Patterns and Skeletons for Parallel and Distributed Computing

Springer-Verlag London Ltd.

Fethi A. Rabhi and Sergei Gorlatch (Eds)

Patterns and Skeletons for Parallel and Distributed Computing

Springer

Fethi A. Rabhi, PhD
School of Information Systems, The University of New South Wales, Sydney 2052, Australia

Sergei Gorlatch, PhD
Technical University of Berlin, Sekr, FR5-6, Franklinstr. 28/29, D-10587, Berlin, Germany

British Library Cataloguing in Publication Data
Patterns and skeletons for parallel and distributed computing
1. Electronic data processing – Distributed processing
2. Parallel processing (Electronic computers)
I. Rabhi, Fethi II. Gorlatch, Sergei
004.3'6
ISBN 978-1-85233-506-9

Library of Congress Cataloging-in-Publication Data
Patterns and skeletons for parallel and distributed computing / Fethi Rabhi and Sergei Gorlatch (eds.).
p. cm.
Includes bibliographical references and index.
ISBN 978-1-85233-506-9 ISBN 978-1-4471-0097-3 (eBook)
DOI 10.1007/978-1-4471-0097-3
1. Parallel processing (Electronic computers) 2. Electronic data processing – Distributed processing. I. Rabhi, Fethi. II. Gorlatch, Sergei.
QA76.58.P385 2002
004'.36-dc21 2002067023

ISBN 978-1-85233-506-9

http://www.springer.co.uk

Originally published by Springer-Verlag London Berlin Heidelberg in 2003

Typesetting: Camera-ready by editors

34/3830-543210 Printed on acid-free paper SPIN 10837441

Foreword

Murray Cole

Computer systems research exhibits a long-standing ambivalence towards parallelism. On the one hand, concurrent activity within and between computational entities is recognised as an entirely normal and desirable phenomenon, both in the natural world and in artificial technologies. On the other, the specification, control and analysis of such activity is perceived as being inherently hard. The application of abstraction is one of our discipline's defining activities and, with respect to parallelism, is often undertaken with a view to hiding that parallelism behind a more tractable illusion of sequentiality. Perhaps this should be no surprise, given that our everyday thoughts and strategies seem to enjoy a similarly sequential gloss over undoubtedly parallel implementations.

The thesis underlying and uniting the systems covered here is simple. It proposes that we can tame the complexity of parallelism by recognising that for many practical purposes its manifestation is far from unpredictable, but in fact follows a number of recurring forms. We can then make progress by recognising, abstracting (of course!) and cataloguing these forms and their exploitation. If we can hide the underlying parallelism entirely, then so much the better. The striking and encouraging message of this book is that the approach can been applied in contexts as superficially diverse as "System on Chip" architectures (just what should we do, and how, with a billion transistors?) and the emerging global "Computational Grids". That its discovery and investigation has thus far proceeded more or less independently in these contexts strengthens the case for its importance.

The book is timely, arriving at an opportune moment to act as a catalyst to fruitful interaction between the skeletons and patterns research communities (and indeed to others, as yet unrecognised, who may be pursuing the same underlying approach). Meanwhile, the increasing mainstream prevalence of parallel and distributed systems, whether explicitly acknowledged or given a veil of familiarity and respectability as "servers" and the like, means that issues which were once matters of concern to a fringe audience are increasingly central to the activities of everyday programmers. The challenges inherent in organising a collection of threads within a multi-CPU chip or a set of agents across the Internet are related.

As will be clear from their own contributions, Fethi and Sergei are long standing and respected members of our research community. It would be impossible in practice to provide encyclopaedic coverage of this fast moving area. Thus, in selecting and coordinating the contributions to this volume they have aimed

for, and produced, a thoroughly representative sample of ongoing research for which they and the contributing authors should be congratulated and thanked.

Looking to the future, one of the most awkward questions one can ask of a research programme is "Does this actually matter?" (as distinct from "Does it matter if this matters?"). That the expression and control of parallelism within computer systems will matter increasingly seems uncontentious. Perhaps we can speculate on the ultimate answer by considering that the alternatives to the programme described here seem to involve either the belief that what is currently perceived to be hard will soon be revealed as easy, or that dramatic progress in the automatic extraction and management of parallelism from sequential specifications will bring us to a point at which the issue disappears.

About the Author

Murray Cole was awarded a BSc and a PhD in Computer Science from the University of Edinburgh in 1984 and 1988. After holding a lecturership and a SERC postdoctoral research fellowship at Glasgow University, he has been a Lecturer then Senior Lecturer in the Division of Informatics at the University of Edinburgh since 1990.

His book *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation* (Pitman 1989), adapted from his PhD thesis, is still regularly cited as seminal in the field of algorithmic skeletons. He has over 20 other publications in reputable books, journals and conferences. He is the founder and maintainer of an international mailing list and online archive on "algorithmic skeletons". He was also the co-organiser of two workshops at the Dagstuhl research centre in Germany on Higher-Order Parallel Programming in 1997 and 1999. He has acted as an academic host for numerous visits and talks by external researchers (including several through the EC TRACS scheme). His current research interests concern the design and implementation of programming language constructs which facilitate the structured expression and analysis of parallel programs, notably through the use of algorithmic skeletons.

Preface

Fethi Rabhi and Sergei Gorlatch

This preface explains the motivations behind the book. First, it identifies the essential characteristics of parallel and distributed applications. It goes on to take a closer look at the development cycle of such applications and outlines the need for integrated approaches that facilitate the reuse of ideas and techniques. Then, it defines the terms "skeletons" and "patterns" and their role in the development cycle. Finally, it discusses opportunities for cross-fertilisation between the two disciplines and outlines this book's contribution towards this goal.

Parallel and Distributed Computing

In recent years, there has been considerable interest in parallel and distributed applications, mainly due to the availability of low-cost hardware and fast computer networks. This is illustrated by the dramatic increase in the use of the Internet and Internet-based applications. In very general terms, a parallel or distributed application can be defined as "a system of several independent software components cooperating in a common purpose or to achieve a common goal", including:

- parallel and high-performance applications (e.g. solving PDE equations)
- fault-tolerant applications and real-time systems (e.g. safety-critical process control)
- applications using functional specialisation (e.g. enterprise information and optimisation systems)
- inherently distributed applications (e.g. Web applications)

Each of these classes has historically emerged from a distinct computing discipline, such as operating systems, networks, high-performance computing, databases and real-time systems. Experiences with and also techniques and tools for software development are usually adapted to the particular requirements of the relevant discipline. However, there is much to learn from adapting concepts from one discipline to another since there are many common problems,

such as specifying the interaction between concurrent activities or mapping a process graph onto a given architectural platform. Moreover, there are many applications which cannot be contained within a single discipline. For example, *metacomputations* are applications intended for both parallel architectures and distributed systems. Another example is *distributed multimedia applications* where real-time constraints often have to be dealt with in a distributed processing context.

Developing Parallel and Distributed Applications

Since the discipline of software engineering is concerned with the application of systematic approaches to the development, operation and maintenance of complex software systems, it can provide a framework for integrating many techniques related to the parallel and distributed software lifecycle. For the sake of simplicity, we confine ourselves to the well-known three phases of requirements analysis, design and implementation in the waterfall model, although issues such as maintenance and testing are still very important in this context. Figure 1 illustrates the additional needs and constraints that should be taken into account at different stages of the basic development cycle when considering parallel and distributed applications.

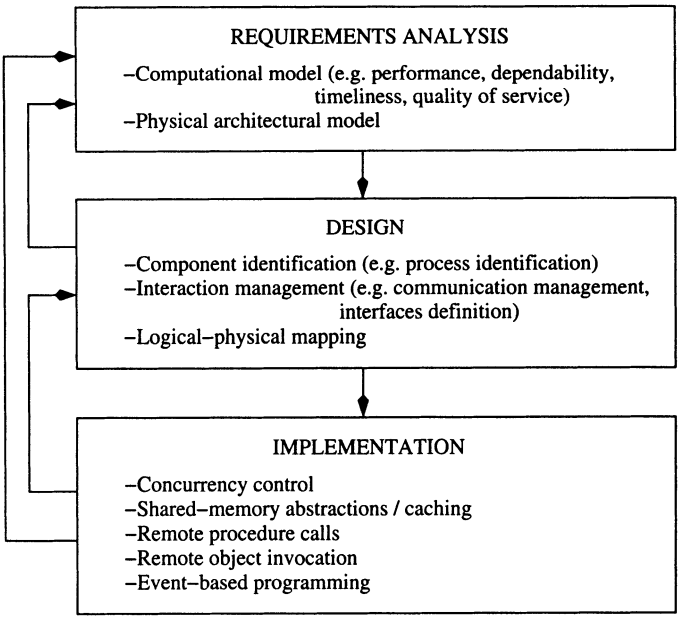

Figure 1: Parallel and distributed application development cycle

Requirements Analysis

Requirement analysis is the first phase in the development process. Here, the requirements for the system are established and specified in detail for further development. The amount of information specified should be minimal yet complete. A common requirement for any application is its *functionality* i.e. the functions it is supposed to perform. Parallel and distributed applications mostly differ in their *non-functional* requirements, some of which relate to the *dynamic behaviour* of the system in terms of its concurrent entities and the interaction between these entities. For these applications, the most important non-functional requirements can be grouped in the following categories:

- *Performance*: This is required for parallel and high-performance applications where maximum speedup and efficiency must be attained according to a specific architectural model.
- *Dependability* (or robustness): This includes availability, reliability, safety and security. These requirements are essential for fault-tolerant applications and those using functional specialisation.
- *Timeliness*: The system must satisfy the established temporal constraints. This is an essential feature of real-time systems.
- *Quality of service*: This is needed for applications using functional specialisation, particularly distributed multimedia. These requirements relate to the quality requirements on the collective behaviour of one or more processes. They are expressed in terms of both timeliness constraints and guarantees on measures of communication rate and latency, probabilities of communication disruptions, etc.
- *Dynamic change management*: The system must accommodate modifications or extensions dynamically. This is needed, for example, in *mobile systems* as the configuration of software components evolves over time.
- *Scalability*: The system must be able to scale up in size for example to cope with a larger database or file sizes, a bigger customer base, etc.

One of the outputs of requirements analysis is a set of system models called the *requirements specification*, which represents an abstraction of the system being studied and serves as a bridge between the analysis and design processes. Examples of formal models include the Calculus of Communicating Systems (CCS) and PetriNets, and examples of semi-formal models include data-flow diagrams and statecharts. Functional and non-functional requirements relate to what the program is supposed to achieve *(computational model)*.

In addition, there may be a specification of the hardware platform on which the program is to be executed *(physical architectural model)*. This is an essential requirement for parallel and inherently distributed applications. A hardware

platform can be homogeneous or heterogeneous. A homogeneous platform consists of a set of identical processing nodes connected by a single network. There are several abstract models for such platforms including the Parallel Random Access Machine (PRAM) and the Bulk Synchronous Parallel (BSP) model. A heterogeneous platform consists of several *resources* (e.g. processors, disks, sensors) connected through one or several high-speed networks. Representations for such platforms include UML Deployment Diagrams.

Software Design

Given certain system requirements and models, the design stage involves developing several more detailed models of the system at lower levels of abstraction. Considering parallel and distributed applications, the main concepts that need to be embodied in every design can be grouped in these three categories:

- *Structure and component identification*: This describes different components of the system, such as processes, modules and data and their abstractions.
- *Interaction management*: This considers the dynamic aspects and semantics of communication, e.g. defining interfaces and communication protocols between components, which components communicate with which, when and how communication takes place, contents of communication, etc.
- *Logical-physical mapping*: This defines the mapping of logical entities from the computational model to physical entities from the architectural model. Such mappings can be defined statically (decided at compile time) or dynamically (decided at run time).

Although strategies for designing sequential systems have been extensively studied, little is known about the design of parallel and distributed applications. Most existing design methods, e.g. UML, address some of the above issues from a real-time systems perspective.

Implementation

The implementation stage consists of transforming the software design models into a set of programs or modules. Parallel and distributed applications are characterised by a broad range of implementation approaches that operate at different levels of abstraction. Concurrency-control mechanisms such as locks, semaphores and monitors are low-level structures that were extensively studied in the 1970s and '80s. Shared-memory models such as Concurrent Read Exclusive Write (CREW) PRAM are relatively high-level structures that are required for shared-memory multiprocessing. Distributed-memory programming can be achieved through message-passing (e.g. MPI), remote procedure calls (e.g. OSF/DCE) or remote object invocations (e.g. CORBA). Event-based programming is also a useful abstraction, particularly for real-time and embedded systems.

What is a Skeleton?

The term *skeleton*, coined by Murray Cole, originates from the observation that many parallel applications share a common set of known interaction patterns such as pipelines, processor farms and data-parallel computations. The study of skeletons (as opposed to specific applications) has many advantages, such as offering higher-level programming interfaces, opportunities for formal analysis and transformation and the potential for "generic" implementations that are both portable and efficient.

Most of the work on skeletons is associated with functional languages, as skeletons can be modelled as higher-order functional structures. Among the variety of skeleton-related projects are those concerned with defining *elementary skeletons* from which parallel programs can be constructed. For example, the two well-known list processing operators *map* and *reduce* form a set of elementary skeletons with inherent parallelism. Despite the fact that equivalent operators are provided in most data-parallel languages (*map* corresponds to an element-wise operation and *reduce* to a reduction), the main advantage of using elementary skeletons is the availability of a formal framework for program composition. This allows a rich set of transformations (e.g. transforming one program into a more efficient one) to be applied. In addition, cost measures can be associated with these elementary skeletons and their compositions.

The main problem with elementary skeletons is that there is little guidance on how to compose them in order to obtain optimal efficiency. In addition, there have been very few practical implementations owing to the difficulty in adapting arbitrary composition structures to a variety of hardware platforms with very different characteristics. To address these problems, more elaborate skeletons that model complex interaction patterns can be defined. For example, the *divide-and-conquer* skeleton captures a well-known algorithmic design strategy for which several efficient implementations can be developed. Several such skeletons have been developed around specific data structures. These skeletons, which are derived from category theory, are known as *homomorphic skeletons*. They provide a similar level of abstraction to the data-parallel operators considered earlier, but in addition they offer a more formal framework for program construction and transformation. Homomorphic skeletons have been proposed for a variety of data structures such as lists, arrays, trees and graphs. They act in a similar way to an abstract data type by providing a set of known-to-be parallel operators while hiding the internal implementation details.

In the skeletons described so far, the communication structure is implied by the (often recursive) way operators are defined. There are skeletons which work around a fixed communication structure, e.g. the Static Iterative Transformation (SIT) skeleton, which captures a series of iterative transformations being applied to a large data structure and for which several programming environments have been proposed and implemented.

What is a Design Pattern?

The concept of a *design pattern* is related to a skeleton, but has consequences across several phases of the development cycle. When designing a new system (particularly a complex one), it is unusual for designers to tackle it by developing a solution from scratch. Instead, they often recall a similar problem that they have already solved and adapt its solution. The idea of design patterns, originally proposed by Gamma et al., is to facilitate the reuse of well-proven solutions based on experience in developing real systems. Given a library of common "patterns" for designing software, developers choose the pattern that is most suited to their needs. Patterns are often associated with object-oriented systems because they support reusability through classes and objects.

Patterns vary greatly in aims and scope. They offer solutions ranging from high-level strategies for organising software to low-level implementation mechanisms. The documentation of design patterns is informal and varies in the literature. In most descriptions, the information associated with the pattern (such as context, problem and solution) is presented in textual form or using UML diagrams.

Historically, most design patterns were identified by developers of object-oriented user interfaces whose main quality criteria were usability, extensibility and portability. However, there are a growing number of patterns which also express known concurrent behaviour of interacting entities over a possibly distributed platform. Examples include *Pipes and Filters*, *Master-Slave* and *Client-Dispatcher-Server*.

The design of a complex application typically involves more than one pattern. Besides design patterns, *implementation patterns* represent higher-level forms of programming abstractions. These patterns (called *idioms*) refer to commonly used language-dependent techniques which can be used to model the behaviour of interacting objects. Their description is informal and includes reusable code in the form of interfaces, classes and objects. Implementation patterns are being applied in a variety of contexts, from concurrent programming in Java to distributed programming in CORBA.

The similarities between these patterns and skeletons are striking. For example, the published Pipes-and-Filters and the Master-Slave patterns correspond to the well-known pipeline and farm skeletons. However, skeletons and patterns are different in many fundamental ways. While skeletons tend to be described in a formal way, patterns are usually loosely described in English and/or a combination of UML diagrams. Another difference is that a skeleton's "raison d'être" is in the design of high-performance systems, whereas behavioural patterns are more general. They tend to tackle other requirements specific to distributed systems such as fault-tolerance, timeliness and quality of service.

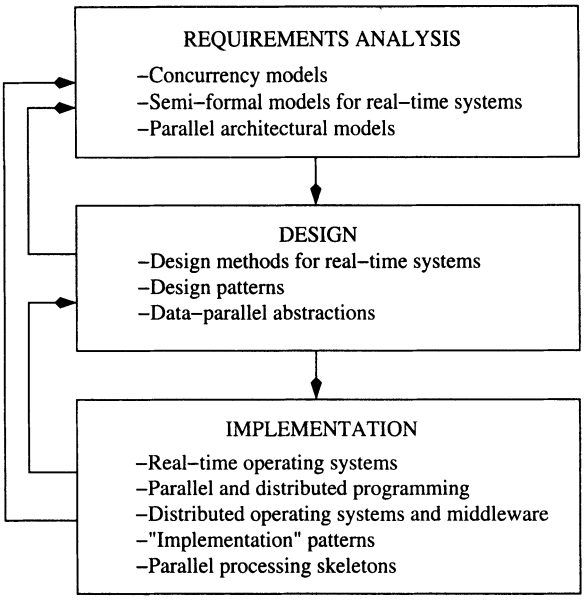

Figure 2: Role and main contribution of existing approaches in the application development cycle

Towards Integrated Approaches

Figure 2 summarises the role and main contribution of existing approaches and techniques in the parallel and distributed application development cycle. Despite their apparent disparity, we believe that there are several common issues, such as process management, communication and synchronization, distribution and mapping processes to processors. Despite the use of different notations and terminology, many similarities exist in areas such as semantics of communication, visual display of information and automatic code generation. As an example of "overlapping" of work, most programming abstractions provided in parallel processing tools are to a large extent already available in software engineering methodologies for real-time systems.

The other reason for integrated approaches is that, with the wider availability and greater ease of use of large computer networks, there will be several applications that cross boundaries. For example, a distributed real-time system may consist of a large number of identical tasks for dealing with the fault-tolerance requirements and for managing identical hardware devices. This requires suitable replication structures to model and implement the concurrency, communication and distribution aspects.

The main problems that still need addressing are:

- Little is known about design strategies for parallel and distributed applications. For example, existing design strategies are not very well suited

to non-functional requirements, and the logical-physical mapping is a neglected part of the development process.

- Many techniques rely on assumptions that are specific to the discipline they originate from, so there are many difficulties associated with adapting concepts from one discipline to another. For example, most structured/OO design methodologies do not provide replication structures, network studies for parallel computers are not relevant to distributed processing, etc.

Future work should concentrate on the *adaptation* of concepts across disciplines and the *integration* of these concepts within all phases of a well-defined development cycle. Considering adaptation, most efforts at the requirements stage have focused on the functional requirements and the dynamic behaviour of systems. New theories and models must be developed to express requirements such as quality of service, dynamic change management and dependability. Improved design abstractions and new ones are needed, e.g. with a capacity to model actors and intelligent agents capable of reactive, proactive and cooperative behaviour. There is also a need for new unified architectural models that can represent physical resources in terms of processors, memory, communication, etc. Finally, while middleware platforms (such as CORBA) have proved useful for applications with loosely coupled tasks and low communication requirements, their appropriateness for highly coordinated tasks that make large demands on communication and synchronisation still requires investigation.

The case for integration should give a greater role to CASE tools that emphasize the importance of formal notation, provide a rich set of design abstractions, allow model checking and provide automatic code generation. Integration of existing or new techniques should be achieved through formally defined, generic, reusable entities and their associated tools. Some of these entities have already been described in this paper as patterns and skeletons. This is not a new tendency but has already been happening to a large extent at the implementation level. For example, standards like CORBA and PVM can be regarded as "patterns" that support location transparency and decouple processes from the underlying communication mechanism. It is expected that similar approaches will be adopted at a much higher level in the development cycle.

Aims of the Book

While there is a profusion of books dedicated to particular languages or algorithms for parallel and distributed processing, there is a clear need for a contribution that centres on the higher-level approaches discussed above.

The aims of this book are two-fold. The first is to collect and publicise most of the work carried out by the skeletons community. At a time when parallel-processing research is perceived to be in decline, it is important for its community to make available important contributions that have implications

beyond the narrow context of parallel architectures. The second aim is to showcase other contributions (from the patterns community, for example) at the cutting edge of distributed systems design. One advantage of this approach is that it minimises the overlapping and duplication of work. Another is to encourage the cross-fertilisation of ideas between all communities involved in the high-level design and implementation of parallel and distributed applications.

Since this is potentially a very wide field, the book's scope is limited in many respects. First, it is not concerned with software engineering methodologies (e.g. UML), specific programming languages (Java) or middleware (e.g. MPI and CORBA) which are deemed to be too low-level. The book aims to give priority to computation and communication structures that go beyond simple message-passing or remote procedure calling (RPC). Secondly, formal concurrency models (e.g. PetriNets) are not covered because the book focuses mainly on pragmatic approaches leading to practical design and programming methodologies with their associated compilers and tools.

Book Overview

This book covers a variety of approaches as broadly as possible. It is organized as a collection of self-contained chapters, written by leading researchers on parallel and distributed systems design. Unlike typical research papers, each chapter is written in a comprehensive, concise and tutorial-like style. As far as possible, obscure technical details are contained in bibliographical and WWW references, which can be easily accessed for further reading.

The book is divided into two parts. Part I presents skeletons-related material, such as the expression and composition of skeletons, formal transformation, cost modelling and languages, compilers and run-time systems for skeleton-based programming. It covers purely functional, hybrid functional/imperative or higher-level imperative platforms.

Chapter 1 by Fischer, Gorlatch and Bischof outlines basic concepts and theoretical results, providing a foundation for the construction and use of skeletons based on data. While the other chapters describe skeletons in specific language contexts (Haskell, C, etc.), this chapter remains language-independent, presenting the basics of skeletons in a general formal setting. The authors explain how skeletons are used in the process of program development, then introduce data-parallel skeletons as higher-order functions on the most popular data type, lists, and present equations over these functions, thus providing an algebra of lists. It is shown how the equations of the algebra of lists can be used as transformation rules in the process of designing an efficient parallel program from an initial specification of a problem. The authors identify a class of skeletons called catamorphisms, which possess a common efficient parallel implementation scheme. The chapter describes an automatic method for finding a well-parallelisable catamorphic representation of a problem using its sequential formulation. Finally, the authors describe the skeleton framework in a more general setting based on category theory. The use of the skeleton framework is demonstrated on a

practically relevant case study – two-dimensional numerical integration.

Chapter 2 by Gorlatch argues against the low-level communication primitives common in contemporary parallel languages and proposes expressing communication in a structured way using collective operations and skeletons. This is accomplished using the SAT (Stages And Transformations) methodology. The methodology's power is demonstrated by several case studies, for which either new parallel solutions are provided or, more often, in which a systematic way is demonstrated by arriving at optimal solutions that were previously obtained in an *ad hoc* manner. The presentation addresses five challenges for collective operations and skeletons as an alternative to *send-receive*: simplicity, expressiveness, programmability, absolute performance, and performance predictability.

Chapter 3 by Herrmann and Lengauer offers a technique for application programmers to investigate different parallel implementations of an algorithm quickly by constructing prototypes in a functional language, making use of predefined parallel skeleton implementations. The programmer views a skeleton as a higher-order function and is not involved in low-level implementations at all. The choices are in the selection of skeletons, their instantiation with parameters controlling the parallelisation and their customisation with problem-specific functions, possibly, nested skeletons. The approach is even appropriate for inexperienced parallel programmers, because the application program can never produce failures because of parallelisation. The authors demonstrate the simplicity of parallel functional programming using the travelling salesperson problem as an example.

Chapter 4 by Loogen, Ortega, Peña, Priebe and Rubio presents the parallel functional programming language Eden, which extends Haskell by expressions for defining and instantiating processes. Parallel programming is done in Eden at two levels. The abstract level is appropriate for building parallel applications with little effort on top of the predefined skeletons. At the lower level, the programmer instantiates processes explicitly, being able to create new skeletons and also to build applications with irregular parallelism for which no appropriate skeleton is available. The authors present several skeletons covering a wide range of parallel structures, together with their implementations and cost models. Some examples of application programming are shown, including predicted and actual results on a Beowulf cluster.

In Chapter 5, Michaelson and Scaife discuss the work of the Heriot-Watt University group in realising skeleton based parallel implementations from functional prototypes. Experiments in hand crafting parallel occam2 programs from SML prototypes, to solve problems in computer vision, have led to the automated exploitation of nested skeleton parallelism from sites of nested higher order functions (HOFs), using proof planning to synthesise HOFs in programs that lack them.

Chapter 6 by Pelagatti describes P3L – a coordination language in which applications can be expressed by means of combining task- and data-parallel skeletons. The programmer concentrates on application structure, without coding single low-level interactions. The main difference between P3L and other coordination languages is that it is designed to make the performance of programs predictable from the cost of their sequential parts and from the knowledge of the constituent skeletons. Costs can be used by programmers to take sensible decisions during parallel software development and by compilers to optimise the global application structure. This chapter presents the parallel model underlying P3L, discusses parallel software development using cost models, and details P3L implementation. Examples and results for a few real size applications are shown.

Chapter 7 by Rabhi focuses on a "coarse-grained" skeleton, namely the Static Iterative Transformation (SIT) skeleton, which can be thought of as a data parallel operator applied through several iteration steps. It describes several parallel programming environments that allow customised applications to be automatically generated for a variety of machines. Projects vary in the choice of notations for skeleton parameters (e.g. functional languages, visual abstractions) and implementation platforms (e.g. PVM, BSP).

Part II is dedicated to design patterns and other related concepts, applied to other areas such as real-time, embedded and distributed systems.

Chapter 8 by Cross and Schmidt studies the design of real-time and embedded distributed applications and in particular how to effectively address Quality of Service (QoS) requirements. They propose a design pattern, called the *Quality Connector Pattern*, which enables application developers to specify their QoS requirements to the middleware infrastructure. The pattern also manages the middleware operations that implement these QoS requirements in an optimal way. A practical implementation using real-time CORBA is described.

Chapter 9 by Rana and Walker is also related to the design of distributed applications using standardised components and middleware. It uses the "Grid" concept in which distributed processes are either providers or consumers of *services*. This enables the separation of concerns: distributed applications can be rapidly assembled as service invocations without concern about the underlying infrastructure, and this infrastructure can be changed or upgraded without affecting the applications. Design patterns fit in very well with this approach. A designer can use patterns to rapidly construct applications, and each pattern can be coded in a particular programming language or makes use of "grid enablers" such as Globus or Legion (this need be of no concern to the designer).

Chapter 10 by Benatallah, Dumas, Fauvet and Rabhi is very similar in its approach: it separates design concerns (using patterns) and implementation (using the service abstraction). The approach focuses on the area of Business-to-Business systems and provides implementation clues that work not only for traditional middleware platforms (e.g. CORBA and Java) but also for XML-based technologies and inter-enterprise workflows. A suite of design patterns, dealing with service wrapping (i.e. integration of legacy applications), service contracting, service composition, service discovery and service execution, are proposed.

Finally, Chapter 11 by Aboulhamid, Bois and Charest addresses the design (called codesign) of systems that involve mixed software and hardware components. The complexity arises from a large design space caused by a multiplicity of decisions and alternatives: identifying parts of the requirements that should be implemented in hardware, software or a mixture of both; defining a schedule of processes allocated to processors while preserving timing constraints; establishing communication and synchronisation links between components, etc. Again, design patterns are proposed as a solution for reducing the complexity of the design activity. On the one hand, they incorporate good-quality design experience that has proved useful in successful design projects. On the other, they free the designer's mind from low-level implementation considerations, while pointing the way to implementation solutions using languages such as VHDL and SystemC.

Target Audience

The book provides an important collection of texts for an advanced undergraduate course or graduate course on Parallel or Distributed Systems Design. It will hopefully become a useful reference work for researchers undertaking new projects in this area. Readers must have a strong background in computer programming languages and computer systems. Part I of the book requires some background in mathematics and formal methods.

Acknowledgements

We wish to thank all the authors for their hard work and effort in creating this book, and in particular Greg Michaelson for his assistance during the proposal phase. We are especially grateful to Feras Dabous, Yun Ki Lee, Holger Bischof and Marie-Christine Fauvet for their help with the formatting in LaTeX and to Phil Bacon for improving our presentation. We would also like to thank Rosie Kemp, Melanie Jackson and Karen Borthwick of Springer-Verlag London for their comments, suggestions and professional advice during the publishing process.

Fethi Rabhi (f.rabhi@unsw.edu.au)
Sergei Gorlatch (gorlatch@cs.tu-berlin.de)

Contents

List of Contributors

Mostapha El Aboulhamid

DIRO, Université de Montréal,
2920 Ch. de la Tour, CP6128 Centre-Ville, Montréal, Québec, Canada
aboulham@iro.umontreal.ca

Boualem Benatallah

School of Computer Science and Engineering
University of New South Wales, Sydney NSW 2052, Australia
boualem@cse.unsw.edu.au

Holger Bischof

Technische Universität Berlin, Fakultät für Elektrotechnik und Informatik
Sekr. FR 5–6, Franklinstraße 28/29, 10587 Berlin, Germany
bischof@cs.tu-berlin.de

Guy Bois

DGEGI, Ecole Polytechnique de Montréal,
CP6079 Centre-Ville, Montréal, Québec, Canada
guy.bois@polymtl.ca

Luc Charest

DIRO, Université de Montréal,
2920 Ch. de la Tour, CP6128 Centre-Ville, Montréal, Québec, Canada
chareslu@iro.umontreal.ca

Murray Cole

Division of Informatics, University of Edinburgh,
King's Buildings, Mayfield Road, Edinburgh, EH9 3JZ, UK
mic@dcs.ed.ac.uk

Joseph K. Cross

Lockheed Martin Tactical Systems, St. Paul, Minnesota, USA
joseph.k.cross@lmco.com

Marlon Dumas

Centre for Information Technology Innovation
Queensland University of Technology, Brisbane QLD 4001, Australia
m.dumas@qut.edu.au

Marie-Christine Fauvet

LSR-IMAG Laboratory, University of Grenoble
BP 53X, 38420 Grenoble Cedex, France
Marie-Christine.Fauvet@imag.fr

Jörg Fischer

Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Sekr. FR 5–6, Franklinstraße 28/29, 10587 Berlin, Germany
jfischer@cs.tu-berlin.de

Sergei Gorlatch

Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Sekr. FR 5–6, Franklinstraße 28/29, 10587 Berlin, Germany
gorlatch@cs.tu-berlin.de

Christoph A. Herrmann

Universität Passau
Fakultät für Mathematik und Informatik
Innstr. 33, 94032 Passau, Germany
herrmann@fmi.uni-passau.de, www.fmi.uni-passau.de/cl/hdc/

Christian Lengauer

Universität Passau
Fakultät für Mathematik und Informatik
Innstr. 33, 94032 Passau, Germany
lengauer@fmi.uni-passau.de, www.fmi.uni-passau.de/cl/hdc/

Rita Loogen

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße, Lahnberge, D-35032 Marburg, Germany
loogen@mathematik.uni-marburg.de

Greg Michaelson

Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, EH14 4AS
greg@cee.hw.ac.uk

Yolanda Ortega

Universidad Complutense de Madrid, Departamento de Sistemas
Informáticos y Programación, E-28040 Madrid, Spain
yolanda@sip.ucm.es

Susanna Pelagatti

Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
susanna@di.unipi.it

Ricardo Peña

Universidad Complutense de Madrid, Departamento de Sistemas
Informáticos y Programación, E-28040 Madrid, Spain
ricardo@sip.ucm.es

Steffen Priebe

Philipps-Universität Marburg
Fachbereich Mathematik und Informatik
Hans-Meerwein-Straße, Lahnberge, D-35032 Marburg, Germany
priebe@mathematik.uni-marburg.de

Fethi A. Rabhi

School of Information Systems, Technology and Management
The University of New South Wales, Sydney 2052, Australia
f.rabhi@unsw.edu.au

Omer F. Rana

Department of Computer Science
Cardiff University, Cardiff CF24 3XF, UK
o.f.rana@cs.cf.ac.uk

Fernando Rubio

Universidad Complutense de Madrid, Departamento de Sistemas
Informáticos y Programación, E-28040 Madrid, Spain
fernando@sip.ucm.es

Norman Scaife

Japanese Advanced Institute of Science and Technology,
Asahidai 1-1, Tatsunokuchi, Nomigun, Ishikawa, 923-1211 Japan
norman@jaist.ac.jp

Douglas C. Schmidt

Electrical & Computer Engineering, University of California, Irvine, USA
schmidt@uci.edu

David W. Walker

Department of Computer Science
Cardiff University, Cardiff CF24 3XF, UK
david.w.walker@cs.cf.ac.uk