Adaptive Structured Parallelism

Horacio González Vélez

Doctor of Philosophy

Institute for Computing Systems Architecture

School of Informatics

University of Edinburgh

2008

Abstract

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction. Parallel programs are expressed by interweaving parameterised skeletons analogously to the way in which structured sequential programs are developed, using well-defined constructs. Skeletons provide top-down design composition and control inheritance throughout the program structure. Based on the algorithmic skeleton concept, structured parallelism provides a high-level parallel programming technique which allows the conceptual description of parallel programs whilst fostering platform independence and algorithm abstraction. By decoupling the algorithm specification from machine-dependent structural considerations, structured parallelism allows programmers to code programs regardless of how the computation and communications will be executed in the system platform.

Meanwhile, large non-dedicated multiprocessing systems have long posed a challenge to known distributed systems programming techniques as a result of the inherent heterogeneity and dynamism of their resources. Scant research has been devoted to the use of structural information provided by skeletons in adaptively improving program performance, based on resource utilisation. This thesis presents a methodology to improve skeletal parallel programming in heterogeneous distributed systems by introducing adaptivity through resource awareness. As we hypothesise that a skeletal program should be able to adapt to the dynamic resource conditions over time using its structural forecasting information, we have developed ASPARA: Adaptive Structured Parallelism. ASPARA is a generic methodology to incorporate structural information at compilation into a parallel program, which will help it to adapt at execution.

By means of the skeleton API, ASPARA instruments a skeletal program with a series of pragmatic rules, which depend on particular performance thresholds based on the nature of the skeleton, the computation/communication ratio of the program, and the availability of resources in the system. Every rule essentially determines the scheduling for the given skeleton. ASPARA is comprised of four phases: programming, compilation, calibration, and execution. We illustrate the feasibility of this approach and its associated performance improvements using independent case studies based on two algorithmic skeletons, the task farm and the pipeline, programmed in C and MPI and executed in a non-dedicated heterogeneous bi-cluster system.

Acknowledgements

This travail would never have been possible without the continuous support of different individuals and organisations, so it is my great pleasure to acknowledge their elbow grease:

- My supervisor, Murray Cole, undoubtedly deserves my uttermost gratitude for providing a great deal of guidance, advice, and insight, which made the whole investigation not only highly didactic, but also very enjoyable. His grasp of the true meaning of "open door" and "lend an ear" has proved to be an invaluable asset during his mentorship.

- My sister, Virginia, has been a great customer for the early deployment of this work. We have not only developed an interesting scientific application, but also continued to cement our lifelong relationship. Without her, the recent loss of our mother would have been even more unbearable.

- Marie Melvin and Gagarine (Greg) Yaikhom have provided me with excellent feedback at different stages of the writing. Furthermore, Greg has proved to be an excellent conversationalist during our sempiternal discussions on academia, life, and related matters.

- Samantha Lyle and Robert (Rob) Hutchison have certainly spiced up my time in Edinburgh by inviting me to their gatherings and sharing their interesting business ideas.

- Robert Beardon has helped me to atone my British verbal and cultural skills during our sixteen years of friendship. As per my dubious spoken accent, he is not to be blamed.

- Colleagues from the EC-funded HealthAgents project have contributed to this endeavour in different manners. In particular, Bonnie Webber, Magí Lluch i Ariet, and Alex Gibb deserve a special mention.

- Anonymous peer reviews to the different conference papers and journal articles have also helped to improve this work.

- Relatives and friends in the UK, the Continent, México, and the rest of the Americas have fervently provided moral support.

- The Consejo Nacional de Ciencia y Tecnología (Conacyt) in México and the Foreign & Commonwealth Office in the UK have financially supported this research through a Chevening-Conacyt scholarship. The Informatics Graduate School and the Institute for Computing Systems Architecture of the University of Edinburgh have provided travel grants to present papers in different international conferences and attend a summer school. The EC-funded HPC-Europa project sponsored a research visit to the HLRS-Stuttgart during the summer of 2004.

The invaluable ideas and critical comments received from family, friends and foes have significantly improved this work. However, I ultimately assume full responsibility for any omission, mistake, or flaw that this manuscript may contain.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification.

Some of the material employed in this work has been previously published in technical conferences and journals. In compliance with the applicable copyright regulations, its use is hereby acknowledged [96, 97, 98, 99, 100, 101, 102, 103, 104, 105].

Horacio González Vélez

To my beloved wife, Jana, and my great daughter, Imalia.
Without them and their love, things would not have been as pleasant,
exciting, and entertaining.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Parallel programming aims to capitalise on concurrency, the execution of different sections of a given program at the same time, in order to improve the overall performance of such program, and, eventually, that of the whole system. Despite major breakthroughs, parallel programming is still a highly demanding activity widely acknowledged to be more difficult than its sequential counterpart, and one for which the use of efficient programming models and structures has long been sought after. These programming models must be necessarily performance-oriented, and they are expected to be defined in a

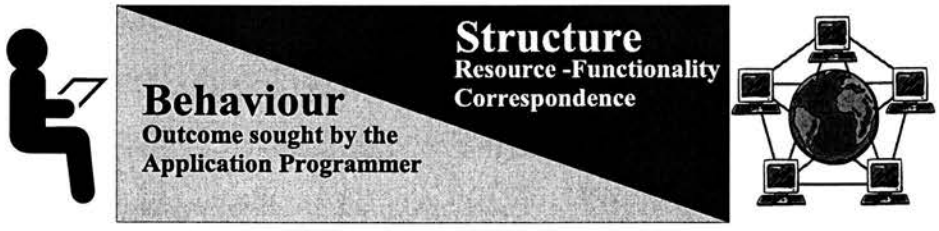


Figure 1.1: The algorithmic skeleton constituents: the behaviour—the outcome sought by the application programmer, and the structure—the resource to functionality correspondence.

scalable structured fashion and to provide guidance on the execution of their jobs in order to assist in the deployment of heterogeneous resources and policies.

This thesis presents a methodology to improve skeletal parallel programming by introducing adaptiveness through resource awareness driven by the program structure. It rests on the resource availability-performance premise, that is to say, the assumption that certain parallel programs can perform more efficiently based on a wise selection from the available system resources.

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction [53]. While computation constructs manage logic, arithmetic, and control flow operations, communication and interaction primitives coordinate inter- and intra-process data exchange, process creation, and synchronisation. Skeletons provide top-down design, composition, and control inheritance throughout the program structure. Parallel programs are expressed by interweaving parameterised skeletons analogously to the way in which sequential structured programs are constructed.

Known as structured parallelism, this design paradigm provides a high-level parallel programming methodology which allows the abstract description of programs and fosters portability by focusing on the description of the algorithmic structure rather than on its detailed implementation. This provides a clear and consistent behaviour across platforms, with the underlying structure depending on the particular implementation. As illustrated in figure 1.1, the skeleton behaviour refers to the outcome sought by the application programmer, and the skeleton structure concerns the resource to functionality correspondence established at the infrastructure level.

Therefore, by decoupling the behaviour from the structure of a parallel program, structured parallelism benefits entirely from any performance improve-

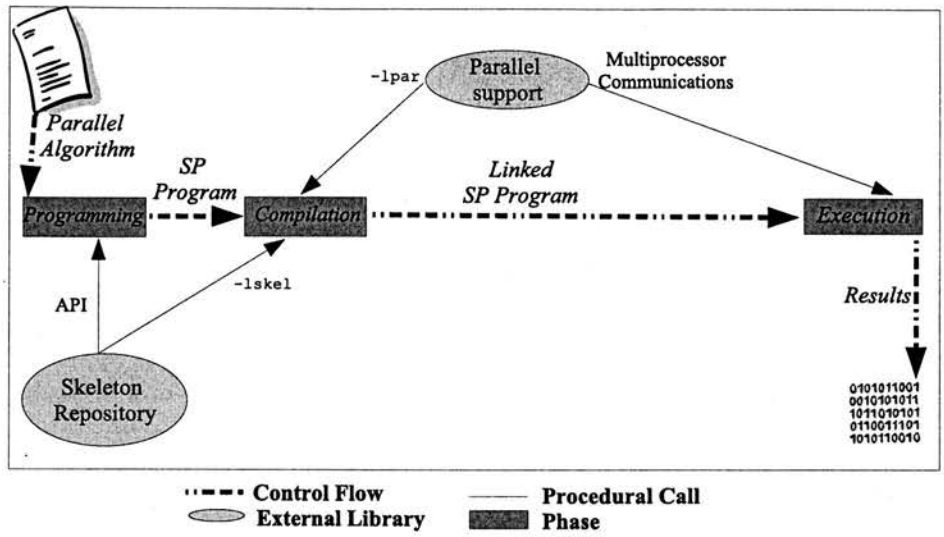


Figure 1.2: The traditional approach to structured parallelism. It does not normally include any provisions for handling resource awareness.

ments in the system infrastructure, while preserving the program results. Figure 1.2 shows the traditional approach to structured parallelism, where a programmer maps a parallel algorithm into a program by selecting a skeleton, or a nesting of them, from a library, and then links the library and the multiprocessing support to produce a skeletal parallel program.

Such behaviour-structure decoupling has allowed the structured parallelism paradigm to be seamlessly deployed on different dedicated and non-dedicated architectures including symmetric multiprocessing, massively parallel processing, and clusters. However, large non-dedicated multiprocessing systems such as metacomputers [177] and computational grids [84]—networks of heterogeneous, computational resources linked by software in such a way that they could be used as easily as a personal computer—, have long posed a challenge to known distributed systems programming techniques, as a result of the inherent heterogeneity and dynamic nature of their resources. Over the last decade, their study has constituted an evolving field in computer science,

and the associated programming frameworks have incorporated assorted paradigms.

The conglomeration of dozens of cores per processor has just increased the complexity of the challenge at hand. A recent report has highlighted the importance of not only producing realistic benchmarks for parallel programming models based on patterns of computation and communication, but also that of developing programming paradigms which efficiently deploy scalable, independent task parallelism [11].

It is widely acknowledged that one of the major challenges in programming support for large heterogeneous distributed systems is the prediction and improvement of performance [133]. Such systems are characterised by the dynamic nature of their heterogeneous components, due to shifting patterns in background load which are not under the control of the individual application programmer. In principle, it is expected that efficient parallel applications must be aware of the system conditions, and adapt their execution according to variations in the available computation and communication resources. The challenge is, therefore, to produce and support applications which can respond automatically to this variability.

Moreover, scant research has been devoted to the adaptive exploitation of the structure of a parallel application to improve the overall resource usage. Since workloads in distributed systems must be divided into tasks in order to minimise communication costs, little attention is paid to partitioning using the application structure. We argue that the intrinsic coordination characteristics of structured parallelism place this paradigm in a preponderant position to explore this area. Based on the central premise of application adaptiveness to resource availability, we would like to research their actual correlation and provide a methodology to enable skeletons to conform to the heterogeneity of

large distributed system.

Furthermore, we consider it relevant to explore this correlation by employing the forecasting information of the structured parallelism model and exploiting its intrinsic adaptiveness through generic conventions. The main difference between this and other performance approaches is that it is intended to be focused on algorithmic skeletons, adaptable by construct, and empirically evaluated on an non-dedicated multi-cluster infrastructure.

Instead of developing a specific skeletal framework, this thesis presents ASPARA (*Adaptive Structured Parallelism*), a generic methodology to optimise performance in heterogeneous distributed systems. ASPARA can forecast and enhance the performance of a skeletal application by exploiting the skeleton structure while preserving the skeleton behaviour.

ASPARA comprises a set of rules to be applied to skeletons, where every rule essentially defines an **application scheduling scheme** which is parameterised in terms of the existing system resources.

As opposed to compile-time optimisation, ASPARA capitalises on the fact that the behaviour of a skeletal application is known prior to its execution. In contrast to run-time optimisation, ASPARA exploits the structural characteristics of skeletons.

We support our claims concerning performance enhancement by presenting positive empirical results based on two task-parallel skeletons: the task farm and the pipeline, implemented as Application Programming Interfaces (APIs) in C with MPI.

1.1 Contribution

The main contribution of this investigation is the methodological exploitation of the forecasting nature of skeletons, which can adapt to different resource constraints in heterogeneous systems.

This methodological approach, synthesised in ASPARA, is evaluated using the task farm and the pipeline skeletons. The importance of these two skeletons has been highlighted in the context of scalable parallel science and engineering applications [70]: the task farm for embarrassingly parallel applications, where parallelism occurs as a result of job scheduling of independent tasks on multiple computers; and the pipeline for lock-step or iterative processing, where the deployment of, for example, linear algebra algorithms on different processing elements, requires the interaction and synchronisation of evolving computational segments.

For convenience, we have implemented simple skeletal APIs for the task farm and the pipeline, but stress that they are merely syntactic vehicles to support the investigation of application scheduling schemes, which forms our main contribution.

The *open question* addressed in this thesis is:

> How much can the structural forecasting information of structured parallelism help to improve the performance of parallel applications executing in a non-dedicated heterogeneous distributed system?

Our *hypothesis* is:

> A skeletal program should be able to adapt to dynamic system conditions over time by steering its execution using structural forecasting information.

Our *objective* is:

To develop performance-wise ways in which to adapt the skeletal structure to the system resource availability.

Should this overall project lead to an increase in the performance of structured parallelism in computational grids, clear potential benefit will be afforded to a significant number of application programmers. That is to say, application programmers will benefit through the utilisation of the skeletal paradigm without suffering degradation of performance.

1.2 Organisation

This thesis follows the standard conventions for doctoral work as surveyed in the literature [157], and can be chiefly classified as a complex variant of the traditional structure. Organised in six chapters, it is divided into three distinct parts: the background, chapters 1 and 2; the core, chapters 3, 4, and 5; and the concluding part, chapter 6.

1: The Introduction Provides an overall description of our approach to using structured parallelism as a way to furnish adaptiveness in computational grids and states the hypothesis, followed by the enumeration of the objectives and structure of the thesis.

2: The Literature Review Presents an outline of the structured parallelism paradigm and its related approaches with strong emphasis on heterogeneous systems in order to place the contribution of this thesis into context. This chapter ends with the identification of the specific research gap addressed by this work, and a description of the computational environment in which all of the experimental results were carried out.

3: The ASPARA Methodology Introduces ASPARA and delineates its four phases: programming, compilation, calibration, and execution. While the former two phases are static phases, as they instrument the skeletons, the latter two are dynamic and enable the ASPARA adaptivity. This chapter is supplemented with a description of the experimental computational environment for this thesis.

4: The Task Farm Includes a self-contained case study exploring the applicability of ASPARA, using the task farm skeleton. It includes the skeleton definitions, the description of its API implementation, an empirical analysis, and a concluding discussion.

5: The Pipeline Renders a self-contained case study analysing ASPARA with the pipeline skeleton. It contains the skeleton concepts, its API, a series of experiments, and a final discussion.

6: The Conclusion Furnishes an introductory restatement of the problem, followed by a description of the research area covered by the thesis. It concludes by discussing possible avenues for future work, as well as those for possible applications of this investigation.

It is important to highlight the fact that the discussion sections in chapters 4 and 5 include an introductory statement of the principal findings of the chapter, a critical comparison with other approaches, and a description of the outcomes.

The notation employed in this work assumes familiarity with parallel computing. Unless otherwise specified, herewith:

- *'processor'*, *'processing element'*, and *'node'* are employed interchangeably to denote a computer with its own private memory capable of execut-

ing an independent stream of instructions. Nodes are assumed to be of heterogeneous nature with disjoint memory spaces;

- *'application'* and *'program'* are utilised interchangeably to denote a sequence of instructions which follows the Single Program Multiple Data (SPMD) model;

- *'skeleton'* and *'algorithmic skeleton'* must be understood as synonyms. Additionally, *'structured'* and *'skeletal parallelism'* should be interpreted in their broadest context to describe the abstraction of patterns of computation and communication; and

- *'workload'* or *'divisible workload'* will denote a computational job which can be arbitrarily partitioned into distinct independent tasks which can be arbitrarily processed in different nodes. Furthermore, in the context of this work, the input and output of the tasks will be either totally disjunct (embarrassingly-parallel computations) or have precedence relations (pipelined computations). Both types of computation modes are well documented in the literature [30, 195], encompassing an important set of problems in parallel computing.

Some of the results of this investigation have been reported in peer-reviewed international conferences and journals. Namely:

- A comparative study of intrinsic parallel programming is presented in [103] and skeletal programming is briefly examined using the eSkel library in [97].

- The ASPARA methodology has been introduced in [101][1].

[1] Although the ASPARA methodology was initially introduced as GRASP, we have renamed it to avoid confusion with the contemporary homonymous project GrASP, Grid Assessment Probes, originated at the University of California-San Diego [51]

- General performance results for a large heterogenous distributed system encompassing servers in Stuttgart and Edinburgh have been initially reported in [98]. The ideas and notation for the task farm skeleton have been studied in the context of the aforementioned infrastructure in [96] and broadened as an adaptive single-round scheduling algorithm in [99].

- The deployment of a calcium currents application is described in [104] and extended to a parameter sweep framework in [105].

- Adaptive pipelining has been initially adumbrated in [100] and, subsequently, generalised to include automatic threshold determination in [102].

In terms of format, the introduction of a new concept is marked by the use of an *italicised* font and its corresponding entry into the index. Literal computer input or output is indicated by the `teletype` font. This manuscript has been produced using the LaTeX text processing system with Palatino fonts and the "infthesis" class from the School of Informatics of the University of Edinburgh. Bibliographic references are listed in alphabetical order using the ACM BibTeX style with ISO journal title abbreviations and DBLP conference short names.

Chapter 2

Review of Literature

This thesis presents an empirical study of the optimisation of structured parallelism in heterogeneous distributed systems, based on the use of skeletal structural foreknowledge. In this chapter, a review of the current literature in this subject is presented. This review has been conducted using a simple hypothesis-centred protocol to provide a top-down analysis of works relevant to the research question.

Firstly, section 2.1 presents an overview of the fundamental concepts of shared-memory and message-passing programming. Secondly, section 2.2 reviews the structured parallelism paradigm and related high-level approaches[1] in order to identify the open issues in current skeletal methodologies. Thirdly, section 2.3 examines existing attempts to develop high-level parallel programming, with strong emphasis on structured solutions. Finally, section 2.4 discusses the specific research gaps addressed by this thesis.

The main objective of this chapter is to set the context of high-level parallel programming on heterogeneous distributed systems. Therefore, parallel languages, auto-tuners, programming toolkits, agent-based and compositional

[1]Having been conceived as self-contained case studies, chapters 4 and 5 furnish contextual inspections of related work on the task farm and the pipeline.

systems, optimisers, and other related topics are purposely omitted.

2.1 Historical Background

Programming methodologies for hardware-independent parallelism based on general constructs has long been sought after. In a seminal work in the late 60s, Dijkstra proposed the use of *semaphores*—special integer variables accessible through atomic actions— as a way to safeguard operations on shared variables and supply an explicit mechanism to synchronise parallel sequential processes [67]. His work defined the canonical problem of *mutual exclusion*, where a collection of concurrent processes each alternately executing *critical regions*—a structured notation to provide exclusive access to program regions—, provided theoretical grounds for harmonious cooperation between multiple concurrent processes.

During the early 70s, Hoare and Brinch Hansen expanded semaphores with the introduction of *monitors*—a collection of local administrative data, shared variables and procedures—to enable concurrent processes to not only share data and resources, but also to synchronise themselves [41, 120]. With the use of critical regions, such as the bounded buffer and the resource array, Hoare delineated the parallel interaction of the disjoint cooperating processes and the producer-consumer pairing: precursors of the task farm and the pipeline constructs, respectively [119].

During the next decade, Lamport formalised a solution to mutual exclusion, based on temporal constraints demonstrating the feasibility of high-level parallel programming from both theoretical and pragmatic standpoints [134]. Later on, orchestrated efforts have led to the introduction of different *shared-memory programming* models, such as OpenMP [60], and thread-only program-

ming like POSIX threads [122], Java threads [156] and C# threads [33]. OpenMP, a set of compiler directives and library routines, allows the programmer to insert implicit parallelism into monolithic blocks of iterative code, while thread-based programming uses light-weight processes to achieve low-level independent control flow.

Monitors, critical regions, and shared-memory models, in general, supply concurrency and synchronisation through the deployment of data structures partaken of by all processes, and their application space is mostly confined to specific intra-process regions. On the other hand, *message-passing programming* provides synchronisation through send-receive pairing between specific processes and concurrency through explicit initialisation of participating processes, and therefore allows the programmer to control inter-process interaction. Standard message passing libraries, such as the Parallel Virtual Machine (PVM) [91] and the Message Passing Interface (*MPI*) [152, 179], allow a heterogeneous collection of interconnected systems, with potentially different architectures and operating systems, to act as a single computing unit.

Although there is not a definitive answer in the subject, experimental studies [45, 129, 141] have consistently demonstrated that shared-memory is easy to program but lacks scalability and coarse-grain scope, while message-passing is portable, tunable, and scalable but error prone. Furthermore, the low-level communication primitives in message passing have long been compared to the assembly language crudeness and even equated to the usage of the infamous 'go-to' statement [106].

A parallel program ought to be conceived as two separate and complementary entities: *computation*, which expresses the calculations in a procedural manner, and *coordination*, which abstracts the interaction and communication. In principle, the coordination and the computation should be orthogonal and

generic, so that a coordination style can be applied to any parallel program, coarse- or fine-grained.

The aforesaid developments in parallel programming foster a bottom-up model of parallel programming, where computation and coordination are not necessarily separated, and communications and synchronisation primitives are typically interwoven with calculations. As a result, the MPI standard has been augmented with *collective operations* which, in essence, provide an upper layer of computation and communication [19, 72]. However, their implementation and its associated performance is dependent on the physical topology of the system, the number of processes involved, the message sizes, the location of the root node, and the actual algorithm employed [49, 160]. Collectives do not necessarily perform well on grids. In an empirical study, using MPI collectives in co-located servers in Stuttgart and Edinburgh, we have previously corroborated their decreasing performance using variable size data packets on shared network connections [96, 98].

Coordination can be directly enforced through syntactic structures in ad-hoc languages such as Linda [92], Opus [149], and Orca [16]. Linda uses a generative model for process creation and communication, where processes interact with each other using primitive data structures, known as tuples. The tuples form a tuple space, which is designed to be independent of the host languages such as C, C++, Fortran, or Postscript. Opus has been designed as a task coordination language for High Performance Fortran, where processes communicate with each other through shared data. Instead of using low-level instructions for reading and writing, programmers define composite operations for data manipulations. Opus deals with the task parallelism using shared data abstractions, which execute autonomously on their own resources. Similarly, Orca data structures have a certain number of pre-defined

operations that can be executed by processes. This interaction can be compared to the object-oriented method principle. Both Linda and Orca have been extended to support newer languages such as Java [86, 135].

Shared-memory, message-passing, or coordination languages do not provide, on their own, a standard and portable way of decoupling the structure from the behaviour in a program. Hence, traditional parallel programs interleave computation and coordination for a certain algorithmic solution, greatly reducing the possibility of using the program structure as a driving criterion for adaptivity. Structured parallelism can, arguably, shed light on this matter.

2.2 Structured Parallelism

This section discusses the structured parallelism paradigm by providing a general overview of its basic concepts, followed by a review of its related approaches.

Algorithmic skeletons abstract commonly-used patterns of parallel computation, communication, and interaction. Skeletal parallel programs can be expressed by interweaving parameterised skeletons using descending composition and control inheritance throughout the program structure, analogously to the way in which sequential structured programs are constructed [158]. This high-level parallel programming technique, known as *structured parallelism*, enables the composition of skeletons for the development of programs where the control is inherited through the structure, and the programmer adheres to top-down design and construction. Thus, it provides a clear and consistent behaviour across platforms, while their structure depends on the particular implementation.

Since skeletons enable programmers to code algorithms without specifying

the machine-dependent computation and coordination primitives, they have been positioned as coordination enablers in parallel programs [56, 62, 159]. Dongarra, Foster, and Kennedy [69] have highlighted the importance of this behaviour-structure decoupling when suggesting that the encapsulation of algorithms and techniques fosters the production of reusable parallel programs, stating that "a pattern might specify the problem-independent structure and note where the problem-specific logic must be supplied". Thus, they advocate raising the level of abstraction without sacrificing performance.

Despite its elegance and potential, it is important to state that structured parallelism still lacks the necessary critical mass to become a mainstream parallel programming technique. Its principal shortcomings are its application space, since it can only address well-defined algorithmic solutions, and the lack of a specification to define and exchange skeletons between different implementations. Some consideration has already been devoted to the matter [55], and future research may lead to a standard.

2.2.1 Background

Cole pioneered the field with the definition of skeletons as "specialised higher-order functions from which one must be selected as the outermost purpose in the program", and the introduction of four initial skeletons: divide and conquer, iterative combination, cluster, and task queue [52, 53]. His work described a software engineering approach to high-level parallel programming using a skeletal (virtual) machine rather than the deployment of a tool or language on a certain architecture. He later surveyed the field, describing formally the functionality of some data- and task-parallel skeletons, using functional programming notation [54].

Loosely coincident to Cole's initial formulation, fundamental work on high-level parallel constructs was being developed elsewhere, including:

- the identification of nine computational models for a one-dimensional processor array [132], including the pipeline and the task queue;
- the use of higher-level portable monitors [142], including an ask-for monitor: a pool of work units being dynamically dispatched to a pool of processes, essentially, an early instantiation of a task farm;
- the initial description of a transputer processor farm [117], and the use of hardware-based scan primitives for data-parallel algorithms [35];
- the foundations of the Bird-Meertens formalism [32], and the subsequent development of second-order functions based on this formalism, to functionally derive parallel algorithms [175];
- a functional specification for pipeline and divide-and-conquer algorithms [124];
- the generic parallel implementations of branch-and-bound and backtrack, precursors of resolution skeletons [81]; and,
- the automated derivation of programs from specifications [178].

Subsequently, the wide acceptance of the algorithmic skeleton concept to govern the common control structure of parallel applications in computational science [40, 113, 136, 176], along with the release of a programming manifesto [55] and a compilation of different research projects [163], has nurtured the field of skeletal parallel programming. It has also guided the development of skeletal frameworks with their associated set of control and data constructs, regulating the flow, nesting, monitoring, and portability of parallel programs.

These frameworks can be grouped according to the programming paradigm:

Coordination This approach advocates the use of a high-level language to describe the algorithmic behaviour and a host language to handle interaction with the infrastructure. The Structured Coordination Language (SCL) [64], the Skeleton Imperative Language (Skil) [39], the Pisa Parallel Programming Language (P3L) [14], the llc language [73], and the Single Assignment C language (SAC) [108] augment imperative languages, with a coordinating language to describe skeletons at high-level. Translating the skeletal description into the host language, they allow the programmer to generate a program by assembling the high-level skeletal portion with the host language structure on top of MPI.

While SCL and Skil propose a functional syntax to denote skeletons on top of Fortran and C respectively, P3L explores composition to abstract the skeletal behaviour. The llc language implements an OpenMP-style syntax to describe skeletal algorithms which use C/MPI as their host. The main innovations in llc are the use of pragma directives in its notation, and the support of multi-threaded environments. SAC, a dedicated array imperative language with HPF-like syntax, supplies multi-threaded vector operators and loop coordination statements on top of a host language and Pthreads. Its contribution is the provision of a clear-cut separation of the behaviour and structure of a parallel program, e.g., while SCL skeletons are instantiated in Fortran, a standard Fortran program cannot directly invoke an SCL primitive [63], and P3L requires the use of a sequential construct to specify any non-parallel succession of instructions [158].

However, the main disadvantages of the coordination approach are the need to learn a new language, and the necessity to prepare an optimised system infrastructure for the host language, in the form of translators and compilers.

Functional Structured parallelism has been incorporated into parallel functional languages as syntactic extensions, or as functors within existing languages. On the one hand, the Higher-order Divide-and-Conquer language (HDC) [115], Eden [140], and Haskell# [114] widen the Haskell scope with parallel extensions to describe skeletal behaviour. They generate executable parallel programs by either translating the program into a C/MPI source in the case of HDC, or by using the Glasgow Haskell Compiler as infrastructure, in the case of Haskell# and Eden. Furthermore, HDC and Eden provide specific statements to manipulate processes and input/output data streams and present complete programming frameworks.

On the other hand, skeletons have been introduced through Haskell functors into Concurrent Clean [121], into ML with the Parallel ML with Skeletons (PMLS) notation [151], or into Hope [61]. These functors allow the expression of skeletons without disrupting the syntax in the original language. Loidl et al. [138] report a comparative performance study with three skeletal programs in Eden, the parallel version of the Glasgow Haskell Compiler, and PMLS, using a C implementation as baseline.

While the functional implementations are consistently more elegant in the abstraction of parallelism, the C implementation provides the best performance.

Object-Oriented Skeletal constructs are introduced into object-oriented lan-

guages using classes. Based on C++ classes and MPI, the Skeletons in Tokyo (SkeTo) project [146], the Münster Skeleton Library (Muesli) [131], and the Málaga-La Laguna-Barcelona (Mallba) library [2], deploy data-parallel, task-parallel, and resolution skeletons respectively. SkeTo focuses on tree structures, Muesli on coarse-grained control structures, and Mallba on resolution constructs. Additionally, using Java classes and the Java Remote Method Invocation programming interface, Lithium [5] and A Software development System based upon Integrated Skeleton Technology (ASSIST) [191], furnish data- and task-parallel skeletons as part of integrated programming environments.

It is important to emphasise that the aforementioned class-centred skeletal libraries rely on the abstraction capabilities of the object-oriented host language and, since they do not impose a special syntax, they rarely introduce a significant overhead into the resulting program. This paradigm has remained buoyant as a result of the popularity of some object-oriented languages, such as Java and C++, and the implementation of skeletal libraries may help to address the performance-portability problem. Nevertheless, the obtained performance does not necessarily match its methodological advantages.

Imperative Skeletons are also deployed as APIs in procedural languages. By using C procedure calls within a pre-initialised MPI environment, the Skeleton-based Integrated Environment (SkIE) [13], and the Edinburgh Skeleton Library (eSkel) [55], deliver data- and task-parallel skeletal APIs. SkIE, the first commercial skeletal programming framework, places particular importance on inter-operability and rapid prototyping, as applications can be formed by encapsulating sequential modules in different lan-

guages, such as C, C++, Fortran 77 & 90, and Java. eSkel concentrates on portability, as it extends the scope of MPI collective operations by providing a constructive data model, and its emphasis on performance has led to further development [23]. Furthermore, the imperative paradigm has enabled the deployment of image-processing skeletal libraries using image application task graphs (IATG) [153], stream programming [43], and field programmable gate arrays [21]; computer vision frameworks with the Skipper system [170]; and resolution skeletons on loosely-coupled network environments with the Paradigm-Oriented Distributed Computing (PODC) framework [130] and the MapReduce system [65]. In particular, MapReduce provides scalable support to a distributed functional mapping and its associated reduction, which is continually employed in numerous applications with large sets of data on hundreds of nodes at Google [66]. This framework is, arguably, the largest skeletal application framework in operation, and has spun off different open-source development projects such as Hadoop [183] and Phoenix [166].

Overall, the portability and performance of this paradigm have greatly benefited from the C/C++ language performance capabilities, and the fact that it does not introduce syntactic extensions allows its insertion into existing application environments.

As summarised in table 2.1, these frameworks provide assorted skeletons that can be categorised into three types based on their functionality as: .

Data-parallel skeletons Work typically on bulk data structures. Their behaviour establishes functional correspondences between data, and their structure regulates resource layout at fine-grain parallelism, e.g. MPI collectives.

Skeleton	*Scope*	*Constructs*	*Frameworks*
Data-Parallel	Data structures	array operations, broadcast, gather, map, reduce, scan, scatter, ...	Eden^{f}, eSkel^{i}, Haskell#^{f}, Lithium^{oo}, llc^{c}, MapReduce^{i}, Muesli^{oo}, P3L^{c}, SAC^{c}, SkeTo^{oo}, SCL^{c}, SkIE^{i}, Skil^{c}
Task-Parallel	Tasks	task farm, pipeline, loop, sequential, ...	ASSIST^{oo}, Eden^{f}, eSkel^{i}, Lithium^{oo}, llc^{c}, Muesli^{oo}, P3L^{c}, PODC^{i}, SCL^{c}, SkIE^{i}, Skipper^{i}
Resolution	Families of problems	divide-and-conquer, branch-and-bound, dynamic programming, heuristic optimisation, genetic programming, finite differences, ...	Eden^{f}, HDC^{f}, Mallba^{oo}, PODC^{i}, Skil^{c}, SCL^{c}

Table 2.1: A taxonomy for the algorithmic skeleton constructs based on their functionality, listing in the last column those frameworks which include skeletons of the given functionality. Key to framework programming paradigm: ^{c}: Coordination; ^{f}: Functional; ^{oo}: Object-Oriented; ^{i}: Imperative

Task-parallel skeletons Operate on tasks. Their behaviour is determined by the interaction between tasks, and their coarse-grain structure establishes scheduling constraints among processes, e.g., task farm and pipeline.

Resolution skeletons Delineate an algorithmic method to undertake a given family of problems. Their behaviour reflects the nature of the solution to a family of problems, and their structure may encompass different computation, communication, and control primitives, e.g, the divide-and-conquer and dynamic programming skeletons.

2.2.2 Related Work

Patterns Having been designed as abstractions of common themes in object-oriented programming [89], *patterns* have been incorporated into parallel programming. Pattern-based parallel programming allows an application programmer the freedom to generate parallel codes by parameterising a framework and adding the sequential parts [143, 150]. The parallel programming pattern concept has been extended into a design method under the umbrella of parallel pattern languages. Unlike other parallel programming languages, *parallel pattern languages* present rules to design parallel codes based on: *archetypes* –problem-class abstractions which describe parallel structure, dataflow, and communication– [147, 193]; *critical region locks*, such as test-and-set and queued for simple mutual exclusion, or reader/writer for concurrent execution [148]; or *socket-based operators* for web applications [168]. Furthermore, Triana deploys a pattern-based programming framework for scientific workflows [94]. Parallel programming patterns and their derived languages have maintained, arguably, the best adoption rate; however, they have become conglomerates of

generic attributes for specific purposes, oriented towards code generation rather than the abstraction of structural attributes.

Templates Commonly denoted in object-oriented programming by *templates* or *generics*, type parameterisation resembles higher-order functions, where instantiation is performed through the types received. Templates provide support for data-parallel programming through object and collective operations such as join, map, and reduce [154], domain-specific operators, e.g., multidimensional arrays and objects to model particle physics experiments [162], or C++ matrix and vector data structures [123]. Incorporated into the C++ standard through the C++ Standard Template Library [12], templates can contain pointers to different language implementations, computation, and tuning considerations, and information on their usage. *Generic programming* expands the templates notion by providing a method for the automated development [93, 172] or the platform optimisation [194] of scientific libraries. Templates possess interesting potential as a parallel programming technique, but have been mostly confined to the development of high-performance computing applications. A survey on the subject is presented in [167].

Components *Components* are objects which associate operations with events. A *component model* is a set of objects (meta-objects) with published interfaces which comply with a set of rules defined by a specific concurrent model. A component model augmented with a set of system components defines a component architecture, where parallel programs are assembled using independent components. Components have been particularly effective in the deployment of infrastructure services—messaging, access and security, information and directory, job submission, schedul-

ing, and user support—in distributed environments [82, 90], leading to the creation of the Common Component Architecture [10], a standard for the development of component-based applications. *Generative programming* proposes the automatic selection and assembly of components on demand, where the programmer specifies the application in a domain-specific language [59]. Despite its popularity in distributed systems programming, component-based programming poses a challenge to system complexity and compatibility, due to the multiplicity of sources and formats of components [57].

There has been cross-pollination of the aforesaid programming paradigms and the skeletal, resulting in evolved hybrid methods which include: skeletal libraries implemented using C++ templates [79]; components deploying skeletal functionalities for mobile environments [130] and grids [76]; and the encapsulation of high-level communications and processing by implementing common parallel blocks and low-level communication primitives, i.e., parallel patterns with skeletal conduct [107].

Nevertheless, we consider it important to emphasise the distinction between the 'pure' skeletal approach and the other approaches presented in this section. In the former, skeletons are the *raison d'être* for the formulation and implementation of all algorithmic solutions; in the latter, the demonstrated capabilities resemble skeletal constructs but do not specifically enforce the use of skeletons as such.

2.3 Application Scheduling in Heterogeneous Systems

The advent of multi-core processors, chip multiprocessors, and multi-node clusters and constellations has steeply increased the number of concurrent pro-

cessors available to a single application. From a single node perspective, tens of cores have begun to be commonplace and, as a result, the most powerful system in the November 2007 Top500 list features 212,992 processors [186]. Having been conceived as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities [83, 84], a *computational grid* may encompass multiple clusters and constellations, requiring a coordinated task parallel method to execute a single application, and, consequently, increasing the infrastructure complexity.

Consequently, the adoption of efficient programming models and structures, which can be staged in a scalable, structured, fashion, has long been sought after [69, 133, 136, 176]. These programming models must be necessarily performance-oriented, and they are expected to provide guidance on the overall application execution in order to assist in the deployment of heterogeneous resources and policies. From a systems administration perspective, the demand for strategies which minimise communication overheads makes resource management and scheduling key to the correct functioning of the underlying platform.

Although different parallel solutions for heterogeneous distributed systems have traditionally exhibited skeletal constructs, their associated optimisations have not necessarily exploited the application structure. They have either modified the scheduler [46] or kept the actual application interlaced [171], without decoupling the structure from the behaviour.

Vadhiyar and Dongarra [188] suggest that a "self-adaptive software system examines the characteristics of the computing environments and chooses the software parameters needed to achieve high efficiency on that environment". Thus, we consider that the key challenges in *adaptively* improving the

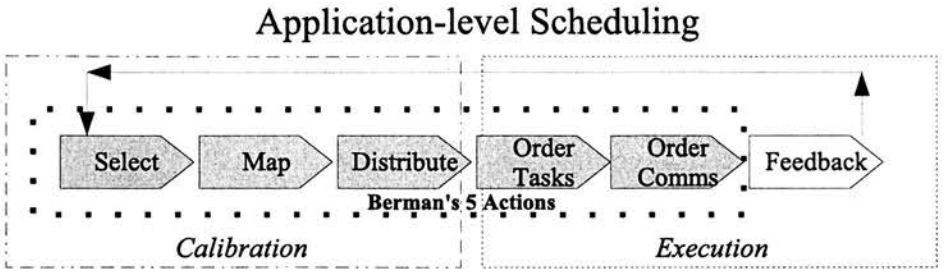


Figure 2.1: Actions for application-level scheduling. The first five actions, represented by the shaded rectangles, are adapted from Berman's view on application scheduling [27]. The sixth, feedback, is an integral part of our methodological approach. These six actions can be chiefly mapped to the two main phases of the ASPARA methodology, calibration and execution, which are to be described in chapter 3.

performance of parallel programs in a heterogeneous distributed system are therefore:

- the correct selection of resources (processors, links) from amongst those available,
- the correct adjustment of algorithmic parameters (for example, blocking of communications, granularity) and, most importantly,
- the ability to adjust all of these factors dynamically in the light of evolving external pressure on the chosen resources.

Such challenges are aligned with Berman's view on intra-application scheduling [27], as illustrated in figure 2.1, which proposes five actions:

1. select resources to schedule the tasks;
2. map tasks to processors;
3. distribute data;
4. order tasks on compute resources; and,

5. order communication tasks.

From this perspective, the ASPARA can be categorised as a *scheduling* methodology for parallel programs executing in heterogeneous distributed systems, which is:

dynamic since the correct selection of resources and the adjustment of algorithmic parameters, are performed at execution time;

adaptive due to the provision of intrinsic mechanisms to dynamically adjust to system performance variations;

application-level because all the decisions are based on the particular requirements of the application at hand;

task-oriented as it assumes that the application is arbitrarily divisible into independent tasks, with or without precedence relations; and

heuristic because it comprises a set of rules intended to increase the probability of enhancing the overall parallel program performance.

Traditional strategies for task scheduling in distributed systems [30, 48, 78, 144] rely on system simulators, dedicated configurations, and/or performance estimators to model the general system, particularly to characterise the background load in terms of its job arrival rate. While much can be said about the reproducibility of their results, one may argue that they artificially create tractable evaluation scenarios for their scheduling policies. Accordingly, the ASPARA methodology cannot be simplistically compared to any task scheduling policy in terms of algorithmic optimality and complexity, but ought to be evaluated in terms of the makespan for a certain workload. To this end, our

case studies report execution times, as well as the relative performance improvement in respect to the baseline for a given workload.

On the other hand, platform-oriented approaches have deployed software extensions to existing frameworks in order to enact application-level scheduling.

1. AppLeS [29] includes a parameter sweep template which implements four different heuristic scheduling policies based on an initial estimate of a task computational cost [47]. Their main disadvantage is the need of instrumenting the application with AppLeS primitives.

2. Nimrod/G [1] incorporates not only the time but also the cost of the resources as the basis for its parameter-sweep scheduling algorithm [42]. It relies on historical data to determine a task computational cost, but receives no guidance from the application structure.

3. Condor [182] has been extended with a master-worker runtime library primarily intended to optimise the number of workers [118]. Based on a usage threshold, it optimises the number of workers by keeping track of the resource consumption over time and returns idle workers to the processor pool. Nonetheless, Condor does not posses any information on the application itself, so all load balancing is carried out using generic guidelines.

4. Workflow systems: Concerned with the automation of scientific processes based on a multiplicity of different data, control, and planning dependencies [200], a scientific workflow is structured using sequential, parallel, choice, and iterative tasks [189]. It is representable through directed acyclic graphs (DAG) and non-DAGs, which include not only communi-

cation, computation, and interaction characteristics, but also application planning requirements and domain knowledge. On the other hand, algorithmic skeletons, and undoubtedly the ASPARA methodology, concentrate on well-defined patterns of computation, communication and interaction, and, consequently, the structure can be used to guide scheduling decisions. Therefore, workflow systems address a different, more generic problem, where the overall application presents more complex control, data, and planning dependencies—which cannot necessarily be representable as a skeletal construct—and, ultimately, scheduling decisions are driven by performance and structural considerations as well as market/economical and security/trust considerations.

While the four cited approaches provide more mature software frameworks with dozens of applications deployed, they typically require:

- the application to be previously instrumented and modified to execute under the specific framework;
- the user to supply application performance estimations or benchmarks; and/or,
- the framework to support complex control, data, and planning dependencies.

That is to say, they do not completely decouple the behaviour from the structure of the application, disallowing the clear advantages of ASPARA, which are its application agnosticism through the use of structural information and its heuristic resource model derived from the system activity.

Finally, recent work on adaptive high-level parallel systems has only reinforced the importance of platform adaptation for the automatic optimisa-

tion of parallel codes in heterogeneous distributed systems. Cunha, Rana, and Medeiros [58] cite a series of component-based problem-solving environments which "allow a clear separation between computation and interaction." While the list is far from comprehensive, it provides clear guidance on the need for enhanced high-level parallel programming tools, and several skeletal libraries now furnish support for heterogeneous distributed systems. Namely:

ASSIST provides load balancing mechanisms through an application manager [6]. This manager uses configuration-safe points within the program to enable load balancing when a bottleneck is encountered. While the reported results on reconfiguration overheads are interesting, scant reference is made to the allocation policies employed, either at the start of the application or at reconfiguration.

eSkel has initially employed Amoget to predict performance using process algebra methods [24]. Amoget, a pre-execution Perl scripting feature, generates a prognostic performance model for a given skeleton which is then fed to the eSkel library to select the resources accordingly. In light of those initial results, where sudden variations in resource usage were not immediately managed by Amoget, eSkel has been extended to incorporate reactive process-algebra scheduling as part of its main library.

Lithium has been extended to provide future-based Java RMI optimisation mechanisms to enhance load-balancing through a statically-defined thread interval [4]. This interval, typically of two to six threads, preemptively controls the node load. Although reported results provide favourable guidance, it is unclear how the interval is defined.

Mallba deploys the Netstream middleware to instrument the skeleton struc-

ture in the library, providing high-level network communication services [3]. Resource selection is based on the readings of network links and node load, but according to the authors, they are "still at the stage of developing intelligent algorithms to use this [network] information to perform a more efficient search [of resources]."

Hence, in contrast to our approach, these skeletal approaches apply *ab-initio* resource-node matching strategies, based on theoretical performance cost modelling (eSkel and Lithium), current resource usage only (Mallba), or reactive modification of the resource allocation regardless of the application structure (ASSIST).

It is also important to stress the fact that ASPARA uses online information, which favourably compares to the use of offline as employed in the initial grid-enabled versions of eSkel. It permits the calibration to automatically feed the node status to the execution, which in turn uses the information as starting point for its operation, allowing a more accurate feedback process.

2.4 Research Gap

As a matter of theory, the *resource availability-performance premise* is widely accepted as the determining factor of application adaptiveness in heterogeneous distributed systems [125, 169, 188]. According to this premise, parallel applications must be able to transform, evolve, or extend their behaviour to conform to the resources present at a certain time to improve their efficiency. The major problem in empirical research based on this premise has arisen from the automatic deployment of generic applications.

Based on the central premise of application adaptiveness to resource availability, we would like to research their actual correlation and provide a meth-

odology to enable parallel patterns (algorithmic skeletons) to conform to the heterogeneity of a given system. Besides, we consider it relevant to explore this correlation by employing the forecasting information of a certain skeleton and fostering adaptiveness through the exploitation of its structure.

Scant research has been devoted to exploiting the structure of the application to improve overall resource management. Since workloads in a large heterogenous distributed system should preferably be divided into independent tasks in order to minimise communication costs, little attention is paid to partitioning using the application structure. Therefore, we shall concentrate on task parallelism and explore two distinct scenarios:

1. The workload is arbitrarily divisible into totally independent tasks (or groups of tasks), and these are scheduled to any available node in a single- or multi-round fashion. We study both scheduling instances in chapter 4 using the task farm skeleton.

2. The workload is decomposed into a sequence of independent computational stages, where the data is passed from one computational stage to another, and each stage, allocated to a different processing element, executes concurrently. We explore this case in chapter 5 using the pipeline skeleton.

We have decided to deploy these two task-parallel skeletons using an imperative skeletal approach based on C and MPI, in order to capitalise on the efficiency of this approach, while providing a higher-level abstraction.

It is crucial to emphasise that task parallel skeletons, and in particular the task farm and the pipeline, have been selected for our evaluation, as they represent a significant set of problems in computational science [70]. Moreover, as described by Brinch Hansen [40], they cover two out of four of the main

paradigms in parallel computing:

1. **pipelining** and ring-based applications;
2. divide and conquer;
3. **master/slave**;
4. and cellular automata applications.

Furthermore, the other two paradigms—cellular automata and divide and conquer—are preeminently concentrated with data and resolution skeletons. We strongly believe that ASPARA can be applied to these two paradigms, since, as documented in table 2.1, they have been previously included in skeletal frameworks—ergo present regular patterns of computation, communication and interaction—, can be divided into independent tasks, and, consequently, be deployed in heterogeneous systems.

By means of these two case studies, this thesis intends to demonstrate the applicability of the ASPARA methodology and its pragmatic approach, which incorporates scheduling rules at compilation time and, based on resource utilisation, adapts at execution time.

Chapter 3

The ASPARA Methodology

This chapter introduces ASPARA, a generic methodology to optimise performance in grids, and its corresponding evaluation environment. Section 3.1 introduces the ASPARA concepts and the four ASPARA phases: programming, compilation, calibration, and execution. Section 3.2 discusses the evaluation strategy, including the case studies and the nature of the workload. Finally, sections 3.3 and 3.4 describe the computational environment and the load conditions in which all the experiments have been carried out.

3.1 ASPARA Phases

ASPARA is a generic methodology to incorporate structural information at compilation into a parallel program that helps it to adapt at execution. It instruments the program with a series of pragmatic rules embedded in the algorithmic skeletons, which depend on particular performance thresholds, based on the nature of the skeleton, the computation/communication ratio of the program, and the availability of system resources. ASPARA comprises a set of rules and their rationale to apply such methods to a set of skeletons, where

Algorithmic Skeleton	*Workload Type*	*Computation Type*	*Application*
Task Farm	Disjunct	Embarrassingly-parallel	Parameter sweeps, Ray tracing, Geometrical image transformation, Mandelbrot set, Monte Carlo methods, N-body simulation, Pseudo-random number generation, ...
Pipeline	Precedence relations	Pipelined	Linear algebra, Adding numbers, Sorting numbers, Prime number generator, Molecular dynamics, ...

Table 3.1: An illustrative listing of parallel algorithms which use a workload-based approach.

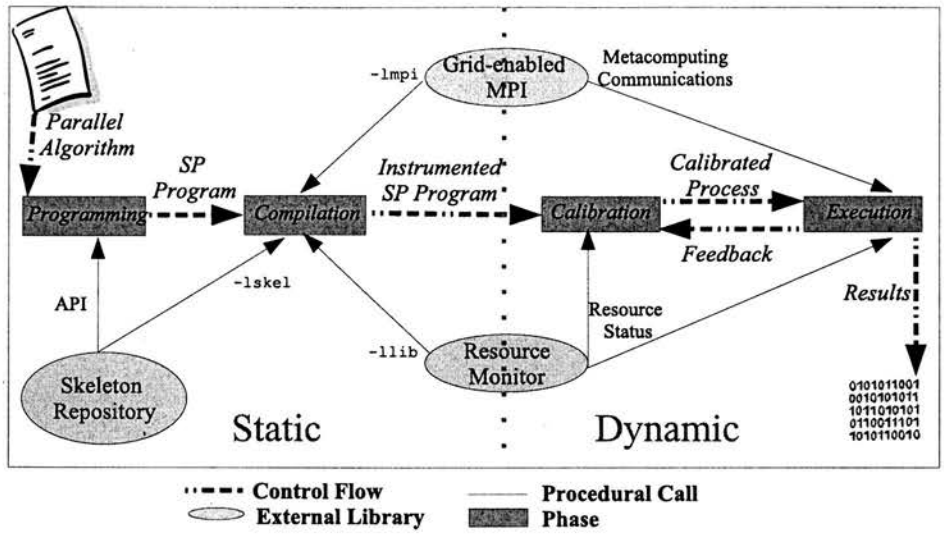


Figure 3.1: The four phases of the ASPARA methodology: programming, compilation, calibration, and execution. Represented by a solid rectangle in the illustration, the first two are the static phases while the latter two are the dynamic phases and the predominant focus of this thesis.

every rule essentially constitutes a defined scheduling method.

ASPARA can forecast and enhance the performance of a skeletal application by exploiting knowledge of the skeletal structure while preserving the skeletal behaviour. The main difference between ASPARA and other performance approaches is that it is intended to be oriented toward structured parallelism, adaptable by construct, and focused on empirical, system-infrastructure methodologies.

This thesis illustrates the applicability of ASPARA with two skeletons: the task farm and the pipeline. As supported by the case studies presented in chapters 4 and 5 respectively, the ASPARA approach is designed independently of any particular application, and is based entirely on the structure encapsulated by the skeletons.

As depicted in figure 3.1, the *four* ASPARA *phases* are:

1. Programming

2. Compilation

3. Calibration

4. Execution

The main differences between ASPARA and the traditional approach to structured parallelism, as illustrated in figure 1.2, are the application-agnostic nature of the program instrumentation at compilation, and its subsequent dynamic steering based on resource utilisations at execution.

In order to illustrate the ASPARA methodology, let us consider a parallel algorithm suitable to be partitioned into a workload, either disjunct, or with precedence relations. Table 3.1 provides a brief account of different parallel applications fitting this profile.

We can now compare both approaches, the traditional and the ASPARA, using task-parallel skeletons which address the same solution. Initially, the algorithm is to be encoded by selecting the appropriate skeleton from each particular implementation.

Then, at compilation, the traditional approach only requires the provision of parallel support while the ASPARA program also compiles the program with resource usage rules.

At execution, the traditional approach spawns tasks immediately, disregarding the existing system conditions. On the other hand, ASPARA first calibrates the nodes by analysing the system resource utilisation and ranking all nodes accordingly. It then schedules the tasks to the most capable nodes within the system, and monitors resource utilisation in order to adapt, periodically or reactively, the task-node allocation.

The previously mentioned capabilities, which constitute the essence of the ASPARA method, are further examined in the following four sections.

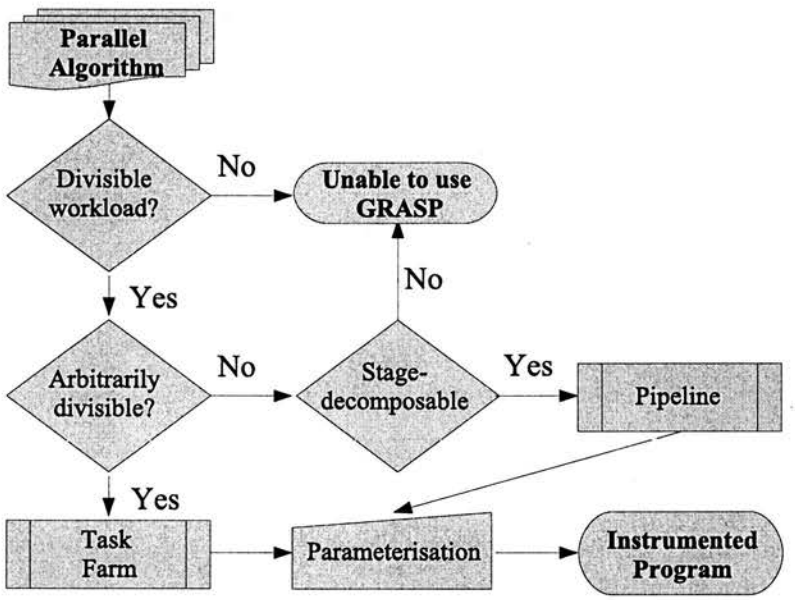


Figure 3.2: The GR programming phase. The figure shows a flowchart to determine whether or not the ASPARA methodology can be applied.

3.1.1 Programming

Programming is a design phase in which the application programmer implements a parallel algorithm by selecting a skeleton through a ASPARA API. Since structured parallelism provides a high-level approach to programming, the programmer is only required to include the initialisation and termination calls for the parallel environment, and the API call in its program.

The skeleton selection depends entirely on the nature of the parallel algorithm in hand, since the programmer should identify the applicable pattern to address the computational and communication requirements of the algorithm. As depicted in figure 3.2, if an algorithm is to be used with ASPARA, it must satisfy the following criteria:

1. The workload must be divisible

   (a) If the workload is arbitrarily divisible, use the task farm

(b) Otherwise, the workload must be decomposable into stages with precedence, and the pipeline can be employed

2. Any other case impedes the use of the ASPARA methodology[1]

Following this, the programmer needs to parameterise the API calls to ASPARA. This parameterisation is crucial because it specialises the generic meaning of the algorithmic skeleton to fit the given problem instance, and, most importantly for our method, to decide on the appropriate scheduling strategy to enact adaptivity at runtime.

3.1.2 Compilation

The compilation phase instruments the structured parallel program. The source parallel program is compiled and linked with the ASPARA library, the parallel environment, and, if statistical calibration is to be employed, a resource monitoring library. The parallel environment handles the underlying metacomputer, including the node initialisation, resource co-allocation, inter-domain scheduling, and other infrastructure matters [87].

The programming and calibration stages are both static, since they do not require any online interaction or feedback from the underlying platform. Moreover, while important in functional terms, their operation is hardly different from that of other skeletal frameworks and, therefore, will be tangentially discussed in the remainder of this thesis.

[1] At this point, ASPARA has only two skeletons implemented. However, the methodology itself is not restricted to these two constructs only, and could be extended to embrace additional skeletons.

3.1.3 Calibration

The *calibration* is an automatic stage, which executes the skeletal program on a sample of the input data on every allocated node, and extrapolates the node performance in order to rank the fittest nodes and, consequently, select the most adequate for the given application.

In this context, the *fitness* is defined as the quality of a node of being the most suitable computational entity to execute a task, or a series of them, on a non-dedicated basis. We then say that "node a is fitter than node b" if, and only if, the measured execution time for node a is less than the measured execution time of node b for a given task, or a series of them.

Measured on a non-dedicated basis in the nodes, the fitness is a transient characteristic. The selection of the fittest nodes from a pool depends not only on the hardware capabilities of each processor, but also on the system software, the existing resource usage, and the parameterised skeletal application at hand. Note that the overall execution time of this phase is determined by the execution time of the least suitable node in the pool, as the correct selection depends on knowing the fitness of all nodes.

Nodes can be selected by extrapolating their performance based on their fitness only. This is known hereafter as *times-only calibration*. This calibration schema has been employed in the case studies presented in section 4.4.2—the multi-round scheduling task farm—and in section 5.3—the pipeline. Optionally, a univariate or multivariate linear regression—correlating the fitness, the node capacity (e.g. node clock frequency or a system benchmark), the processor load, or the bandwidth utilisation—can be applied to statistically adjust the execution time. This is known hereafter as *statistical calibration*, and has employed in section 4.4.1—the single-round scheduling task farm.

Algorithm 3.1: The ASPARA Calibration Algorithm

```
Data: f: Set of Functions;
P: Processor Pool;
Result: Chosen: Table of fittest processing elements;

Execute f over P nodes;
if root node then
    Collect t from P nodes;
    Set t ← execution times(f);
    if Statistical Calibration then
        Collect processor and bandwidth values;
        Adjust t statistically;
    end
    Rank P based on t;
    Select Chosen from P;
    Send Chosen to all nodes;
else
    Send time from this node to root node;
    Receive Chosen;
end
Return(Chosen);
```

Algorithm 3.1 details the calibration procedure. This procedure applies the computing function, f, over an input data subset in order to select the fittest nodes, *Chosen*, from the processor pool, P. Given the existing resource-usage conditions, a *root node* collects the execution times from all processors in the pool and ranks them accordingly. As this ranking involves the actual execution of the application with a reduced data set on the complete processor pool, its temporal complexity is bound by the execution time of the slowest processor.

The calibration phase essentially drives the task-to-node allocation based on the intrinsic properties of the algorithmic skeleton and the resource characteristics and usage. It provides the optimised conditions for execution of the skeletal parallel program.

3.1.3.1 Calibration Issues

Despite its positive qualities, there are two main issues associated with the calibration phase:

Overhead As inferred from the visual inspection of figures 1.2 and 3.1, the calibration phase introduces a delay in the normal control flow, as it stops the traditional execution by imposing an execution barrier. Determined by the slowest node in the pool, the performance impact of the delay has been minimised in our case studies by employing calibration functions with similar complexity for all workers/stages.

Transience The calibration outcome—the set of the fittest nodes—is completely valid for as long as the resource conditions remain the same. Assuming steady resource conditions is not only unrealistic, but also against our initial assumptions. Therefore, we have experimented with calibration under two execution adjustment modes, periodic and reactive, to extend its temporal validity.

Sudden variations in resource conditions render the calibration invalid, but increasing the frequency of calibrations leads to performance penalties. Therefore, a balance between transience and validity must be preserved to avoid thrashing and keep the set of fittest nodes effectual.

3.1.4 Execution

As the set of fittest nodes is initially determined by the calibration phase, the *execution* phase is responsible for keeping this set valid. By monitoring the performance throughout the course of the program execution, this phase:

- *periodically* adjusts the relative nodes' fitness; or

- *reactively* triggers a re-calibration to adjust the set of nodes once a threshold is reached.

The overall functionality of the execution phase is illustrated by algorithm 3.2.

Algorithm 3.2: The ASPARA Execution Algorithm

```
Data: f: Set of Functions;
Chosen: Table of fittest nodes;

Map workload to the Chosen nodes;
Set M as the monitor node;
while ¬ end_of_workload do
    Execute f over Chosen nodes concurrently;
    if M then
        Set t ← execution times(f);
        Adjust Chosen according to t/* This adjustment can be
            carried out either periodically or reactively      */
    else
        Send time from this node to monitor node;
    end
end
Return();
```

Periodic The set of fittest nodes is dynamically adjusted by ranking the nodes according to their relative fitness within the set. Thus, if a node increases its availability during the latest task allotment, not only is its relative fitness affected, but also that of the other nodes. That is to say, the API tracks the execution times of every subset of the workload, allowing the nodes to be correctly ranked at all times. Periodic adjustment is applied for the multi-round scheduling task farm case study presented in section 4.4.2.

Reactive In this execution mode, we formally define a *threshold* as a percentage value of performance fluctuation. This threshold typically expresses the dispersion of the calibration times in the node pool. The more dispersed the times are, the lower the threshold value should be. ASPARA

then makes the application adapt to the infrastructure by allowing performance variations up to this threshold. That is to say, the threshold determines how permissible a performance variation is. Once the threshold is surpassed, the API takes action, e.g., feeding back to the calibration phase and modifying the task scheduling according to the properties of the skeleton in hand. By setting the right threshold for a given platform, one can avoid thrashing due to frequent re-scheduling. Reactive adaptiveness is deployed for the pipeline case study presented in section 5.3.

3.1.4.1 Execution Issues

Ideally, the parametric variations of the periodic and reactive execution modes must be automatically determined according to the existing system load conditions at the start of the execution, rather than requesting any human intervention or feedback. Nonetheless, both execution modes imply an overhead.

- In the periodic mode, the dynamic adjustment is carried out by timing the execution of a set of tasks in a given node and, subsequently, calculating the relative fitness for every node. Both the timers and the arithmetic calculations require processing cycles.

- In the reactive mode, the re-scheduling of tasks requires to stop the normal execution to migrate processes with a consequent idle system time. The full extent of the performance implications of this migration are difficult to estimate since they involve four steps: stop and checkpoint the input stream, pipeline draining, re-calibration, and pipeline filling-up.

Intended as a proof of concept rather than a industrial-grade framework, our methodology has produced performance improvements in our evaluative

case studies. Nevertheless, more realistic applications, possibly with processing peaks or short-lived cyclical variations, might produce idle times and/or thrashing, which would eventually render the adjustments counterproductive. Under those circumstances, the full extent of the the overheads in the different execution modes deserves further examination.

It is crucial to note that both stages, **calibration and execution**, must be dynamically determined, since their actual execution varies according to the application—defined by the skeleton parametrisation and the algorithm itself—, the size of the overall workload, and the resource conditions. It is precisely these two stages which represent the **main differentiator of the ASPARA methodology**, and, therefore, the case studies will examine in detail their instantiation in the context of the task farm and pipeline skeletons.

3.2 Evaluation Strategy

This section describes the overall evaluation strategy and is divided into three complementary parts: the case studies, the workload characterisation, and the application benchmarks utilised.

3.2.1 Case Studies

Chapters 4 and 5 furnish case studies for the task farm and the pipeline skeletons respectively. They are intended to evaluate not only the feasibility of ASPARA but also the performance improvements derived from its application.

Task farm It examines the application of the ASPARA method for the task

farm skeleton to a computational biology parameter sweep, in two distinct modes: single-round and multi-round scheduling. The programming phase requires the `pfarm` API call, parameterised with the worker function, the input and output vectors, the MPI communicator and, most importantly, the scheduling mode.

- Section 4.4.1 applies ASPARA for single-round scheduling. The times-only and statistical calibration modes have been carried out and, therefore, it has required the linkage with a resource monitoring library. The execution phase distributes the workload at once and therefore does not perform any monitoring and/or adjustment.
- Section 4.4.2 applies ASPARA for multi-round scheduling. This is the generalisation of the methodology, as the number of rounds for the scheduling is dynamically defined based on the general node activity. It has employed times-only calibration for three different parameter-sweep scenarios. The execution phase employs periodic adaptation by re-ranking the fittest nodes according to their latest execution times.

Pipeline Section 5.3 reports the application of ASPARA to the pipeline skeleton, employing a standard numeric benchmark. In this case, the programming phase uses the `pipeline` API call, parameterised simply with the stage functions, the input vector, and the MPI communicator. During calibration, the pipeline allocates, in a greedy fashion, stages to processors. The execution monitors the correct functioning based on an intrinsically-defined performance threshold, and reactively triggers a calibration if the threshold is surpassed.

No.	*Scenario*	*Calibration*		*Execution*	
		Times-only	*Statistical*	*Periodic*	*Reactive*
1	Single-round Task Farm	✓	✓		
2	Multi-round Task Farm	✓		✓	
3	Pipeline	✓			✓

Table 3.2: An overview of the three different scenarios addressed in the two case studies, the task farm and the pipeline, presented in chapters 4 and 5 respectively.

Table 3.2 summarises the three different scenarios derived from the two aforementioned case studies. Note that statistical calibration has not been included in the multi-round task farm or the pipeline case studies following the superior results obtained for the times-only calibration in the single-round task farm. Instead, we have decided to increase the number of experiments in order to explore in-depth the skeletal behaviour under different circumstances.

It is crucial to emphasise that, although this thesis analyses two skeletons, the methodology is not circumscribed or restricted to these particular constructs. ASPARA is a generic methodology, and its actual application varies depending on the nature of the skeleton.

3.2.2 Workload Characterisation

In terms of the workload, the underlying assumptions for the three evaluation scenarios are:

1. the selected parallel algorithms have been pre-qualified to be divisible workloads of independent nature or stage-decomposable, that is to say, they deploy embarrassingly-parallel or pipelined computations;

2. the computational complexity of each task in the workload is identical, in the sense that all tasks would take the same time to process one item if executed on a dedicated reference processor. In effect, this is to assume that the programmer has done a sound abstract job of balancing complexity; and

3. the communication time is not significant. Note that this is not to assume that communication is negligible, but rather to assume that communication costs hinder all consumer-producer pairing similarly and, therefore, the overall impact is spread across all nodes on an equal basis.

First Assumption: Independent Tasks

This assumption assures the applicability of the ASPARA methodology, and is therefore *sine qua non*.

Nevertheless, it is important to assert that there is a significant number of real problems in computational science, which can be modelled as divisible workloads of independent tasks, with or without precedence relations [30].

Second Assumption: Similar Task Complexity

This assumption defines the complexity of the calibration phase. Undoubtedly a crucial component of the methodology which provides a standardised initial execution foundation, the calibration phase is bound in its complexity by the slowest processing of all the nodes, and introduces an overhead in the total processing of the workload due to its inherent control barrier.

Let us consider that this assumption is relaxed, such that there are $\jmath$ different task functions with distinct complexities. For a heterogeneous system with P nodes, this will imply the calibration of the P nodes for the $\jmath$ functions which, in itself, will be a hard optimisation problem. Since there exists a sequential constraint to avoid contention, and the execution of each function is bound by that of the slowest node, one would expect, in the worst case, the total time to be defined as the sum of the slowest times in P for each of the $\jmath$ functions. Furthermore, let us assume that the node availability changes during the calibration of the nodes for the ℓ function ($\ell \leq \jmath$). This will render invalid the previous $\ell - 1$ calibrations, which rely on a ranking based on the availability conditions present. As it stands, our constraint of a single function complexity for each task allows us to maintain an acceptable balance between system dynamism and usability.

Although this treatment considerably simplifies the calibration, there are a significant number of problems in computational science, e.g. parameter sweeps, which are modelled as similar-complexity tasks [195]. Nonetheless, different heuristics have been suggested to circumvent this hindrance [137], such as:

1. the definition of a **generic cost function**, in terms of a weight-based computation-to-communication ratio;

2. a **random** stage/worker-to-node allocation; and,

3. the initial calibration of all nodes on a **dedicated** basis.

All the same such approaches may still increase the overall complexity of the calibration process, imply a loss of generality, and/or deactivate our non-dedicated systems approach, affecting particularly the reactive adjustment mode of the execution.

Third Assumption: Communication is not Significant

The third assumption reduces the complexity of the task scheduling problem, as it is widely known that several instances become computationally intractable when communication is considered [78].

In essence, this assumption implies that our performance improvement strategies have not concentrated on node-specific communication issues for simplicity purposes, but could eventually be added.

3.2.3 Application Benchmarks

Realistic benchmark selection for parallel computing has long posed a challenge to academic researchers and industrial bodies. Different groups have embraced distinct testbeds for their performance evaluative endeavours, such as the high-performance computing suite of the Standard Performance Evaluation Corporation (SPEChpc) [9], the Numerical Aerodynamic Simulation (NAS) suite [15], and the Linpack benchmark [71].

On the other hand, structured parallelism practitioners have entertained the idea of standardising the field [55], and such endeavours may eventually lead to the development of a standard skeletal benchmark suite. But, at this point, there does not exist one.

	bw240	bw530
Hardware	64 nodes	16 nodes
CPU	Uni-core Intel P4	Uni-core Intel Xeon
Memory	1 GB / node	2 GB / node
Network	2x100Mb/s	1x100Mb/s, 1x1Gb/s Myrinet
BogoMips	3350–3555	3326–3359

Table 3.3: The hardware configuration of our experimental multi-cluster environment

Therefore, if we had used a realistic popular benchmark in our case studies, we would have required to recode such benchmark to suit our needs, therefore reducing its validity for direct comparative purposes.

We have then employed a hybrid application benchmark approach:

- adapt and solve a real application—a computational cell biology code [104]—in the case of the task farm; and,
- extract a parameterisable processing generation function—the `whetstones` procedure from the 1997 version [139] of the Whetstone benchmark—for the pipeline.

3.3 Computational Environment

The reported results have been obtained employing two interconnected, non-dedicated Beowulf clusters, co-located across the University of Edinburgh, and configured as shown in Table 3.3.

Note that the last row of the table reports the *BogoMips* [190] benchmark value range of the multi-cluster environment. Designed as a comparative pro-

	Scenario		
Software	*1*	*2*	*3*
Linux Red Hat	FC3 (kernel 2.6)	FC5 (kernel 2.6)	FC5 (kernel 2.6)
gcc Compiler	3.4.4	4.1.1	4.1.1
LAM/MPI	7.1.1	7.1.2	7.1.2
GSL	1.5	1.7	1.7
NWS	2.10.1		

Table 3.4: The software versions of our cluster environment during the execution of the three different evaluation scenarios

cessor performance measurement, this benchmark is included with the Red Hat's Fedora Core (FC) distributions.

The complete configuration includes a non-dedicated network, a storage sub-system, and individual cluster management software, enables the interconnection and storage virtualisation while maintaining both clusters as two separate entities.

We consider this multi-cluster environment representative as it:

- spans different administrative domains,
- comprises heterogeneous nodes and links, and
- does not have dedicated resource co-allocation or reservation.

Table 3.4 shows the different software versions for the three evaluation

scenarios. All C programming modules were compiled with `gcc` using the `-pedantic -ansi -Wall -O2` flags.

The co-allocation of multiple nodes in different systems across distinct administrative domains was achieved with the use of LAM/MPI [180].

3.3.1 Stuttgart-Edinburgh Infrastructure

In order to define the design characteristics of our evaluation skeletons for this thesis, we have initially employed the first version of Cole's eSkel library and an artificial integer application composed of basic SAXPY—scalar multiplication and vector ádditión— operations. The overall system has been configured by evenly distributing the processes between a 16-node Beowulf cluster located at the High Performance Computing Centre (HLRS) in the University of Stuttgart, and 16 nodes from the `bw240` cluster.

The farmer node has been positioned at HLRS. We have used the PACX-MPI library for interconnection, with the allocation of two pairs of communication nodes to interconnect both installations. This is the standard requirement for soundly executing PACX-MPI. The MPI versions are MPICH and LAM/MPI and the nodes and communication channels are in non-dedicated mode.

Figure 3.3 shows the channel utilisation from the worker standpoint on the eSkel Task Farm version 1.0 and PACX-MPI 5 for a simplistic application. It presents different input sizes, I, ranging from 160 bytes or $I = 20$ to 1.6 megabytes (MB) or $I = 200{,}000$. While we will defer complete discussion on the nature of the task farm to chapter 4, it is important to mention that half of the workers are located in Edinburgh, while the other half are in Stuttgart.

Although the communication channel at 270 kilobytes per second (KB/s) is

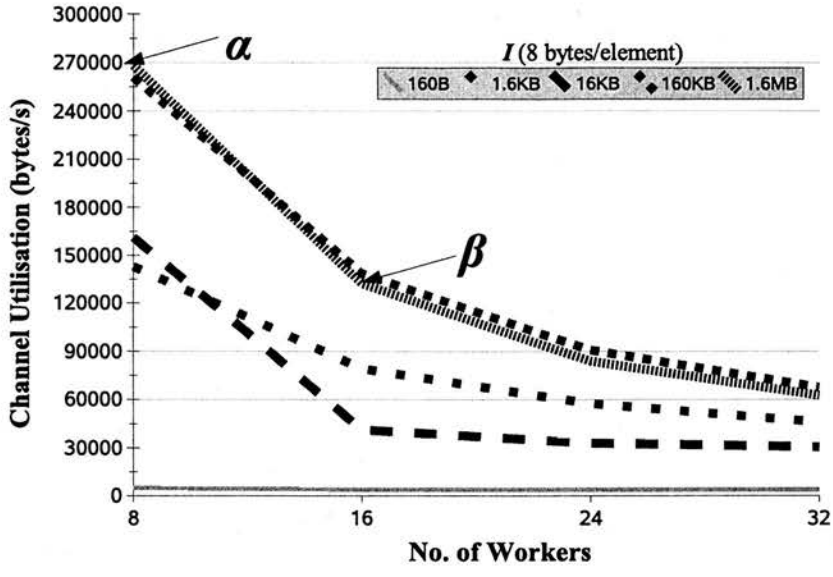


Figure 3.3: Worker Channel Utilisation for five different values of I= 20, 200, 2,000, 20,000, and 200,000, corresponding to 160 bytes, 1.6 kilobytes (KB), 16KB, 160KB, and 1.6 megabytes (MB) respectively. We have employed the eSkel Task Farm version 1.0 and PACX-MPI 5, and half of the workers are located in Edinburgh and the other half in Stuttgart

not saturated while working with 8 processes and 1.6MB and 160KB vectors, equivalent to an I of 200,000 and 20,000 8-byte data elements respectively (α in Fig. 3.3), the increase to 16 processes with the same amount of data implies a 50% reduction in the ability to transmit (β in Fig. 3.3). The complete set of experiments is discussed in [98].

After the analysis of the communication patterns when increasing the number of processes, the performance degradation in the communications was attributed to the use of MPI collective communication operations. As previously reported [160], the optimisation of collectives for generic multi-cluster configurations remains an open issue and is, therefore, beyond the scope of this work.

Two important lessons have been derived from this initial experimentation:

1. we ought to base our skeletons on direct MPI send-receive pairing; and

2. the multi-cluster located at Edinburgh should suffice as the testbed for our evaluation, as the geographical distribution has not crucially modified the environment.

3.4 External System Load

Our testbed significantly differs from others employed for scheduling evaluation as we have assumed that:

1. there is no exclusive or dedicated access to the multi-cluster configuration at Edinburgh;

2. nodes are not restricted to the execution of a single task at a time, i.e., any user in the system can interactively submit tasks in addition to the operating system and administrative processes; and

3. the overall system is always busy, as we have arbitrarily loaded it with instances of a CPU exercising program for system benchmarking, based on [111] and delineated in algorithm 3.3. This injection of additional load has not been systematic and its main purpose is to randomly create load flurries to evaluate the responsiveness of the execution phase.

Hence, the system behaviour cannot be modelled as a closed system where jobs arrive at a fixed or deterministic rate, and it is simply unfeasible to estimate in advance the load distribution: no *a priori* assumptions or foreknowledge on the system load behaviour exist, rendering it impossible to implement any static scheduling policy or load balancing strategy.

Our evaluation programs have competed for time slices with all the other processes present in each employed node and the relevance of our reported results is qualified by two factors:

Algorithm 3.3: A Load-Generating Algorithm

```
Data: a,m: Vectors of length ℓ;

for k ← 0; k < ℓ; k ← k + 1 do
    a[k] ← random;
    m[k] ← random;
end
for ι ← 0; ι < ℓ; ι ← ι + 1 do
    a[ι] ← a[ℓ − ι] ∗ m[ι];
    if ι = ℓ − 1 then
        ι ← 0;
    end
end
        /* N.B. This procedure should be explicitly terminated */
```

1. all reported figures represent the average of a series of executions (five repetitions, unless otherwise specified); and

2. the coefficient of variation in the actual figures is typically less or equal to 10%, indicating that they remain consistent in spite of the load variations.

Reproducing all experimental conditions in full is nearly impossible, due to the presence of load produced by operating system processes, administrative tasks, external user jobs, and our own evaluation programs. Nonetheless, we have provided provide system load charts along with our experimental results in chapters 4 and 5.

As per *stress testing* is concerned, the non-dedicated nature of this environment maintained dynamic system load conditions, which allow us to informally evaluate our application code for robustness, availability, and error handling. Of course, one has to bear in mind that all codes are proofs of con-

cepts rather than industrial-grade frameworks. An alternate approach could have been to simulate a real load environment, e.g. using the Parallel Workloads Archive [80]. However, such approach may have incurred some loss of generality, as our heuristic could have been trained to solve a load scenario as opposed to coping with any variation.

To the best of our knowledge, we have therefore sufficiently exerted ourselves to avoid common pitfalls, and have employed best practices in our experimental evaluation [85], including:

- the provision of as much detail as possible on the experimental environment conditions;
- the use of a realistic system, open to any interactive jobs and system administration tasks;
- the calibration of the nodes without assuming any previous user estimate or application benchmark;
- the deployment of a sustainable workload for a relevant time;
- the avoidance of running the experiments concurrently to reduce contention, and
- the deployment of a representative number of nodes for scalability purposes

To avoid any confusion, it is important to clarify that the terms *'system load'* and *'load'* are utilised interchangeably to denote the total amount of work that a node is doing—including all system and user processes—, and is expressed using the 1-minute average of the load number reading displayed in the Linux/Unix `uptime` command. On the other hand, as initially defined in

section 1.2 and later characterised in section 3.2.2, the term *'workload'* must be interpreted as referring to the skeleton-based application under study.

Chapter 4

Task Farm

The problem addressed in this chapter is as follows: given a task farm program, establish an ingenious way in which to enhance its performance in a heterogeneous distributed environment by effectively determining the number of installments, task sizes, and task-to-processor mapping, and dynamically evolving according to external load variations.

This chapter is organised as follows. Firstly, section 4.1 reviews some fundamental concepts of the task farm. Secondly, section 4.2 discusses the instantiation of two phases of the ASPARA methodology: the calibration and the execution. Thirdly, section 4.3 describes the implementation of the task farm skeleton and the parametric variations of its API. Fourthly, section 4.4 presents the experimental results, divided into two different sections based on the scheduling of the task farm. Finally, section 4.5 concludes with a discussion on this topic.

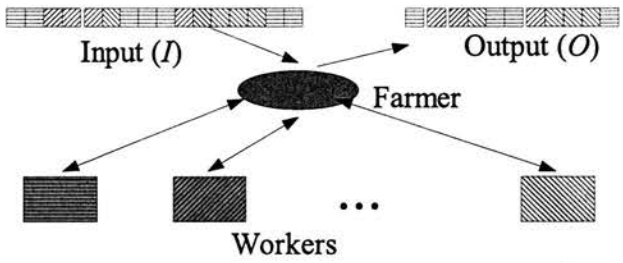


Figure 4.1: Functional representation of the task farm construct. The farmer distributes I, the input, among the workers, which process it into O, the output.

4.1 Background

A *task farm* (TF) consists of a *farmer* process which administers N independent *worker* processes to concurrently execute a set of independent tasks, collectively comprising a *divisible workload*. Symbolically, $TF = \langle I, O, f \rangle$, where I is the input, O is the output and f is the processing function.

A worker executes a task by mapping f into a subset of I (*task size*), computing a subset of O, and then reporting back to the farmer for the next unit of work or termination. This is shown schematically in figure 4.1.

The TF has traditionally dealt with fine-grained data parallelism, where the time taken for communication between the farmer and the workers can be adjusted to be constant, and much less than the computation time [116, 117]. All workers are allocated to a dedicated processor in a parallel machine, the computation of each element in O is independent, and f does not generate the same amount of work for different elements in I. The TF aims at fairly distributing the elements in I to avoid worker starvation and node contention, while minimising communication in order to produce the best load-balancing.

However, it is unreal and/or unfeasible to adjust all communication times in heterogeneous systems. Consequently, different TF implementations assign distinct task sizes to workers based on their *scheduling*. In fact, as the schedul-

ing deals out the workload, it ultimately determines the execution time on a per-node basis. Scheduling variants are typically classified by the number of *rounds* or *installments* in which the total workload is distributed.

Multi-round Scheduling In its canonical form, the TF task-to-node mapping is based on a self-scheduled work queue [112], where the farmer supplies one task to any available worker at a given time. After processing, a worker reports back to the farmer for the next unit of work or termination. For a given workload, each worker normally processes several tasks in multiple installments, constituting a *multi-round scheduling* schema. The work queue strategy provides an acceptable load balancing strategy for large workloads of undetermined size in dedicated systems with fixed network latency. The greedy nature of self-scheduling allows the assign-to-idle-node scheme to balance the system load over time.
The generalisation of the work queue model allocates more than one task per round and takes into account variable network latency, effectively distributing small chunks of the workload in a greedy fashion.

Single-round Scheduling In contrast to multi-round, *single-round scheduling* distributes the entire workload among the workers in one installment. In this scenario, the task size is statically estimated at once to minimise idle time and ensure that minimal scheduling is required from the farmer's side. This is particularly relevant to fixed-size workloads in dedicated homogeneous systems.

Single-round and multi-round scheduling are considered open-ended problems in computational science [20]. In particular, non-dedicated heterogeneous systems, ergo grids, pose an increased challenge, as the farmer is required to adapt the task size assigned to workers because:

- The underlying architecture can maintain multiple communication links between the farmer and workers with different bandwidths and latencies.

- The workers and the farmer can run on non-dedicated nodes with distinct workloads in a distributed environment.

In the following sections, we discuss the application of the ASPARA methodology to the TF skeleton in non-dedicated heterogeneous systems. We describe the relevant concepts for the instantiation of the methodology from a generalised multi-round approach, considering the single-round as a special case.

The underlying assumptions are that the workload is embarrassingly parallel, i.e., all tasks are independent; the TF input, output, and parameters are totally disjunct; and each task has similar computational complexity. Moreover, as the farmer and each worker process are presumed to be mapped to different nodes of a heterogeneous distributed system.

4.1.1 Related Work

Strict semantic connotations aside, the task farm construct is referred to in the literature as the "master-worker", "master-slave", or "bag-of-tasks", which accordingly denotes the farmer as master, originator, emitter, or home, and the workers as processors, slaves, or hosts. Its conceptualisation has been previously formalised, and can be derived from the local computational and task queue models by Kung [132], later refined into the processor farm by Hey [117] and the skeletal task queue [53]. The task-oriented, embarrassingly-parallel nature of the construct, which is the dominant underlying concept, typically

assumes the division of a large workload into independent tasks to be processed.

Widely known as divisible workload scheduling or divisible load theory, the generic problem of mapping groups of totally independent tasks with similar algorithmic complexity to distinct computational nodes has been previously studied in the literature [30, 34, 31]. Proven to be NP-hard [198], it remains an open problem in computational science.

Abstract models for divisible load scheduling in heterogeneous systems provide near-optimal theoretical solutions to particular cases. Banino et al. develop a polynomial solution for the steady-state case, where all processing capabilities, applications requirements, and communication links are known in advance [18]. The Uniform Multi-Round algorithm assumes that every node receives decreasing, fixed-size chunks in every round and provides an approximation to the optimal number of rounds by minimising the application makespan in a simulated environment [199]. Drozdowski and Lawenda also tackle the problem as an optimisation of the application makespan, but relax the assumption on fixed-size chunks, approximating the solution via branch-and-bound and genetic algorithms on a simulated heterogeneous environment [74].

Although the aforementioned approaches undoubtedly provide valuable guidelines, they rely on deterministic arbitrary assumptions on the underlying infrastructure such as the network topology, processor capacity, and the termination time which cannot easily be generalised to accommodate the dynamics of a realistic heterogeneous distributed system.

4.2 Adaptive Task Farming

Adequate scheduling rests on the premise that the workload can be optimally distributed to the nodes with the most convenient resources for a given application, so it is crucial to be able to automatically enable an application to cope with resource variability. Our approach intends to optimise the application performance from a non-invasive systems infrastructure standpoint, using real resource measurements and application times.

The core of our adaptive task farm is the instantiation of the ASPARA calibration and execution algorithms into different scheduling methods for assigning task sizes to different nodes according to their capacity, at once (single-round scheduling), or in several installments (multi-round scheduling).

We suggest quantifying the worker resources at a given time on a certain system topology from an application-specific perspective, by means of a *fitness index* F. Defined during the calibration phase, F is to be used by the TF to determine the task size on a per node basis and, consequently, define the TF scheduling. Moreover, in the generic multi-round scheduling, its value is also adjusted during execution.

Let S denote the workload assigned to the farmer, expressed as the number of tasks in I ($S = |I|$), and N the number of participating workers (typically $N \ll S$). Thence, our objective is to calculate α_i, the task size for each worker:

$$\alpha_i \; \forall \, i \in [1, N] \text{ subject to } \sum_{i=1}^{N} \alpha_i = S$$

If we construct F as the sum of the relative fitness F_i of each node:

$$\sum_{i=1}^{N} F_i \;=\; 1 \tag{4.1}$$

The actual values for F_i are transient, as they periodically change, according

to the latest execution time of every node. Indeed, F_i ought to be formally expressed as a function of time t, $F_i(t)$, where t is the time when the calibration snapshot is taken. However, since all decisions are local to each snapshot, we have simplified its notation by omitting t for readability purposes. This temporal behaviour of F_i is further discussed in section 4.2.2.2.

We can determine α_i as:

$$\alpha_i = S \times F_i \ \forall i \in [1, N] \tag{4.2}$$

Note that α_i is the total number of elements assigned to node i and the key differentiator for the scheduling lies in how this amount is distributed. If distributed in one installment, then the scheduling will be considered single-round, otherwise it will be multi-round. Therefore, we can extend equation (4.2) to consider an *installment factor* k[1]:

$$\alpha_i = \frac{S}{k} \times F_i \ \forall i \in [1, N] \ \wedge \ k \geq 1 \tag{4.3}$$

Since k and F are crucial to our approach, sections 4.2.1 and 4.2.2 discuss the calibration and execution phases respectively, with special emphasis on the determination of both parameters.

4.2.1 Calibration

During this phase, the N nodes are automatically calibrated with the execution of one element from the workload stored in I, the execution times are written to t, and the processed results are stored in O. Then, F is computed using the inverse of the t, either direct or adjusted. These steps have been abstracted in algorithm 4.1.

[1]The installment factor k is formally defined and analysed in section 4.2.2.1

Algorithm 4.1: Calibration Algorithm for the Task Farm

Data: I: The input vector containing the divisible workload;
O: The output/result vector;
f: The worker function;
N: Number of workers;
CM: Calibration mode; /* Two options: Times-only and Statistical */
Result: $\langle F_1, F_2, \ldots, F_N \rangle$; /* Each node individual fitness F */

Calibrate N nodes using f; /* Assumes $N \ll S$ */
Store execution times in t and results in O;
/* Must wait for all nodes to complete before proceeding further */
if $CM = Statistical$ **then**
  /* Adjust t via a curve-fitting method */
  Collect $a_\imath, \ell_\imath, bm_i \; \forall \imath \in [1, N]$;
  Determine $a'_\imath \; \forall \imath \in [1, N]$;
  if *univariate regression* **then**
    Calculate $t = f(a')$;
  else
    Calculate $t = f(a', \ell)$;
  end
end
forall $\imath \in [1, N]$ **do**
  /* Both statistical and times-only use this formula to calculate F */
  Compute $F_\imath \leftarrow \frac{\frac{1}{t_\imath}}{\sum_{j=1}^{N} \frac{1}{t_j}}$;
end
Return $\langle F_1, F_2, \ldots, F_N \rangle$;

It is important to highlight that the calculation of F varies according to the calibration method, which can be:

- *Times-only*: The basic way to calculate F, times-only calibration defines F as a normalised decreasing function based on the inverse of t_i for each i node as shown in equation (4.4).

$$F_i = \frac{\frac{1}{t_i}}{\sum_{j=1}^{N} \frac{1}{t_j}} \tag{4.4}$$

- *Statistical*: F is determined by first employing a curve-fitting method for t, and then using the fitted t in equation 4.4.
  - *Univariate Linear Regression*: t is considered dependent on the processor availability.
  - *Multivariate Regression*: Processor availability and network latency are considered independent and are employed to fit the t values.

While the overhead in the calibration is somehow reduced as this initial processing counts towards the overall processing, its complexity is still bound by the slowest node.

4.2.1.1 Statistical Calibration

Statistical calibration has been widely used in the physical sciences [145] to describe the use of measured physical variables in order to extrapolate a certain unknown via a series of mathematical transformations.

In our case, the idea is to calculate the fitness of a certain node via the statistical extrapolation of its execution time, using the processor availability and the communication latency. This extrapolated fitness will ultimately determine the task size assigned to a node.

Given a certain node, its *processor availability* measures the processing fraction allocatable to a new process to be executed, while its *communication latency* is the time taken to receive a message from the farmer. Let a_ι and ℓ_ι be the processor availability and the communication latency for the ι-node respectively. Consequently, α, t, a, ℓ are vectors of size N which store the values for task size, execution time, availability, and latency. The a and ℓ vectors contain measured physical values typically supplied by a resource monitoring tool.

F is directly determined using t and, transitively, so is α. As t is application-dependent, its value can be correlated with the resources available at a given time.

Such correlation can therefore be explored using:

- a only, *uni-variate linear regression*, or
- a and ℓ, *multi-variate linear regression*.

Univariate Linear Regression

Let us define $a_\iota\prime$, the scaled availability for worker ι, as:

$$a_\iota\prime = a_\iota \times rp_\iota\prime \text{ where } rp_\iota \text{ is the relative performance of worker } \iota$$

$$rp_\iota\prime = \frac{bm_\iota}{\max_N(bm)} \text{ where } bm_\iota \text{ is any known benchmark value for worker } \iota$$

$$\text{and } \max_N(bm) \text{ the maximum } bm_\iota \text{ among } N \text{ workers}$$

Using linear least-squares regression, we set $a\prime$, the vector of $a_\iota\prime$ for the N workers as a predictor (independent variable) and allow t to be the dependent variable. Then, we attempt to fit a curve along the observed values in t using the regression function in equation (4.5), which can be

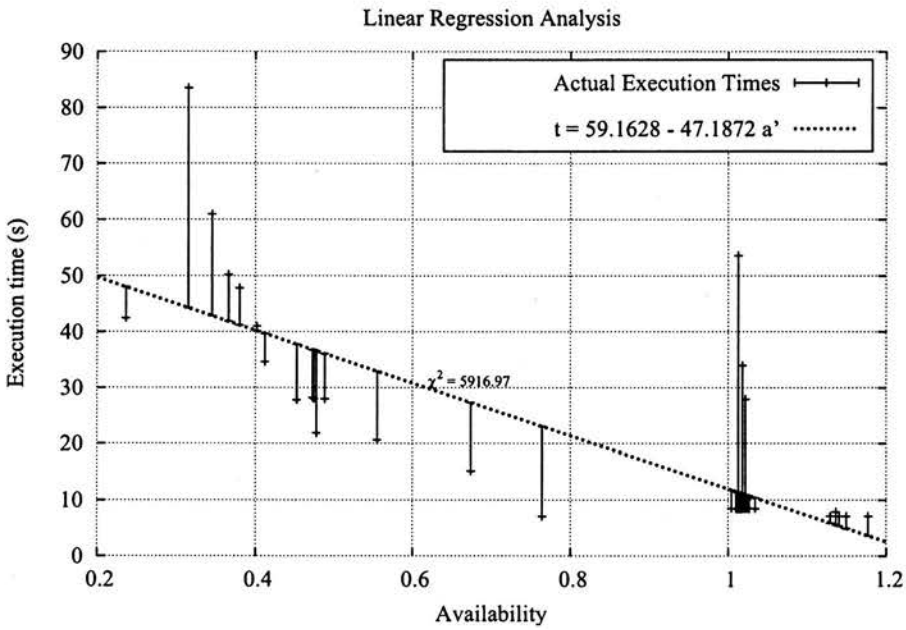


Figure 4.2: Analysis of a 48-worker problem instance using univariate linear regression. The scaled availability ($a\prime$) is used as the independent variable and is plotted in the x-axis, while the execution time t, measured in seconds, is the dependent variable and is plotted in the y-axis.

determined by minimising the sum of squared residuals applying equation (4.6).

$$t = c_0 + c_1 a\prime \tag{4.5}$$

$$\chi^2 = \sum_{i=1}^{N} (t_i - (c_0 + c_1 a_i \prime))^2 \tag{4.6}$$

An example of this linear least-squares fitting method, using the $a\prime$ as predictor and t as dependent variable, is presented in figure 4.2.

Our objective is to assign fewer tasks to the workers which executed tasks more slowly and, in consequence, minimise the overall execution time.

Hence, we calculate the F in equation (4.4), using the estimated (fitted) values t shown in expression (4.5).

Multivariate Linear Regression

Since processor availability is not necessarily the only determining factor, further exploration needs to take into account additional system parameters. In order to provide ground for discussion, figure 4.3 introduces the schematic representation of the relation between processor availability, communication latency, and execution times for the case study to be discussed in section 4.4.1.

It is clear from figure 4.3 that the shortest execution times, represented by the darkest segments, tend to gravitate towards the right following the higher values of a, while the longest times are located in the upper left segment (lowest a). In this particular case, the strong implication of the trend is that execution time on a given node is determined by the processor availability and is influenced, to a lesser extent, by the latency.

Using multi-variate linear least-squares regression, we set $a\prime$ and ℓ as the predictor vector within a matrix (X) and allow t to be the dependent

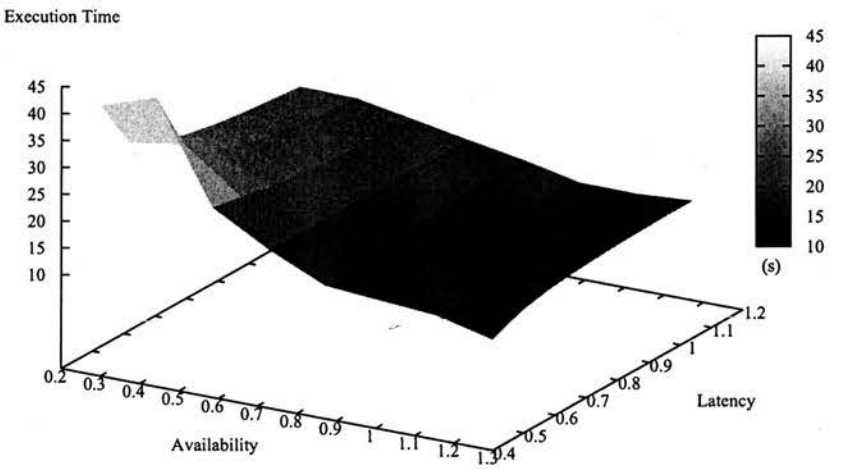


Figure 4.3: The correlation between scaled availability ($a\prime$), network latency (ℓ), and execution times (t), where $a\prime$ and ℓ are used as predictors and t as the dependent variable.

vector. Analogously to figure 4.3, we fit a surface along the observed values in t using the regression function in equation (4.7), which can be determined by minimising the sum of squared residuals in equation (4.8).

$$t = Xc \tag{4.7}$$

$$\chi^2 = (t - Xc)^T(t - Xc) \tag{4.8}$$

Then, similar to the univariate case, we use t as expressed in equation (4.9) to calculate F in equation (4.4).

$$t = c_0 + c_1\ell + c_2a\prime^2 \tag{4.9}$$

4.2.2 Execution

Single isometric installments are well suited to a dedicated homogeneous system, as its node processing capabilities are even. However, in a dynamic system with heterogeneous nodes, single installments should be determined using the node fitness and, in the case of multiple rounds, their actual number and size ought to be dynamically adjusted according to the system load and the prevalent fitness of the system nodes.

The ASPARA execution phase instantiation for our TF can therefore be described as follows:

- if single-round scheduling, substitute $k = 1$ in equation (4.3) and, consequently, distribute the workload in one round, using F to calculate the task size for each node;
- otherwise, assume multi-round scheduling and calculate k based on the node dispersion. As the quotient $\frac{S}{k}$ in equation (4.3) implies multiple installments (if and only if $k > 1$), adapt the task size accordingly during the execution by refreshing F according to the most recent execution time for each node and the remaining amount of work to be completed.

4.2.2.1 Installment Factor

One of the key issues when determining the task size is the initial number of tasks to be distributed. While single-round scheduling directly distributes the entire workload in the first round, generic multi-round scheduling is more complex. In the work queue case, it uses one task per node utilising as many tasks as nodes in the pool in the initial round, and continues in this fashion throughout the entire execution. Nonetheless, there is a potentially large number of possible combinations, which can potentially use a larger number of

tasks in each round and minimise the farmer-worker communication.

To this end, we have proposed to define a new concept: the *installment factor*. Denoted by k, this constant is intended to adaptively regulate the workload distribution in order to determine the installment size for a given worker in multi-round scheduling.

We suggest to determine k in terms of S, the workload, and the dispersion in the calibration times of the N nodes in the pool. This dispersion can be estimated by their coefficient of variation (CV), as represented in equation (4.10), and, as S can be easily conceived as a continually growing function for different problem instances, we can express k using equation (4.11).

$$\text{Given that} \quad \bar{t} = \frac{1}{N}\sum_{i=1}^{N} t_i \quad \text{and} \quad \sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(t_i - \bar{t})^2}$$

$$CV = \frac{\sigma}{\bar{t}} \tag{4.10}$$

$$k = \ln(S)^{CV} \tag{4.11}$$

Assuming that the differences in calibration times reflect not only the system heterogeneity but also its dynamism, a highly-dynamic system will have a series of calibration times with a significantly large standard deviation, σ, and k will grow accordingly, while a steady system will have a negligible standard deviation and therefore k will approach 1, regardless of the input size. Nonetheless, for a given CV, the k will increase logarithmically on S [2]. The behaviour of k for different values of S and CV is plotted in figure 4.4.

We would like to emphasise that calculating k in this generic way relieves the programmer from statically defining the best scheduling, as the skeletal

[2] We have tacitly assumed that $S > e$, i.e., the workload is composed of at least three elements.

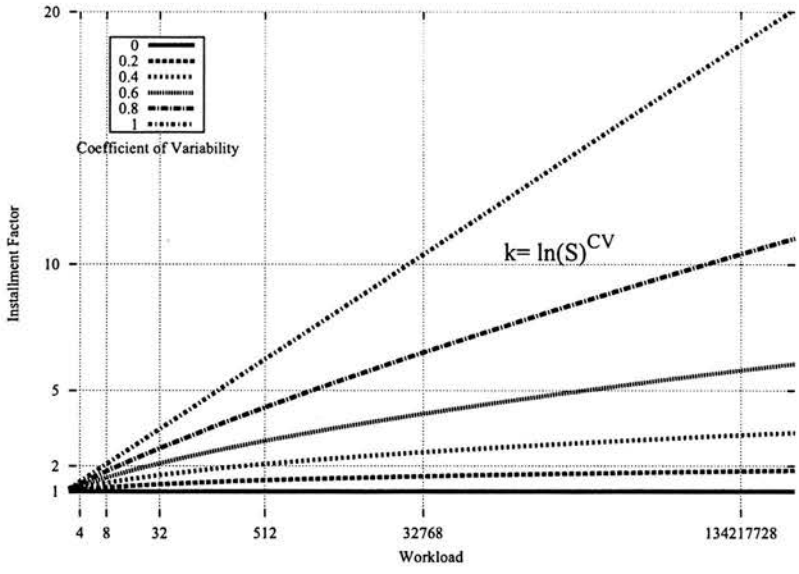


Figure 4.4: The installment factor, k is a function of the workload, S, and the coefficient of variability, CV, defined as $k = \ln(S)^{CV}$. The six different lines of k are defined by the variation of CV from 0 to 1 in intervals of size 0.2.

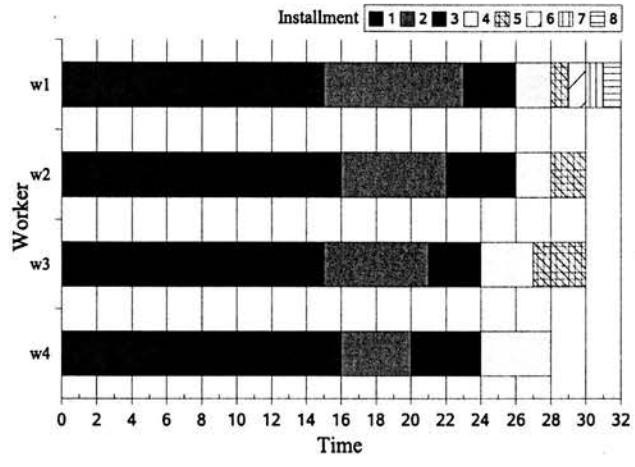
API automatically provides the most suitable number of rounds according to the dispersion in the system and the application at hand. Single-round scheduling simply becomes a special case of the adaptive multi-round scheduling for systems with complete node homogeneity.

4.2.2.2 Adapting the Task Size

The initial calculation of F, the fitness index, abstracts *ab-initio* the resource availability in a given system, but its temporal validity is not necessarily assured as the load conditions frequently vary over time.

In our adaptive approach to multi-round scheduling, we propose to adapt F *periodically* according to the latest performance reading for a node.

Let us examine an illustrative case involving four workers, w_1, w_2, w_3, and w_4, with calibration times of 1, 2, 3, and 4 time units respectively. Suppose



Installment	α_1	α_2	α_3	α_4
1	15	8	5	4
2	8	3	2	1
3	3	2	1	1
4	2	1	1	1
5	1	1	1	
6	1			
7	1			
8	1			

Figure 4.5: The left-side Gantt chart graphically represents the installment timing sequence for each of the four participating workers $w1$, $w2$, $w3$, and $w4$, taking into account their associated calibration times of 1, 2, 3, and 4 time units respectively. The right-side table provides the actual installment sizes for each worker. As an example, the fourth worker, $w4$, has a sequence of installments of size 4, 1, 1, 1 (or a task size of $\alpha_4 = 7$) with corresponding durations of 16, 4, 4, 4.

that initially $S = 68$ and bear in mind that the first four elements are processed during calibration. Thus,

$F_1 = 0.48,\ F_2 = 0.24,\ F_3 = 0.16,$ and $F_4 = 0.12$ by equation (4.4)

$\bar{t} = 2.5,\ \sigma = 1.3,$ and $CV = 0.5\ \ (k \approx 2)$ by equations (4.10) and (4.11)

As per equation (4.3), half of the remaining workload ($S = 64/2$) will be initially distributed to the workers w_1, w_2, w_3, and w_4 in chunks of size 15, 8, 5, 4[3].

Let us suppose that the initial calibration times are preserved as a result of an unchanging node availability, hence the expected execution times for the assigned task sizes will be 15, 16, 15, 16. Given that w_1 reports first for the next installment, the farmer will then assign 8 elements as now $S = 32$. Then, if w_3 follows, the installment will be 2 as now $S = 24$ after the assignation to the first worker. Figure 4.5 shows the full installment sequence for each worker and the timing chart for the example.

Note that the resulting installment sequence chiefly follows a geometrical progression with ratio $1/k$ and reflects the load balancing spirit of the algorithm. Furthermore, as the task sizes $\alpha_1 = 32, \alpha_2 = 15, \alpha_3 = 10, \alpha_4 = 7$ ponder the fitness of every worker, so does the number of installments per node 8, 5, 5, 4. The combination of these characteristics intrinsically reduces the possibility of load imbalances, as larger chunks are initially assigned to reduce scheduling overhead, then smaller chunks are distributed and, at the end, their size is always one, reducing the load imbalance while maintaining resource awareness.

The aforementioned conditions hold true if and only if the fitness of every node remains constant over the execution of the workload. However, one of the main premises for grids is their dynamism. Let us assume that the w_4 performance/availability doubles during the execution of its first chunk com-

[3]The actual arithmetic expression employed to calculate the task size is $\alpha_i = \lfloor \frac{S}{k} \times F_i + 0.5 \rfloor$.

posed of 4 elements, resulting in an execution time of 8 instead of the expected 16. As per equation (4.4), this modifies its own and the other nodes' fitness as ($F_1 = 0.43$, $F_2 = 0.215$, $F_3 = 0.14$, $F_4 = 0.215$) and, consequently, the installment sizes, e.g., the next installment for w_4 is 3.

As a result of this feedback through the latest execution time for each node, the fitness index value is constantly refreshed for each processor. Nonetheless, it is also important to emphasise that the summation of the fitness indices (F_i) is always equal to one, as initially defined in equation 4.1, regardless of the number of processor and the value of the installment factor.

It should be clear that, by recalculating the fitness of every node according to its latest execution time, the execution phase assimilates immediate feedback not only to the node but also to the system as a whole. As the performance of a node is mainly defined by its system load, this technique arguably adapts the TF execution according to the prevailing load conditions.

Figure 4.6 illustrates a realistic 8-worker example in a non-dedicated heterogeneous cluster using $S = 9600$ and $k = 2.735$. Each chart depicts the installment sequence as a continual line, where the size of every installment is indexed to the left y-axis and correlated with the prevailing load conditions indicated with dashed bars indexed to the right y-axis. The load value is the system load average for the previous minute as displayed by the Linux `uptime` command.

Thus, chart (b) represents the 7-installment sequence for w_2 with sizes 454, 263, 161, 159, 2, 1, and 1 under loading conditions of 0.94, 2.3, 1.14, 0.96, 7.47, and 7.23. Note the dramatic reduction between the fourth and the fifth installments as a result of the 7-fold load increase, or the nearly-constant size between the third and the fourth installments as a result of the load reduction. Chart (f) has a more linear behaviour, as the w_6 load follows a more steady

pattern.

Although it is difficult to accurately characterise the entire system and the algorithm behaviour, the eight charts provide a succinct illustration of the overall functionality.

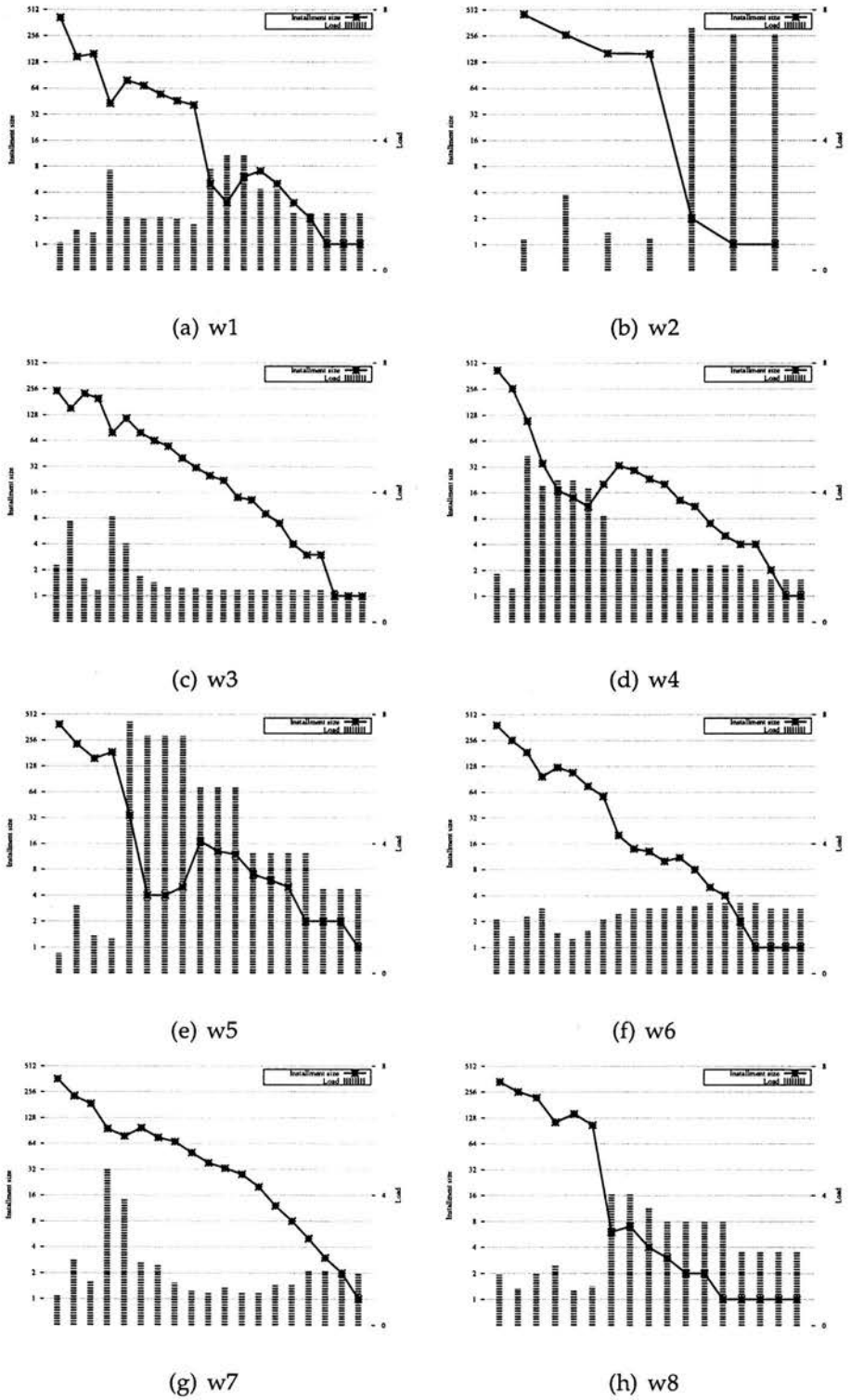
Although k is dependent on the calibration times of the nodes and can arguably be modified every time the fitness is affected, we have decided not to recalculate it every time to avoid overhead, as it only serves as a geometric ratio in the progression, rather than a determining factor for feedback.

4.3 Implementation

In order to use our implementation, a programmer only needs to define the tuple $\langle I, O, f \rangle$—where I and O are the input and output vectors and f is the worker function—and the scheduling mode. It requires no further input from the user. Based on the prevalent load conditions of the defined platform, the calibration phase then automatically calculates the F and the corresponding number of tasks per node α_i and proceeds according to the selected TF scheduling.

Figure 4.7 presents the algorithmic skeleton API implementing the TF. It provides sufficient flexibility to accommodate different options in terms of the worker function (`worker`); the type and size of the input (`in_data`, `in_length`, and `in_type`) and output (`out_data`, `out_length`, and `out_type`); the MPI communicator (`comm`); and the scheduling mode (`sched`). In particular, the valid scheduling modes are presented in table 4.1

That is to say, this skeleton can be used unaltered with single-round scheduling either simple `SCH_DEAL` or `SCH_DEALDYN` in its three variants, and with multi-round scheduling either non-adaptive `SCH_TRAD` or adaptive `SCH_MULTI`. Note



(a) w1 (b) w2

(c) w3 (d) w4

(e) w5 (f) w6

(g) w7 (h) w8

Figure 4.6: An empirical example of the functionality of the task farm adaptiveness on an actual 8-worker system. For each worker, $w1$ to $w8$, the chart depicts with a solid line the installment size sequence with its value indexed to the left y-axis, and, with dashed bars, the load present when that given installment is distributed with its value indexed to the right y-axis.

No	*Name*	*Values*	*Description*
1	SCH_TRAD	$\alpha_i = 1$	Traditional multi-round scheduling based on a work queue (1 by 1)
2	SCH_DEAL	Equation (4.3) holds. ($k = 1 \wedge F_i = \frac{1}{N}$)	Single-round scheduling assuming equal task sizes for the N nodes
3	SCH_DEALDYN_LR	Equations (4.3) and (4.4) hold. ($k = 1$)	Single-round scheduling with statistical univariate calibration (t adjusted via curve-fitting)
4	SCH_DEALDYN_MV	Equations (4.3) and (4.4) hold. ($k = 1$)	Single-round scheduling with statistical multivariate calibration (t adjusted via curve-fitting)
5	SCH_DEALDYN_SM	Equations (4.3) and (4.4) hold. ($k = 1$)	Single-round scheduling with times-only calibration
6	SCH_MULTI	Equations (4.3) and (4.4) hold. ($k = \ln(S)^{CV}$)	Generic variable chunk-size multi-round scheduling with times-only calibration, and single-round as special case ($CV \simeq 0$)

Table 4.1: The six different scheduling modes for our task farm skeleton.

that SCH_MULTI generalises the scheduling mode, as single-round scheduling effectively becomes a special case of the multi-round scheduling for systems with low dispersion.

It is required to be linked with the following libraries:

- The GNU Scientific Library (GSL) [88] for the deployment of the statistical routines. In particular we use:
  - gsl_fit_linear and gsl_multifit_linear for the statistical univariate and multivariate calibrations respectively; and
  - gsl_stats_mean and gsl_stats_sd to calculate the arithmetic mean, standard distribution and, accordingly, the coefficient of variability for the adaptive multi-round scheduling.
- The Network Weather Service [196] for the forecasts of processor availability (a) and latency (ℓ), for the statistical calibration to be enabled. It also requires a NWS sensor per node and a NWS system clique mirroring the MPI_COMM_WORLD encompassing the whole configuration

As described in section 3.3.1, our previous experiences with skeletons on geographically dispersed grids have empirically demonstrated the costly implications of the inherent synchronisation of MPI collectives, resulting in a significant degradation in inter-network communication. Thus, this TF implementation uses MPI_Send and MPI_Irecv for message exchange and MPI_Wait and MPI_Test for synchronisation.

```
void pfarm(void (*worker) (), void *in_data, int in_length, MPI_Datatype in_type, void *out_data,
int out_length, MPI_Datatype out_type, MPI_Comm comm, enum scheduling sched)
```

Figure 4.7: The application program interface (API) to our adaptive task farm algorithmic skeleton

4.4 Experimental Evaluation

Our experiments have been designed to take advantage of the TF intrinsic task parallelism—which presents virtually no inter-process communication—and the ability to access different data sources—inherent to any heterogeneous distributed system. As a result, they deploy a parameter-sweep for a series of independent executions of a stochastic simulation algorithm of voltage-gated calcium channels on the membrane of a spherical cell. Parameterised in terms of number of channels and time resolution, the algorithm calculates the calcium current and generates a calcium concentration graph per run.

A spherical cell possesses thousands of voltage-gated channels, and simulating their stochastic behaviour implies the processing of a large number of random elements with different parametric conditions. Such parameters describe the associated currents, the calcium concentrations, the base and peak depolarising voltages, and the time resolution of the experiment. This process can be modelled stochastically, defining a threshold based on voltage and time constraints, and aggregating individual calcium currents for a given channel population [104].

Furthermore, as the voltages and the peak duration can be varied without affecting the complexity, the parameter space can be explored while preserving the complexity constant at each run. The model has been abstracted as the function f where the number of channels (*channels*) and time resolution, de-

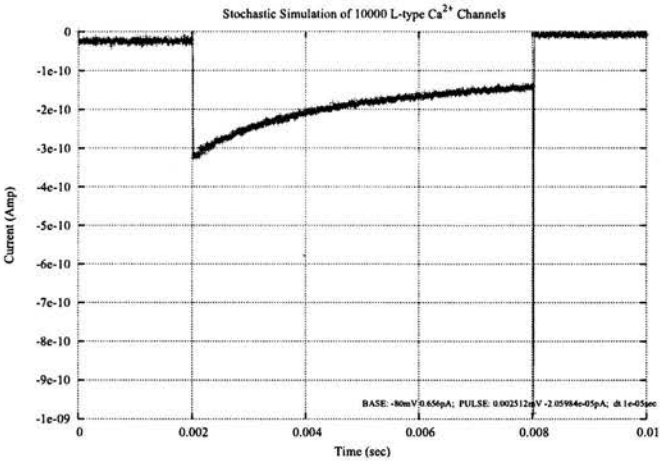


Figure 4.8: An example of a calcium concentration graph generated by each run of the parameter sweep, employing 10^4 channels and simulation time $10ms$ in intervals of $10\mu s$.

fined as the number of steps (*steps*), determine its temporal complexity on a per-experiment basis as shown in equation (4.12).

$$Time(model) \quad = \quad Order(channels \times steps) \tag{4.12}$$

Thus, a typical experiment involving the simulation of 10^4 channels for a second in $10\mu s$ intervals (10^5 steps) will have $Order(10^9)$ temporal complexity. Each experiment generates two result files: a data file which records the calcium currents values over time and a gnuplot script to automatically produce graphs for these values.

Figure 4.8 presents a typical processed calcium concentration graph for 10^4 channels and a $10ms$ simulation time with a time interval of $10\mu s$, i.e., a time resolution of 10^3 steps and a complexity $Order(10^7)$.

The physiological interpretation of the algorithm is beyond the scope of this thesis, nonetheless it is interesting to underscore its relevance to the biomedical community. A complete description of the simulation algorithm, a compre-

	Parameter	*Value*
Experiment	Number of Channels	10^4
	Time Resolution	10^4
	Peak Duration	$0.06s$
Sweep	Peak Voltage Steps	$0.125mV$
	Total Number of Experiments	960

Table 4.2: Parameter space for the single-round scheduling task farm (experimental scenario no. 1 in table 3.2).

hensive parameter sweep, and the physiological interpretation of the results are reported in [105].

In the following sections we present a series of experiments which explore the parameter space in breadth and width: the single-round TF ones cover statistical and times-only calibration for a single problem size (breadth), while the multi-round focus on times-only for different problem sizes (width).

4.4.1 Single-round Scheduling

For this case study, we have instantiated the parameter space with 960 experiments of similar complexity, $S = 960$, by varying the peak voltage, and have defined O to store the individual times for each experiment. The full instantiation is shown in Table 4.2, using a simulation time of 0.1s with an interval of $10\mu s$, and the peak voltage varied in $0.125mV$ steps.

Initially and as a sanity check, we have implemented the sequential version of the workload, executed it in a dedicated reference node, and observed its performance under increasing load conditions. Figure 4.9 plots the execution times in seconds under increasing load conditions. The values in the x-

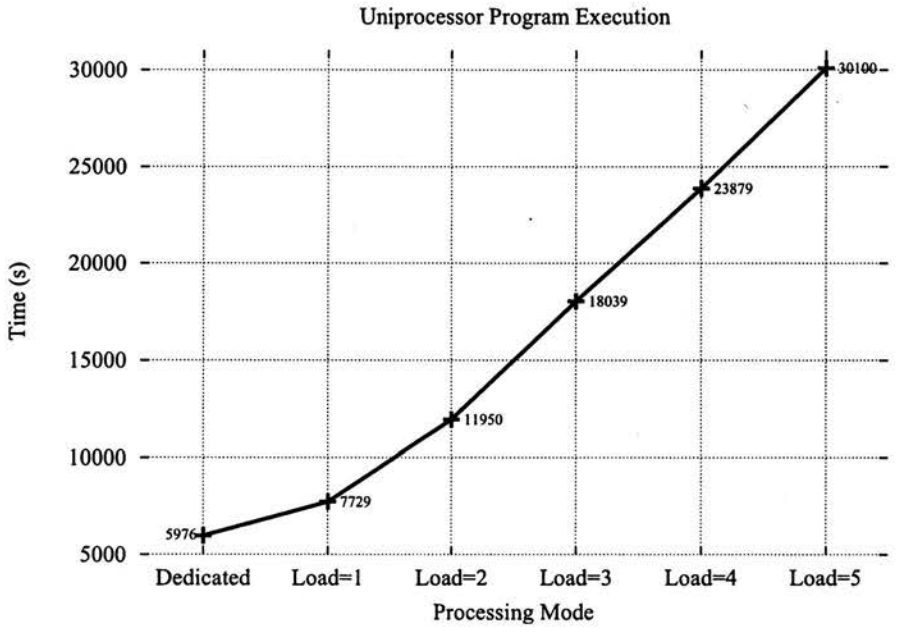


Figure 4.9: Uni-processor execution of the workload under variable load conditions. It employs a sequential version of the worker function in a single processor, and load-generating function. In the x-axis, the values represent the number of instances of the load-generating program, and the y-axis indicates the execution time in seconds.

No.	*Version*	*Execution time*
1	MPI single-round	5890s
2	MPI work queue	7330s
3	Baseline (uni-processor)	5976s

Table 4.3: Execution times, in seconds, of the task farm skeleton for the 960-experiment parameter-sweep. Cases no. 1 and 2 are the MPI version using one farmer and one worker, with single-round and work queue scheduling respectively. Case no. 3 is a sequential version employed as the baseline. The three cases represent the average of five executions on a dedicated system.

axis represent the number of instances of the load-generating program, which is equivalent to 1 in the 1-minute reading from the Linux/Unix `uptime` command. As expected, it degrades linearly when the system load is increased.

Table 4.3 presents the execution times of a simple TF version on a 1-farmer 1-worker dedicated configuration, and compare them to the uni-processor version. The MPI version with single-round scheduling, where the farmer assigns the 960 elements at once to the worker, performs roughly on a par with its uni-processor counterpart (5890*s* versus 5976 or $< 2\%$ difference). The MPI version with work queue scheduling, where the farmer assigns a single task at once, is 23% slower than the uni-processor version (7330*s* versus 5976*s*), and this is mainly due to the overhead incurred by the frequent communication. All entries represent the arithmetic mean, with a small variance, of a series of executions.

For our main evaluation, we have deployed three variants of the single-round scheduling: times-only and linear regression in univariate and multivariate modes. For the uni-variate case we have used the scaled availability, $a\prime$, as predictor variable, and for the multi-variate, we have additionally employed the network latency, ℓ. Both fit the execution times t using linear

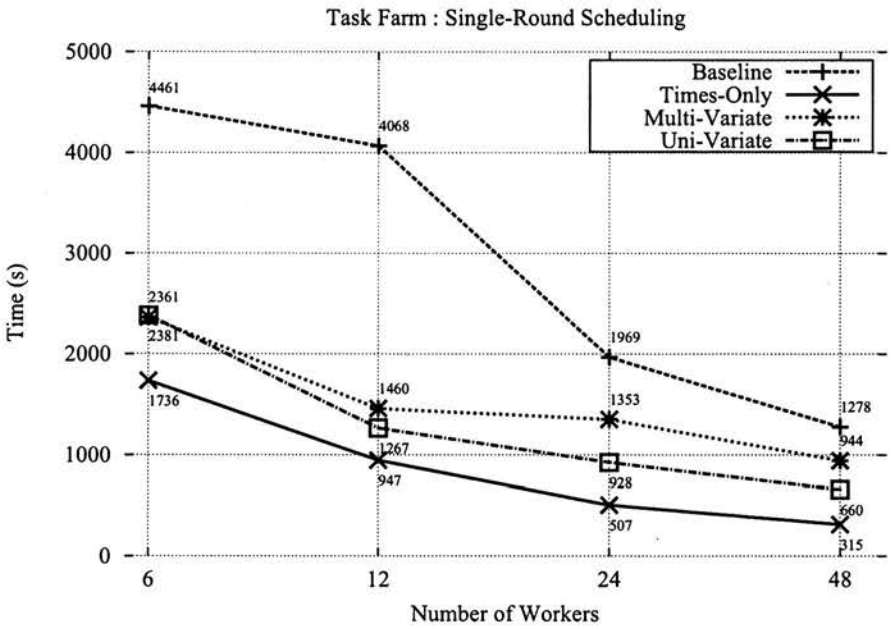


Figure 4.10: Summary of the execution times, in seconds, of the task farm with single-round scheduling using 6, 12, 24 and 48 workers. Key: [Baseline] Isometric-installment single-round scheduling; [Times-Only] Single-round scheduling with times-only calibration;[Multivariate] Single-round scheduling with statistical calibration using $a\prime$ and ℓ as predictors; [Univariate] Single-round scheduling with statistical calibration using $a\prime$ as predictor.

regression. The baseline is an isometric-installment single-round scheduling, i.e., equal task sizes to all participating workers.

We have run a series of experiments with 6, 12, 24, 48 worker processes mapped to an equal number of nodes with the farmer located at process 0. The results are presented in Fig. 4.10 and are based on the aforementioned 960-experiment parameter sweep.

We have chosen the *BogoMips* [190] as the known benchmark value in order to scale the availability values. This decision has been based on the following two factors:

1. Bogomips value claims to reflect more accurately the processing power of a node than the standard CPU frequency and, therefore, provide a better scaling factor; and,

2. Bogomips is widely available in Linux systems, providing a clear advantage in terms of ease of implementation for our evaluation programs.

Nonetheless, the API is not tied to this benchmark and, alternatively, our evaluation can potentially use the 1-processor figures from any other widely-used benchmark such SPEC [77], NAS [15], or Linpack [71].

Each value in the chart represents the average of the executions run under different conditions on three different days. All times are measured at system level and include not only the TF processing but also the calibration and startup-termination periods. The experiments did not run concurrently, in order to avoid any contention.

During the three different days, the evaluation series coped with different system loads and network conditions. Figure 4.11 provides a summary of the system load on a per-node basis for the three days. Each bar represents the average load for the farmer (node 0) and the workers (node 1–48) on a given

day, and includes the load generated by each of the experiments. The flat line depicts the average load for the whole environment.

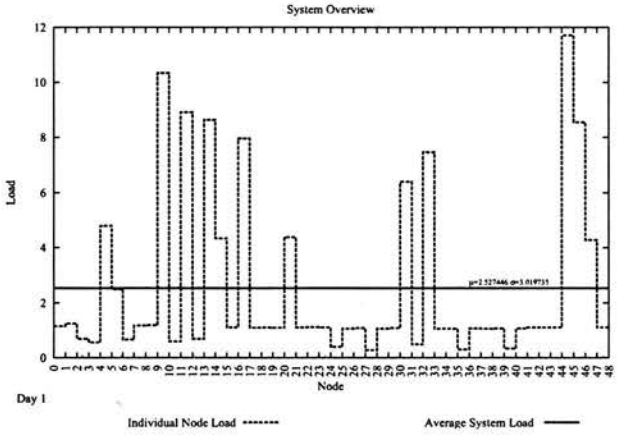
Our single-round scheduling evaluation consistently outperforms the single-round version using isometric installments by 70% for the times-only calibration and 56% and 48% respectively for the univariate and multivariate linear regression cases. Furthermore, if we compare the different modalities of our single-installment scheduling, the times-only calibration performs 43% and 33% better than the statistical calibration modes. Such performance superiority can be possibly associated with the arithmetic operations generated by the linear regression. Thence, we intend to use times-only calibration for the remainder of our experiments.

4.4.2 Multi-round Scheduling

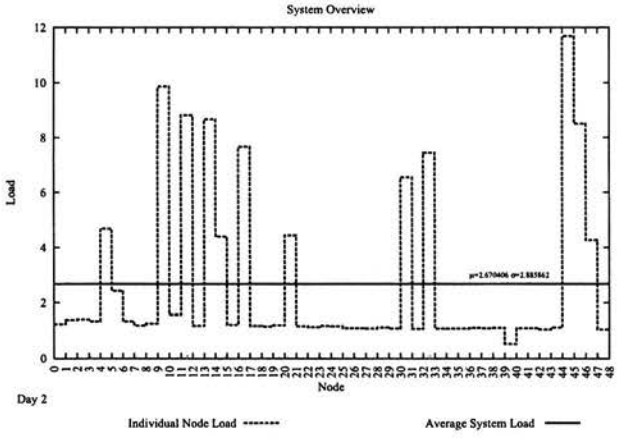
Here we have run a more comprehensive series of experiments incorporating three different settings: light, medium, and heavy.

At the single experiment level, while maintaining the number of channels, the time interval, and the base voltage constant at 10000, $10\mu s$, and $-80mV$ respectively, we have varied the simulation time for each experiment using $0.01s$, $0.1s$, and $1s$, i.e., a time resolution of 10^3, 10^4, and 10^5 steps respectively. Note that the complexity of the experiments in the medium case is similar to that of those employed in the preceding section for the single-round scheduling.

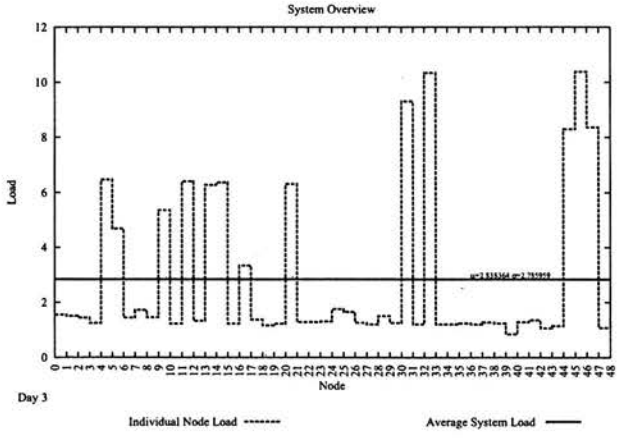
At the parameter space level, the parameter sweep looks upon peak voltages in $[-60, 60mV]$. Employing steps of $0.0125mV$, $0.03125mV$, and $0.125mV$, this range is evenly divided, producing an associated number of experiments of $S = 9600, 3840$, and 960 respectively. Note that the variation in the value of S has no bearing on the complexity of the experiments as described in equa-



(a) Day 1



(b) Day 2



(c) Day 3

Figure 4.11: Overview of the system load for the three days of experimentation for the single-round scheduling. Each chart plots with a dashed line the average load (y-axis) for a worker (x-axis) for the given day. The solid line represents the average of all workers during the day.

	Parameter	*Light*	*Medium*	*Heavy*
Experiment	Number of Channels	10^4	10^4	10^4
	Time Resolution	10^3	10^4	10^5
	Peak Duration	$0.006s$	$0.06s$	$0.6s$
Sweep	Peak Voltage Steps	$0.0125mV$	$0.03125mV$	$0.125mV$
	Number of Experiments	9600	3840	960
	8-worker Time	868.4s	3199.6s	7720.8s
	16-worker Time	470.4s	1730.2s	4205.1s
	32-worker Time	235.3s	869.2s	2125.1s

Table 4.4: Parameter space for the multi-round scheduling task farm (experimental scenario no. 2 in table 3.2). The final time rows show the average execution time, in seconds, for whole parameter sweep on 8, 16, and 32 workers using adaptive multi-round scheduling.

tion (4.12). Table 4.4 shows the three instances of the parameter space.

We have assembled nine different scenarios by varying the number of workers 8, 16, 32 executing the light, medium, and heavy problem instances, and compared our adaptive scheduling with the work queue which is the de-facto scheduling for heterogeneous, dynamic systems. While the results have demonstrated a modest performance improvement of 3% for the light case and a slightly negative decrement for the medium $-.1\%$ and heavy $-.3\%$ cases, the automatic calculation of the installment factor and the periodic refinement of the task size should be considered important contributions for self-scheduling parallelism. A summary of the results is presented in figure 4.12.

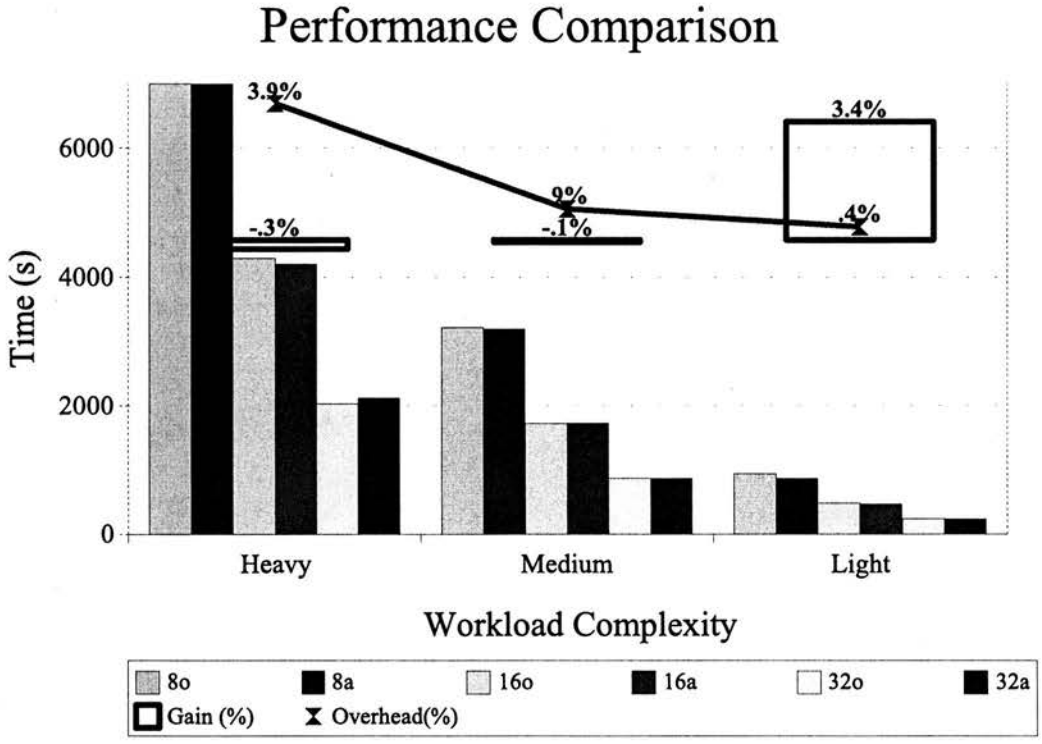


Figure 4.12: Execution time summary for the task farm using adaptive multi-round scheduling on 8, 16, and 32 workers with light, medium, and heavy experimental workload settings, as described in table 4.4. The shaded bars represent the execution times for each combination of worker-scheduling-setting. The top thick line represents the overhead incurred by the initial calibration while the thick-line rectangles plot the average gain from using the adaptive scheduling for all executions in a setting. Key: [processors no.][scheduling], e.g., 8a means 8 workers and adaptive scheduling model and, analogously, 8o represents 8 workers and one-by-one scheduling.

4.5 Discussion

Embracing the ASPARA phases introduced in chapter 3, this chapter has presented a self-scheduling task farm. Initially, the calibration phase ranks processors according to their fitness and determines an installment factor based on how different their execution times are. Subsequently, the execution phase iteratively distributes the workload according to the processor fitness, which is continuously refreshed throughout the program. Programmed as an algorithmic skeleton to be parameterised with the worker function, variable-size input and output data vectors, and the scheduling mode, this task farm has been evaluated for the special case of single-round scheduling and the generic multi-round one using a computational biology parameter-sweep in our non-dedicated multi-cluster.

4.5.1 Outcomes

As proven by the uni-processor figures, the load in the system directly impacts the execution times. Hence, it is crucial to note that the adaptive method shows advantages regardless of the system load. In other words, the adaptive farm is able to adjust itself to the dynamism of the environment.

Section 4.4.1 has provided evidence that the ASPARA calibration phase of the worker function on the nodes can considerably enhance the performance of a task farm, and F can help in predicting variations in system conditions.

The corrective properties of linear regression to relieve exogenic factors in larger runs, such as the arrival of administrative jobs or indiscriminate interactive usage, have been reported to be useful in different scheduling scenarios [192]. However, times-only calibration has proven to be the most effective for our purposes.

From an efficiency perspective, it is arguable that the single-round performance ought to be enhanced by conveying dynamic re-calibration into the distribution when any performance bottlenecks arise. Therefore, we have presented the evaluation of the generic multi-round case which automatically calculates the installment factor based on the dispersion of the calibration times of the nodes, and pervades the impact of changes in the nodes' fitness through a periodic adjustment. Despite their modest performance results, the proposed algorithms have substantial implications for self-scheduling and load-balancing. In particular, their resource awareness, combined with their decreasing task size, reduces the trade-off between scheduling overhead and load imbalance, and has important applications in parallel processing (e.g self-guided parallel loop scheduling [161]).

With respect to the analysis of divisible workloads, the findings of the case study provide an alternate approach to the single-round scheduling problem using forecasts of resource utilisation. This tacitly reinforces the notion that although computational grids are highly dynamical, forecasts based on historical resource utilisation can accurately provide some guidance for distributing workloads.

Chapter 5

Pipeline

The problem addressed in this chapter is: given a parallel pipeline program, find an effective way to improve its performance in a heterogeneous distributed environment by effectively mapping the pipeline stages to the best available processors and adapting dynamically to external load variations.

This chapter is structured in the following way. Firstly, section 5.1 provides the background for this case study. Secondly, section 5.2 describes an adaptive pipeline parallelism approach as an instantiation of the ASPARA methodology. Thirdly, section 5.3 describes the API implementation, followed by the experimental evaluation in section 5.4. Finally, section 5.5 presents a discussion on the topic.

5.1 Background

A *pipeline* enables the decomposition of a repetitive sequential process into a succession of distinguishable sub-processes called *stages*, each of which can be efficiently executed on a distinct processing element or elements which operate concurrently.

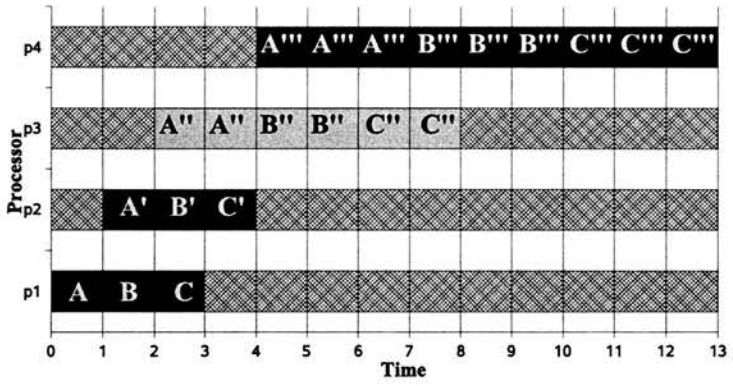


Figure 5.1: Graphical representation of the pipeline construct. A sequence of stage functions are applied to a given data element.

Ranked, behind Amdahl's law and the Internet, as the third most influential parallel and distributed concept of the past millennium [184], the pipeline paradigm has been widely studied in the literature.

Pipelines are exploited at fine-grain level in loops through compiler directives and in operating system file streams, and at coarse-grain level in parallel applications employing multiple processors. With application in grand-challenge computational problem solving, numerical linear algebra, signal/image processing, and scientific workflows, coarse-grained pipelines refine complex algorithms into a sequence of independent computational stages where the data is "piped" from one computational stage to another. Each stage, composed by a simple consumer, a computational function, and a simple producer, is then allocated to a processing element in order to compose a parallel pipeline. Our pipeline follows this model.

The performance of a pipeline can be characterised in terms of *latency*—the time taken for one input to be processed by all stages—and *throughput*—the rate at which inputs can be processed when the pipeline reaches a steady state. Throughput is primarily influenced by the processing time of the slow-

est stage, or *bottleneck*. Figure 5.1 shows an illustrative pipeline composed of four stages allocated to an equal number of processors, $p1$ to $p4$, with different processing speeds: processors $p1$ and $p2$ take 1 unit of time to process an element, $p3$ takes 2 units, and $p4$ 3. Consequently, the pipeline latency is equal to 7 units, its throughput is $\frac{1}{3}$ as it completes an element transformation every three units of time, and the bottleneck is the stage executed in $p4$.

When handling a large number of inputs, it is throughput rather than latency which determines overall efficiency, since the latency is only relevant to measure the time to fill up the pipeline initially. Once full, the pipeline steadily delivers results at the throughput ratio. Hence, in order to improve the overall efficiency of a parallel pipeline, one is required to minimise the bottleneck processing time.

Unlike the task farm where an input element can be independently processed by any given node, an input element in a pipeline must be transformed consecutively by every stage. That is to say, the processing of every element in the data stream depends on the successful completion of the preceding stage—except for the first one. This stage interdependence is called a *precedence relation*, hence the overall workload is not arbitrarily divisible, but stage-decomposable with precedence relations.

5.1.1 Performance Considerations

We must review some generic performance issues in pipelined processing. Suppose that the original sequential process requires time t_s to process a single input. Consider an n-stage pipeline, in which t_i is the execution time for the i^{th} stage.

In an idealised model, without significant communication costs, the se-

quential (T_{seq}) and parallel (T_{par}) times to process S inputs are expressed by equations (5.1) and (5.2) respectively, where $\max(t_i)$ is the bottleneck stage time.

$$T_{seq} = S \times t_s \tag{5.1}$$

$$T_{par} = \sum_{i=1}^{n} t_i + (S-1) \times \max(t_i) \tag{5.2}$$

It is well known that perfect pipelined performance is obtained when the stage times t_i are all equal to $\frac{t_s}{n}$, and T_{seq} and T_{par} can therefore be expressed as noted in equations (5.3) and (5.4), so that as S grows large, speed-up asymptotically approaches n.

$$T_{seq} = S \times t_s \tag{5.3}$$

$$T_{par} = t_s + (S-1) \times \frac{t_s}{n} \tag{5.4}$$

Outside this perfect situation, it is more important to reduce the bottleneck time than the latency, since the former affects the multiplicative term in T_{par}, whereas the latter affects only the asymptotically insignificant additive term.

As previously discussed in section 3.2.2, we assume that:

1. the computational complexity of each stage function, designated as f, is identical and

2. that the communication time is not significant.

Both assumptions intend to keep the number of experimental variables manageable. In particular, the former allows the calibration process time to be determined by the execution time of the slowest processor in the pool. Had

this assumption been relaxed, the calibration phase would require a different design: every node would require to be calibrated with every different stage, making it a longer process with the possibility of losing validity—as the system resources could change during the calibration—and flexibility.

This allows us to focus on addressing issues which arise when the available processors vary dynamically in performance with respect to such a reference processor, as the stage function f can be used to determine the fitness of any processor. Furthermore, by continually monitoring the performance of every node at execution—using its initial fitness as baseline—, one can detect performance variations.

The ASPARA methodology can be applied thence. In the programming phase, the application programmer is required to write sequential code for the body of each pipeline stage. In the compilation phase, the programmer makes a call to the pipeline skeleton to apply these stages to a set of inputs. Then, during the calibration phase, the system maps the stages to (a subset of) the nodes in the system and calculates a performance threshold. It may choose to map several stages to the same processor when this processor is more powerful than the others. During the execution phase, our system periodically checks the progress of the computation and decides whether to remap some or all of the stages based on the performance threshold. The performance improvements are illustrated using a pipeline with 8, 16, and 32 stages.

5.1.2 Related Work

A core topic in textbooks on parallelism and concurrency, the pipeline has been covered by several surveys which deal with its different aspects including: performance models for the general case and hardware implementations [165];

hardware-oriented performance modelling [75]; its exploitation via instruction scheduling in processors [7]; and different aspects in the conceptualisation and design of algorithms [126, 127].

While the general pipeline mapping problem is known to be intractable for a representative number of processors [155] and its full solution therefore remains indeterminate, different approaches have been conceived to address less-complex cases. Following the seminal work of Bokhari [36], where he first suggested its intractability and developed a heuristic algorithm based on a probabilistic pairwise method, Berman and Snyder [28] identified two orthogonal problem dimensions—the number of stages and the topology of the machine—and developed a solution for a variable number of stages in two separate parallel architectures.

Following Bokhari's lead for the instantiation of the problem in distributed systems [37], where he stated its NP-Complete complexity, a few algorithmic solutions for special cases have been independently formulated by several authors over the years [38, 174, 181]. Based on direct-acyclic graphs, the macro-pipelining method [17] gives a theoretical framework for scheduling parallel pipelines. While macro-pipelining provides guidance on the coarse distribution of work to different stages, its approach is limited to dedicated digital-signal processing systems. Another approach presents a multi-layer framework for the stage scheduling in dedicated real-time systems [50]. This work describes a series of steps to calculate end-to-end latencies based on a time-series model for a video-conferencing application.

Due to the importance of this construct, several of the skeletal and pattern frameworks listed in section 2.2 include a pipeline construct. In particular, we would like to draw our attention to the following recent advances:

- La Laguna Pipeline (llp) [95] furnishes a conceptual tool for static multi-stage allocation using algorithmic skeletons. By approaching the problem with a 0/1 knapsack problem method, llp is employed to develop a theoretical solution to stage scheduling. Interesting empirical results have been reported for heterogeneous systems using this analytical modeller in multi-cluster configurations with short-span tasks [8].

- Using process algebra to enable performance modelling, the eSkel and the ASSIST libraries have been augmented with performance-based mappings for distributed systems [22, 191]. In particular, eSkel has been successfully applied to the reformulation of the parallel solution of a problem in queueing theory [197].

- A series of heuristics have been developed for a simple pipeline skeleton executing on homogeneous and heterogeneous systems [26, 25].

- The Resource Optimisation Under Throughput rEquirements (ROUTE) method proposes a DAG-based mapping approach for a generic pipeline construct on heterogeneous systems [110], which has been applied in image processing [109].

Nonetheless, the main difference between our ASPARA–based pipeline and the aforementioned approaches is that it is intended to be adaptable by construct, and focused on empirical, system-infrastructure methodologies. Our pipeline can forecast and enhance the performance of a skeletal application by exploiting the knowledge of the skeletal structure while efficiently preserving the skeletal behaviour.

Our methodology is fundamentally different to the aforementioned results since it provides a generic systems-oriented methodology to:

- pragmatically tune up the pipeline parallelism skeleton regardless of the complexity of the stage functions, the system load, and the capacity of the nodes; and
- dynamically adapt to non-dedicated heterogeneous environments once the pipeline processing is established.

5.2 Adaptive Pipeline

The core of our adaptive pipeline is the instantiation of the ASPARA calibration and execution phases into the mapping of stages to processors and the feedback respectively.

5.2.1 Calibration

The purpose of this phase is twofold: calculate the stage-to-node mapping and determine the performance threshold which governs the feedback.

The assignment of stages to processors is widely known as stage mapping, or *mapping*, and finding the optimal mapping for a pipeline of any given length on a certain number of processors is the *mapping problem* [36]. This problem is also referred to in the literature as pipeline scheduling as, in general, the scheduling of parallel tasks deals with the order in which tasks are executed and their assignment to nodes.

The mapping problem can be enunciated as follows. Let n be the number of stages in a pipeline and P the total number of processors of a distributed system. A mapping M is a pair $(Chosen, \vec{x})$, where *Chosen* is the set of processors which are the fittest to execute the pipeline stages and $\vec{x}$ is the number of stages allocated to each of the processors in *Chosen*. Such mapping must

comply with properties (5.5) and (5.6).

$$Chosen \quad \text{is a finite set of processors and } Chosen \subseteq P \tag{5.5}$$

$$\vec{x} \quad \text{is a vector of size } |Chosen| \text{ such that } \sum_{\imath=1}^{|Chosen|} x_\imath = n \tag{5.6}$$

In our case, the first step in determining M consists of finding the fitness of each available processor. This is achieved by a times-only calibration, which runs an instance of f on each processor and measures its execution time. In practical terms, any stage will do, since we have assumed that all stages are equally representative in computational terms, making the execution time of this overall phase determined by the execution time of the slowest node. Note that if this were not the case, the overall complexity of the calibration would be greatly increased, derived from the number of combinations of stage functions and node fitness, with the augmented complexity of the changing conditions of the underlying infrastructure.

The calibration allows us to rank processors by descending fitness, i.e., by increasing calibration time. We can immediately discard all but the n fittest processors from this initial mapping M.

Formally, assume that M and $t_\imath$ are the initial mapping and its calibration time for node $\imath$ respectively. Properties (5.5) and (5.6) hold for M and, by construction, $|Chosen| = n$, $\vec{x} = \vec{1}$, and p_b, the last node in $Chosen$, is the bottleneck since $t_{\imath-1} \leq t_\imath \leq t_{\imath+1}\ \forall\, \imath \in Chosen$.

Then, a greedy strategy uses this initial (naïve) mapping M, in which one stage is assigned to each of the n fittest nodes, and iteratively tries to improve its stage allocation. It compares the impact of moving one stage from the processor p_b to the processor p_1, the first one in the active mapping one and therefore the one with lowest processing time. If the new processing time at p_1—the

product of the number of its allocated stages plus one and its original calibration time—is smaller than the original processing time at p_b then it makes the switch. The strategy then re-ranks p_1 according to its new processing time. Iteration proceeds until no further improvement is possible. We call the resulting mapping $M\prime$. Let us illustrate this with a simple example.

Example 1: Assume that $P = 32, n = 16$, and $\vec{t}$ stores the following 32 values, corresponding to the calibration times of an equal number of nodes $p_1, \ldots, p_{32}$:

$$\vec{t} = \left\{ \begin{array}{cccccccc} 22, & 13, & 29, & 16, & 5, & 11, & 1, & 19, \\ 17, & 29, & 14, & 31, & 36, & 15, & 10, & 25, \\ 19, & 18, & 30, & 18, & 20, & 24, & 31, & 25, \\ 22, & 23, & 32, & 33, & 29, & 38, & 19, & 17 \end{array} \right\}$$

Initially,

$$M = \left\{ \begin{array}{l} Chosen = \{p_7, p_5, p_{15}, p_6, p_2, p_{11}, p_{14}, p_4, p_9, p_{32}, p_{18}, p_{20}, p_8, p_{17}, p_{31}, p_{21}\} \\ \vec{x} = \{1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1\} \end{array} \right.$$

After the application of the greedy algorithm,

$$M\prime = \left\{ \begin{array}{l} Chosen = \{p_5, p_{15}, p_6, p_7\} \\ \vec{x} = \{2,1,1,12\} \end{array} \right.$$

Table 5.1 shows the steps performed by the execution of the greedy algorithm. •

In addition to the mapping, the calibration phase performs the calculation of the performance threshold, key to the adaptivity mechanism of the pipeline.

In the idealistic case of a totally-dedicated fully-homogeneous system, the threshold ought to be large, as the overall system will not benefit from a re-calibration since all nodes are equally fit. In a highly-dynamic system with highly heterogeneous nodes, the threshold must be small to allow rapid reaction to load changes while maintaining a balance to avoid unnecessary re-

Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}	p_{20}	p_8	p_{17}	p_{31}	p_{21}
$\vec{t}$	01	05	10	11	13	14	15	16	17	17	18	18	19	19	19	20
$\vec{x}$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}	p_{20}	p_8	p_{17}	p_{31}	
$\vec{t}$	01	05	10	11	13	14	15	16	17	17	18	18	19	19	19	
$\vec{x}$	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}	p_{20}	p_8	p_{17}		
$\vec{t}$	01	05	10	11	13	14	15	16	17	17	18	18	19	19		
$\vec{x}$	3	1	1	1	1	1	1	1	1	1	1	1	1	1		
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}	p_{20}	p_8			
$\vec{t}$	01	05	10	11	13	14	15	16	17	17	18	18	19			
$\vec{x}$	4	1	1	1	1	1	1	1	1	1	1	1	1			
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}	p_{20}				
$\vec{t}$	01	05	10	11	13	14	15	16	17	17	18	18				
$\vec{x}$	5	1	1	1	1	1	1	1	1	1	1	1				
Chosen	p_5	p_7	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}	p_{18}					
$\vec{t}$	05	01	10	11	13	14	15	16	17	17	18					
$\vec{x}$	1	6	1	1	1	1	1	1	1	1	1					
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9	p_{32}						
$\vec{t}$	01	05	10	11	13	14	15	16	17	17						
$\vec{x}$	6	2	1	1	1	1	1	1	1	1						
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4	p_9							
$\vec{t}$	01	05	10	11	13	14	15	16	17							
$\vec{x}$	7	2	1	1	1	1	1	1	1							
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}	p_4								
$\vec{t}$	01	05	10	11	13	14	15	16								
$\vec{x}$	8	2	1	1	1	1	1	1								
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}	p_{14}									
$\vec{t}$	01	05	10	11	13	14	15									
$\vec{x}$	9	2	1	1	1	1	1									
Chosen	p_7	p_5	p_{15}	p_6	p_2	p_{11}										
$\vec{t}$	01	05	10	11	13	14										
$\vec{x}$	10	2	1	1	1	1										
Chosen	p_5	p_{15}	p_7	p_6	p_2											
$\vec{t}$	05	10	01	11	13											
$\vec{x}$	2	1	11	1	1											
Chosen	p_5	p_{15}	p_6	p_7												
$\vec{t}$	05	10	11	01												
$\vec{x}$	2	1	1	12												

Table 5.1: Step-by-step illustrative execution of the greedy algorithm to refine a pipeline mapping.

calibrations. A small threshold can cause too many re-mappings (thrashing), while a very large one effectively deactivates the adaptiveness. Hence, we argue that the threshold ought to be automatically determined, based on the fitness of all nodes in the system.

We propose a performance threshold for our adaptive pipeline as the inverse standard deviation of the calibration times of all nodes in the processor pool. If we consider the differences between the times as a measure of the dispersion of the overall system, highly-dispersed—typically heterogeneous—systems have calibration times with a small threshold, which will trigger a node calibration with subtle variation in node performance. Conversely, more steady—typically homogeneous—systems present a large threshold which will not react as promptly to variations in node performance.

Assuming a normal distribution of the calibration times, their standard deviation, σ, can be calculated using equation (5.7), where $|P|$ and $\bar{t}$ are the number of nodes in the processor pool and the average calibration time respectively. As σ quantifies the dispersion of the calibration times, its inverse, presented in equation (5.8), can arguably be used as a measure of adaptivity, e.g., using the execution times shown for the 16 nodes in *Chosen* from the initial M in example 1, $\sigma = 5.404$ and $threshold = 0.185$.

$$\sigma = \sqrt{\frac{1}{|P|}\sum_{i=1}^{|P|}(t_i - \bar{t})^2} \tag{5.7}$$

$$threshold = \frac{1}{\sigma} \tag{5.8}$$

While the aforementioned procedure to automatically determine the threshold does not explicitly correlate the times with the actual node load, it provides an accurate steering criteria for adaptivity, as it is arguable that unchanging, ei-

Algorithm 5.1: Calibration Algorithm for the Pipeline

Data: f: Stage Functions;
n: Number of Stages;
P: Nodes;
Result: *Chosen*: Lookup table of fittest processing elements;
$\vec{x}$: Number of processes per *Chosen* node;
threshold: Performance threshold to drive adaptivity;

forall *nodes in* P **do**
  Execute f concurrently ;
  Set $t_\imath \leftarrow$ execution_time(f);
end
if *root node* **then**
  collect $t_\imath$ into $\vec{t}$;
  set *Chosen* $\leftarrow$ fittest(n,P); /* n fittest nodes from P based on $\vec{t}$ */
  set $\vec{x} \leftarrow \vec{1}$; /* The n entries in $\vec{x}$ are set to 1 (one stage per node) */
  set $M \leftarrow$ (*Chosen*, $\vec{x}$) ;
  set *threshold* $\leftarrow$ inverse_standard_deviation($\vec{t}$) ;
  set $\imath \leftarrow 0$;
  set $\ell \leftarrow n$;
  while $\imath < \ell - 1$; /* Start of greedy strategy */
  do
    set $flag \leftarrow false$;
    set $k \leftarrow \imath + 1$;
    while $k < \ell \ \wedge \ \neg flag$ **do**
      set $\alpha \leftarrow \lfloor \frac{t_k - t_i}{t_i / x_i} \rfloor$;
      if $\alpha \neq 0$ **then**
        set $t_i \leftarrow t_i + t_i / x_i$;
        set $x_i \leftarrow x_i + 1$;
        set $\ell \leftarrow \ell - 1$;
        insert_in_order (t_i,$\vec{t}$);
        set $flag \leftarrow true$;
      end
      if $\neg flag$ **then**
        $\imath \leftarrow \imath + 1$;
      end
    end
  end
  set $M\imath \leftarrow$ (*Chosen*, $\vec{(x)}$);
  send $M\imath$ and *threshold* to other nodes
else
  /* All other nodes */
  send $t_\imath$ to root node;
  receive $M\imath$ and *threshold*;
end

ther heavily- or lightly-loaded, systems will not benefit from a re-calibration.

In summary, the calibration stage performs the following steps:

1. records the execution time of the stage function in every node using $\vec{t}$;
2. initialises M with the first n fittest nodes, sorted using $\vec{t}$ as key and allocated one stage each;
3. calculates the *threshold* as the inverse standard deviation of $\vec{t}$;
4. employs a greedy strategy to determine the final mapping $M\prime$; and, finally,
5. broadcasts $M\prime$ and the *threshold* to all nodes

This is outlined in algorithm 5.1, which is effectively an instantiation of algorithm 3.1. This algorithm requires f, the stage function to be used in the calibration, n, the number of stages, and P, the processor pool. It generates $M\prime$, the final mapping, and *threshold*, the performance parameter to determine a performance bottleneck.

5.2.2 Execution

The purpose of the feedback phase is the detection of performance fluctuations in the pipeline and *reactively* triggering re-mappings.

Once the pipeline is in operation, this phase detects performance fluctuations by checking whether all processors are functioning according to the initial calibration. Each stage times itself and propagates its current t_i through the pipeline, piggy-backed with the real data. The final stage verifies that the T_{par} is acceptable by comparing with the original calibration times, using the performance threshold to determine acceptability. The threshold regulates the

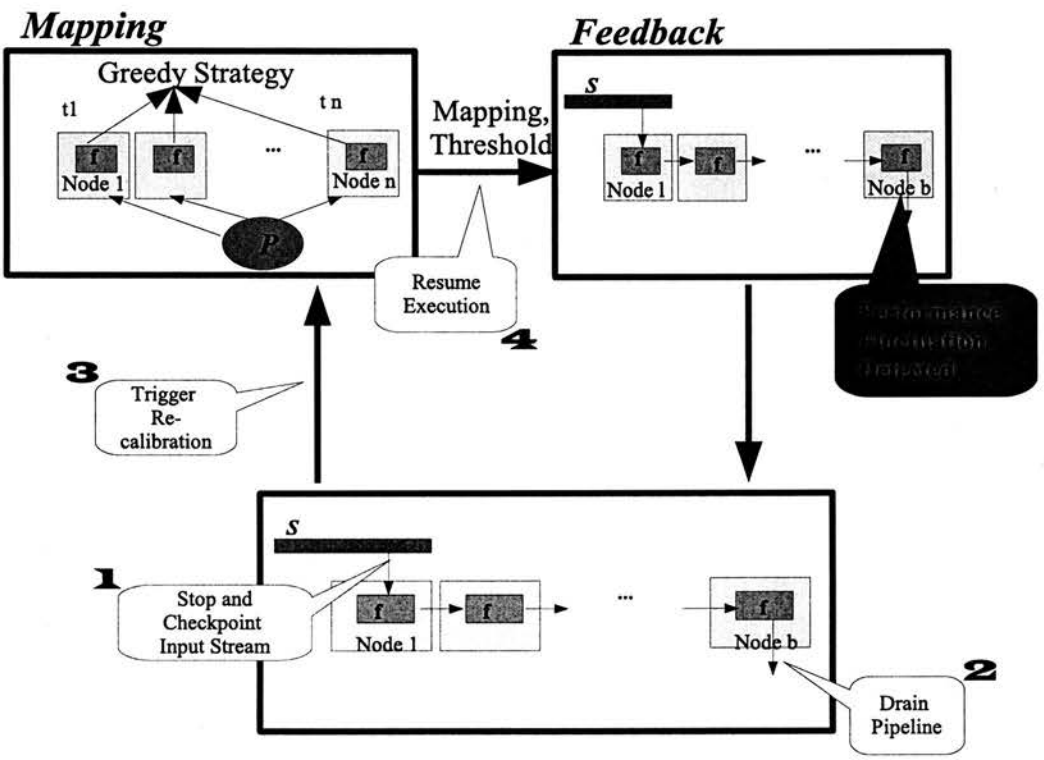


Figure 5.2: The operation of the feedback mechanism of our pipeline. Once a performance fluctuation is detected, it entails four steps: stop and checkpoint the input, drain the pipeline, trigger a re-calibration and, finally, resume the execution.

margin before a re-calibration takes place and is expressed as a fraction of the original value.

The principle is simple: assume that the threshold has been set at X during calibration and let t_i and t_b be the execution times for a certain stage i during normal operation and for the bottleneck at calibration respectively. By construction, t_b corresponds to the last node in the pipeline and is the worst time in the mapping. Thus, if $(1 + X) \times t_i > t_b$, a performance fluctuation beyond the threshold has been detected and a re-calibration is scheduled. Once this decision has been taken:

1. the input stream is stopped and check-pointed,

2. the pipeline is allowed to drain,

3. the re-calibration takes place, and

4. execution is resumed under the new mapping.

Figure 5.2 schematically illustrates this procedure. Note that the checkpoint is required to allow a complete re-calibration, otherwise the execution of the pipeline itself can produce additional load in the nodes, impairing the proper selection of the fittest processors.

While a re-calibration implies an overhead, we have empirically corroborated that the performance gain is superior as the pipeline throughput is determined by the bottleneck. It is important to highlight the the our empirical evaluation has indicated that the performance improvement becomes more representative as the input size increases—typically beyond 64 elements—, since the cost of a slow processor has a greater impact on the total execution time than a timely checkpoint and re-calibration.

5.3 Implementation

Based on our C/MPI platform, figure 5.3 presents the pipeline API. Following the notation defined in section 5.1:

- `stages` is an array of pointers to functions, which contains the f stages;
- `no_stages` stores n, the number of stages;
- `in_data` is a vector representing the input data stream S; and
- `comm` is an MPI communicator encompassing P, the complete processor pool

It is important to emphasise that the API provides only the pipeline structure, and its four externally-instantiated parameters regulate the pipeline behaviour, maintaining the philosophy of the skeletal paradigm.

We have based this design on explicit send-receive pairing. Internally, each stage is composed of an `MPI_Recv` call, the invocation to the f function, and an `MPI_Send` call. Pre-determined processors are not required for the execution of the pipeline. That is to say, after calibration, not even the `comm` root process must belong to the set of fittest processes.

Our pipeline is stateless and does not furnish a dedicated output data stream in order to allow the programmer to distribute the output according to the application requirements, taking advantage of the parallel capabilities of the underlying platform. Furthermore, this characteristic allows us not only to choose any processor in the pool for the initial mapping, but also to migrate to any other mapping at a re-calibration.

The processor pool is represented as a lookup table of active processors. Referred to as *Chosen* in algorithm 5.1, this table is built during the calibration

```
void pipeline(stage_t *stages, int no_stages, MPI_Datatype in_data[], MPI_Comm comm)
```

Figure 5.3: Application program interface to the algorithmic skeleton of our adaptive pipeline

process. It is also of particular importance during process migration, since the migration is, in essence, an exchange of its entries.

Since a key criterion is the provision of simple process migration with low overhead, we have opted for this simplistic swapping mechanism, in spite of the existence of generic libraries which provide MPI process migration. Mainly devoted to preserving index and context variables in loops, they address generic MPI programs, but their use requires:

- specialised underlying distributed filesystems (SRS) [187]; or
- daemon-based services (MPI_swap) [173].

In essence, our approach does not provide a built-in preservation mechanism for program variables, which in our case is not required due to the statelessness of our pipeline. We have provided an initial assessment of the overhead incurred by the re-calibration for a simple stateless pipeline application in figure 5.5. However, as our migration is performed on a statelessness basis, we are unable to objectively compare our approach with other SRS or MPI_swap, as those systems require dedicated software infrastructures but provide a more generic solution for the state-preserving case. It will require a different approach with a different software infrastructure to be able to provide a definite answer to this.

As part of the implementation, we have included two basic T_{par} estimators, which are based on equation 5.2. One, denoted as naïve, forecasts T_{par} using

the initial 1-to-1 stage-to-processor mapping. The other, denoted as calibrated, approximates T_{par} using the final refined mapping.

As the threshold and estimators require to be dynamically determined, we employ the GSL [88] on the calibration times. In particular, we use the `gsl_stats_sd` routine to calculate their standard deviation, `gsl_stats_mean` to compute their arithmetic mean, and `gsl_stats_max` and `gsl_stats_min` to find their maximum and minimum values. This statistical functionality can be overridden to allow fixed-value thresholds.

5.4 Experimental Evaluation

For reproducibility purposes, we have employed as stage function the `whetstones` procedure from the 1997 version [139] of the Whetstone benchmark with parameters $(256, 100, 0)$. It accounts for some 5 seconds of double-precision floating-point processing in an unloaded node of our experimental computational environment presented in section 3.3.

All variability in the system is due to external load and, to a lesser extent, to the difference in performance among processors.

Figure 5.4 shows an initial exploration of the parameter space, running pipelines with 2,4,8 and 16 identical stages, one per processor (note that a pipeline with more stages is doing more work in absolute terms) on increasingly large inputs. The execution times are primarily determined by S, the input size of the data stream, and are marginally influenced by the number of stages n in the pipeline as previously assumed in equation (5.2).

In order to initially estimate the overhead caused by the re-calibration itself, we have tested the correlation between the re-calibration time and the size of the pipeline. Figure 5.5 shows that the overhead is minimal and increases at

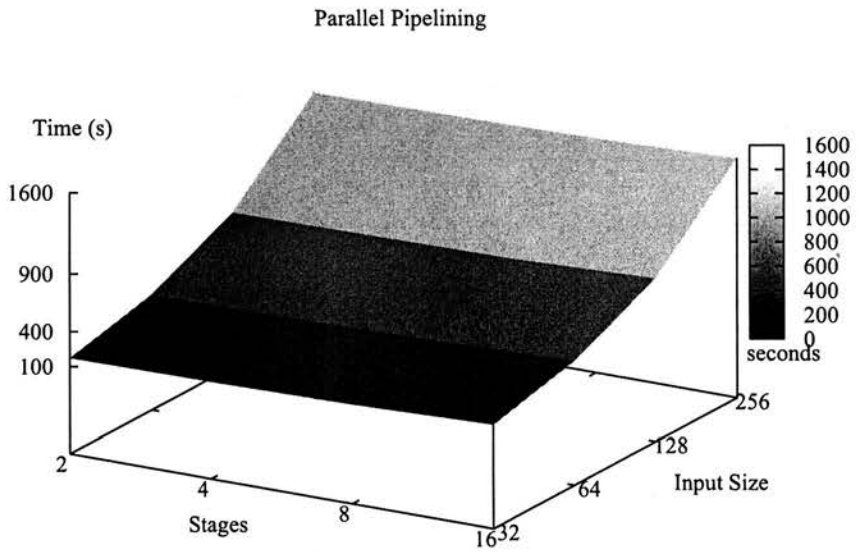


Figure 5.4: A sanity check on the pipeline correlating the data input size and the number of stages with the execution times (in seconds). The plotted surface corroborates the fact that the execution times are determined by the input data size and, to a much lesser extent, by the number of stages in the pipeline.

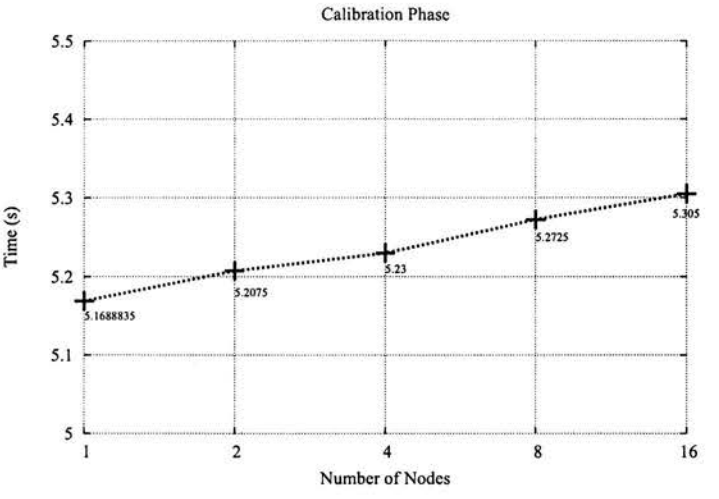


Figure 5.5: Estimate of the re-calibration overhead as a function of the pipeline size. The x-axis shows the number of nodes (stages) in the pipeline and the y-axis the actual re-calibration time. The time increases in less than one percent for every power-of-two increment in the pipeline size.

a slow rate ($< 1\%$ for every power-of-two increment in the number of stages). Although the actual calibration time is determined by the slowest processor in the entire pool, the total overhead increases with the number of stages, due to the pipeline drainage which occurs previous to the re-calibration and independent of the input data size.

These first two scenarios are simply sanity checks which have further verified that our pipeline works as expected. The reported execution times represent the average of three measurements and all the nodes employed in these two test series have no external processes or users.

We have then measured the performance impact of our system under controlled load conditions and fixed-value thresholds, 0.5 and ∞, for a given pipeline ($n = 8$ and $S = 128$).

Taking into account the fair CPU allocation algorithm used in Linux and to ensure the existence of changing load conditions, we have incrementally

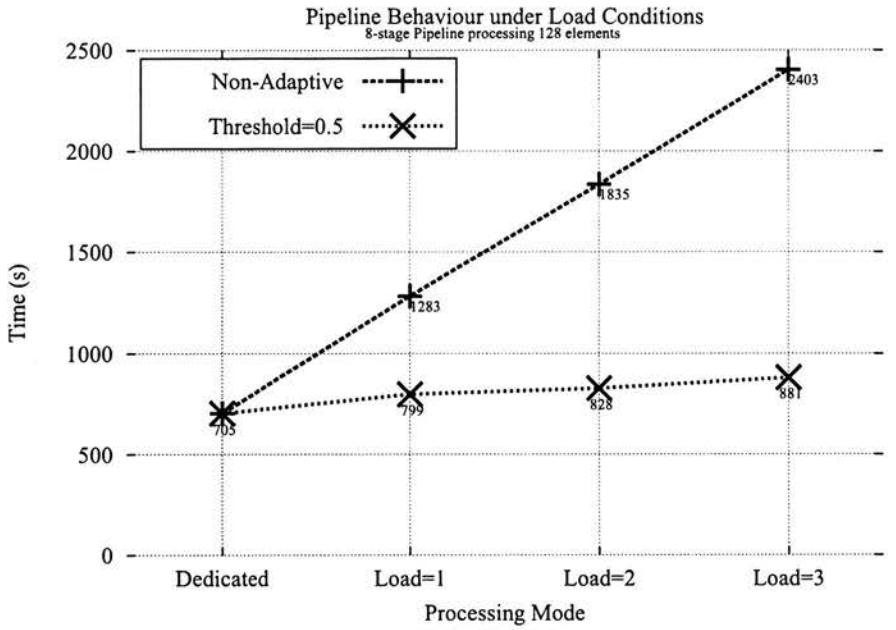


Figure 5.6: Comparison of the execution times, in seconds, for our adaptive pipeline under increasing, controlled load conditions using two thresholds. The upper dashed line plots the non-adaptive times ($threshold = \infty$) and the bottom fine-point curve represents the fixed-threshold times ($threshold = 0.5$).

injected load dynamically into the system using a simple load generation program based on algorithm 3.3. Each instance of this program added 1 to the load displayed by the Linux `uptime` command in a certain node until this node became the bottleneck, while the rest of the processors did not experience any significant load variation. Thus a $Load = 0$ implies that no instances of the load generator were being executed, $Load = 1$ that an instance of the load generator was running on the bottleneck and so on. The instances were triggered after 60 seconds from the start of the program. The threshold was artificially fixed at ∞, which implies a non-adaptive pipeline, and at 0.5.

Figure 5.6 shows a comparison of the measured execution times. The x-axis indicates the injected load, i.e., the number of instances of the load generator running on one processor, which were triggered during the pipeline operation.

The adaptive approach has responded well under varying load conditions, since the execution times in the non-adaptive parametrisation have increased at a considerably higher rate than the adaptive ones.

However, in this initial analysis, we have controlled the load injection and arbitrarily fixed the threshold at 0.5 in a given pipeline. In order to assemble a more complete set of results, let us examine a typical execution log from our adaptive pipeline as shown in listing 5.1.

Firstly, our pipeline provides a disclaimer message which includes the version employed (lines 1–3), the time of day (line 4), and the values of n and $S+1$ respectively (line 5). In this case, $S = 512$ as the final character is a dummy, to signal the end of stream.

Secondly, during the calibration phase, it logs the calibration times for all nodes in P (lines 7–15) and their associated statistical values: arithmetic mean, standard deviation, inverse standard deviation, largest value, smallest value, and the threshold (lines 16–18). This particular threshold allows a 35.4714% performance variation.

It then prints the *Chosen* pool of processors (lines 19–28). Figure 5.7 illustrates the sequence of steps performed by the greedy strategy of algorithm 5.1 in order to refine the mapping M into $M\prime$.

$$M = \begin{cases} Chosen = \{bw240n04,\ bw240n02,\ bw240n03,\ bw240n21, \\ \ bw530n05,\ bw530n10,\ bw530n07,\ bw240n23\} \\ \vec{x} = \{1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1,\ 1\} \end{cases}$$

$$M\prime = \begin{cases} Chosen = \{bw240n21,\ bw530n05,\ bw240n04,\ bw240n02, \\ \ bw240n03\} \\ \vec{x} = \{1,\ 1,\ 2,\ 2,\ 2\} \end{cases}$$

Notice that the process-to-node values illustrated in lines 8–15 and 21–28

Listing 5.1: Execution Log of the Adaptive Skeletal Pipeline

```
Adaptive Skeletal Pipeline -- version 3.3
Copyright (C) 2006-7 Horacio Gonzalez-Velez, U.Edinburgh
This program comes with ABSOLUTELY NO WARRANTY
Wed Aug  1 18:30:08 2007
Initializing: Stages= 8, Input Size= 513...

[Process]  Hostname              Time           Stages/CPU
[14]      bw240n04.inf.ed.ac.uk   4.24372        1
[00]      bw240n02.inf.ed.ac.uk   4.25524        1
[07]      bw240n03.inf.ed.ac.uk   4.36799        1
[21]      bw240n21.inf.ed.ac.uk   4.73571        1
[35]      bw530n05.inf.ed.ac.uk   8.21054        1
[49]      bw530n10.inf.ed.ac.uk   9.10425        1
[42]      bw530n07.inf.ed.ac.uk   9.25595        1
[28]      bw240n23.inf.ed.ac.uk   11.18540       1
## mean          sd              inverse_sd       largest          smallest
6.919850         2.819169         0.354714         11.185399        4.243722
 THRESHOLD = 0.354714
 CHOSEN PROCESSORS
[Process]  Hostname                Time           Stages/CPU
[21]      bw240n21.inf.ed.ac.uk    4.73571        1
[35]      bw530n05.inf.ed.ac.uk    8.21054        1
[14]      bw240n04.inf.ed.ac.uk    8.48744        2
[00]      bw240n02.inf.ed.ac.uk    8.51047        2
[07]      bw240n03.inf.ed.ac.uk    8.73599        2
[49]      bw530n10.inf.ed.ac.uk    9.10425        0
[42]      bw530n07.inf.ed.ac.uk    9.25595        0
[28]      bw240n23.inf.ed.ac.uk    11.18540       0

Calibration time: 11.14 seconds
Wed Aug  1 18:30:19 2007
Estimated execution times: (N)aive 5771.10 (C)alibrated 4412.77
Worst stage time N=11.18540 C= 8.51047

[08]:.
Reschedule programmed
SLOW: Proc. [15] Time 12.1959 secs
Wed Aug  1 18:47:13 2007
[08]:.
RE-CALIBRATING...


Total Processing time: 5037.65 seconds
Wed Aug  1 19:54:06 2007
```

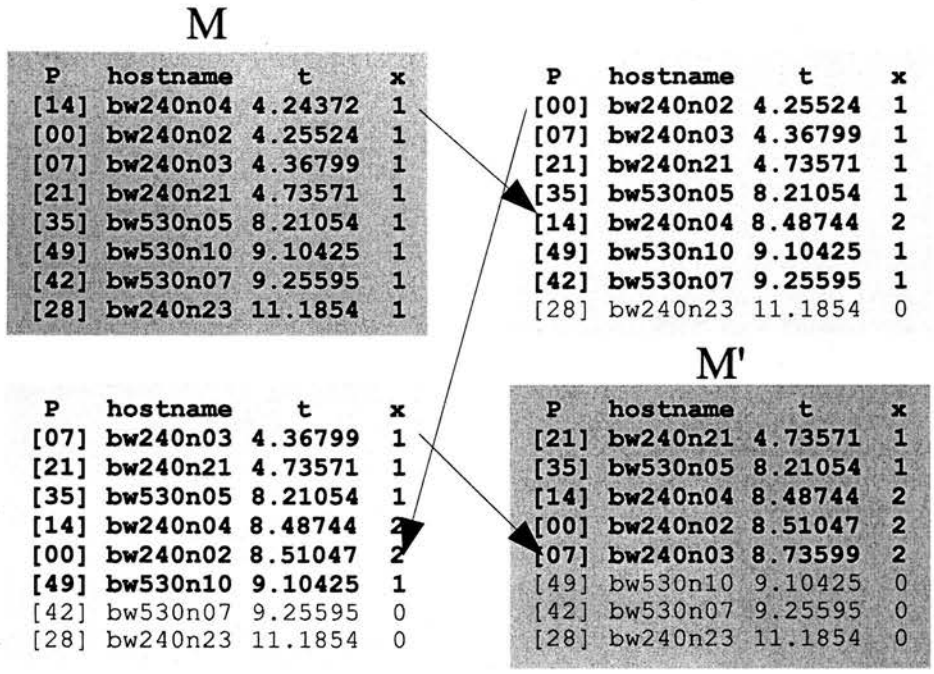
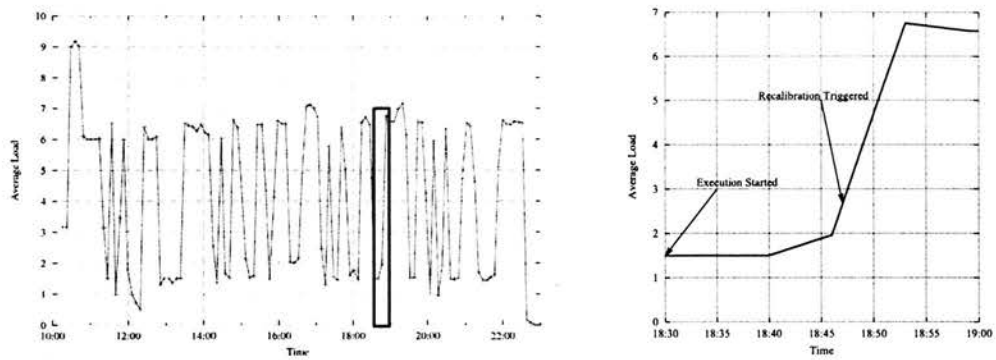


Figure 5.7: Pipeline mapping refinement through the greedy strategy derived from algorithm 5.1

are multiples of 7. This is the maximum number of processes per node determined in the LAM/MPI hostfile used to initialise the MPI environment and is totally independent of the pipeline implementation.

During the last step of the calibration, our pipeline records the total execution time of the calibration phase (line 30) and the time of day when the calibration was completed (line 31). Note that, as expected, the total calibration time, 11.14*s*, is determined by the calibration time of the slowest stage, 11.18540*s*. Then, our pipeline estimates T_{par} (line 32) for the naïve and calibrated cases, using the corresponding bottleneck times (line 33). Equations (5.9) and (5.10) detail these calculations based on equation (5.2). We will employ T_{Naive} as a baseline to evaluate the performance of our pipeline.



(a) Full (b) 18:30–19:00

Figure 5.8: Illustrative example of the responsiveness of the pipeline execution. Section (a) of the figure shows the full load pattern for the `bw240n04` node for the given day (10:00-23:59hrs). Section (b) presents an amplified view (18:30-19:00hrs), which shows the start of the program execution and that of the re-calibration once the threshold has been reached.

$$T_{Naive} = \tag{5.9}$$
$$4.24372 + 4.25524 + 4.36799 + 4.73571 + 8.21054 + 9.10425 +$$
$$9.25595 + 11.18540 + 11.18540 \times 511 = 5771.1$$

$$T_{Calibrated} = \tag{5.10}$$
$$4.73571 + 8.21054 + 8.48744 \times 2 + 8.51047 \times 2 +$$
$$8.51047 \times 2 + 8.51047 \times 511 = 4412.8$$

Thirdly, our pipeline prints the date to signal the start of the execution phase (line 35). In this case, the `[08]` value is the process number where the last stage is executed, i.e., the second process allocated to node `bw240n03`. This is printed repetitively per each entry in input data stream. This line has been truncated for readability.

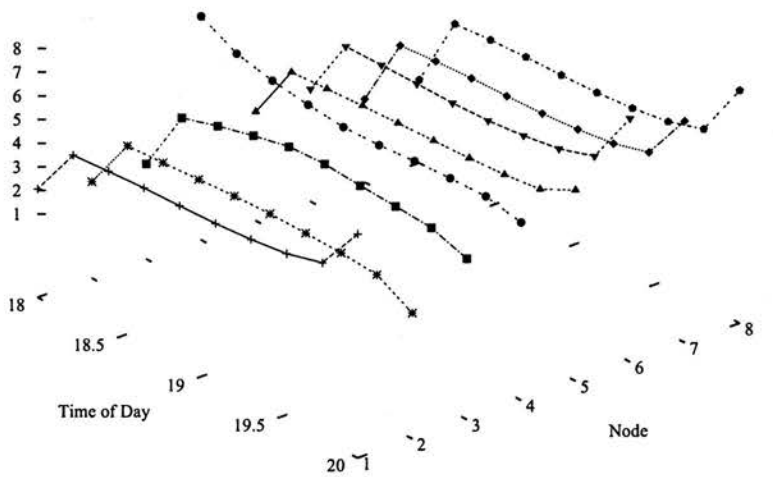


Figure 5.9: An amplified view of the system load (18:30-20:00), which corresponds to the execution of the example shown in listing 5.1.

Then, there is a performance variation beyond the defined threshold during execution which has triggered a re-calibration (line 36). In this case, processes 15, the second process allocated to node `bw240n04`, has executed its stage on 12.1959*s* or a 43.7% performance variation (line 37). This has been detected at 18:47 on 1/Aug/07 (line 38). The execution continues to drain the pipeline (line 39), until the actual re-calibration starts (line 40). Once this is done, the standard calibration information, similar to lines 7–33, is printed, and the execution continues. The remaining part of the execution has been omitted here for manageability.

We shall now examine the responsiveness of the pipeline to the load variation. As per lines 37–38 of listing 5.1, we know that there was a re-calibration triggered by node `bw240n04` at 18:47. Figure 5.8 presents the load pattern for this node. Part (a) presents its pattern for the entire day, highlighting the re-

gion where the re-calibration was triggered, and part (b) shows an amplified view of that region (18:30–19:00).

Finally, at the end of the execution, the total processing/clock time, 5037.65*s*, is printed (line 43), along with the time of day (line 44). Note that the final time is different from the estimators shown in line 32, as, in this case, the overall system load has increased during the execution of the program, as depicted in figure 5.9. Except for node 4, all nodes steadily increased their load during the life of this execution, with an average load of 2.3 at the start and 5.8 at the end. In fact, there were two additional re-calibrations, activated by major load fluctuations, with resulting thresholds of 0.307956 and 0.391052.

5.4.1 Bulk Experiments

In this vein, we have assembled an empirical evaluation using three node pools, $n = \{8,16,32\}$, handling five input data sizes $S = \{32,64,128,256,512\}$. The fifteen scenarios were staged on three different days, based on the number of nodes, with each scenario yielding to the execution of five experiments. To make the execution of each of the twenty-five experiments independent in a day, we have interleaved their executions according to the input data size, ensuring that just one pipeline experiment was executed at a time.

Each entry in table 5.2 presents the average of three experiments along with its coefficient of variation (cv), the average threshold for the series, and the naïve time estimate.

The obtained figures attest to the performance improvements attributed to the use of our pipeline.

Figure 5.10 presents a summary of the overall performance gain due to its use. Our empirical evaluation demonstrates a correlation between the size of

n	S	32	64	128	256	512
8	*Adaptive*	347.69s	672.69s	1289.56s	2501.27s	5193.20s
		$cv = 4.82\%$	$cv = 5.51\%$	$cv = 4.62\%$	$cv = 3.18\%$	$cv = 2.75\%$
	Threshold	0.47	0.38	0.33	0.36	0.36
	Naïve	344.43s	778.58s	1665.33s	2994.19s	5891.82s
16	*Time*	399.09s	714.18s	1384.75s	2617.53s	5311.50s
		$cv = 5.03\%$	$cv = 5.89\%$	$cv = 0.56\%$	$cv = 0.87\%$	$cv = 1.33\%$
	Threshold	0.45	0.50	0.44	0.42	0.40
	Naïve	436.10	711.38s	1532.29s	2734.57s	6304.72s
32	*Time*	495.84s	821.76s	1467.12s	2774.43s	5560.20s
		$cv = 2.34\%$	$cv = 0.57\%$	$cv = 0.60\%$	$cv = 7.57\%$	$cv = 2.79\%$
	Threshold	0.53	0.55	0.56	0.47	0.51
	Naïve	493.02s	774.13s	1381.95s	3323.52s	6533.42s

Table 5.2: Listing of the execution times of the adaptive pipeline using different pipeline ($n = \{8,16,32\}$ and input $|S| = \{32,64,128,256,512\}$) sizes.

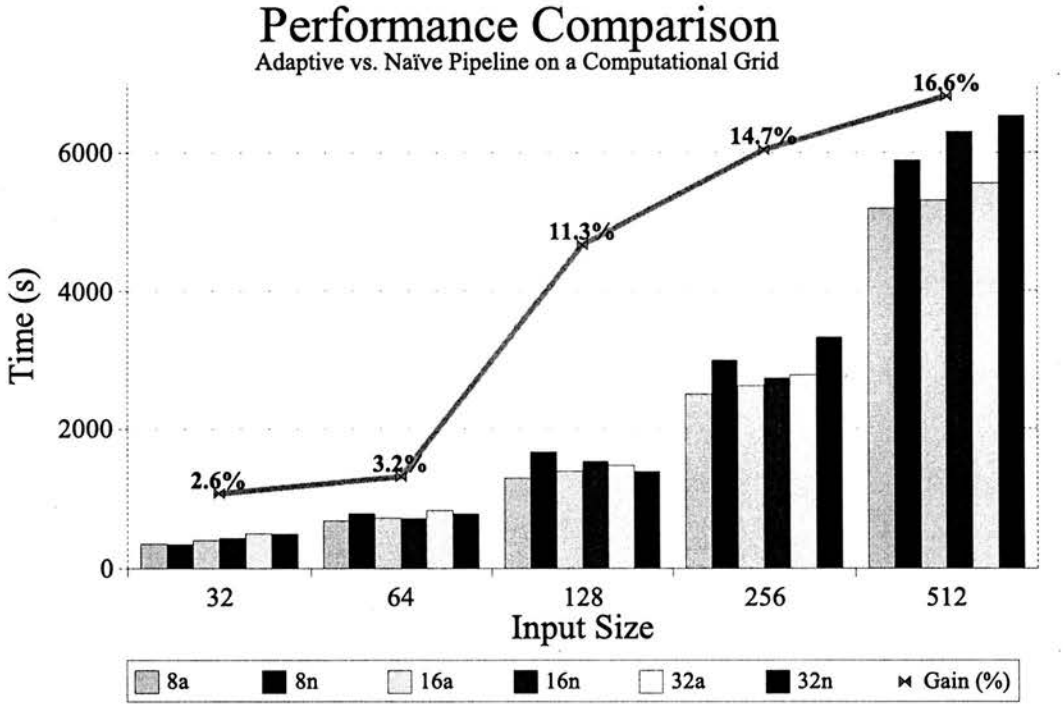


Figure 5.10: Execution time summary and overall improvement due to the use of our adaptive pipeline. The x-axis indicates the number of elements processed by the pipeline (input size). Each shaded bar plots the execution time for a given pipeline size and scheduling method (adaptive or naïve). The top thick line represents the aggregated gain as a percentage. Key: [processors no.][scheduling], e.g., 8a means 8-stage pipeline and adaptive scheduling model and, analogously, 8n represents 8-stage naïve scheduling.

the data input, S, and performance gains, as load variations impact more as the processing increases. While the estimate presumes an execution of the pipeline under steady conditions and neglects any load variation, it is a guideline for quantifying the performance improvements.

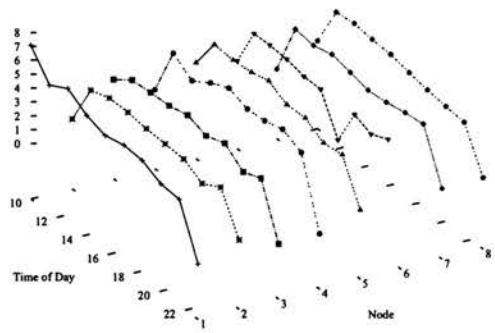
In the course of the 3-day empirical evaluation, our multi-cluster configuration entertained different workloads from a multiplicity of users. Such workloads provided a realistic environment in which to assert the adaptivity of our pipeline. To monitor the overall system load, we recorded the the Linux `uptime` command output (1-minute reading) during the entire duration of the 75 experiments. Figure 5.11 shows the results. Each row presents two views from an evaluation day.

Left A 3-D graph with each node load during the entire day. The x-axis specifies the time of the day and the y-axis the nodes. The different load readings are plotted in the z-axis using different line styles

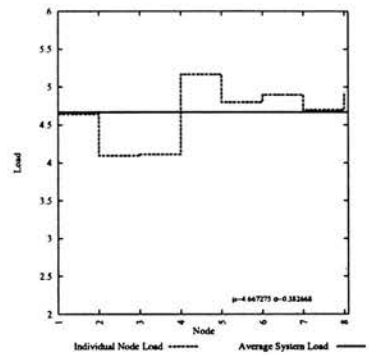
Right A summary chart with the average load of each node and of the whole system. The x-axis designates the node and the dashed steps in the y-axis indicate the average load per node. The solid line in the middle indicates the average of the overall system load

The load average, μ, and the load standard deviation, σ, are written in the bottom-right corner of charts (b), (d), and(f) of figure 5.11. It is self-evident that the highest load variation was detected with 32 nodes ($\sigma = 0.78$) and the lowest with 16 ($\sigma = 0.19$). Needless to say, it has been encouraging that our pipeline adapted well under these dynamic conditions.

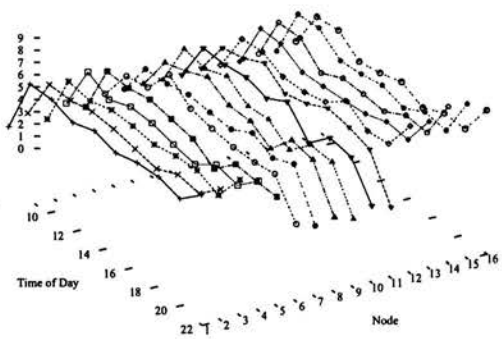
Consistent with our initial reality check, these experiments have systematically demonstrated that the execution time increases with the number of stages for a fixed input size. This is due to the pipeline filling-up and draining times



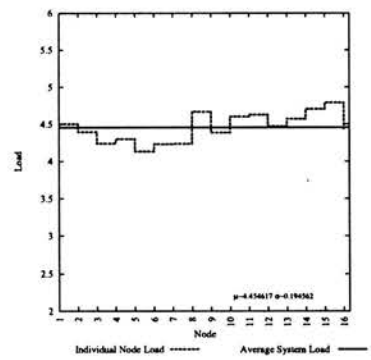
(a) 8p



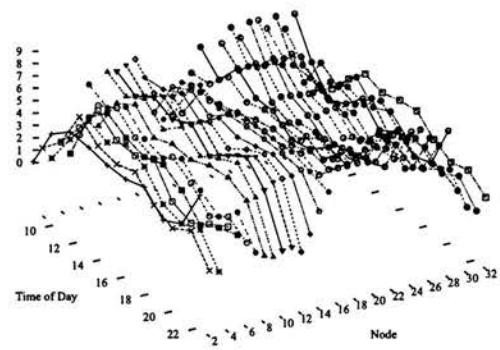
(b) 8p



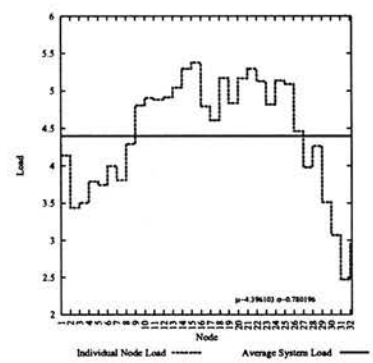
(c) 16p



(d) 16p



(e) 32p



(f) 32p

Figure 5.11: Load patterns for the 8, 16, and 32 node configurations.

during normal execution and re-calibration. This can also be exacerbated by the allocation of the different stages across multiple administrative domains, particularly for the case of a re-calibration, which implies an execution barrier.

It is crucial to emphasise that these comparative figures have been based on total execution time, as opposed to throughput, because our evaluation is concerned with a complete pipeline application with a fixed-size input on a given number of stages. As mentioned in section 5.1, throughput provides a completion rate, which is relevant for stream processing, i.e., continual pipeline processing of an infinite-size input, where the pipeline filling-up and draining times are irrelevant.

5.5 Discussion

In this chapter we have proposed an adaptive parallel pipeline which follows the ASPARA methodology presented in chapter 3. Initially, the calibration phase maps stages to the best processors available in the system, and calculates a performance threshold based on their processing times. Subsequently, a feedback mechanism monitors the behaviour at run-time and re-maps the stages to other processors if the threshold is surpassed. This pipeline is implemented as an algorithmic skeleton using a variable-size input data vector and a stage function array. We have evaluated its efficiency using a numerical benchmark stage function in a non-dedicated computational environment.

5.5.1 Outcomes

The adaptive approach has responded well under varying load conditions, since the execution times in the non-adaptive parametrisation have increased

at a considerably higher rate than the adaptive ones.

With respect to the analysis of the mapping problem, the findings of this chapter provide an alternate approach, using an application-oriented calibration in order to forecast resource utilisation. This tacitly reinforces the notion that although computational grids are highly dynamic, forecasts based on historical resource utilisation can accurately provide guidance to distribute workloads.

From a performance standpoint, it is arguable that the overall performance of our pipeline pattern improves as long as variations in the bottleneck stage are controlled through periodic performance monitoring and process migration. Note that our pipeline reacted promptly to the load variations, including flurries, preventing the development of a bottleneck for the pipeline processing. This is particularly useful, as workload flurries—repetitive activity surges, caused by a single user, which dominate the workload for a relatively short period—have been traced as a cause of performance disruption, and many performance models filter them out [185].

It is important to emphasise that our evaluation has covered load variations attributable to different processing capabilities, while preserving the complexity of the stage function constant. Although this scenario does not comprehensively address all possible pipeline applications, it certainly provides guidance on the behaviour of the general case on distributed systems.

A meticulous examination of the methodology reveals that there is room for a more instrumented approach to the determination of the re-calibration threshold. Nonetheless, our work provides evidence that the proposed adaptive method enhances pipeline parallelism performance: execution times are almost an order of magnitude greater when the adaptive pipeline is not used.

Chapter 6

Conclusion

6.1 Introductory restatement

While simulation and theoretical formulations have traditionally provided a preponderant foundation for generic approaches in the study of structured parallelism, there is a clear need for empirical work to inquire into the performance of skeletons.

Although the performance modelling of algorithmic skeletons has been extensively studied in the past, the use of the skeletal structure for performance models has remained relatively unexplored. In particular, scant research has been devoted to investigate the correlation between the skeleton structure, and the application performance from a pragmatic standpoint. It is arguable that such correlation requires some further study, in light of the possibility of devising generic strategies to enhance the performance of skeletal parallel programs, executing in non-dedicated heterogeneous systems.

6.2 Consolidation of Research Space

In this work, we have investigated the feasibility of using the information provided by the structure of an algorithmic skeleton to enhance the performance of the corresponding parallel program during execution. Being agnostic to the skeleton behaviour, our 4-step methodology, ASPARA, has deployed a pragmatic approach in order to instrument the parallel application at compilation and allow the resulting parallel program to adapt at execution.

Having been conceived to function under the dynamic conditions of a non-dedicated heterogeneous distributed system, we have evaluated ASPARA under non-controlled circumstances in an open computational environment. This thesis has employed two crucial task-parallel skeletons, the task farm and the pipeline, to illustrate the feasibility of ASPARA.

Task Farm Implementing a parameter sweep application, this skeleton has evaluated ASPARA using two different scenarios: a single-round scheduling, which employed times-only and statistical calibrations with no feedback, and a multi-round scheduling with times-only calibration and periodic adaptation throughout the execution. In addition to this continual adaptivity, this case has illustrated the ability to automatically discriminate between the multi- and single-round scheduling by introducing an installment factor based on the dispersion of the calibration times of the participating nodes.

Pipeline Using a known benchmark code, this skeleton assessed ASPARA using times-only calibration with a reactive strategy to adapt at execution. By defining a threshold based on the dispersion of the calibration times of the nodes, the skeleton has been able to expeditiously trigger a re-

calibration to correct performance deviations and favourably conform to resources variations in the system.

Our main findings can be summarised as:

1. the introduction of resource-awareness to a skeleton, using its structural information and without altering its behaviour, has consistently improved the performance of the resulting parallel program; and

2. the generation of automatic scheduling strategies without the need of application foreknowledge, e.g. benchmarks or execution traces, is not only feasible, but also efficient.

The former confirms our initial hypothesis as expressed in section 1.1, and the latter derives from the course of action of this research as the calibration-execution phase pairing has become in essence a scheduling scheme which is both application-agnostic and automatic. While the application independence is inherent to the skeletal paradigm, the self-direction has been achieved through sampling and statistical functions which reflect the state of the nodes and the actual requirements of the application at hand.

6.3 Retrospective Analysis

This section furnishes a self-critical review of the doctoral project. It is presented in an informal manner, as it relates the occurrences which determined our specific choices and, ultimately, defined the course of the project.

After the initial inspection of the state of the field, initially adumbrated in [103] and subsequently updated for the literature review in chapter 2, we came up with the initial thesis proposal during the first year of the project.

At that point, the proposal entailed the introduction of resource awareness into parallel programs based on task-parallel skeletons, but did not explicitly contained any specifics.

Following the attendance to the international summer school on grid computing and a research visit to the HLRS Stuttgart, we started to work on the introduction of a heuristic definition of the task size for a single-round task farm [98]. This became in turn the first significant result of the investigation, as reported in [96]. It proposed to characterise the workload resource requirements as a parametric function using NWS readings in order to determine the task size at once. The relevance of this work is indicated by a couple of organic citations from two different research groups in the States, the NeuronGrid group from Georgia Tech [44] and the Foster's group at Argonne National Labs/University of Chicago [164], and a related citation from the eSkel group at Edinburgh [197].

However, the actual characterisation was a parametric expression, defined before the task farm execution, and subsequently fed into the task farm skeleton. Such 'off-line' characterisation while useful, could easily become superannuated due to resource variability. As a result, the dynamic calculation of parametric expression using a calibration phase was introduced. It executed a sample of the workload on the processor pool and defined the task sizes accordingly. Since then, a more complete empirical study was later carried out, using the calcium current parameter-sweep, and published as a journal article [99].

The pipeline skeleton was later developed using the same conceptualisation of a calibration phase as the execution of a sample of the workload in order to select the fittest nodes from the pool, as described in [100].

At this point, ASPARA was fully conceptualised as a generic four phase

methodology—which included programming, compilation, calibration, and execution—and reported in [101]. From then on, most of the effort was concentrated on obtaining empirical results and introducing the feedback, either periodic or reactive, to the execution phase. The latest paper [102] described the empirical results for the pipeline skeleton.

6.3.1 Lessons Learnt

Working with real application is very time-consuming We were able to evaluate the resource awareness capabilities using a real-world application, and demonstrate the applicability of this investigation and yielded to two publications in computational cell biology [104, 105]. Nonetheless, it is also important to say that it was a lengthy endeavour, which could not be replicated for the pipeline case, as it could have delayed the investigation.

Working with non-dedicated environments is challenging As documented along with the case studies, the system load patterns greatly varied during the empirical evaluation. While this provided an interesting testbed, it made our runs virtually irreproducible. Demonstrated via the coefficient of variation, the statistical relevance of our results proved to be acceptable for the external reviewers in our contributions to specialised conferences.

Publishing partial results is beneficial Continual publication of partial results of this investigation greatly helped to compile, in a structured fashion, the final thesis, as well as provided feedback from independent reviewers.

Administrative issues are complex Our initial experiments were executed on

a geographically-distributed heterogenous multi-cluster system. While the results obtained were interesting, the administrative matters were daunting: it took several days to have both sides, Edinburgh and Stuttgart, to first agree on the network set-up—even though there has been a bilateral cooperation agreement in place for a number of years—, and then such configuration was available for a very limited period of time. The UK's National Grid Service did not have, nor plan to have, node co-allocation and resource monitoring tools configured with its Globus infrastructure, at least during the initial years of the project. In a nutshell, it could have been interesting to execute all experiments using a multi-site configuration, potentially located across several countries in the world, but, as it stands, the administrative side is lagging far behind the technology.

6.4 Practical Application

In addition to the immediate possibility of parallelising applications with the developed skeletons, we believe this thesis can be applied in computational biological patterns and scheduler evaluation.

6.4.1 Computational Biological Patterns

As illustrated by our computational biology application, the use of resource-aware skeletons has yielded to the effective implementation of real-world applications in different fields of study. Their efficient use of computational resources poses them in a preeminent position to effectively enable parallel applications in heterogeneous distributed environments.

Hence, the development of resource-aware resolution skeletons for systems biology can be further explored. This may create *computational biological patterns*, which systematically name, explain, and evaluate a recurring design in systems biology, analogous to a software pattern. The identification of these biological patterns, from a computational standpoint, is key to characterise the behaviour of a biological system. As an example, when the structure and properties of a set of regulatory circuits have been realised, it is crucial to be able to characterise their pattern to "reveal a possible evolutionary family of circuits as well as a periodic table for functional regulatory circuits." [128]

The ultimate objective is to produce a "pattern-based" framework to allow biological models—expressed as biological patterns—to be implemented as parallel applications using computational biological patterns—in the form of adaptive algorithmic skeletons. This patterns will help bench scientists to execute multiple parametric simulations of a biological system. Based on their biological patterns, different models will be able to be evaluated against a given hypothesis. These different models, comprised into the representative set of algorithms, can be deployed using adaptive skeletons in a heterogeneous distributed system. In particular, we strongly believe this has immediate application to the modelling of cellular processes.

6.4.2 Scheduler Evaluation

From its conception, ASPARA has been designed to include application-agnostic APIs, which are automatically parameterised at execution time, according to the application and node dispersion. To this end, ASPARA comprises a series of statistical methods of processor scheduling heuristics.

ASPARA lays only the foundations for the skeleton scheduling, and it can

be easily conceived as an evaluation framework where different scheduling schemes can be plugged into, evaluated, and incorporated into an algorithmic skeleton.

6.5 Final Remarks

The ASPARA methodology certainly merits further investigation, especially in the development of a generic calibration phase which can incorporate functions with different complexities. While the complexity of such functions suggests heuristical approaches, they may shed additional light on the different calibration options.

As discussed, an open possibility is the use of ASPARA combined with resolution skeletons in order to encapsulate domain-specific knowledge for the development of resource-aware applications and, eventually, of frameworks for a given field of study. Furthermore, as a resource-aware resolution skeleton would reflect the properties of the application at hand, one could speculate on the possibility of developing ad-hoc resource-aware constructs for a determined field of study, e.g., resource-aware computational biology patterns.

As a byproduct, this research has provided a useful input to the application of MPI collectives in geographically-distributed computational grids, as their performance has dwindled as the number of nodes increased in our initial experimentation using the Stuttgart-Edinburgh configuration. As the study of collective constructs remains an active area of research in parallel computing [49, 160], this reinforces the need for the development of topology and resource-aware MPI collectives.

In summary, whatever issues remain in the management of system resources, there exists little reason to doubt the resource availability-performance premise.

Within a framework in which resource availability is treated as an unobserved variable, the data seems fully compatible with the hypothesis first introduced in section 1.1, provided the right load patterns are acknowledged.

Certainly, acceptance of the hypothesis does not yield a generic resource characterisation function for parallel programming, since no general equation for system performance using resource variables has been developed. Hence, empirical studies, such as this one, will continue to shed light on the system performance research area.

Bibliography

[1] ABRAMSON, D., BUYYA, R., AND GIDDY, J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Futur. Gener. Comp. Syst. 18*, 8 (2002), 1061–1074. [Cited p. 29].

[2] ALBA, E., ALMEIDA, F., BLESA, M., CABEZA, J., COTTA, C., DIAZ, M., DORTA, I., GABARRO, J., LEON, C., LUNA, J., MORENO, L., PABLOS, C., PETIT, J., ROJAS, A., AND XHAFA, F. MALLBA: A library of skeletons for combinatorial optimisation. In *Euro-Par 2002* (Paderborn, 2002), vol. 2400 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 927–932. [Cited p. 20].

[3] ALBA, E., ALMEIDA, F., BLESA, M., COTTA, C., DÍAZ, M., DORTA, I., GABARRÓ, J., LEÓN, C., LUQUE, G., PETIT, J., RODRÍGUEZ, C., ROJAS, A., AND XHAFA, F. Efficient parallel LAN/WAN algorithms for optimization. the MALLBA project. *Parallel Comput. 32*, 5-6 (2006), 415–440. [Cited p. 31].

[4] ALDINUCCI, M., DANELUTTO, M., DÜNNWEBER, J., AND GORLATCH, S. Optimization techniques for skeletons on grids. In *Grid Computing and New Frontiers of High Performance Computing*, L. Grandinetti, Ed., vol. 14 of *Advances in Parallel Computing*. Elsevier, 2005, pp. 255–273. [Cited p. 31].

[5] ALDINUCCI, M., DANELUTTO, M., AND TETI, P. An advanced environment supporting structured parallel programming in Java. *Futur. Gener. Comp. Syst. 19*, 5 (2003), 611–626. [Cited p. 20].

[6] ALDINUCCI, M., PETROCELLI, A., PISTOLETTI, E., TORQUATI, M., VANNESCHI, M., VERALDI, L., AND ZOCCOLO, C. Dynamic reconfiguration of grid-aware applications in ASSIST. In *Euro-Par 2005* (Lisbon, Aug. 2005), vol. 3648 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 771–781. [Cited p. 31].

[7] ALLAN, V. H., JONES, R. B., LEE, R. M., AND ALLAN, S. J. Software pipelining. *ACM Comput. Surv. 27*, 3 (1995), 367–432. [Cited p. 100].

[8] ALMEIDA, F., GONZÁLEZ, D., MORENO, L. M., AND RODRÍGUEZ, C. Pipelines on heterogeneous systems: models and tools. *Concurr. Comput.-Pract. Exp. 17*, 9 (2005), 1173–1195. [Cited p. 101].

[9] ARMSTRONG, B., AND EIGENMANN, R. A methodology for scientific benchmarking with large-scale applications. In Eigenmann [77], ch. 3, pp. 109–127. [Cited p. 51].

[10] ARMSTRONG, R., GANNON, D., GEIST, A., KEAHEY, K., KOHN, S. R., MCINNES, L. C., PARKER, S. R., AND SMOLINSKI, B. A. Toward a common component architecture for high-performance scientific computing. In *HPDC'99* (Redondo Beach, Aug. 1999), IEEE, pp. 115–124. [Cited p. 25].

[11] ASANOVIC, K., BODIK, R., CATANZARO, B. C., GEBIS, J. J., HUSBANDS, P., KEUTZER, K., PATTERSON, D. A., PLISHKER, W. L., SHALF, J., WILLIAMS, S. W., AND YELICK, K. A. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, University of California, Berkeley, EECS Department, Dec. 2006. Available online at: `http://view.eecs.berkeley.edu/` (Last accessed: 10 Feb 2007). [Cited p. 4].

[12] AUSTERN, M. H. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley Longman, Boston, 1998. [Cited p. 24].

[13] BACCI, B., DANELUTTO, M., PELAGATTI, S., AND VANNESCHI, M. SkIE: A heterogeneous environment for HPC applications. *Parallel Comput. 25*, 13 (1999), 1827–1852. [Cited p. 20].

[14] BACCI, B., DANELUTTO, M., PELAGATTI, S., VANNESCHI, M., AND ORLANDO, S. Summarising an experiment in parallel programming language design. In *HPCN Europe 1995* (Milan, May 1995), vol. 919 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 7–13. [Cited p. 18].

[15] BAILEY, D., BARSZCZ, E., BARTON, J., BROWNING, D., CARTER, R., DAGUM, L., FATOOHI, R., FREDERICKSON, P., LASINSKI, T., SCHREIBER, R., SIMON, H., VENKATAKRISHNAN, V., AND WEERATUNGA, S. The nas parallel benchmarks. *Int. J. High Perform. Comput. Appl. 5*, 3 (1991), 63–73. [Cited pp. 51 and 88].

[16] BAL, H. E., KAASHOEK, M. F., AND TANENBAUM, A. S. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng. 18*, 3 (1992), 190–205. [Cited p. 14].

[17] BANERJEE, S., HAMADA, T., CHAU, P. M., AND FELLMAN, R. D. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE Trans. Acoust. Speech Signal Process. 43*, 6 (1995), 1468–1484. [Cited p. 100].

[18] BANINO, C., BEAUMONT, O., CARTER, L., FERRANTE, J., LEGRAND, A., AND ROBERT, Y. Scheduling strategies for master-slave tasking on heterogeneous processor platforms. *IEEE Trans. Parallel Distrib. Syst. 15*, 4 (2004), 319–330. [Cited p. 65].

[19] BARNETT, M., SHULER, L., GUPTA, S., PAYNE, D. G., VAN DE GEIJN, R., AND WATTS, J. Building a high-performance collective communication library. In *SC'94* (Washington, Nov. 1994), IEEE, pp. 107–116. [Cited p. 14].

[20] BEAUMONT, O., LEGRAND, A., AND ROBERT, Y. Scheduling divisible workloads on heterogeneous platforms. *Parallel Comput. 29*, 9 (2003), 1121–1152. [Cited p. 63].

[21] BENKRID, K., CROOKES, D., AND BENKRID, A. Towards a general framework for FPGA based image processing using hardware skeletons. *Parallel Comput. 28*, 7-8 (2002), 1141–1154. [Cited p. 21].

[22] BENOIT, A., COLE, M., GILMORE, S., AND HILLSTON, J. Evaluating the performance of skeleton-based high level parallel programs. In *ICCS 2004* (Kraków, June 2004), vol. 3038 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 289–296. [Cited p. 101].

[23] BENOIT, A., COLE, M., GILMORE, S., AND HILLSTON, J. Flexible skeletal programming with eSkel. In *Euro-Par 2005* (Lisbon, 2005), no. 3648 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 761–770. [Cited p. 21].

[24] BENOIT, A., COLE, M., GILMORE, S., AND HILLSTON, J. Scheduling skeleton-based grid applications using PEPA and NWS. *Comput. J. 48*, 3 (2005), 369–378. [Cited p. 31].

[25] BENOIT, A., REHN-SONIGO, V., AND ROBERT, Y. Multi-criteria scheduling of pipeline workflows. Tech Report 6232, INRIA, ISSN: 0249-6339, June 2007. [Cited p. 101].

[26] BENOIT, A., AND ROBERT, Y. Mapping pipeline skeletons onto heterogeneous platforms. In *ICCS 2007* (Beijing, May 2007), no. 4487 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 591–598. [Cited p. 101].

[27] BERMAN, F. High-performance schedulers. In *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, Eds., first ed. Morgan Kaufmann, San Francisco, 1999, ch. 12, pp. 279–309. [Cited p. 27].

[28] BERMAN, F., AND SNYDER, L. On mapping parallel algorithms into parallel architectures. *J. Parallel Distrib. Comput. 4*, 5 (1987), 439–458. [Cited p. 100].

[29] BERMAN, F., WOLSKI, R., CASANOVA, H., CIRNE, W., DAIL, H., FAERMAN, M., FIGUEIRA, S., HAYES, J., OBERTELLI, G., SCHOPF, J., SHAO, G., SMALLEN, S., SPRING, N., SU, A., AND ZAGORODNOV, D. Adaptive computing on the grid using AppLeS. *IEEE Trans. Parallel Distrib. Syst. 14*, 4 (2003), 369–382. [Cited p. 29].

[30] BHARADWAJ, V., GHOSE, D., MANI, V., AND ROBERTAZZI, T. G. *Scheduling Divisible Loads in Parallel and Distributed Systems*. IEEE, Los Alamitos, 1996. [Cited pp. 9, 28, 49, and 65].

[31] BHARADWAJ, V., GHOSE, D., AND ROBERTAZZI, T. G. Divisible load theory: A new paradigm for load scheduling in distributed systems. *Cluster Comput. 6*, 1 (2003), 7–17. [Cited p. 65].

[32] BIRD, R. S. Lectures on constructive functional programming. Tech. Monograph PRG-89. ISBN: 0-902928-51-1, Oxford University Computing Laboratory, Programming Research Group, 1988. [Cited p. 17].

[33] BIRRELL, A. An introduction to programming with C# threads. Tech. Rep. MSR-TR-2005-68, Microsoft Research, Redmond, May 2005. [Cited p. 13].

[34] BLAZEWICZ, J., DROZDOWSKI, M., AND MARKIEWICZ, M. Divisible task scheduling - concept and verification. *Parallel Comput. 25*, 1 (1999), 87–98. [Cited p. 65].

[35] BLELLOCH, G. E. *Vector Models for Data-Parallel Computing*. MIT Press, Cambridge, 1990. [Cited p. 17].

[36] BOKHARI, S. H. On the mapping problem. *IEEE Trans. Comput. 30*, 3 (1981), 207–214. [Cited pp. 100 and 102].

[37] BOKHARI, S. H. A shortest tree algorithm for optimal assignments across space and time in a distributed processor system. *IEEE Trans. Softw. Eng. 7*, 6 (1981), 583–589. [Cited p. 100].

[38] BOKHARI, S. H. Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Trans. Comput. 37*, 1 (1988), 48–57. [Cited p. 100].

[39] BOTOROG, G. H., AND KUCHEN, H. Skil: an imperative language with algorithmic skeletons for efficient distributed programming. In *HPDC'96* (Syracuse, Aug. 1996), IEEE, pp. 243–252. [Cited p. 18].

[40] BRINCH HANSEN, P. Model programs for computational science: A programming methodology for multicomputers. *Concurr. Comput.-Pract. Exp. 5*, 5 (1993), 407–423. [Cited pp. 17 and 33].

[41] BRINCH HANSEN, P. Monitors and concurrent Pascal: a personal history. *ACM Sigplan Not. 28*, 3 (1993), 1–35. [Cited p. 12].

[42] BUYYA, R., MURSHED, M., ABRAMSON, D., AND VENUGOPAL, S. Scheduling parameter sweep applications on global grids: a deadline and budget constrained cost-time optimization algorithm. *Softw. Pract. Exper. 35*, 5 (2005), 491–512. [Cited p. 29].

[43] CAARLS, W., JONKER, P. P., AND CORPORAAL, H. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *IPDPS 2006* (Rhodes Island, Apr. 2006), IEEE. [Cited p. 21].

[44] CALIN-JAGEMAN, R. J., XIE, C., PAN, Y., VANDENBERG, A., AND KATZ, P. S. NEURONgrid: A toolkit for generating parameter-space maps using NEURON in a grid environment. In *ISBRA'07* (Atlanta, May 2007), vol. 4463 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 182–191. [Cited p. 132].

[45] CAPPELLO, F., AND ETIEMBLE, D. MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks. In *SC'00* (Dallas, Nov. 2000), IEEE, p. 12. [Cited p. 13].

[46] CASANOVA, H., KIM, M.-H., PLANK, J. S., AND DONGARRA, J. Adaptive scheduling for task farming with grid middleware. *Int. J. High Perform. Comput. Appl. 13*, 3 (1999), 231–240. [Cited p. 26].

[47] CASANOVA, H., OBERTELLI, G., BERMAN, F., AND WOLSKI, R. The AppLeS parameter sweep template: user-level middleware for the grid. In *SC'00* (Dallas, Nov. 2000), IEEE, p. 60. [Cited p. 29].

[48] CASAVANT, T., AND KUHL, J. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng. 14*, 2 (1988), 141–154. [Cited p. 28].

[49] CHAN, E., HEIMLICH, M., PURKAYASTHA, A., AND VAN DE GEIJN, R. Collective communication: theory, practice, and experience. *Concurr. Comput.-Pract. Exp. 19*, 13 (2007), 1749–1783. [Cited pp. 14 and 136].

[50] CHATTERJEE, S., AND STROSNIDER, J. K. Distributed Pipeline Scheduling: A framework for distributed, heterogeneous real-time system design. *Comput. J. 38*, 4 (1995), 271–285. [Cited p. 100].

[51] CHUN, G., DAIL, H., CASANOVA, H., AND SNAVELY, A. Benchmark probes for grid assessment. In *IPDPS'04 (HPGC Wksp)* (Santa Fe, Apr. 2004), IEEE, pp. 276–. [Cited p. 9].

[52] COLE, M. *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*. PhD thesis, University of Edinburgh, Computer Science Dpt, Edinburgh, 1988. [Cited p. 16].

[53] COLE, M. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, London, 1989. [Cited pp. 2, 16, and 64].

[54] COLE, M. Algorithmic skeletons. In *Research Directions in Parallel Functional Programming*, K. Hammond and G. Michaelson, Eds. Springer-Verlag, London, 1999, ch. 13, pp. 289–304. [Cited p. 16].

[55] COLE, M. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Comput. 30*, 3 (2004), 389–406. [Cited pp. 16, 17, 20, and 51].

[56] COLE, M., AND ZAVANELLA, A. Coordinating heterogeneous parallel systems with skeletons and activity graphs. *J. Syst. Integrat. 10*, 2 (2001), 127–143. [Cited p. 16].

[57] CRNKOVIC, I., AND LARSSOM, M. Challenges of component-based development. *J. Syst. Softw. 61*, 3 (2002), 201–212. [Cited p. 25].

[58] CUNHA, J. C., RANA, O. F., AND MEDEIROS, P. D. Future trends in distributed applications and problem-solving environments. *Futur. Gener. Comp. Syst. 21*, 6 (2005), 843–855. [Cited p. 30].

[59] CZARNECKI, K., AND EISENECKER, U. W. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Longman, Boston, 2000. [Cited p. 25].

[60] DAGUM, L., AND MENON, R. OpenMP: an industry standard API for shared-memory programming. *IEEE Computat. Sci. Eng. 5*, 1 (1998), 46–55. [Cited p. 12].

[61] DARLINGTON, J., FIELD, A. J., HARRISON, P. G., KELLY, P. H. J., SHARP, D. W. N., AND WU, Q. Parallel programming using skeleton functions. In *PARLE'93* (Munich, June 1993), vol. 694 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 146–160. [Cited p. 19].

[62] DARLINGTON, J., GUO, Y., TO, H. W., AND YANG, J. Functional skeletons for parallel coordination. In *Euro-Par 1995* (Stockholm, Aug. 1995), vol. 966 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 55–66. [Cited p. 16].

[63] DARLINGTON, J., GUO, Y., TO, H. W., AND YANG, J. Parallel skeletons for structured composition. In *PPOPP'95* (Santa Barbara, July 1995), ACM, pp. 19–28. [Cited p. 18].

[64] DARLINGTON, J., AND TO, H. W. Building parallel applications without programming. In *Abstract Machine Models for Highly Parallel Computers*, J. R. Davy and P. M. Dew, Eds. Oxford University Press, Oxford, 1995, ch. 8, pp. 140–154. [Cited p. 18].

[65] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *OSDI'04* (San Francisco, Dec. 2004), USENIX, pp. 137–150. [Cited p. 21].

[66] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (2008), 107–113. [Cited p. 21].

[67] DIJKSTRA, E. W. The structure of the THE-multiprogramming system. *Commun. ACM 11*, 5 (1968), 341–346. [Cited p. 12].

[68] DONGARRA, J., FOSTER, I., FOX, G., GROPP, W., KENNEDY, K., TORCZON, L., AND WHITE, A., Eds. *The Sourcebook of Parallel Computing*. Morgan Kaufmann, San Francisco, 2002. [Cited pp. 144 and 151].

[69] DONGARRA, J., FOSTER, I., AND KENNEDY, K. Reusable software and algorithms. In Dongarra et al. [68], ch. 17, pp. 483–490. [Cited pp. 16 and 26].

[70] DONGARRA, J., GANNON, D., FOX, G., AND KENNEDY, K. The impact of multicore on computational science software. *CTWatch Quarterly 3*, 1 (2007), 3–10. [Cited pp. 6 and 33].

[71] DONGARRA, J. J., LUSZCZEK, P., AND PETITET, A. The LINPACK benchmark: Past, present and future. *Concurr. Comput.-Pract. Exp. 15*, 9 (2003), 803–820. [Cited pp. 51 and 88].

[72] DONGARRA, J. J., OTTO, S. W., SNIR, M., AND WALKER, D. A message passing standard for MPP and workstations. *Commun. ACM 39*, 7 (1996), 84–90. [Cited p. 14].

[73] DORTA, A. J., GONZÁLEZ, J. A., RODRIGUEZ, C., AND DE SANDE, F. LLC: A parallel skeletal language. *Parallel Process. Lett. 13*, 3 (2003), 437–448. [Cited p. 18].

[74] DROZDOWSKI, M., AND LAWENDA, M. Multi-installment divisible load processing in heterogeneous distributed systems. *Concurr. Comput.-Pract. Exp. 19*, 17 (2007), 2237–2253. [Cited p. 65].

[75] DUBEY, P. K., AND FLYNN, M. J. Optimal pipelining. *J. Parallel Distrib. Comput. 8*, 1 (1990), 10–19. [Cited p. 100].

[76] DÜNNWEBER, J., AND GORLATCH, S. HOC-SA: A grid service architecture for higher-order components. In *SCC'04* (Shanghai, Sept. 2004), IEEE, pp. 288–294. [Cited p. 25].

[77] EIGENMANN, R., Ed. *Performance Evaluation and Benchmarking with Realistic Applications*. MIT Press, Cambridge, 2001. [Cited pp. 88 and 139].

[78] EL-REWINI, H., LEWIS, T. G., AND ALI, H. H. *Task Scheduling in Parallel and Distributed Systems*. Innovative Technology Series. Prentice-Hall, New Jersey, 1994. [Cited pp. 28 and 51].

[79] FALCOU, J., SEROT, J., CHATEAU, T., AND LAPRESTE, J. Quaff: efficient C++ design for parallel skeletons. *Parallel Comput. 32*, 7-8 (2006), 604–615. [Cited p. 25].

[80] FEITELSON, D. G. Parallel workloads archive. website, Hebrew University, `http://www.cs.huji.ac.il/labs/parallel/workload/`, 2005. [Cited p. 58].

[81] FINKEL, R., AND MANBER, U. DIB–a distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst. 9*, 2 (1987), 235–256. [Cited p. 17].

[82] FOSTER, I., AND KESSELMAN, C. Globus: a metacomputing infrastructure toolkit. *Int. J. High Perform. Comput. Appl. 11*, 2 (1997), 115–128. [Cited p. 25].

[83] FOSTER, I., AND KESSELMAN, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure*, second ed. Morgan Kaufmann, San Francisco, 2003. [Cited p. 26].

[84] FOSTER, I., KESSELMAN, C., AND TUECKE, S. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl. 15*, 3 (2001), 200–222. [Cited pp. 3 and 26].

[85] FRACHTENBERG, E., AND FEITELSON, D. G. Pitfalls in parallel job scheduling evaluation. In *JSSPP'05* (Cambridge, June 2005), vol. 3834 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 257–282. [Cited p. 58].

[86] FREEMAN, E., HUPFER, S., AND ARNOLD, K. *Javaspaces: Principles, Patterns and Practices*. Jini Technology Series. Prentice-Hall, Reading, 1999. [Cited p. 15].

[87] GABRIEL, E., RESCH, M., BEISEL, T., AND KELLER, R. Distributed computing in a heterogeneous computing environment. In *PVM/MPI 1998* (Liverpool, 1998), vol. 1497 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 180–187. [Cited p. 40].

[88] GALASSI, M., DAVIES, J., THEILER, J., JUNGMAN, B. G. G., BOOTH, M., AND ROSSI, F. Least-squares fitting. In *GNU Scientific Library Reference Manual*. Network Theory, Bristol, Sept. 2005, ch. 36, pp. 361–369. [Cited pp. 82 and 113].

[89] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP'93* (Kaiserslautern, July 1993), vol. 707 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 406–431. [Cited p. 23].

[90] GANNON, D., BRAMLEY, R., FOX, G., SMALLEN, S., ROSSI, A., ANANTHAKRISHNAN, R., BERTRAND, F., CHIU, K., FARRELLEE, M., GOVINDARAJU, M., KRISHNAN, S., RAMAKRISHNAN, L., SIMMHAN, Y., SLOMINSKI, A., MA, Y., OLARIU, C., AND REY-CENVAZ, N. Programming the grid: Distributed software components, P2P and grid web services for scientific applications. *Cluster Comput. 5*, 3 (2002), 325–336. [Cited p. 25].

[91] GEIST, A., BEGUELIN, A., DONGARRA, J., JIANG, W., MANCHEK, R., AND SUNDERAM, V. S. *PVM: Parallel Virtual Machine*. MIT Press, Cambridge, 1994. [Cited p. 13].

[92] GELERNTER, D. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (1985), 80–112. [Cited p. 14].

[93] GERLACH, J., AND KNEIS, J. Generic programming for scientific computing in C++, Java, and C#. In *APPT'03* (Xiamen, Sept. 2003), X. Zhou, S. Jahnichen, M. Xu, and J. Cao, Eds., vol. 2834 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 301–310. [Cited p. 24].

[94] GOMES, M. C., RANA, O. F., AND CUNHA, J. C. Extending grid-based workflow tools with patterns/operators. *Int. J. High Perform. Comput. Appl.* (2007). In press. [Cited p. 23].

[95] GONZÁLEZ, D., ALMEIDA, F., MORENO, L. M., AND RODRÍGUEZ, C. Towards the automatic optimal mapping of pipeline algorithms. *Parallel Comput. 29*, 2 (2003), 241–254. [Cited p. 101].

[96] GONZÁLEZ-VÉLEZ, H. An adaptive skeletal task farm for grids. In *Euro-Par 2005* (Lisbon, Aug. 2005), no. 3648 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 401–410. [Cited pp. iii, 10, 14, and 132].

[97] GONZÁLEZ-VÉLEZ, H. On the abstraction of message-passing communications using algorithmic skeletons: A case study. In *ISSADS'05* (Guadalajara, Jan. 2005), no. 3563 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 43–50. [Cited pp. iii and 9].

[98] GONZÁLEZ-VÉLEZ, H. Structured parallelism for the grid. In *Science and Supercomputing in Europe*, P. Alberigo, G. Erbacci, and F. Garofalo, Eds., HPC-Europa. CINECA, Bologna, 2005, pp. 201–203. [Cited pp. iii, 10, 14, 55, and 132].

[99] GONZÁLEZ-VÉLEZ, H. Self-adaptive skeletal task farm for computational grids. *Parallel Comput. 32*, 7-8 (2006), 479–490. [Cited pp. iii, 10, and 132].

[100] GONZÁLEZ-VÉLEZ, H., AND COLE, M. Towards fully adaptive pipeline parallelism for heterogeneous distributed environments. In *ISPA'06* (Sorrento, Dec. 2006), no. 4330 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 916–926. [Cited pp. iii, 10, and 132].

[101] GONZÁLEZ-VÉLEZ, H., AND COLE, M. Adaptive structured parallelism for computational grids. In *PPoPP'07* (San Jose, Mar. 2007), ACM, pp. 140–141. [Cited pp. iii, 9, and 133].

[102] GONZÁLEZ-VÉLEZ, H., AND COLE, M. An adaptive parallel pipeline pattern for grids. In *IPDPS'08* (Miami, Apr. 2008), IEEE, pp. 1–11. To appear. [Cited pp. iii, 10, and 133].

[103] GONZÁLEZ-VÉLEZ, H., DE LUCA, A., AND GONZÁLEZ-VÉLEZ, V. A comparative study of intrinsic parallel programming methodologies. In *ICEEE'04* (Acapulco, June 2004), IEEE, pp. 200–205. [Cited pp. iii, 9, and 131].

[104] GONZÁLEZ-VÉLEZ, V., AND GONZÁLEZ-VÉLEZ, H. A grid-based stochastic simulation of unitary and membrane Ca^{2+} currents in spherical cells. In *CBMS'05* (Dublin, June 2005), IEEE, pp. 171–176. [Cited pp. iii, 10, 52, 83, and 133].

[105] GONZÁLEZ-VÉLEZ, V., AND GONZÁLEZ-VÉLEZ, H. Parallel stochastic simulation of macroscopic calcium currents. *J. Bioinform. Comput. Biol. 5*, 3 (2007), 755–772. [Cited pp. iii, 10, 84, and 133].

[106] GORLATCH, S. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst. 26*, 1 (2004), 47–56. [Cited p. 13].

[107] GOSWAMI, D., SINGH, A., AND PREISS, B. R. From design patterns to parallel architectural skeletons. *J. Parallel Distrib. Comput. 62*, 4 (2002), 669–695. [Cited p. 25].

[108] GRELCK, C. Shared memory multiprocessor support for functional array processing in SAC. *J. Funct. Program. 15*, 3 (2005), 431–475. [Cited p. 18].

[109] GUIRADO, F., RIPOLL, A., ROIG, C., HERNANDEZ, A., AND LUQUE, E. Exploiting throughput for pipeline execution in streaming image processing applications. In *Euro-Par 2006* (Dresden, Aug. 2006), vol. 4128 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 1095–1105. [Cited p. 101].

[110] GUIRADO, F., RIPOLL, A., ROIG, C., AND LUQUE, E. Exploitation of parallelism for applications with an input data stream: Optimal resource-throughput tradeoffs. In *PDP 2005* (Lugano, Feb. 2005), IEEE, pp. 170–178. [Cited p. 101].

[111] GUNTHER, N. J. *Analyzing Computer System Performance with Perl: PDQ*. Springer-Verlag, Berlin, 2004. [Cited p. 56].

[112] HAGERUP, T. Allocating independent tasks to parallel processors: An experimental study. *J. Parallel Distrib. Comput. 47*, 2 (Dec. 1997), 185–197. [Cited p. 63].

[113] HASSELBRING, W. Programming languages and systems for prototyping concurrent applications. *ACM Comput. Surv. 32*, 1 (2000), 43–79. [Cited p. 17].

[114] HERON DE CARVALHO JUNIOR, F., AND DUEIRE LINS, R. Topological skeletons in Haskell#. In *IPDPS'03* (Nice, Apr. 2003), IEEE, p. 53. CDROM. [Cited p. 19].

[115] HERRMANN, C. A., AND LENGAUER, C. HDC: A higher-order language for divide-and-conquer. *Parallel Process. Lett. 10*, 2–3 (2000), 239–250. [Cited p. 19].

[116] HEY, A. J. G. Practical parallel processing with transputers. In *HyperCube'88* (Pasadena, 1988), ACM, pp. 115–121. [Cited p. 62].

[117] HEY, A. J. G. Experiments in MIMD parallelism. *Futur. Gener. Comp. Syst. 6*, 3 (1990), 185–196. [Cited pp. 17, 62, and 64].

[118] HEYMANN, E., SENAR, M. A., LUQUE, E., AND LIVNY, M. Adaptive scheduling for master-worker applications on the computational grid. In *Grid 2000* (Bangalore, 2000), vol. 1971 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 214–227. [Cited p. 29].

[119] HOARE, C. A. R. Towards a theory of parallel programming. In *Operating Systems Techniques*, C. A. R. Hoare and R. H. Perrott, Eds., no. 9 in A.P.I.C. Studies in Data Processing. Academic Press, London, 1972, pp. 61–71. [Cited p. 12].

[120] HOARE, C. A. R. Monitors: an operating system structuring concept. *Commun. ACM 17*, 10 (1974), 549–557. [Cited p. 12].

[121] HORVATH, Z., ZSOK, V., SERRARENS, P., AND PLASMEIJER, R. Parallel elementwise processable functions in Concurrent Clean. *Math. Comput. Model. 38*, 7–9 (2003), 865–875. [Cited p. 19].

[122] IEEE AND THE OPEN GROUP. Standard for information technology - portable operating system interface (POSIX). Issue 6 IEEE Std 1003.1 and ISO/ISO/IEC 9945, The Open Group Technical Standard Base Specifications, New York/Berkshire, Apr. 2004. [Cited p. 13].

[123] JOHNSON, E., AND GANNON, D. HPC++: experiments with the parallel standard template library. In *ICS '97* (Vienna, 1997), ACM, pp. 124–131. [Cited p. 24].

[124] KELLY, P. H. J. *Functional Programming for Loosely-coupled Multiprocessors*. Research Monographs in Parallel and Distributed Computing. Pitman/MIT Press, London, 1989. [Cited p. 17].

[125] KENNEDY, K., MAZINA, M., MELLOR-CRUMMEY, J., COOPER, K., TORCZON, L., BERMAN, F., CHIEN, A., DAIL, H., SIEVERT, O., ANGULO, D., FOSTER, I., GANNON, D., JOHNSSON, L., KESSELMAN, C., AYDT, R., REED, D., DONGARRA, J., VADHIYAR, S., AND WOLSKI, R. Toward a framework for preparing and executing adaptive grid programs. In *IPDPS 2002* (Fort Lauderdale, 2002), IEEE, pp. 171–175. [Cited p. 32].

[126] KING, C.-T., CHOU, W.-H., AND NI, L. M. Pipelined data parallel algorithms–I: Concept and modeling. *IEEE Trans. Parallel Distrib. Syst. 1*, 4 (1990), 470–485. [Cited p. 100].

[127] KING, C.-T., CHOU, W.-H., AND NI, L. M. Pipelined data parallel algorithms–II: Design. *IEEE Trans. Parallel Distrib. Syst. 1*, 4 (1990), 486–499. [Cited p. 100].

[128] KITANO, H. Systems biology: A brief overview. *Science 295*, 5560 (2002), 1662–1664. [Cited p. 135].

[129] KRAWEZIK, G. Performance comparison of MPI and three OpenMP programming styles on shared memory multiprocessors. In *SPAA'03* (San Diego, June 2003), ACM, pp. 118–127. [Cited p. 13].

[130] KUANG, H., BIC, L. F., AND DILLENCOURT, M. B. PODC: Paradigm-oriented distributed computing. *J. Parallel Distrib. Comput. 65*, 4 (2006), 506–518. [Cited pp. 21 and 25].

[131] KUCHEN, H., AND STRIEGNITZ, J. Features from functional programming for a C++ skeleton library. *Concurr. Comput.-Pract. Exp. 17*, 7-8 (2005), 739–756. [Cited p. 20].

[132] KUNG, H. Computational models for parallel computers. *Philos. Trans. R. Soc. A-Math. Phys. Eng. Sci. 326*, 1591 (1988), 357–371. [Cited pp. 17 and 64].

[133] LAFORENZA, D. Grid programming: some indications where we are headed. *Parallel Comput. 28*, 12 (2002), 1733–1752. [Cited pp. 4 and 26].

[134] LAMPORT, L. The mutual exclusion problem: part I-a theory of interprocess communication. *J. ACM 33*, 2 (1986), 313–326. [Cited p. 12].

[135] LAURE, E. OpusJava: A Java framework for distributed high performance computing. *Futur. Gener. Comp. Syst. 18* (2001), 235–251. [Cited p. 15].

[136] LEOPOLD, C. High-level programming models. In *Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches*. John Wiley & Sons, New York, 2001, ch. 11. [Cited pp. 17 and 26].

[137] LI, M., AND BAKER, M. Grid scheduling and resource management. In *The Grid Core Technologies*. John Wiley & Sons, Chichester, 2005, ch. 6, pp. 243–300. [Cited p. 50].

[138] LOIDL, H.-W., RUBIO, F., SCAIFE, N., HAMMOND, K., HORIGUCHI, S., KLUSIK, U., LOOGEN, R., MICHAELSON, G., PENA, R., PRIEBE, S., REBÓN PORTILLO, Á. J., AND TRINDER, P. W. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput. 16*, 3 (2003), 203–251. [Cited p. 19].

[139] LONGBOTTOM, R. C/C++ Whetstone benchmark single or double precision. `ftp.nosc.mil/pub/aburto`, 1997. Official version approved by H. J. Curnow on 21/Nov/97. [Cited pp. 52 and 113].

[140] LOOGEN, R., ORTEGA-MALLÉN, Y., AND PEÑA MARÍ, R. Parallel functional programming in Eden. *J. Funct. Program. 15*, 3 (2005), 431–475. [Cited p. 19].

[141] LUECKE, G. R., AND LIN, W.-H. Scalability and performance of OpenMP and MPI on a 128-processor SGI Origin 2000. *Concurr. Comput.-Pract. Exp. 13*, 10 (2001), 905–928. [Cited p. 13].

[142] LUSK, E. L., AND OVERBEEK, R. A. A minimalist approach to portable, parallel programming. In *The Characteristics of Parallel Algorithms*, L. H. Jamieson, D. B. Gannon, and R. J. Douglass, Eds., Scientific Computation Series. MIT Press, Cambridge, 1987, ch. 14, pp. 351–362. [Cited p. 17].

[143] MACDONALD, S., ANVIK, J., BROMLING, S., SCHAEFFER, J., SZAFRON, D., AND TAN, K. From patterns to frameworks to parallel programs. *Parallel Comput. 28*, 12 (2002), 1663–1683. [Cited p. 23].

[144] MAJUMDAR, S., EAGER, D. L., AND BUNT, R. B. Scheduling in multiprogrammed parallel systems. *SIGMETRICS Perform. Eval. Rev. 16*, 1 (1988), 104–113. [Cited p. 28].

[145] MARTENS, H., AND NAES, T. *Multivariate Calibration*. John Wiley & Sons, Chichester, 1989. [Cited p. 69].

[146] MATSUZAKI, K., IWASAKI, H., EMOTO, K., AND HU, Z. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale'06* (Hong Kong, May 2006), vol. 152 of *ACM Int Conf Proc Series*, ACM, p. 13. [Cited p. 20].

[147] MATTSON, T. G., SANDERS, B. A., AND MASSINGILL, B. L. *Patterns for Parallel Programming*. Software Patterns Series. Addison-Wesley Longman, Boston, 2004. [Cited p. 23].

[148] MCKENNEY, P. E. Selecting locking primitives for parallel programming. *Commun. ACM 39*, 10 (1996), 75–82. [Cited p. 23].

[149] MEHROTRA, P., AND HAINES, M. An overview of the Opus language and runtime system. In *LCPC'94* (Ithaca, 1995), vol. 892 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 346–360. [Cited p. 14].

[150] MEHTA, P., AMARAL, J. N., AND SZAFRON, D. Is MPI suitable for a generative design-pattern system? *Parallel Comput. 32*, 7-8 (2006), 616–626. [Cited p. 23].

[151] MICHAELSON, G., SCAIFE, N., BRISTOW, P., AND KING, P. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms Appl. 16* (2001), 18–206. [Cited p. 19].

[152] MPI FORUM. The official message passing interface standard. web site, 2007. `http://www.mpi-forum.org` (Last accessed: 10 Feb 2007). [Cited p. 13].

[153] NICOLESCU, C., AND JONKER, P. A data and task parallel image processing environment. *Parallel Comput. 28*, 7-8 (2002), 945–965. [Cited p. 21].

[154] NOLTE, J., SATO, M., AND ISHIKAWA, Y. Exploiting cluster networks for distributed object groups and collective operations. *Futur. Gener. Comp. Syst. 18*, 4 (2002), 461–476. [Cited p. 24].

[155] NORMAN, M. G., AND THANISCH, P. Models of machines and computation for mapping in multicomputers. *ACM Comput. Surv. 25*, 3 (1993), 263–302. [Cited p. 100].

[156] OAKS, S., AND WONG, H. *Java Threads*, third ed. O'Reilly, Sebastopol, 2004. [Cited p. 13].

[157] PALTRIDGE, B. Thesis and dissertation writing: an examination of published advice and actual practice. *Engl. Specif. Purp. 21*, 2 (2002), 125–143. [Cited p. 7].

[158] PELAGATTI, S. *Structured Development of Parallel Programs.* Taylor & Francis, London, 1998. [Cited pp. 15 and 18].

[159] PELAGATTI, S. Task and data parallelism in P3L. In Rabhi and Gorlatch [163], pp. 155–186. [Cited p. 16].

[160] PJESIVAC-GRBOVIC, J., ANGSKUN, T., BOSILCA, G., FAGG, G. E., GABRIEL, E., AND DONGARRA, J. Performance analysis of MPI collective operations. *Cluster Comput. 10,* 2 (2007), 127–143. [Cited pp. 14, 55, and 136].

[161] POLYCHRONOPOULOS, C. D., AND KUCK, D. J. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput. 36,* 12 (1987), 1425–1439. [Cited p. 94].

[162] QUINLAN, D. J., SCHORDAN, M., MILLER, B., AND KOWARSCHIK, M. Parallel object-oriented framework optimization. *Concurr. Comput.-Pract. Exp. 16,* 2–3 (2004), 293–302. [Cited p. 24].

[163] RABHI, F. A., AND GORLATCH, S., Eds. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer-Verlag, London, 2003. [Cited pp. 17 and 151].

[164] RAICU, I. Harnessing grid resources with data-centric task farms. Dissertation proposal, Department of Computer Science, University of Chicago, 2007. Available online at: `http://people.cs.uchicago.edu/~iraicu/publications/2007_UChicago_CandidacyExam-proposal.` (last accessed: 20/May/2008). [Cited p. 132].

[165] RAMAMOORTHY, C. V., AND LI, H. F. Pipeline architecture. *ACM Comput. Surv. 9,* 1 (1977), 61–102. [Cited p. 99].

[166] RANGER, C., RAGHURAMAN, R., PENMETSA, A., BRADSKI, G., AND KOZYRAKIS, C. Evaluating MapReduce for multi-core and multiprocessor systems. In *HPCA-13* (Phoenix, Feb. 2007), IEEE, pp. 13–24. [Cited p. 21].

[167] REYNDERS, J., GANNON, D., AND CHANDY, K. M. Parallel object-oriented libraries. In Dongarra et al. [68], ch. 13, pp. 383–407. [Cited p. 24].

[168] SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. *Pattern-Oriented Software Architecture,* vol. 2: Patterns for Concurrent and Networked Objects of *Software Design Patterns.* John Wiley & Sons, Chichester, 2000. [Cited p. 23].

[169] SCHOPF, J. M. Ten action when grid scheduling. In *Grid Resource Management : State of the Art and Future Trends,* J. Nabrzyski, J. M. Schopf, and J. Weglarz, Eds., Operations Research and Management Science. Kluwer Academic, Boston, 2003, ch. 2, pp. 15–23. [Cited p. 32].

[170] SEROT, J., AND GINHAC, D. Skeletons for parallel image processing: an overview of the SKiPPER project. *Parallel Comput. 28,* 12 (2002), 1785–1808. [Cited p. 21].

[171] SHAO, G., BERMAN, F., AND WOLSKI, R. Master/slave computing on the grid. In *HCW'00* (Cancun, May 2000), IEEE, pp. 3–16. [Cited p. 26].

[172] SIEK, J. G., AND LUMSDAINE, A. The Matrix Template Library: A generic programming approach to high performance numerical linear algebra. In *ISCOPE'98* (Santa Fe, Dec. 1998), vol. 1505 of *Lect. Notes Comput. Sc.*, Springer-Verlag, pp. 59–70. [Cited p. 24].

[173] SIEVERT, O., AND CASANOVA, H. A simple MPI process swapping architecture for iterative applications. *Int. J. High Perform. Comput. Appl. 18*, 3 (2004), 341–352. [Cited p. 112].

[174] SINCLAIR, J. B. Efficient computation of optimal assignments for distributed tasks. *J. Parallel Distrib. Comput. 4*, 4 (1987), 342–362. [Cited p. 100].

[175] SKILLICORN, D. B. Architecture-independent parallel computation. *Computer 23*, 12 (1990), 38–50. [Cited p. 17].

[176] SKILLICORN, D. B., AND TALIA, D. Models and languages for parallel computation. *ACM Comput. Surv. 30*, 2 (1998), 123–169. [Cited pp. 17 and 26].

[177] SMARR, L., AND CATLETT, C. E. Metacomputing. *Commun. ACM 35*, 6 (1992), 44–52. [Cited p. 3].

[178] SMITH, D. R. KIDS: A semiautomatic program development system. *IEEE Trans. Softw. Eng. 16*, 9 (1990), 1024–1043. [Cited p. 17].

[179] SNIR, M., AND GROPP, W. *MPI: The Complete Reference*, second ed., vol. 1–2. MIT Press, Cambridge, 1998. [Cited p. 13].

[180] SQUYRES, J. M., AND LUMSDAINE, A. A component architecture for LAM/MPI. In *EuroPVM/MPI 2003* (Venice, Sept. 2003), no. 2840 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 379–387. [Cited p. 54].

[181] SUBHLOK, J., AND VONDRAN, G. Optimal mapping of sequences of data parallel tasks. In *PPOPP'95* (Santa Barbara, July 1995), pp. 134–143. [Cited p. 100].

[182] THAIN, D., TANNENBAUM, T., AND LIVNY, M. Distributed computing in practice: the Condor experience. *Concurr. Comput.-Pract. Exp. 17*, 2-4 (2005), 323–356. [Cited p. 29].

[183] THE APACHE SOFTWARE FOUNDATION. Hadoop Map-Reduce tutorial. Manual version 0.15, Hadoop Project, `http://hadoop.apache.org/`, Jan. 2008. [Cited p. 21].

[184] THEYS, M. D., ALI, S., SIEGEL, H. J., CHANDY, K. M., HWANG, K., KENNEDY, K., SHA, L., SHIN, K. G., SNIR, M., SNYDER, L., AND STERLING, T. L. What are the top ten most influential parallel and distributed processing concepts of the past millenium? *J. Parallel Distrib. Comput. 61*, 12 (2001), 1827–1841. [Cited p. 96].

[185] TSAFRIR, D., AND FEITELSON, D. G. Instability in parallel job scheduling simulation: the role of workload flurries. In *IPDPS 2006* (Rhodes Island, Apr. 2006), IEEE. [Cited p. 128].

[186] U. MANNHEIM, U. TENNESSEE AND NERSC. TOP500 supercomputer sites. web site, nov 2007. `http://www.top500.org/` (Last accessed: 10 Dec 2007). [Cited p. 26].

[187] VADHIYAR, S. S., AND DONGARRA, J. SRS: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Process. Lett. 13*, 2 (2003), 291–312. [Cited p. 112].

[188] VADHIYAR, S. S., AND DONGARRA, J. Self adaptivity in grid computing. *Concurr. Comput.-Pract. Exp. 17*, 2-4 (2005), 235–257. [Cited pp. 26 and 32].

[189] VAN DER AALST, W. M. P., TER HOFSTEDE, A. H. M., KIEPUSZEWSKI, B., AND BARROS, A. P. Workflow patterns. *Distrib. Parallel Databases 14*, 1 (2003), 5–51. [Cited p. 29].

[190] VAN DORST, W. The quintessential Linux benchmark: All about the "BogoMips" number displayed when Linux boots. *Linux J. 1996*, 21es (1996), 4. [Cited pp. 52 and 88].

[191] VANNESCHI, M. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Comput. 28*, 12 (2002), 1709–1732. [Cited pp. 20 and 101].

[192] VAZHKUDAI, S., AND SCHOPF, J. M. Using regression techniques to predict large data transfers. *Int. J. High Perform. Comput. Appl. 17*, 3 (2003), 249–268. [Cited p. 93].

[193] VILLACIS, J., AND GANNON, D. A web interface to parallel program source code archetypes. In *SC'95* (San Diego, 1995), IEEE, p. 18. [Cited p. 23].

[194] WHALEY, R. C., PETITET, A., AND DONGARRA, J. Automated empirical optimizations of software and the ATLAS project. *Parallel Comput. 27*, 1-2 (2001), 3–35. [Cited p. 24].

[195] WILKINSON, B., AND ALLEN, M. *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*. Prentice-Hall, Upper Saddle River, 1999, ch. 3-Embarrassingly Parallel Computations and 5-Pipelined Computations, pp. 82–106 and 139–161. [Cited pp. 9 and 50].

[196] WOLSKI, R., SPRING, N., AND HAYES, J. The Network Weather Service: A distributed resource performance forecasting service for metacomputing. *Futur. Gener. Comp. Syst. 15*, 5–6 (1999), 757–768. [Cited p. 82].

[197] YAIKHOM, G., COLE, M., AND GILMORE, S. Combining measurement and stochastic modelling to enhance scheduling decisions for a parallel mean value analysis algorithm. In *ICCS 2006* (Reading, May 2006), no. 3992 in Lect. Notes Comput. Sc., Springer-Verlag, pp. 929–936. [Cited pp. 101 and 132].

[198] YANG, Y., CASANOVA, H., DROZDOWSKI, M., LAWENDA, M., AND LEGRAND, A. On the complexity of multi-round divisible load scheduling. Tech Report 6096, INRIA, ISSN: 0249-6339, Jan. 2007. [Cited p. 65].

[199] YANG, Y., VAN DER RAADT, K., AND CASANOVA, H. Multiround algorithms for scheduling divisible loads. *IEEE Trans. Parallel Distrib. Syst. 16*, 11 (2005), 1092–1102. [Cited p. 65].

[200] YU, J., AND BUYYA, R. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec. 34*, 3 (2005), 44–49. [Cited p. 29].

Index