application level and provide acceptable performance results. This can be achieved by tuning different functional and non-functional parameters in order to fully exploit the underlying architecture capabilities, guaranteeing performance portability across different platforms. Another key feature is code reusing, that is the possibility of combining existing components to create more complex components and work-flows graphs, increasing the degree of flexibility in designing and developing distributed parallel applications. This requires a high level approach to parallel programming, since developing directly with message passing and shared memory libraries is too rigid and often does not provide code and performance portability. In this direction, structured parallel programming [7] has been proposed as a possible approach to parallel programming, since it provides abstract parallel patterns and performance portability. According to this methodology, parallel computations can be expressed by using a small set of well known patterns, namely parallelism scheme. Each of these schemes has a clear and well-defined abstract structure, with possibly multiple implementation of the communication and computational patterns. This methodology helps in achieving the performance predictability of applications. By exploiting a formal model of the abstract parallel computation and by defining a parametric cost model (notable parameters are communication latency and calculation time), it is possible to predict performance using
tools like Queuing Theory and Queuing Networks. The actual implementation can take advantage of the performance modeling to operate various optimization, both at compile time and at run-time. As we will see, it is possible to implement dynamic run-time supports for these parallel structures, able to perform functional and non-functional reconfigurations according to system dynamics. Run-time reconfiguration depend upon the component capability of take decisions according to a certain strategy.

2.1 Structured Parallelism

The structured parallelism programming approach is founded upon the concept of parallel scheme, also called skeleton [7]. They sport the following features:

- they have constraints on 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 more complex computational graphs.

These properties clearly define what we intend for structured parallelism. Each computation entity has a precise role in the parallel computation logic, which has an unambiguous structure associated with a specific semantics. We can distinguish two main categories of parallelism paradigm: stream-parallel paradigm and data-parallel paradigm:

**Stream-parallel paradigm**: parallelism schemes belonging to this family are used to improve the computation throughput in presence of a large stream of input elements. This is a necessary condition in order to have performance improvements by using these parallelization techniques. Stream parallel computations exploit parallelism on tasks: different stream elements are computed in parallel.

**Data-parallel paradigm**: if the computation operates on a single or a very restrict set of elements, performance can be improved by exploiting parallelism on data. Typically this parallelism schemes are used to reduce the computation latency, but, for stream based computations they are amenable to improve also the throughput.

Throughput and computation latency are two important performance parameters. Parallelism schemes for stream parallel computations generally
Structured Parallelism

lead to a better throughput at a price of an increased latency with respect to the sequential computation. Data parallel schemes instead tend to decrease both latency and throughput.

The behavior of a structured parallel computation is expressed by a set of computational units (processes or threads) that communicate by executing communication primitives (send and receive operations) on communication channels. Each computational unit has a precise role in the parallel scheme. To stress this aspect, we report some notable examples.

**Task Farm scheme**: it is a parallelism scheme operating on stream which consists on the replication of the entire sequential computation on $n$ identical units (namely workers). Apart from the workers, in this scheme we distinguish two special units. The first is called emitter which is in charge of receiving input elements from the input stream and distribute them according to a certain strategy to the workers. The second is the collector, a unit in charge of collecting results from workers and send them to the output stream. The scheme is represented in Figure 2.1.

![Figure 2.1: Task-farm scheme.](image)

**Pipeline scheme**: it is a simple parallelism scheme that operates on stream. Let us suppose that our sequential computation is a function expressed by a composition of $n$ functions: $f(x) = f_n(f_{n-1}(\ldots f_1(x)))$. Then we can define a linear graph of computation units (called pipeline stages) each of them implementing a function of the chain. This parallelism scheme increases the throughput, but also the latency proportionally to the number of stages due to the communication overhead.

**Data Parallel scheme**: in a data-parallel computation, the data are partitioned among execution units (workers) which apply the same function
to their portion of input data. The emitter distribute the input data by exploiting collective operation such as scatter or multicast. The collection of the partial result and the building of the output data is performed by the collector. Depending on the semantics of the sequential computation that workers apply to their own data partition, they may require to communicate each other. In this case we talk about stencil-based computations, where a stencil is a particular communication pattern between workers. We talk about map computation if the workers are completely independent, i.e. no communication between workers occurs.

Distributed parallel applications can be expressed by complex computational graphs (workflows), in which each node is a sequential or parallel module structured according to these well known parallelism schemes (namely intra-module parallelism). We call the inherent parallel behavior of computational graphs inter-module parallelism. The semantics and the entire computation can be expressed as a composition of the semantics of individual modules. Also the computation performance (intended as the bandwidth of the entire graph) is obtained as a function of the performance of each module. These composability properties are extremely important since they assure the necessary flexibility in designing and developing parallel programs, discover bottlenecks and parallelize them.

### 2.1.1 Dynamic reconfigurations

Given a parallel application defined as a direct graph of parallel modules, an application configuration is completely identified by:

- the parallelism scheme adopted by each parallel module;
- the parallelism degree of each parallel module (i.e. the number of parallel unit executing the computation);
- the mapping between application components and the actual platform in which they are executed.

All of these aspects can be subject of dynamic reconfiguration. In fact, even if by using a proper application model we can individuate an optimal initial configuration, that could change at run-time due to computation and system dynamics. There is a variety of factor that can influence the application optimal configuration at execution time:
2.1. Structured Parallelism

- often distributed applications are executed on non-dedicated environment in which computing and networking availability can change randomly and consistently over the time;

- QoS requirements can change dynamically during the execution;

- the irregularity of the input data can be a critical source of inefficiency (e.g. data-parallel programs optimal configuration depends on the input size).

Considering the previous described configuration aspects, we can discern two types of reconfiguration:

**Functional Reconfiguration**: they involve the semantics of the parallel computation. In this case the reconfiguration can involve algorithms, parallelism schemes, and even sequential computation exploited by our parallel module. A necessary condition is that the newly chosen configuration is semantically compatible with the previous one: the module reconfiguration must not change the whole application behavior.

**Non-functional Reconfiguration**: they involve non-functional aspects of the parallel computation. Parallelism degree and execution platform are notable examples.

Both of these reconfiguration types can be necessary at execution time. Let us consider the case in which a distributed parallel application works on a stream of data of different type. The computational graph is composed by six parallel modules: one dispatcher and four executor (functional modules). Each executor implements a different algorithm for each data type, but only one algorithm implementation can be active at time. According to its current configuration an executor is eligible to be scheduled for processing the next compatible input data. It could happens that during the execution the frequency of a certain data type changes, leading to an unbalanced tasks allocation between executors. In this case a functional reconfiguration could be necessary: by changing the algorithm in one or more executors we can overcome this situation. If also the size of input data changes (e.g. matrix of increasing size) it could also be necessary to change the parallelism degree or the parallelism scheme of some executors.

In our approach, dynamic reconfigurations rely on a proper behavioral model. By exploiting a formal cost model of the computation, it is possible to evaluate the impact of computation dynamics on the application performance. In the following section we will give some basic notions on performance modeling of computational graphs. We will exploit fundamental results in the area
Chapter 2. Overview of Structured Parallel Computations

of Queuing theory to express cost models for parallel computational graphs. This will allow to formally evaluate the performance of cooperating parallel modules, starting from the individual modules performance.

2.2 Performance modeling

A parallel computation can be represented in an abstract way by a directed graph (i.e. a workflow) in which nodes are sequential or parallel modules, and arcs are typed communication channels.

![Figure 2.2: Generic Computational Graph.](image)

The methodology we exploit to express the performance model of parallel computations is found upon the results obtained in the area of Queuing Networks. In stream-based computations, each computational module can be represented by a queuing node [23, 24], as shown in Figure 2.3. The system is described analytically by five parameters:

- **service discipline**: if not stated differently, it is assumed to be FIFO;
- **queue size**: number of incoming stream elements that the module can store;
- probability distribution of the random variable **service time** \( t_s \): it is the time that pass between the beginning of processing two consecutive stream elements. It has mean value \( T_S \) and variance \( \sigma_s \);
- probability distribution of the random variable **inter-arrival time** \( t_a \): it represent the time between the arrival of two consecutive stream element to the queue. It has mean value \( T_A \) and variance \( \sigma_a \);
2.2. Performance modeling

- probability distribution of the random variable *inter-departure time* \( t_p \): it is the time between two consecutive results departure from the module. It has mean value \( T_P \) and variance \( \sigma_p \).

**Figure 2.3:** Queuing system

From the performance viewpoint, computational graphs can be seen as a network of queues. A network of queues is composed by a set of queues interconnected in an arbitrary way, as shown in figure 2.4.

**Figure 2.4:** Network of queues

Queuing networks can be either *open* or *closed*. In the first case, an infinite number of task is generated by a set of sources, pass through the nodes and leave the system. In a *closed* network instead, a fixed number of tasks circulate in the network. We are interested in graph computations that can be modeled by open acyclic queuing networks, a sub-set of open networks in which a task can pass through a node at most once.

In order to evaluate the performance of a graph structured computation, the analysis is split in two phases:

**Transient Analysis** is the study of the initial phase of the computation. In this period the performance of each module can rapidly vary due to
the initial conditions of the graph. The main goal of this early analysis is discovering computational bottlenecks, which as we will see shortly impact the performance of others modules. In this phase performance parameters are intended as ideal.

**Steady-state Analysis** gives the effective performance of each module during the steady state phase. In this phase the performance level of each module is stabilized. The performance results are given in terms of effective inter-departure times. This phase is crucial both for evaluating the real application behavior and for applying adaptation strategies that dictate reconfigurations.

A central parameter is the **utilization factor** of a queue $\rho$:

$$\rho = \frac{T_S}{T_A} \quad (2.1)$$

It express the average level of congestion of a node. If $\rho \geq 1$ than the module is not able to serve requests at the rate they arrive to its queue. As said before the actual performance of a module is expressed by its mean inter-departure time $T_P$, that can be really different from its mean ideal service time $T_S$. We can express the inter-departure time of a module M in function of its service time by putting $T_P = T_S + \Delta$, where $\Delta$ is a delay that can incur due to two possible situations:

- the inter-arrival times to the module M is higher than its ideal service time. This imply that the module, after completing the processing of a stream element, will be blocked waiting for the next one;

- if the node to which module M is connected has expired its buffer capacity, then the module cannot send the result he computed. In queuing networks, this phenomenon is known as *blocking-after-service*, since the module cannot compute other stream elements until the destination node has freed a buffer position.

When the inter-arrival time to module M is greater than its ideal service time, this imply that module M is not a bottleneck and, with sufficiently large queue sizes, its inter-departure time equals its inter-arrival time. On the contrary, if the inter-arrival time is greater than its ideal service time, then module M is a bottleneck and its transient utilization factor $\rho$ is greater than 1. Considering that the queue size is not infinite, this imply that after the transient phase the others module will start to block, and as a consequence will increase their inter-departure times. From module M point of view this will be reflected in an increased inter-arrival time, until it will coincide with
its ideal service time. Hence at steady-state the utilization factor $\rho$ of any node will be lower or equal to 1. We can summarize this observations with the following proposition:

**Proposition 2.2.1** (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.*

### 2.2.1 Performance modeling of Acyclic Computational Graphs

In the rest of this section we will give some basic results for analyzing the transient and steady state behavior of acyclic computational graphs. We are interested in identifying bottlenecks and the impact that they will have on other modules. Our modeling approach simplifies some results of *Queuing Networks* in order to be more general: addresses the performance modeling problem without any assumption on probability distributions and considering configurable and sufficiently large buffers in front of each module.

#### 2.2.1.1 Pipeline Graphs

We start from a situation in which a module $M_1$ is connected to a module $M_2$ as shown in figure 2.3. Module $M_1$ is in charge of generating a stream of requests that will arrive to module $M_2$ buffer. The two nodes are characterized by ideal service time $T_{S1}$ and $T_{S2}$. In the initial transient phase the inter-arrival time to module $M_2$ is equal to the inter-departure time from module $M_1$, which is equal to $T_{S1}$. The second module utilization factor is equal to:

$$\rho = \frac{T_{S2}}{T_A} = \frac{T_{S2}}{T_{S1}} \quad (2.2)$$

![Figure 2.5: Two-module pipeline system](image-url)
Chapter 2. Overview of Structured Parallel Computations

We are now interested in identifying which modules are bottlenecks, in order to evaluate the performance at steady-state (i.e. their effective inter-departure times). Let’s consider the case in which the utilization factor of module M2 is lower than one. This means that it is under-utilized, and periodically remains blocked waiting for incoming requests, hence at steady state its inter-departure time will equal the ideal service time $T_{S1}$ of module M1. In the opposite case the utilization factor of module M2 is greater than one. Module M1 after a first transient phase will start to produce stream elements at a lower rate, precisely at a rate equal to module M2 ideal service time. This implies that the inter-departure time $T_{P1}$ will be equal to the ideal service time $T_{S2}$, which is also equal to the inter-departure time $T_{P2}$. This result is summarized by the following relation:

$$ T_{P1} = T_{P2} = \max\{T_{S1}, T_{S2}\} $$ (2.3)

At steady-state the effective behavior of the two modules is equal to the maximum ideal service time of the two modules in the network. This result can be generalized for a pipeline with an arbitrary number of nodes (Figure 2.6):

![n-stages pipeline system](image)

**Figure 2.6**: n-stages pipeline system

**Proposition 2.2.2.** In a pipeline graph the bottleneck is the module with the highest ideal service time. Furthermore, at steady state, the inter-departure time of each module in the graph will stabilize on that ideal service time.

The proposition can be proved by induction on the queue length. We sketch here the proof. Let us consider the two stage pipeline. If we add a new node at the end of it, two situations are possible: either the new node is a bottleneck or it is not. If it is a bottleneck, at steady state the inter-departure time from the two-stages pipeline will be equal to the service time of the newly added node. If it is not a bottleneck, its effective service time will be equal to the inter-departure time of the two-stages pipeline.

### 2.2.1.2 Queuing system with a source and multiple destinations

Consider the case in which the graph is composed by a single source and multiple destination nodes $M_1, M_2...M_n$ (Figure 2.7). In this case the source
2.2. Performance modeling

Figure 2.7: Queuing network with single source and multiple destinations.

$S$ sends stream elements to the node $M_i$ with probability $p_i$, where the condition $\sum_{i=1}^{n} p_i = 1$ holds.

In the transient phase the inter-arrival time $T_{A_i}$ to destination node $M_i$ is given by:

$$T_{A_i} = \frac{T_{Ps}}{p_i}$$  \hspace{1cm} (2.4)

This equation allows to discover whether a destination is a bottleneck or not. If $T_{A_i} > T_{S_i} \forall i = 1 \ldots n$, then no destination node is a bottleneck and at steady state equation 2.4 still holds, implying that the inter-departure time $T_{P_i}$ will be equal to $T_{A_i}$. On the contrary, if at least one destination node $M_k$ is a bottleneck then at steady state it will influence the inter-arrival time to other nodes. We can express the steady state inter-arrival times $T'_{A_i}$ as a function of the service time of the node $M_k$ with the highest utilization factor ($p_k = \max_i \rho_i$):

$$T'_{A_i} = T_{M_i} \cdot \frac{p_k}{p_i}$$  \hspace{1cm} (2.5)

Hence at steady state the inter-departure times from destination nodes are $T_{P_i} = T'_{A_i}$, meaning that each node has an effective service time greater than its ideal service time.

2.2.1.3 Queuing system with multiple sources and a single destination

Let us consider the case in which a queuing system is composed of multiple sources $M_1, M_2 \ldots, M_n$ and a single destination $D$, as shown in Figure 2.8.
Chapter 2. Overview of Structured Parallel Computations

We denote with $T_{S_i}$ and $T_{P_i}$ respectively the ideal service time and the inter-departure time of the generic source $M_i$; with $T_{A_D}$ and $T_D$ the inter-arrival time and the ideal service time of the destination $D$. In the transient phase the inter-arrival time to the destination is given by:

$$T_{A_D} = \frac{1}{n} \sum_{i=1}^{n} \frac{1}{T_{P_i}}$$ (2.6)

Once the inter-arrival time to the destination is known, it is possible to analyze the steady state behavior of the system. If the utilization factor $\rho_D$ is less then one (destination is not a bottleneck), then the inter-arrival time to $D$ is the same as in the transient phase. In the opposite case in which the destination is a bottleneck, then it is required a more complex analysis to understand which will be the inter-departure time of each source. In [14] is given an exhaustive treatise of this problematic providing results in the case in which service times random variables are exponential distributed.

To model the performance of acyclic graphs, we use a simple yet powerful approach valid for a large class of graphs: i.e. acyclic graphs with a single source. This approach originally introduced in [16], makes it possible to evaluate the steady state behavior of a graph without any assumption on the probability distributions of service times and in the case of large enough buffers. It allows an elegant closed-form modeling extremely important for the goals of this thesis.

2.2.1.4 Performance modeling of Acyclic Single-source Graph structures

A simple mathematical model can be achieved if we assume that stream elements are generated by a single source. As we said in section 2.1 each
module of the graph can be either sequential or parallel. To analyze the steady state performance of a graph with \( N \) nodes, we need to estimate the inter-departure time from each module. For a generic module \( i \) it is calculated as the point-wise maximum of \( N \) functions \( f_{i,j} \) with \( j = 1, 2, \ldots, N \):

\[
T_{pi} = \max\left\{ f_{i,1}(T_{S1}), f_{i,2}(T_{S2}), \ldots, f_{i,N}(T_{SN}) \right\} \quad (2.7)
\]

Function \( f_{i,j}(T_{S_j}) \) is defined by the following expression:

\[
f_{i,j}(T_{S_j}) = T_{S_j} \frac{\sum_{\forall \pi \in P(S \rightarrow j)} \left( \prod_{\forall e \in \pi} e.p \right)}{\sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right)} \quad (2.8)
\]

where \( P(S \rightarrow j) \) is the set of all paths from the source \( S \) to the destination node \( j \). A path \( \pi \) is an ordered set of edges, appearing in the order they must be traversed. An edge \( e \) of the path is a triple \((N_s, N_d, p)\) where the first and the second element represent the source and the destination node of the arc, whether the third element represent the probability of traversing the edge. We denote with \( e.p \) the probability of traversing edge \( e \). Each function

\[f_{i,j} \text{ (2.8)} \text{ addresses the case in which node } j \text{ is the bottleneck from node } i \text{ perspective. Notice that, as we have already seen in the previous section, if node } i \text{ is the bottleneck then its inter-departure time at steady state equals}\]
Chapter 2. Overview of Structured Parallel Computations

its ideal service time:

\[ T_{pi} = \max \left\{ f_{i,1}(T_{S_1}), f_{i,2}(T_{S_2}), \ldots, f_{i,N}(T_{S_N}) \right\} = f_{i,i}(T_{S_i}) = T_{S_i} \frac{\sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right)}{\sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right)} = T_{S_i} \]

In the opposite case in which node \( i \) is not the bottleneck, its steady state inter-departure time will depend from node \( j \) (the current bottleneck) performance. Since we do not know in advance which module is the bottleneck, then we find it by calculating the maximum between each function \( f_{i,j} \).
Chapter 3

Adaptive Parallel Modules

In this chapter we report the structure and the features of an adaptive parallel module. We first define what a parallel module is, focusing on its structure in terms of operating and control parts. In the second part we will focus on the issue of controlling distributed graphs of adaptive parallel modules.

3.1 Parallel Module definition

A parallel module (ParMod) is an independent and active unit featuring a parallel computation and an adaptation strategy for reacting to computation dynamics [1]. The ParMod structure is divided into two interconnected parts:

Operating part: it is the part in charge of performing the actual computation. Internally, it is structured according to a certain structured parallelism scheme. It is connected to one or more input streams from which receive data to be processed. Depending on the chosen semantics, it can expose either a non-deterministic or a data-flow behavior. In the first case, the computation is activated if at least one stream element is present in its input interfaces, whether in the second case at least one element must be present in each input interface. Once the computation has finished it sends on its output interfaces the processed result.

Control part: it implements the control logic. This part is responsible for monitoring the operating part behavior. According to a certain adaptation strategy, it is capable of generating reconfiguration commands and send them to the operating part.
As already hinted in Chapter 2, a ParMod can behave according several alternative versions, also called operations. An operation completely identifies the functional semantics of the parallel module, which is given in terms of the parallelized algorithm and the parallelism scheme used during the execution. The ParMod configuration is fully specified by the operation, the parallelism degree and the execution platform. A necessary condition is that alternate configurations preserve the input-output interfaces integrity in order to not modify the graph computation semantics. Under precise directives supplied by the control part, the operating part run-time support is able to perform dynamic reconfigurations during the execution. We can distinguish between two types of reconfiguration:

**Functional reconfigurations** involve the operation currently used by the parallel module. Furthermore, often changing the operation implies modifications of the current execution platform and the parallelism degree.

**Non-functional reconfigurations** consist in changing the parallelism degree exploited by the parallel module and/or the current execution platform.

In Figure 3.1 is reported the parallel module structure. The operating part is in charge of collecting several measurements, such as: number of queued elements, actual number of processed elements per time unit or mean arrival date. Once they are collected, according to a certain time scheduling, they are sent to the control part as monitored data. The controller in its turn, given the monitored data and an adaptation strategy, will provide reconfiguration commands to the operating part.
3.1. Parallel Module definition

3.1.1 Control Part behavior

In the control theory literature, controllers can be classified according to two main categories: event-driven and time-driven controllers. An event-driven controller executes the control logic only on the occurrence of predefined events. A time-driven controller instead, execute the control logic periodically in a synchronous way. In this approach time is usually discretized in periods of fixed length. In this work the ParMods control parts are time-driven controllers. We call control step the period between two successive evaluation of the adaptation strategy with fixed duration $\tau$. At the beginning of each control step, the controller will acquire the monitored data from the operating part. Once all the data has been received, the control algorithm can be executed, and eventually a set of reconfiguration commands will be sent to the control part. In Algorithm 1 is reported the described procedure.

```
Algorithm 1: Control Part synchronous behavior.

begin
  foreach control step $\tau$ do
    Receive data from the Operating Part;
    Execution of the adaptation algorithm;
    Transmission of processed reconfiguration commands to the Operating Part;
  end
end
```

At this point we should specify few things. The control phase is composed by the adaptation algorithm previously described plus the time needed by the operating part to apply the desired changes. Submitted reconfigurations will be active for the whole current control step. Therefore, the step length plays an important role: shorter steps permit to react in a fast and effective way to system dynamics. This could be an advantageous set-up in rapidly changing scenarios, even if the resource utilization overhead incurred by frequent execution of the control phase could be expensive. On the other hand longer control steps induce less over head, but could be inefficient in responding to changes that occurs during the execution.

3.1.2 The Operating Part as a Hybrid System

In this section we give a description of parallel modules as a particular type of dynamical systems. They belongs to the class of hybrid systems,
which feature both continuous and discrete parts, formalized by an unified model. As already mentioned, time is divided in discrete steps of duration $\tau$, where with the nomenclature control step $k$ we refer to the time interval $[k\tau, (k+1)\tau)$ as shown in Figure 3.2.

![Figure 3.2: Time discretization in control steps.](image)

The operating part behavior is modeled by means of a set of mathematical equations. This permit us to observe and predict future QoS evolution throughout the computation. To exploit this, it is needed a set of variables that properly model the system:

**QoS vector**: denoted by the term $x(k)$\(^1\), it is a real valued vector representing the system QoS (or, in some cases, stateful information) assumed at the beginning control step $k$;

**Observed output vector**: indicated by $y(k)$, it represent the observed output of the system at the beginning of control step $k$. It can be derived with proper transformations of the QoS variables;

**Disturbance input vector**: represents all those factors that affect the computation but that cannot be controlled. Examples of disturbances can be: mean calculation time per element, arrival rate of a certain data type and communication latency. The effect of these uncontrolled variables is to affect the relationship between control actions and QoS variables. We denote them with $d(k)$;

**Control input vector**: indicated by $u(k)$, it represent the reconfiguration signal issued by the control part for step $k$. Its elements corresponds to the configuration parameters of the ParMod, i.e. operation used ($op(k)$), parallelism degree ($n(k)$) and execution platform ($p(k)$):

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

\(^1\)With lowercase boldface letters we indicate vectors
Notice that the control input vector can take only a finite number of discrete values. This restriction is expressed by the following admissible set:

\[ U = \{ u(k) | op(k) \in M.Op \wedge p(k) \in Platforms(op(k)) \wedge 1 \leq n(k) \leq Nodes(p(k)) \} \]  

(3.1)

where the constraints are: (i) \( op(k) \) belongs the set of operations implemented by the parallel module (\( M.Op \) for brevity); (ii) \( p(k) \) must be a platform on which \( op(k) \) is executable; (iii) \( n(k) \) is the actual number of processing elements present available in platform \( p(k) \).

At this point we have all the ingredients to describe the evolution of the system as a function of the QoS state, the disturbances and the current control input:

\[
\begin{align*}
    x(k+1) &= \Phi(x(k), d(k), u(k)) \\
    y(k) &= H(x(k))
\end{align*}
\]  

(3.2)

where the first difference equation expresses the QoS of the system at the beginning of control step \( k + 1 \) as a function \( \Phi \) of the values taken by the variables at step \( k \) (and, optionally, state information at step \( k \)). The second equation instead establishes the relationship between observed outputs and the QoS variables, expressed by the function \( H \). For our purpose this function will be the identity function, i.e. the QoS is completely observable.

### 3.1.3 Adaptation strategy based on the MPC approach

In this section we present the control-theoretic adaptation strategy exploited by parallel modules. Model Predictive Control (MPC) is a form of optimal control that consists in computing the reconfiguration plan by solving an on-line optimization problem at each step [15]. The core of this approach relies on predicting the behavior of the controlled system for a finite number of steps ahead (prediction horizon). In order to compute the optimal control input trajectory (i.e. the control trajectory that minimize a properly defined objective cost function), we need to estimate what values will the disturbances take in the future. This can be achieved by exploiting well-known statistical forecasting techniques widely used in the field of time-series analysis, as we will see in next chapters. We denote the predicted trajectory of disturbance inputs for \( h \) control steps with:

\[ \hat{D}(k) = \{ \hat{d}(k|k), \hat{d}(k+1|k), \ldots, \hat{d}(k+h-1|k) \} \]  

(3.3)

where the notation \( \hat{d}(k+i|k) \) is used to express the predicted value for step \( k+i \) using the knowledge available at step \( k \) and \( \hat{D}(k) \) indicates the vector
of predicted values for a fixed horizon $h$. A control input trajectory for $h$ future control steps starting from the $k$-th step is denoted by:

$$U(k) = \{u(k|k), u(k+1|k), \ldots, u(k+h-1|k)\}$$  \hspace{1cm} (3.4)$$

Finally, the system QoS trajectory, derived by considering the disturbances $D(k)$ and by applying a reconfiguration plan $U(k)$ is indicated with:

$$\hat{X}(k) = \{\hat{x}(k+1|k), \hat{x}(k+2|k), \ldots, \hat{x}(k+h|k)\}$$  \hspace{1cm} (3.5)$$

We are interested in finding the control input trajectory that minimizes a properly defined objective cost function $J$, which should be structured in a way that reflects the QoS requirements of the computation (notable examples can be performance, memory usage, efficiency). The optimization problem can be stated as it follows:

$$\underset{U(k)}{\text{argmin}} \ J(\hat{X}(k), U(k))$$

subject to:

$$u(i|k) \in U \quad i = k, k+1, \ldots, k+h-1$$

$$\hat{x}(i+1|k) = \Phi(\hat{x}(i|k), \hat{d}(i|k), u(i|k)) \quad i = k, k+1, \ldots, k+h-1$$

$$\hat{x}(k|k) = x(k)$$  \hspace{1cm} (3.6)$$

The controller will exploit the so called receding horizon technique: instead of applying the reconfiguration plan for all the prediction horizon, only the first control input $u(k|k)$ of the optimal sequence will be issued to the operating part. At the beginning of the next control step the whole procedure will be repeated, causing the prediction horizon to shift forward in time (Figure 3.3).

**Figure 3.3:** Receding Horizon technique with horizon length of two control steps.

It is clear that the formulation of the objective function is of central importance: it is the way in which we can express what are the goals of the adaptation strategy. As an example we can:

- optimize the performance (e.g. completed tasks per time unit) of the computation;
3.2. Controlling graphs of Parallel Modules

- find proper trade-offs between performance, number of reconfigurations and resource utilization.

This approach is particularly interesting since allows several degree of freedom in specifying which QoS parameters are most important in the computation cost model.

3.2 Controlling graphs of Parallel Modules

In order to deal with the problem of controlling complex graphs, we will use an extended model w.r.t the one introduced in the previous section. Several aspect must be considered, such as the way in which controllers are interconnected (fully-interconnected or partial-interconnected control schemes), which kind of information they exchange and how they exploit these information to establish a reconfiguration plan. First we should remark the fact that a distributed parallel application is made from a proper composition of interacting execution unit (ParMods), each one exploiting the necessary resources for execute both the functional logic and the control logic. In a distributed scenario, the original control problem (consisting in a formal system model and an adaptation strategy) is decomposed in multiple sub-problems. Depending on the global QoS objective, possibly individual units need to interact to take reconfiguration decisions, since actions for solving one sub-problem could influence actions to solve others. Some objectives can be fulfilled without interaction between ParMods; we can think of total memory usage minimization: in this case if trivially each controller minimize its memory consumption, the global goal will be consequently achieved. On the other hand, if for instance the QoS objective consists in maximizing the performance (in terms of tasks completed per time unit), in that case interactions between units is needed, since reconfigurations of one module impact performance of the others.

3.2.1 Control scheme and distributed model definition

The distributed control approach adopted in this work follows a distributed control scheme based on a single-layer organization, in which:

- the original control problem is decomposed in multiple a set of sub-problems (i.e. one for each parallel module of the application graph);

- each sub-problem is assigned to and solved by a dedicated controller; we assume one-to-one relationship between a control problem and a control entity.
controllers may eventually interact to reach an agreement on their control decisions.

Notice that usually sub-problems are not independent from each other, on the contrary they expose some degree of inter-dependency. This is reflected in the fact that actions for solving one sub-problem can affect the decisions taken by other controllers. This type of interaction is called **coupling relationship**, defined as the law that describe how actions for solving one sub-problem influence the others and vice-versa.

The structural characteristics of our control scenario are summarized by the following points:

- a distributed parallel application is represented as a direct graph of parallel modules $P_1, P_2, \ldots, P_N$;
- a generic parallel module $P_i$ is composed by a control part $PC_i$ and an operating part $PO_i$, interconnected in the way described in section 3.1;
- operating parts $PO_1, PO_2, \ldots, PO_N$ are interconnected by means of data streams that implement the distributed functional logic;
- control parts are synchronized on the same control step of duration $\tau$;
- control parts are *partially interconnected*: the controllers communication pattern is the same of their corresponding operating parts (Figure 3.4).

![Figure 3.4: Partially interconnected control scheme](image-url)

The parallel module model is extended with a new set of variables called **interconnecting variables**:
• **input interconnecting variables**: indicated with $v_{\text{in}^{-i}}(k)$, it is a real valued vector that models the interaction between other ParMods and ParMod $P_i$.

• **output interconnecting variables**: indicated with $v_{\text{out}^{-i}}(k)$, it is a real valued vector representing information about the local problem that will be sent to the interconnected control parts.

We denote with $v_{\text{in}^{-i,j}}$ the set of variables received by $PC_i$ from $PC_j$ and with $v_{\text{out}^{-i,j}}$ the set of variables sent by $PC_i$ to $PC_j$. In Figure 3.5 is shown the described interconnection scheme. The model of the generic $ParMod_i$ is extended by considering the newly introduced interconnecting variables:

$$x_i(k+1) = \Phi_i(x_i(k), d_i(k), u_i(k), v_{\text{in}^{-i}}(k))$$ (3.7)

On the other hand it is needed a way to generate output interconnecting variables. This is achieved by means of a *output generation* function $Z_i$:

$$v_{\text{out}^{-i}}(k) = Z_i(x_i(k), d_i(k), u_i(k))$$ (3.8)

Example of output (input) interconnecting variables can be: an appropriate subset of the disturbances, reconfiguration commands or part of the internal state.

### 3.2.2 A distributed formulation of the MPC approach

We will give now a formulation of the MPC approach described in Section 3.1.3 for distributed graphs of parallel modules. We are interested in optimizing at each control step a system-wide objective function $J_G$ that is
obtained as a linear combination of the local objective functions, each one representing a control sub-problem:

\[ J_G(k) = \sum_{i=1}^{N} w_i J_i(k) \]  

(3.9)

where \( w_i \) is the weight assigned to the \( i \)-th objective function. The parallel modules involved in the graph computation will exploit a cooperative behavior in order to find a control sequence that minimizes the global objective function. The main principle of this approach is that each controller optimizes its local objective function not in a selfish fashion, but rather considering the effect that its actions will have on others. Hence they exchange information to reach agreement on the reconfiguration plan, which could be very different w.r.t the one obtained by minimizing the local cost function in a selfish way (i.e. pursue the individual profit without caring of the global one).

The local optimization problem is stated as it follows:

\[
\begin{align*}
\text{argmin} & \quad J_i(\overline{X}_i(k), \overline{U}_i(k), \overline{V}_{in-i}(k)) \\
\text{subject to} & \quad u_i(j|k) \in U_i \ j = k, k+1, \ldots, k+h-1 
\end{align*}
\]  

(3.10)

where \( \overline{X}_i(k) \) is the local state trajectory, \( \overline{U}_i(k) \) is the local trajectory of reconfiguration commands and \( \overline{V}_{in-i}(k) \) is the received trajectory of input interconnecting variables, defined as usual as \( \overline{V}_{in-i}(k) = \{v_{in-i}(k|k), v_{in-i}(k+1|k), \ldots, v_{in-i}(k+h-1|k)\} \). As already mentioned, only the first control input of \( \overline{U}_i(k) \) will be issued to the operating part. Notice that we made no assumption on how often interconnecting variables are exchanged. In fact, depending on the control problem, it could be necessary to iterate the data exchanging several times within each control step till controllers reach a termination condition. A possible iterative interaction protocol can be the following:

1. \( PC_i \) retrieves the monitored data from the operating part;
2. predicts the future values of the disturbances for \( h \) step ahead;
3. assumes a initial trajectory for its input interconnecting variables \( \overline{V}_{i-in}(q) \) (where \( q \) is the current iteration of the protocol, initialized to zero);
4. solves the local optimization problem and obtains the reconfiguration trajectory \( \overline{U}_i^{(q)}(k) \);
3.2. Controlling graphs of Parallel Modules

5. checks the global termination condition (e.g. \( q = \text{max iteration number} \)):
   if true then terminates the protocol and issues the first control input to the operating part; if false continues with the next step.

6. calculates and sends output interconnecting variables trajectories \( V_{i-out}^{(q+1)}(k) \) to the interconnected control parts;

7. acquires the input interconnecting variables trajectories \( V_{i-in}^{(q+1)}(k) \) and returns to step 4.

We stress the fact that the global termination condition must be reached by all controllers in order to guarantee the procedure termination and effectiveness.

In this chapter we gave a control-theoretic characterization of our control framework. We summarize what are its main traits with the following remark:

Remark 3.2.1 (Control framework characterization). Our controlled system is an acyclic direct graph of parallel modules, each one featuring an operating part and a control part. The control scheme adopted expose a single-layer distributed organization, with a partially interconnected communication pattern between controllers that follows the same interconnections between operating parts. Control parts implement a distributed iterative protocol for establishing an optimal reconfiguration input, following a distributed Model-Based Predictive Control approach.

A crucial point consists in providing a method which allows the distributed MPC strategy to achieve globally optimal control trajectories. We address this aspect in the next chapter in two steps. First we will show how the application QoS has been modeled and give an analytical formulation of the ParMods objective function. Then we describe the numerical method exploited for solving the distributed optimization problem and the algorithmic procedure implemented by the controllers.
Chapter 4

Accounting for reconfiguration stability: a new distributed control model

In the last chapter it has been depicted the system model and the control approach that allows us to exploit adaptiveness in distributed parallel applications. In this chapter we will dive into the actual control problem definition, in terms of the objective function analytical formulation and the interactions between controllers. In past works [16] it has been proposed a QoS modeling aimed at controlling the performance and the resource utilization of distributed parallel applications. In this way it was possible to express proper trade-offs between the actual number of tasks computed and the resources utilized during the execution. Controllers exploited two types of interactions: competitive and cooperative. In the first case the goal of each control part was to minimize its local objective function, regardless of the effects that its actions would cause to others. In the second case instead, control parts cooperate for minimizing the whole system objective function, equal to the weighted sum of the individual objective functions. The main study was focused on the impact that adaptiveness can have on distributed computations and on the qualitative differences between the cooperative and competitive approaches. The aim of this work is of a different nature: fixed a distributed cooperative MPC control strategy, we want to study what solutions can improve the general stability of the system. In this thesis stability is expressed in two terms. The first is the total number of reconfigurations enacted for the entire execution duration. The second is the average number of consecutive steps for which control inputs do not modify the operating part behavior (i.e. they are equal to the ones issued at the previous step). Both of these aspects are of crucial importance, since reflect the quality
Chapter 4. Accounting for reconfiguration stability: a new distributed control model

of control decisions. Our goal is to maximize the execution performance, but at the same time, exploit the less number of reconfiguration possible, guaranteeing general system stability. If a reconfiguration plan lasts for a reasonably long amount of time, it implies that control decisions were robust and durable. We therefore introduce a further metric in order to take into account reconfigurations overhead and we study what are the benefits of exploiting multiple-step ahead QoS predictions. In this section we present in detail these aspects, providing also the algorithmic and analytical tools for solving the control problem.

4.1 Control problem instance and QoS modeling

Before presenting the QoS modeling and the techniques used to solve the distributed optimization problem, we need to define what are the variables that model our system. We will exploit reconfigurations involving only the parallelism degree of each ParMod, hence the operation used and the execution platform will be fixed. The local control input for the $i$-th parallel module is characterized by the following admissible set:

$$U_i = \{ n_i(k) | n_i(k) \in \mathbb{R} \land 1 \leq n_i(k) \leq n_{i}^{max} \} \quad (4.1)$$

where $n_{i}^{max}$ is the maximum number of computing nodes available in the current execution platform for ParMod$_i$. Notice that actually parallelism degrees will always take positive integers values in the interval $[1, n_{i}^{max}]$, but we exploit a continuous relaxation of the problem for tractability issues. In fact, we consider scenarios in which the number of computing nodes available to each parallel module can be very high (more than 20), resulting in an unfeasible number of states to be explored to find the integer optimum profile. This also implies that the real-valued profile we choose is not so far from the optimal integer one, allowing us to sacrifice a little in the optimality for the sake of simplicity and speed. Proper rounding techniques will be used before sending the reconfiguration command to the operating part. We can see the parallelism degree chosen by a parallel module as its strategy for control step $k$. A strategy profile is thus a vector composed by the strategies of each parallel module at step $k$:

$$s = [n_1(k), n_2(k), \ldots, n_N(k)]^T \quad (4.2)$$
Consequently, the system-wide control input admissible set is given by the cartesian product of the individual admissible sets:

\[ \mathbf{U} = U_1 \times U_2 \times \cdots \times U_N \]  

We will consider the case in which the system parameters that vary in time are the mean calculation time \( T_{\text{calc} - i} \) and the tasks routing probability \( p \), hence the disturbance input vector \( \mathbf{d}_i(k) \) has at least two real-valued components.

In Chapter 2 it has been shown as in a graph structured computation the effective performance of each component is represented by its steady state mean inter-departure time \( T_p \). In remark 4.1.1 is recalled this fundamental result:

**Remark 4.1.1** (Steady-state performance of single-source acyclic graphs). The inter-departure time from a generic module \( i \) can be expressed as the point-wise maximum between \( N \) functions \( f_{i,j}(T_{S_j}) \), each one addressing the case in which node \( j \) is the bottleneck.

The mean service time \( T_{S_i} \) can be expressed in terms of the mean calculation time and the parallelism degree of \( \text{ParMod}_i \):

\[ T_{S_i}(k) = \frac{T_{\text{calc} - i}(k)}{n_i(k)} \]  

where \( n_i(k) \) is the parallelism degree chosen for step \( k \). In order to make things simpler perfect scalability is assumed: we will shortly see that even if this assumption could seems rather strong, it is of little impact on our formulation, which can address non-ideal behavior of structured parallel computations. We can then rewrite the general inter-departure time expression as a function of the strategy profile \( \mathbf{s} \) and the control step:

\[ T_p(\mathbf{s}, k + 1) = \max_{j=1,2,\ldots,N} \left\{ f_{i,j}(n_j(k)) \right\} \]  

where functions \( f_{i,j}(n_j(k)) \) can be rewritten in the following way:

\[ f_{i,j}(n_j(k)) = \frac{T_{\text{calc} - j}(k)}{n_j(k)} \frac{\sum_{\forall \pi \in P(S \rightarrow j)} \left( \prod_{\forall e \in \pi} e.p \right)}{\sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right)} \]
Chapter 4. Accounting for reconfiguration stability: a new distributed control model

Summarizing, thankfully to these equations we are able to express the effective performance of each module in a single-source acyclic graph as a function of the parallelism degree.

In the MPC approach perspective, it is required an objective function that each parallel module optimizes at the beginning of each control step. The local cost function used till now, which constitutes a starting point for our work, is the following:

$$J_i(s, k) = \alpha_i T_{pi}(s, k + 1) + \gamma_i n_i(k)$$  \hspace{1cm} (4.7)

where $\alpha_i$ and $\gamma_i$ are two coefficient of proportionality which express the importance (cost) of each part of the local objective. The first part of equation 4.7 is related to the performance level of the parallel module: higher inter-departure times (i.e. the module is slow in terms of task computed) corresponds to a higher cost w.r.t lower inter-departure times. The second part of the equation instead expresses a cost proportional to the actual number of nodes used by parallel module $i$ at step $k$. It is clear that by properly tuning parameters $\alpha_i$ and $\gamma_i$ it is possible to express different trade-offs between performance and resource utilized: we can choose to have maximum performance by setting $\alpha_i \gg \gamma_i$, or to be more resource conservative at a price of lower performances by increasing the resource utilization cost.

4.1.1 Introducing a metric for reconfiguration overhead

As we have hinted in Chapter 2 the reconfiguration process possibly induce costs, both in terms of overhead or of monetary charges due to the dynamic provisioning of resources. Grid platforms [8] where very popular in the last decade and Cloud infrastructures are gaining more and more attention by the scientific computing community (see [13] for a survey on this topic). It is clear that, given the nature of these execution environments, reconfigurations do not come for free. Even if, from the application viewpoint, resources (that can be of any kind, from computing power to storage volumes) are presented as homogeneous, dedicated and possibly infinite, they are not. They are instead distributed and shared, meaning that, under sustained load situations, finding appropriate additional resources could be tricky. Regarding elasticity (i.e. the ability of dynamically allocate/release resources [12]) cloud infrastructures are still not mature enough for supporting adaptiveness in a proper way. As an example, public clouds infrastructures such as Ama-
zon EC2\textsuperscript{1}, support elasticity but in terms of deploying/undeploying virtual machines. Clearly these operations may take minutes to complete, inducing both performance and monetary cost. Given these considerations, we wanted to introduce a further metric that allows to take into account these overheads, namely \textit{switching costs}, defined as:

\[
\Delta(k) = (n_i(k) - n_i(k-1))^2
\]

where \(n_i(k-1)\) is the parallelism degree used by the \(i\)-th parmod at step \(k-1\). The intent of the \textit{switching cost} is the following: it binds control decisions between consecutive steps. In this way reconfiguration plans at step \(k\) are not independent from what happened at step \(k-1\). This permit us to overcome two limitations of the previous formulation:

- it did not consider what choices where made in the past: control inputs of different control steps are uncorrelated;
- it exploited only one-step ahead predictions.

The first point could be an issue in scenarios in which disturbances values sport a relevant variance. In that case it was possible that parallelism degree variations where big in magnitude and prone to some kind of up and down fluctuations. In this sense, the \textit{switching cost} act as a break, mitigating this undesired effect. Regarding the second point instead, we overcome that limitation by introducing multiple-step ahead predictions. This is an extremely important improvement since, as it will be shown experimentally, considering a longer prediction horizon leads to more stable and efficient control decisions. We define a matrix \(S \in \mathbb{R}^{N \times h}\) (namely \textit{strategy profiles matrix}), in which column \(i\) represent the strategy profile of \(ParMod_i\) (i.e. the trajectory of parallelism degrees for \(h\) control step ahead) and row \(q\) consists of the parallel degrees chosen by parallel modules for the \(q\)-th step of the horizon\textsuperscript{2}:

\[
S = \begin{bmatrix}
    n_1(k) & n_2(k) & \cdots & n_N(k) \\
    n_1(k+1) & n_2(k+1) & \cdots & n_N(k+1) \\
    \vdots & \vdots & \ddots & \vdots \\
    n_1(k+h-1) & n_2(k+h-1) & \cdots & n_N(k+h-1)
\end{bmatrix}
\]

\textsuperscript{1}http://aws.amazon.com/ec2/
\textsuperscript{2}For the sake of a more compact notation, we do not indicate the control step, which will be specified in case of ambiguity.
We denote with $S_q$ the $q$-th matrix row. The resulting local objective function is the following:

$$J_i(S, k) = \sum_{q=k}^{k+h-1} \alpha_i(q) \cdot T_{pi}(S_q, q + 1) + \sum_{q=k}^{k+h-1} \beta_i(q) \cdot \Delta_i(q) + \sum_{q=k}^{k+h-1} \gamma_i(q) \cdot n_i(q)$$

(4.9)

where $\alpha_i$, $\beta_i$ and $\gamma_i$ are proportional coefficients. It is worth to notice the flexibility of this formulation. It is completely parametric, allowing different trade-offs between performance, resource utilization and number of reconfigurations. Also the coefficients may vary as a function of the control step and/or the horizon. Dynamic tuning is a compelling feature for two reasons:

- in some cases, coefficients represent a sort of monetary cost. Especially in cloud environments it is not unusual to have time-varying resources pricing (e.g. data transfer bandwidth costs more in certain part of the day\(^3\)).

- future disturbance values are determined by forecasts based on the previous history, hence they could be wrong. It can be useful to give less weight decreasing weights as we move farther in the horizon, in order to smooth the effect of prediction errors.

In the following section we introduce the method exploited by the controllers to cooperatively minimize the system-wide objective function, starting from their local knowledge.

### 4.2 Distributed optimization based on the sub-gradient method

In this section we present the iterative optimization method used in this thesis to exploit the distributed MPC strategy described in the last chapter. Introduced in [18], it is based on the sub-gradient method (see [21]), and addresses the problem of optimizing in a distributed fashion the weighted sum of (possibly) non-smooth convex functions $\sum_{i=1}^{N} w_i f_i(x)$. In a multi-agent context, each function $f_i: \mathbb{R}^n \rightarrow \mathbb{R}$ represent the cost function of agent (parallel module) $i$, and it is known only by its agent. The main algorithm is the following: each agent computes and maintains a local estimate of

\(^3\)Switzerland cloud provider CloudSigma is a notable example, see http://www.cloudsigma.com/en/pricing/price-schedules
the optimal value based on its local knowledge. By exchanging directly or indirectly its estimate with other agents, at the end of an iterative protocol, they reach consensus on value that approximate the global, optimal one [19]. We denote with $S_{i}^{(t)}$ the strategy profiles matrix estimation of parallel module $i$ after $t$ steps of the iterative protocol. This optimization method fits particularly well our needs, since:

- it allows to minimize possibly non-smooth convex functions, which is our case since the non-smoothness induced by the presence of the piece-wise maximum operator;
- we are interested in solving the optimization problem in a distributed way: the possibility of a centralized approach it is not considered;
- our control goal is to optimize the system wide objective function in a cooperative way.

In order to exploit the cooperative minimization method, in [18] authors define two different sub-models: an information exchange model, which establish how agents' information change in time, and an optimization model, which specify how each agent contribute to the system wide objective function minimization.

**Information exchange model** It is assumed that agents update and exchange information synchronously at discrete time intervals, i.e. they advance to the next iteration in a synchronous way. Each agent $i$ maintains a vector of weights $\alpha_i$ representing the importance given to the information received by agent $j$, denoted by $\alpha_{i,j}$. The information exchange model is based on two assumptions ([19], [2]), reported in the following.

**Assumption 4.1** (Weight Rule). We have that:

(a) there exists a positive scalar $\eta$ with $0 < \eta < 1$ such that for all $i = 1, \ldots, N$ the following conditions hold:

- $\alpha_{i,i} \geq \eta$;
- $\alpha_{i,j} \geq \eta$ for each agent $j$ communicating directly with agent $i$;
- $\alpha_{i,j} = 0$ if $i$ and $j$ does not communicate directly.

(b) vectors $\alpha_i$ must be stochastic, that is $\sum_{j=1}^{N} \alpha_{i,j} = 1$;

(c) the matrix $W$ in which row $i$ is the vector $\alpha_i$, must be doubly stochastic.
Chapter 4. Accounting for reconfiguration stability: a new distributed control model

In word, it is required that it is given a non-null weight to information coming from directly interconnected agents, and zero to everyone else. A weight assignment that satisfies Assumption 4.1 is the so call equal weighting, where vector $\alpha_i$ is constructed in this way:

$$
\alpha_{i,j} = \begin{cases} 
\min \left\{ \frac{1}{|Nh(i)|+1}, \frac{1}{|Nh(j)|+1} \right\} & \text{if } j \in Nh(i) \\
0 & \text{if } j \notin Nh(i) \\
1 - \sum_{j \neq i} \alpha_{i,j} & \text{if } i = j
\end{cases}
$$

where $Nh(i)$ is the set of neighbors of agent $i$.

**Example.** If we consider a three stages pipeline graph in which controllers are partially interconnected, the resulting weight matrix (i.e. a matrix in which the $i$-th row is the $i$-th vector $\alpha_i$) is the following:

$$
W = \begin{pmatrix}
2/3 & 1/3 & 0 \\
1/3 & 1/3 & 1/3 \\
0 & 1/3 & 2/3
\end{pmatrix}
$$

**Assumption 4.2** (Connectivity Rule). It is assumed that the agent graph is strongly connected: there must exit a path between from each agent to every other agent. Our model satisfies this condition: controllers interconnection graph is connected by construction.

These two assumptions must be satisfied in order to reach consensus among agents. The first one is required to assure that each agent estimate is influenced by the others, the second one instead assures that information from one agent propagates to all other agents after a finite number of steps, under the assumption of bounded communication delay. In our framework information exchange between control parts is modeled by means of interconnecting variables. Since we are interested in controlling the parallelism degree of each parallel module, input-output interconnecting variables $\vec{V}_{in-i}(k)$ and $\vec{V}_{out-i}(k)$ will contain strategy profile estimation.

**Optimization Model** We now must show how agents can reach a consensus on the global optimal solution of the minimization problem. We recall that the system wide objective is to minimize the sum of the parallel modules’ individual cost functions:

$$
\min_{S} J_G(S, k) = \sum_{i=1}^{N} w_i J_i(S, k) \quad (4.10)
$$

subject to: $S_{-j} \in U_j$
4.2. Distributed optimization based on the sub-gradient method

A basic requirement for finding a global solution to this minimization problem is that each function $J_i$ must be convex. This requirement is satisfied by our functions since:

- the admissible set for parallelism degree $n_i$ is convex;
- each function $f_{i,j}$ is convex in the parallelism degree $n_i(k)$;
- the piece-wise maximum of convex functions is also a convex function;
- multiplication of convex function by a positive preserve convexity;
- functions $\beta_i \cdot \Delta(k)$ and $\gamma_i \cdot n_i(k)$ are convex;
- summation preserve convexity.

Given this, the fundamental relation that establishes how control parts update their estimate is the following:

$$S_{[i]}^{(t+1)} = P_U \left[ \sum_{j=1}^{N} \alpha_{i,j} S_{[j]}^{(t)} - a^{(t)} g \right]$$  \hspace{1cm} (4.11)

where $t$ is the current iteration, $a^{(t)} > 0$ is the step-size and $g$ is a subgradient of function $J_i$ at point $S_{[i]}^{(t)}$. $P_U$ is the euclidian projection onto the admissible set $U$. The meaning of this update rule is very simple: the obtained value its a combination between others estimations (averaged according to the assigned weights) and its local contribution (calculated by means of the subgradient of the local cost function). If all the assumptions are satisfied (connectivity rule, weight rule and convexity of the cost functions) then the controllers at the end of the iterative protocol will reach consensus on an approximation of the optimal value $S^{(opt)}$:

$$\lim_{t \to \infty} S_{[i]}^{(t)} \simeq S^{(opt)}$$

A very important parameter in this process is the step size. In fact it is strictly related to the convergence rate of the iterative protocol: smaller step sizes leads to better approximations of the optimal values, but require a higher number of iterations to reach it. If the optimal value is known in advance, it is possible to find a value of the step size that minimize the number of iterations [17]. Nevertheless, in general, the optimal solution is not known in advance hence proper trade-offs between accuracy and speed should be found empirically.


4.2.1 Weak sub-gradient calculus and its application

In this section we address the problem of calculating the sub-gradient of each cost function $J_i$. This is a crucial point, since control parts update their estimate according to its value, as shown in equation 4.11. A sub-gradient $g \in \mathbb{R}^n$ at $x \in \text{Dom}(f)$ is defined as:

$$f(z) \geq f(x) - g^T(z - x) \quad \forall \quad z \in \text{Dom}(f)$$  \hspace{1cm} (4.12)

If $f$ is not differentiable, then at point $x$ there can be more than one sub-gradient. The set of all sub-gradients at point $x$ is called sub-differential and is denoted by $\partial f(x)$. It is well known that finding all of them at any point is a hard task, and fortunately it is not required by the distributed sub-gradient method. We will exploit the weak sub-gradient calculus rules in order to calculate one sub-gradient of each cost function, which is enough for our scope. The basic rules are the following:

**Differentiable functions**  If $f$ is convex and $\partial f(x) = \{g\}$ (i.e. its sub-differential in $x$ is a singleton) then $f$ is differentiable and its sub-gradient is equal to its gradient at point $x$: $g = \nabla f(x)$.

**Non-negative scaling**  For $\alpha \geq 0$:

$$\partial(\alpha f(x)) = \alpha(\partial f(x))$$

**Summation**  Let $f(x) = f_1(x) + f_2(x) + \ldots + f_n(x)$, where $f_i(x)$ are convex functions, then we have that:

$$\partial f(x) = \partial f_1(x) + \partial f_2(x) + \ldots + \partial f_n(x)$$

where the $+$ sign corresponds to the Minkowski addition of two sets.

**Point-wise maximum**  Let us suppose that $f$ is defined as the point-wise maximum of $n$ differentiable convex functions:

$$f(x) = \max_{i=1,...,n} f_i(x)$$

We consider the set:

$$\mathcal{A}(x) = \{f_i(x) | f_i(x) = f(x)\}$$

composed by all the functions maximized in $x$ (active functions). Then the weak sub-gradient is given by the gradient of one function belonging to $\mathcal{A}(x)$.
4.2. Distributed optimization based on the sub-gradient method

In the following we will show as by applying these rules it is possible to obtain the sub-gradients of our objective functions in a simple way. For the sake of simplicity, we will start our analysis from a generic \( n \)-stages pipeline, and then we will extend the results to generic single-source acyclic graphs.

Sub-gradient calculation for pipeline graphs Consider an \( n \)-stages pipeline graph, as depicted in Figure 4.1. From Proposition 2.2.2 and Equation 4.9 we know that the \( i \)-th objective function has the following form:

\[
J_i(S,k) = \sum_{q=k}^{k+h-1} \alpha_i \cdot \max_{j=1,2,\ldots,N} \left\{ \frac{T_{calc_j(q)}}{n_j(q)} \right\} + \sum_{q=k}^{k+h-1} \beta_i \cdot \Delta_i(q) + \sum_{q=k}^{k+h-1} \gamma_i \cdot n_i(q) \quad (4.13)
\]

where the proportional coefficients are assumed fixed. We denote with \( g \in \mathbb{R}^{k \cdot N} \) the sub-gradient of \( J_i \) in \( S \). The entry \( g_{q,j} \) corresponds to the parallelism degree \( S_{q,j} = n_j(q) \). We now can proceed calculating each component of the sub-gradient. We have that, \( \forall q = k, k+1, \ldots, k+h-1 \):

\[
g_{q,i} = \begin{cases} 
\alpha_i \cdot \frac{\partial T_{S_i}(q)}{\partial n_i(q)} + \beta_i \cdot \frac{\partial \Delta_i(q)}{\partial n_i(q)} + \gamma_i \cdot \frac{\partial n_i(q)}{\partial n_i(q)} & (4.14a) \\
\beta_i \cdot \frac{\partial \Delta_i(q)}{\partial n_i(q)} + \gamma_i \cdot \frac{\partial n_i(q)}{\partial n_i(q)} & (4.14b)
\end{cases}
\]
Chapter 4. Accounting for reconfiguration stability: a new distributed control model

where Case 4.14a is selected if \( \max_{j=1,\ldots,n} \left\{ \frac{\text{Calc}_{j}(q)}{n_{j}(q)} \right\} = T_{S_{j}} \).

The sub-differentials are:

\[
\alpha_{i} \cdot \frac{\partial T_{S_{i}}(q)}{\partial n_{i}(q)} = -\alpha_{i} \cdot \frac{T_{\text{Calc}_{i}}(q)}{n_{i}(q)^{2}}
\]

\[
\beta_{i} \cdot \frac{\partial \Delta_{i}(q)}{\partial n_{i}(q)} = \begin{cases} 
\beta_{i} \left[ 2n_{i}(q) - 2n_{i}(q - 1) \right] & \text{if } q = h - 1 \\
\beta_{i} \left[ 4n_{i}(q) - 2n_{i}(q - 1) + 2n_{i}(q + 1) \right] & \text{otherwise}
\end{cases}
\]

\[
\gamma_{i} \cdot \frac{\partial n_{i}(q)}{\partial n_{i}(q)} = \gamma_{i}
\]

The other components are given by:

\[
g_{q,j} = \begin{cases} 
\alpha_{i} \cdot \frac{\partial T_{S_{j}}(q)}{\partial n_{j}(q)} = -\alpha_{i} \cdot \frac{T_{\text{Calc}_{j}}(q)}{n_{j}(q)^{2}} & \text{if } \max_{j=1,\ldots,n} \left\{ \frac{\text{Calc}_{j}(q)}{n_{j}(q)} \right\} = \frac{T_{\text{Calc}_{j}}(q)}{n_{j}(q)} \\
0 & \text{otherwise}
\end{cases}
\]

We stress the fact that, if in the point-wise maximum operator more than one function is active, we choose one between them in a non-deterministic way. Once the \( g \) has been constructed, it can be plugged directly in Equation 4.11 for update the local estimation. In the following we describe the generalization of the above for generic acyclic single-source graphs, applying the results shown in Chapter 2.

**Sub-gradient construction generalization** From an analytical point of view, the only difference between the cost functions of a pipeline graph and those of generic graphs, resides on the \( f_{i,j} \) functions, since in the latter, the topology dependent information in general are not equal to one:

\[
f_{i,j}(n_{j}(q)) = \frac{\text{Calc}_{-j}(q)}{n_{j}(q)} \sum_{\forall \pi \in P(S \rightarrow j)} \left( \prod_{\forall e \in \pi} e.p \right) \sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right) \]  \hspace{1cm} (4.15)

The tasks routing probabilities may not be known in advance, and furthermore they can change at run-time (in that case we treat them as disturbances). A simple method for taking into account these factors, without
4.2. Distributed optimization based on the sub-gradient method

Figure 4.2: Example of a graph composed by five ParMods.

affecting the sub-gradient construction, is to update the calculations time of each parallel module at the beginning of each control step, in the following way:

\[
T'_{\text{calc} - j}(q) = T_{\text{calc} - j}(q) \cdot \frac{\sum_{\forall \pi \in P(S \rightarrow j)} \left( \prod_{\forall e \in \pi} e.p \right)}{\sum_{\forall \pi \in P(S \rightarrow i)} \left( \prod_{\forall e \in \pi} e.p \right)} \quad (4.16)
\]

where \( j = 1, \ldots, N \) and \( q = k, \ldots, k + h - 1 \). Then it is sufficient to substitute each \( T_{\text{calc} - j}(q) \) with its corresponding \( T'_{\text{calc} - j}(q) \) in Equation 4.13. In this way it is like the graph has been “flattened” and the same sub-gradient construction defined for pipeline graphs can be applied.

Before introducing the cooperative algorithm, we should describe how disturbance inputs and interconnecting variables are defined.
Chapter 4. Accounting for reconfiguration stability: a new distributed control model

**Disturbance Inputs** Each ParMod control part maintains a matrix $D(k)$ defined as it follows:

$$D(k) = [D_1(k), D_2(k), \ldots, D_N(k)] = \begin{bmatrix} d_1(k) & \cdots & d_N(k) \\ d_1(k+1) & \cdots & d_N(k+1) \\ \vdots & & \vdots \\ d_1(k+h-1) & \cdots & d_N(k+h-1) \end{bmatrix}$$

in which the $i$-th column is the disturbance trajectory of ParMod $i$. The entry $d_i(k)$ consists of two real-valued components: the calculation time $T_{\text{calc}}-i(k)$ and the tasks routing probability vector $p_i$, which can be of variable dimension according to the node’s number of output data streams.

**Example.** In Figure 4.2 is reported an example of single-source acyclic graph. The task routing probability vector $p_i$ in this case is of dimension two, since both $p$ (i.e. the probability of sending tasks from ParMod$_1$ to ParMod$_2$) and $p^t$ (i.e. the probability of sending tasks from ParMod$_3$ to ParMod$_4$) can vary in time.

**Interconnecting Variables** These variables are used to exchange strategy profiles matrix estimations. We denote with $V_{\text{in-}i}(k) \in \mathbb{R}^{N \times h \times Nh(i)}$ the set of input interconnecting variables of node $i$. We can see it as a set of matrices in which the $j$-th element is the strategy profiles matrix estimation of neighbor $j$:

$$V_{\text{in-}i}(k) = \{S_{ij} | \forall j \in \text{neighbors of node } i \}$$

With

$$V_{\text{in-out}}(k) = S_{ij}$$

we denote the output interconnecting variables, consisting of the strategy profiles matrix estimation of ParMod $i$. We can finally describe the cooperative algorithm implemented by each control part $PC_i$. The procedure consists of two phases (Algorithm 2). In the first phase local disturbance trajectories must be produced and sent to other operating parts. Depending on the network topology this can involve more than one information exchange, hence a flooding algorithm can be used to disseminate data. Once all the control parts have acquired the necessary information, the actual distributed subgradient calculation can be started, and it will involve continuous estimation exchanges until a global termination condition is reached. At the end of the procedure control parts will issue the (possibly) new parallelism degree to their corresponding operating part.
4.2. Distributed optimization based on the sub-gradient method

Algorithm 2: Cooperative interaction protocol for single-source acyclic graphs based on the Distributed Subgradient Method.

1. **foreach control step** $k$ **each** PC$_i$ **do**
   2. $T_{calc-i}(k) = \text{Predictive\_Filter}(...)$;
   3. $p_i(k) = \text{Predictive\_Filter}(...)$;
   4. Send/Receive disturbance inputs from/to other PCs;
   5. $S^{(0)}_i(k) = \text{initial\_point}$;
   6. $t = 0$;
   7. **while** $t \neq \text{max\_iterations}$ **do**
      8. transmit the current estimate to neighbors;
      9. receive estimates from neighbors;
      10. calculate the subgradient $g_i$ of $J_i$ at point $S^{(t)}_i(k)$;
      11. calculate the new local estimate $S^{(t+1)}_i(k)$;
      12. $t = t + 1$;
   13. **end**
   14. use a rounding of the $i$-th component of $S^{(t)}_i(k)$ as the new parallelism degree for step $k$;

15. **end**

The number of iterations the algorithm must perform is specified by the constant `max_iterations`. In the literature it is known that the sub-gradient method is relatively slow, in the sense that it could require a high number of iterations in order to reach the convergence on the optimal solution. In our algorithm however, we can afford to reduce them consistently. This is due to two main factors:

1. we exploit an integer approximation of the real-valued parallelism degrees. In this perspective, we can be satisfied by a sub-optimal solution, since in any case we will round it to an integer value.

2. at step $k$, we use as a starting point $S^{(0)}_i$ for the sub-gradient method the strategy profiles matrix calculated at the previous step $k-1$ (*warm start* technique). Since we expect that optimal strategy profiles does not change so much between consecutive control steps, it is highly probable that the new optimal strategy profiles matrix is near the previous one. Hence reaching convergence requires a fewer number of iterations with respect to a arbitrary starting point.

47
4.2.2 Considerations on model assumptions

We should discuss two aspects of the proposed model. The first is the assumption that parallel modules scale perfectly, i.e. in isolation it holds:

\[ T_{s_i}(k) = \frac{T_{\text{calc}}(k) - i}{n_i(k)} \]

As we have seen, in order to find the optimal reconfiguration plan, each function must be convex in the parallelism degree. In this sense our hypothesis is non-restrictive: if we know the scalability behavior of the parallel computation implemented by a parallel module, we can replace the ideal one with that in the cost function. The only requirements to be satisfied is the function convexity. The second assumption regards the continuous relaxation of the optimization problem, in which we select real-valued parallelism degrees and then we apply a proper integer rounding. This choice has been made for tractability issues. In fact we consider distributed parallel computations in which the number of nodes available to parallel modules can be relatively high. Of course an integer approach is not feasible, since in that case the number of states to be explored is simply too large and it grows exponentially with the horizon length and with the number of ParMods. Considering this, the optimal solution that we obtain by exploiting the sub-gradient method is not so far from the integer one, and furthermore computing much less computationally expensive.

We are left with the problem of forecasting the values that disturbance variables will take in the future. In the next section we provide a possible solution to this problem, based on well know results from the field of time-series analysis.

4.3 The Holt-Winters forecasting method

As we have already said, at execution time, different model parameters can vary. We can model their evolution in time as a temporally ordered sequence of values, namely a time series. In general, the sequence of values taken by a variable can be a non-stationary process. In this kind of processes, the average value experience level movement in the in the upward or in the downward direction, namely trends (see [10] for an extensive treatise on this subject). In order to exploit the cooperative MPC approach, we need a method for predicting future time-series values for a reasonable number of step ahead. In this direction, several well known uni-
4.3. The Holt-Winters forecasting method

Various forecasting methods have been defined, where, given a single-variable time-series, predictions are based only on the current value and on its past history [6], [5]. In this thesis we exploit the so-called Holt-Winters procedure [25]. It is a simple (yet powerful) method that allows to forecast single-variable time series taking into account trends and seasonality. We denote with $Y = [y_0, y_1, \ldots, y_m, \ldots]$ the original series and with $Y_t$ the value taken at time $t$. With $X = [x_0, x_1, \ldots, x_m, \ldots]$ we denote the predicted time series and with $X_t$ the predicted value at time $t$. We will use a non-seasonal\footnote{Our time-series does not expose seasonal behaviors} Holt-Winters predictor. It is made by the combination of two filters, one for smoothing the average and one for forecasting the trend. The predicted value at time $t+1$ is given by the following difference equations:

\[
X_{t+1} = X_{t+1}^s + X_{t+1}^f \\
X_{t+1}^s = \alpha Y_t + (1 - \alpha)X_t^f \\
X_{t+1}^f = \beta(X_t^s - X_{t-1}^s) + (1 - \beta)X_{t-1}^f
\]

where (4.17b) is the smoothing component and (4.17c) is the trend component. Constants $\alpha$ and $\beta$ are the smoothing factors and range between zero and one. The forecasts obtained with this method can be extremely accurate in several cases. In the next chapter we will show two complete examples of time-series forecasting, along with their error estimate. In figure 4.3 it is shown an example of time-series forecasting, where the blue line represent

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{example_time_series}
\caption{Example of time-series forecasting.}
\end{figure}
the series of predicted values, whether the dotted black line represent the actual values. In order to make $m$ step-ahead predictions, Equation 4.17a is modified in the following way [9]:

$$X_{t+m} = X_{t+1}^s + mX_{t+1}^f$$ (4.18)

In words, to obtain the $m$-th future value, we shift forward the trend component of $m$ positions.
Chapter 5

Evaluating the impact of the switching cost on reconfiguration stability

In this last chapter we will show how considering a switching cost influences the reconfiguration choices of parallel modules. To do that, we will exploit some examples of application graphs in which parallel modules implement our control framework. Experiments are performed in a simulation environment based on the OmNet++ discrete event simulator. The simulations goal is to analyze from a qualitative point of view the reconfiguration decisions taken by the parallel modules involved in the computation. We first analyze the impact of the switching cost on the application stability, in terms of number and frequency of reconfigurations. A second point is to understand what stability implies in terms of task computed and resources utilization efficiency. In the first example, the examined graph is simple and each module has all the parameters set to the same value, in order to evict all the factors that can affect our analysis, and allowing us to focus only on the formulation properties. The second case instead, is a more realistic one, where we observe the parallel modules behavior in a heterogeneous scenario, where each one is executed in a different platform each one characterized by distinct parameter values. As we expect, we will see that introducing a cost proportional to the number of nodes allocated/deallocated, drastically decrease the number of reconfiguration. However in some cases, this could result in a reduced number of completed tasks and in efficiency degradation. We show how, by exploiting multiple-step ahead predictions of the application QoS, we can approach the same results in terms of task completed as if the switching cost where not considered, while at the same time reducing of a significant factor the number of reconfiguration and augmenting in general
5.1 A simulation environment for ParMod graphs

In this section we will describe the simulation environment 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 communication 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 reproduces the behavior of a generic ParMod, i.e. the basic building block of our approach for composing adaptive distributed applications.

5.1.1 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 5.1.

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

---

1I have to thank Daniele Buono and Gabriele Mencagli for his crucial help in the extension of the simulation environment.
5.1. A simulation environment for ParMod graphs

Figure 5.1: Adaptive ParMod simulated through an OMNeT module.

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 5.2: 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 5.2. In other words the operating part is simulated through a queueing station. Tasks from other operating parts are received
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

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 \texttt{SEND} and \texttt{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;

- 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;
5.1. A simulation environment for ParMod graphs

- 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.

![Diagram](image)

**Figure 5.3:** 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;
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

- **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;

- the magnitude of the parallelism degree change with respect to the previous control step;
5.2. An example of functional partitioning with three parallel modules

- the average number of consecutive control step for which a parallelism degree remains constant;
- 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.

5.2 An example of functional partitioning with three parallel modules

We will start our analysis from a simple scenario. In this example we will considered a graph composed by three parallel modules, with the following organization:

- $ParMod_1$ is in charge of generating a stream of tasks in which elements can have two different types, let us call them $t_1$ and $t_2$. With probability $p$ it sends tasks to $ParMod_2$ and with probability $1 - p$ to $ParMod_3$. The second parallel module is in charge of processing elements of type $t_1$, whether the third parallel module processes elements of type $t_2$;
- task routing probabilities reflects the occurrence of a certain type: if $p = \frac{1}{3}$, it means that on average, in the stream generated by the first parmod one task out of three is of type $t_1$;
- as defined in the previous chapters control parts exchange control information by means of bidirectional data streams, following the operating parts interconnection scheme, as depicted in Figure 5.4.

In order to simplify the analysis and for the sake of a more intuitive explanation, we will made the assumption that probability $p$ is the only system parameter that can vary in time. Each parallel module will have a fixed mean calculation time $T_{calc-i}$. We recall here the local cost function formulation, the defined as it follows:

$$J_i(s, k) = \sum_{q=k}^{k+h-1} \alpha_i \cdot T_{p_i}(s, q + 1) + \sum_{q=k}^{k+h-1} \beta_i \cdot \Delta_i(q) + \sum_{q=k}^{k+h-1} \gamma_i \cdot n_i(q)$$

We will made the following assumptions:

- parameters $\alpha_i$, $\beta_i$ and $\gamma_i$ are fixed, their value does not change in time;
- we will consider a fixed horizon of length $h$ for the entire execution duration.
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

The application QoS goal is to maximize performances. It can be expressed by setting the cost function weights in the following way: $\alpha_i \gg \gamma_i \gg \beta_i$. Parallel modules implement the distributed procedure defined in Algorithm 2, summarized by the following point:

1. At the beginning of control step $k$, $PC_1$ will acquire from its operating part the measurements of the value assumed by the probability $p$ during step $k-1$;

2. based on the past observation, it will make a statistical forecast of what possible value probability $p$ will assume in the future, for a prediction horizon of length $h$;

3. it sends to control parts $PC_2$ and $PC_3$ the predicted values, and starts the iterative protocol described in Algorithm 2;

4. after a fixed number of iterations, the distributed procedure ends and each control part will issue the parallelism degree for step $k$ to their operating part.

We will consider a control step of length 120\(t\) and an execution of 600 control steps. In this context we use as time unit the standardized time unit $t$. In OMNet++ the concept of time is abstract, hence $t$ can be instantiated to different time units in order to have different time-scales (e.g. milliseconds, seconds or hours). In the following sections, we will discuss several aspects.
5.2. An example of functional partitioning with three parallel modules

In a first instance we will show how the predictive filters perform, showing the mean absolute error in predicting the probability time-series values. Then we will discuss the effects in term of stability that a model with a switching cost can induce. Furthermore will be given a perspective on what stability implies in terms of application performance level and resources exploitation efficiency.

5.2.1 Time series forecasting errors

We are interested in understanding the reliability of the Holt-Winter filters in terms of the average prediction error. In Figure 5.5 is shown the evolution of the probability, observed for 600 time steps. In this series, several trends can be spotted. From step zero to step 200 we can observe an uncertain phase where the average value floats between 0.2 and 0.4. From steps 200 to 250 we can see a sudden upward movement that brings the probability from 0.2 to 0.6. From step 250 to step 350 the growing trend become linear till it stabilizes around probability 0.9. We will measure the level of accuracy of our forecasts utilizing the Mean Absolute Percentage Error (MAPE) criterion, defined as it follows:

\[ MAPE = \frac{100}{n} \cdot \sum_{i=0}^{n} \left| \frac{Y_i - X_i}{Y_i} \right| \]  

Figure 5.5: Probability evolution for 600 time steps.

Since we consider a prediction horizon of length \( h \), we are interested on evaluating at each control step the mean error as a function of the horizon length. Hence, we calculate at step \( k \) the error for the entire prediction trajectory:

\[ Error(k) = \frac{100}{n} \cdot \sum_{i=k}^{k+h-1} \left| \frac{Y_i - X_i}{Y_i} \right| \]
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

The mean absolute error for the entire computation duration (global error estimation) is obtained by averaging the error estimates of each control step:

\[ Error_{tot} = \frac{1}{n} \sum_{i=0}^{n} Error(k) \]  

(5.3)

In Figure 5.6 are reported the histograms relative to the mean absolute percentage error per step for various horizon length.

(a) Forecasts errors for horizon of length 1.

(b) Forecasts errors for horizon of length 2.

Figure 5.6: Mean-Absolute Percentage Error per step - Horizon 1 and 2

We can notice that the largest errors occurs in the last part of the simulation. This is due to two main factors. The first is that Holt-Winter filters need some time to react to trend changes. In fact, in the interval between steps 350-450, predicted values reach peaks of 80% of difference with respect to the original value. The second is that, even if the series stabilizes on a fixed mean, it expose a certain amount of variance that severely affects our predictions. We can observe that as we exploit a longer horizon, the number of error peaks decrease. This is an effect of the averaging, since even if peaks
5.2. An example of functional partitioning with three parallel modules

![Graph of mean absolute percentage error per step - Horizon 3](image)

(c) Forecasts errors for horizon of length 3.

![Graph of mean absolute percentage error per step - Horizon 4](image)

(d) Forecasts errors for horizon of length 4.

Figure 5.6: Mean-Absolute Percentage Error per step - Horizon 3 and 4

<table>
<thead>
<tr>
<th>Error Percentage</th>
<th>Horizon 1</th>
<th>Horizon 2</th>
<th>Horizon 3</th>
<th>Horizon 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8.92%</td>
<td>9.22%</td>
<td>9.49%</td>
<td>9.6884%</td>
</tr>
</tbody>
</table>

Table 5.1: Global error percentage

are still there, they are smoothed by the averaging on the horizon length. The global error as expected grows as we increase the horizon length, and that it is represented graphically by the denser lines in graphs 5.17c and 5.17d. In Table 5.1 are reported the numerical values of global error estimates.

5.2.2 Effectiveness of a multiple-step ahead approach: performance versus stability

In this section we study what benefits in terms of stability will bring our formulation. We will compare it with a formulation that does not take into account a cost proportional to the parallelism degree switching. It is defined
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

as it follows:

\[ J_i(s, k) = \alpha_i \cdot T_{p_i}(s, k + 1) + \gamma_i \cdot n_i(k) \]

and we refer it as no-switching cost function. Notice that it cannot take advance of horizons greater than one, since each step is independent from the others and we apply only the first parallelism degree of the trajectory. We denote with \( p(k) \) the probability of sending tasks to \( ParMod_3 \) at step \( k \). The inter-departure times from each parallel modules are given by the following equations:

\[
T_{p_1}(k + 1) = \max \left\{ \frac{T_{calc-1}}{n_1(k)} , \frac{T_{calc-2}(1-p(k))}{n_2(k)} , \frac{T_{calc-3}p(k)}{n_3(k)} \right\}
\]

\[
T_{p_2}(k + 1) = \max \left\{ \frac{T_{calc-1}}{n_1(k)(1-p(k))} , \frac{T_{calc-2}}{n_2(k)} , \frac{T_{calc-3}p(k)}{n_3(k)(1-p(k))} \right\}
\]

\[
T_{p_3}(k + 1) = \max \left\{ \frac{T_{calc-1}}{n_1(k)p(k)} , \frac{T_{calc-2}(1-p(k))}{n_2(k)p(k)} , \frac{T_{calc-3}}{n_3(k)} \right\}
\]

The model parameters are set as shown in Table 5.2. As we have already hinted, except from the calculation time each parallel module has the parameter equal to the others. We have imagined a scenario in which the source \( (ParMod_1) \) is twice as fast the destinations, whether \( ParMod_2 \) is slightly faster than \( ParMod_3 \). Given this setting, we expect the series of parallelism

<table>
<thead>
<tr>
<th></th>
<th>ParMod 1</th>
<th>ParMod 2</th>
<th>ParMod 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_{calc}</td>
<td>15t</td>
<td>30t</td>
<td>40t</td>
</tr>
<tr>
<td>( \alpha )</td>
<td>10</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>( \beta )</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
</tr>
<tr>
<td>( \gamma )</td>
<td>3.5</td>
<td>3.5</td>
<td>3.5</td>
</tr>
</tbody>
</table>

Table 5.2: First experiment: model parameters
degree adopted by \( ParMod_3 \) to follow strictly the probability trend. In fact, low values of \( p \) corresponds to a low workload for the third parallel module. If the probability experiences a decreasing trend, it will results in a phase in which \( ParMod_3 \) release resource that are no more necessary. In the opposite case, \( ParMod_2 \) will acquire resources in order to overcome the increasing number of requests. As an example, in the interval between step 200 to step 300, \( ParMod_3 \) will be particularly eager of resources, in order to face the sudden increment in task arrivals. We can see it graphically in Figure 5.7. From plots 5.7b and 5.7c we can notice that the two parallel modules expose a specular behavior: the phases in which \( ParMod_3 \) acquire resources corresponds to resource releasing by \( ParMod_2 \). It is evident that the amount
5.2. An example of functional partitioning with three parallel modules

(a) Parallelism degrees used by ParMod₁.

(b) Parallelism degrees used by ParMod₂.

(c) Parallelism degrees used by ParMod₃.

Figure 5.7: Reconfigurations of the three parallel modules without considering the switching cost

of reconfiguration is high. Even in phases in which the probability is stable on a certain mean (steps from 350 to 600) we can observe a large number of reconfiguration of little magnitude (parmods allocates/deallocates a single node).
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

Figure 5.8: Reconfigurations of ParMod1 considering the switching cost
5.2. An example of functional partitioning with three parallel modules

Figure 5.9: Reconfigurations of ParMod$_2$ considering the switching cost
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

Figure 5.10: Reconfigurations of ParMod₃ considering the switching cost
5.2. An example of functional partitioning with three parallel modules

We now consider the parallel modules behavior in the case in which they optimize the objective function with the switching cost (Equation 5.2) for different horizon length $h = 1, 2, 3, 4$. In Figure 5.7b are depicted the sequences of parallelism degree that ParMod$_2$ uses throughout the computation, together with the series obtained without taking into account the switching cost as a reference. It is possible to see as the function trend is much smoother with respect to the previous case. As we consider longer horizons instead, the sequence of parallelism degrees tends to the one obtained optimizing the cost function without switching cost. We can observe the same phenomenon in ParMod$_2$ (Figure 5.9) and ParMod$_3$ (Figure 5.10). This is due to the fact that the switching cost acts as a disincentive to parallelism degree variations. In workload descending phases, it brakes the release of resources, whether in phases in which the workload increases it brakes the resources allocation. By considering a longer prediction horizon we can mitigate this effect, and as we will see shortly, approach the same performance level of the previous function, saving a consistent number of reconfiguration. It is interesting to see in which period of the computation reconfigurations occur most.

Figure 5.11: Reconfiguration magnitude histogram
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

Figure 5.11: Reconfiguration magnitude histogram

In figure 5.11 we can observe the histograms of the reconfiguration magnitude (i.e. the number of nodes allocated/deallocated) of the third parallel module\(^2\). Each histogram is denser between steps 200-250, meaning that in that period ParMod\(_3\) changes continuously its parallelism degree. This is coherent with the probability trend, since in that interval experience an abruptly change. We can notice however, that in the histogram related to the no-switching cost formulation (Figure 5.11a) are almost equally distributed for the entire observed period. On the contrary, the others expose an evident stabilization from step 350, which coincides with the probability horizontal trend. Hence we can say that the switching cost acts also as a stabilizer in presence of disturbances with a consistent variance.

\(^2\)For brevity we report only the histograms regarding ParMod\(_3\)
5.2. An example of functional partitioning with three parallel modules

We are also interested in observing what effects taking into account the switching cost has on the parallel modules utilization factor $\rho$. We recall that it is a relative parameter that gives an insight on how the resources allocated are exploited. It is the ratio between the module ideal behavior (i.e. the mean service time that it can achieve with a given parallelism degree) and its actual behavior (i.e. its inter-departure time at steady state). An utilization factor lower than one, means that resources are under-utilized. On the contrary, values of $\rho$ greater or equal then one imply that the node is fully exploiting the resources allocated. An ideal situation is to have all the utilization factors as much close to one as possible, so that all the resources allocated are entirely utilized.

In Figures 5.12, 5.13 and 5.14 are depicted the utilization factor histograms of the three parmods. Let us consider the histograms relative to $\text{ParMod}_2$ (Figure 5.13). If we look at the histogram relative to a prediction horizon equal to one, we can see a severe efficiency degradation in the interval between steps 200 and 500. Observe that in that period $\text{ParMod}_2$ is deallocating resources. As already said, the switching cost acts as a brake, causing a slower resources release. As a consequence, the second parallel module is under-utilized. However, in this case considering a longer prediction horizon brings evident benefits: in fact as $h$ grows, the utilization factor envelope tends to one, as we can appreciate from histograms 5.13b, 5.13c and 5.13d. If we consider the first and the third parallel modules instead, their utilization factor is slightly influenced by the switching cost. In fact both of them, exhibits a crescent trend in the parallelism degree sequence and, in these cases, the utilization factor remains close to one. Notice that, on average, $\rho$ is less than one, and this is mainly due to two factors. A first factor are prediction errors, specifically the predicted probability induces an inter-arrival time smaller than the actual one, causing unnecessary nodes allocations. The second factor is due to the rounding of the parallelism degree, that make it slightly different from the optimal real value.
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

Figure 5.12: ParMod_1 utilization factor histograms

Figure 5.13: ParMod_2 utilization factor histograms

Figure 5.14: ParMod_3 utilization factor histograms
5.2. An example of functional partitioning with three parallel modules

We are now ready to show some quantitative results. We introduce an index that quantify the duration of a reconfiguration:

**Definition 5.1.** *We denote as Mean Stability Index (MSI) the average of number of steps for which a reconfiguration remains active, i.e. no changes in the parallelism degree occurs.*

In Table 5.3 are reported the global simulation results, where the number of reconfigurations is the sum of the individual partials of the three parallel modules, whether the completed tasks is the number of task that leaves the systems (i.e. the sum of the tasks processed by *ParMod*$_2$ and *ParMod*$_3$). The global mean stability index is the average of the individual MSIs. We can see how the formulation without switching cost induces 789 reconfigurations in order to complete 207,863 tasks. Using the formulation with the switching cost and by considering a prediction horizon of length one, parallel modules are able to compute in total 184,399 tasks with 162 total reconfigurations, which is an 11% less of the tasks computed in the previous case, but saving a 79% of reconfigurations. If we consider a longer prediction horizon we can reach even more satisfying trade-offs from the performance viewpoint. With a horizon equal to two, we are able to compute a 1% reduced number of tasks, but decreasing the reconfiguration numbers of a 48% with respect to the no-switching cost formulation. Also the *mean stability index* gives us insight on the better reconfiguration stability. In fact we can reach values up to 10.62 (horizon 1), meaning that a given configuration on average remains stable for nine consecutive control steps. At this regard we can appreciate Table 5.4 in which individual results for the MSI are reported. Notice that consistent results in terms of stability are achieved by *ParMod*$_2$, that passes from an MSI of 1.64 to values up to 11.85 (horizon 1).

<table>
<thead>
<tr>
<th>Reconfigurations number</th>
<th>MSI</th>
<th>Tasks Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Switching Cost</td>
<td>789</td>
<td>3.20</td>
</tr>
<tr>
<td>Horizon 1</td>
<td>162</td>
<td>10.62</td>
</tr>
<tr>
<td>Horizon 2</td>
<td>287</td>
<td>6.19</td>
</tr>
<tr>
<td>Horizon 3</td>
<td>346</td>
<td>5.64</td>
</tr>
<tr>
<td>Horizon 4</td>
<td>408</td>
<td>5.18</td>
</tr>
</tbody>
</table>

*Table 5.3: Experiment 1 global results*

5.2.3 On the approach feasibility

As we have already mentioned, the iterative protocol based on the distributed sub-gradient method exploited by the parallel modules could be
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

<table>
<thead>
<tr>
<th>ParMod 1</th>
<th>ParMod 2</th>
<th>ParMod 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Switching Cost</td>
<td>6.16</td>
<td>1.64</td>
</tr>
<tr>
<td>Horizon 1</td>
<td>11.32</td>
<td>11.85</td>
</tr>
<tr>
<td>Horizon 2</td>
<td>7.09</td>
<td>5.51</td>
</tr>
<tr>
<td>Horizon 3</td>
<td>8.18</td>
<td>4.06</td>
</tr>
<tr>
<td>Horizon 4</td>
<td>8.55</td>
<td>3.42</td>
</tr>
</tbody>
</table>

Table 5.4: Experiment 1 Mean Stability Index per ParMod

rather slow, i.e. a high number of iterations could be required to compute the optimal value. In our case, this is not a problem for two reasons:

- we exploit a warm start technique for the initial point: the starting point for the iterative protocol at step $k$, is set equal to the optimal reconfiguration plan computed at step $k - 1$. This technique allows to reduce consistently the number of iterations required to reach an optimal solution, since, in the considered scenarios, it is highly likely that between consecutive control steps, optimal solutions are close.

- we use integer approximations of the real-valued optimal parallelism degrees. In this perspective, it is useless to search for extremely accurate real-valued solutions, since in any case they will be approximated to an integer.

In order to prove this, we performed a series of tests in which the iterative algorithm where set to perform a different amount of iterations, starting from 125 till 1500. In Tables 5.6 and 5.5, are reported respectively the cost of the entire computation and the number of reconfigurations as a function of the number of iterations and of the considered horizon. We can state that even performing much more iterations, the advantages in terms of quality of the solution are negligible (and in some cases, completely absent). For instance, if we consider the difference in terms of total cost between an execution with 1500 iterations and one with 150 with an horizon of length four, the difference is of a 0.1%. This confirms that an approach based on this method is feasible in many practical scenario, considered that few tens of iterations give very good results.

5.2.4 Conclusions

It is clear that considering a switching cost brings important improvements in terms of reconfiguration stability. A compelling feature of a formulation such that of Equation 5.2, is the possibility of fine-tuning the system,
5.2. An example of functional partitioning with three parallel modules

<table>
<thead>
<tr>
<th></th>
<th>1500 iter</th>
<th>750 iter</th>
<th>375 iter</th>
<th>150 iter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a = 0.005$</td>
<td>$a = 0.05$</td>
<td>$a = 0.05$</td>
<td>$a = 0.05$</td>
</tr>
<tr>
<td>No Switching</td>
<td>789</td>
<td>801</td>
<td>787</td>
<td>786</td>
</tr>
<tr>
<td>Horizon 1</td>
<td>162</td>
<td>162</td>
<td>158</td>
<td>162</td>
</tr>
<tr>
<td>Horizon 2</td>
<td>287</td>
<td>281</td>
<td>295</td>
<td>289</td>
</tr>
<tr>
<td>Horizon 3</td>
<td>346</td>
<td>344</td>
<td>344</td>
<td>340</td>
</tr>
<tr>
<td>Horizon 4</td>
<td>408</td>
<td>410</td>
<td>410</td>
<td>381</td>
</tr>
</tbody>
</table>

Table 5.5: Reconfigurations with fewer sub-gradient iterations

<table>
<thead>
<tr>
<th></th>
<th>1500 iter</th>
<th>750 iter</th>
<th>375 iter</th>
<th>150 iter</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$a = 0.005$</td>
<td>$a = 0.05$</td>
<td>$a = 0.05$</td>
<td>$a = 0.05$</td>
</tr>
<tr>
<td>Horizon 1</td>
<td>57,125</td>
<td>57,152</td>
<td>57,148</td>
<td>57,112</td>
</tr>
<tr>
<td>Horizon 2</td>
<td>54,798</td>
<td>54,811</td>
<td>54,804</td>
<td>54,798</td>
</tr>
<tr>
<td>Horizon 3</td>
<td>53,794</td>
<td>53,790</td>
<td>53,790</td>
<td>53,819</td>
</tr>
<tr>
<td>Horizon 4</td>
<td>53,497</td>
<td>53,496</td>
<td>53,484</td>
<td>53,564</td>
</tr>
</tbody>
</table>

Table 5.6: Computation cost as a function of the iteration number

in order to obtain the desired trade-offs between completed tasks, number of reconfigurations and resources utilized. We have seen how considering a longer horizon, brings performance improvement and a reduced stability, i.e. the observed statistics tend to the statistics of the no-switching cost formulation. This is shown graphically in plots 5.8, 5.9 and 5.10, where we can see that, as the prediction horizon become longer the sequence of parallelism degree become similar to the one obtained without a switching cost. Hence, we underline that depending on the QoS goals of the computation, different solution can be choose. In this example for instance, we can individuate in the switching cost formulation with a prediction horizon of length three a balanced solution, that gives satisfactory results from performance, efficiency and stability point of view.
5.3 Stability in a heterogeneous scenario: a second example

In the first experiment, we have shown how considering a switching cost in taking reconfiguration decisions impacts on the distributed computation performance and stability. In this experiment instead, we want to stress once more the benefits that can bring multiple-step ahead predictions provided good forecasts of disturbance values. We will consider an application graph in which modules are distributed among four different execution platforms, as shown in Figure 5.15. The stream source is sequential, i.e. it will not exploit any reconfiguration at run-time. The computation QoS goal is again to maximize performances, but in this example to each platform will be associated different values for model parameters, in a way that reflect their local resources and switching costs. In this scenario we will consider as variable parameter the mean calculation time of the sequential source. It is clear that also in this case, parallel modules will need to adjust their parallelism degree in order to respond to the source dynamicity.

In Table 5.7 are reported the values assigned to system model parameters. The $\alpha_i$ value is dominant with respect to others, and we can notice that $ParMod_1$ and $ParMod_3$ have all the parameters set to the same values: they are executed in the same platform. In this experiment, we want to stress
again the importance of adaptiveness. In this regard, beside the comparison between the model without switching cost and our formulation, we introduce a third strategy in which modules do not reconfigure and use for the entire computation their maximum parallelism degree $n_{\text{max}}$. We will see that, as already pointed out in [16], in terms of efficiency there is a consistent advantage on exploiting dynamic reconfiguration.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
Source & ParMod 1 & ParMod 2 & ParMod 3 & ParMod 4 \\
\hline
$T_{\text{calc}}$ & - & 90t & 20t & 90t & 22t \\
$\alpha$ & 10 & 10 & 10 & 10 & 10 \\
$\beta$ & - & 4.5 & 1.5 & 4.5 & 2.8 \\
$\gamma$ & 0.5 & 0.3 & 0.8 & 0.3 & 0.4 \\
$n_{\text{max}}$ & 1 & 50 & 15 & 55 & 20 \\
\hline
\end{tabular}
\caption{Second experiment: model parameters}
\end{table}

The inter departure times from each module are given by the following equations:

\begin{align*}
T_{p_1}(k + 1) &= \max \left\{ T_{\text{calc}}(k), \frac{T_{\text{calc}}(k)}{n_1(k)}, \frac{T_{\text{calc}} - 1}{n_2(k)}, \frac{T_{\text{calc}} - 2}{n_3(k)}, \frac{T_{\text{calc}} - 3}{n_4(k)} \right\} \\
T_{p_2}(k + 1) &= \max \left\{ T_{\text{calc}}(k), \frac{T_{\text{calc}} - 1}{n_1(k)}, \frac{T_{\text{calc}} - 2}{n_2(k)}, \frac{T_{\text{calc}} - 3}{n_3(k)}, \frac{T_{\text{calc}} - 4}{n_4(k)} \right\} \\
T_{p_3}(k + 1) &= \max \left\{ T_{\text{calc}}(k), \frac{T_{\text{calc}} - 1}{n_1(k)}, \frac{T_{\text{calc}} - 2}{n_2(k)}, \frac{T_{\text{calc}} - 3}{n_3(k)}, \frac{T_{\text{calc}} - 4}{n_4(k)(1 - p)} \right\} \\
T_{p_4}(k + 1) &= \max \left\{ T_{\text{calc}}(k), \frac{T_{\text{calc}} - 1}{n_1(k)}, \frac{T_{\text{calc}} - 2}{n_2(k)}, \frac{T_{\text{calc}} - 3}{n_3(k)}, \frac{T_{\text{calc}} - 4}{n_4(k)} \right\}
\end{align*}

In Figure 5.16 is reported the sequence of values assumed by the mean calculation time of the source. In this case predictive filters behaves even better than in the previous experiment. In Table 5.8 are reported the global error estimates for the various horizon length. Besides trends, other types of irregularities that can heavily affect predictions are level shifts and outliers, where the first is a sudden change in the series mean and the second is a measurement significantly different from the closely observed values that can affect the prediction statistics [11]. The considered calculation time series does not expose neither of these non-stationarities, and furthermore
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

Figure 5.16: Mean calculation time of the Source.

is characterized by smooth trends. In fact the predictions are extremely accurate, with an average error of 3%, as shown in Table 5.8, and in Figure 5.17 we can see that even error peaks do not exceed the 20%.

<table>
<thead>
<tr>
<th>Error Percentage</th>
<th>Horizon 1</th>
<th>Horizon 2</th>
<th>Horizon 3</th>
<th>Horizon 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2.64%</td>
<td>2.77%</td>
<td>2.84%</td>
<td>2.93%</td>
</tr>
</tbody>
</table>

Table 5.8: Global error percentage

(a) Forecasts errors for horizon of length 1.

(b) Forecasts errors for horizon of length 2.

Figure 5.17: Mean-Absolute Percentage Error per step - Horizon 1 and 2
5.3. Stability in a heterogeneous scenario: a second example

In Figures 5.18 and 5.19 are reported the parallelism degrees series for the four ParMods for various prediction horizon length. In this example, we can observe that from a certain point on, parallel modules do not reconfigure anymore, and stabilize their parallelism degree on a certain value. This phenomenon happens as a consequence of ParMod$_4$ behavior. In fact, since the graph source decreases its service time, all the parallel modules need to acquire resources in order to face the growing workload. In this way, ParMod$_4$ reaches its maximum parallelism degree: he can not acquire any other resources, and as a consequence become the graph bottleneck. This results in a stabilization of the system, since other parallel modules adjust their parallelism degree in order to adapt to the service time of ParMod$_4$. In the first part of the computation instead, we can appreciate once again the fact that, as we consider a longer prediction horizon, the sequence of parallelism degrees of each parallel module tends to the sequence obtained without considering the switching cost. We have already mentioned that the switching cost acts as a stabilizer: if a reconfiguration is not strictly necessary, it will not be issued. This is an important aspect of our model. The parallelism degree obtained by the distributed optimization algorithm proposed in the previous chapter are real valued, requiring an integer approximation. In some unfortunate cases, it could happen that due to the sub-optimality of the approach, some parallelism degrees tend to oscillate between two consecutive values due
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

to the integer rounding. In this case, the switching cost can strongly moderate this undesired effect, as we can appreciate in Figure 5.19. In the first fifty steps, even if the source calculation time has a stable mean, ParMod\textsubscript{3} continuously switches its parallelism degree between twenty and twenty-one when it optimizes the formulation without a switching cost. In the case in which a switching cost is considered instead, a lot of this reconfiguration are spared, meaning that they where not strictly necessary from a performance point of view.

![Graphs showing ParMod\textsubscript{1} and ParMod\textsubscript{2} parallelism degree evolution throughout the computation.](image)

Figure 5.18: Reconfigurations of ParMod\textsubscript{1} and ParMod\textsubscript{2} with the switching cost.
5.3. Stability in a heterogeneous scenario: a second example

![Graphs showing ParMod3 and ParMod4 parallelism degree evolution throughout the computation for different horizons](image)

**Figure 5.19:** Reconfigurations of ParMod$_3$ and ParMod$_4$ with the switching cost.

In Figure 5.20 are reported the efficiency plots of the sequential source. We can see as from step 300 it starts to decrease. In fact, if we look at its $T_{calc}$ time-series, we can see that from that point on its inter-departure time becomes smaller than the minimum service time that ParMod$_4$ can achieve. As a consequence, it starts to block and an efficiency degradation occurs. The efficiency of the other modules instead, is always very close to one since they choose the parallelism degree that allows to match their service time to their maximum parallel degree for the entire computation, without making any reconfiguration. It is evident that in this case, this kind of strategy brings...
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

![Graphs showing Source Utilization factor for different horizons](image1)

(a) Horizon 1.

(b) Horizon 2.

(c) Horizon 3.

(d) Horizon 4.

**Figure 5.20:** Efficiency of the Source node.

half of the allocated resources are wasted. ParMod 3 Utilization factor - Max Strategy

No switching Cost

Source Utilization factor - Horizon 1

Horizon 1

Source Utilization factor - Horizon 2

Horizon 2

Parmod 3 Utilization factor - Max Strategy

No switching Cost

Source Utilization factor - Horizon 3

Horizon 3

Source Utilization factor - Horizon 4

Horizon 4

Parmod 4 Utilization factor - Max Strategy

No switching Cost

![Graphs showing Parmod Utilization factor for different modules](image2)

(a) ParMod1.

(b) ParMod2.

(c) ParMod3.

(d) ParMod4.

**Figure 5.21:** Efficiency of the parallel modules when they exploit the maximum parallelism degree.

As an example, ParMod2 5.21b for a large part of the computation is under-utilized, and half of the allocated resources are wasted. In Table 5.9 are reported the global simulation results. A first interesting aspect comes from the number of computed tasks. As we expect, when each module in the graph is set to use its maximum parallelism degree, the number of tasks completed is greater w.r.t the case in which adaptiveness is exploited. Anyway the difference is not big: if we consider the no-switching
cost formulation, only the 8% less of tasks has been computed, but much benefits has been earned in efficiency. Also in this case, we can see as, by increasing the prediction horizon, the number of tasks computed rises together with the number of reconfiguration enacted. With respect to the previous experiment, parallel modules reconfigure much less, since there is a large part of the computation in which their behavior is influenced by the service time of \textit{ParMod}_4. At this regard, the mean stability index exposes a different behavior. Considering a horizon of length one, reconfigurations are twice as stable w.r.t. the no-switching cost case. As expected, as we consider longer horizons its value decrease (14.08 and 13.02 for horizon two and three respectively), but using a horizon equal to four it suddenly increase, reaching values even better then shorter horizons. The explanation is very simple: as we have already seen, in resource allocation phases (i.e. a phase in which at each step a parallel module increases its parallel degrees) considering a longer prediction horizon mitigates the brake effect of the switching cost. In this example as a consequence, \textit{ParMod}_4 reaches its maximum parallelism degree in less time, and stops to reconfigure. If we look at Table 5.10, we can see that with a horizon equal to three he as a mean stability index of 18.74, whether with horizon equal to four a mean stability index of 23.27. It is evident as in this kind of scenarios, considering a longer prediction horizon brings great benefits. Reconfiguration are much less and more stable with respect to the no-switching cost formulation and furthermore, only the 0.5 % less of tasks are computed.

<table>
<thead>
<tr>
<th>Max Strategy</th>
<th>Reconfigurations number</th>
<th>MSI</th>
<th>Tasks Completed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max Strategy</td>
<td>296</td>
<td>10.84</td>
<td>99,246</td>
</tr>
<tr>
<td>No Switching Cost</td>
<td>135</td>
<td>20.92</td>
<td>95,020</td>
</tr>
<tr>
<td>Hor 1</td>
<td>166</td>
<td>14.08</td>
<td>97,784</td>
</tr>
<tr>
<td>Hor 2</td>
<td>177</td>
<td>13.02</td>
<td>98,470</td>
</tr>
<tr>
<td>Hor 3</td>
<td>181</td>
<td>16.02</td>
<td>98,783</td>
</tr>
</tbody>
</table>

\textit{Table 5.9:} Experiment 2 global results
Chapter 5. Evaluating the impact of the switching cost on reconfiguration stability

<table>
<thead>
<tr>
<th>No Switching Cost</th>
<th>ParMod 1</th>
<th>ParMod 2</th>
<th>ParMod 3</th>
<th>ParMod 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Horizon 1</td>
<td>11.90</td>
<td>23.79</td>
<td>11.90</td>
<td>13.97</td>
</tr>
<tr>
<td>Horizon 2</td>
<td>7.16</td>
<td>20.83</td>
<td>7.16</td>
<td>10.18</td>
</tr>
<tr>
<td>Horizon 3</td>
<td>6.25</td>
<td>18.74</td>
<td>6.04</td>
<td>9.57</td>
</tr>
<tr>
<td>Horizon 4</td>
<td>5.71</td>
<td>23.27</td>
<td>5.71</td>
<td>8.90</td>
</tr>
</tbody>
</table>

Table 5.10: Experiment 1 Mean Stability Index per ParMod

5.3.1 Considerations on performances and stability

As we have said in Section 5.1.1, both of the presented experiments have been performed with the OMNeT++ simulation environment. At this regard, we recall that due to the simulation context parallel module behavior has been simplified in several aspects. One of them is that when a reconfiguration is issued, it does not induces any overhead, i.e. changing the number of nodes that a parallel module can exploit does not induce performance penalties. In a real scenario often this is not the case. Performing reconfiguration could bring performance degradation, since parallel modules could be blocked waiting for the allocation/deallocation process to complete. In this cases, considering a switching cost, and hence avoiding unnecessary reconfigurations, could bring even more benefits, allowing to exceed in terms of task computed formulations that do not take into account reconfiguration overheads.
Chapter 6

Conclusions

The aim of this thesis was to provide mechanisms to ensure reconfiguration stability in distributed parallel computations. In our model, each parallel module being part of a complex application graph possesses the necessary resources for taking reconfiguration decisions. We exploit the Model-based Predictive Control technique, in which multiple-step ahead prediction of the graph behavior are used in order to make long-term reconfiguration plans. In our model, reconfigurations are driven by QoS requirements. As we have shown, a proper QoS modeling is necessary in order to obtain satisfying results in terms of performance and stability.

At this regard, in Chapter 4 we have proposed a QoS modeling for considering the costs that a reconfiguration phase could induce. It has been introduced a quantitative metric, namely a switching cost, proportional to the variation of the parallelism degree between consecutive control steps. In order to compute the optimal reconfiguration trajectory, parallel modules exploited the distributed sub-gradient method, which ensure the optimality of the control decisions in a system wide sense. Each ParMod makes its choices in a way that the total sum of the local cost functions is minimized, without having knowledge of the objective functions of the other modules. We have also provided an algorithmic solution for solving this distributed optimization problem, based on an iterative interaction scheme.

Finally, the effectiveness of the approach has been tested on several example. According to our results, our model is able to reduce drastically the number of reconfigurations that the system would make if performances and resource utilization where the only metrics accounted. We have shown that, the stability of the system, expressed in terms of persistence of a configuration, is notably increased, and furthermore the number of tasks computed is only slightly less.
Bibliography


