UNIVERSITÉ DE NICE - SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC

SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

THÈSE

pour obtenir le titre de

Docteur en Sciences

de l'Université de Nice - Sophia Antipolis

Mention Informatique

présentée et soutenu par

Mario LEYTON

ADVANCED FEATURES FOR ALGORITHMIC SKELETON PROGRAMMING

Thèse dirigée par Denis CAROMEL,

au sein de l'équipe OASIS,
*équipe commune de l'*INRIA Sophia Antipolis*, du* CNRS *et du laboratoire* I3S

soutenue le 23 octobre 2008

Jury:

Rapporteurs	Murray COLE	University of Edinburgh
	Marco DANELUTTO	Università di Pisa
	Frédéric LOULERGUE	Université d'Orléans
Examinateurs	Gilles BERNOT	Université de Nice - Sophia Antipolis
	Xavier LEFUMEUX	HP Centre de Compétences
Directeur de thèse	Denis CAROMEL	Université de Nice - Sophia Antipolis

To my family,
– Mario

Acknowledgements

This thesis was co-financed by: *INRIA Sophia Antipolis (OASIS Team)*, and
The Republic of Chile via *CONICYT*.

I would like to begin by thanking Murray Cole, Marco Danelutto, and Frédéric Loulergue for reviewing this thesis and providing valuable suggestions. Also, I would like to thank Gilles Bermot, and Xavier Lefumeux for being members of the jury.

My advisor Denis Caromel deserves special thanks. For his trust, which allowed me to pursue my research interest with freedom; and for his pragmatism, which pulled me back to earth for ProActive version releases.

The permanent staff of the OASIS Team have my profound gratitude. Eric Madelaine for his invaluable help to overcome any administrative obstacle. Fabrice Huet for giving me root access to my machine, and then taking it away. Françoise Baude for kindly guiding me through my first steps in the research world. Ludovic Henrio, whom I am specially thankful, for being always open for discussion. Where it not for Ludovic, this thesis would have never ventured into the unfriendly realms of formalisms.

To my former office mates, thanks. Romain for showing me the ropes, Vincent for the *citronade*, Clement for teaching me "correct" french, and Viet for his good humor. Of course I would also like to thank former and present members of the OASIS Team, in no particular order and at the risk of missing many: Christian, Alexandre, Igor, Felipe, Laurent, Paul, Nikos, Guilherme, Johann, Ian, Cedric, Guillaume, Brian, Bastien, Florin, Patricia, Emil, Vladimir, Claire, Christiane, Arnaud, Virginie, Muhammad, Imen, Fabien, Franca, Vasile, Elaine, Yu, Abhijeet, Regis, Germain, Elton, and innumerable interns.

I am also thankful to the chilean mafia friends. To the founder Tomas for showing me around and looking after me at the beginning of my stay in France. Tamara for her inexhaustible enthusiasm. Matthieu, my former french flatmate, for his enlightenment on cheese. Javier for his straightforwardness. Angela for her tennis playing. Pato for his races planning. Marcelo for his timely pictures. Cristian for his Nintendo Wii. Maria Jose for her books suggestions. Diego for his infiltration. Alonso for his infinite questions. Carlos for his entertaining coffee-break stories. Marcela for her effective system shortcuts. Antonio, for his long time friendship.

And specially Beatriz for having the courage to come with me to France.

Contents

Acknowledgements **v**

Contents **x**

List of Figures **xi**

List of Tables **xiii**

List of Listings **xv**

I Thesis **1**

1 Introduction **3**
1.1 Problematic 3
1.2 Objectives and Contributions 4
1.3 Overview 5
1.4 Reading Itineraries 6

2 State of the Art **7**
2.1 Distributed Programming Models 7
2.1.1 Remote Procedure Calls 7
2.1.2 Message passing 8
2.1.3 Bulk Synchronous Parallelism (BSP) 8
2.1.4 Distributed Objects 9
2.1.5 Distributed Components 9
2.1.6 Workflows and Service Oriented Architectures 10
2.2 Algorithmic Skeletons Frameworks 10
2.2.1 Alt and HOC-SA 11
2.2.2 ASSIST 12
2.2.3 CO_2P_3S 12
2.2.4 Eden 13
2.2.5 eSkel 13
2.2.6 HDC 14
2.2.7 JaSkel 14
2.2.8 Lithium and Muskel 15
2.2.9 Mallba 15

2.2.10 P^{3}L, SkIE, SKElib 16
2.2.11 PAS and EPAS 17
2.2.12 SBASCO 17
2.2.13 SCL 18
2.2.14 SKiPPER, QUAFF 18
2.2.15 SkeTo 18
2.2.16 Skil and Muesli 19
2.2.17 TBB 19
2.2.18 Others 20
2.3 Discussion 20
2.3.1 Programming Models 20
2.3.2 Algorithmic Skeletons 21
2.3.2.1 Cole's Manifesto Principles 25
2.3.2.2 Characteristics vs Manifesto Principles 25
2.4 Context: The ProActive Library 27
2.4.1 Active Object Model 27
2.4.2 The ProActive library: principles, architecture and usages . 28
2.4.2.1 Implementation language 28
2.4.2.2 Implementation techniques 28
2.4.2.3 Semantics of communications 29
2.4.2.4 Library Features 30
2.4.2.5 Large scale experiments and usages 31
2.5 Conclusion 33

3 Algorithmic Skeletons in Calcium 35
3.1 An Algorithmic Skeleton Language 37
3.1.1 Muscle (Sequential) Blocks 37
3.1.2 The Skeleton Patterns 38
3.1.3 Skeleton Instructions 39
3.1.3.1 The Instruction Language 40
3.1.3.2 From Skeletons to Instructions 41
3.1.4 Instructions Reduction Rules 42
3.1.4.1 Non-atomic Seq Instruction 42
3.1.4.2 Reduction Rules 43
3.1.4.3 Task and Data Parallelism Rules 44
3.1.5 Summary 44
3.2 A Java Implementation 45
3.2.1 The Skeleton Library 45
3.2.2 Muscles 45
3.2.3 Instruction Stacks 47
3.2.4 Tasks 48
3.3 Execution Environments 50
3.3.1 General Principle 53
3.3.2 Multithreaded Environment 53
3.3.3 ProAcitve Environment 54
3.3.4 ProActive Scheduler Environment 55

3.4 Example: Find Primes . . . 56
3.4.1 Data Exchange Types . . . 56
3.4.2 Muscles . . . 57
3.4.3 Skeleton Definition and Execution . . . 59
3.5 Conclusion . . . 60

4 Performance Tuning **61**
4.1 Muscle Tuning of Algorithmic Skeletons . . . 62
4.1.1 Performance Diagnosis . . . 62
4.1.2 Performance Metrics . . . 62
4.1.3 Muscle Workout . . . 63
4.1.4 Code Blaming . . . 63
4.2 NQueens Test Case . . . 64
4.3 Conclusions and Future Work . . . 66

5 Type Safe Algorithmic Skeletons **69**
5.1 Related Work . . . 71
5.2 A typed algorithmic skeleton language . . . 72
5.2.1 Skeleton Language Grammar . . . 72
5.2.2 Reduction Semantics . . . 72
5.2.3 Type System . . . 73
5.2.4 Typing Property: Subject Reduction . . . 75
5.2.5 Sub-typing . . . 78
5.3 Type safe skeletons in Java . . . 79
5.4 Conclusions . . . 80

6 File Access Model **83**
6.1 File Transfer with Active Objects . . . 84
6.1.1 Asynchronous File Transfer with Futures . . . 84
6.1.2 Push & Pull Benchmarks . . . 86
6.1.3 Benchmarks Discussion . . . 87
6.2 File Transfer Model for Skeletons . . . 88
6.2.1 Transparency with FileProxy . . . 88
6.2.2 Stage-in and Stage-out . . . 89
6.2.2.1 Initial and Final Staging . . . 89
6.2.2.2 Intermediate Staging . . . 89
6.2.3 The Workspace Abstraction . . . 91
6.2.3.1 Data Division . . . 92
6.2.3.2 Data Reduction . . . 92
6.2.4 File References and Data . . . 92
6.2.4.1 Storage Server . . . 92
6.2.4.2 Reference Counting . . . 93
6.2.4.3 Update Cases . . . 94
6.3 Efficiency . . . 94
6.3.1 BLAST Case Study . . . 95
6.4 Conclusion . . . 96

7 Perspectives and Conclusions **97**
7.1 Perspectives . . . 97
7.1.1 Performance Tuning Extensions . . . 97
7.1.2 Skeleton Stateness . . . 98
7.1.3 AOP for Skeletons . . . 98
7.2 Conclusion . . . 99

II Résumé étendu en français *(Extended french abstract)* **101**

8 Introduction **103**
8.1 Problematic . . . 103
8.1.1 Objectifs et Contributions . . . 104
8.2 Présentation . . . 105
8.3 Itinéraires de Lecture Suggérés . . . 106

9 Résumé **107**
9.1 Réglage du Performance . . . 107
9.1.1 Les Paramètres de Performance . . . 108
9.1.2 Diagnostic de Performance . . . 108
9.1.3 Workout des Muscles . . . 108
9.1.4 Blâmer de Code . . . 109
9.2 System de Typage . . . 109
9.2.1 Règle de Typage . . . 110
9.2.2 Réduction de Sujet . . . 110
9.2.3 Squelettes avec Type Sûr en Java . . . 111
9.3 Transfert de Fichiers . . . 112
9.3.1 Transparence avec FileProxy . . . 113
9.3.2 Stage-in & Stage-out . . . 113
9.3.3 L'abstraction Workspace . . . 113
9.3.4 Stockage de Données & Compteur de Références . . . 114

10 Perspectives et Conclusions **115**
10.1 Perspectives . . . 115
10.1.1 Squelettes sans état . . . 115
10.1.2 AOP pour les squelettes . . . 116
10.2 Conclusions . . . 117

III Appendix **119**

Appendix A: Subject Reduction Proofs **121**

Appendix B: The Skeleton Kitchen **125**

Bibliography **131**

List of Figures

1.1 Suggested Chapter Reading Itineraries 6

2.1 Seamless sequential to multithreaded to distributed objects 27
2.2 Meta-object architecture 28
2.3 Layered features of the ProActive library 31
2.4 Grid PlugTests Evolution 32

3.1 Instructions Reduction Semantics 43
3.2 Stack Interpretation Semantics 49
3.3 Task Abstraction 50
3.4 Task State Diagram 50
3.5 Calcium Framework Task Flow 54
3.6 Calcium interfaced with ProActive Scheduler 55

4.1 Finding the tunable muscle code 62
4.2 Generic Cause Inference Tree for Data Parallelism 63
4.3 N-Queens Skeleton Program 64
4.4 N-Queens Chessboard Filling 65
4.5 Number of ready and processing subtasks for $n = 20$, with 100 nodes. 67

5.1 Skeleton's Reduction Semantics 73
5.2 Skeleton's Type System 75
5.3 From theory to practice: *T-PIPE* rule. 79

6.1 Push Algorithm. 85
6.2 Pull Algorithm. 86
6.3 Performance comparisons. 87
6.4 Proxy Pattern for Files 88
6.5 FileProxy Behavior Example 88
6.6 File Reference Passing Example 93
6.7 BLAST Case Study 95

8.1 Itinéraires de Lecture Suggérés 106

9.1 Performance Tuning 108
9.2 From theory to practice: *T-PIPE* rule. 111
9.3 Proxy Pattern for Files 112

List of Tables

2.1 Non-Object-Oriented Algorithmic Skeleton Frameworks 22
2.2 Object-Oriented Algorithmic Skeleton Frameworks 23
2.3 Skeleton Frameworks Characteristics vs Manifesto Principles . . . 26

3.1 Theory: From Skeletons to Instructions 42
3.2 Practice: From Skeletons to Instructions 47
3.3 Infrastructure Characteristics 52

4.1 Performance Metrics & Workout Summary for $n = 20$, $w \in \{16, 17, 18\}$ 65

6.1 File Scenarios after muscle invocation 94

7.1 Calcium Summary 100

10.1 Calcium Summary 117

List of Listings

3.1 The `Map` skeleton in Calcium 46
3.2 Muscles in Calcium 46
3.3 Calcium Usage Example 59
4.1 Fine Tuning Output for $n = 20, w \in \{16, 17, 18\}$ 66
5.1 Motivation Example: Unsafe skeleton programming 70
5.2 Example of a type safe skeleton program 80
5.3 Typed skeletons with Java Generics 81
6.1 File Transfer API 85
6.2 Calcium Input and Output Example 90
6.3 Muscle Function Example 91

Part I

Thesis

Chapter 1

Introduction

Contents

1.1 Problematic **3**

1.2 Objectives and Contributions **4**

1.3 Overview **5**

1.4 Reading Itineraries **6**

The relevance of parallel programming is evident. There has never been a point in time where we have had a dearer need for parallel programming models to harness the power of increasingly complex parallel systems [Yel08]. On one side large scale distributed-memory computing such as cluster and grid computing [FK99]; and on the other parallel shared-memory computing through new multi-core processors [ABC+06].

The difficulties of parallel programming have led to the development of many parallel programming models, each having its particular strengths. One thing which they have in common is that parallel programming models pursue a balance between simplicity (abstractions) and expressiveness (details), measured ultimately by performance.

Nevertheless, of all the programming models out there, only a few have been embraced as mainstream, while most remain confined in niches. In this thesis we address such a model, algorithmic skeletons.

1.1 Problematic

As recognized in Cole's manifesto [Col04], algorithmic skeletons offer *simplicity*, *portability*, *re-use*, *performance*, and *optimization*; but have yet failed to reach mainstream in parallel programming. With this goal in mind, Cole's manifesto has proposed four key principles to guide the development of algorithmic skeleton systems: *minimal conceptual disruption*, *integrate ad-hoc parallelism*, *accommodate diversity*, and *show the pay-back*.

If we look at the evolution of algorithmic skeleton programming, we can see that it has varied greatly since the term was coined by Cole [Col91]. Perhaps

the most significant change has been the realization that algorithmic skeletons should be provided as libraries instead of languages. Indeed, as will be discussed later on this thesis, most of the recently developed skeleton frameworks have been provided as libraries in object-oriented languages. This is in accordance with the first principle of Cole's manifesto: *minimal conceptual disruption*.

The implications of having skeletons as libraries has changed the way in which we envision the design and implementation of skeleton frameworks, and more importantly how the programmer interacts with the framework.

We believe that offering algorithmic skeletons as libraries implies that skeletons no longer have to provide support for all types of parallel applications, but can concentrate on Cole's first and fourth principles: *minimal conceptual disruption* and *showing the pay-back*. The assumption is that programmers will use algorithmic skeleton libraries for what they are good at: structured parallel programming, while they will choose some other parallel programming model for what they are not good at: irregular parallel applications. Thus, for complex applications, a mixture of libraries, each implementing a different parallel programming model, will likely be used.

Several works on the literature have argued that cost models are one of the strength in algorithmic skeleton programming. While this is a good property to have, other programming models such as MPI which do not have cost models have proliferated, and are the *de facto* standard way of achieving parallel and distributed programming. Indeed, we believe that by them selves cost models are not enough to tip the balance in favor of algorithmic skeletons, and thus further advances in other aspects are needed.

Therefore, in this thesis we pursue other features which might help programmers adopt algorithmic skeleton programming. Namely performance tuning, type safe composition and file access/transfer.

1.2 Objectives and Contributions

The main objective of this thesis is the design and implementation of a skeleton library capable of executing applications in parallel and distributed infrastructures.

The main contributions of this thesis are:

- A survey on the state of the art of algorithmic skeleton programming.
- A model for algorithmic skeleton programming and its implementation as a Java library, Calcium, featuring: nestable, task and data parallel, skeletons; and multiple environments for parallel and distributed executions among others.
- A performance tuning model for algorithmic skeletons and its implementation in Calcium [CL07].
- A type system for nestable algorithmic skeleton and its implementation using Java Generics in Calcium [CHL08].

- File access/transfer contributions. First, an active object based file transfer model and its implementation in ProActive [BCLQ06, BCL07]. Second, a transparent file access model for algorithmic skeletons and its implementation in Calcium [CL08].

1.3 Overview

This document is organized as follows:

- Chapter 2 provides a state of the art on algorithmic skeleton programming frameworks. The chapter begins with a description of well-known parallel and distributed programming models. Then, the chapter provides a description of several well known algorithmic skeleton frameworks. The descriptions try to provide a brief summary and highlight the main characteristics of each framework. Also, a discussion is made where the skeleton frameworks are compared and the work in this thesis is positioned with respect to the state of the art.
- Chapter 3 provides an introduction and a description of the proposed skeleton framework: Calcium. The chapter begins with a description of the supported skeleton patterns in Calcium. Then, the chapter describes the hypotheses upon which Calcium's skeleton programming model is built. The chapter continues with the formalization of the programming model which shows how parallelism is achieved. Then, the chapter moves on to describing how this model is implemented in Java, and also how the framework can support multiple execution environments. The chapter finishes with a concrete example: a naive solution for finding prime numbers.
- Chapter 4 presents a performance tuning model for algorithmic skeletons. The chapter aims at bringing performance debugging to the abstraction level of algorithmic skeletons. For this, performance metrics are gathered and an inference tree is used to find the possible cause of the *performance bug*. The cause is then related back to the skeleton code suspected of causing the performance bug. Experimental validations are made with an NQueens skeleton based application.
- Chapter 5 defines a theoretical type system for skeleton and proves that this type system is indeed safe as it guarantees the subject reduction property. The chapter then addresses the implementation of such a type system in Java using Generics.
- Chapter 6 defines a transparent, non-invasive, file access/transfer model for algorithmic skeletons. The model is implemented in Calcium and experimental benchmarks are made with a BLAST skeleton based application.
- Chapter 7 concludes this thesis by providing future research perspectives and summarizing the contributions.

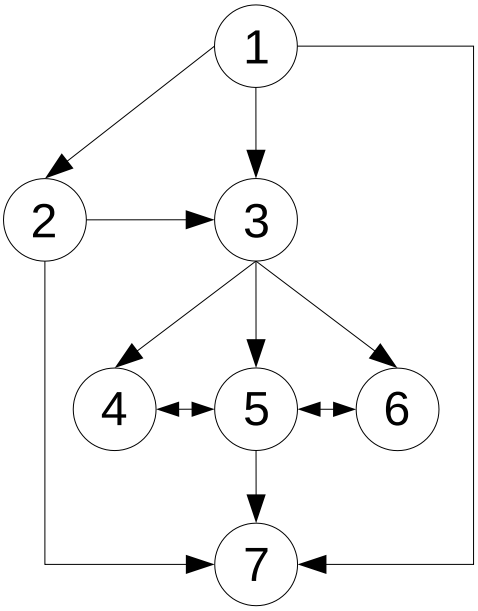


Figure 1.1: Suggested Chapter Reading Itineraries

1.4 Reading Itineraries

This thesis can be navigated in several ways depending on the reader's preference. The suggested itineraries are shown in Figure 1.1 and are detailed as follows:

- **Contribution Summary**. This corresponds to the introduction and the conclusion of the thesis which outline the main contributions.
  The suggested itinerary is: 1→7.
- **Contribution Summary and Context**. Shows the main contributions and the context of this thesis.
  The suggested itinerary is: 1→2→7.
- **Contribution Core**. Shows the main contribution of this thesis in depth. The orders in which Chapters 4, 5, and 6 are read is indifferent.
  The suggested itinerary is: 3→ (4 | 5 | 6).
- **Complete**. This is the complete reading of this thesis.
  The suggested itinerary is: 1→2→ 3→ (4 | 5 | 6) → 7.

Chapter 2

State of the Art

Contents

2.1 Distributed Programming Models 7

2.2 Algorithmic Skeletons Frameworks 10

2.3 Discussion 20

2.4 Context: The ProActive Library 27

2.5 Conclusion 33

This chapter provides relevant background and a state of the art on algorithmic skeleton programming. The chapter begins with a general background on distributed programming models. Then, the second section provides a non-exhaustive survey on algorithmic skeleton frameworks. For each framework, a brief description is provided and its main features are highlighted. Then, the third section discusses and compares the different algorithmic skeleton frameworks. The fourth section provides relevant background on the ProActive Library, and the final section concludes the chapter.

2.1 Distributed Programming Models

2.1.1 Remote Procedure Calls

Remote Procedure Calls (RPC) [TA90] is the mechanism by which procedures are executed on distant machines with a different address space. Several software tools have been implemented on top of RPC, such as the popular NFS (Network File System) protocol, NIS (Network Information System), etc. The advantage of RPC over regular network connections is that programmers can invoke remote procedures without the burdens of network communications such as: port connections, socket creations, data writing/reading, etc. A drawback of RPC is that client programs are not aware if remote procedure are successfully executed on the remote address space. Additionally, both client and server must have a prearranged understanding on the procedure's identification numbers and their versions.

More recently, GridRPC has been proposed [SNM+02], offering services such as asynchronous coarse-grained parallels tasks. In GridRPC, remote calls are delegated to intermediate agents which handle the scheduling and load balancing of invocations. Each remote call is assigned an identifier which can be used to cancel or await the call.

Unfortunately, both traditional RPC and GridRPC are limitted to well defined point to point client-server interactions.

2.1.2 Message passing

Message passing programming models, such as MPI [(MP96], are probably the most popular programming models in the absence of shared memory environments (e.g. clusters). This is likely due to their efficiency, latency management, modularity and synchronization operations. Compared with RPC, MPI not only provides point to point but also one to many communications. MPI provides other important operations such as send/receive, gather and reduction, and synchronization (barriers) among others.

Unfortunately, MPI's high efficiency is achieved at the cost of low level abstractions. Programmers are burdened with low level communication details. For example, MPI programmers must manually deal with data serialization and type casting. Also, the message passing paradigm is error prone, being relatively easy for a programmer to obtain deadlocks. Furthermore, Gorlatch has also pointed out the dangers of send/receive operations in contrast with collective operations [Gor04].

2.1.3 Bulk Synchronous Parallelism (BSP)

Bulk-Synchronous Parallel (BSP) is a programming model introduced by Valiant [Val90]. BSP offers an abstract view of PRAM like models which allows portable prediction of performance on many architectures.

In BSP, a computer has a set of processor memory pairs. Processors are connected through a network and exchange information at well defined points in the application. A BSP program is executed as a sequence of super-steps, each one divided into three phases. In the first phase computations take place using local available data and requests for remote data are sent to other processors. In the second stage, the requested data is delivered to the processors, and in the third stage a global synchronization takes place.

The main advantages of BSP are its simplicity, portability, and the ability to predict the execution time of an application with cost calculus models [LHF00, Lou01]. The execution time of an application is the sum of all its super-steps. The execution time of a super-step is the sum of the maximum time for each phase. Therefore, the performance of an application on a different architecture can be predicted by adjusting the architecture dependant parameters of the cost calculus model [HM99].

Besides cost models, BSP formalisms have been defined to address other issues. For example, exception management [GGLD06, GL07], type checking for

invalid compositions [GL03, GL05], and parallel juxtaposition for divide-and-conquer algorithms [LBGLR06, Lou03] among others.

2.1.4 Distributed Objects

Distributed objects provide a higher level of abstraction than both RPC and MPI. In a distributed object environment, objects can be remote entities accessible via remote method invocations. A remote method invocation can change the state of the object; and can be achieved as a standard part of the language (like Java RMI) or through a middleware layer like CORBA [Gro99]. There are many distributed object libraries, such as the already mentioned CORBA and Java RMI; but also DCOM which is a Microsofit platform; Distributed Ruby; etc.

As stated by Emmerich [Emm00], there are several differences between regular object oriented programming and distributed objects. Distributed objects must also deal with: life cycle, object references, request latency, object activation, parallelism, communication, failure, and security. Additionally, one of the goals of distributed objects is to bridge the heterogeneity of different architectures. In CORBA, this is achieved with an interface description language (IDL) and its mapping to other languages. In Java RMI the idea is to have heterogeneous systems running portable Java applications, furthermore unknown object types can be downloaded from distant nodes and instantiated at runtime.

2.1.5 Distributed Components

There are many component models for distributed computing both industry or academy oriented. No standard definition of component exists, but a frequently cited is [Szy98]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Therefore, component based programming addresses three concerns of software development: encapsulation, composition and description. The encapsulation sees components as black-boxes, with well defined services accessible through interfaces. The composition means that components can be assembled to work together, and in hierarchical models new components can be created from sub-components. The description means that the assembly of components is defined with architecture description languages, which allows for the composition of tools and type verification.

Industry oriented components tend to focus more on the development life cycle of components. Some of the most known industry oriented components models are Sun's Enterprise Java Beans (EJBs) [MICb], Microsft's .NET [Mica], and the Corba Component Model (CCM) [Gro99]. CCM components define a role for each actor of the programming process: designer, implementer, packager, deployer and end-user.

On the other side, academy oriented components target specific research interests. For example the Commmon Component Architecture (CCA) [For] targets high-performance computing with a minimal component architecture. Another example is the Grid Component Model (GCM) [Cor06] which is based on Fractal [BCL+06] but extends it for distributed and Grid computing. As in Fractal, GCM allows for hierarchical composition, separation of functional and non-functional interfaces; but also considers deployment, collective communications [BCHM07], and autonomic behavior [ACD+08] among others.

2.1.6 Workflows and Service Oriented Architectures

We consider a workflow to be a coordination of a set of loosly coupled and remotely available services (and tasks). A workflow system is composed of mainly four elements: design, scheduling, fault tolerance, and data movement [YB05]. Workflow systems come in many flavors depending on their domains such as scientific applications (e.g. SAWE [TZ07a, TZ07b]), problem solving environment (e.g. SEGL [CLKRR05]), industry oriented (e.g. BPEL [OAS07]), etc. In general, workflows are programmed with a high-level language (or GUI) which describes the flow of data from one service to the next. A workflow enactment system is in charge of the workflow's scheduling in accordance with the program's control logic, fault tolerance, and data transfer management.

A feature that distinguishes workflows from other programming models is the role of the data movement. While other models are mostly concerned with transfer of small quantities of data (stored in main memory), workflows provide abstraction to transfer large amounts of data (located in secondary storage) between workflow units. For example, Java CoG Kit's [vLFGL01] data transfer operations are explicitly defined like any other task, in the sense that a data transfer operation must be submitted for execution as a data-transfer-task [vLAG+03, vLGP+04]. Another example is Unicore [ES01, Uni], which uses a workflow programing model to order dependencies between tasks. All tasks belonging to the same job share a jobspace file system abstraction. The job description also specifies which files must be imported into the jobspace before the execution of the job, and which files must be exported after the job is finished. Files that must be imported and exported to the jobspace are staged before and after the job begins. Additionally, it is also possible to interact with sub-jobs (which have their own jobspace) by explicitly adding file transfer modules in the workflow. The file transfer modules handle the input and output of files between the jobspace and the sub-jobspaces.

Thus workflows require programmers to explicitly add data management units to their applications.

2.2 Algorithmic Skeletons Frameworks

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing introduced by Cole in [Col91]. Skeletons take advantage of common programming patterns to hide the complexity

of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones.

The most outstanding feature of algorithmic skeletons, which differentiates them from other high-level programming models, is that message passing is implicitly defined by the skeleton patterns. Programmers do not have to specify the message passing between the application's sequential parts. This yields two implications. First, as the communication patterns are known in advance, cost models can be applied to schedule skeletons programs [HM99]. Second, that skeleton programming is deadlock free.

There are not many surveys concerning algorithmic skeleton in the literature. Of those available, each addresses a specific concern. The first one is that of Hammond and Michaelson [HM99] which provides a broad background on parallel programming models related with functional programming. This survey covers a variety of topics such as cost models, BSP models, coordination languages and algorithmic skeletons. Another survey corresponds to that of Hamdan [Ham99] which focuses on cost models for algorithmic skeletons. A more recent survey, and dedicated to the subject of algorithmic skeletons and patterns, has been compiled by Rabhi and Gorlatch [RG03]. This survey provides in depth description of several well known skeleton frameworks such as HDC, Eden, and P3L among others. A more general survey addressing environments for parallel and distributed computing can be found in [DDdSL02].

In lack of a recent survey in the literature, the rest of this section provides a non-exhaustive survey on algorithmic skeleton frameworks, which focuses on research pluralism and recent results. For each skeleton framework in the survey, a brief description is provided and its relevance and main features are highlighted. Then, in the Section 2.3 a comparison between the frameworks is presented.

2.2.1 Alt and HOC-SA

Alt et al. [Alt07] have proposed a skeleton framework for the Grid [ABG02, AG03c, AG03b, AG03a]. We shall call this framework *Alt* for lack of a better name. In Alt, skeletons are offered as services accessible through Java Remote Method Invocation (RMI). Once a skeleton service has been found, the skeleton is remotely called from the client program using an `invoke` API. Special container objects are used to pass parameters and obtain results between different skeleton invocations. Contrary to other frameworks, skeletons are not nestable. The control flow between skeletons is explicitly manipulated by the programmer from inside the client application. Thus, type verification is explicitly performed by Java in the client program. Additionally, a Grid Workflow Description Language (GWorkflowDL) can be used to define the execution flow between skeleton services.

Higher Order Components (HOC) combine concepts from skeleton, component, and services [DG04]. HOC are remotely accessible by clients as distant services which implement a parallelism pattern. A remote client provides the specific application code to the HOC and the input data. The code and data

are shipped from the client host to the remote service host. Then, the HOC is deployed and executed on the remote infrastructure in accordance with the specific HOC pattern. Once the computation is finished, the result is delivered back to the programmer. The main challenges behind HOC correspond to the storage and look up of remote services, and the code shipping from the client. In other words, HOC is mainly focused on how skeleton can be remotely accessed as services. For example, a HOC wrapping of eSkel's (see 2.2.5) *pipeline* skeleton is described in [DBCG05], and of Lithium's *farm* skeleton in [DGC+05]. On the backend, several middlewares have been studied for HOC such as: Globus [DG04], ProActive [DGB+06], and KOALA [DEDG06].

2.2.2 ASSIST

ASSIST [ACD+06] is a programming environment which provides programmers with a structured coordination language. The coordination language can express parallel programs as an arbitrary graph of software modules. The module graph describes how a set of modules interact with each other using a set of typed data streams. The modules can be sequential or parallel. Sequential modules can be written in C, C++, or Fortran; and parallel modules are programmed with a special ASSIST parallel module (*parmod*).

AdHoc, a hierarchical and fault-tolerant Distributed Shared Memory (DSM) system is used to interconnect streams of data between processing elements by providing a repository with: get/put/remove/execute operations [AADJ07, ADG+05, AT04]. Research around AdHoc has focused on transparency, scalability, and fault-tolerance of the data repository.

While not a classical skeleton framework, in the sense that no skeletons are provided, ASSIST's generic `parmod` can be specialized into classical skeletons such as: *farm*, *map*, etc. ASSIST also supports autonomic control of `parmod`s [APP+05], and can be subject to a performance contract by dynamically adapting the number of resources used.

2.2.3 CO_2P_3S

CO_2P_3S (Correct Object-Oriented Pattern-based Parallel Programming System), is a pattern oriented development environment [MAB+02, MSS99, MSSB00], which achieves parallelism using threads in Java.

CO_2P_3S is concerned with the complete development process of a parallel application. Programmers interact through a programming GUI to choose a pattern and its configuration options. Then, programmers fill the hooks required for the pattern, and new code is generated as a framework in Java for the parallel execution of the application. The generated framework uses three levels, in descending order of abstraction: patterns layer, intermediate code layer, and native code layer. Thus, advanced programmers may intervene the generated code at multiple levels to tune the performance of their applications. The generated code is mostly type safe, using the types provided by the programmer which do

not require extension of superclass, but fails to be completely type safe such as in the `reduce(..., Object reducer)` method in the *mesh* pattern.

The set of patterns supported in CO_2P_3S corresponds to *method-sequence*, *distributor*, *mesh*, and *wavefront*. Complex applications can be built by composing frameworks with their object references. Nevertheless, if no pattern is suitable, the MetaCO_2P_3S [BMA+02, MSS+02] graphical tool addresses extensibility by allowing programmers to modify the pattern designs and introduce new patterns into CO_2P_3S.

Support for distributed memory architectures in CO_2P_3S was introduced in [TSS+03]. To use a distributed memory pattern, programmers must change the pattern's memory option from shared to distributed, and generate the new code. From the usage perspective, the distributed memory version of the code requires the management of remote exceptions.

2.2.4 Eden

Eden [LOmRP05] is a parallel programming language for distributed memory environments, which extends Haskell [HJW+92]. Processes are defined explicitly to achieve parallel programming, while their communications remain implicit [BLMP97]. Processes communicate through unidirectional channels, which connect one writer to exactly one reader. Programmers only need to specify which data a processes depends on. Eden's process model provides direct control over process granularity, data distribution and communication topology.

Eden is not a skeleton language in the sense that skeletons are not provided as language constructs. Instead, skeletons are defined on top of Eden's lower-level process abstraction, supporting both task and data parallelism. Eden introduces the concept of *implementation skeleton* [KLPR01], which is an architecture independent scheme that describes a parallel implementation of an algorithmic skeleton.

Recent research on Eden has focused on scalability. A hierarchical taskpool has been provided, capable of handling dynamically created tasks in the computation nodes [Pri06, BDLP08].

2.2.5 eSkel

The Edinburgh Skeleton Library (eSkel) is provided in C and runs on top of MPI [(MP96]. The first version of eSkel was described in [Col04], while a later version is presented in [BCGH05b]. An example of a baseline stereo application programmed with eSkel can be found in [BCGH05d].

In [BC05], nesting-mode and interaction-mode for skeletons are defined. The nesting-mode can be either transient or persistent, while the interaction-mode can be either implicit or explicit. Transient nesting means that the nested skeleton is instantiated for each invocation and destroyed afterwards, while persistent means that the skeleton is instantiated once and the same skeleton instance will be invoked throughout the application. Implicit interaction means that the

flow of data between skeletons is completely defined by the skeleton composition, while explicit means that data can be generated or removed from the flow in a way not specified by the skeleton composition. For example, a skeleton that produces an output without ever receiving an input has explicit interaction.

Performance prediction for scheduling and resource mapping, mainly for *pipelines*, has been explored by Benoit et al. [BCGH04, BCGH05a, BCGH05c, BR07]. They provided a performance model for each mapping, based on process algebra, and determine the best scheduling strategy based on the results of the model.

More recent works have addressed the problem of adaptation on structured parallel programming [YCGH07, GVC07], in particular for the pipe skeleton [GVC06, GVC08].

2.2.6 HDC

Higher-order Divide and Conquer (HDC) [HL00] is a subset of the functional language Haskell [HJW+92]. Functional programs are presented as polymorphic higher-order functions, which can be compiled into C/MPI, and linked with skeleton implementations. The language focus on divide and conquer paradigm, and starting from a general kind of divide and conquer skeleton, more specific cases with efficient implementations are derived. The specific cases correspond to: fixed recursion depth, constant recursion degree, multiple block recursion, elementwise operations, and correspondant communications [Her00].

HDC pays special attention to the subproblem's granularity and its relation with the number of available processors. The total number of processors is a key parameter for the performance of the skeleton program as HDC strives to estimate an adequate assignment of processors for each part of the program. Thus, the performance of the application is strongly related with the estimated number of processors leading to either exceeding number of subproblems, or not enough parallelism to exploit available processors.

2.2.7 JaSkel

JaSkel [FSP06] is a Java based skeleton framework providing skeletons such as *farm*, *pipe* and *heartbeat*. Skeletons are specialized using inheritance. Programmers implement the abstract methods for each skeleton to provide their application specific code. Skeletons in JaSkel are provided in both sequential, concurrent and dynamic versions. For example, the concurrent farm can be used in shared memory environments (threads), but not in distributed environments (clusters) where the distributed farm should be used. To change from one version to the other, programmers must change their classes' signature to inherit from a different skeleton. The nesting of skeletons uses the basic Java `Object` class, and therefore no type system is enforced during the skeleton composition.

The distribution aspects of the computation are handled in JaSkel using AOP, more specifically the AspectJ [KHH+01] implementation. Thus, JaSkel can be deployed on both cluster and Grid like infrastructures [SP07, AHM+07]. Nevertheless, a drawback of the JaSkel approach is that the nesting of the skeleton

strictly relates to the deployment infrastructure. Thus, a double nesting of farm yields a better performance than a single farm on hierarchical infrastructures. This defeats the purpose of using AOP to separate the distribution and functional concerns of the skeleton program.

2.2.8 Lithium and Muskel

Lithium [AD99, ADD04, ADT03, DT02] and its successor Muskel [DD06b] are skeleton frameworks developed at University of Pisa, Italy. Both of them provide nestable skeletons to the programmer as Java libraries. The evaluation of a skeleton application follows a formal definition of operational semantics introduced by Aldinucci and Danelutto [AD04, AD07b], which can handle both task and data parallelism. The semantics describe both functional and parallel behavior of the skeleton language using a labeled transition system. Additionally, several performance optimization are applied such as: skeleton rewriting techniques [ADT03, AD99], task lookahead, and server-to-server lazy binding [ADD04].

At the implementation level, Lithium exploits macro-data flow [Dan01, Dan99] to achieve parallelism. When the input stream receives a new parameter, the skeleton program is processed to obtain a macro-data flow graph. The nodes of the graph are macro-data flow instructions (MDFi) which represent the sequential pieces of code provided by the programmer. Tasks are used to group together several MDFi, and are consumed by idle processing elements from a task pool. When the computation of the graph is concluded, the result is placed into the output stream and thus delivered back to the user.

Muskel also provides non-functional features such as Quality of Service (QoS) [Dan05]; security between task pool and interpreters [AD07a, AD08]; and resource discovery, load balancing, and fault tolerance when interfaced with *Java / Jini Parallel Framework* (JJPF) [DD05], a distributed execution framework. Muskel also provides support for combining structured with unstructured programming [DD06a] and recent research has addressed extensibility [ADD07].

2.2.9 Mallba

Mallba [AAB+02] is a library for combinatorial optimizations supporting *exact*, *heuristic* and *hybrid* search strategies [AAB+06]. Each strategy is implemented in Mallba as a generic skeleton which can be used by prodiving the required code. On the exact search algorithms Mallba provides branch-and-bound and dynamic-optimization skeletons. For local search heuristics Mallba supports: hill climbing, metropolis, simulated annealing, and tabu search; and also population based heuristics derived from evolutionary algorithms such as genetic algorithms, evolution strategy, and others (CHC). The hybrid skeletons combine strategies, such as: GASA, a mixture of genetic algorithm and simulated annealing, and CHCCES which combines CHC and ES.

The skeletons are provided as a C++ library and are not nestable but type safe. A custom MPI abstraction layer is used, `NetStream`, which takes care

of primitive data type marshalling, synchronization, etc. A skeleton may have multiple lower-level parallel implementations depending on the target architectures: sequential, LAN, and WAN. For example: centralized master-slave, distributed master-slave, etc.

Mallba also provides state variables which hold the state of the search skeleton. The state links the search with the environment, and can be accessed to inspect the evolution of the search and decide on future actions. For example, the state can be used to store the best solution found so far, or α, β values for branch and bound pruning [ALG+07].

Compared with other frameworks, Mallba's usage of skeletons concepts is unique. Skeletons are provided as parametric search strategies rather than parametric parallelization patterns.

2.2.10 P^{3}L, SkIE, SKElib

P^{3}L [BDO+95, BCD+97] is a skeleton based coordination language. P^{3}L provides skeleton constructs which are used to coordinate the parallel or sequential execution of C code. A compiler named Anacleto [CDF+97] is provided for the language. Anacleto uses implementation templates to compile P^{3}L code into a target architecture. Thus, a skeleton can have several templates each optimized for a different architecture. A template implements a skeleton on a specific architecture and provides a parametric process graph with a performance model. The performance model can then be used to decide program transformations which can lead to performance optimizations [ACD98].

A P^{3}L module corresponds to a properly defined skeleton construct with input and output streams, and other sub-modules or sequential C code. Modules can be nested using the two tier model, where the outer level is composed of task parallel skeletons, while data parallel skeletons may be used in the inner level [DPP97]. Type verification is performed at the data flow level, when the programmer explicitly specifies the type of the input and output streams, and by specifying the flow of data between sub-modules.

SkIE [BDPV99] is quite similar to P^{3}L, as it is also based on a coordination language, but provides advanced features such as debugging tools, performance analysis, visualization and graphical user interface. Instead of directly using the coordination language, programmers interact with a graphical tool, where parallel modules based on skeletons can be composed.

SKELib [DS00] builds upon the contributions of P^{3}L and SkIE by inheriting, among others, the template system. It differs from them because a coordination language is no longer used, but instead skeletons are provided as a library in C, with performance similar as the one achieved in P^{3}L. Contrary to Skil (see 2.2.16), another C like skeleton framework, type safety is not addressed in SKELib.

2.2.11 PAS and EPAS

PAS (Parallel Architectural Skeletons) is a framework for skeleton programming developed in C++ and MPI [GSP99, GSP02]. Programmers use an extension of C++ to write their skeleton applications[1]. The code is then passed through a Perl script which expands the code to pure C++ where skeletons are specialized through inheritance.

In PAS, every skeleton has a Representative (`Rep`) object which must be provided by the programmer and is in charge of coordinating the skeleton's execution. Skeletons can be nested in a hierarchical fashion via the `Rep` objects. Besides the skeleton's execution, the `Rep` also explicitly manages the reception of data from the higher level skeleton, and the sending of data to the sub-skeletons. A parametrized communication/synchronization protocol is used to send and receive data between parent and sub-skeletons.

An extension of PAS labeled as SuperPas [AGL04] and later as EPAS [ASGL05] addresses skeleton extensibility concerns. With the EPAS tool, new skeletons can be added to PAS. A Skeleton Description Language (SDL) is used to describe the skeleton pattern by specifying the topology with respect to a virtual processor grid. The SDL can then be compiled into native C++ code, which can be used as any other skeleton.

2.2.12 SBASCO

SBASCO (Skeleton-BAsed Scientific COmponents) is a programming environment oriented towards efficient development of parallel and distributed numerical applications [DRST04]. SBASCO aims at integrating two programming models: skeletons and components with a custom composition language. An *application view* of a component provides a description of its interfaces (input and output type); while a *configuration view* provides, in addition, a description of the component's internal structure and processor layout. A component's internal structure can be defined using three skeletons: *farm*, *pipe* and *multi-block*.

SBASCO's addresses domain decomposable applications through its *multi-block* skeleton. Domains are specified through arrays (mainly two dimensional), which are decomposed into sub-arrays with possible overlapping boundaries. The computation then takes place in an iterative BSP like fashion. The first stage consists of local computations, while the second stage performs boundary exchanges. A use case is presented for a reaction-diffusion problem in [DRR+06b].

Two type of components are presented in [DRR+05]. Scientific Components (SC) which provide the functional code; and Communication Aspect Components (CAC) which encapsulate non-functional behavior such as communication, distribution processor layout and replication. For example, SC components are connected to a CAC component which can act as a manager at runtime by dynamically re-mapping processors assigned to a SC. A use case showing improved performance when using CAC components is shown in [DRR+06a].

[1]The authors claim that it is not really a C++ extension, but a textual interface, as programmers can also code directly in pure C++.

2.2.13 SCL

The Structured Coordination Language (SCL) was one of the first languages introduced for skeleton programming [DFH+93, DGTY95, DkGTY95]. SCL is considered a base language, and was designed to be integrated with a host language, for example Fortran. In SCL, skeletons are classified into three types: configuration, elementary and computation. Configuration skeletons abstract patterns for commonly used data structures such as distributed arrays (`ParArray`). Elementary skeletons correspond to data parallel skeletons such as `map`, `scan`, and `fold`. Computation skeletons which abstract the control flow and correspond mainly to task parallel skeletons such as `farm`, `SPMD`, and `iterateUntil`.

2.2.14 SKiPPER, QUAFF

SKiPPER is a domain specific skeleton library for vision applications [SGD99, SG02] which provides skeletons in CAML [Mau95], and thus relies on CAML for type safety. Skeletons are presented in two ways: declarative and operational. Declarative skeletons are directly used by programmers, while their operational versions provide an architecture specific target implementation. From the runtime environment, CAML skeleton specifications, and application specific functions (provided in C by the programmer), new C code is generated and compiled to run the application on the target architecture. One of the interesting things about SKiPPER is that the skeleton program can be executed sequentially for debugging.

Different approaches have been explored in SKiPPER for writting operational skeletons: static data-flow graphs, parametric process networks, hierarchical task graphs, and tagged-token data-flow graphs [SG02].

QUAFF [FSCL06] is a more recent skeleton library written in C++ and MPI. QUAFF relies on template-based meta-programming techniques to reduce runtime overheads and perform skeleton expansions and optimizations at compilation time. Skeletons can be nested and sequential functions are stateful. Besides type checking, QUAFF takes advantage of C++ templates to generate, at compilation time, new C/MPI code. QUAFF is based on the CSP-model, where the skeleton program is described as a *process network* and production rules (single, serial, par, join) [FS07].

2.2.15 SkeTo

The SkeTo [MIEH06] project is a C++ library which achieves parallelization using MPI. SkeTo is different to other skeleton libraries because instead of providing nestable parallelism patterns, SkeTo provides parallel skeletons for parallel data structures such as: lists [MIEH06], trees [MHT03, MHT06], and matrices [EHKT05]. The data structures are typed using templates, and several parallel operations can be invoked on them. For example the list structure provides parallel operations such as: *map*, *reduce*, *scan*, *zip*, *shift*, etc...

Additional research around SkeTo has also focused on optimizations strategies by transformation, and more recently domain specific optimizations [EMHT07].

For example, SkeTo provides a fusion transformation [MKI+04] which merges two successive function invocations into a single one, thus decreasing the function call overheads and avoiding the creation of intermediate data structures passed between functions.

2.2.16 Skil and Muesli

Skil [BK95, BK96a, BK96b, BK96c, BK98] is an imperative language for skeleton programming. Skeletons are not directly part of the language but are implemented with it. Skil uses a subset of C language which provides functional language like features such as higher order functions, curring and polymorphic types. When Skil is compiled, such features are eliminated and a regular C code is produced. Thus, Skil transforms polymorphic high order functions into monomorphic first order C functions. Skil does not support nestable composition of skeletons. Data parallelism is achieved using specific data parallel structures, for example to spread arrays among available processors. Filter skeletons can be used.

The Muesli skeleton library [KS02, Kuc02, KS05] re-implements many of the ideas introduced in Skil, but instead of a subset of the C language, skeletons are offered through C++. Contrary to Skil, Muesli supports nesting of task and data parallel skeletons [KC02] but is limited to P^3L's two tier approach (see 2.2.10). C++ templates are used to render skeletons polymorphic, but no type system is enforced. The supported skeletons are distributed *array* and *matrix* for data parallelism; and *pipeline*, *farm*, and *parallel composition* (a farm variant).

Recent research has focused mostly on optimizations [Kuc04] and scalability for specific skeletons such as: farm [PK05b, PK05a, PK08a], branch and bound [PK06] and divide and conquer [PK08b].

2.2.17 TBB

TBB (Threading Building Blocks) is a C++ library for parallel programming developed by Intel to take advantage of multi-core architectures [Int]. TBB offers parallel patterns such as: *for*, *reduce*, *scan*, *do*, *sort*, and *pipeline*; and concurrent data structures as: *hashmap*, *vector*, and *queue*. An interesting aspect of TBB is that, contrary to most of the frameworks described here, TBB was created by an industry company instead of an academic institution.

To our knowledge, TBB does not label itself as a skeleton framework, but we provide a description here because of the evident relationship with skeleton programming: parallelism is abstracted through patterns. Compared with skeleton frameworks, TBB provides lower level abstractions with more control on lower level parallelism aspects such as: granularity, the possibility to combine with other thread libraries, and direct access to the task scheduler. Additionally, TBB is only aimed at shared-memory infrastructures, in particular multi-core.

2.2.18 Others

Besides the previously mentioned frameworks, there are other works which have drawn our attention for different reasons. We briefly describe them here.

Maude [CDE+05] is a high level, general purpose language and high performance system supporting both equational and rewriting logic computations. Maude is not solely dedicated to skeleton programming, as it can be used to specify and analyze a wider variety of systems and protocols. For example, Eden's semantics have been analyzed with Maude [HHVOM07]. Concerning skeletons, Maude has implemented non nestable skeletons to show how distributed computation can be achieved with Maude [RV07a]. These skeletons correspond to *farm*, *systolic*, *d&c*, *branch & bound*, and *pipeline* [RV07b].

PLPP (Pattern Language for Parallel Programming) is a collection of patterns which are structured to provide a design methodology [MMS00, MMS01]. The methodology helps programmers understand the parallelization opportunities of their problem, and then work through a series of patterns to finally arrive at a solution for their application. Thus, contrary to other skeleton frameworks discussed here, PLPP focuses mainly on the design methodology of the application, rather than the specific parallelism patterns.

MapReduce is a library developed by Google, Inc. to perform computations on large amounts of data in parallel [DG08, Läm07]. MapReduce is not a self proclaimed skeleton framework, and is quite different from other skeleton frameworks we have described. Nontheless, we mention it for two reasons. First, because MapReduce is an industry developed tool, much like TBB, but also because it is currently used in production environments extensively. The second reason is that MapReduce is concerned with data intensive applications, while most skeleton frameworks are focused mainly on computation intensive applications.

MapReduce is strongly related with algorithmic skeleton programming because it provides a parametric pattern of parallelism. MapReduce provides a single skeleton targetted for the division of large data sets, distribution of the data blocks, computation of the blocks and then reduction of the results. Additionally, MapReduces applies a set of particular features which are key to improve performance on a production environment. First, *fault tolerance* for worker failures. Also *locality awareness*, which performs computations in machines where a replica of the data already exists. Management of *stragglers*, towards the end of the computation, with redundancy to decrease the time waiting for the last task completion.

2.3 Discussion

2.3.1 Programming Models

It is evident from the descriptions outlined in Section 2.1 that each programming model provides different abstractions. Ideally, one would like to order the programming models from the least abstract to the most abstract, but this is

unrealistic. An alternative would be to classify them into two groups: low level and high level abstractions. Thus, in the low level abstractions group we could find RPC, MPI and BSP; while on the higher level abstractions we could find object-oriented, components, workflows, and algorithmic skeletons.

Of course this classification is completely arbitrary as it would depend on the particular implementation of the programming model. Thus, instead of arguing about the abstraction of the programming models, we are more interested in pointing out how strongly they are intertwined.

Indeed, few implementations of a programming model are directly implemented with sockets. More often we see that a particular implementation is based on another programming model. For example, the reference implementation of the Grid Component Model (GCM) is built on top of ProActive which implements a distributed object programming model: active objects.

With respect to algorithmic skeleton, we have found that they have be constructed on top of several programming models. For example, many frameworks (e.g. eSkel) are implemented on top of MPI. Zavanella has studied the relationship between skeletons and BSP [Zav01], and SBASCO uses a BSP like approach as well. Other skeleton frameworks are constructed on top of distributed object (e.g. Lithium).

Nevertheless, what is also interesting is that skeletons concepts have permeated into other programming models. For example, Alt [Alt07] has provided a workflow language to compose skeletons (GWorkflowDL). Or in the case of components Haskel# provides a mechanism to specify component bindings from skeleton patterns [dCJL03]. Or in the case of Fractal components [BCL+06], where in [BPP06] the master-slave pattern has been studied.

Therefore, it is our opinion that algorithmic skeletons are not meant to replace other programming models, but can ultimately be combined with other models to facilitate parallel and distributed programming.

2.3.2 Algorithmic Skeletons

From Section 2.2, it is clear that algorithmic skeletons have evolved greatly since their introduction by Cole [Col91]. Tables 2.1 and 2.2 provide a summary of the skeleton frameworks. Table 2.1 shows classical non-object-oriented based skeleton frameworks, while Table 2.2 shows object-oriented skeleton frameworks. Note that while ASSIST, SBASCO and SkiE are implemented on top of C++, we have placed them in Table 2.1 because their programming languages do not correspond to object-oriented languages.

The tables consider several characteristics such as: programming language, execution language, distribution environment, type safety, pattern nesting, file access, and skeleton set.

Programming Language is the interface with which programmers interact to code their skeleton applications. These languages are diverse, encompassing paradigms such as: functional languages, coordination languages, markup languages, imperative languages, object oriented languages, and

	Activity Years	Programming Language	Execution Language	Distribution Library	Type Safe	Skeleton Nesting	File Access	Skeleton Set
ASSIST	2004-2007	Custom Control Language	C++	TCP/IP + ssh/scp	yes	no	explicit	seq, parmod
SBASCO	2004-2006	Custom Composition Language	C++	MPI	yes	yes	no	farm, pipe, multi-block
eSkel	2004-2005	C	C	MPI	no	?	no	pipeline, farm, deal, butterfly, hallowSwap
HDC	2000-2000	Haskell subset	C	MPI	yes	?	no	dcA, dcB, dcD, dcE, dcF, map, red, scan, filter
SKELib	2000-2000	C	C	MPI	no	no	no	farm, pipe
SkiPPER	1999-2002	CAML	C	SynDex	yes	limited	no	scm, df, tf, intermem
SkIE	1999-1999	GUI / Custom Control Lang	C++	MPI	yes	limited	no	pipe, farm, map, reduce, loop
Eden	1997-2008	Haskell extension	C	PVM/MPI	yes	?	no	map, mr, dc, pipe, iterUntil, torus, ring
P3L	1995-1998	Custom Control Lang.	C	MPI	yes	limited	no	map, reduce, scan, comp, pipe, farm, seq, loop
Skil	1995-1998	C subset	C		yes	no	no	pardata, map, fold
SCL	1993-1995	Custom Control Language	Fortran	CS Tools	yes	limited	no	map, scan, fold, farm, SPMD, iterateUntil

Table 2.1: Non-Object-Oriented Algorithmic Skeleton Frameworks

	Activity Years	Programming Language	Execution Language	Distribution Library	Type Safe	Skeleton Nesting	File Access	Skeleton Set
TBB	2006-2008	C++	C++	Threads	no	yes	n/a	for, reduce, scan, do, pipeline hashmap, vector, queue
QUAFF	2006-2007	C++	C	MPI	yes	yes	no	seq, pipe, farm, scm, pardo
JaSkel	2006-2007	Java	Java / AspectJ	Java MPP / Java RMI	no	yes	no	farm, pipeline, heartbeat
Muskel	2005-2008	Java	Java	Java RMI	no	yes	no	farm, pipe, seq, + custom MDF Graphs
HOC-SA	2004-2006	Java	Java	Globus, KOALA	no	no	no	farm, pipeline, wavefront
SkeTo	2003-2007	C++	C++	MPI	yes	no	no	list, matrix, tree
Mallba	2002-2007	C++	C++	NetStream / MPI	yes	no	no	exact, heuristic, hybrid
Muesli	2002-2005	C++	C++	MPI	partial	limited	no	array, matrix, farm, pipeline, parallel composition
Alt	2002-2003	Java / GworkflowDL	Java	Java RMI	yes	no	no	map, zip, reduction, scan, dh, replicate, apply, sort
(E)PAS	1999-2005	C++ extension	C++	MPI	no	yes	no	singleton, replication, compositional, pipeline, divideconquer, dataparallel
Lithium	1999-2004	Java	Java	Java RMI	no	yes	no	pipe, map, farm, reduce
CO_2P_3S	1999-2003	GUI / Java	Java generated	Threads / RMI / Sockets	partial	no	no	method-sequence, distributor, mesh, wavefront

Table 2.2: Object-Oriented Algorithmic Skeleton Frameworks

even graphical user interfaces. Inside the programming language, skeletons have been provided either as language constructs or libraries. Providing skeletons as language construct implies the development of a custom domain specific language and its compiler. This was clearly the stronger trend at the begging of skeleton research. The more recent trend is to provide skeletons as libraries, in particular with object oriented languages such as C++ and Java.

Execution Language is the language in which the skeleton applicatons are run or compiled. It was recognized very early that the programming languages (specially in the functional cases), were not efficient enough to execute the skeleton programs. Therefore, skeleton programming languages were simplified by executing skeleton application on other languages. Transformation processes were introduced to convert the skeleton applications (defined in the programming language) into an equivalent application on the target execution language. Different transformation processes were introduced, such as code generation or instantiation of lowerlevel skeletons (sometimes called operational skeletons) which where capable of interacting with a library in the execution language. The transformed application also gave the opportunity to introduce target architecture code, customized for performance, into the transformed application. Table 2.1 shows that a favorite for execution language has been the C language.

Distribution Library provides the functionality to achieve parallel/distributed computations. The big favorite in this sense has been MPI, which is not surprising since it integrates well with the `C` language, and is probably the most used tool for parallelism in cluster computing. The dangers of directly programming with the distribution library are, of course, safely hidden away from the programmers who never interact with the distribution library. Recently, the trend has been to develop skeleton frameworks capable of interacting with more than one distribution library. For example, CO_2P_3S can use Threads, RMI or Sockets; Mallba can use Netstream or MPI; or JaSkel which uses AspectJ to execute the skeleton applications on different skeleton frameworks.

Type Safety refers to the capability of detecting type incompatibility errors in skeleton program. Since the first skeleton frameworks were built on functional languages such as Haskell, type safety was simply inherited from the host language. Nevertheless, as custom languages where developed for skeleton programming, compilers had to be written to take type checking into consideration; which was not as difficult as skeleton nesting was not fully supported. Recenlty however, as we begun to host skeleton frameworks on object oriented languages with full nesting, the type safety issue has resurfaced. Unfortunately, type checking has been mostly overlooked (with the exception of QUAFF), and specially in Java based skeleton frameworks.

Skeleton Nesting is the capability of hierarchical composition of skeleton patterns. Skeleton Nesting was identified as an important feature in skeleton

programming from the very beginning, because it allows the composition of more complex patterns starting from a basic set of simpler patterns. Nevertheless, it has taken the community a long time to fully support arbitrary nesting of skeletons, mainly because of the scheduling and type verification difficulties. The trend is clear that recent skeleton frameworks support full nesting of skeletons.

File Access is the capability to access and manipulate files from an application. In the past, skeleton programing has proven useful mostly for computational intensive applications, where small amounts of data require big amounts of computation time. Nevertheless, many distributed applications require or produce large amounts of data during their computation. This is the case for astrophysics, particle physics, bioinformatics, etc. Thus, providing file transfer tools that integrate with skeleton programming is a key concern which has been completely overlooked with the exception of ASSIST.

Skeleton Set is the list of supported skeleton patterns. Skeleton sets vary greatly from one framework to the other, and more shocking, some skeletons with the same name have different semantics on different frameworks. The most common skeleton patterns in the literature are probably *farm*, *pipe*, and *map*.

2.3.2.1 Cole's Manifesto Principles

Let us recall the goals of Cole's four manifesto principles [Col04]:

Minimal Conceptual Disruption. "...we should build bridges to the de facto standards of the day, refining or constraining only where strictly necessary. We should respect the conceptual models of these standards, offering skeletons as enhancements rather than as competition...."

Integrate ad-hoc Parallelism. "...It is unrealistic to assume that skeletons can provide all the parallelism we need. We must construct our systems to allow the integration of skeletal and ad-hoc parallelism in a well defined way."

Accommodate Diversity."...We must be careful to draw a balance between our desire for abstract simplicity and the pragmatic need for flexibility...."

Show the pay-back. "...We must also be able to show that there are benefits which outweigh the initial overheads and that it is possible to experience these early on the learning curve... we should be able to show that skeletal programs can be ported to new architectures, with little or no amendment to the source..."

2.3.2.2 Characteristics vs Manifesto Principles

Table 2.3 shows how each of the skeleton framework characteristics can contribute to the goal of each principle. In detail, we believe that an ideal skeleton framework should resolve this concerns as follows:

Characteristics	Minimal conceptual disruption	Integrate ad-hoc parallelism	Accommodate diversity	Show the pay-back
Prog. Lang.	✔	✔		
Exec. Lang.				✔
Distrib. Lib.		✔		✔
Type Safety	✔			✔
Nesting			✔	
File Access	✔			✔
Skeleton Set	✔			✔

Table 2.3: Skeleton Frameworks Characteristics vs Manifesto Principles

Programming Language. Minimal conceptual disruption should be achieved by providing a library in an already existant and popular language. Integration with ad-hoc parallelism in other libraries can also be achieved at this level.

Execution Language: To show the pay-back, the execution language should be multi-platform and even better support heterogeneous systems.

Distribution Library: Integration with ad-hoc parallelism can also be achieved at the distribution library level. To show the pay-back, support for multiple distribution libraries is needed, capable of executing the skeleton application on different infrastructure and thus show the pay-back.

Type Safety: Programmers should not be forced to use `Object` or `void*` types, which goes against minimal conceptual disruption. Additionally, a type system for nestable skeletons increases programmers efficiency and thus shows the pay-back.

Skeleton Nesting: To accommodate diversity, support of arbitrary skeleton nesting must be available.

File Access: Must provide file data abstractions, preferably in a transparent way, to minimize the conceptual disruption; and to empower skeletons for data intensive applications. In addition, to show the pay-back, the file access must be optimized for each type of infrastructure where the application is run.

Skeleton Set: Must provide both task and data parallel skeleton patterns which are intuitive, minimal conceptual disruption, and at the same time capable of showing the pay-back.

To our knowledge, there is currently no algorithmic skeleton framework which addresses all of this characteristics in a satisfactory manner. Therefore, this thesis is dedicated to the design and implementation of such a skeleton framework.

2.4 Context: The ProActive Library

This sections provides a general background on ProActive. Its contents have been adapted from [Mor06] with the author's consent.

The *ProActive* library [CDdCL06, Pro] is a 100% Java middleware, which aims for seamless programming for concurrent, parallel, distributed, and mobile computing. Programming with ProActive does not require any modification of the standard Java execution environment, nor does it make use of a special compiler, pre-processor, or modified virtual machine.

2.4.1 Active Object Model

ProActive is based on the *active object* model. Active objects are a medium-grained entity with their own configurable activity.

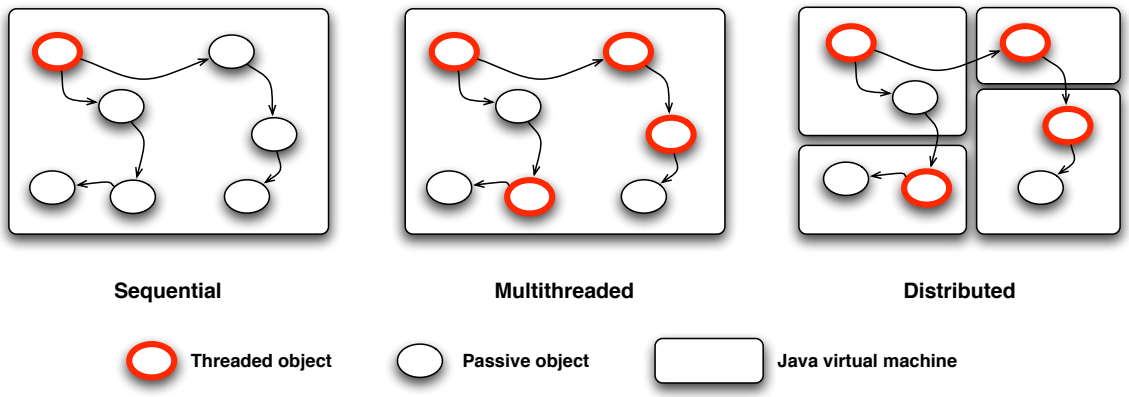


Figure 2.1: Seamless sequential to multithreaded to distributed objects

A distributed or concurrent application built with ProActive is composed of a number of active objects (Figure 2.1). Each active object has one distinguished element, the root, which is the only entry point to the active object. Each active object has its own thread of control and is granted the ability to decide in which order to serve the incoming method calls that are automatically stored in a queue of pending requests. Method calls sent to active objects are asynchronous with transparent future objects. Synchronization is data-based and handled by a mechanism known as wait-by-necessity [Car93]. A future is a placeholder for the result of an invocation on an active object. For the caller, futures are the result of the method invocation since they are transparently updated when the value of the invocation is actually computed. There is a short rendez-vous at the beginning of each asynchronous remote call, which blocks the caller until the call has reached the context of the callee, in order to ensure causal dependency.

The active object model of ProActive guaranties determinism properties and is formalized in the ASP (Asynchronous Sequential Processes) calculus [CHS04].

2.4.2 The ProActive library: principles, architecture and usages

The ProActive library provides an implementation of the active object model.

2.4.2.1 Implementation language

ProActive is developed in Java, a cross-platform language, and therefore ProActive application may run on any operating system proposing a compatible virtual machine. Moreover, ProActive uses only standard APIs and does not use any operating-system specific routine, other than to run daemons or to interact with legacy applications. There are no modifications to the JVM nor to the semantics of the Java language, and the bytecode of the application classes is never modified.

2.4.2.2 Implementation techniques

ProActive uses an extensible meta-object protocol architecture (MOP), which takes advantage of reflective techniques to abstract the distribution layer and offer features such as asynchronism or group communications.

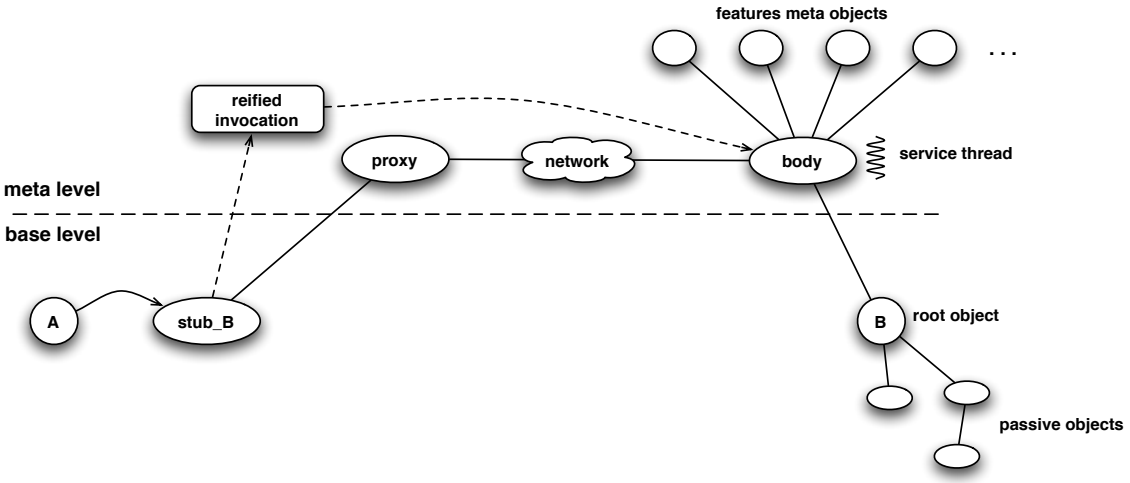


Figure 2.2: Meta-object architecture

The architecture of the MOP is presented in Figure 2.2. An active object is concretely built out of a root object (here of type B), with its graph of passive objects. A body object is attached to the root object, and this body references various features meta-objects, with different roles. An active object is always indirectly referenced through a proxy and a stub which is a sub-type of the root object. An invocation to the active object is actually an invocation on the stub object, which creates a reified representation of the invocation: the method called and the parameters. The reified representation is then given to a proxy object. The proxy transfers the reified invocation to the body, possibly through the network, and places the reified invocation in the request queue of the active object.

```
// instantiate active object of class B on node (a possibly
   remote location)
B b = (B) ProActive.newActive(''B'', new Object[] {
   aConstructorParameter}, node);

// use active object as any object of type B
Result r = b.foo();

...

// possible wait-by-necessity
System.out.println(r.printResult());
```

The request queue is one of the meta-objects referenced by the body. If the request will return a result, a future object is created and returned to the proxy, to the stub, then to the caller object.

The active object has it own activity thread, which is usually used to pick-up reified invocations from the request queue and serve them, i.e. execute them by reflection on the root object. Reification and interception of invocations, along with ProActive's customizable MOP architecture, provide both transparency and the ground for adaptation of non-functional features of active objects to fit various needs. It is possible to add custom meta-objects which may act upon the reified invocation, for instance to provide mobility features.

Active objects are instantiated using the ProActive API, by specifying the class of the root object, the instantiation parameters, and an optional location information:

Invoking the method `foo()` on `b` returns a future of type `Result`. The computation can then continue until a wait-by-necessity is reached. The thread accessing the future will be blocked only if the result is not yet available when it is actually required.

2.4.2.3 Semantics of communications

In ProActive, the distribution is transparent: invoking methods on remote objects does not require the developer to design remote objects with explicit remote mechanism (like `Remote` interfaces in Java RMI). Therefore, the developer can concentrate on the business logic as the distribution is automatically handled and transparent. Moreover, the ProActive library preserves polymorphism on remote objects (through the reference stub, which is a subclass of the remote root object).

Communications between active objects are realized through method invocations, which are reified and passed as messages. These messages are serializable Java objects which may be compared to TCP packets. Indeed, one part of the message contains routing information towards the different elements of the library, and the other part contains the data to be communicated to the called object.

The MOP used in ProActive has some limitations. Therefore, although all communications proceed through method invocations, the communication's semantics depends upon the method's signature, and the resulting invocation may not always be asynchronous.

Three cases are possible: synchronous invocation, one-way asynchronous invocation, and asynchronous invocation with future result.

- *Synchronous invocation*:
  - the method return a non reifiable object: primitive type or final class:
    `public boolean foo()`
  - the method declares throwing an exception:
    `public void bar() throws AnException`

  In this case, the caller thread is blocked until the reified invocation is processed and the eventual result (or Exception) is returned. It is fundamental to keep this case in mind, because some APIs define methods which throw exceptions or return non-reifiable results.
- *One-way asynchronous invocation*: the method does not throw any exception and does not return any result:
  `public void gee()`
  The invocation is asynchronous and the process flow of the caller continues once the reified invocation has been received by the active object (in other words, once the rendez-vous is finished).
- *Asynchronous invocation with future result*: the return type is a reifiable type, and the method does not throw any exception:
  `public MyReifiableType baz()`
  In this case, a future object is returned and the caller continues its execution flow. The active object will process the reified invocation according to its serving policy, and the future object will then be updated with the value of the result of the method execution.

If an invocation from an object A on an active object B triggers another invocation on another active object C, the future result received by A may be updated with another future object. In that case, when the result is available from C, the future of B is automatically updated, and the future object in A is also update with this result's value, through a mechanism called *automatic continuation* [CDHQ].

2.4.2.4 Library Features

As stated above, the MOP architecture of the ProActive library is flexible and configurable; it allows the addition of meta-objects for managing new required features. Moreover, the library also proposes a deployment framework, which allows the deployment of active objects on various infrastructures.

The features of the library are represented in Figure 2.3.

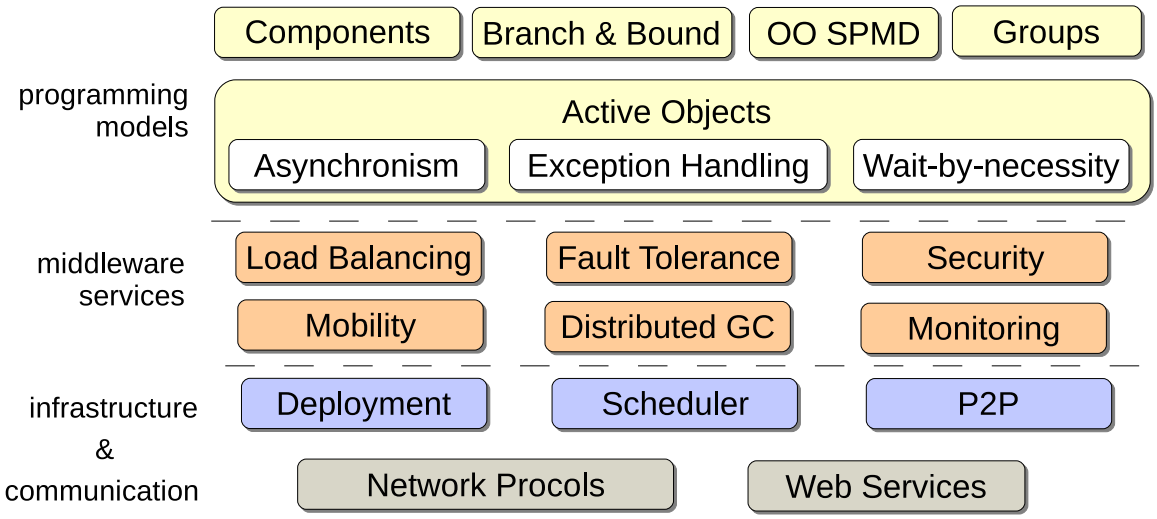


Figure 2.3: Layered features of the ProActive library

The active object model is formalized through the ASP calculus [CH05], and ProActive may be seen as an implementation of ASP. The library may be depicted in three layers: programming models, middleware features and infrastructure/communication facilities.

The programming model consists of the active objects model which offers asynchronous communications with wait-by-necessity [Car93] and exception management [CC05]; typed group communications [BBC02]; object-oriented SPMD [BBC05]; branch and bound [CdCBM07]; and a grid component model (GCM) [Cor06].

The middleware features include a transparent fault-tolerance mechanism based on a communication-induced checkpointing protocol [BCDH05], a security framework for communications between remote active objects [ACC05], migration capabilities for the mobility of active objects [BCHV00], and a mechanism for wrapping legacy code, notably as a way to control and interact with MPI applications.

The deployment layer includes: a deployment framework [BCM+02] which allows the creation of remote active objects on various infrastructures (detailed further in the next section); a peer-to-peer infrastructure [CdCM07] which allows the acquisition of idle desktop resources; and a scheduler/resource manager [RT+08] for dynamic scheduling of tasks and ProActive applications. Several network protocols are supported for the communication between active objects: Java RMI as the default protocol, HTTP, tunneled RMI. It is also possible to export active objects as web services, which can then be accessed using the standard SOAP protocol.

2.4.2.5 Large scale experiments and usages

Large scale experiments have been conducted with ProActive to validate many of its features: deployment on a large number of hosts from various organizations, on heterogeneous environments and using different communication, and

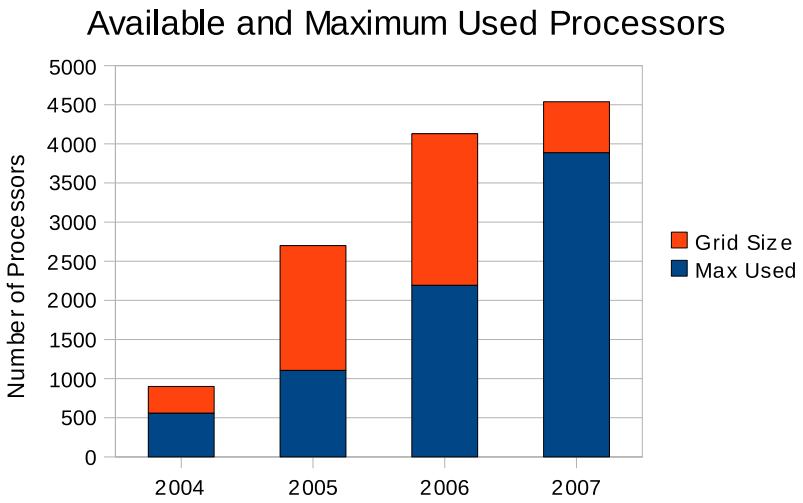


Figure 2.4: Grid PlugTests Evolution

network protocols, etc...

We outline two series of events which illustrate the capabilities of ProActive for large scale deployments.

First, the n-queens computational problem was solved for n=25 using a distributed peer-to-peer infrastructure on about 260 desktop machines in INRIA Sophia Antipolis, with an idle cycle stealing approach. The n-queens problem is a classical computational problem which consists of finding the placements of n non-attacking queens on a $n \times n$ chessboard. It is an NP-hard problem, which means that high values of n require years of computation time on a single machine, making this problem ideal in the context of challenge for distributed computing.

The peer-to-peer infrastructure was highly heterogeneous: Linux, Windows, various JVMs, Pentium II to Xeon bi-pro from 450 Mhz to 3.2 GHz, etc, and the total duration time was slightly over 6 months (4444h 54m 52s 854), starting October 8th until June 11th, using the spare CPU cycles of about 260 machines. The cumulative computing time was over 50 years.

Second, the Grid PlugTests events held at ETSI in 2004, 2005, 2006 and at China in 2007, have demonstrated the capacity of the ProActive library to create virtual organizations and to deploy applications on various clusters from various locations world wide. The first Grid PlugTests event (2004) [TE05b] gathered competing teams which had to solve the n-queens challenge by using the Grid built by coordinating universities and laboratories of 20 different sites in 12 different countries, resulting in 900 processors and a computing power of 100 Gigaflops (SciMark 2.0 benchmark for Java). In the second Grid PlugTests (2005) [TE05a] event, a second computational challenge was added: the permutation flow-shop, in which the deployment of an optimal schedule for N jobs on M machines must be computed. Heterogeneous resources from 40 sites in 13 countries were federated, resulting in a 450 Gigaflops grid, with a total of 2700 processors. In 2006 [TE06] and 2007 [Pro07] both challenges remained the

same and the number of total processors was 4130 and 4538 respectively. Nevertheless, more interesting than the Grid size is the evolution on the maximum number of simultaneous processors used by contestants to solve a problem: 560 in 2004, 1106 in 2005, 2193 in 2006 and 3888 in 2007. Figure 2.4 shows a summary, which clearly validates ProActive as a distributed computation library.

2.5 Conclusion

This chapter has provided a state of the art in algorithmic skeleton programming. In the first section we began by describing related programming models. Then we provided an outline of several algorithmic skeleton frameworks with a description of each. In the following section we continued by discussing and comparing the different skeleton frameworks, and conclude that there is no framework which currently satisfies our requirements. Then in the final section of the chapter we provided a brief description of the underlying middleware which will be used for implementations throughout this thesis: ProActive.

In the following chapter we provide a general description of our particular algorithmic skeleton programming model and its implementation: Calcium. Calcium was developed as part of this thesis and is used throughout the rest of the chapters.

Chapter 3

Algorithmic Skeletons in Calcium

Contents

3.1 An Algorithmic Skeleton Language 37
3.1.1 Muscle (Sequential) Blocks 37
3.1.2 The Skeleton Patterns 38
3.1.3 Skeleton Instructions 39
3.1.4 Instructions Reduction Rules 42
3.1.5 Summary 44
3.2 A Java Implementation 45
3.2.1 The Skeleton Library 45
3.2.2 Muscles 45
3.2.3 Instruction Stacks 47
3.2.4 Tasks 48
3.3 Execution Environments 50
3.3.1 General Principle 53
3.3.2 Multithreaded Environment 53
3.3.3 ProAcitve Environment 54
3.3.4 ProActive Scheduler Environment 55
3.4 Example: Find Primes 56
3.4.1 Data Exchange Types 56
3.4.2 Muscles 57
3.4.3 Skeleton Definition and Execution 59
3.5 Conclusion 60

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [Col91]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones.

To write an application, programmers must compose a skeleton pattern and fill the skeleton pattern with the sequential blocks specific to the application. The skeleton pattern implicitly defines the parallelization and distribution aspects, while the sequential blocks provide the application's functional aspects (i.e. business code). As a result, skeletons achieve a natural separation of parallelization and functional concerns.

Traditionally skeletons have been classified into two groups: task and data parallelism. For us, task parallel skeletons are those skeleton where each data input triggers only one parallel activity. Parallelism is achieved by processing several inputs simultaneously on the same skeleton. On the other hand data parallel skeletons, beside processing several inputs simultaneously, achieve parallelism by executing several parallel activies for a single data input.

The task and data parallelism classification was once relevant since the skeleton nesting in some frameworks was limited on the type of skeletons. For example, the two tier model on P3L and Skil (see 2.2.10 and 2.2.16). On recent frameworks the relevance is less evident because arbitrary nestings are supported.

As a skeleton framework we developed *Calcium*, which is greatly inspired by Lithium and its successor Muskel (see 2.2.8). Calcium is written in Java [Micc] and is provided as a library. To achieve distributed computation Calcium uses ProActive. ProActive is a Grid middleware [CDdCL06] providing, among others, a deployment framework [BCM+02], and a programming model based on active objects with transparent first class futures [Car93].

Hypotheses

We have adopted some hypotheses on the programming model with respect to other frameworks. Their goal is to simplify the programming model, avoid programming errors, and increase the parallelism degree/efficiency. We describe them here as to clearly outline the context of skeleton programming in Calcium.

Single input/output *Skeletons can only receive/produce scalar inputs/outputs*. Therefore the proposed programming model does not provide skeleton such as *reduce* or *split*. Instead, these parallels behaviors are embedded directly into higher level skeleton such as *map*, *d&c*, and *fork*. This hypothesis is relevant to simplify skeleton nesting and avoid programming errors. At the same time this hypothesis is not limiting, since inputs and outputs are objects which can encapsulate multiple data.

Passive Skeletons *Each skeleton output is directly related to a previously received input.* Therefore *heartbeat* like skeletons, where a skeleton can produce outputs without receiving an input, are not allowed. This hypothesis is relevant to simplify termination detection of an application.

Stateless Skeletons *Skeletons are stateless and therefore their sequential blocks (muscles) are also stateless*. Thus, the result produced for a parameter P_i passed to a skeleton is independent of previous parameter that have passed through the same skeleton P_j ($\forall j < i$). In other words a parameter P_i cannot communicate with the following parameters traversing a skeleton P_k

($\forall k > j$). This hypothesis is relevant to improve the parallelism degree and efficiency.

3.1 An Algorithmic Skeleton Language

In Calcium, skeletons are provided as a Java library. The library can nest task and data parallel skeleton in the following way:

$$\begin{aligned}\triangle ::= &\ \textbf{seq}(f_e) \mid \textbf{farm}(\triangle) \mid \textbf{pipe}(\triangle_1, \triangle_2) \mid \textbf{while}(f_b, \triangle) \mid \\ &\ \textbf{if}(f_b, \triangle_{true}, \triangle_{false}) \mid \textbf{for}(i, \triangle) \mid \textbf{map}(f_d, \triangle, f_c) \mid \\ &\ \textbf{fork}(f_d, \{\triangle_i\}, f_c) \mid \textbf{d\&c}(f_d, f_b, \triangle, f_c)\end{aligned}$$

Each skeleton represents a different pattern of parallel computation. All the communication details are implicit for each pattern, hidden away from the programmer.

3.1.1 Muscle (Sequential) Blocks

The nested skeleton pattern ($\triangle$) relies on sequential blocks of the application. These blocks provide the business logic and transform a general skeleton pattern into a specific application. We denominate these blocks *muscles*, as they provide the *real* (non-parallel) functionality of the application.

In Calcium, muscles come in four flavors:

Execution	$f_e : P \rightarrow R$
Division	$f_d : P \rightarrow \{R\}$
Conquer	$f_c : \{P\} \rightarrow R$
Condition	$f_b : P \rightarrow \text{boolean}$

Where P is the parameter type, R the result type, and $\{X\}$ a list of parameters or results of type X.

For the skeleton language, muscles are black boxes invoked during the computation of the skeleton program. Multiple muscles may be executed either sequentially or in parallel with respect to each other, in accordance with the defined $\triangle$. The result of a muscle is passed as a parameter to other muscle(s). When no further muscles need to be executed, the final result is delivered to the user.

Since algorithmic skeleton originated from functional programming, it is not surprising that muscles have been traditionally provided as functions. Functions are naturally stateless, unless they modify global variables. Stateless functions are very convenient for distributed computing as a higher degree of parallelism can be achieved and because the scheduling complexity decreases. In Calcium muscles are represented as objects which provide a function. Nevertheless, in accordance with the hypotheses defined at the beginning of this chapter, muscle objects cannot have mutable global variables.

3.1.2 The Skeleton Patterns

The skeleton patterns supported in Calcium are described as follows.

SEQ terminates recursive nesting of other skeletons, and as result does not really exploit parallelism. The *seq* skeleton wraps execution muscle which are then nested into the skeleton program as terminal leafs of the skeleton nesting tree.

FARM represents task replication and is also known as master-slave. Skeletons nested inside a farm are meant to be replicated for task parallelism. Following the Lithium approach, Calcium's reduction behavior is equivalent to specifying a farm skeleton to each sub-skeleton. In other words, the farm skeleton is automatically applied to all skeleton nestings. The redundant farm skeleton is provided as part of the library, since the reduction rules may change in future versions of the model.

PIPE is one of the most classical skeletons in the literature. The *pipe* skeleton represents staged computations, where parallelism can be achieved by computing different stages simultaneously on different inputs. The number of stages provided by *pipe* skeletons can be variable or fixed, but it is worth noting that fixed staged *pipe*s can be nested inside other fixed staged *pipe*s to create a *pipe* with any number of stages.

IF provides conditional branching. Two sub-skeletons are provided as parameters, along with a condition muscle. When an input arrives to the *if* skeleton, either one or the other sub-skeleton is executed, depending on the result of the condition muscle.

FOR as in standard programming languages the *for* skeleton represents fixed numbered iterations. The *for* skeleton receives a sub-skeleton and an integer as parameters. The semantics are simple, the sub-skeleton will be executed the number of times specified by the integer parameter. The result of one invocation of a sub-skeleton is passed as parameter to the following invocation of the sub-skeleton. Eventually, the result of the last sub-skeleton is provided as the *for* skeleton's result.

WHILE skeleton is analogous to the *for* skeleton, but instead of iterating a fixed number of times, a condition muscle decides wether the iteration must continue or stop. On each iteration, the result of the previous execution of the sub-skeleton is passed as input to the same sub-skeleton.

MAP is the most classical skeleton representing data parallelism. Its origin is closely related with lists and functional languages. The semantics behind *map* specify that a function (or sub-skeleton) can be applied simultaneously to all the elements of a list to achieve parallelism. The concept of data parallelism is reflected in the sense that a single data element can be splitted into multiple data, then the sub-skeleton is executed on each data element, and finally the results are united again into a single result. The *map* skeleton thus represents single instruction multiple data parallelism (SIMD).

FORK is not a classical skeleton, but introduced in this thesis. The *fork* skeleton behaves like *map*. The difference is that instead of applying the same function (sub-skeleton) to all elements of a list, a different function (sub-skeleton) is applied to each list element. Thus fork represent multiple instruction multiple data parallelism (MIMD).

D&C is another classical skeleton in the literature. To put it simply, *d&c* is a generalization of the *map* skeleton, where *map*s are recursively applied when a condition is met. The semantics of *d&c* are as follows. When an input arrives, a condition muscle is invoked on the input. Depending on the result two things can happen. Either the parameter is passed onto the sub-skeleton, or the input is divided with the custom divide muscle into a list of data. Then, for each list element the same process is applied recursively. When no further recursions are performed, the results obtained at each level are merged using a conquer muscle. Eventually, the merging of merged results yields one result which corresponds to the final result of the *d&c* skeleton. Thus, in our case, the depth of the recursion and width of the division are not fixed, but will depend on the muscles and data provided by the user.

Readers familiar with other skeleton frameworks may be surprised by the absence of some well known skeleton patterns, in particular *split* and *reduce*. Nevertheless, the parallelism provided by these patterns is indeed present, embedded directly into data parallel skeletons (*map*, *fork*, *d&c*). We have chosen this approach to comply with the *single output*/*input* hypothesis. Furthermore, by embedding these patterns into other skeletons we guarantee that whenever a *split* is executed, the symmetric *reduced* is performed later on; thus reducing programming errors.

Another family of skeletons which readers may miss are those which operate over arrays, such as *scan* and *stencil*. These skeleton are relevant in frameworks where arrays are used extensively. The relevance of arrays is less evident on object oriented languages, and as such we believe these kind of skeletons are not as crucial as is in other contexts.

3.1.3 Skeleton Instructions

There are many ways in which a skeleton pattern ($\triangle$) can be transformed into a parallel or distributed application. Some frameworks derive a process network (e.g. Eden, QUAFF), others use techniques such as macro data flow (e.g. Lithium, Muskel). We now describe the methodology used in Calcium.

Notation. The semantics in this section distinguish three types of reductions. The "$\Rightarrow$" arrow used for global reductions where the context of the execution (e.g. other parallel activities) is explicitly expressed; the "$\rightarrow$" arrow used for local reductions where the context of the program has been omitted for clarity; and the "$\twoheadrightarrow$" arrow used for transformations between languages (i.e. compiler).

3.1.3.1 The Instruction Language

For almost every skeleton there is an internal representation called an instruction. Instructions are in charge of executing parallel activities, consolidating results, and dynamically deciding the program's execution flow. The full list of instructions is as follows:

$$\begin{aligned} I ::= & \; seq_I(f) \mid while_I(f_b, S) \mid if_I(f_b, S_{true}, S_{false}) \mid for_I(i, S) \mid \\ & map_I(f_d, S, f_c) \mid fork_I(f_d, \{S\}, f_c) \mid d\&c_I(f_d, f_b, S, f_c) \mid \\ & id(f) \mid choice(p, S_{true}, S_{false}) \mid div_I(\{S\}, f_c) \mid conq_I(f_c) \end{aligned}$$

Besides the homonymous instructions four utility instructions are presented. The id which is the identity instruction, the $choice$ which operates in combination with if, and two instructions which harness data parallelism div_I, $conq_I$. The div_I instruction generates parallelism, while the $conq_I$ instruction merges the results once they are available.

The structure of the homonymous instructions is very similar to that of the corresponding skeletons, with the difference that instead of referencing subskeletons ($\triangle$) a stack is passed as parameter (S). A stack is defined as a sequence ($\cdot$) of instructions:

$$\begin{array}{l} \text{STACK-DEF} \\ S ::= I_1 \cdot \ldots \cdot I_n \end{array}$$

Several stacks can exist in parallel ($\|$), and we define this as:

$$\begin{array}{l} \text{CONTEX-DEF} \\ \Omega ::= p \cdot S \quad | \quad S(p) \quad | \quad r \quad | \quad \Omega \| \Omega \end{array}$$

Where p is a parameter, $p \cdot S$ is the stack S having received p, $S(p)$ is the stack S processing the parameter p, and r is the result. There are no fundamental differences between p and r, but for clarity we will use r to emphasize results.

Parallel activities are commutative, as the order in which stacks can be computed in parallel is irrelevant:

$$\begin{array}{l} \text{COMMUTATIVITY} \\ \Omega \| \Omega' \equiv \Omega' \| \Omega \end{array}$$

When a parameter is delivered it is processed by the stack as follows:

$$\begin{array}{l} \text{STACK-NEXT} \\ p \cdot S \| \Omega \quad \Rightarrow \quad S(p) \| \Omega \end{array}$$

Its application to the stack is equivalent as giving the same parameter to the first (or top most) instruction of the stack:

$$\begin{array}{l} \text{STACK-APP} \\ \dfrac{S = I_1 \cdot \ldots \cdot I_n}{S(p) = I_1(p) \cdot \ldots \cdot I_n} \end{array}$$

Finally, the computation of an instruction can yield a result and a new stack of instructions. If this is the case, then the old stack (S) is appended to the new stack (S'), and the intermediate result (p') is passed as parameter.

$$\text{INSTRUCTION-EVAL}\quad \frac{I(p) \rightarrow S'(p')}{I(p) \cdot S \| \Omega \quad \Rightarrow \quad (S' \cdot S)(p') \| \Omega}$$

3.1.3.2 From Skeletons to Instructions

The problem remains on how to transform the skeleton language constructs into their corresponding instructions. This is straightforward with the following transformation semantics, where $\triangle \twoheadrightarrow S$ denotes the transformation of the program $\triangle$ into the stack of instructions S:

$$\text{SEQ-TRANS}\quad seq(f) \twoheadrightarrow seq_I(f)$$

$$\text{FARM-TRANS}\quad \frac{\triangle \twoheadrightarrow S}{farm(\triangle) \twoheadrightarrow S}$$

$$\text{PIPE-TRANS}\quad \frac{\triangle_1 \twoheadrightarrow S_1 \quad \triangle_2 \twoheadrightarrow S_2}{pipe(\triangle_1, \triangle_2) \twoheadrightarrow S_1 \cdot S_2}$$

$$\text{WHILE-TRANS}\quad \frac{\triangle \twoheadrightarrow S}{while(f_b, \triangle) \twoheadrightarrow while_I(f_b, S)}$$

$$\text{IF-TRANS}\quad \frac{\triangle_{true} \twoheadrightarrow S_{true} \quad \triangle_{false} \twoheadrightarrow S_{false}}{if(f_b, \triangle_{true}, \triangle_{false}) \twoheadrightarrow if_I(f_b, S_{true}, S_{false})}$$

$$\text{FOR-TRANS}\quad \frac{\triangle \twoheadrightarrow S}{for(n, \triangle) \twoheadrightarrow for_I(n, S)}$$

$$\text{MAP-TRANS}\quad \frac{\triangle \twoheadrightarrow S}{map(f_d, \triangle, f_c) \twoheadrightarrow map_I(f_d, S, f_c)}$$

$$\text{FORK-TRANS}\quad \frac{\forall \triangle_i \in \{\triangle\} \quad \triangle_i \twoheadrightarrow S_i}{fork(f_d, \{\triangle\}, f_c) \twoheadrightarrow fork_I(f_d, \{S\}, f_c)}$$

$$\text{D\&C-TRANS}\quad \frac{\triangle \twoheadrightarrow S}{d\&c(f_b, f_d, \triangle, f_c) \twoheadrightarrow d\&c_I(f_b, f_d, S, f_c)}$$

These transformation semantics are very straightforward, except for the $farm$ and $pipe$ skeletons which do not require an instruction. The $farm$ and $pipe$ skeletons are directly transformed into their stack representations because their behaviors are known statically, as they do not depend on the results of muscle invocations.

On the other hand, the rest of the skeletons can invoke muscles (except for) and as such their behavior is only known at runtime. For example, consider the if skeleton which invokes a muscle f_b to decide between the execution of $\triangle_{true}$ and $\triangle_{false}$. Since f_b is a black box to the skeleton language, its result can only be known at runtime, and therefore during the transformation process we cannot know which sub-skeleton will be executed. Thus if requires an instruction to dynamically evaluate the flow of the program.

It is worth noticing that, as *farm* and *pipe*, the *for* skeleton can also be expanded statically without an instruction. Nevertheless, with a memory efficient implementation in mind (for large values of n), we prefer to have an instruction to unfold the *for* skeleton one step at a time.

A summary of the mappings between skeletons and instructions is shown in Table 3.1.

Skeleton	Instruction
$seq(f)$	$seq_I(f)$
$farm(\triangle)$	—
$pipe(\triangle_1, \triangle_2)$	—
$if(f_b, \triangle_{true}, \triangle_{false})$	$if_I(f_b, S_{true}, S_{false})$
	$choice(p, S_{true}, S_{false})$
$for(n, \triangle)$	$for_I(n, S)$
$while(f_b, \triangle)$	$while_I(f_b, S)$
$map(f_d, \triangle, f_c)$	$map_I(f_d, S, f_c)$
$fork(f_d, \{\triangle\}, f_c)$	$fork_I(f_d, \{S\}, f_c)$
$d\&c(f_b, f_d, \triangle, f_c)$	$d\&c_I(f_b, f_d, S, f_c)$
—	$id(f)$
—	$div_I(\{S\}, f_c)$
—	$conq_I(f_c)$

Table 3.1: Theory: From Skeletons to Instructions

3.1.4 Instructions Reduction Rules

This section begins by introducing semantics which allow non-atomic execution of the seq_I instruction. Then, seq_I is used as a basis to construct simpler reduction semantics for the rest of the instructions. Thus also rendering the other instructions non-atomic. An alternative approach, and much closer to the implementation, would have been to define non-atomic semantics for each instruction, but the result would have been a higher complexity of the semantics without an increase of expressiveness.

Finally this section presents semantics which express task and data parallelism. Task parallelism is straightforward given the stateless hypothesis defined in Section 3, while data parallelism is harnessed through the div_I and $conq_I$ instructions.

3.1.4.1 Non-atomic Seq Instruction

Non-atomicity is important to allow concurrent executions. For simplicity, the reduction rules presented in Figure 3.1 base all concurrent executions on the non-atomicity of seq_I as it is used to wrap the evaluation of muscles.

Therefore, for non-atomic execution of seq_I we define the following rule:

$$\text{NON-ATOMIC-SEQ-INST}\quad \frac{f(p) \rightarrow f'(p')}{seq_I(f)(p) \rightarrow seq_I(f')(p')} \qquad \text{RETURN-VALUE-SEQ-INST}\quad \frac{f(p) \rightarrow r}{seq_I(f)(p) \rightarrow r}$$

The **non-atomic-seq-inst** rule states that if muscles are non-atomic ($f(p) \rightarrow f'(p')$) the seq_I instruction is also non-atomic. In other words, if a function $f(p)$ can be evaluated to an intermediate state $f'(p')$, then the $seq_I(f)(p)$ instruction can also be evaluated to an intermediate state $seq_I(f')(p')$.

The **return-value-seq-inst** rule states that once a result is available it is delivered.

3.1.4.2 Reduction Rules

IF-INST

$$if_I(f_b, S_{\text{true}}, S_{\text{false}})(p) \rightarrow seq_I(f_b)(p) \cdot choice(p, S_{\text{true}}, S_{\text{false}})$$

ID-INST

$$id(p) \rightarrow p$$

CHOICE-INST-TRUE

$$choice_I(p, S_{\text{true}}, S_{\text{false}})(\textsf{true}) \rightarrow S_{\text{true}}(p)$$

CHOICE-INST-FALSE

$$choice_I(p, S_{\text{true}}, S_{\text{false}})(\textsf{false}) \rightarrow S_{\text{false}}(p)$$

WHILE-INST

$$while_I(f_b, S)(p) \rightarrow if_I(f_b, S \cdot while_I(f_b, S), id)(p)$$

FOR-INST-N

$$\frac{n > 0}{for_I(n, S)(p) \rightarrow S(p) \cdot for_I(n-1, S)}$$

FOR-INST-0

$$for_I(0, S)(p) \rightarrow p$$

MAP-INST

$$map_I(f_d, S, f_c)(p) \rightarrow \overbrace{seq_I(f_d)(p)}^{\text{lenght } k} \cdot div_I(\overbrace{[S, \ldots, S]}^{k \text{ times}}, f_c)$$

FORK-INST

$$fork_I(f_d, \{S\}, f_c)(p) \rightarrow seq_I(f_d)(p) \cdot div_I(\{S\}, f_c)$$

D&C-INST

$$\frac{I = d\&c_I(f_b, f_d, S, f_c)}{I(p) \rightarrow if_I(f_b, seq_I(f_d) \cdot div_I([I, \ldots, I], f_c), S)(p)}$$

Figure 3.1: Instructions Reduction Semantics

The rest of the reduction rules are defined in Figure 3.1. They are described as follows:

- The **if-inst** reduces if_I to a sequence of $seq_I \cdot choice$, where the seq_I executes the boolean condition and the $choice$ stores a copy of p.
- The **id-inst** rule is a convenient tool which delivers the given paremeter.
- The **choice-inst-true**, and **choice-inst-false** rules evaluate to either S_{true} or S_{false} depending on the value of their parameter. In each cases the stored p is passed as parameter to S_{true} or S_{false}.
- The **while-inst** rule reduces the $while_I$ into an if_I which decides (using f_b) between the iterative execution of S or the end of the iteration (id).
- The **for-inst-n** rule states that for $n > 0$ the for_I instruction is reduced to a sequence of S and itself. On the other hand, the **for-inst-0** rule ends the for_I iteration when $n = 0$ and reduces the instruction to p.

- The **map-inst** rule states that the map_I instruction is reduced to a sequence of seq_I and div_I. The seq_I will perform the division of p (using f_d) into k parameters. Then, div_I will take the splitted parameters and spread them over k copies of S.
- The **fork-inst** rule is analogous to the map-inst rule but instead of copying S, the given list of $\{S\}$ is used.
- The **d&c-inst** rule reduces the $d\&c_I$ into an if_I. The if_I will choose using f_b to either divide the parameter p and, for each element, evaluate a recursive execution of the original $d\&c_I$ instruction (I); or directly reduce to S (terminal case of the recursion).

3.1.4.3 Task and Data Parallelism Rules

If a skeleton receives multiple parameters, and the instructions of a stack are stateless, then they can be parallelized as follows:

$$\text{TASK-}\| \quad \frac{\forall I_i \in S \qquad \text{stateless}(I_i)}{S(p_1, \ldots, p_m) \rightarrow S(p_1) \| \ldots \| \cdot S(p_m)}$$

This reduction is what we call task parallelism. The stack S is copied m times, and each parameter is applied to one of the copies.

On the other hand, data parallelism is expressed with the div_I and $conq_I$ instructions. The div_I instruction is reduced as follows:

$$\text{DATA-}\| \quad \frac{\{S\} = \{S_1, .., S_n\}}{div_I(\{S\}, f_c)([p_1, ..., p_n]) \rightarrow conq_I(f_c)(S_1(p_1) \| \ldots \| S_n(p_n))}$$

The list of parameters ($[p_1, ..., p_n]$) is spread over the list of stacks ($\{S\}$) and applied as $S_i(p_i)$. Data parallelism is achieved since each stack application can be computed in parallel with the others.

The progress of the parallel activities is reflected in the *conq-inst-progress* rule. When the evaluation of the parallel activities is concluded, they are passed as parameters to the $conq_I$ instruction and reduced by the *conq-inst-reduce* rule:

$$\text{CONQ-INST-PROGRESS} \quad \frac{\Omega \rightarrow \Omega'}{conq_I(f_c)(\Omega) \rightarrow conq_I(f_c)(\Omega')}$$

$$\text{CONQ-INST-REDUCE} \quad conq_I(f_c)(r_1 \| \ldots \| r_n) \rightarrow seq(f_c)([r_1, ..., r_n])$$

3.1.5 Summary

This section has provided formal semantics which express the parallel evaluation of a skeleton program. The skeleton programs was transformed into an instruction level representation where reductions semantics where presented capable of exploiting the parallelism offered by the different skeleton frameworks.

The following section takes a more practical approach by describing an implementation of the theoretical concepts developed in this section.

3.2 A Java Implementation

3.2.1 The Skeleton Library

The language constructs are provided to programmers as a set of classes. Each skeleton has a corresponding class which is part of Calcium's public API. Unlike other Java skeleton frameworks (e.g. JaSkel), skeletons are not specialized by inheritance but using parametric constructors. Skeleton classes have no embedded distributed behavior. Their sole objective is to allow type safe composition of the skeleton pattern by holding references to sub-skeletons and muscle objects. Skeleton instances are immutable and cannot be modified by the programmer once instantiated. In Calcium, programmers never need to know or worry about the implementation of a skeleton class.

There are several advantages of this approach. First, that muscles and skeletons can be shared. For example, it is possible to have something like

$$\triangle = \text{pipe}(\text{pipe}(\text{seq}(f_1), \text{seq}(f_2)), \text{seq}(f_1)))$$

where the same instance of f_1 can be present inside two skeletons. The second is that parallel/distribution semantics are added later on, and therefore can be customized without having to change the skeleton's signature or generate new code. This is akin with Eden's implementation skeletons, P3L's templates, or SKiPPER's operational skeletons.

Listing 3.1 shows an example for the map skeleton class. The first thing to notice is that `Map` implements the `Skeleton` interface and is generic with respect to `P` and `R` (the input and ouput types of the skeleton). The global variables are two muscle objects and a skeleton object. The muscle objects correspond to `Divide` and `Conquer` used for the division and reduction respectively. The skeleton object corresponds to the nested skeleton which is executed after performing the division.

As shown in the listing, multiple constructors can be used to instantiate skeletons. Some may provide default parameters, or like in the `Map` case, they can simplify the burden of using `Seq` skeleton to terminate the nesting recursion. The objective of the constructors is to instantiate the global variables, and type check the parameters[1]. Finally, an `accept(...)` method is provided to navigate the skeleton nesting with the *visitor pattern* [GHJV95] and generate the corresponding instruction stack.

3.2.2 Muscles

Muscles are the pieces of code which provide the functional (business) behavior to a skeleton (see 3.1.1). In Calcium muscles are identified by Java interfaces.

[1]Type safety is discussed in detail in Chapter 5.

```
public class Map<P, R > implements Skeleton<P, R> {

  Divide<P, ?> div;
  Skeleton<?, ?> subskelton;
  Conquer<?, R> conq;

  public <X,Y> Map(Divide<P,X> div, Skeleton<X,Y> nested,
     Conquer<Y,R> conq) {
    this.div = div;
    this.subskelton = nested;
    this.conq = conq;
  }

  public <X,Y> Map(Divide<P, X> div, Execute<X, Y> muscle,
     Conquer<Y, R> conq) {
    this(div, new Seq<X,Y>(muscle), conq);
  }

  void accept(SkeletonVisitor visitor) {
    visitor.visit(this);
  }
}
```

Listing 3.1: The `Map` skeleton in Calcium

```
public interface Condition<P> extends Muscle<P, Boolean> {
  public boolean condition(P param, SkeletonSystem system)
    throws Exception;
}

public interface Conquer<Y, R> extends Muscle<Y[], R> {
  public R conquer(Y[] param, SkeletonSystem system)
    throws Exception;
}

public interface Divide<P, X> extends Muscle<P, X[]> {
  public X[] divide(P param, SkeletonSystem system)
    throws Exception;
}

public interface Execute<P, R> extends Muscle<P, R> {
  public R execute(P param, SkeletonSystem system)
    throws Exception;
}
```

Listing 3.2: Muscles in Calcium

There are four interfaces, each corresponding to a muscle flavor, as depicted on Listing 3.2.

All of the interfaces inherit from the general abstract interface `Muscle`. Each interface requires the programmer to implement a specific method. The method uses generics to correctly type its input and output. Therefore, no casts are imposed by the skeleton language inside the muscle's method (see Chapter 5).

As can be seen in the listing, muscles can throw `Exception`s. The exception treatment is very basic. If an exception is raised by any muscle of the skeleton program, the rest of the program's computation (running in parallel) is stopped, and the exception is shipped back to the user.

Muscles receive two parameters. The first is the default parameter of the method, which will correspond to the result of some other muscle previously computed. The second parameter is an abstraction to local dependent information. For example, the current working directory. This second parameter is relevant for the content of Chapter 6, but may be omitted in some examples throughout this document for the sake of simplicity.

For an example of a muscle implementation refer to Section 3.4.

3.2.3 Instruction Stacks

Instructions are lower level representations that exploit the parallel behavior of each skeleton pattern (see 3.1.3). Instructions are never seen by programmers. Table 3.2 provides a mapping of the Skeleton API with the instructions. The table is analogous to Table 3.1 but at the implementation level. It is worth noticing that the instructions *id* and *choice* are not present as their semantics are directly embedded into the implementations of other instructions.

Skeleton	**Instruction**
`Seq`	`SeqInst`
`Farm`	—
`Pipe`	—
`If`	`IfInst`
`For`	`ForInst`
`While`	`WhileInst`
`Map`	`MapInst`
`DaC`	`DacInst`
`Fork`	`ForkInst`
—	`DivideInst`
—	`ConquerInst`

Table 3.2: Practice: From Skeletons to Instructions

As noted in section 3.1.3.2, `Farm` and `Pipe` skeletons do not require associated instructions. On the other hand, there are some instructions which do not have an associated skeleton. This is the case with `ConquerInst` and `DivideInst`. The parallel behavior provided by these instructions is accessible through the data parallel skeletons (`Map`, `DaC`, `Fork`).

Using the visitor pattern, the skeleton nesting tree is transformed into an instruction stack. Interpreters, available on the computation nodes, can consume instructions from the top of the stack and perhaps generate new stacks in accordance with the formal semantics introduced section 3.1.3.

Figure 3.2 shows a graphical representation of the formal semantics' implementation. With optimization and simplification in mind, the implemented semantics are somewhat different from the formal ones. The first difference is that the seq_I instruction wrapper is not used as extensively because invocations of muscles f are made directly. The second is that the if_I instruction is not used by $while_I$ nor $d\&c_I$ since its behavior is also directly implemented by them. A side effect of this is that id is no longer required. Fortunately, the differences between the theory and implementation are not a concern, as the overall outcome of the application remains the same.

At any given point, an interpreter will have a single instruction stack. To perform the computation, the interpreter will pop the uppermost instruction from the stack and execute the instruction with the semantics shown in Figure 3.2. The interpretation can yield a modification of the stack (e.g. if_I) and/or the modification of P (e.g. seq_I). When there are no further instructions in the stack the final state of the parameter P (also noted as R) is delivered as the result of the execution.

3.2.4 Tasks

Internally in Calcium, a task abstraction is used to distribute and keep track of a program's execution. As shown in Figure 3.4(a), a task is mainly composed of an instruction stack and references to sub-tasks. Nevertheless, tasks can also act as placeholders for information such as performance metrics, exceptions, file dependencies, etc.

Sub-tasks are created when data parallelism is encountered. A sub-task can also spawn its own sub-tasks, and thus a task tree as the one shown in Figure 3.4(b) can be generated. The root of the tree corresponds to a task inputted into the framework by the user, while the rest of the nodes represent dynamically generated tasks. In the task tree, leaf nodes are tasks ready for execution or being processed, while the inner nodes represent tasks that are waiting for their sub-tasks to finish.

When a task has no unfinished sub-tasks and the instruction stack is empty, then the task is finished. When all brother tasks are finished they are returned to their parent task for reduction. The parent may then continue with the execution of its own instruction tasks and perhaps generate new sub-tasks later on.

As shown in Figure 3.4, a task can be in either *ready*, *processing*, *waiting* or

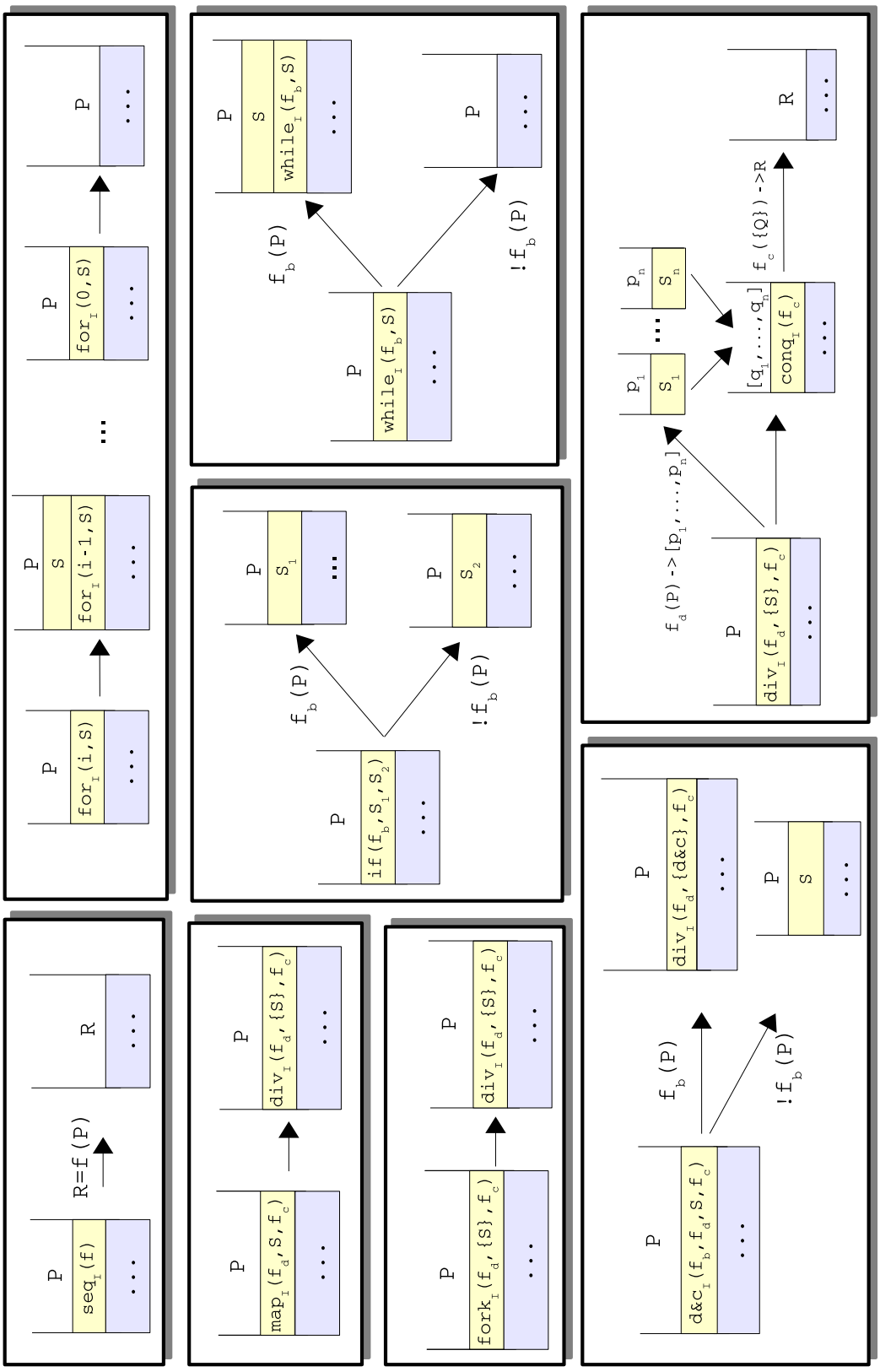
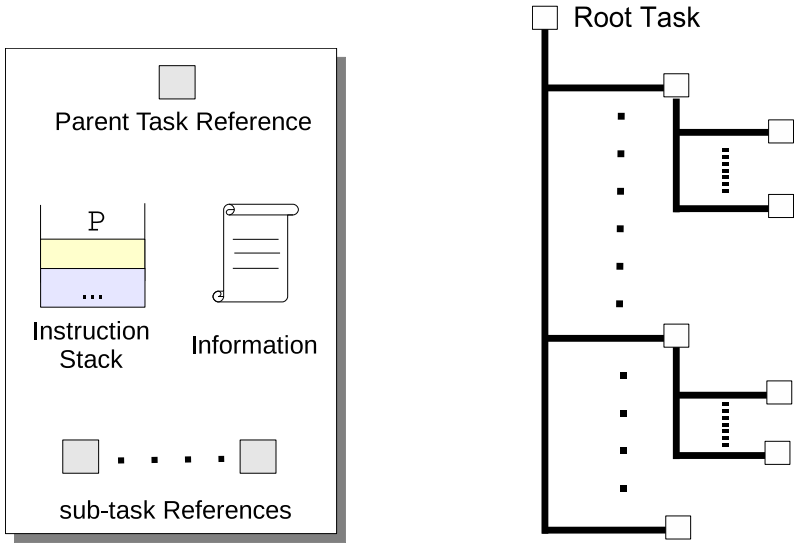


Figure 3.2: Stack Interpretation Semantics



(a) Task Definition (b) Task Tree

Figure 3.3: Task Abstraction

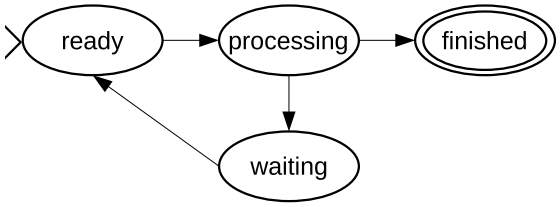


Figure 3.4: Task State Diagram

finished state. A task in the ready state is ready for execution, but has not yet been assigned to a resource. A task in the processing state has been assigned to a resource for computation. A task in the waiting state is waiting for some event, e.g. the completion of sub-tasks. The finish state is reached when a task has no more instructions to compute (which implies that it has no sub-tasks). If it is a finished sub-task then it is delivered back to the waiting parent task. On the other hand, when a root task reaches the finished state, its result is delivered to the user.

3.3 Execution Environments

Calcium is designed to support the execution of a skeleton program with different environments, each targetted for a specific infrastructure. Currently three environments are supported: Multithread, ProActive, and ProActive Scheduler. A user can run her algorithmic skeleton application on a different environment without modifying the skeleton program.

The motivation for different execution environments lies in the fact that each

type of infrastructures has different requirements and advantages. Tailored environments are capable of addressing the specific requirements and advantages of each type of infrastructure.

The three type of infrastructures we consider are the following ones.

SMP stands for Symmetric Multi-Processing, which is a multiprocessor computer architecture where two or more uniform processors can connect to a single shared memory. While other architectures exist for shared memory processors, we will refer to all shared memory architectures as SMP. In the past, SMP like infrastructures with many processing elements were the privilege of custom made super computers. Now a days, with the advent of multi-core programming, it is natural to have personal computers with 2 or 4 cores; with chips soon featuring from tens to hundreds of cores (eg: Nvida GeForce 200, 10 cores; Intel Polaris, 80 cores; Plurality Hal, 16-256 cores; etc) [Wik]. Furthermore, we can envision many-core personal computers which will soon have not hundreds, but possibly thousands of cores on a single chip [Bor07]. In other words, SMP has finally arrived for the masses, which justifies having a specific execution environment for SMP like infrastructures.

Clusters are multi-computer architectures which are interconnected through a network for parallel computations. Cluster architectures have, for a time now, become the dominant architectures on the Top 500 ranking [top] by displacing custom made super computers with their lower costs. Of the top 500 ranking machines in 2008, 80% correspond to Clusters, and 98% of the machines have more than 1024 processors.

Grids are composed of interconnected cluster sites and loosely-couple nodes to act like a virtual super computer. The advantages of Grids are that they can be formed and dissolved as needed, and can provide a huge amount of resources. On the other hand, their complexity makes them difficult to exploit, and are mostly suited for loosly coupled applications. A few examples of existing Grid infrastructures are Grid5000 (France), NGS (UK), INFN Grid (Italy), TeraGrid (US), NorduGrid (International), and OurGrid (International).

Each one of these types of infrastructures is characterized by a set of properties. These properties correspond to requirements that the infrastructure enforces on applications. In other words, the infrastructure supposes that applications are capable of handling these requirements.

Scalability Take advantage of new resources when the number of processing elements increases. The exact type of what corresponds to a processing element is different for each infrastructure but can be, for example, something like: cores for SMP, nodes for Clusters, and sites for Grids. As the number of processing elements increases for an infrastructure, the application must cope and take advantage of the increased number of resources, up to the applications maximum parallelism degree (Amdahl's Law).

Fault Tolerance Support system faults. An application must be able to cope and continue its execution when a fault of the infrastructure takes place. What exactly constitutes a fault differs for each infrastructure, but examples are: a processing element which crashes while computing part of the application, or a node which may become disconnected from the network.

Deployment The capability of acquiring the processing elements; and then performing environment setup before the actual computation takes place, such as: code shipping and logic instantiation (objects, components, etc).

Distribution The capability of handling processing elements which are remote with respect to each other. In other words, processing elements do not share memory and must communicate by some other means such as message passing, channels, etc. The application must therefore handle issues such as deadlocks or latencies which arise from a remote communication model.

Heterogeneity Support of non-uniform resources. The application must cope with issues such as processing elements with different architecture, nodes having different operating systems, different libraries or protocols.

Multiple Administrative Domains The capability of coordinating divers resource policies. Applications must be aware that parts of the infrastructure may be maintained and operated by different organizations, each with its own policies. For example, some resources may be available for students only during the night.

Dynamicity The capability of adapting to infrastructure changes at runtime. Applications must cope with runtime evolutions of the infrastructure. For example, resource may be added or removed from the infrastructure as the application is being executed.

Characteristics	SMP	Cluster	Grid
Scalability	✓	✓	✓
Fault Tolerance	✗ → ✓	✓	✓
Deployment	✗	✓	✓
Distribution	✗	✓	✓
Heterogeneity	✗	✗	✓
Mult. Adm. Domain	✗	✗	✓
Dynamicity	✗	✗	✓

Table 3.3: Infrastructure Characteristics

Table 3.3 provides a summary of which requirements are made by each infrastructure: SMP, Cluster, and Grids. We can see that SMP is by far the most relaxed infrastructure and as such, an application running on SMP does not

have to worry about deployment, distribution, heterogeneity, multiple administrative domains, and dynamicity. On the other hand, an application running on an SMP infrastructure must be capable of handling scalability and fault tolerance as the number of processing elements becomes very large. In contrast, Cluster infrastructures have to additionally consider deployment and distribution. Finally, Grids represent the most restrictive scenario where heterogeneity, dynamicity and multiple administrative domains are added to the burden of the application.

We believe that programmers should not have to change their application's code when running it on a differnet target infrastructure. The simplest, yet inefficient solution is to program on the most restrictive scenario, where the application has to handle everything. The drawback with this approach is that performance opportunities offered by each infrastructure are not considered. For example, fast communication via shared memory in SMP vs slow network communication.

The standard way to achieve infrastructure abstraction is by providing a middleware layer between the application and the infrastructure. The middleware is thus in charge of providing a uniform access to the application, and fullfilling the requirements of each infrastructure. Environments in Calcium are aligned with this idea. Each environments in Calcium represents an interface of the skeleton framework with a middleware layer. The environment specifies how the middleware is plugged into the skeleton framework.

3.3.1 General Principle

From a general perspective, parallelism or distribution of an application in Calicum is a producer/consumer problem, where the shared buffer is a task pool and the produced/consumed data are tasks. Nevertheless, since tasks have states, it is more precise to identify three task pools: ready, finished and waiting; as depicted in Figure 3.5.

A ready pool stores ready tasks. Root tasks are entered into the ready pool by users, who provide the initial parameter and the skeleton program.

Interpreters on the computation nodes consume tasks from the ready pool and compute tasks according to their skeleton instruction stack. When the interpreters cannot compute a task any further the task is either in the finished or in the waiting state. If the task is in the finished state it is delivered to the finished pool. If the task is in the waiting state, then it has generated new sub-tasks (which represent data parallelism map, $fork$, $d\&c$ skeletons). The sub-tasks are delivered to the ready pool while the parent task is stored in the waiting pool.

The execution of a skeleton program is thus reduced to the problem of scheduling tasks in accordance with the producer/consumer problem.

3.3.2 Multithreaded Environment

The simplest way to compute an application with Calicum is using threads. Both *threads* and *processes* are the standard way of exploiting SMP like infrastruc-

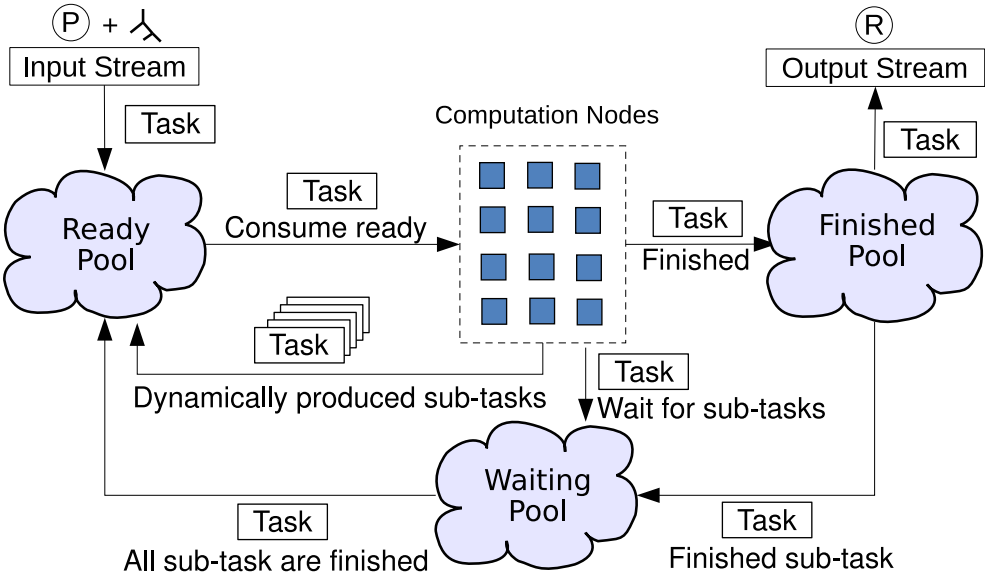


Figure 3.5: Calcium Framework Task Flow

tures. In particular threads are lighter and oriented towards shared memory programming, while processes are more expensive and use inter-process communication. We have chosen threads for the previous reasons, and in particular because our Multithreaded Environment is not fault tolerant. Nevertheless, as the number of processing elements in a machine increase, along with the probability of failure, processes may become a more attractive alternative.

To use this environment, the only parameter that must be specified is the maximum number of threads.

```
Environment env = new MultiThreadedEnvironment(10);
Calcium calcium = new Calcium(env);
```

Internally, a thread pool will be used with up to the specified maximum threads. The computation of the application is performed in parallel on the same machine. Tasks are consumed from the task-pool and assigned to interpreters running on different threads. This environment is ideal for preliminary tests and debugging of an application before using a distributed environment.

3.3.3 ProAcitve Environment

For distributed computing an environment based on ProActive active objects and deployment descriptors is available. The ProActive middleware provides *fault tolerance*, *distribution* via remote objects, and *deployment* via descriptors; thus making it ideal for Cluster infrastructures.

The ProActive Environment is instantiated by specifying a ProActive deployment descriptor. The deployment descriptor contains a description of the resources which will be used during the computation, and a `virtualnode-name` which corresponds to the entry point to the resource description.

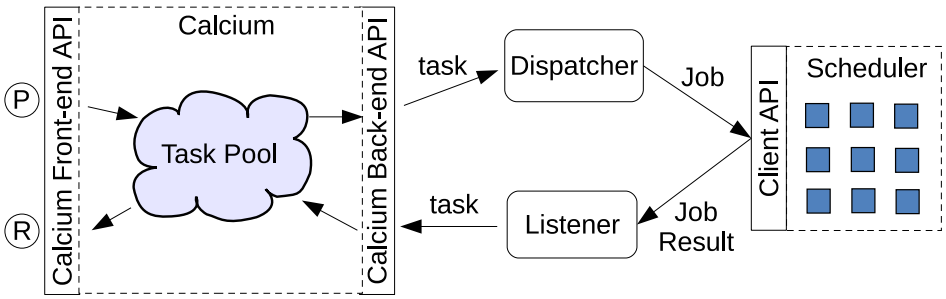


Figure 3.6: Calcium interfaced with ProActive Scheduler

```
String desc = "/home/user/descriptor.xml";
Environment env = new ProActiveEnv(desc, "virtualnode-name");
Calcium calcium = new Calcium(env);
```

Once the resources specified in the deployment descriptor are acquired, a remotely accessible active object is instantiated on each node. The active object consists of an interpreter capable of computing instructions in a task. Interpreters are fed with tasks from the task-pool and their results are collected and stored back into the task-pool.

This environment is suitable for a controlled environment, such as a Cluster infrastructures, where resources can be described in advanced via deployment descriptors.

3.3.4 ProActive Scheduler Environment

The scheduler environment targets the execution of skeleton applications on more complex infrastructures. As the ProActive Environment, the ProActive Scheduler Environment provides support for *fault tolerance*, *distribution*, and *deployment*. But in addition also supports *heterogeneity* of resources, as they can be composed of desktop machines, shared clusters, etc; *dynamicity* by dynamically acquiring and releasing resources; and *multiple administrative domains* as resource may be harnessed from different organizations.

The usage of the ProActive scheduler environment is straightforward. A scheduler front-end `host` name is required with a `user` name and a `passwd`.

```
Environment env = new PAScheduler("host","user","passwd");
Calcium calcium = new Calcium(env);
```

The integration between Calcium and the ProActive scheduler is shown in Figure 3.6. A *Dispatcher* module, links the task-pool with the scheduler interface. To enter the scheduler, Calcium tasks are wrapped in a `Job` abstraction which can be submitted to the scheduler. A *Listener* module is in charge of waiting for finished `Job` events from the scheduler. When a `Job` is finished, the *Listener* modules retrieves the `Job Result` and returns the result to the Calcium task-pool. A `Job Result` can contain finished tasks or new sub-tasks (which where dynamically generated). Using the Calcium back-end API, the *Listener*

module stores the finished and new sub-tasks into the Calcium task-pool. Resources management, scheduling and fault tolerance are handled directly by the scheduler.

3.4 Example: Find Primes

This section illustrates the programming of a simple application with Calcium. The application corresponds to a naive search of prime numbers on a given interval. We have chosen this problem as an example because of its algorithmic simplicity.

Using algorithmic skeletons, a parallel application for searching prime numbers can be written in less than a 100 lines of code. This is achieved with the $d\&c$ skeleton. An interval is split into smaller intervals until a threshold is reached. Then the sub-intervals are searched (in parallel) with a naive approach, and the resulting primes are then merged back and delivered to the user.

3.4.1 Data Exchange Types

We begin by identifying two data types which are passed between the different muscles. These are named `Interval` and `Primes`.

The `Interval` class holds the minimum (`min`) and maximum (`max`) bounds for the interval, and a variable which indicates the threshold of a given problem (`threshold`). An initial `Interval` object triggers the computation, and new `Interval` objects can be dynamically created during the evaluation of the $d\&c$ skeleton.

```
class Interval implements Serializable {
   public int min;
   public int max;
   public int threshold;

   public Interval(int min, int max, int threshold) {
      this.min = min;
      this.max = max;
      this.threshold = threshold;
   }
}
```

The `Prime` class is holds the output of the computation: a list of prime numbers. It contains a `list` of integers with the prime numbers, a method to `add` a single number, a method to `add` a list of primes, and a method to `sort` the internal list.

```
class Primes implements Serializable {
   public Vector<Integer> list;

   public Primes() {
     list = new Vector<Integer>();
```

```
    }

    public add(int i){ list.add(new Integer(i)); }

    public add(Primes p){ list.addAll(p.list); }

    public void sort(){ Collections.sort(list); }
}
```

Both classes, `Interval` and `Primes`, must implement the `Serializable` interface, as they will be transferred through the network to the computation nodes. Failure to do so results on a compilation error.

3.4.2 Muscles

Now the programmer must write the functional (business) code of the application. For $d\&c$ this requires four muscles: f_b, f_d, f_e, f_c, which conceptually correspond to `Condition`, `Divide`, `Execute` and `Conquer` types.

Condition (f_b)

The f_b type muscles are identified with the `Condition` interface. In this example, this class corresponds to `IntervalDivideCondition` which evaluates if an `Interval` should be subdivided or not. Since we are using a naive approach the decision is really simple, the function returns `true` if the size of the interval is above a given threshold and `false` otherwise.

```
class IntervalDivideCondition implements Condition<Interval> {
    public boolean condition(Interval param) {
        return (param.max - param.min) > param.threshold;
    }
}
```

Divide (f_d)

Then there are f_d muscles which are identified with the `Divide` interface. In the example, the class `IntervalDivide` implements the `Divide` interface and provides the functional code on how an `Interval` is divided.

```
class IntervalDivide implements Divide<Interval, Interval> {
  public Interval[] divide(Interval p) {

    int middle = p.min + ((p.max - p.min) / 2);

    Interval top = new Interval(middle + 1, p.max, p.threshold);
    Interval bottom = new Interval(p.min, middle, p.threshold);

    return new Interval[]{bottom, top};
  }
}
```

In the naive algorithm we simply divide the interval into two parts: top and bottom. We begin by finding the middle of the interval. Then we create two knew intervals, the `top` which goes from `middle+1` to the `maximum`, and the `bottom` interval which goes from the minimum to the `middle`. Once we have the new sub-intervals we put them into a list and return it as the result.

Execute (f_e)

The f_e muscles are identified with the `Execute` interface. In our example, the actual prime searching is done in the `SearchInterval` class which implements the `Execute` interface.

```
class SearchInterval implements Execute<Interval, Primes> {

   public Primes execute(Interval p) {
      Primes primes = new Primes();

      for (int i = p.min; i <= p.max; i++) {
         if (isPrime(i)) {
            primes.add(i);
         }
      }

      return primes;
   }

   boolean isPrime(int i){...}
```

Again, the search algorithm is very naive. For a given interval `p`, it iterates over every element of the interval and checks wether the element is a prime or not. If an element is found to be prime, then it is added to a list of `primes`. Once the iteration is over the list of `primes` is returned.

Conquer (f_c)

The final type of muscles corresponds to f_c which are identified with the `Conquer` interface. In our example, the `MergePrimes` class implements the `Conquer` interface.

```
class MergePrimes implements Conquer<Primes, Primes> {
   public Primes conquer(Primes[] p) {
      Primes conquered = new Primes();

      for (Primes primes : p) {
         conquered.add(primes);
      }

      conquered.sort();
      return conquered;
   }
```

```
1  Environment env = new MultiThreadedEnvironment(10);
   Calcium calcium = new Calcium(env);
3
   calcium.boot();
5
   Stream<Interval, Primes> stream = calcium.newStream(findprimes);
7
   Future<Primes> f1 = stream.input(new Interval(1, 6400, 300)));
9  Future<Primes> f2 = stream.input(new Interval(1, 100, 20)));
   Future<Primes> f3 = stream.input(new Interval(1, 640, 64)));
11
   //do something here ...
13
   Primes result1 = f1.get();
15 Primes result2 = f2.get();
   Primes result3 = f3.get();
17
   calcium.shutdown();
```

Listing 3.3: Calcium Usage Example

```
}
```

The objective of `MergePrimes` is to aggregate several list of primes into a single one. First a new container is created (`conquered`), then the aggregation is performed by iterating over all the lists. Finally, the `conquered` list of primes is sorted.

3.4.3 Skeleton Definition and Execution

Once all the muscles have been defined with the application's specific code, programmers can proceed with the definition of the skeleton program (△).

```
Skeleton<Interval,Primes> findprimes =
        new DaC<Interval, Primes>( new IntervalDivide(),
                                   new IntervalDivideCondition(),
                                   new SearchInterval(),
                                   new MergePrimes());
```

In this example we only require a single *d&c* skeleton so there is no nesting. If the application had required nesting, the nested skeleton would have been passed instead of the `SearchInterval` muscle.

Then, a Calcium instance can be created and the application executed as shown in Listing 3.3. Lines 1-2 instantiate an execution environment (see 3.3), which in this case corresponds to a `MultiThreadedEnvironment` with a maximum of 10 threads. A booting and shutdown process are required, as depicted on lines 4 and 18, to instantiate and clean resources.

Afterwards, line 6 instantiates a `Stream` associated with the defined skeleton (`findprimes`). A `Stream` provides a mechanism to compute, with the same

skeleton, multiple input parameters. A single `Calcium` instance can have multiple `Stream`s which share the resources offered by the same environment. Thus, different skeleton programs can be computed simultaneously in the same resource environment. In this simple example we only use one `Stream`.

Lines 8-10 show the input of the parameters, which for this example correspond to three different intervals: $[1, 6400], [1, 100], [1, 640]$. Each parameter has a different `threshold`: $300, 20, 64$ respectively. For each input a `Future` container is created. The `Future` allows for asynchronism and data synchronization. When the program requires the result of the skeleton computation, the `Future.get()` method can be invoked to block the application until the result is available, as shown on lines 14-16.

3.5 Conclusion

This chapter has presented the Algorithmic Skeleton programming model used in Calcium. The first section of this chapter described the programming model from a theoretical point of view, and then the second section showed how this theoretical model is applied to a a Java based skeleton framework: Calcium. The third section provided a description of how Calcium can be used with different execution environments. Then, the forth section provided a full example of skeleton programming using a naive example: searching for primes on a given interval.

In the following chapters we will address three pragmatic features for algorithmic skeleton programming which are unique to Calcium. First in Chapter 4 we will address performance tuning of algorithmic skeleton programs. Then in Chapter 5 we will formally define a type system and then show how it can be implemented on top of Java Generics. Finally, in Chapter 6 we will show how transparent file access can be added to algorithmic skeletons.

Chapter 4

Performance Tuning

Contents

4.1 Muscle Tuning of Algorithmic Skeletons 62

4.1.1 Performance Diagnosis 62

4.1.2 Performance Metrics 62

4.1.3 Muscle Workout 63

4.1.4 Code Blaming 63

4.2 NQueens Test Case 64

4.3 Conclusions and Future Work 66

Skeletons are considered a high level programming paradigm because lower level details are hidden from the programmer. Achieving high performance for an application becomes the responsibility of the skeleton framework by performing optimizations on the skeleton structure [AD99, ADD04], and adapting to the environment's dynamicity [Dan05]. However, while these techniques are known to improve performance, by themselves they are not sufficient. The functional aspects of the application (i.e. the muscle), which are provided by the programmer, can be inefficient or generate performance degradations inside the framework.

The main motivations of this chapter are to detect the performance degradations, provide programmers with an explanation, and suggesting how to solve the performance bugs. The challenge arises because skeleton programming is a high-level programming model. All the complex details of the parallelism and distribution are hidden from programmers. Therefore, programmers are unaware of how their muscle code will affect the performance of their applications. Inversely, low level information of the framework has no meaning for programmers to fine tune their muscle blocks.

In this chapter we contribute by: (i) providing performance metrics that apply, not only to master-slave, but also to other common skeleton patterns; (ii) taking into consideration the nesting of task and data parallel skeletons as a producer/consumer problem; (iii) introducing the concept of muscle workout; and

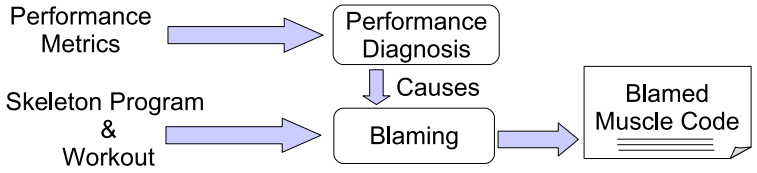


Figure 4.1: Finding the tunable muscle code

(iv) introducing a blaming phase that relates performance inefficiency causes with the actual muscle code.

4.1 Muscle Tuning of Algorithmic Skeletons

A birds eye view of the methodology presented in this chapter is depicted in Figure 4.1. After the execution of an application, the performance metrics are used to determine the causes of the performance inefficiency. Once the causes are identified, a blaming process takes places by considering the workout of the muscle code. The result of the blaming process yields the blamed muscle code. Programmers can then analyze the blamed code to fine tune their applications.

4.1.1 Performance Diagnosis

Figure 4.2 shows a generic inference tree, which can be used to diagnose the performance of data parallelism in algorithmic skeletons. The diagnosis uses the metrics identified in section 4.1.2 to find the causes of performance bugs. Causes can correspond to two types. *External causes*, such as the framework deployment overhead, or framework overload; and *tunable causes*, which are related with the muscle code of the skeleton program. Since external causes are unrelated with the application's code, in this chapter we focus on the tunable causes.

4.1.2 Performance Metrics

Task-Pool Number of tasks in each state: $N_{processing}$, N_{ready}, $N_{waiting}$, and $N_{finished}$.

Time Time spent by a task in each state: $T_{processing}$, T_{ready}, $T_{waiting}$, and $T_{finished}$. For $T_{processing}$ and T_{ready} this represents the accumulated time spent by all the task's subtree family members in the state. For the $T_{waiting}$ and $T_{finished}$, this represents only the time spent by the root task in the waiting state. There is also the $T_{wallclock}$ and $T_{computing}$. The overhead time is defined as: $T_{overhead} = T_{processing} - T_{computing}$, and represents mainly the cost of communication time between the task-pool and the interpreters.

Granularity Provides a reference of the task granularity achieved during the execution of data parallel skeletons by monitoring the task tree: size, span, and depth ($span^{depth} = size$), and the $granularity = \frac{T_{computing}}{T_{overhead}}$.

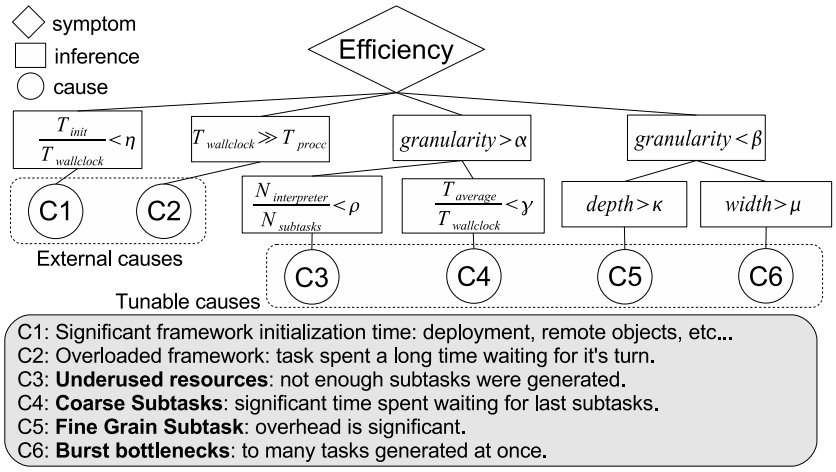


Figure 4.2: Generic Cause Inference Tree for Data Parallelism

4.1.3 Muscle Workout

Let $m_0, ..., m_k$ be an indexed representation of all the muscle code inside a skeleton program $\triangle$. We will say that workout is a function that, given a skeleton program and a state parameter, after the application is executed, returns a list of all the executed muscle codes with the computation time for each instruction:

$$workout(\triangle, p) = [(m_i, t_0), ..., (m_j, t_n)]$$

The skeleton workout represents a trace of how the muscle codes were executed for this skeleton program. The same muscle code can appear more than once in the workout having different execution times.

4.1.4 Code Blaming

Since algorithmic skeletons abstract lower layers of the infrastructure from the programming of the application, low level causes have no meaning to the programmer. Therefore, we must link the causes with something that the programmer can relate to, and this corresponds to the muscle codes which have been implemented by the programmer. The process of relating lower level causes with the responsible code is what we refer to as *code blaming*.

A blaming mechanism must link each inferred cause with the relevant muscle codes. Thus, the blaming must consider: lower level causes (the result of the performance diagnosis), the skeleton program, and the muscle's workout.

Let us recall that for any skeleton program, its muscle codes must belong to one of the following types: f_e, f_b, f_d, f_c. Additionally, the semantics of the muscle code depend on the skeleton pattern where it is used. A simple implementation of a blaming algorithm is the following:

(C3) Underused resources Blame the most invoked $f_b \in \{d\&q\}$ and $f_d \in \{map, d\&q, fork\}$. Suggest incrementing the times f_b returns *true*, and suggest that f_d divides into more parts.

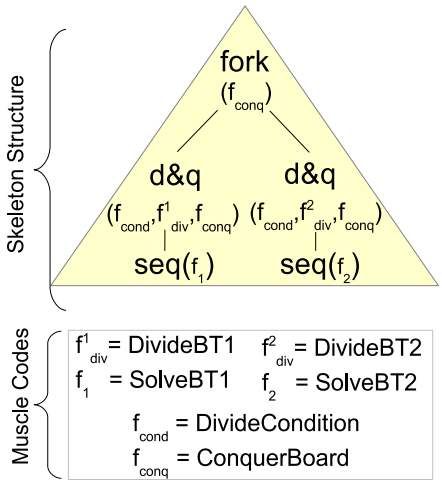


Figure 4.3: N-Queens Skeleton Program

(C4) Coarse subtasks Blame the least invoked $f_b \in \{d\&q\}$. Suggest incrementing the times f_b returns *true*.

(C5) Fine grain subtasks Blame the most invoked $f_b \in \{d\&q\}$. Suggest reducing the number of times f_b returns *true*.

(C6) Burst Bottlenecks Blame the $f_d \in \{map, d\&q, fork\}$ which generated the most number of subtasks. Suggest modifying f_d to perform less divisions per invocation.

While more sophisticated blaming mechanisms can be introduced, as we shall see in the NQueens test case, even this simple blaming mechanism can provide valuable information to the programmer.

4.2 NQueens Test Case

The experiments were conducted using the *sophia* and *orsay* sites of Grid5000 [Gri], a french national Grid infrastructure. The machines used AMD Opteron CPU at 2Ghz, and 1 GB RAM. The task-pool was located on the sophia site, while the interpreters where located on the orsay site. The communication link between sophia and orsay was of $1[\frac{Mbit}{sec}]$ with $20[ms]$ latency.

As a test case, we implemented a solution of the NQueens counting problem: How many ways can n non attacking queens be placed on a chessboard of nxn? Our implementation is a skeleton approach of the Takaken algorithm [Tak], which takes advantage of symmetries to count the solutions. The skeleton program is shown in Figure 4.3, where two `d&c` are forked to be executed simultaneously. The first `d&c` uses the *backtrack1* algorithm which counts solutions with 8 symmetries, while the other `d&c` uses the *backtrack2* algorithm that counts solutions with 1, 2, and 4 symmetries.

We have chosen the NQueens problem because the fine tuning of the application is centered in one parameter, and therefore represents a comprehensible

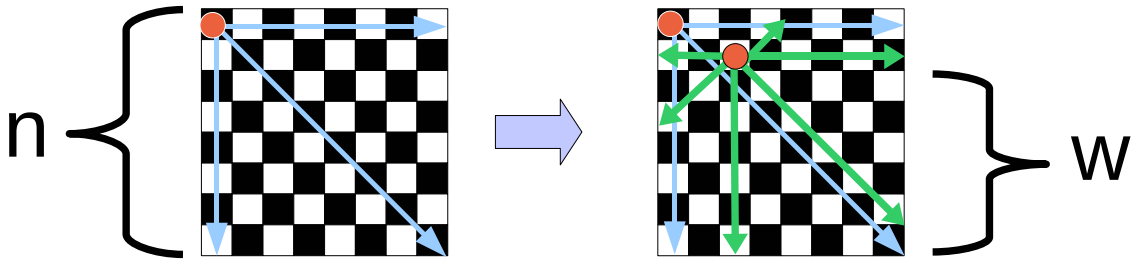


Figure 4.4: N-Queens Chessboard Filling

(a) Performance Metrics

W	Speedup	Efficiency	Granularity	Size	Width	Depth	T_avg [ms]	T_avg/T_wall
16	11,68	0,12	0,13	24892	11	4,2	0,59	0,05
17	97,27	0,97	5,33	2178	13	3	6,8	3,45
18	67,59	0,68	59,18	166	13	2	90,99	33,89

(b) Workout Summary

	w = 16		w = 17		w = 18	
Muscle Code	**Invoqued [#]**	**Time[ms]**	**Invoqued [#]**	**Time[ms]**	**Invoqued [#]**	**Time[ms]**
DivideBT1	259	19	18	1	1	1
DivideBT2	1918	154	147	26	10	1
ForkDefaultDivide	1	2	1	2	1	2
SolveBT2	19872	13619957	1771	13730299	137	13910314
ConquerBoard	2178	175	166	15	12	1
SolveBT1	2842	1103186	241	1097325	17	1207444
DivideCondition	24891	523	2177	64	165	10

Table 4.1: Performance Metrics & Workout Summary for $n = 20$, $w \in \{16, 17, 18\}$

test case. As shown in Figure 4.4, task division is achieved by specifying the first $n - w$ queens on the board, where w is the number of queens that have to be backtracked. The problem is thus known to be $O(n^n)$, with exponential increase of the number of tasks as w decreases. Therefore, the programmer of the NQueens problem must tune a proper value for w.

Since the output of the blaming phase corresponds to the blamed muscle code, it is up to the user to identify the parameters that change the behavior of the muscle code. Therefore, the same approach can be used for applications that require tuning of multiple parameters.

We tested a problem instances for $n = 20$, $w \in \{16, 17, 18\}$, $nodes = 100$. Relevant metrics are shown in Table 4.1(a), and a summary of the workout is shown in Table 4.1(b). From the performance point of view, several guides can be obtained by simple inspection of the workout summary. For example, that the method `DivideCondition` must remain computationally lite because it is called the most number of times, and that further optimizations of `SolveBT2` will provide the most performance gain.

The result of the blaming process is shown in Figure 4.1, where the f_b muscle code (`DivideCondition`) appears as the main tuning point of the application. For $w = 16$ the results suggest that the granularity of the subtasks is to fine,

```
//(n = 20, w = 16)
  Performance inefficiency found.
    Cause: Subtask are too fine grain, overhead is significant.
    Blamed Code: public boolean nqueens.DivideCondition.
       evalCondition(nqueens.Board)
    Suggested Action: This method should return true less often.

//(n = 20, w = 17)
  No inefficiency found.

//(n = 20, w = 18)
  Performance inefficiency found.
    Cause: Subtask are too coarse, significant time spent waiting
        for last subtasks.
    Blamed Code: public boolean nqueens.DivideCondition.
       evalCondition(nqueens.Board)
    Suggested Action: This method should return true more often.
```

Listing 4.1: Fine Tuning Output for $n = 20, w \in \{16, 17, 18\}$

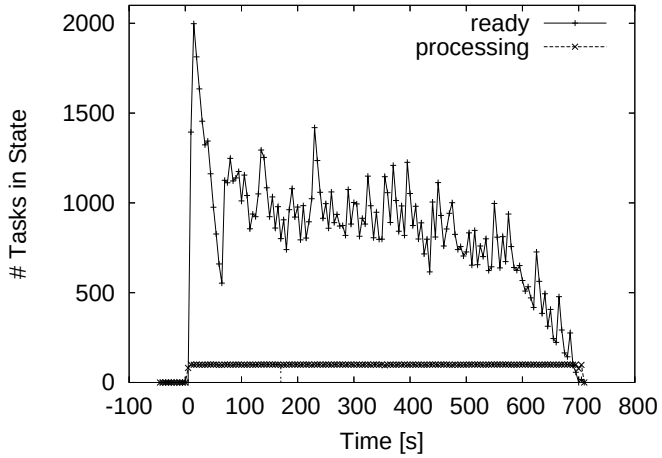
while for $w = 18$ the granularity is to coarse. To better understand why this takes place Figures 4.6(a), 4.6(b), 4.6(c) show the number of subtasks in ready and processing state during the computation. The figures concur with the fine tuning output. As we can see, the fine tuning of a skeleton program can have a big impact on the performance of the application. In the case of the NQueens test case, the performance improves up to 4 times when choosing $w = 17$ instead of $w \in \{16, 18\}$.

4.3 Conclusions and Future Work

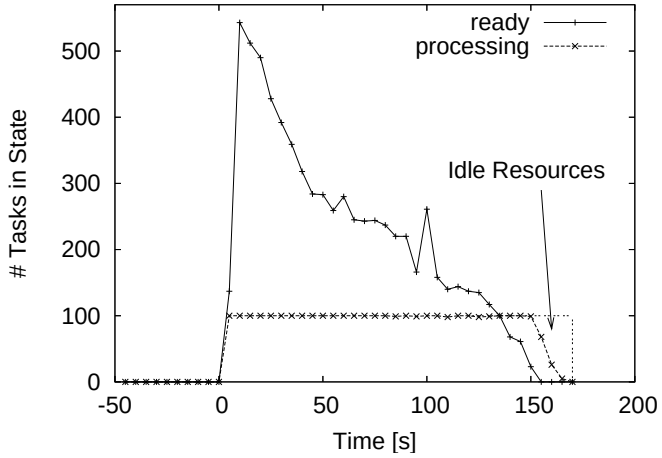
Dynamic performance tuning tools have been designed to aid developers in the process of detecting and explaining performance bugs. An example of such tools is POETRIES [CMSL04], which proposes taking advantage of the knowledge about the structure of the application to develop a performance model. Hercules [LM06] is another tool that has also suggested the use of pattern based performance knowledge to locate and explain performance bugs. Both POETRIES and Hercules have focused on the master-slave pattern, and promoted the idea of extending their models to other common patterns. Nevertheless, to our knowledge, none have yet considered performance tuning of *nestable patterns* (i.e. skeletons), nor have provided a mechanism to relate the performance bugs with the responsible muscle codes of the program.

For skeletons, in [BCGH04] performance modeling is done using process algebra for improving scheduling decisions. Contrary to the previously mentioned approaches, this approach is static and mainly aimed at improving scheduling decisions, not at providing performance tuning.

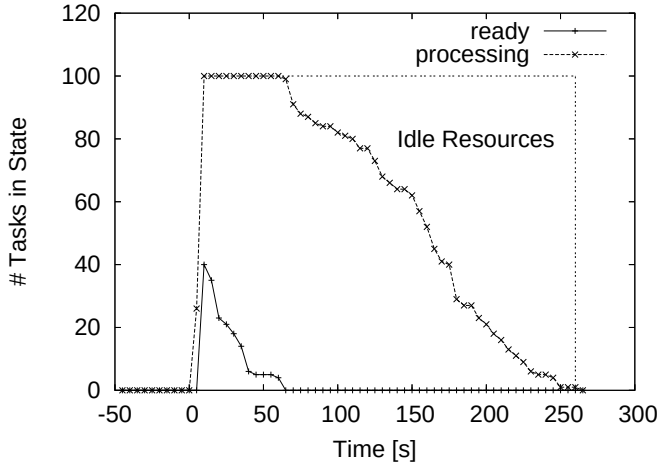
In this chapter we have presented a mechanism to perform fine tuning of al-



(a) $w = 16$



(b) $w = 17$



(c) $w = 18$

Figure 4.5: Number of ready and processing subtasks for $n = 20$, with 100 nodes.

gorithmic skeletons' muscle code. The approach extends previous performance diagnosis techniques that take advantage on pattern knowledge by: taking into consideration netsable skeleton patterns, and relating the performance inefficiency causes with the skeleton's responsible muscle code. This is necessary because skeleton programming is a higher-level programming model, and as such, low level causes of performance inefficiencies have no meaning to the programmer.

The proposed approach can be applied to fine tune applications that are composed of nestable skeleton patterns. To program such applications we used the Calcium skeleton framework. The relation of inefficiency causes with the responsible muscle code is found by taking advantage of the skeleton structure, which implicitly informs the role of each muscle code.

We have validated the approach with a test case of the NQueens counting problem. The experiments where conducted on Grid5000 with up to a 100 nodes.

Chapter 5

Type Safe Algorithmic Skeletons

Contents

5.1 Related Work 71
5.2 A typed algorithmic skeleton language 72
5.2.1 Skeleton Language Grammar 72
5.2.2 Reduction Semantics 72
5.2.3 Type System 73
5.2.4 Typing Property: Subject Reduction 75
5.2.5 Sub-typing 78
5.3 Type safe skeletons in Java 79
5.4 Conclusions 80

From a general point of view, typing ensures safety properties of programming languages. For instance, in object-oriented languages, typing ensures that each field or method access will reach an existing field or method: "message not understood" errors will not occur. Nevertheless, type systems are limited, that is why in most object-oriented languages, type-casts have been introduced. In exceptional cases, type-casts allow programmers to precise the type of a known object, but this step is error-prone. Therefore, it is important to reduce the number of necessary type-casts to increase the type safety of a program; this is one of the purposes of Java generics or C++ templates, for example.

Algorithmic skeletons (*skeletons* for short) are a high level programming model for parallel and distributed computing, introduced by Cole in [Col91]. Skeletons take advantage of common programming patterns to hide the complexity of parallel and distributed applications. Starting from a basic set of patterns (skeletons), more complex patterns can be built by nesting the basic ones. All the non-functional aspects regarding parallelization and distribution are implicitly defined by the composed parallel structure. Once the structure has been defined, programmers complete the program by providing the application's sequential blocks, called *muscle* functions.

Let us recall that muscle functions correspond to the sequential functions of the program (see Section 3.2.2). We classify muscle functions using the following

```
Skeleton stage1= new Seq(new Execute1());
Skeleton stage2= new Seq(new Execute2());
Skeleton skeleton=new Pipe(stage1, stage2); //type error undetected during composition
----------------------------------------------------------------------------------------

   class Execute1 implements Execute{      |   class Execute2 implements Execute{
      public Object exec(Object o){        |      public Object exec(Object o){
         A a = (A)o; /* Runtime cast */    |         B b = (B)o; /* Runtime cast */
         ...                               |         ...
         return x; /* Unknown type */      |         return y; /* Unknown type */
      }                                    |      }
   }                                       |   }
```

Listing 5.1: Motivation Example: Unsafe skeleton programming

categories: execution (f_e), evaluation of conditions (f_b), division of data (f_d), and conquer of results (f_c). Each muscle function is a black box unit for the skeleton language/framework, and while each may be error free on its own, an inadequate combination with a skeleton pattern can yield runtime typing errors. This happens because, during the evaluation of the skeleton program, the result of one muscle function is passed to "the next" muscle function as parameter. Where "the next" is determined at runtime by the specific skeleton assembly. An incompatible type between the result of a muscle function and the parameter of the next one yields runtime errors, which are difficult to detect and handle on distributed environments.

Untyped skeleton programming forces the programmer to rely on type-casts in the programming of muscle functions. Indeed, if the higher-level language (skeletons) is not able to transmit types between muscle functions, then the poorest assumption will be taken for typing muscle codes (e.g., in an untyped skeleton implementation in Java, muscle functions accept only `Object` as type for parameter/result). In that case, every object received by a muscle function has to be casted into the right type, which is highly error-prone.

On the other hand, typed skeletons relieve the programmer from having to type-cast every muscle function argument: basically, the type system will check that the input type of a skeleton is equal to the declared output type of the preceding skeleton. Type-casts remain necessary only when it would be required by the underlying language. *To summarize, the type safety ensured by the underlying language is transmitted by the skeletons: type safety is raised to the skeleton level.*

Let us consider the example shown in Listing 5.1, which exposes the dangers of typeless skeleton programming. In the example two functions $\{f_1, f_2\}$ are executed sequentially the other using a *pipe* skeleton: $pipe(seq(f_1), seq(f_2))$. During the evaluation of the skeleton program, the unknown return type of `Execute1.exec` will be passed as parameter to `Execute2.exec` which is expecting a type `B` parameter. While `Execute1.exec` (f_1) and `Execute2.exec` (f_2) may be correct on their own, there is no guarantee that piping them together will not yield an error.

As shown in the example, type safe skeleton programming does not only require a method for expressing the types of muscle functions (parameters/return values), but also an underlying type-system expressing how the typed muscle

functions interact with the skeleton patterns.

Indeed, one of the original aspects of the proposed type system for algorithmic skeletons is that it transmits types between muscle functions. As a consequence, the type preservation property has a greater impact than in usual type systems: it also ensures that muscle functions will receive and produce correct types relatively to the assembly.

Our contribution consists in detecting type errors in skeleton programs, thus improving the safeness of algorithmic skeleton programming. From the theoretical side, we contribute by providing type semantics rules for each of the skeleton patterns in the language. Then, we prove that these typing rules indeed satisfy type safety properties. On the practical side, we show how the proposed type system can be implemented in a Java [Micc] skeleton library by taking advantage of Java Generics [BOSW98]. As such, we show that no further type-casts are imposed by the skeleton language inside muscle functions, and more importantly, that all the typing validations of skeleton compositions are performed at compilation time.

Section 5.1 presents related works. In section 5.2 we formally define a type system for skeletons and prove its correctness. Then, section 5.3 shows an implementation of the type system in a skeleton framework based on Java.

5.1 Related Work

Besides the semantics introduced in Section 3.1.3, other works have also provided parallelism semantics for skeletons. For example, Aldinucci et al. have provided semantics that can handle both task and data parallelism [AD04, AD07b]. The semantics describe both functional and parallel behavior of the skeleton language using a labeled transition system. This is also the case for QUAFF where the semantics are transformed into a process network.

Therefore, with generality in mind, in this chapter we do not focus on the parallelism aspects of the reduction semantics, and instead use big-step reduction semantics. Thus this chapter can focuses exclusively on the typing rules, which allows us to deal with type safety.

The Muesli skeleton library provides some of its skeletons as generics using C++ templates. Nevertheless, no type system is enforced with the templates, allowing type unsafe skeleton compositions. Concerning typing, the P3L skeleton language [BDO+95] provides type verification at the data flow level, which must be explicitly handled by the programmer using intermediate variables. Compared with Calcium, the type system proposed in Calcium enforces safeness at a higher level of abstraction: the skeleton level, where the data flow is implicit. In Skil typed skeletons are used as a mean to make skeletons polymorphic. Skil translates polymorphic high order functions into monomorphic first order C functions. Nevertheless, the type system described in Skil is not a formal type system, and hence does not prove type safety of skeleton composition.

QUAFF is the only other skeleton library which support nestable skeletons and is concerned with type checking (through C++ templates). Nevertheless, to

our knowledge the type checking used in QUAFF has not been formalized into a type system nor has the subject reduction property been proven.

5.2 A typed algorithmic skeleton language

This section defines a type theory for skeleton programming. In order to present such a theory, we first specify a syntax for algorithmic skeletons in a very classical way. Then section 5.2.2 defines a big-step reduction semantics for skeletons; this semantics, though not as rich as the one presented in [AD04, AD07b] is sufficient for proving the type properties that interest us. We then provide a simple type system [Pie02], and prove that this type system provides the usual property of type preservation: subject-reduction. Subject-reduction ensures that skeleton compositions do not threaten typing, and that type information can be passed by skeletons to be used in the programming of muscle function codes. This property greatly improves the correctness of skeleton programs by allowing the underlying language and the algorithmic skeletons to cooperate on this aspect.

5.2.1 Skeleton Language Grammar

Let us recall the skeleton grammer introduced in Section 3.1. The task parallel skeletons are: *seq* for wrapping execution functions; *farm* for task replication; *pipe* for staged computation; *while*/*for* for iteration; and *if* for conditional branching. The data parallel skeletons are: *map* for single instruction multiple data; *fork* which is like *map* but applies multiple instructions to multiple data; and *d&c* for divide and conquer.

$$\begin{aligned}\triangle ::= \ & \text{seq}(f_e) \mid \text{farm}(\triangle) \mid \text{pipe}(\triangle_1, \triangle_2) \mid \text{while}(f_b, \triangle) \mid \\ & \text{if}(f_b, \triangle_{true}, \triangle_{false}) \mid \text{for}(i, \triangle) \mid \text{map}(f_d, \triangle, f_c) \mid \\ & \text{fork}(f_d, \{\triangle_i\}, f_c) \mid \text{d\&c}(f_d, f_b, \triangle, f_c)\end{aligned}$$

Notations In the following, f_x denotes muscle functions, $\triangle$ denotes skeletons, terms are lower case identifiers and types upper case ones. Also, brackets denote lists (or sets) of elements, i.e., $\{p_i\}$ is the list consisting of the terms p_i, and $\{Q\}$ is the type of a list of elements of type Q.

5.2.2 Reduction Semantics

Figure 5.1 presents a big-step operational semantics for skeletons. It relies on the fact that semantics for muscle functions are defined externally. In other words, for any function f_x and for any term p we have a judgment of the form: $f_x(p) \Downarrow r$.

$$\text{R-FARM}\ \frac{\triangle(p) \Downarrow r}{farm(\triangle)(p) \Downarrow r} \qquad \text{R-PIPE}\ \frac{\triangle_1(p) \Downarrow s \qquad \triangle_1(s) \Downarrow r}{pipe(\triangle_1, \triangle_2)(p) \Downarrow r} \qquad \text{R-SEQ}\ \frac{f_e(p) \Downarrow r}{seq(f_e)(p) \Downarrow r}$$

$$\text{R-IF-TRUE}\ \frac{f_b(p) \Downarrow true \qquad \triangle_{true}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r} \qquad \text{R-IF-FALSE}\ \frac{f_b(p) \Downarrow false \qquad \triangle_{false}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}$$

$$\text{R-WHILE-TRUE}\ \frac{f_b(p) \Downarrow true \qquad \triangle(p) \Downarrow s \qquad while(f_b, \triangle)(s) \Downarrow r}{while(f_b, \triangle)(p) \Downarrow r} \qquad \text{R-WHILE-FALSE}\ \frac{f_b(p) \Downarrow false}{while(f_b, \triangle)(p) \Downarrow p}$$

$$\text{R-FOR}\ \frac{\forall i < n \quad \triangle(p_i) \Downarrow p_{i+1}}{for(n, \triangle)(p_0) \Downarrow p_n} \qquad \text{R-MAP}\ \frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{map(f_d, \triangle, f_c)(p) \Downarrow r}$$

$$\text{R-FORK}\ \frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle_i(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r} \qquad \text{R-D\&C-FALSE}\ \frac{f_b(p) \Downarrow false \qquad \triangle(p) \Downarrow r}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}$$

$$\text{R-D\&C-TRUE}\ \frac{f_b(p) \Downarrow true \qquad f_d(p) \Downarrow \{p_i\} \qquad \forall i\ d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}$$

Figure 5.1: Skeleton's Reduction Semantics

For example, the *pipe* construct corresponds to the sequential composition of two algorithmic skeletons. Thus R-PIPE states that if a skeleton $\triangle_1$ applied to a parameter p ($\triangle_1(p)$) can be reduced to s, and $\triangle_2(s)$ can be reduced to r, then $pipe(\triangle_1, \triangle_2)(p)$ will be reduced to r. In other words, the result of $\triangle_1$ is used as parameter for $\triangle_2$: $\triangle_2(\triangle_1(p)) \Downarrow r$.

The *d&c* construct performs recursive divide and conquer. It recursively splits its input by the divide function f_d until the condition f_b is false, processes each piece of the divided data, and merges the results by a conquer function f_c.

5.2.3 Type System

Figure 5.2 defines a type system for skeletons. It assumes that each muscle function is of the form: $f_x : P \rightarrow R$, and verifies the following classical typing rule:

$$\text{APP-F}\ \frac{p : P \qquad f_x : P \rightarrow R}{f_x(p) : R}$$

We first define a typing rule for each of the skeleton constructs. These typing

rules allow us to infer the type of an algorithmic skeleton based on the type of the skeletons and muscle functions composing it. Typing judgments for skeletons are of the form $\triangle : P \rightarrow R$. We explain below the typing of two representative rules: T-PIPE and T-D&C.

T-PIPE Consider a *pipe* formed of skeletons $\triangle_1$ and $\triangle_2$. First, the input type of *pipe* is the same as the input type of $\triangle_1$, and the output type of *pipe* is the same as the output of $\triangle_2$. Moreover, the output type of $\triangle_1$ must match the input type of $\triangle_2$. In other words, consider the typeless skeleton example shown in Listing 5.1, which would be equivalent to the following code:

```
Object a = ...; //previous result
Object b = Execute1.execute(a);
Object c = Execute2.execute(b);
```

Without a typing system, the *pipe* skeleton cannot ensure type safety. The *pipe* typing ensures that types transmitted by the *pipe* parameters are compatible; and the recursive nature of the typing system ensures the correct typing of skeletons containing a *pipe*, but also of skeletons nested inside the *pipe*. Here, type compatibility ensures that the type of `b` is compatible with the type of the parameter for `Execute2.execute`. Also, that `a` is compatible with the parameter of `Execute1.execute`, and the type of `c` is compatible with the following muscle instruction (belonging to another skeleton construct).

With a typing system, the *pipe* skeleton yields a code equivalent to:

```
A a = ...; //previous muscle result
B b = Execute1.execute(a);
C c = Execute2.execute(b);
```

Where the types of a, b, and c are $a{:}A$, $b{:}B$, and $c{:}C$ respectively.

T-D&C Consider now a *d&c* skeleton that accepts an input of type P and returns an output of type R. Therefore, the choice function f_b must also accept an input of type P, and return a boolean. Secondly, the divide function f_d produces a list of elements of P. Then, each element is computed by a sub-skeleton of type $P \rightarrow R$. Finally, the conquering function f_c must accept as input a list of elements of R and return a single element of type R, which corresponds to the return type of the *d&c* skeleton.

To summarize, the typing rules follow the execution principles of the skeletons, attaching a type to each intermediate result and transmitting types between skeletons.

Finally, a skeleton applied to a term follows the trivial typing rule:

$$\text{APP-}\triangle \quad \frac{p : P \qquad \triangle : P \rightarrow R}{\triangle(p) : R}$$

$$\text{T-FARM}\ \frac{\triangle : P \to R}{farm(\triangle) : P \to R} \qquad \text{T-PIPE}\ \frac{\triangle_1 : P \to X \qquad \triangle_2 : X \to R}{pipe(\triangle_1, \triangle_2) : P \to R} \qquad \text{T-SEQ}\ \frac{f_e : P \to R}{seq(f_e) : P \to R}$$

$$\text{T-IF}\ \frac{f_b : P \to boolean \qquad \triangle_{true} : P \to R \qquad \triangle_{false} : P \to R}{if(f_b, \triangle_{true}, \triangle_{false}) : P \to R}$$

$$\text{T-WHILE}\ \frac{f_b : P \to boolean \qquad \triangle : P \to P}{while(f_b, \triangle) : P \to P} \qquad \text{T-FOR}\ \frac{i : integer \qquad \triangle : P \to P}{for(i, \triangle) : P \to P}$$

$$\text{T-MAP}\ \frac{f_d : P \to \{Q\} \qquad \triangle : Q \to S \qquad f_c : \{S\} \to R}{map(f_d, \triangle, f_c) : P \to R}$$

$$\text{T-FORK}\ \frac{f_d : P \to \{Q\} \qquad \triangle_i : Q \to S \qquad f_c : \{S\} \to R}{fork(f_d, \{\triangle_i\}, f_c) : P \to R}$$

$$\text{T-D\&C}\ \frac{f_b : P \to boolean \qquad f_d : P \to \{P\} \qquad \triangle : P \to R \qquad f_c : \{R\} \to R}{d\&c(f_d, f_b, \triangle, f_c) : P \to R}$$

Figure 5.2: Skeleton's Type System

5.2.4 Typing Property: Subject Reduction

A crucial property of typing systems is subject reduction. It asserts the type preservation by the reduction semantics; this means that a type inferred for an expression will not change (or will only become more precise) during the execution: the type of an evaluated expression is compatible with the type of this expression before evaluation. Without this property, none of the properties ensured by the type-system would be useful. For skeletons, subject-reduction can be formalized as follows.

Theorem 1 (Subject Reduction)**.** *Provided muscle functions ensure application and subject reduction, i.e.:*

$$\text{SR-F}\ \frac{f_x(p) : Q \qquad f(p) \Downarrow q}{q : Q}$$

The skeleton type system ensures subject reduction:

$$\text{SR-}\triangle\ \frac{\triangle(p) : R \qquad \triangle(p) \Downarrow r}{r : R}$$

While the property should be proven for every single skeleton construct, with conciseness in mind, we only illustrate here the proof in the representative cases of: *pipe*, *for*, and *d&c* constructs. Please refer to Appendix A for the other constructs.

The general structure of the proof is straightforward. For each skeleton construct we: particularize subject-reduction; decompose the inference that can lead to the correct typing of the skeleton; and verify that, for each possible reduction rule for the skeleton, the type is preserved by the reduction. The proof also involves some double recursions in the most complex cases.

To prove this property, an alternative approach can be to design a type-system closer to the one of λ-calculus (but with a fixed point operator). Nevertheless, we have chosen the operational semantics approach because of its simplicity, and direct meaning in terms of typed skeletons.

Pipe Preservation

Subject-reduction for *pipe* skeletons means:

$$\text{SR-PIPE}\ \frac{pipe(\triangle_1, \triangle_2)(p) : R \qquad pipe(\triangle_1, \triangle_2)(p) \Downarrow r}{r : R}$$

Proof. Let us decompose the inferences asserting that $pipe(\triangle_1, \triangle_2)(p) : R$ and $pipe(\triangle_1, \triangle_2)(p) \Downarrow r$, necessarily:

$$\dfrac{p : P \qquad \dfrac{\triangle_1 : P \to X \qquad \triangle_2 : X \to R}{pipe(\triangle_1, \triangle_2) : P \to R}\ \text{T-PIPE}}{pipe(\triangle_1, \triangle_2)(p) : R}\ \text{APP-}\triangle \qquad \dfrac{\triangle_1(p) \Downarrow x \qquad \triangle_2(x) \Downarrow r}{pipe(\triangle_1, \triangle_2)(p) \Downarrow r}\ \text{R-PIPE}$$

Finally, we prove that r has the type R as follows:

$$\dfrac{\dfrac{\dfrac{\dfrac{p : P \quad \triangle_1 : P \to X}{\triangle_1(p) : X}\ \text{APP-}\triangle \qquad \triangle_1(p) \Downarrow x}{x : X}\ \text{SR-}\triangle \qquad \triangle_2 : X \to R}{\triangle_2(x) : R}\ \text{APP-}\triangle \qquad \triangle_2(s) \Downarrow r}{r : R}\ \text{SR-}\triangle$$

To summarize, we proved the subject-reduction property for (T-PIPE) combined with (APP-$\triangle$), which is the only way to obtain a correctly typed and reducible expression involving a *pipe* construct. □

For Preservation

In the case of the *for* skeleton, we must prove:

$$\text{SR-FOR}\ \frac{for(n, \triangle)(p) : P \qquad for(n, \triangle)(p) \Downarrow r}{r : P}$$

Proof. We decompose $for(n, \triangle)(p) : P$ and $for(n, \triangle)(p) \Downarrow r$, noting $p_0 = p$, $p_n = r$ we have:

$$\dfrac{p_0 : P \qquad \dfrac{n : integer \quad \triangle : P \to P}{for(n, \triangle) : P \to P}\ \text{T-FOR}}{for(n, \triangle)(p_0) : P}\ \text{APP-}\triangle$$

$$\dfrac{\forall i < n \quad \triangle(p_i) \Downarrow p_{i+1}}{for(n, \triangle)(p_0) \Downarrow p_n}\ \text{R-FOR}$$

We prove that $\forall i \leq n$, $p_i : P$ using induction on i. The base case is true $p_0 : P$, the inductive hypothesis is that $p_i : P$, and we must prove that $p_{i+1} : P$. Applying the recurrence hypothesis SR-$\triangle$, and APP-$\triangle$ we have:

$$\dfrac{\text{APP-}\triangle\ \dfrac{p_i : P \qquad \triangle : P \to P}{\triangle(p_i) : P} \qquad \triangle(p_i) \Downarrow p_{i+1}}{p_{i+1} : P}\ \text{SR-}\triangle$$

Therefore, $p_n : P$, and in the orginal notation $r : P$. □

D&C Preservation

For *d&c* skeletons, subject-reduction becomes:

$$\dfrac{\text{SR-}d\&c \qquad d\&c(f_d, f_b, \triangle, f_c)(p) : R \qquad d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}{r : R}$$

Proof. $d\&c(f_d, f_b, \triangle, f_c)(p) : R$ necessarily comes from

$$\dfrac{p : P \qquad \dfrac{\begin{array}{c} f_b : P \to boolean \\ f_d : P \to \{P\} \\ \triangle : P \to R \\ f_c : \{R\} \to R \end{array}}{d\&c(f_d, f_b, \triangle, f_c) : P \to R}\ \text{T-D\&C}}{d\&c(f_d, f_b, \triangle, f_c)(p) : R}\ \text{APP-}\triangle$$

In addition to the skeleton based recurrence, we need to use another recurrence on p for which the base case is $f_b(p) \Downarrow false$ and the inductive case is $f_b(p) \Downarrow true$. This recursion is finite because the division of the problem must always terminate (one can formalize the recurrence based on a $size$ function such that $f_d(p) \Downarrow \{p_i\} \Rightarrow size(p_i) < size(p)$ and $size(p) \leq 0 \Rightarrow f_b(p) \Downarrow false$).

CASE 1: if $f_b(p) \Downarrow false$

$$\dfrac{f_b(p) \Downarrow false \qquad \triangle(p) \Downarrow r}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r}\ \text{R-}D\&C\text{-FALSE}$$

by hypothesis, $\triangle : P \rightarrow R$ and thus:

$$\text{APP-}\triangle \ \frac{\triangle : P \rightarrow R \qquad p : P}{\triangle(p) : R}$$

By the recurrence hypothesis ($\triangle$ is sub-skeleton of $d\&c(f_d, f_b, \triangle, f_c)$), $\triangle$ verifies subject reduction:

$$\frac{\triangle(p) : R \qquad \triangle(p) \Downarrow r}{r : R} \ \text{SR-}\triangle$$

which ensures that $r : R$ and the subject reduction for *d&c*.

CASE 2: if $f_b(p) \Downarrow true$

$$\frac{\begin{array}{c} f_b(p) \Downarrow true \qquad f_d(p) \Downarrow \{p_i\} \\ \forall i \quad d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i \\ f_c(\{r_i\}) \Downarrow r \end{array}}{d\&c(f_d, f_b, \triangle, f_c)(p) \Downarrow r} \ \text{R-D\&C-TRUE}$$

First, each p_i is of type P:

$$\begin{array}{r} \text{APP-F} \\ \text{SR-F} \end{array} \ \frac{\begin{array}{c} f_d : P \rightarrow \{P\} \qquad p : P \\ \hline f_d(p) : \{P\} \end{array} \qquad f_d(p) \Downarrow \{p_i\}}{\{p_i\} : \{P\}}$$

and thus, using the "sub"recurrence hypothesis, that is subject reduction on p_i:

$$\frac{d\&c(f_d, f_b, \triangle, f_c)(p_i) : R \qquad d\&c(f_d, f_b, \triangle, f_c)(p_i) \Downarrow r_i}{r_i : R} \ \text{SR-D\&C}$$

Therefore, $\forall i,\ r_i : R$, and then $\{r_i\} : \{R\}$. Finally $r : R$, because (by subject reduction on f_c):

$$\begin{array}{r} \text{APP-F} \\ \text{SR-F} \end{array} \ \frac{\begin{array}{c} f_c : \{R\} \rightarrow R \qquad \{r_i\} : \{R\} \\ \hline f_c(\{r_i\}) : R \end{array} \qquad f_c(\{r_i\}) \Downarrow r}{r : R}$$

Note that the typing rule for d&c also ensures that $f_b(p) : boolean$ (and thus $f_b(p)$ is necessarily true or false). □

5.2.5 Sub-typing

This section briefly discusses how sub-typing rules can be safely added to the type-system without major changes. Classically, if the underlying language supports sub-typing, see for example [Pie02], we allow the skeleton typing to reuse

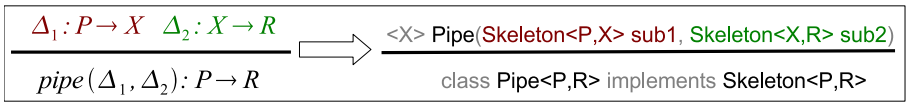


Figure 5.3: From theory to practice: *T-PIPE* rule.

the sub-typing relation of the language. Suppose a sub-typing relation ($\trianglelefteq$) is defined on the underlying language; then sub-typing can be raised to the skeleton language level as specified by the rule:

$$\text{T-SUB}\quad \frac{\triangle : P \to R \qquad P' \trianglelefteq P \qquad R \trianglelefteq R'}{\triangle : P' \to R'}$$

The subject-reduction and most classical sub-typing properties can be proved like in other languages.

5.3 Type safe skeletons in Java

In this section, we illustrate the type-system designed above in the context of a skeleton library implemented over the Java programming language. We show how the type-system of the Java language can be used at the skeleton level to check the type-safety of the skeleton composition. We have chosen Java for an implementation because Java provides a mechanism to communicate the type of an object to the compiler: *Generics*. More precisely, we use Java Generics to specify our skeleton API: constraints on type compatibility expressed as typing rules in Figure 5.2 are translated into the fact that, in the skeleton API, several parameters have the same (Generic) type.

Typing defined in Figure 5.2 is then ensured by the type system for Java and Generics. Using Generics, we do not need to implement an independent type system for algorithmic skeletons. Additionally, Generics provide an elegant way to blend the type system of the skeleton language with the type system of the underlying language. In other words, because skeletons interact with muscle functions, the proposed skeleton type system also interacts with the Java language type system.

In the Calcium skeleton framework, skeletons are represented by a corresponding class, and muscle functions are identified through interfaces. A muscle function must implement one of the following interfaces: `Execute`, `Condition`, `Divide`, or `Conquer`. Instantiation of a skeleton requires as parameters the corresponding muscle functions, and/or other already instantiated skeletons.

As discussed in section 5.2.3, the idea behind the type semantics is that, if it is possible to guarantee that the skeleton parameters have compatible types, then the skeleton instance will be correctly typed.

Therefore, since the skeleton program is defined during the construction of the skeleton objects, the proper place for performing type validation corresponds to the constructor methods of the skeleton classes.

```
1  Skeleton<A,B> stage1= new Seq<A,B>(new Execute1());
   Skeleton<B,C> stage2= new Seq<B,C>(new Execute2());
3  Skeleton<A,C> skeleton = new Pipe<A,C>(stage1, stage2); //type safe composition
   ------------------------------------------------------------------------------------

5  class Execute1 implements Execute<A,B>{   |  class Execute2 implements Execute<B,C>{
      public B exec(A param){                 |     public C exec(B param){
7        ...     /* No cast required */      |       ...     /* No cast required */
         return x; /* instanceof B */        |       return y; /* instanceof C */
9     }                                       |     }
   }                                          |  }
```

Listing 5.2: Example of a type safe skeleton program

Figure 5.3 shows the analogy between the type semantics and the Java implementation with generics for the *T-PIPE* rule. The premises of the typing rules are enforced in the signature of the skeleton constructor, and the conclusion of the typing is reflected on the signature of the skeleton class.

A skeleton class is now identified as a skeleton that receives a Generic parameter of type `<P>` and returns a parameter of type `<R>`, where `<P>` and `<R>` are specified by the programmer. Additionally, all parameters must be coherently typed among themselves, and with the skeleton. This type coherence will be specific for each skeleton, following the rules of the proposed typing system 5.2.3. The typing rules are enforced in Calcium as shown in Listing 5.3.

As a result, the unsafe skeleton composition shown in Listing 5.1 is transformed into the type safe skeleton program shown in Figure 5.2. The constructors of `Pipe` and `Seq` enforce that the return type of `Execute1.exec` must be the same type as the parameter of `Execute2.exec`: `B`. If this is the case, then the `Pipe` skeleton will be of type `<A,C>`, where `A` is the parameter type of `Execute1.exec` and `C` is the return type of `Execute2.exec`.

The benefits of using a type system for skeletons with Java generics are clear: no need to implement an additional type validation mechanism; no type-cast are imposed by the skeleton language inside muscle functions; and most importantly, type safe validation when composing the skeletons.

5.4 Conclusions

This chapter has defined a type system for algorithmic skeletons. We have tackled this problem from both a theoretical and a practical approach. On the theoretical side we have contributed by: formally specifying a type system for algorithmic skeletons, and proving that this type system guarantees that types are preserved by reduction. Type preservation guarantees that *skeletons can be used to transmit types between muscle functions*.

On the practical side, we have implemented the type system using Java and Generics. The type enforcements are ensured by the Java type system, and reflect the typing rules introduced in the theoretical section. Globally, this ensures the correct skeleton composition. As a result, no further type-casts are imposed by the skeleton language inside muscle functions; and most importantly, *type errors can be detected when composing the skeleton program*.

```
interface Execute<P,R> extends Muscle<P,R> {
    public R exec(P param);
}
interface Condition<P> extends Muscle<P,Boolean> {
  public boolean evalCondition(P param);
}
interface Divide<P,X> extends Muscle<P,X[]> {
  public X[] divide(P param);
}
interface Conquer<Y,R> extends Muscle<Y[],R> {
  public R conquer(Y[] param);
}

class Farm<P,R> implements Skeleton<P,R> {
  public Farm(Skeleton<P,R> child);
}
class Pipe<P,R> implements Skeleton<P,R> {
  public <X> Pipe(Skeleton<P,X> s1, Skeleton<X,R> s2);
}
class If<P,R> implements Skeleton<P,R> {
  public If(Condition<P> cond, Skeleton<P,R> ifsub, Skeleton<P,
     R> elsesub);
}
class Seq<P,R> implements Skeleton<P,R> {
  public Seq(Execute<P,R> secCode);
}
class While<P> implements Skeleton,P> {
  public While(Condition<P> cond, Skeleton<P,P> child);
}
class For<P> implements Skeleton<P,P> {
  public For(int times, Skeleton,P> sub);
}
class Map<P,R> implements Skeleton<P,R> {
  public <X,Y> Map(Divide<P,X> div, Skeleton<X,Y> sub, Conquer<
     Y,R> conq);
}
class Fork<P,R> implements Skeleton<P,R> {
  public <X,Y> Fork(Divide<P,X> div, Skeleton<X,Y>... args,
     Conquer<Y,R> conq);
}
class DaC<P,R> implements Skeleton<P,R> {
  public DaC(Divide<P,P> div, Condition<P> cond, Skeleton<P,R>
     sub, Conquer<R,R> conq);
}
```

Listing 5.3: Typed skeletons with Java Generics

Chapter 6

File Access Model

Contents

6.1 File Transfer with Active Objects 84
6.1.1 Asynchronous File Transfer with Futures 84
6.1.2 Push & Pull Benchmarks 86
6.1.3 Benchmarks Discussion 87
6.2 File Transfer Model for Skeletons 88
6.2.1 Transparency with FileProxy 88
6.2.2 Stage-in and Stage-out 89
6.2.3 The Workspace Abstraction 91
6.2.4 File References and Data 92
6.3 Efficiency 94
6.3.1 BLAST Case Study 95
6.4 Conclusion 96

Scientific and engineering applications that require, handle, and generate large amounts of data represent an important part of distributed applications. For example, some of the areas that require handling large amounts of data are: bioinformatics, high-energy physics, astronomy, etc.

The algorithmic skeleton model presented in Chapter 3 supposes that the data passed between muscles is small enough to be encapsulated inside a task's state memory. This is suitable for transferring small amounts of data between muscles, as would be done in regular non-parallel programming. Nevertheless, when the size of the data is too big to hold in runtime memory, non-parallel programming uses secondary memory storage abstraction: *files*.

Therefore, skeleton programming requires a mechanism that allows programmers to use their standard non-parallel way of reading/writing files inside muscles (non-invasive). Which, at the same time, does not force programmers to specify code for transferring files; i.e. enables transparent and efficient support for transferring files between the execution of muscles.

Nevertheless, there is a surprising lack of support for file data management in algorithmic skeleton programming models. The support of file data access

has been overlooked in most skeleton frameworks (see Section 2.3). Most of them could be enhanced with file data support by addressing file distribution aspects from inside muscles, as is the case with ASSIST. Unfortunately, this strategy leads to the tangling of non-functional code (data distribution) with the functional code (business logic).

Therefore, in this chapter we focus on the integration of data abstractions with an algorithmic skeletons programing model. We address the data problem by considering usability and performance from the programming model perspective. We believe that the integration of data files with algorithmic skeletons must be achieved in a transparent non-invasive manner, as not to tangle data distribution with functional concerns, while also taking efficiency into consideration.

Non-invasive transparency Programmers should not have to worry about data location, movement, or storage. Furthermore, programmers should not have to change their standard way of working with data. This means that transparency should be non-invasive, i.e. without imposing an ad hoc language nor library.

Efficiency is a double edged problem: computation and bandwidth. A suitable approach must balance the tradeoff between idle computation time, and bandwidth usage.

This chapter is organized as follows. We begin by describing how file transfer can be achieved with active objects in section 6.1. Afterwords section 6.2 presents the file data model for algorithmic skeletons. Section 6.3 studies efficiency concerns using BLAST as a case study. Finally, section 6.4 provides the conclusions and future work.

6.1 File Transfer with Active Objects

Before describing the file transfer model for skeletons, we begin by showing how file transfer can be achieved using an active object programming model. This is necessary for the ProActive execution environment, as ProActive did not provide file transfer tools.

This section shows how a message passing model, based on the active object concept, can be used as the ground for a portable and efficient file transfer service for large files, where large means bigger than available runtime memory. Additionally, by using active objects as transport layer for file transfer, file transfer operations can benefit from automatic continuation to improve the file transfer between peers, as can be confirmed by the benchmarks shown in section 6.1.2.

6.1.1 Asynchronous File Transfer with Futures

We have implemented file transfer between nodes as service methods available in the ProActive library, as shown in Figure 6.1. Given a ProActive **Node** *node*,

```
   //Send file(s) to Node node
2  static public Boolean pushFile(Node node, File source, File destination);
   static public Boolean[] pushFile(Node node, File[] source, File[] destination);
4
   //Get file(s) from Node node
6  static public File pullFile(Node node, File source, File destination);
   static public File[] pullFile(Node node, File[] source, File[] destination);
```

Listing 6.1: File Transfer API

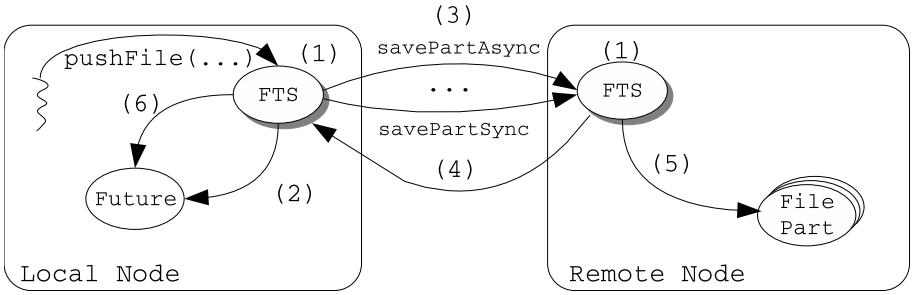


Figure 6.1: Push Algorithm.

a **File**(s) called *source*, and a **File**(s) called *destination*, the *source* can be pushed or pulled from a *node* using the API.

The *push* algorithm depicted in Figure 6.1 and detailed as follows:

1. Two File Transfer Service (FTS) active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The `pushFile(...)` function is invoked by the caller on the local FTS.
2. The local FTS immediately returns a Boolean type future to the caller. The calling thread can thus continue with its execution, and is subject to a wait-by-necessity on the future in case the file transfer has not yet been completed.
3. The file is read in parts by the local FTS, and up to $(o-1)$ simultaneous overlapping parts are sent from the local node to the remote node by internally invoking `savePartAsync(`p_i`)` on the remote FTS.
4. Then, a `savePartSync(`p_{i+o}`)` invocation is sent to synchronize the parameter burst, as not to drown the remote node. This will make the sender wait until all the parts $p_i, \ldots, p_{i+o}$ have been served (i.e. the `savePartSync` method is executed).
5. The `savePartSync(...)` and `savePartAsync(...)` invocations are served in FIFO order by the remote FTS. These methods will take the part p_i and save it on the disk.
6. When all parts have been sent or a failure is detected, the local FTS will update the future, created in step 2, with the result of the file transfer operation.

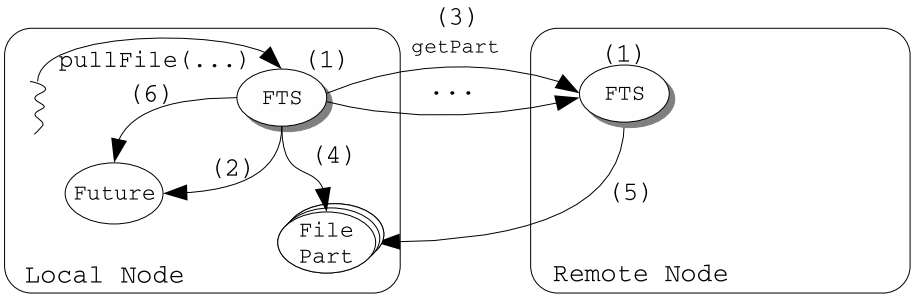


Figure 6.2: Pull Algorithm.

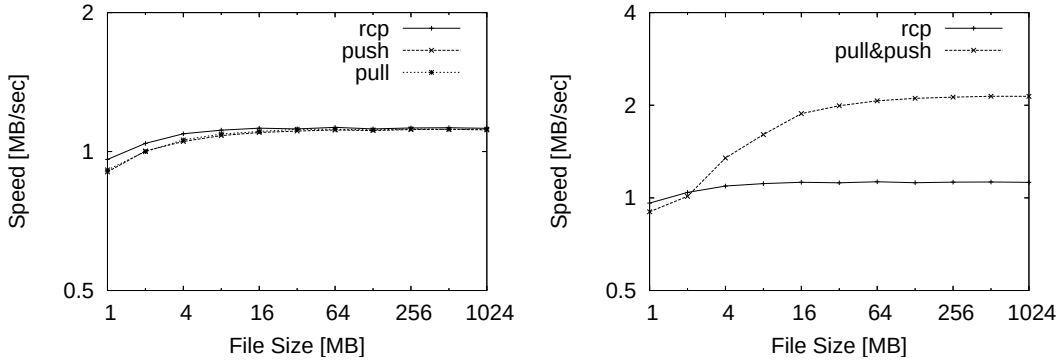
The `pullFile` method is implemented using the *pull algorithm* shown in Figure 6.2, and is detailed as follows:

1. Two FTS active objects are created (or obtained from a pool): a local FTS, and a remote FTS. The `pullFile(...)` function is invoked by the caller on the local FTS.
2. The local FTS immediately returns a File future, which corresponds to the requested file. The calling thread can thus continue with its execution and is subject to a wait-by-necessity on the future.
3. The `getPart(`i`)` method is internally invoked up to o overlapping times on the remote FTS.
4. The local FTS will immediately create a future of a type representing a *file part* for every invoked `getPart(`i`)`.
5. The `getPart(...)` invocations are served in FIFO order by the remote FTS. The function `getPart` consists on reading a file part on the remote node, and as such, automatically updating the corresponding local future created in step 4.
6. When all parts have been transferred, then the local FTS will update the future created in step 2, unblocking any thread that was subject to a wait-by-necessity on the future.

6.1.2 Push & Pull Benchmarks

A 100$[Mbit]$ LAN network with a 0.25$[ms]$ ping, and our laboratory desktop Intel Pentium 4 (3.60$[GHz]$) machines were used. It was experimentally determined that overlapping 8 parts of size 256$[KB]$ provides a good performance and guarantees that at the most 2$[MB]$ will be enqueued in the remote node. Since the aim of the experiment was to evaluate the proposed file transfer model, and not the lower level communication protocols between active objects, the default protocol used for lower level communication was `RMI`.

Since peers usually have independent download and upload channels, the network was configured at 10$[\frac{Mbits}{sec}]$ duplex. Figure 6.4(a) shows the performance results of `pull`, `push`, and *remote copy protocol* (`rcp`) for different file sizes.



(a) Push, pull, and rcp speed.

(b) Pushing while pulling a file.

Figure 6.3: Performance comparisons.

The performance achieved by `pull` and `push` approaches `rcp`, which can be considered an ideal reference because of its wide adoption and usage.

More interestingly, Figure 6.4(b) shows the performance of getting a file from a remote site, and then sending this file to a new site. This corresponds to a recurrent scenario in data sharing peer to peer networks [Ora01], where a file can be obtained from a peer instead of the original source.

6.1.3 Benchmarks Discussion

From Figure 6.4(a) we can see that `pull` and `push` perform as good as `rcp` in the general file transfer case. Additionally, in the specific case of sharing files between peers, Figure 6.4(b) shows that `rcp` is outperformed when using `pull` and `push` algorithms. While `rcp` must wait for the complete file to arrive before sending it to a peer, the `pull` algorithm can pass the future file parts (Figure 6.2) to the `push` algorithm even before the actual data is received. Once the future of the file parts are available, automatic continuation [CH05, EAC98] will take care of transparently updating the parts that had been "virtually" transferred to the concerned peers. The user can achieve this automatic continuation-based behavior with the API shown in Figure 6.1, by simply passing the result of an invocation as parameter to another.

Therefore, the proposed file transfer model during the application execution achieves improved usability by integrating the abstractions of the programming model into the file transfer operations. It also simplifies the infrastructure maintenance and configuration effort by construction the file transfer on top of message passing protocol; and generally provides as good, and in some scenarios better performance than third party tools.

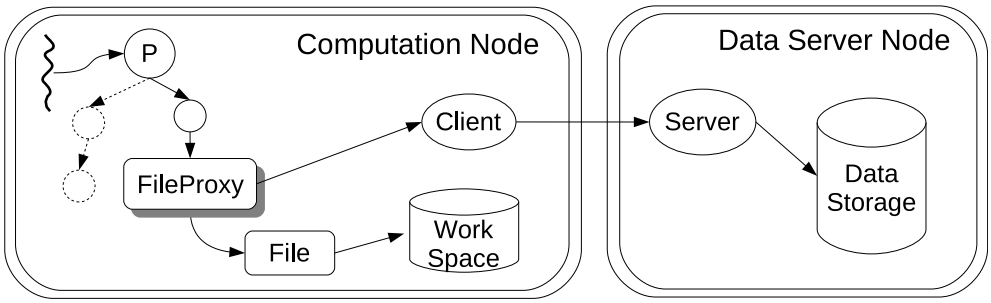


Figure 6.4: Proxy Pattern for Files

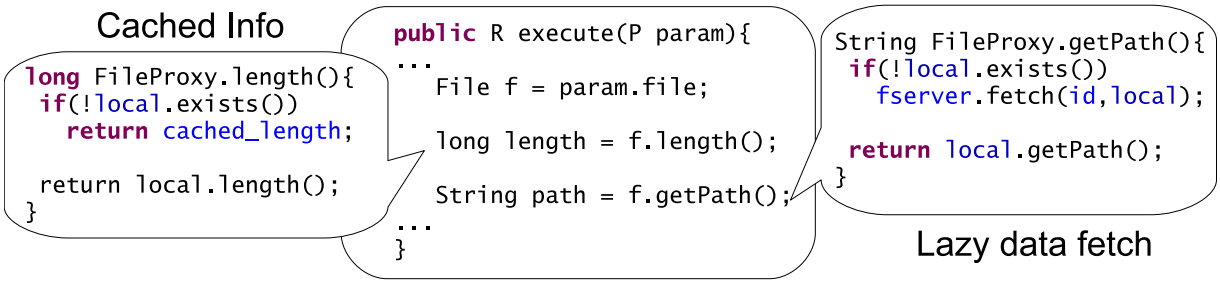


Figure 6.5: FileProxy Behavior Example

6.2 File Transfer Model for Skeletons

Now that we have clearly shown that file transfer operations are possible with an active object programming model, we continue by discussing the proposed file transfer model for algorithmic skeletons.

6.2.1 Transparency with FileProxy

The Proxy Pattern [GHJV95] is used to achieve transparent access to files. Files are rendered accessible using the `FileProxy` object as shown in Figure 6.4. By intercepting calls at the proxy level, the framework is able to determine when a muscle is accessing a file. In a way, the `FileProxy` illuminates a specific aspect inside black box muscles.

Figure 6.5 provides an example. When an interpreter thread invokes a muscle, all `File` type references inside `param` are indeed `FileProxy` instances. A `FileProxy` can transparently intercept a thread's access to the actual `File` object. A `FileProxy` can add new non-functional behavior such as caching of metadata (file names, size, etc...), transparent file fetching on demand, and blocking on a wait-by-necessity style [Car93]. Afterwards, the `FileProxy` can resume the thread's execution by delegating method calls to the *real* `File` object.

6.2.2 Stage-in and Stage-out

Listing 6.2 provides an example on the usage of Calcium. Line 1 defines the skeleton pattern, and is omitted here but detailed in Figure 6.7(a). Lines 3-4 instantiates an execution environment, which in this case corresponds to a `ProActiveEnvironment`, and creates the `Calcium` instance. The boot and shutdown of the framework are done in lines 6 and 23 respectively. Then in lines 8-9, a new input `Stream` is created with the blast skeleton pattern. Lines 12-15 illustrate the creation of a new `BlastQuery` paremeter, which receives three `File` type arguments: blast binary, query, and database files on the client machine.

The interesting part takes place in line 17. The `BlastQuery` is entered into the framework, and a `Future<File>` is created to hold the result once it is available. During the `input` process each file's data is remotely stored; and all `File` type objects are replaced by `FileProxy` instances, capable of fetching the data when required by remote nodes during the computation. Once the result is available, and before unblocking threads waiting on line 21, all remotely stored data referenced by `FileProxy` instances are copied to the client machine, and all `FileProxy` instances are replaced with regular `File` type instances. Hence, the result in line 21 is a regular `File` with its data stored on the client machine.

6.2.2.1 Initial and Final Staging

In general, when a parameter P is submitted into the skeleton framework, as shown in Listing 6.2 (line 17), a `File` stage-in takes place. First, all references of type `File` in P's object graph are replaced with `FileProxy` references. Then, the files' data are stored in the data server. If a name clash occurs or a data transfer error takes place, an exception is immediately raised to the user, before the parameter is actually entered into the skeleton framework.

When the final result R has been computed, but before it is returned, a stage-out process takes place. Every reference of type `FileProxy` in R's object graph is replaced by a regular `File` type pointing to a local file, and the remote data is stored in the local file, before returning R to the user.

6.2.2.2 Intermediate Staging

Before an interpreter invokes a muscle, a staging process takes place on the interpreter nodes. If not already present, a unique and independent workspace is created. Then, depending on the desired behavior (see section 6.3) all, some, or none of the `FileProxy` type objects in P's object graph are downloaded into the workspace, and the `FileProxy` references are updated with the new location of the file.

After the invocation of a muscle, new files referenced in R's object graph, and present in the workspace, are stored on the data server. Actually, files are only stored on the data server if the file reference is passed on to other tasks, i.e. returned to the task pool. Further details of how the references are updated are discussed in Section 6.2.4.

```
1  Skeleton skel = ...;

3  Environment env = new ProActiveEnvironment(...);
   Calcium calcium = new Calcium(env);
5
   calcium.boot();
7
    Stream<BlastQuery,File> stream =
9                    calcium.newStream(skel);

11   //Initial stage-in
     BlastQuery blast = new BlastQuery(
13                   new File("/home/user/blast.bin"),
                     new File("/home/user/query.dat"),
15                   new File("/home/user/db.dat"));

17   Future<File> future = stream.input(blast);

19   ...
     //Final stage-out, the file is locally available
21   File result = future.get();

23 calcium.shutdown();
```

Listing 6.2: Calcium Input and Output Example

```
   public File execute(WSpace wspace, BlastQuery blast){
2
     //Get parameters
4    File command = blast.blastProg;
     String arguments = blast.getArguments();
6
     //Execute the native blast in the wspace
8    wspace.exec(command, arguments);

10   //Create a reference to a file in the wspace
     File result = wspace.newFile("result.blast");
12
     return result;
14 }
```

Listing 6.3: Muscle Function Example

6.2.3 The Workspace Abstraction

The workspace abstraction provides muscles with a local disk space on the computation nodes. If several muscles are executed simultaneously on the same interpreter node, each muscle is guaranteed to have its own independent workspace.

The workspace abstraction provides the following methods:

```
interface WSpace{
  public File newFile(String name);
  public void exec(File bin, String args);
}
```

Where the `WSpace.newFile()` factory can be used to create a file reference on the workspace, and `WSpace.exec(...)` can be used to execute a native command with a properly configured execution environment (e.g. current working directory).

Listing 6.3 provides an example. A muscle of type f_e : `BlastQuery` → `File` is shown. Lines 4-5 get a reference on the native `command` and its `arguments`. For the programmer, `command` is of type `File`, but is indeed a `FileProxy` instance. The `command`'s data was stored somewhere else during the computation (Listing 6.2 line 17), and is transparently made available on the interpreter node. Line 8 invokes the native blast command which outputs its results to a file named `result.blast`, located in some directory, specified by the workspace, on the interpreter node. Then line 11 uses the workspace factory to get a reference on the `result.blast` file. The workspace factory returns a reference object of type `File` which is indeed an instance of type `FileProxy`. Finally, line 13 returns the `File` object as a result. If the result is passed to another computation node, or delivered as final result to the user, then the file will be transparently transferred.

An alternative approach to providing a workspace factory method would have been to use other aspect-oriented programming [KLM+97] methodologies that, for example, manipulate Java bytecode to intercept calls on the `File` class con-

structor. Nevertheless, as noted by Cohen et al. [CG07], factories provide several benefits over traditional constructor anomalies.

After a `File` reference is created through the workspace abstraction, the framework transparently handles reference passing; creation, modification and deletion of file's data; and remote data storage/fetching.

6.2.3.1 Data Division

When data parallelism is encountered, such as in $\{d\&c, map, fork\}$ skeletons, new sub-tasks are spawned and assigned with a new workspace.

Instead of copying all of the original workspace's files into each sub-task's workspace, only referenced files are copied. For example, if the muscle $f_d : P \rightarrow \{R\}$ assigns at some point

$$R_i.file \leftarrow P.file_1$$
$$R_j.file \leftarrow P.file_2$$

then only $file_1$ will be copied into R_i's workspace, while $file_2$ will be copied into R_j's workspace.

The advantage of this approach is that the mapping of files with workspaces is transparent for the programmer. Contrary to what happens on workflow environments (see Section 2.1.6), there is no need for the programmer to explicitly map which files are copied into which workspace. This is automatically inferred from the `FileProxy` references.

6.2.3.2 Data Reduction

The symmetrical case is the reduction (conquer) case, where several sub-tasks are reduced into a single one. This is done with a muscle of type $f_c : \{P\} \rightarrow R$, which takes n object elements and returns a single one.

Before invoking the conquer muscle, a new workspace is created, and all the files referenced in $\{P\}$ are copied into the new workspace. Nevertheless, a name space clash is likely to happen when two files originating from different workspaces have the same name.

A simple solution is to have a renaming function which provides a unique name when a name clash is detected. The clashing file is then renamed, and the `FileProxy` reference is transparently updated with the new name. While this solution can yield unexpected file renaming behavior for the programmer, no problems will be encountered as long as the programmer consistently uses the `File` references.

6.2.4 File References and Data

6.2.4.1 Storage Server

We assume the existence of a data storage server[1], capable of storing data, retrieving data, and keeping track on the reference count of each data. The storage

[1]For an example of a scalable data storage system refer to [AADJ07].

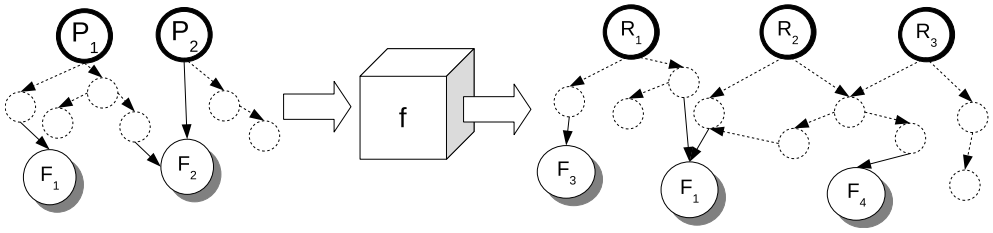


Figure 6.6: File Reference Passing Example

server provides the following operations:

- $store(F_x, k) \rightarrow i$, stores the data represented in file F_x, with an initial reference count $k > 0$. The function returns a unique identifier for this data on the storage server.
- $retrieve(i) \rightarrow F_x$, retrieves data located on the server and identified by i.
- $add(i, \delta) \rightarrow boolean$, updates a reference count by δ. Returns $true$ if the reference count is equal to or smaller than zero, and $false$ otherwise.

Once the reference count reaches zero for a file's data, no further operations will be performed on the data, and the server may delete the data at its own discretion.

6.2.4.2 Reference Counting

During the execution of a skeleton program, data can be created, modified, and deleted. Also, `File` references pointing to data can be created, deleted, and passed (copied). Therefore, it is up to the skeleton framework to provide support for these behaviors, by storing new/modified data; and keeping track of `File` references to delete data when it is no longer accessible.

Consider the example shown in Figure 6.6 where P_1, P_2 are input parameters of a muscle $f : \{P\} \rightarrow \{R\}$; R_1, R_2, R_3 are the output results; and F_i are `FileProxy` references. We are interested on knowing, for a given F_i, how many P_j/R_k have a directed path from P_j/R_k to F_i, before/after the execution. We call this the reference count, and we write it as [before,after].

In the example, the reference counts are:

$$F_1 \rightarrow [1, 3] \qquad F_2 \rightarrow [2, 0]$$
$$F_3 \rightarrow [0, 1] \qquad F_4 \rightarrow [0, 2]$$

Thus we know that F_1 has incremented its reference count by 2; F_2 is no longer referenced and has decreased its reference count by 2; and F_3, F_4 are new files created inside f.

Case	[Before,After]	Action
New	$[b = 0, a > 0]$	store(F_x, a) $\rightarrow i$
Read-only	$[b > 0, a > 0]$	add(i, $a - b$)
Modified		add(i, $-b$) store(F_x, a) $\rightarrow j$
Dereferenced	$[b > 0, a = 0]$	add(i,$-b$)
Unreferenced	$[b = 0, a = 0]$	———

Table 6.1: File Scenarios after muscle invocation

6.2.4.3 Update Cases

In general, after invoking a muscle f, a file F_x can be in one of the cases shown in Table 6.1.

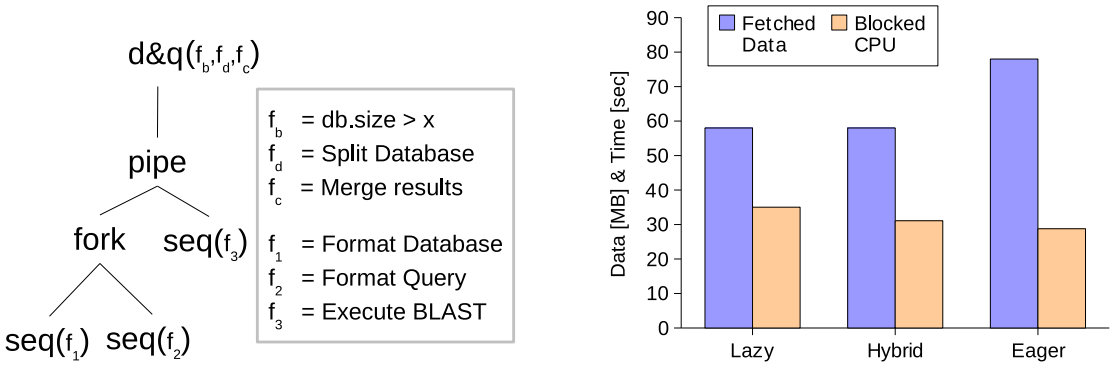
Where the cases are described as follows:

- **New** files are created during the execution of f. A new file's data is uploaded to the storage server with its after reference count by invoking $store(F_x, k) \rightarrow i$, with $k = a$.
- **Read-only** files only require an update on their reference count, since data has not been modified. This is done by invoking $add(i, \delta)$ with $\delta = b - a$.
- **Modified** files have been modified during the execution of f. Conceptually, modified files are treated as new files. Therefore if i is the identifier of the original file on the storage server, then $add(i, \delta)$ with $\delta = -b$ is invoked to discount the before references on the original file. Then, the modified file is treated as a new one, by uploading its data to the storage server and obtaining a new file identifier: $store(F_x, k) \rightarrow j$ with $k = a$.
- **Dereferenced** files have no references after the execution of f, and therefore it is irrelevant if the file was modified during the execution. Thus they only require a $add(\delta)$ on the server, with $\delta = -b$.
- **Unreferenced** files are temporary files used inside f, and can be locally deleted from the workspace after the execution of f.

6.3 Efficiency

An efficient approach minimizes both bandwidth usage and CPU idle time (blocked waiting for data). To minimize the CPU idle time, a file's data should already be locally available when a muscle wants to access it. On the other hand, to minimize the bandwidth usage, a file's data should only be transferred if it is going to be used by the muscle. This presents a problem since muscles are black boxes.

We suppose a three staged pipeline on each interpreters where: the first stage is the *prefetch*, which downloads candidate files in advance; the *compute* stage invokes the muscles; and the *store* stage uploads files' data to the storage server.



(a) BLAST Skeleton Program

(b) Bandwidth / CPU Tradeoff

Figure 6.7: BLAST Case Study

Thus, in any given moment, three tasks can be present on an interpreter pipeline performing different aspects: download, computation, and upload.

Two strategies are identified. A **lazy** strategy which transfers a file's data on demand using the `FileProxy` (bandwidth friendly), and an **eager** strategy which transfers all the files' data in advance (CPU friendly) using the interpreter pipeline. Additionally, we propose a third **hybrid** strategy which uses annotated muscles to decide which file's data to transfer in advance.

For example, a muscle can be annotated to prefetch files matching a regular expression pattern or files bigger/smaller than a specified size:

```
@PrefetchFilesMatching(name="db.*|query.*",
                       sizeMin=10000,
                       sizeMax=20000)
public File execute(WSpace wspace, BlastQuery param){
  ...
}
```

While the separation of concerns is kept using the proposed annotation, one may argue that the transparency of the approach is hindered. Nevertheless, it is important to emphasize that the annotation is not a file transfer definition (source and destination are not specified), and as such does not fall back into the non-transparent case. Furthermore, the presence of the annotation is not mandatory, being its only objective the improvement of performance.

6.3.1 BLAST Case Study

BLAST [BLA] corresponds to Basic Local Alignment Search Tool. It is a popular tool used in bioinformatics to perform sequence alignment of DNA and proteins. In short, BLAST reads a query file and performs an alignment of this query against a database file. The results of the alignment are then stored in an output file. BLAST is a good case study because it performs intensive data access, computation, and requires the execution of native code.

A BLAST parallelization using skeleton programming is shown in Figure

6.7(a). The strategy is to divide the database until a suitable size is reached and then merge the results of the BLAST alignment. The result of applying lazy, hybrid and eager strategies are shown in Figure 6.7(b). The figure shows that a lazy strategy performs the least amount of data transfers, but blocks the application for the longest time waiting for the data. On the other hand, an eager strategy performs the most data transfer, blocking the application for the least time.

For BLAST, a good tradeoff can be reached using the proposed hybrid strategy, which can transfer as few data bytes as the lazy strategy, and block the application at least as the eager strategy. In general, the performance of the hybrid strategy may vary, depending on the application, but the hybrid strategy's performance is bounded by the lazy and eager strategies.

6.4 Conclusion

This chapter has proposed a file data access model for algorithmic skeletons by focusing on transparency and efficiency.

Transparency is achieved using a workspace abstraction and the Proxy pattern. A `FileProxy` type intercepts calls on the real `File` type objects, providing transparent access to the workspace. Thus allowing programmers to continue using their accustomed programming libraries, without having the burden of explicitly introducing non-functional code to deal with the distribution aspects of their data.

From the efficiency perspective we have proposed a hybrid approach that takes advantage of annotated muscle functions and pipelined interpreters to transfer files in advance, but can also transfer the file's data on-demand using the `FileProxy`. We have experimentally shown with a BLAST skeleton, that a hybrid approach provides a good tradeoff between bandwidth usage and CPU idle time.

Chapter 7

Perspectives and Conclusions

Contents

7.1 Perspectives 97
7.1.1 Performance Tuning Extensions 97
7.1.2 Skeleton Stateness 98
7.1.3 AOP for Skeletons 98
7.2 Conclusion 99

7.1 Perspectives

We believe that research around algorithmic skeletons is a fertile area with many possible directions. Nevertheless, we restrict this section to research directions which we believe are aligned with the underlying goal of this thesis: the adoption of skeleton libraries as a mainstream parallel programming model.

7.1.1 Performance Tuning Extensions

The performance tuning model introduced in Chapter 4 opens further research perspectives. So far the choice of metrics has been somewhat arbitrary, and we believe that other metrics may provide new insights on the execution of the program. Also, the inference tree, used to derive a performance inefficiency cause, can be further refined to provide newer and more precise causes. For example, the current inference tree is very computation oriented. Nevertheless, with the addition of file transfer support for skeletons presented in Chapter 6, data distribution will have a significant influence on performance inefficiencies. Additionally, while the rudimentary blaming mechanism we presented was suitable for our testcase, we believe that there is still significant room for improvement.

Another research perspective is that the performance tuning methodology is postmortem. The analysis is performed after the execution of the application. We believe that the performance tuning model would be a good starting point to explore autonomic behavior of the skeleton program. Autonomicity would allow

dynamic tuning of the skeleton applications while being executed. Indeed, we can expect that some performance inefficiency causes may be dealt with automatically, while others are reported back to the user.

7.1.2 Skeleton Stateness

One of the hypothesis in this thesis corresponds to having stateless algorithmic skeletons, as defined in Chapter 3, which in turn implies that muscles are also stateless functions. If we recall, skeletons were conceived from the domain of functional programming where this kind of reasoning seems natural and acceptable to achieve higher degrees of concurrency.

Indeed, while stateless muscles is a somewhat reasonable hypothesis in functional programming, this is very unnatural when reasoning on an imperative, and more precisely, in an object oriented programming paradigm. For non-parallel programmers, it is intuitive to reason about algorithms where objects are shared (referenced) by different parts of the program. These programmers are thus bewildered by the fact that object sharing between different parts of a skeleton program is not supported. Something as simple as counting the number of times a muscle f has been invoked is not possible with stateless skeletons. The sequential way of solving this problem is very simple: with a counter variable (i) and having f increment its value every time it is invoked ($i \leftarrow i + 1$).

The simplest way to deal with a stateful muscles is to sequentialize all access to it. The danger with this approach is that the lock is too coarse grain. All concurrent access to a muscle function are serialized, and thus parallelism gained from the skeleton pattern may be lost. Consider for example the nesting of a stateful muscle f inside a $farm$ skeleton: $farm(f)$. A correctly written sequential muscle f is now incorrect because concurrent evaluations of f can coexist, all of them competing for the same variable i. The worst solution to this problem is acquiring a lock before the evaluation of f and releasing it once f is done, since this sequentalizes the invocations on f and cancells out the parallelism introduced by the $farm$ skeleton.

A better approach is to reduce the span of the lock to the access on i. This requires to write f differently, since now the programmer must be aware that the evaluations of f can occur concurrently. In addition the programmer must have access to some locking mechanism, which up to now was unnecessary. Unfortunately, this approach brings back the complexities of regular parallel programming to skeletons, which defeats the purpose of introducing skeletons: providing higher abstractions for parallel programming.

7.1.3 AOP for Skeletons

Readers familiar with Aspect-Oriented Programming (AOP) [KLM+97] will have noticed that many of the techniques used throughout this thesis, in particular in Chapter 6, resemble those of AOP.

Indeed, the idea of weaving non-functional aspects using inheritance [CEF02], in the same way that the `FileProxy` abstraction has been used to intercept calls

on `File` type objects is not new. The dilemma of instantiating aspect augmented objects has been addressed in AOP using factories [CG07] in similar fashion as the workspace abstraction factory introduced in Section 6.2.3. And the transformation of `File` → `FileProxy` → `File`, can be framed in the domain of dynamic aspects [Tan07] and object reclassification [DDDCG01, DDDCG02].

From the AOP perspective, Chapter 6 has provided a specific methodology for weaving file transfer aspects with algorithmic skeletons, and as such has shown that AOP like methodologies can be applied to algorithmic skeletons. More generally, the integration of AOP with distributed programming has already been proposed for other middlewares such as JAC[PSD+04], J2EE [CG04], ReflexD [TT06], etc.

Therefore, as [MAD07], we believe that the integration of AOP with algorithmic skeleton is a promissing mechanism to support other non-functional aspects in skeleton programming.

As future work, we would like to generalize the methodologies presented in this thesis to support other non-functional aspects in algorithmic skeletons. Indeed, our goal is to provide an AOP model for algorithmic skeleton, which will allow a tailored integration of other non-functional aspects into skeletons, such as stateful muscles, statistics gathering, event dispatching, etc.

7.2 Conclusion

The complexity and difficulties of parallel programming have led to the development of many parallel and distributed programming models, each having its particular strengths. Only a few of these models have been embraced as mainstream, while most remain confined in niches.

In this thesis, we have focused on one of this niche programming models: algorithmic skeletons, which exploits concurrency through recurrent parallelism patterns. The ambition begind this thesis is to transform algorithmic skeletons, from a niche, into a mainstream parallel programming model. The idea behind skeletons is to factor out recurrent parallel parts of applications and provide a customizable way of using the pattern: a skeleton. The skeleton pattern implicitly defines the parallelization and distribution aspects, while sequential blocks written by programmers provide the application's functional aspects (i.e. business code).

We have proposed a model for algorithmic skeleton programming and implemented this model in Java as the Calcium library. Table 7.1 shows a profile of Calcium's characteristics as portrayed for other skeleton frameworks in Section 2.3.

The table confirms that Calcium's combination of characteristics is unique among other known skeleton frameworks. To summarize, the contributions presented in this thesis are listed as follows:

- A survey on the state of the art of algorithmic skeleton programming.
- Calcium, a model and a framework for algorithmic skeleton programming in Java featuring:

	Calcium
Activity Years	2006-2008
Programming Language	Java
Execution Language	Java
Distribution Library	Threads & ProActive Active Objects & ProActive Scheduler
Type Safe	yes
Skeleton Nesting	yes
File Access	transparent
Skeleton Set	farm, pipe, seq, if, for, while, d&c, map, fork

Table 7.1: Calcium Summary

  - A library for skeleton programming, with
  - Nestable task and data parallel skeletons, and
  - Parallel and distributed execution on multiple environments.
- A performance tuning model for algorithmic skeletons and its implementation in Calcium.
- A type system for nestable algorithmic skeleton and its implementation using Java Generics in Calcium.
- A File access/transfer support with:
  - An active object based file transfer model and its implementation in ProActive.
  - A transparent file access model for algorithmic skeletons and its implementation in Calcium.

During this thesis we have made an effort to address issues both from theoretical and practical perspectives. Therefore, most of our results are independent from our implementation, while at the same time Calcium is available for practical experiences.

While we recognize the need for further research and work on the domain, it is our hope that the issues we have addressed during this thesis will contribute to help programmers embrace algorithmic skeletons as a main stream programming model.

Part II

Résumé étendu en français

(Extended french abstract)

Chapter 8

Introduction

De nos jours, la programmation parallèles s'est imposée comme une évidence. Jamais existé nous avons eu plus de besoin des modèles de programmation parallèle pour exploiter la puissance, de plus en plus complexe, des systèmes parallèles [Yel08]. D'un côté, des systèmes informatiques a mémoire distribuée à grande échelle tels que le *cluster* et le *grid computing* [FK99] et, de l'autre des systèmes informatiques parallèles à mémoire partagée grâce à de nouveaux processeurs multi-core [ABC+06].

Les difficultés de la programmation parallèle ont conduit au développement de nombreux modèles de programmation parallèle, chacun ayant ses points forts. Une chose qu'ils ont en commun est que les modèles de programmation parallèle cherchent un équilibre entre la simplicité (abstractions) et l'expressivité (détails), mesuré par la performance.

Néanmoins, de tous les modèles de programmation, seuls quelques-uns ont été accueillis comme dominants, alors que la plupart restent confinés dans des créneaux. Dans cette thèse nous nous traitons à un tel modèle: les squelettes algorithmiques.

8.1 Problematic

Comme l'a reconnu dans le manifeste Cole [Col04], les squelettes algorithmique offre *simplicité*, *portabilité*, *réutilisation*, *performances*, et *optimisation*, mais n'ont pas encore pas réussi à parvenir à intégrer dans la programmation parallèle. Avec cet objectif en tête, Cole du manifeste a proposé quatre grands principes pour guider le développement de systèmes des squelettes algorithmiques: *minimal conceptual disruption*, *integrate ad-hoc parallelism*, *accommodate diversity*, and *show the pay-back*.

Si nous regardons l'évolution de la programmation des squelettes algorithmiques, nous pouvons voir qu'il a beaucoup varié depuis son invention par Cole [Col91]. Peut-être que le changement le plus significatif a été de réaliser que les squelettes algorithmiques devraient être fournis comme des bibliothèques au lieu de langages. En effet, comme on le verra plus tard dans cette thèse, la plupart des systèmes de squelettes ont été fournis comme des bibliothèques en

langages orientés objet. Ceci est en accord avec le premier principe du manifeste de Cole: *minimal conceptual disruption*.

Les implications d'avoir des squelettes fournis comme des bibliothèques a changé la façon dont nous envisageons la conception et la mise en œuvre des systèmes de squelettes, et surtout comment le programmeur interagit avec les systèmes.

Nous pensons qu'en fournissant des squelettes algorithmiques comme des bibliothèques, les systèmes de squelettes n'ont plus à fournir de support pour tous les types d'applications parallèles, mais peuvent se concentrer sur le premier et quatrième principes de Cole: *minimal conceptual disruption* et *showing the pay-back*. L'hypothèse est que les programmeurs utiliseront les bibliothèques des squelettes algorithmiques dans leur specialité pour ce dans quoi ils sont bons: la programmation parallèle structurée, alors qu'ils choisiront un autre modèle de programmation parallèle pour un domaine dans le quel ils ne sont pas bons: applications parallèles irrégulières. Ainsi, pour des applications complexes, un mélange de bibliothèques, chacun implementant un modèle de programmation parallèle different, seront probablement utilisés.

Plusieurs ouvrages dans la littérature ont affirmé que les modèles de coût sont une force de la programmation par des squelettes algorithmiques. Bien que cette propriété soit bonne, d'autres modèles de programmation tels que MPI (qui n'ont pas de modèles de coûts) se sont multipliés, et sont le utilisés *de facto* pour la programmation distribuée. En effet, nous pensons que les modèles de coûts ne vont pas être la caractéristique qui penchera la balance en faveur des squelettes algorithmiques.

Par conséquent, dans cette thèse nous poursuivrons d'autres caractéristiques qui pourraient aider les programmeurs à adopter la programmation avec des squelette algorithmiques. Les caracteristiques sont la mise au point de la performance, le systeme de typage et accès aux fichiers et les transferts transparents.

8.1.1 Objectifs et Contributions

L'objectif principal de cette thèse est la conception et la mise en œuvre d'une bibliothèque de squelettes algorithmiques capable d'exécuter des applications en parallèle et distribué sur plusiers infrastructures.

Les principales contributions de cette thèse sont les suivantes:

- Une enquête sur l'état de l'art de la programmation par squelettes algorithmiques.
- Un modèle de programmation des squelettes algorithmiques et son implémentation en Java, au sein de la une bibliothèque Calcium, comportant: des squelettes composables parallèlisables sur les tâches et les données, et l'exécution dans de multiples environnements parallèles et distribués.
- Un modèle d'ammélioration de la performance pour les squelettes algorithmiques, et son implémentation dans Calcium [CL07]

- Un système de typage pour squelettes algorithmiques composables et sa implémentation en profitant de Java Generics dans Calcium [CHL08].
- L'accès au transfert de fichier transparent. Tout d'abord, un modèle basé sur des objets actifs et son implémentation ProActive [BCLQ06, BCL07]. Deuxièmement, un modèle transparent d'accès aux fichiers pour les squelettes algorithmiques et son implémentation dans Calcium [CL08].

8.2 Présentation

Ce document est organisé comme suit:

- Le Chapitre 2 fournit un état de l'art des squelettes algorithmiques. Le chapitre commence par une description de modèles bien-connus pour la programmation parallèle et distribuée. Ensuite, le chapitre fournit une description de plusieurs systèmes des squelettes algorithmiques. Les descriptions essayent de fournir un bref résumé et de distinguer les principales caractéristiques de chaque système. Le systèmes des squelettes sont également comparés et le travail dans cette thèse est positionné par rapport à l'état de l'art.
- Le Chapitre 3 fournit une introduction et une description du système proposé: Calcium. Le chapitre commence par une description des squelettes fournis dans Calcium. Ensuite, le chapitre décrit l'hypothèse sur laquelle le modèle de programmation par squelettes de Calcium est construite. Le chapitre se poursuit avec la formalisation du modèle de programmation qui montre comment le parallélisme est atteint. Ensuite, le chapitre décrit comment ce modèle est implémenté en Java, et aussi comment le système peut supporter de multiples environnements d'exécution. Le chapitre se termine par un exemple concret: une solution naïve pour trouver des nombres premiers.
- Le Chapitre 4 présente un modèle d'amélioration de performances pour les squelettes algorithmiques. Ce chapitre vise à rapprocher le *performance debugging* au niveau d'abstraction de squelettes algorithmiques. Pour cela, des mesures de performance sont recueillies et un arbre d'inférence est utilisé pour trouver la cause possible du *performances bug*. La cause est alors liée au squelette et au code qui sont soupçonnés de causer l'exécution du *performances bug*. Validations expérimentales sont fait avec un application des squelettes pour le problème des NQueens.
- Le Chapitre 5 définit un système de type théorique pour les squelettes et prouve que ce type de système est *safe* car il garantit la propriété de *subject reduction*. Le chapitre aborde ensuite l'implèmentation d'un tel système de type en Java en utilisant des *Generics*.
- Le Chapitre 6 définit un modèle transparent et non-invasif pour l'accès et le transfert des fichiers pour les squelettes algorithmiques. Le modèle

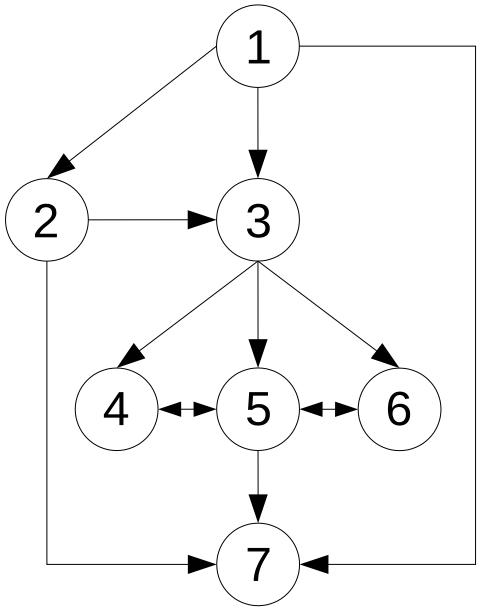


Figure 8.1: Itinéraires de Lecture Suggérés

est implèmenté en Calcium et des expérimenttations sont faites avec une application BLAST basée sur des squelettes.

- Le Chapitre 7 conclut cette thèse en fournissant des perspectives de recherche à venir et en résumant les contributions.

8.3 Itinéraires de Lecture Suggérés

Cette thèse peut être lue de plusieurs manières selon les préférences du lecteur. Les propositions d'itinéraires sont présentées dans la Figure 8.1 et se détaillent comme suit:

- **Résumé de la Contribution**. Cela correspond à l'introduction et la conclusion de la thèse qui donnent un aperçu des principales contributions. L'itinéraire proposé est: 1→7.
- **Résumé de la Contribution et Contexte** Met en évidence les principales contributions et le contexte de cette thèse.
  L'itinéraire proposé est: 1→2→7.
- **Cœur de la Contribution** Met en évidence la principale contribution de cette thèse en profondeur. L'ordre dans lequel les chapitres 4, 5, et 6 sont lus est indifférent.
  L'itinéraire proposé est: 3→ (4 | 5 | 6)
- **Complet** C'est la lecture complète de cette thèse.
  L'itinéraire proposé est: 1→2→ 3→ (4 | 5 | 6) → 7.

Chapter 9

Résumé

Calcium prévoit trois caractéristiques principales qui élargissent le modèle de programmation des squelettes algorithmiques: réglage des performances [CL07], un système de type [CHL08], et un modèle d'accès aux fichiers transparent [CL08]. Cette chaptire les décrit brièvement. Les lecteurs intéressés devraient se référer à leurs références pour de plus amples détails.

9.1 Réglage du Performance

Les squelettes sont considérés comme un paradigme de programmation de niveau élevé parce que les détails de niveau inférieur sont cachés du programmeur. La réalisation de haute performance pour une application est la responsabilité du système des squelettes en effectuant des optimisations sur le structure de le squelettes [AD99, ADD04], et l'adaptation dynamique à l'environnement [Dan05]. Toutefois, si ces techniques sont connues pour améliorer les performances, par eux-mêmes, ils ne sont pas suffisantes. Les aspects fonctionnels de la demande (c'est-à-dire le muscle), qui sont fournis par le programmeur, peut être inefficace ou générer des dégradations de performance dans le système.

Les objectifs principaux du réglage des performances sont les détection de dégradations de performance, fournir une explication aux programmeurs, et en suggérant des moyens de résoudre les *performance bugs*. Le défi se pose parce que la programmation par squelettes est une modèle programmation de haut niveau. Tous les détails complexes du parallélisme et de distribution sont cachés des programmeurs. Par conséquent, les programmeurs ne savent pas comment leurs muscles auront une incidence sur les performances des applications. Inversement, le bas niveau d'information du système n'a pas de sens pour les programmeurs pour perfectionner les muscles de leurs applications.

Une vue global de la méthodologie est illustrée dans la Figure 9.1). Après l'exécution d'une demande, les mesures de performance sont utilisés pour déterminer les causes de l'inefficacité de performance. Une fois les causes sont identifiées, un processus de blâmer a lieu en considerent le *workout* des muscles. Le résultat du processus de blâmer rendre le code muscle suspecté. Les programmeurs peuvent ensuite analyser les code suspecté, pour affiner leurs applications.

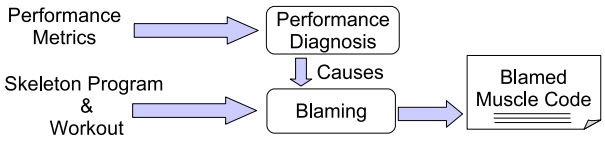


Figure 9.1: Performance Tuning

9.1.1 Les Paramètres de Performance

Plusieurs mesures peuvent être obtenus auprès de l'exécution du squelette. Par exemple, le temps passé par une tâche dans chaque état: $T_{processing}$, T_{ready}, $T_{waiting}$, et $T_{finished}$. Il a aussi $T_{wallclock}$, le temps *wallclock* de l'application; $T_{computing}$, le temps accumulés du CPU; et les sur coût de temps: $T_{overhead} = T_{processing} - T_{computing}$, ce qui représente essentiellement le coût des temps de communication entre le *task-pool* et les interprèteurs.

De nombreux autres mesures peuvent également être collectées, comme les granularité des tâches qui peuvent être définies en fonction de la taille, de portée et la profondeur de l'arbre de sous-tâche ($span^{depth} = size$), ou en utilisant le mesures de temps $granularity = \frac{T_{computing}}{T_{overhead}}$.

9.1.2 Diagnostic de Performance

Un arbre d'inférence est utilisé pour diagnostiquer les inefficacités de performance des squelettes algorithmique. Le diagnostic utilise les mesures de performance pour trouver les causes des *performance bugs*. Les causes peuvent correspondre à deux types. Les *causes externes*, telles que le sur coût du système de déploiement, de surcharge du système et les *causes réglables* telles que la souscharge des ressources, très grand tâches, très petite tâches, et les *bottlenecks*.

9.1.3 Workout des Muscles

Soit $m_0, ..., m_k$ être une représentation indexé de tous les muscles de code dans un programme avec des squelettes $\triangle$. On dira que le *workout* est une fonction qui, étant donné un programme des squelettes et un paramètre d'entrée, après l'application est exécutée, renvoie une liste de tous les muscles exécutés avec le temps de calcul pour chaque un:

$$workout(\triangle, p) = [(m_i, t_0), ..., (m_j, t_n)]$$

Le *workout* du squelette représente une trace de la façon dont le muscle ont été exécutés pour ce squelette. Le même muscle peut apparaître plusiers fois dans le *workout* avec des différents temps d'exécution.

9.1.4 Blâmer de Code

Comme les squelettes algorithmiques font des abstractions pour la programmation des couches plus bas de l'infrastructure, le causes des niveau plus bas n'ont pas de signification pour le programmeur. Par conséquent, nous devons lier les causes avec quelque chose que le programmeur peut se rapportent, ce qui correspond au muscles qui ont été mis en place par le programmeur. Le processus de lier de causes de bas niveau avec les code est ce que nous appelons blâmer.

Un mécanisme pour blâmer doit lier chaque cause avec les codes du muscle suspecté. Ainsi, le blâme doit prendre en considération: les causes de bas niveau (le résultat du diagnostic de performance), le programme des squelettes, et le *workout* des muscles.

9.2 System de Typage

Les muscles sont des boîtes noires pour le bibliothèque des squelettes, et même si chaque un est exempt d'erreurs, une combinaison inapproprié avec un squelette peut donner d'erreurs de typage a l'exécution. Cela se produit parce que, au cours de l'évaluation du programme de squelettes, le résultat d'un muscle est donné à *le prochain* muscle comme paramètre. Lorsque *le prochain* est déterminé au moment de l'exécution par le assemblage de squelettes. Un type incompatibles entre le résultat d'un muscle et le paramètre du prochaine donne des erreurs pendant l'exécution, qui sont difficiles à détecter et maîtriser, spécialement sur les environnements distribués.

Des squelettes non types amènent le programmeur à utiliser des *type casts* dans la programmation des muscles. En effet, si le langue de plus haut niveau des (squelettes) n'est pas capable de transmettre les types entre les muscles, alors les plus pauvres hypothèse seront prises pour le typage des muscles (par exemple, dans un langage des squelette non types en Java, les fonctions musculaires accepteron `Object` comme le type des paramètre et résultats). Dans ce cas, chaque objet reçu par un muscle doit être *type casts* dans le bon type, qui est très senible aux erreurs.

D'autre part, les squelettes types soulage le programmeur de gérer le trasntypage de tous les argument des muscles: le système typage vérifié que le type d'entrée d'un muscle est égal à celui qui a été déclaré comme le type de sortie du muscle précédent. Les transtypages restent nécessaires lorsque celle-ci serait exigé par le langage sous'jacent. *Pour résumer, le sécurité du type assurée par la langage sous-jacente est transmis par les squelettes: le sécurité de type est élevé au niveau du squelette.*

Notre contribution consiste à détecter les erreurs de typage des programmes avec des squelettes, ce qui améliore le *safeness* de la programmation avec des squelettes algorithmiques. Despuis l'aspect théorique, nous contribuons en fournissant des règles de typage sémantique pour chacun des squelettes dans la langage. Ensuite, nous prouvons que ces règles de typage en effet satisfont la sureté du typage. Sur le plan pratique, nous montrons comment le système de typage proposé peut être implemente dans un bibliothèque des squelettes Java [Micc]

en prenant l'avantage de Java Generics [BOSW98]. À ce titre, nous montrons qu'il n'y a plus de transtypages qui sont imposées par le langage des squelettes dans le muscles et, plus important encore, que toutes les validations de typage des composition des squelette sont réalisées lors compilation.

Par souci de clarté, nous ne montrerons que le squelette *pipe*, se référer à [CHL08] pour les autres squelettes. Pour généralité, nous supposons que les squelettes sont réduits aux *big step* sémantiques telles que:

$$\text{R-PIPE}\quad \frac{\triangle_1(p) \Downarrow s \qquad \triangle_1(s) \Downarrow r}{pipe(\triangle_1, \triangle_2)(p) \Downarrow r}$$

9.2.1 Règle de Typage

Pour chaque squelette de la bibliothèque, nous définissons une règle de typage, comme ce qui suit pour le squelette *pipe*:

$$\text{T-PIPE}\quad \frac{\triangle_1 : P \to X \qquad \triangle_2 : X \to R}{pipe(\triangle_1, \triangle_2) : P \to R}$$

La règle stipule que si la première étape ($\triangle_1$) peut transformer un paramètre de type P en un résultat de type X, et la deuxième étape ($\triangle_2$) peut prendre un paramètre de type X et rendre un résultat de type R, alors le *pipe* est typé pour recevoir un paramètre de type P et rendre un résultat de type R.

9.2.2 Réduction de Sujet

Néanmoins, la règle de typage n'est pas suffisante pour assurer la sécurité du type. La propriété de réduction de sujet exprime la préservation de type par la réduction sémantique, ce qui signifie que un type déduire pour une expression ne changera pas (ou deviendra plus précise) au cours de l'exécution: le type d'une expression est compatible avec le type de cette expression avant l'évaluation. Sans cette propriété, aucune des propriétés assurées par le système de typage est utile. Pour les squelettes, la propriété de réduction de sujet peut prendre la forme suivante.

$$\text{SR-}\triangle\quad \frac{\triangle(p) : R \qquad \triangle(p) \Downarrow r}{r : R}$$

Cette propriété doit être prouvée pour toutes les règles du système de typage. Nous ilustrate ici l'exemple de le squelette *pipe*.

La propriété de réduction de sujet pourle squelette *pipe* est:

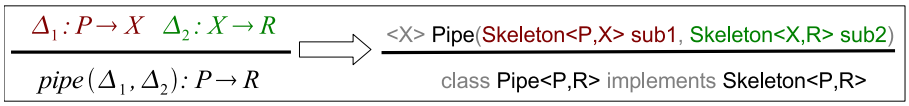


Figure 9.2: From theory to practice: *T-PIPE* rule.

$$\text{SR-PIPE}\ \frac{pipe(\triangle_1, \triangle_2)(p) : R \qquad pipe(\triangle_1, \triangle_2)(p) \Downarrow r}{r : R}$$

Proof. Décomposons maintenant ces conclusions en affirmant que $pipe(\triangle_1, \triangle_2)(p) : R$ et $pipe(\triangle_1, \triangle_2)(p) \Downarrow r$, nécessairement:

$$\frac{p : P \qquad \dfrac{\triangle_1 : P \to X \qquad \triangle_2 : X \to R}{pipe(\triangle_1, \triangle_2) : P \to R}\ \text{T-PIPE}}{pipe(\triangle_1, \triangle_2)(p) : R}\ \text{APP-}\triangle \qquad \frac{\triangle_1(p) \Downarrow x \qquad \triangle_2(x) \Downarrow r}{pipe(\triangle_1, \triangle_2)(p) \Downarrow r}\ \text{R-PIPE}$$

Enfin, nous prouvons que r a le type R comme suit:

$$\dfrac{\dfrac{\dfrac{p : P \quad \triangle_1 : P \to X}{\triangle_1(p) : X} \qquad \triangle_1(p) \Downarrow x}{x : X} \qquad \triangle_2 : X \to R}{\triangle_2(x) : R} \qquad \triangle_2(s) \Downarrow r}{r : R}$$

□

9.2.3 Squelettes avec Type Sûr en Java

Nous allons maintenant illustrer le système de typage dans le cadre d'un bibliothèque des squelettes en Java. Plus précisément, nous utilisons Java Generics pour préciser notre API des squelettes: les contraintes sur la compatibilité des types exprimés comme des règles de typage sont traduits par le fait que, dans l'API des squelettes, plusieurs paramètres ont les mêmes types (générique).

L'idée derrière la sémantique de typage est que, s'il est possible de garantir la compatibilité des types des paramètres, alors le squelette sera correctement typé. Par conséquent, puisque le programme des squelettes est défini lors de la construction du objets des squelettes, la validation des types eu effectué au niveau des constructéurs des classes de squelette. La Figure 9.2 montre l'analogie entre la sémantique de type et l'implémentation avec Java Generics pour la règle *T-PIPE*.

Les premises de les règles du typage sont appliquées à la signature du méthode constructeur du squelette, et la conclusion de les règles de typage se reflète sur la signature du classes de squelette.

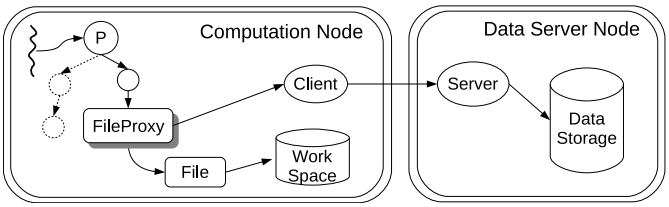


Figure 9.3: Proxy Pattern for Files

```
class Pipe<P,R> implements Skeleton<P,R> {
 public <X> Pipe(Skeleton<P,X> a, Skeleton<X,R> b);
}
```

Les avantages d'utiliser un système de typage pour skeletons avec Java Generics sont clairs: il n'est pas nécessaire d'implémenter un mécanisme de validation du typage supplémentaire; les squelettes n'imposent aucun transtypage dans le code des muscles et, plus important encore, la validation du typage est assurée lors de la composition des squelettes.

9.3 Transfert de Fichiers

Une grande partie des applications scientifiques distribuées génèrent et manipulent de grandes quantités d'information. On trouve de telles applications dans, par exemple, les domaines de la bio-informatique, de la physique des hautes énergies ou encore l'astronomie.

La programmation à squelettes rend nécessaire un mécanisme permettant aux programmeurs d'utiliser leurs méthodes habituelles (non parallèles) de lecture-écriture de fichiers dans les muscles. Ce mécanisme ne doit pas forcer l'écriture de code spécifique pour le transfert de fichiers. Il doit fournir un support efficace et transparent pour le transfert de fichiers entre l'exécution de muscles.

Malgré l'importance de ce mécanisme, il y a étonnament peu de mécanismes pour la gestion des fichiers dans les modèles de programmation à squelettes. La plupart d'entre eux pourrait être amélioré en traitant les aspects de distribution des fichiers à l'intérieur des muscles, comme par exemple dans ASSIST [ACD+06]. Malheureusement, cette stratégie conduit à un mélange de code non fonctionnel (distribution de données) avec le code fonctionnel.

Par conséquent, nous nous sommes concentré sur l'intégration des abstractions des données avec un modèles de programmation à squelettes. Nous abordons ce problème en considérant la facilité d'utilisation et la performance du modèle de programmation. Nous pensons que l'intégration des fichiers de données avec les squelettes doit se faire de manière transparente et non intrusive, pour préserver la séparation entre le code fonctionnel et le code non-fonctionnel. Les programmeurs ne devraient pas avoir à se préoccuper de l'emplacement, de la circulation ou du stockage des données. En outre, ils ne devraient pas avoir à modifier leur façon de travailler avec les données. Cela signifie que le mécanisme ne doit pas imposer de language ou de bibliothèque particulière.

9.3.1 Transparence avec FileProxy

The Proxy Pattern [GHJV95] is used to achieve transparent access to files. Files are rendered accessible using the `FileProxy` object as shown in Figure 9.3. By intercepting calls at the proxy level, the library is able to detect when a muscle is accessing a file. In a way, the `FileProxy` illuminates a specific aspect inside black box muscles.

9.3.2 Stage-in & Stage-out

Lorsque un paramètre P est soumis à l'environnement du squelette, un Phase-Soumission (*stage-in*) des fichiers est effectuée. Tout d'abord, toutes les références de type File dans P sont remplacées par des références vers des FileProxy. Ensuite, les données de ces fichiers sont stockées dans le serveur de données. Si une collision de nom se produit ou qu'une erreur de transmission de donnée a lieu, une exception est immédiatement levée, avant que le paramètre soumis ne soit effectivement soumis à l'environnement du squelette.

Lorsque le résultat final R a été calculé, une phase-résultat (*stage-out*) précède le retour de ce résultat à l'utilisateur. Toutes les références vers des `FileProxy` présents dans R sont ainsi remplacées par un objet de type `File` standard pointant vers un fichier local qui contient les données distantes.

De même, au cours de l'exécution, et avant qu'un interpréteur n'invoque un muscle, une phase-soumission est effectuée sur les nœuds de l'interpréteur. S'il n'est pas déjà présent, un espace de travail (*workspace*) unique et indépendant est créé. Tout, partie ou aucun des objets `FileProxy` présents dans le graphe d'objets de P y seront alors téléchargés, et les possibles références vers ces `FileProxy` seront mises à jour avec le nouvel emplacement du fichier.

9.3.3 L'abstraction Workspace

L'abstraction de *workspace* permet aux muscles de disposer sur le nœud de calcul d'un espace disque local. Dans le cas ou plusieurs muscles sont exécutés simultanément sur un même nœud, chacun des muscles aura la garantie de disposer de son propre *workspace* indépendant.

Le *workspace* fournit les méthodes suivantes :

```
interface WSpace{
  public File newFile(String name);
  public void exec(File bin, String args);
}
```

La fabrique `WSpace.newFile()` pourra être utilisée pour créer une référence de type `File` dans le workspace. La méthode `WSpace.exec(...)` pourra être utilisée pour exécuter une commande native avec des environnements d'exécution correctement configurés (par exemple le répertoire de travail courant).

9.3.4 Stockage de Données & Compteur de Références

Nous supposons l'existence d'un serveur de stockage de données, capable de stocker des données, la récupération des données, et de garder une trace sur la référence de chaque donné. Le serveur de stockage fournit les opérations suivantes:

- $store(F_x, k) \rightarrow i$. Stocke les données représentées dans le fichier F_x, avec une compte initial des références $k > 0$. La fonction retourne un identifiant unique pour ces données sur le serveur de stockage.
- $retrieve(i) \rightarrow F_x$. Récupère les données situé sur le serveur et identifié par i.
- $count(i, \delta) \rightarrow boolean$. Mises à jour le compter de références par δ. Retourne $true$ si les références est inférieur ou égal à zéro, et $false$ autrement.

Une fois que le compteur de références à atteint zéro pour un fichier, aucune opération ne sera plus effectuée sur ses données, et la suppression de celles-ci sera alors laissée à la discrétion du serveur de données.

Au cours de l'exécution d'un programme avec des squelette, les données peuvent être créés, modifiés et supprimés. Aussi, le références de type `File` pointant vers des données peut être créé, supprimé, et copiés. Par conséquent, il c'est à la bibliothèque squelette de fournir un appui pour ces comportements par: le stockage de fichiers nouvelles et modifiées et de garder une compte sur le nombre de reference vers un fichier pour supprimer les données lorsqu'elle n'est plus accessible.

Au cours de l'exécution d'un programmes avec des squelettes, des données peuvent être créées, modifiées et supprimées. De même, des références vers des objets de type `File` pointant vers ces données peuvent être créées, copiées et supprimées. Par conséquent, c'est à l'environnement des squelettes de fournir un support pour ces comportements par: le stockage de données nouvelles ou modifiées, et la conservation du nombre de références vers un fichier pour permettre la suppression de données lorsqu'elles ne sont plus accessibles.

Chapter 10

Perspectives et Conclusions

10.1 Perspectives

Nous pensons que la recherche autour des squelettes algorithmiques est une région fertile avec des nombreuses directions possibles. Néanmoins, nous limitons cette section sur de recherche qui, selon nous, est alignés avec l'objectif sous-jacent de cette thèse: l'adoption du squelette bibliothèques comme un modèle de programmation dominant.

10.1.1 Squelettes sans état

L'une des hypothèse de cette thèse correspond à avoir des squelettes algorithmiques sans état, tel que défini au Chapitre 3, ce qui implique que les muscles n'ont pas d'état également. Si nous nous rappelons, les squelettes sont issus du domaine de la programmation fonctionnelle où ce type de raisonnement semble naturel et acceptable pour obtenir un plus grand degré de parallélisation.

En effet, alors que l'utilisation de muscles sans état est une hypothèse raisonnable dans la programmation fonctionnelle, elle n'est pas naturelle quand on raisone sur un model impératif, et plus particulierment, dans une programmation orientée objet. Pour les programmeurs non-parallèle, il est intuitif de raisonner sur des algorithmes où les objets sont partagés (par référence) par les différentes parties de l'application. Ces programmeurs sont donc étonnés par le fait que le partage d'objets entre les différentes parties du squelette sont interdit. Quelque chose simple comme compter le nombre de fois qu'un muscle f a été invoquée n'est pas possible avec les squelettes sans état. Le moyen de résoudre ce problème dans le monde séquentiel est très simple: avec un compteur (i) et ayant f incrementant la valeur de i chaque fois qu'il est invoqué ($i \leftarrow i + 1$).

La façon la plus simple de gérér des muscles avec état est de séquencer tout accès à celui-ci. Le danger avec cette approche est que la serrure est trop gros. Tous les accès simultané à une fonction musculaire sont séquencés, et donc le parallélisme acquis dans le squelette peut être perdu. Considérez par exemple la composition d'un muscle f avec état a l'intérieur d'un squelette $farm$: $farm(f)$. Un muscle séquentiel f correctement écrit est maintenant incorrect parce que plusiuers évaluations concurrentes de f peuvent coexister, toutes pouvant etre

en train de modifier la même variable i. La pire solution à ce problème est d'acquérir un verrou avant l'évaluation de f et le relâcher une fois f terminé, parce que cette solution séquence toutes les invocations de f et annule le parallélisme mis en place par le squelette $farm$.

Une meilleure approche est de réduire la taille du verrou pour permettant d'acceder á i. Cela nécessite d'écrire f différemment, car maintenant le programmeur doit être conscient que les évaluations de f se peuvent produire simultanément. En outre, le programmeur doit avoir accès à un mécanisme de verrouillage, qui, jusqu'à présent, était inutile. Malheureusement, cette approche met en evidence a complexité bien connue de la programmation parallèle, ce qui va á l;encontre du but de l'introduction de squelettes: fournir des abstractions plus élevé pour la programmation parallèle.

10.1.2 AOP pour les squelettes

Les lecteurs familiers avec Aspect-Oriented Programming (AOP) [KLM+97] aurez remarqué qu'un bon nombre des techniques utilisées tout au long de cette thèse, en particulier dans le Chapitre 6, ressemblent à celles de l'AOP.

En effet, l'idée de ne pas melanger les aspects fonctionnels en utilisant l'héritage [CEF02], de la même façon que l'abstraction `FileProxy` a été utilisé pour intercepter des appels des objets de type `File` n'est pas nouveau. Le dilemme de l'instanciation des objets augmentée par des aspects a été resolu en utilisant des methods factory [CG07] de la même manière que la method factory a été utilisé pour l'abstraction le workspace présenté dans la section 9.3.3. Et les transformations de `File` → `FileProxy` → `File`, peuvent être encadrées dans le domaine des aspects dynamiques [DDDCG01, DDDCG02] et de la reclassification d'objet.

De la perspective AOP, le Chapitre 6 a fourni une méthodologie spécifique pour le tissage des aspects transfert de fichiers et squelettes algorithmiques, et a montré méthodologie comme l'AOP peut être appliquée aux squelettes algorithmiques. Plus généralement, l'intégration de l'AOP avec la programmation distribuée a déjà été proposé pour d'autres bibliotheques paralleles tels que la JAC[PSD+04], J2EE [CG04], ReflexD [TT06], etc.

Par conséquent, comme [MAD07], nous pensons que l'intégration de l'AOP avec les squelettes algorithmiques est un mécanisme prometeur pour soutenir d'autres aspects non-fonctionnels dans la programmation par squelettes.

Comme travaux futurs, nous souhaitons généraliser les méthodes présentées dans cette thèse pour supporter d'autres aspects non-fonctionnels dans les squelettes algorithmiques. En effet, notre objectif est de fournir un modèle AOP pour les squelettes algorithmiques, ce qui permettra une intégration sur mesure des autres aspects non-fonctionnels dans les squelettes, tels que les état des muscles, la collecte de statistiques, l'envoi d'événement, etc.

10.2 Conclusions

La complexité et les difficultés de la programmation parallèle ont conduit au développement de nombreux modèles de programmation parallèles et distribués, chacun ayant ses points forts. Seuls quelques-uns de ces modèles sont devenus dominant, alors que la plupart restent confinés dans des niches.

Dans cette thèse, nous avons mis l'accent sur l'un de ces modèles de programmation de niche: les squelettes algorithmiques, qui exploitent la concurrence par des schémas de parallélisme récurrent. L'idée sous-jacente des squelettes est de factoriser les parties récurrentes des applications parallèles et de fournir un moyen de personnaliser les schemas: un squelette. Le squelette définit implicitement les aspects de parallélisation et de distribution, alors que les blocs séquentiels écrit par les programmeurs fournissent les aspects fonctionnels de l'application.

Nous avons proposé un modèle pour les squelettes algorithmiques et nous avons fait un implémentation en Java sous fomre d'une bibliothèque: Calcium. Le Tableau 10.1 montre un profil des caractéristiques de Calcium comme décrit pour d'autres bibliothèques de squelettes dans la Section 2.3.

	Calcium
Activity Years	2006-2008
Programming Language	Java
Execution Language	Java
Distribution Library	Threads & ProActive Active Objects & ProActive Scheduler
Type Safe	yes
Skeleton Nesting	yes
File Access	transparent
Skeleton Set	farm, pipe, seq, if, for, while, d&c, map, fork

Table 10.1: Calcium Summary

Le tableau confirme que la combinaison de caractéristiques dans Calcium est unique parmi les autres bibliothèques de squelettes connus.

Au cours de cette thèse, nous avons fait un effort pour aborder des questions tant du côté théorique que du côté pratique. Par conséquent, la plupart de nos résultats sont indépendants de notre implémentation , cependent Calcium est disponible pour des expériences pratiques.

Pour résumer, les contributions présentées dans cette thèse sont les suivantes:

- Une enquête sur l'état de l'art de la programmation avec des squelettes algorithmiques.
- Calcium, une bibliothèque de programmation avec des squelettes algorithmiques en Java avec:
  - Une bibliothèque pour la programmation squelette, proposant
  - Des squelettes de type tâche et de données parallèle composables, et
  - Multiples environnements parallèles et distribués pour l'exécution.
- Un modèle de réglage des performances pour les squelettes algorithmiques et son implémentation dans Calcium.
- Un système de typage pour les squelettes algorithmiques composables et son implémentation en profitant du Java Generics dans Calcium.
- Support pour l'accès et transfert des fichiers:
  - Un modèle de transfert de fichier basé sur les objets actifs et sa mise en oeuvre dans ProActive.
  - Un modèle transparent d'accès aux fichiers pour les squelettes algorithmiques et son implémentation en Calcium.

Bien que nous reconnaissions la nécessité de poursuivre les recherches et travaux dans ce domaine, il est de notre espoir que les questions que nous avons abordées au cours de cette thèse contribuerons à aider les programmeurs à adopter les squelettes algorithmiques comme l'un des principaux modèles de programmation.

Part III

Appendix

Appendix A: Subject Reduction Proofs

This appendix details the subject reduction proofs that were not given in Section 5.2.4.

Seq Preservation

We must prove:

$$\text{SR-SEQ}\ \frac{seq(f_e)(p) : R \qquad seq(f_e)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $seq(f_e)(p) : R$ and $seq(f_e)(p) \Downarrow r$ we obtain:

$$\text{SEQ-T}\ \frac{f_e : P \to R}{\text{APP-}\triangle\ \dfrac{seq(f_e) : P \to R \qquad p : P}{seq(f_e)(p) : R}} \qquad\qquad \frac{f_e(p) \Downarrow r}{seq(f_e)(p) \Downarrow r}\ \text{R-SEQ}$$

Applying APP-F and SR-F we obtain the following inference:

$$\text{APP-F}\ \frac{p : P \qquad f_e : P \to R}{\text{SR-F}\ \dfrac{f_e(p) : R \qquad\qquad f_e(p) \Downarrow r}{r : R}}$$

□

Farm Preservation

We must prove:

$$\text{SR-FARM}\ \frac{farm(\triangle)(p) : R \qquad farm(\triangle)(p) \Downarrow r}{r : R}$$

which is done similarly to the case *Seq* above, except that it uses APP-$\triangle$ and SR-$\triangle$ instead of APP-F and SR-F.

If Preservation

We must prove:

$$\text{SR-IF}\ \frac{if(f_b, \triangle_{true}, \triangle_{false})(p) : R \quad if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}{r : R}$$

Proof. By decomposing $if(f_b, \triangle_{true}, \triangle_{false})(p) : R$ and $if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r$ we obtain:

$$\frac{p : P \qquad \dfrac{f_b : P \to boolean \quad \triangle_{true} : P \to R \quad \triangle_{false} : P \to R}{if(f_b, \triangle_{true}, \triangle_{false}) : P \to R}\ \text{T-IF}}{if(f_b, \triangle_{true}, \triangle_{false})(p) : R}\ \text{APP-IF}$$

$$\frac{f_b(p) \Downarrow \{true|false\} \quad \triangle_{\{true|false\}}(p) \Downarrow r}{if(f_b, \triangle_{true}, \triangle_{false})(p) \Downarrow r}\ \text{R-IF}$$

Applying app-$\triangle$ and SR-$\triangle$ on the right skeleton (Δ_{true} or Δ_{false}), we obtain the following inference:

$$\frac{\dfrac{p : P \quad \triangle\{true|false\} : P \to R}{\triangle_{\{true|false\}}(p) : R}\ \text{APP-}\triangle \qquad \triangle_{\{true|false\}}(p) \Downarrow r}{r : R}\ \text{SR-}\triangle$$

□

While Preservation

We must prove:

$$\text{SR-WHILE}\ \frac{while(f_b, \triangle)(p) : P \qquad while(f_b, \triangle)(p) \Downarrow r}{r : P}$$

Proof. By decomposing $while(f_b, \triangle)(p) : P$, and $while(f_b, \triangle)(p) \Downarrow r$ we obtain:

$$\frac{p : P \qquad \dfrac{f_b : P \to boolean \qquad \triangle : P \to P}{while(f_b, \triangle) : P \to P}\ \text{T-WHILE}}{while(f_b, \triangle)(p) : P}\ \text{APP-}\triangle \qquad \frac{f_b(p) \Downarrow false}{while(f_b, \triangle)(p) \Downarrow p}\ \text{R-WHILE-FALSE}$$

$$\frac{f_b(p) \Downarrow true \quad \triangle(p) \Downarrow s \quad while(f_b, \triangle)(s) \Downarrow r}{while(f_b, \triangle)(p) \Downarrow r}\ \text{R-WHILE-TRUE}$$

This is done again by a sub-recurrence on the number of times the skeleton Δ is executed. The case where $f_b(p) \Downarrow false$ is trivial because $p : P$. For the case

where $f_b(p) \Downarrow true$, we must first determine that $s : P$. Using the recurrence hypothesis (SR-$\triangle$) and app-$\triangle$:

$$\text{APP-}\triangle\ \frac{p : P \qquad \triangle : P \to P}{\triangle(p) : P} \qquad \triangle(p) \Downarrow s \ \text{SR-}\triangle \over s : P$$

Now, by applying the sub-recurrence hypothesis on s we have:

$$\begin{array}{r} \text{APP-WHILE}\ \dfrac{s : P \qquad while(f_b, \triangle) : P \to P}{while(f_b, \triangle)(s) : P} \qquad while(f_b, \triangle)(s) \Downarrow r \\ \text{SR-WHILE}\ \dfrac{}{r : P} \end{array}$$

□

Map Preservation

We must prove:

$$\frac{\text{SR-MAP}\quad map(f_d, \triangle, f_c)(p) : R \qquad map(f_d, \triangle, f_c)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $map(f_d, \triangle, f_c)(p) : R$ and $map(f_d, \triangle, f_c)(p) \Downarrow r$ we obtain:

$$\frac{p : P \qquad \dfrac{f_d : P \to \{Q\} \qquad \triangle : Q \to S \qquad f_c : \{S\} \to R}{map(f_d, \triangle, f_c) : P \to R}\ \text{T-MAP}}{map(f_d, \triangle, f_c)(p) : R}\ \text{APP-}\triangle$$

$$\frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{map(f_d, \triangle, f_c)(p) \Downarrow r}\ \text{R-MAP}$$

Applying APP-F and SR-F:

$$\begin{array}{r} \text{APP-F}\ \dfrac{p : P \qquad f_d : P \to \{Q\}}{f_d(p) : \{Q\}} \qquad f_d(p) \Downarrow \{p_i\} \\ \text{SR-F}\ \dfrac{}{\{p_i\} : \{Q\}} \end{array}$$

Therefore $p_i : Q$, and applying APP-$\triangle$ SR-$\triangle$:

$$\begin{array}{r} \text{APP-}\triangle\ \dfrac{p_i : Q \qquad \triangle : Q \to S}{\triangle(p_i) : S} \qquad \triangle(p_i) \Downarrow r_i \\ \text{SR-}\triangle\ \dfrac{}{r_i : S} \end{array}$$

Thus, $\{r_i\} : \{S\}$, and by APP-F and SR-F:

$$\begin{array}{r} \text{APP-F}\ \dfrac{\{r_i\} : \{S\} \qquad f_c : \{S\} \to R}{f_c(\{r_i\}) : R} \qquad f_c(\{r_i\}) \Downarrow r \\ \text{SR-F}\ \dfrac{}{r : R} \end{array}$$

□

Fork Preservation

We must prove:

$$\text{SR-FORK}\ \frac{fork(f_d, \{\triangle_i\}, f_c)(p) : R \qquad fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r}{r : R}$$

Proof. By decomposing $fork(f_d, \{\triangle_i\}, f_c) : R$ and $fork(f_d, \{\triangle_i\}, f_c) \Downarrow r$ we obtain:

$$\frac{p : P \qquad \dfrac{f_d : P \to \{Q\} \quad \triangle_i : Q \to S \quad f_c : \{S\} \to R}{fork(f_d, \{\triangle_i\}, f_c) : P \to R}\ \text{T-FORK}}{fork(f_d, \{\triangle_i\}, f_c)(p) : R}\ \text{APP-}\triangle$$

$$\frac{f_d(p) \Downarrow \{p_i\} \qquad \forall i \quad \triangle_i(p_i) \Downarrow r_i \qquad f_c(\{r_i\}) \Downarrow r}{fork(f_d, \{\triangle_i\}, f_c)(p) \Downarrow r}\ \text{R-FORK}$$

Applying APP-F and SR-F:

$$\text{SR-F}\ \frac{\text{APP-F}\ \dfrac{p : P \qquad f_d : P \to \{Q\}}{f_d(p) : \{Q\}} \qquad f_d(p) \Downarrow \{p_i\}}{\{p_i\} : \{Q\}}$$

Then $p_i : Q$, and applying APP-$\triangle$ SR-$\triangle$, we have for all i:

$$\text{SR-}\triangle\ \frac{\text{APP-}\triangle\ \dfrac{p_i : Q \qquad \triangle_i : Q \to S}{\triangle_i(p_i) : S} \qquad \triangle_i(p_i) \Downarrow r_i}{r_i : S}$$

Therefore, $\{r_i\} : \{S\}$, and from APP-F and SR-F:

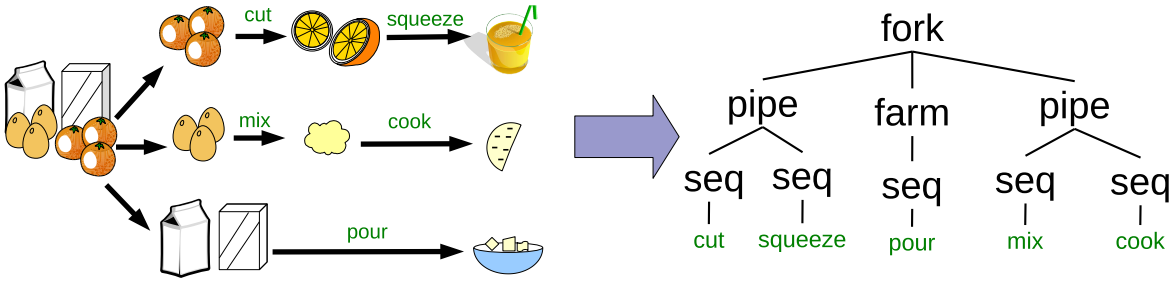
$$\text{SR-F}\ \frac{\text{APP-F}\ \dfrac{\{r_i\} : \{S\} \qquad f_c : \{S\} \to R}{f_c(\{r_i\}) : R} \qquad f_c(\{r_i\}) \Downarrow r}{r : R}$$

□

Appendix B: The Skeleton Kitchen

This appendix provides a simplified and reader-friendly overview of the concepts in Chapter 5.

The Skeleton Kitchen is the only kitchen in the world where the preparation of dishes is organized with skeletons. For example, a breakfast could be prepared with skeleton in the following way:



The Chef and his apprentices

The Chef is the boss of the kitchen. He decides which skeletons are used to prepare a dish. Formerly a computer science professor, the Chef decided many years ago to retire and pursue tastier interests in the domain of gastronomy. Since then, he has become knowledgeable and skilled in many cooking recipes, to which he refers as algorithms.

To help the Chef, the kitchen has an army of apprentices (students) who aspire to learn the ways of the kitchen. Their limited abilities are restricted mostly to menial tasks such as dish washing, but at least each apprentice is known to be capable of performing one basic cooking task.

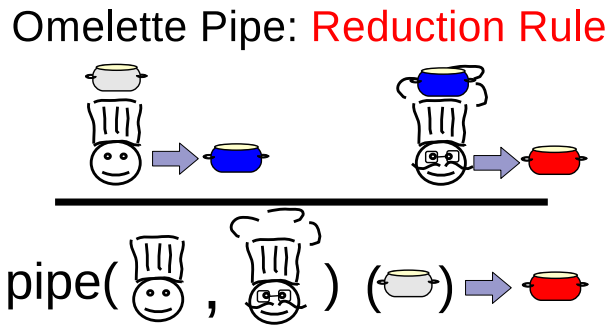
Since the apprentices are so numerous, the Chef actually has a hard time telling them appart. Most of the time this is not a problem, except when their basic skill is called upon to handle the most simple and tedious cooking tasks. On such days the absentminded Chef tends to assign the wrong apprentice to

the wrong job. Which, unfortunately for the apprentices, results on the Chef scolding them later on, and of course cleaning up the kitchen mess.

The Omelette Pipe

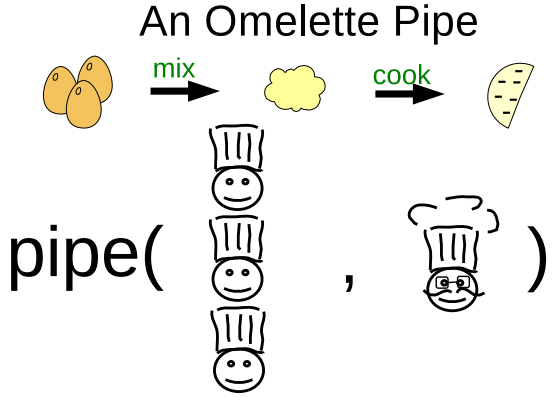
On the second Friday of October, the Chef likes to celebrates Egg Day by cooking a tasty omelette. The job is fairly simple and composed of two stages. First the eggs must be broken and mixed, and then the mix has to be cooked into a tasty omelette.

After careful thought, the Chef decides, using his vast experience, that *pipe* is the best skeleton for the job. He then lectures his students, like every other year, on how the pipe skeleton works.



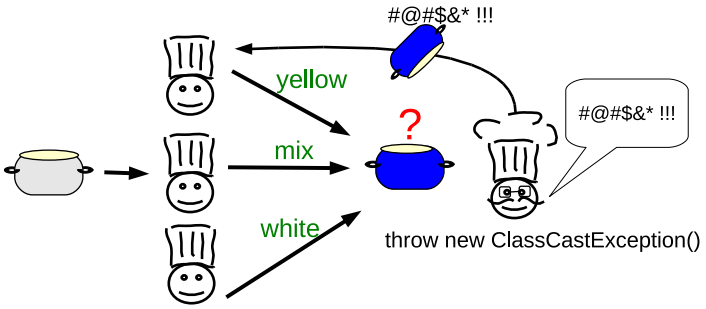
"The gray pot is passed to the pipe, taken by an apprentice in the first stage and cooked. The result contained in the blue pot is then passed to me (the Chef) waiting on the second stage. I then put my glasses on, check that the contents of the blue pot are what is expected, cook it, and provide the final result of the pipe in a red pot."

As in every year, the Chef decides to assign an apprentice for the first tedious task of preparing the mix, while he takes responsibility of cooking the mix into an omelette.



The Chef knows he has three egg skilled apprentices. The first apprentice is skilled in extracting the yellow from the eggs, the second the white, and the third

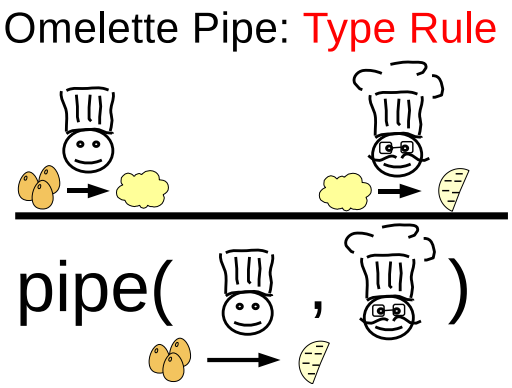
the mix. Since he is unable to distinguish them appart, the only alternative is to choose one of them at random and assign him to the task. Then, before the Chef begins the second stage of the pipe, he must put his glasses on and inspect the blue pot to see if its contents are indeed a mix. If not, the kitchen must stopped, cleaned up and restarted. This time with another apprentice on the first pipe stage.



Ofcourse this manual verification process is very tedious for the Chef, as he never remembers where his glasses are. For the students it is very annoying as well. First, because the Chef is very secretive and makes them work on a need to know basis. Thus they cannot tell the Chef when their skill is not suited for the dish. Second, because its up to them to clean the mess and restart the cooking from scratch.

Name Tags

The apprentices tired of the situation propose a typing system to the Chef. Their idea is to wear name tags with their cooking skills.

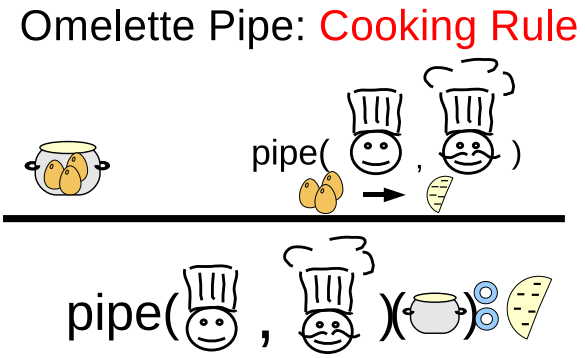


The students present their idea to the Chef and argue that as long as name tags are used extensively in the kitchen, there will never be cooking pots with wrong contents. Furthermore, as a consequence, the Chef will no longer need to put his glasses on to verify the contens of the pot before cooking.

After careful thought, the Chef tells the apprentices that he is not completely convinced by their proposal. So he tells them that before financing the manufacture of the name tags, they must prove to him that they will indeed work.

Taking The Glasses Off

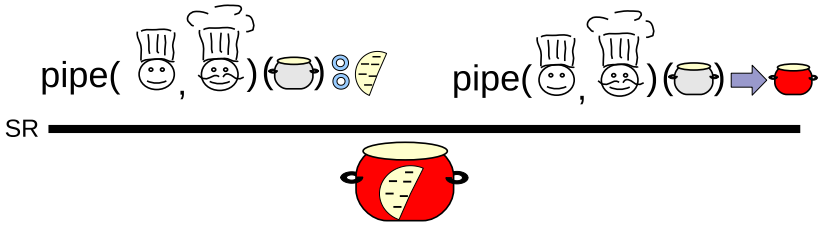
The students, encouraged by the challenge, decide to prove to the Chef that he no longer requires his glasses to cook. Nevertheless, before they begin they need to define what cooking is. They use the following rule:



Which states that, when a pipe is cooking, if the contents of the gray pot are eggs and the pipe can transform eggs into an omelette, then the pipe is cooking an omelette.

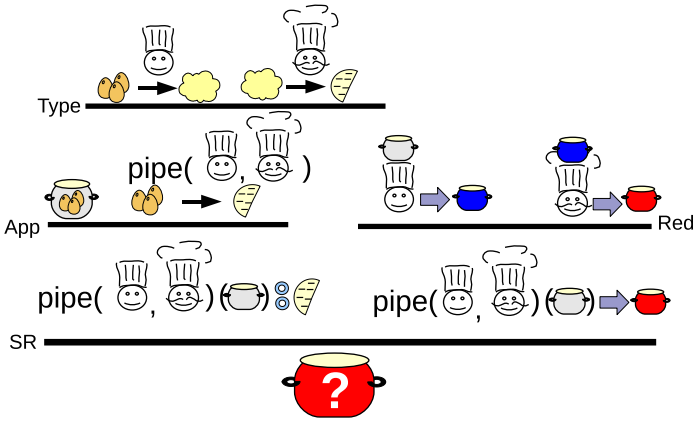
Subject Reduction

Now that everything is ready, the students identify the property which they want to prove. This property corresponds to subject reduction:

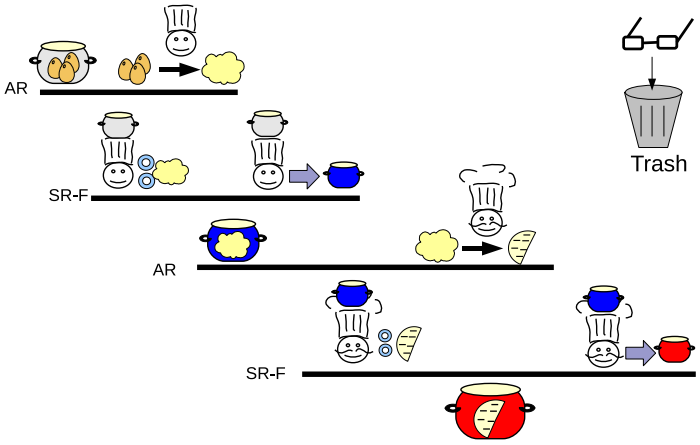


Which states that if the pipe is cooking an omelette, and the pipe does produce a result (cooking is successful), then this result is indeed and omelette. To prove this property the students must be able to derive the conclusion of the rule from the identified premises.

Therefore, the students expand the premises of the property using the reduction, type, and cooking rules:



Now they prove the property by restructuring the premises and arriving to the conclusion of the subject reduction property:



If a gray pot containing eggs is given to an apprentice capable of transforming eggs into mix, then when the student cooks the pot he is cooking a mix. If the student does produce a result in a blue pot, the contents of the blue pot will indeed be a mix. If the blue put is then given to the Chef who can transform a mix into an omelette, then when the chef cooks the blue pot he will indeed cook an omelette. Finally, when the Chef is finished cooking and produces a result in the red pot, we can then be sure that the contents of the red pot is an omelette, and thus the output of the pipe is indeed an omelette.

Which proves that the Chef no longer requires his glasses for cooking an omelette.

Bibliography

[AAB+02] Enrique Alba, Francisco Almeida, Maria J. Blesa, J. Cabeza, Carlos Cotta, M. Díaz, I. Dorta, Joachim Gabarró, C. León, J. Luna, Luz Marina Moreno, C. Pablos, Jordi Petit, A. Rojas, and Fatos Xhafa. Mallba: A library of skeletons for combinatorial optimisation (research note). In *Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing*, pages 927–932, London, UK, 2002. Springer-Verlag.

[AAB+06] E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Díaz, I. Dorta, J. Gabarró, C. León, G. Luque, J. Petit, C. Rodríguez, A. Rojas, and F. Xhafa. Efficient parallel lan/wan algorithms for optimization: the mallba project. *Parallel Computing*, 32(5):415–440, 2006.

[AADJ07] Marco Aldinucci, Gabriel Antoniu, Marco Danelutto, and Mathieu Jan. Fault-tolerant data sharing for high-level grid programming: A hierarchical storage architecture. In Marian Bubak, Sergei Gorlatch, and Thierry Priol, editors, *Achievements in European Research on Grid Systems*, CoreGRID Series, pages 67–82. Springer-Verlag, 2007.

[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[ABG02] Martin Alt, Holger Bischof, and Sergei Gorlatch. Algorithm design and performance prediction in a Java-based Grid system with skeletons. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par 2002*, volume 2400 of *Lecture Notes in Computer Science*, pages 899–906. Springer-Verlag, August 2002.

[ACC05] Isabelle Attali, Denis Caromel, and Arnaud Contes. Deployment-based security for grid applications. In *The International Conference on Computational Science (ICCS 2005), Atlanta, USA, May 22-25*, LNCS. Springer Verlag, 2005.

[ACD98] Marco Aldinucci, Massimo Coppola, and Marco Danelutto. Rewriting skeleton programs: How to evaluate the data-parallel stream-

parallel tradeoff. In Sergei Gorlatch, editor, *Proc of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 44–58. Fakultät für mathematik und informatik, Uni. Passau, Germany, May 1998.

[ACD+06] Marco Aldinucci, Massimo Coppola, Marco Danelutto, Nicola Tonellotto, Marco Vanneschi, and Corrado Zoccolo. High level grid programming with ASSIST. *Computational Methods in Science and Technology*, 12(1):21–32, 2006.

[ACD+08] Marco Aldinucci, Sonia Campa, Marco Danelutto, Marco Vanneschi, Peter Kilpatrick, Patrizio Dazzi, Domenico Laforenza, and Nicola Tonellotto. Behavioural skeletons in gcm: Autonomic management of grid components. In *PDP '08: Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, pages 54–63, Washington, DC, USA, 2008. IEEE Computer Society.

[AD99] Marco Aldinucci and Marco Danelutto. Stream parallel skeleton optimization. In *Proc. of PDCS: Intl. Conference on Parallel and Distributed Computing and Systems*, pages 955–962, Cambridge, Massachusetts, USA, November 1999. IASTED, ACTA press.

[AD04] Marco Aldinucci and Marco Danelutto. An operational semantics for skeletons. In G. R. Joubert, W. E. Nagel, F. J. Peters, and W. V. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, PARCO 2003*, volume 13 of *Advances in Parallel Computing*, pages 63–70, Dresden, Germany, 2004. Elsevier.

[AD07a] Marco Aldinucci and Marco Danelutto. The cost of security in skeletal systems. In Pasqua D'Ambra and Mario Rosario Guarracino, editors, *Proc. of Intl. Euromicro PDP 2007: Parallel Distributed and network-based Processing*, pages 213–220, Napoli, Italia, February 2007. IEEE.

[AD07b] Marco Aldinucci and Marco Danelutto. Skeleton based parallel programming: functional and parallel semantic in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, October 2007.

[AD08] Marco Aldinucci and Marco Danelutto. Securing skeletal systems with limited performance penalty: the muskel experience. *Journal of Systems Architecture*, 2008. To appear.

[ADD04] Marco Aldinucci, Marco Danelutto, and Jan Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *Proc. of CMPP: Intl. Workshop on Constructive Methods for Parallel Programming*, pages 35–47, Stirling, Scotland, UK, July 2004. Universität Münster, Germany.

[ADD07] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, December 2007.

[ADG+05] Marco Aldinucci, Marco Danelutto, Gianni Giaccherini, Massimo Torquati, and Marco Vanneschi. Towards a distributed scalable data service for the grid. In G. R. Joubert, W. E. Nagel, F. J. Peters, O. Plata, P. Tirado, and E. Zapata, editors, *Parallel Computing: Current & Future Issues of High-End Computing (Proc. of PARCO 2005, Malaga, Spain)*, volume 33 of *NIC*, pages 73–80, Germany, December 2005. John von Neumann Institute for Computing.

[ADT03] Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, July 2003.

[AG03a] Martin Alt and Sergei Gorlatch. Algorithmic skeletons for metacomputing. In *Fifth International Workshop on Advanced Parallel Processing Technology*, volume 2934 of *Lecture Notes in Computer Science*, pages 363–372. Springer-Verlag, 2003.

[AG03b] Martin Alt and Sergei Gorlatch. A prototype Grid system using Java and RMI. In Victor Malyshkin, editor, *Parallel Computing Technologies (PACT 2003), Nizhni Novgorod, Russia*, volume 2763 of *Lecture Notes in Computer Science*, pages 401–414. Springer-Verlag, 2003.

[AG03c] Martin Alt and Sergei Gorlatch. Using Skeletons in a Java-based Grid system. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par 2003*, volume 2790 of *Lecture Notes in Computer Science*, pages 682–693. Springer-Verlag, August 2003.

[AGL04] Mohammad Mursalin Akon, Dhrubajyoti Goswami, and Hon Fung Li. Superpas: A parallel architectural skeleton model supporting extensibility and skeleton composition. In *Parallel and Distributed Processing and Applications Second International Symposium, ISPA*, Lecture Notes in Computer Science, pages 985–996. Springer-Verlag, 2004.

[AHM+07] J. Alonso, V. Hernandez, G. Molto, A. Proença, and J. Sobral. Grid enabled jaskel skeletons with gmarte. In *1st Iberian Grid Infrastructure Conference*, 5 2007.

[ALG+07] Enrique Alba, Gabriel Luque, Jose Garcia-Nieto, Guillermo Ordonez, and Guillermo Leguizamon. Mallba: a software library to design efficient optimisation algorithms. *International Journal of Innovative Computing and Applications*, 1(1):74–85, 2007.

[Alt07] Martin Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, July 2007.

[APP+05] M. Aldinucci, A. Petrocelli, A. Pistoletti, M. Torquati, M. Vanneschi, L. Veraldi, and C. Zoccolo. Dynamic reconfiguration of Grid-aware applications in ASSIST. In *Proceedings of 11th Intl Euro-Par 2005: Parallel and Distributed Computing*, volume 3648 of *Lecture Notes in Computer Science*, August 2005.

[ASGL05] Mohammad Mursalin Akon, Ajit Singh, Dhrubajyoti Goswami, and Hon Fung Li. Extensible parallel architectural skeletons. In *High Performance Computing HiPC 2005, 12th International Conference*, volume 3769 of *Lecture Notes in Computer Science*, pages 290–301, Goa, India, December 2005. Springer-Verlag.

[AT04] Marco Aldinucci and Massimo Torquati. Accelerating apache farms through ad-hoc distributed scalable object repository. In M. Danelutto, M. Vanneschi, and Domenico Laforenza, editors, *Proc. of 10th Intl. Euro-Par 2004: Parallel and Distributed Computing*, volume 3149 of *LNCS*, pages 596–605. Springer Verlag, August 2004.

[BBC02] Laurent Baduel, Françoise Baude, and Denis Caromel. Efficient, Flexible, and Typed Group Communications in Java. In *Joint ACM Java Grande - ISCOPE 2002 Conference*, pages 28–36, Seattle, 2002. ACM Press. ISBN 1-58113-559-8.

[BBC05] Laurent Baduel, Françoise Baude, and Denis Caromel. Object-Oriented SPMD. In *Proceedings of Cluster Computing and Grid*, Cardiff, United Kingdom, May 2005.

[BC05] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In V. Sunderam, D. van Albada, P. Sloot, and J. Dongarra, editors, *The International Conference on Computational Science (ICCS 2005), Part II*, LNCS 3515, pages 764–771. Springer Verlag, 2005.

[BCD+97] B. Bacci, B. Cantalupo, M. Danelutto, S. Orlando, D. Pasetto andS. Pelagatti, and M. Vanneschi. An environment for structured parallel programming. In L. Grandinetti, M. Kowalick, and M. Vaitersic, editors, *Advances in High Performance Computing*, pages 219–234. Kluwier, Dordrecht, The Netherlands, 1997.

[BCDH05] F. Baude, D. Caromel, C. Delbé, and L. Henrio. A hybrid message logging-cic protocol for constrained checkpointability. In *Proceedings of EuroPar2005*, number 3648 in LNCS, pages 644–653, Lisbon, Portugal, August-September 2005. Springer.

[BCGH04] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Evaluating the performance of skeleton-based high level parallel programs. In M. Bubak, D. van Albada, P. Sloot, and J. Dongarra,

editors, *The International Conference on Computational Science (ICCS 2004), Part III*, LNCS, pages 299–306. Springer Verlag, 2004.

[BCGH05a] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Evaluating the performance of pipeline-structured parallel programs with skeletons and process algebra. *Scalable Computing: Practice and Experience*, 6(4):1–16, December 2005.

[BCGH05b] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eskel. In José C. Cunha and Pedro D. Medeiros, editors, *Euro-Par*, volume 3648 of *Lecture Notes in Computer Science*, pages 761–770. Springer, 2005.

[BCGH05c] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Scheduling skeleton-based grid applications using pepa and nws. *The Computer Journal, Special issue on Grid Performability Modelling and Measurement*, 48(3):369–378, 2005.

[BCGH05d] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Using eskel to implement the multiple baseline stereo application. [JNP+05], pages 673–680.

[BCHM07] Francoise Baude, Denis Caromel, Ludovic Henrio, and Matthieu Morel. Collective interfaces for distributed components. In *CCGRID '07: Proceedings of the Seventh IEEE International Symposium on Cluster Computing and the Grid*, pages 599–610, Washington, DC, USA, 2007. IEEE Computer Society.

[BCHV00] Françoise Baude, Denis Caromel, Fabrice Huet, and Julien Vayssière. Communicating mobile active objects in java. In Roy Williams Marian Bubak, Hamideh Afsarmanesh and Bob Hertzberger, editors, *Proceedings of the 8th International Conference - High Performance Computing Networking'2000*, volume 1823 of *Lecture Notes in Computer Science*, pages 633–643, Amsterdam, May 2000. Springer-Verlag.

[BCL+06] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software, Practice & Experience*, 36(11-12):1257–1284, 2006.

[BCL07] F. Baude, D. Caromel, and M. Leyton. File transfer in grid applications at deployment, execution and retrieval. *Multiagent and Grid Systems*, 3(4):381–391, 2007.

[BCLQ06] Françoise Baude, Denis Caromel, Mario Leyton, and Romain Quilici. Grid file transfer during deployment, execution, and retrieval. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA,*

and ODBASE, volume 4276 of *Lecture Notes in Computer Science*, pages 1191–1202. Springer-Verlag, 2006.

[BCM+02] F. Baude, D. Caromel, L. Mestre, F. Huet, and J. Vayssière. Interactive and descriptor-based deployment of object-oriented grid applications. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, pages 93–102, Edinburgh, Scotland, July 2002. IEEE Computer Society.

[BDLP08] Jost Berthold, Mischa Dieterle, Rita Loogen, and Steffen Priebe. Hierarchical master-worker skeletons. In Paul Hudak and David Scott Warren, editors, *PADL*, volume 4902 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2008.

[BDO+95] Bruno Bacci, Marco Danelutto, Salvatore Orlando, Sussana Pelagatti, and Marco Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[BDPV99] B. Bacci, M. Danelutto, S. Pelagatti, and M. Vanneschi. Skie: a heterogeneous environment for hpc applications. *Parallel Comput.*, 25(13-14):1827–1852, 1999.

[BK95] George Horatiu Botorog and Herbert Kuchen. Algorithmic skeletons for adaptive multigrid methods. pages 27–41, 1995.

[BK96a] G. H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In *Euro-Par '96: Proceedings of the Second International Euro-Par Conference on Parallel Processing*, pages 718–731, London, UK, 1996. Springer-Verlag.

[BK96b] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *HPDC '96: Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, page 243, Washington, DC, USA, 1996. IEEE Computer Society.

[BK96c] George Horatiu Botorog and Herbert Kuchen. Using algorithmic skeletons with dynamic data structures. In *IRREGULAR '96: Proceedings of the Third International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 263–276, London, UK, 1996. Springer-Verlag.

[BK98] G. H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theor. Comput. Sci.*, 196(1-2):71–107, 1998.

[BLA] BLAST. Basic local alignment search tool. http://www.ncbi.nlm.nih.gov/blast/.

[BLMP97] Silvia Breitinger, Rita Loogen, Yolanda Ortega Mallen, and Ricardo Pena. The eden coordination model for distributed memory systems. In *HIPS '97: Proceedings of the 1997 Workshop on High-Level Programming Models and Supportive Environments (HIPS*

'97), page 120, Washington, DC, USA, 1997. IEEE Computer Society.

[BMA+02] S. Bromling, S. MacDonald, J. Anvik, J. Schaeffer, D. Szafron, and K. Tan. Pattern-based parallel programming. In *ICPP '02: Proceedings of the 2002 International Conference on Parallel Processing (ICPP'02)*, page 257, Washington, DC, USA, 2002. IEEE Computer Society.

[Bor07] Shekhar Borkar. Thousand core chips: a technology perspective. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 746–749, New York, NY, USA, 2007. ACM.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: adding genericity to the java programming language. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 183–200, New York, NY, USA, 1998. ACM Press.

[BPP06] Hinde Bouziane, Christian Pérez, and Thierry Priol. Modeling and executing master-worker applications in component models. In *11th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS)*, Rhodes Island, Greece, 25 April 2006.

[BR07] Anne Benoit and Yves Robert. Mapping pipeline skeletons onto heterogeneous platforms. In *ICCS'2007, the 7th International Conference on Computational Science*, LNCS 4487, pages 591–598. Springer Verlag, 2007.

[Car93] Denis Caromel. Toward a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[CC05] D. Caromel and G. Chazarain. Robust exception handling in an asynchronous environment. In A. Romanovsky, C. Dony, JL. Knudsen, and A. Tripathi, editors, *Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems*. Tech. Report No 05-050, Dept. of Computer Science, LIRMM, Montpellier-II Univ. July. France, 2005.

[CdCBM07] Denis Caromel, Alexandre di Costanzo, Laurent Baduel, and Satoshi Matsuoka. Grid'BnB: A Parallel Branch & Bound Framework for Grids. In *Proceedings of the HiPC'07 - International Conference on High Performance Computing*, 2007.

[CdCM07] D. Caromel, A. di Costanzo, and C. Mathieu. Peer-to-Peer for Computational Grids: Mixing Clusters and Desktop Machines. *Parallel Computing*, 33(4-5):275–288, 2007.

[CDdCL06] Denis Caromel, Christian Delbé, Alexandre di Costanzo, and Mario Leyton. Proactive: an integrated platform for programming

and running applications on grids and p2p systems. *Computational Methods in Science and Technology*, 12, 2006.

[CDE+05] Manuel Clavel, Francisco Duran, Steven Eker, Patrick Lincoln, . Narciso Marti-Oliet, Jose Meseguer, and Carolyn L. Talcott. Maude manual. `http://maude.cs.uiuc.edu/maude2-manual/`, December 2005.

[CDF+97] S. Ciarpaglini, M. Danelutto, L. Folchi, C. Manconi, and S. Pelagatti. ANACLETO: a template-based p3l compiler. In *Proceedings of the Seventh Parallel Computing Workshop (PCW '97)*, Australian National University, Canberra, August 1997.

[CDHQ] Denis Caromel, Christian Delbé, Ludovic Henrio, and Romain Quilici. Dispositifs et procé dé s asynchrones et automatiques de transmission de ré sultats entre objets communicants (asynchronous and automatic continuations of results between communicating objects). French patent FR03 13876 - US Patent Pending.

[CEF02] Constantinos A. Constantinides, Tzilla Elrad, and Mohamed E. Fayad. Extending the object model to provide explicit support for crosscutting concerns. *Softw. Pract. Exper.*, 32(7):703–734, 2002.

[CG04] Tal Cohen and Joseph (Yossi) Gil. Aspectj2ee = aop + j2ee: Towards an aspect based, programmable and extensible middleware framework. In Martin Odersky, editor, *ECOOP 2004 - Object-Oriented Programming, 18th European Conference*, volume 3086 of *Lecture Notes in Computer Science*, pages 219–243. Springer-Verlag, 2004.

[CG07] Tal Cohen and Joseph Gil. Better construction with factories. *Journal of Object Technology*, 6(6):109–129, 2007.

[CH05] D. Caromel and L. Henrio. *A Theory of Distributed Object*. Springer-Verlag, 2005.

[CHL08] Denis Caromel, Ludovic Henrio, and Mario Leyton. Type safe algorithmic skeletons. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 45–53, Toulouse, France, February 2008. IEEE CS Press.

[CHS04] D. Caromel, L. Henrio, and B. Serpette. Asynchronous and deterministic objects. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 123–134, New York, NY, USA, 2004. ACM Press.

[CL07] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *13th International Euro-Par Conference: Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer-Verlag, 2007.

[CL08] Denis Caromel and Mario Leyton. A transparent non-invasive file data model for algorithmic skeletons. In *22nd International Par-*

allel and Distributed Processing Symposium (IPDPS), pages 1–8, Miami, USA, March 2008. IEEE Computer Society.

[CLKRR05] Natalia Currle-Linde, U. Küster, Michael M. Resch, and B. Risio. Science experimental grid laboratory (segl) dynamic parameter study in distributed systems. In *PARCO*, pages 49–56, 2005.

[CMSL04] E. Cesar, J. G. Mesa, J. Sorribes, and E. Luque. Modeling master-worker applications in poetries. *hips*, 00:22–30, 2004.

[Col91] Murray Cole. *Algorithmic skeletons: structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[Col04] Murray Cole. Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. *Parallel Computing*, 30(3):389–406, 2004.

[Cor06] CoreGRID, Programming Model Institute. Basic features of the grid component model (assessed). Technical report, 2006. Deliverable D.PM.04, `http://www.coregrid.net/mambo/images/stories/Deliverables/d.pm.04.pdf`.

[Dan99] Marco Danelutto. Dynamic run time support for skeletons. Technical report, 1999.

[Dan01] Marco Danelutto. Efficient support for skeletons on workstation clusters. *Parallel Processing Letters*, 11(1):41–56, 2001.

[Dan05] Marco Danelutto. Qos in parallel programming through application managers. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP'05)*, pages 282–289, Washington, DC, USA, 2005. IEEE Computer Society.

[DBCG05] Jan Dünnweber, Anne Benoit, Murray Cole, and Sergei Gorlatch. Integrating mpi-skeletons with web services. [JNP+05], pages 787–794.

[dCJL03] Francisco Heron de Carvalho Junior and Rafael Dueire Lins. Topological skeletons in haskell#. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 53.1, Washington, DC, USA, 2003. IEEE Computer Society.

[DD05] Marco Danelutto and Patrizio Dazzi. A Java/Jini framework supporting stream parallel computations. In *Proc. of Intl. PARCO 2005: Parallel Computing*, September 2005.

[DD06a] Marco Danelutto and Patrizio Dazzi. Joint structured/non structured parallelism exploitation through data flow. In V. Alexandrov, D. van Albada, P. Sloot, and J. Dongarra, editors, *Proc. of ICCS: International Conference on Computational Science, Workshop on Practical Aspects of High-level Parallel Programming*, LNCS, Reading, UK, May 2006. Springer Verlag.

[DD06b] Marco Danelutto and Patrizio Dazzi. Joint structured/unstructured parallelism exploitation in Muskel. In *Proc. of ICCS 2006 / PAPP 2006*, LNCS. Springer Verlag, May 2006. to appear.

[DDDCG01] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object reclassification. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 130–149, London, UK, 2001. Springer-Verlag.

[DDDCG02] Sophia Drossopoulou, Ferruccio Damiani, Mariangiola Dezani-Ciancaglini, and Paola Giannini. More dynamic object reclassification: Fickle∥. *ACM Trans. Program. Lang. Syst.*, 24(2):153–191, 2002.

[DDdSL02] Pasqua D'Ambra, Marco Danelutto, Daniela di Serafino, and Marco Lapegna. Advanced environments for parallel and distributed applications: a view of current status. *Parallel Computing*, 28(12):1637–1662, 2002.

[DEDG06] Catalin Dumitrescu, Dick Epema, Jan Dünnweber, and Sergei Gorlatch. Reusable cost-based scheduling of grid workflows operating on higher-order components. Technical Report TR-0044, Institute on Resource Management and Scheduling, CoreGRID - Network of Excellence, July 2006.

[DFH+93] John Darlington, A. J. Field, Peter G. Harrison, Paul H. J. Kelly, D. W. N. Sharp, and Q. Wu. Parallel programming using skeleton functions. In *PARLE '93: Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, pages 146–160, London, UK, 1993. Springer-Verlag.

[DG04] Jan Dünnweber and Sergei Gorlatch. Hoc-sa: A grid service architecture for higher-order components. In *International Conference on Services Computing, Shanghai, China*. IEEE Computer Society Press, September 2004.

[DG08] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[DGB+06] Jan Dünnweber, Sergei Gorlatch, Francoise Baude, Virginie Legrand, and Nikos Parlavantzas. Towards automatic creation of web services for grid component composition. In Vladimir Getov, editor, *Proceedings of the Workshop on Grid Systems, Tools and Environments, 12 October 2005, Sophia Antipolis, France*, number TR-0043, December 2006.

[DGC+05] Jan Dünnweber, Sergei Gorlatch, Sonia Campa, Marco Aldinucci, and Marco Danelutto. Using code parameters for component adaptations. In Sergei Gorlatch and Marco Danelutto, editors, *Proc. of the Integrated Research in Grid Computing Workshop*, volume

TR-05-22, Pisa, Italy, November 2005. Universitý di Pisa, Dipartimento di Informatica.

[DGTY95] John Darlington, Yike Guo, Hing Wing To, and Jin Yang. Functional skeletons for parallel coordination. In *Euro-Par '95: Proceedings of the First International Euro-Par Conference on Parallel Processing*, pages 55–66, London, UK, 1995. Springer-Verlag.

[DkGTY95] John Darlington, Yi ke Guo, Hing Wing To, and Jin Yang. Parallel skeletons for structured composition. In *PPOPP '95: Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 19–28, New York, NY, USA, 1995. ACM.

[DPP97] Marco Danelutto, Fabrizio Pasqualetti, and Susanna Pelagatti. Skeletons for data parallelism in p3l. In *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*, pages 619–628, London, UK, 1997. Springer-Verlag.

[DRR+05] Manuel Diaz, Sergio Romero, Bartolome Rubio, Enrique Soler, and Jose M. Troya. An aspect oriented framework for scientific component development. In *PDP '05: Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, pages 290–296, Washington, DC, USA, 2005. IEEE Computer Society.

[DRR+06a] Manuel Diaz, Sergio Romero, Bartolome Rubio, Enrique Soler, and Jose M. Troya:. Dynamic reconfiguration of scientific components using aspect oriented programming: A case study. In Robert Meersman and Zahir Tari, editors, *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, volume 4276 of *Lecture Notes in Computer Science*, pages 1351–1360. Springer-Verlag, 2006.

[DRR+06b] Manuel Díaz, Sergio Romero, Bartolomé Rubio, Enrique Soler, and José M. Troya. Using sbasco to solve reaction-diffusion equations in two-dimensional irregular domains. In *Practical Aspects of High-Level Parallel Programming (PAPP), affiliated to the International Conference on Computational Science (ICCS)*, volume 3992 of *Lecture Notes in Computer Science*, pages 912–919. Springer, 2006.

[DRST04] Manuel Díaz, Bartolomé Rubio, Enrique Soler, and José M. Troya. Sbasco: Skeleton-based scientific components. In *PDP*, pages 318–. IEEE Computer Society, 2004.

[DS00] Marco Danelutto and Massimiliano Stigliani. Skelib: Parallel programming with skeletons in c. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 1175–1184, London, UK, 2000. Springer-Verlag.

[DT02] Marco Danelutto and Paolo Teti. Lithium: A structured parallel programming enviroment in Java. In *Proc. of ICCS: International Conference on Computational Science*, volume 2330 of *LNCS*, pages 844–853. Springer Verlag, April 2002.

[EAC98] S. Ehmety, I. Attali, and D. Caromel. About the automatic continuations in the eiffel model. In *International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA'98*, Las Vegas, USA., 1998. CSREA.

[EHKT05] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. A compositional framework for developing parallel programs on two dimensional arrays. Technical report, Department of Mathematical Informatics, University of Tokyo, 2005.

[EMHT07] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Domain-specific optimization strategy for skeleton programs. In Anne-Marie Kermarrec, Luc Bougé, and Thierry Priol, editors, *Euro-Par*, volume 4641 of *Lecture Notes in Computer Science*, pages 705–714. Springer, 2007.

[Emm00] Wolfgang Emmerich. *Engineering distributed objects,*. John Wiley & Sons Ltd., 2000.

[ES01] Dietmar W. Erwin and David F. Snelling. Unicore: A grid computing environment. In *Lecture Notes in Computer Science*, volume 2150, pages 825–834. Springer, 2001.

[FK99] Ian Foster and Carl Kesselman, editors. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[For] CCA Forum. Common component architecture. `http://www.cca-forum.org`.

[FS07] Joel Falcou and Jocelyn Sérot. Formal semantics applied to the implementation of a skeleton-based parallel programming library. In G. R. Joubert, C. Bischof, F. J. Peters, T. Lippert, M. Bücker, P. Gibbon, and B. Mohr, editors, *Parallel Computing: Architectures, Algorithms and Applications (Proc. of PARCO 2007, Julich, Germany)*, volume 38 of *NIC*, pages 243–252, Germany, September 2007. John von Neumann Institute for Computing.

[FSCL06] J. Falcou, J. Sérot, T. Chateau, and J. T. Lapresté. Quaff: efficient c++ design for parallel skeletons. *Parallel Computing*, 32(7):604–615, 2006.

[FSP06] J. F. Ferreira, J. L. Sobral, and A. J. Proenca. Jaskel: A java skeleton-based framework for structured cluster and grid computing. In *CCGRID '06: Proceedings of the Sixth IEEE International Symposium on Cluster Computing and the Grid*, pages 301–304, Washington, DC, USA, 2006. IEEE Computer Society.

[GGLD06] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk Synchronous Parallel ML with Exceptions. In Peter Kacsuk, Thomas Fahringer, and Zsolt Nemeth, editors, *Distributed and Parallel Systems (DAPSYS 2006)*, pages 33–42. Springer, 2006.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[GL03] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[GL05] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.

[GL07] L. Gesbert and F. Loulergue. Semantics of an Exception Mechanism for Bulk Synchronous Parallel ML. In *International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, pages 201–208. IEEE Computer Society, 2007.

[Gor04] Sergei Gorlatch. Send-receive considered harmful: Myths and realities of message passing. *ACM Trans. Program. Lang. Syst.*, 26(1):47–56, 2004.

[Gri] Grid5000. Official web site. http://www.grid5000.fr.

[Gro99] Object Management Group. The common object request broker: Architecture and specification, 12 1999.

[GSP99] Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Preiss. Using object-oriented techniques for realizing parallel architectural skeletons. In *ISCOPE '99: Proceedings of the Third International Symposium on Computing in Object-Oriented Parallel Environments*, Lecture Notes in Computer Science, pages 130–141, London, UK, 1999. Springer-Verlag.

[GSP02] Dhrubajyoti Goswami, Ajit Singh, and Bruno R. Preiss. From desgign patterns to parallel architectural skeletons. *J. Parallel Distrib. Comput.*, 62(4):669–695, 2002.

[GVC06] Horacio González-Vélez and Murray Cole. Towards fully adaptive pipeline parallelism for heterogeneous distributed environments. In *Parallel and Distributed Processing and Applications, 4th International Symposium (ISPA)*, Lecture Notes in Computer Science, pages 916–926. Springer-Verlag, 2006.

[GVC07] Horacio González-Véléz and Murray Cole. Adaptive structured parallelism for computational grids. In *PPoPP '07: Proceedings of*

the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 140–141, New York, NY, USA, 2007. ACM.

[GVC08] Horacio Gonzalez-Velez and Murray Cole. An adaptive parallel pipeline pattern for grids. In *22nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–8, Miami, USA, March 2008. IEEE Computer Society.

[Ham99] M. Hamdan. A survey of cost models for algorithmic skeletons. Technical report, Department of Computing and Electrical Engineering, Heriot-Watt University, UK, 1999.

[Her00] Christoph Armin Herrmann. *The Skeleton-Based Parallelization of Divide-and-Conquer Recursions*. PhD thesis, 2000. ISBN 3-89722-556-5.

[HHVOM07] Mercedes Hidalgo-Herrero, Alberto Verdejo, and Yolanda Ortega-Mallén. Using Maude and its strategies for defining a framework for analyzing Eden semantics. 174(10):119–137, 2007.

[HJW+92] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, 1992.

[HL00] C. A. Herrmann and C. Lengauer. Hdc: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2–3):239–250, 2000.

[HM99] Kevin Hammond and Greg Michelson, editors. *Research Directions in Parallel Functional Programming*. Springer-Verlag, London, UK, 1999.

[Int] Intel. Threading building blocks. `http://www.threadingbuildingblocks.org`.

[JNP+05] Gerhard R. Joubert, Wolfgang E. Nagel, Frans J. Peters, Oscar G. Plata, P. Tirado, and Emilio L. Zapata, editors. *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005, 13-16 September 2005, Department of Computer Architecture, University of Malaga, Spain*, volume 33 of *John von Neumann Institute for Computing Series*. Central Institute for Applied Mathematics, Jülich, Germany, 2005.

[KC02] Herbert Kuchen and Murray Cole. The integration of task and data parallel skeletons. *Parallel Processing Letters*, 12(2):141–155, 2002.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[KLPR01] Ulrike Klusik, Rita Loogen, Steffen Priebe, and Fernando Rubio. Implementation skeletons in eden: Low-effort parallel programming. In *IFL '00: Selected Papers from the 12th International Workshop on Implementation of Functional Languages*, pages 71–88, London, UK, 2001. Springer-Verlag.

[KS02] Herbert Kuchen and Jörg Striegnitz. Higher-order functions and partial applications for a c++ skeleton library. In *JGI '02: Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 122–130, New York, NY, USA, 2002. ACM Press.

[KS05] Herbert Kuchen and Jörg Striegnitz. Features from functional programming for a c++ skeleton library. *Concurrency - Practice and Experience*, 17(7-8):739–756, 2005.

[Kuc02] Herbert Kuchen. A skeleton library. pages 620–629, 2002.

[Kuc04] Herbert Kuchen. *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, chapter Optimizing Sequences of Skeleton Calls, pages 254–273. Springer-Verlag, 2004.

[Läm07] Ralf Lämmel. Google's MapReduce Programming Model – Revisited. Accepted for publication in the Science of Computer Programming Journal; Online since 2 January, 2006; 42 pages, 2006–2007.

[LBGLR06] F. Loulergue, R. Benheddi, F. Gava, and D. Louis-Regis. Bulk Synchronous Parallel ML: Semantics and Implementation of the Parallel Juxtaposition. In *International Computer Science Symposium in Russia (CSR 2006)*, volume 3967 of *LNCS*, pages 475–486. Springer, 2006.

[LHF00] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[LM06] Li Li and A. Malony. Model-based performance diagnosis of master-worker parallel computations. In *Proceedings of the 12th International Euro-Par Conference: Parallel Processing*, volume 4128 of *LNCS*, pages 35–46, Dresden, Germany, August 2006. Springer-Verlag.

[LOmRP05] Rita Loogen, Yolanda Ortega-mallén, and na-marí Ricardo Pe˙ Parallel functional programming in eden. *Journal of Functional Programming*, 15(3):431–475, 2005.

[Lou01] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.

[Lou03] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszorményi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

[MAB+02] S. MacDonald, J. Anvik, S. Bromling, J. Schaeffer, D. Szafron, and K. Tan. From patterns to frameworks to parallel programs. *Parallel Comput.*, 28(12):1663–1683, 2002.

[MAD07] Marco Danelutto Marco Aldinucci and Patrizio Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), December 2007. To appear.

[Mau95] Michel Mauny. Functional programming using caml light. Technical report, INRIA, 1995.

[MHT03] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallelization with tree skeletons. In Harald Kosch, László Böszörményi, and Hermann Hellwagner, editors, *Euro-Par*, volume 2790 of *Lecture Notes in Computer Science*, pages 789–798. Springer, 2003.

[MHT06] Kiminori Matsuzaki, Zhenjiang Hu, and Masato Takeichi. Parallel skeletons for manipulating general trees. *Parallel Comput.*, 32(7):590–603, 2006.

[Mica] Microsoft. .net framework. http://www.microsoft.com/net/.

[MICb] Sun MICROSYSTEMS. Enterprise java beans. `http://java.sun.com/producs/ejb/`.

[Micc] Sun Microsystems. Java. `http://java.sun.com`.

[MIEH06] Kiminori Matsuzaki, Hideya Iwasaki, Kento Emoto, and Zhenjiang Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, page 13, New York, NY, USA, 2006. ACM.

[MKI+04] Kiminori Matsuzaki, Kazuhiko Kakehi, Hideya Iwasaki, Zhenjiang Hu, and Yoshiki Akashi. A fusion-embedded skeleton library. In Marco Danelutto, Marco Vanneschi, and Domenico Laforenza, editors, *Euro-Par*, volume 3149 of *Lecture Notes in Computer Science*, pages 644–653. Springer, 2004.

[MMS00] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. A pattern language for parallel application programs (research

note). In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 678–681, London, UK, 2000. Springer-Verlag.

[MMS01] Berna L. Massingill, Timothy G. Mattson, and Beverly A. Sanders. Parallel programming with a pattern language. *STTT*, 3(2):217–234, 2001.

[Mor06] Matthieu Morel. *Components for Grid Computing*. PhD thesis, Université de Nice-Sophia Antipolis, November 2006.

[(MP96] Message Passing Interface Forum (MPIF). Mpi-2: Extensions to the message-passing interface. Technical report, University of Tennessee, Knoxville, 1996.

[MSS99] Steve MacDonald, Duane Szafron, and Jonathan Schaeffer. Pattern-based object-oriented parallel programming. In *Proceedings of the 5th USENIX Conference on Object-Oriented Technology and Systems*, pages 29–49, San Diego, California, USA, May 1999.

[MSS+02] S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns. In *ASE '02: Proceedings of the 17th IEEE international conference on Automated software engineering*, page 23, Washington, DC, USA, 2002. IEEE Computer Society.

[MSSB00] Steve MacDonald, Duane Szafron, Jonathan Schaeffer, and Steven Bromling. Generating parallel program frameworks from parallel design patterns. In *Euro-Par '00: Proceedings from the 6th International Euro-Par Conference on Parallel Processing*, pages 95–104, London, UK, 2000. Springer-Verlag.

[OAS07] OASIS. Web services business process execution language version 2.0. http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.pdf, April 2007.

[Ora01] A. Oram. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, March 2002.

[PK05a] Michael Poldner and Herbert Kuchen. On implementing the farm skeleton. In *Proceedings of the 3rd International Workshop HLPP 2005*, 2005.

[PK05b] Michael Poldner and Herbert Kuchen. Scalable farms. In *Proceedings of the International Conference ParCo*, pages 795–802, 2005.

[PK06] Michael Poldner and Herbert Kuchen. Algorithmic skeletons for branch & bound. In *In Proceedings of 1st International Conference on Software and Data Technologies, ICSOFT*, volume 1, pages 291–300. INSTICC Press, 2006.

[PK08a] Michael Poldner and Herbert Kuchen. On implementing the farm skeleton. *Parallel Processing Letters*, 18(1):117–131, 2008.

[PK08b] Michael Poldner and Herbert Kuchen. Skeletons for divide and conquer algorithms. In *Proceedings of Parallel and Distributed Computing and Networks, PDCN*, 2008.

[Pri06] Steffen Priebe. Dynamic task generation and transformation within a nestable workpool skeleton. In *Proceedings of the 12th International Euro-Par Conference: Parallel Processing*, volume 4128 of *LNCS*, pages 615–624, Dresden, Germany, August 2006. Springer.

[Pro] ProActive. `http://proactive.objectweb.org`.

[Pro07] EchoGRID Project. D.3.1 Plugtests Event Reports, 2007.

[PSD+04] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, Gérard Florin, Fabrice Legond-Aubry, and Laurent Martelli. Jac: an aspect-based distributed dynamic framework. *Softw. Pract. Exper.*, 34(12):1119–1148, 2004.

[RG03] Fethi A. Rabhi and Sergei Gorlatch, editors. *Patterns and skeletons for parallel and distributed computing*. Springer-Verlag, London, UK, 2003.

[RT+08] Nadia Ranaldo, Giancarlo Tretola, , Eugenio Zimeo, Natalia Currle-Linde, Wasseim AL Zouab, Oliver Mangold, Michael Resch, Denis Caromel, Alexandre di Costanzo, Christian Delbé, Johann Fradj, Mario Leyton, and Jean-Luc Scheefer. A Scheduler for a Multi-paradigm Grid Environment. Technical Report 155, CoreGrid, July 2008.

[RV07a] Adrián Riesco and Alberto Verdejo. Distributed applications implemented in Maude with parameterized skeletons. 4468:91–106, 2007.

[RV07b] Adrián Riesco and Alberto Verdejo. Parameterized skeletons in Maude. Technical Report SIC-1/07, Dpto. Sistemas Informáticos y Computación, Universidad Complutense de Madrid, January 2007.

[SG02] J. Sérot and D. Ginhac. Skeletons for parallel image processing : an overview of the SKiPPER project. *Parallel Computing*, 28(12):1785–1808, Dec 2002.

[SGD99] J. Sérot, D. Ginhac, and J.P. Dérutin. SKiPPER: a skeleton-based parallel programming environment for real-time image processing applications. In V. Malyshkin, editor, *5th International Conference on Parallel Computing Technologies (PaCT-99)*, volume 1662 of *LNCS*, pages 296–305. Springer, 6–10 September 1999.

[SNM+02] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. Overview of gridrpc: A

remote procedure call api for grid computing. In *GRID '02: Proceedings of the Third International Workshop on Grid Computing*, pages 274–278, London, UK, 2002. Springer-Verlag.

[SP07] J. Sobral and A. Proenca. Enabling jaskel skeletons for clusters and computational grids. In *IEEE Cluster*. IEEE Press, 9 2007.

[Szy98] Clemens Szyperski. *Component software: beyond object-oriented programming*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1998.

[TA90] B. H. Tay and A. L. Ananda. A survey of remote procedure calls. *SIGOPS Oper. Syst. Rev.*, 24(3):68–79, 1990.

[Tak] Takaken. N queens problem. http://www.ic-net.or.jp/home/takaken/e/queen/.

[Tan07] Éric Tanter. On dynamically-scoped crosscutting mechanisms. *ACM SIGPLAN Notices*, 42(2):27–33, February 2007.

[TE05a] INRIA OASIS Team and ETSI. 2nd Grid Plugtests Report, 2005. `http://www-sop.inria.fr/oasis/plugtest2005/2ndGridPlugtestsReport.pdf`.

[TE05b] INRIA OASIS Team and ETSI. Grid Plugtest: Interoperability on the Grid. Grid Today online, January 2005.

[TE06] INRIA OASIS Team and ETSI. 3rd Grid Plugtest Report, 2006. `http://www-sop.inria.fr/oasis/plugtest2006/plugtests_report_2006.pdf`.

[top] Top 500. http://www.top500.org/.

[TSS+03] Kai Tan, Duane Szafron, Jonathan Schaeffer, John Anvik, and Steve MacDonald. Using generative design patterns to generate parallel code for a distributed memory environment. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 203–215, New York, NY, USA, 2003. ACM.

[TT06] Éric Tanter and Rodolfo Toledo. A versatile kernel for distributed AOP. In *Proceedings of the IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS 2006)*, volume 4025 of *Lecture Notes in Computer Science*, pages 316–331. Springer-Verlag, June 2006.

[TZ07a] Giancarlo Tretola and Eugenio Zimeo. Activity pre-scheduling for run-time optimisation of grid workflows. *Journal of Systems Architecture (JSA) (to appear)*, 2007.

[TZ07b] Giancarlo Tretola and Eugenio Zimeo. Activity pre-scheduling in grid workflows. In *Proceedings of the 15th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP 2007)*, pages 63–71, Naples (Italy), 2007. IEEE Computer Society.

[Uni] Unicore. http://www.unicore.org.

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[vLAG+03] Gregor von Laszewski, Beulah Alunkal, Jarek Gawor, Ravi Madhuri, Pawel Plaszczak, and Xian-He Sun. A File Transfer Component for Grids. In H.R. Arabnia and Youngson Mun, editors, *Proceedings of the International Conferenece on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 24–30. CSREA Press, 2003.

[vLFGL01] Gregor von Laszewski, Ian Foster, Jarek Gawor, and Peter Lane. A Java commodity grid kit. *Concurrency and Computation: Practice and Experience*, 13(8–9):645–662, /2001.

[vLGP+04] Gregor von Laszewski, Jarek Gawor, Pawel Plaszczak, Mike Hategan, Kaizar Amin, Ravi Madduri, and Scott Gose. An overview of grid file transfer patterns and their implementation in the java cog kit. *Neural, Parallel Sci. Comput.*, 12(3):329–352, 2004.

[Wik] Wikipedia. Multi-core. http://en.wikipedia.org/wiki/Multi-core.

[YB05] Jia Yu and Rajkumar Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[YCGH07] Gagarine Yaikhom, Murray Cole, Stephen Gilmore, and Jane Hillston. A structural approach for modelling performance of systems using skeletons. *Electr. Notes Theor. Comput. Sci.*, 190(3):167–183, 2007.

[Yel08] Katherine Yelick. Keynote: Programming models for petascale to exascale. In *22nd International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–1, Miami, USA, March 2008. IEEE Computer Society.

[Zav01] Andrea Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–407, 2001.

Advanced Features for Algorithmic Skeleton Programming

Abstract

This thesis proposes a model for algorithmic skeleton programming. The model focuses on programming abstractions which offer *minimal conceptual disruption* for non-parallel programmers and *showing the pay-back*. The model aims towards a library implementation, and therefore focuses on problems and opportunities which arise by having skeletons as libraries instead of languages.

In summary, this thesis presents a model for algorithmic skeleton programming and its implementation in Java: *Calcium*. Among others, Calcium features nestable task and data parallel skeletons, and supports the execution of skeleton applications on several parallel and distributed infrastructures. In other words, Calcium provides a single way of writing skeleton programs which can be deployed and executed on different parallel and distributed infrastructures.

Calcium provides three main contributions to algorithmic skeleton programming. First, a performance tuning model which helps programmers identify code responsible for performance bugs. Second, a type system for nestable skeletons which is proven to guaranty subject reduction properties and is implemented using Java Generics. Third, a transparent algorithmic skeleton file access model which enables skeletons for data intensive applications.

Keywords: Algorithmic Skeletons, Performance Tuning, Type Systems, File Transfer

---

Caractéristiques Avancées pour la programmation avec des squelettes algorithmiques

Résumé

Cette thèse propose un modèle de programmation basé sur le concept de squelettes algorithmiques. Ce modèle se concentre sur les abstractions de programmation parallèle qui offrent une conception proche de celle des application non-parallèles permettant de simplifier leur développment. Ce modèle est destiné à être implémenté sous forme de librairie, et par conséquent se concentre sur les problèmes et les opportunités qu'offre cette approche plutôt qu'une approche orienté langage.

En résumé, cette thèse présente un modèle de programmation basé sur les squelettes algorithmiques et son implémentation en Java, nommé Calcium. Calcium supporte entre autre l'utilisation de squelettes hiérarchiques, les squelettes exploitant le parallélisme de données ainsi que l'exécution de ces applications sur différentes infrastructures parallèles et distribuées. En d'autres termes, Calcium unifie la façon d'écrire les programmes basés sur les squelettes et permet le déploiement et l'exécution de ceux-ci sur différentes infrastructures parallèles et distribuées.

Calcium apporte trois contributions principales dans le domaine de la programmation basé sur les squelettes. Premièrement, un modèle d'optimisation des performances qui permet aux programmeurs d'identifier les parties de codes responsables d'une dégradation des performances. Deuxièmement, un système de typage prouvé, pour squelettes hiérarchiques, qui garantie les propriétés de réduction et est implémenté a l'aide du typage générique de Java. Troisièmement, un modèle d'accès aux fichiers qui permet l'utilisation de squelettes dans les applications traitant des données de façon intense.

Mots-clefs : Squelettes Algorithmiques, Réglage de Performance, Système de Typage, Transfert de Fichiers