

+ $T + t_c$ otherwise

993) that the minimum of $T_c(N)$

$$= \min \left\{ \frac{T}{\max\{t_c\}}, N \right\}$$

del of the temple

problem

model. The

stream of

menting the

output chan

the process

$T_c(N)$

nd it can b

formation

parts.

uations (

compos

the PD

the ge

be par

es. Let

single m

these

the

the first one producing

and the intermediate sta

Figure 8.11 needs to be

generating the coordinat

rot set and a third stage

the last stage and the pi

) is a predefined librari

l_exit() takes care of th

m.

compiler

in Figur

del struct

and a single output channel

we can compose the

recognizing a

input data stream, the last one dealing

the actual computation

line with three stage

the second

the color nat

called

cedu

rect

a U

m can

parallel

Structured Development of Parallel Programs

Susanna Pelagatti



Taylor & Francis
Publishers since 1798

TP301.6
w 3

Structured Development of Parallel Programs

SUSANNA PELAGAT
University of Pisa, Italy

江苏工业学院图书馆
藏书章

30019037



Taylor & Francis
Publishers since 1798

UK Taylor & Francis Ltd, 1 Gunpowder Square, London EC4A 3DE
USA Taylor & Francis Inc., 1900 Frost Road, Suite 101, Bristol, PA 19007

Copyright © Susanna Pelagatti 1998

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, electrostatic, magnetic tape, mechanical photocopying, recording or otherwise, without the prior permission of the copyright owner.

British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library
ISBN 0-7484-0655-7

Library of Congress Cataloging-in-Publication Data are available

Cover design by Youngs Design in Production

Typeset in Times 10/12pt by Focal Image Ltd, London, UK

Printed by T.J. International Ltd, Padstow, UK



Structured Development of Parallel Programs



0-211-737


Taylor & Francis

Founded 1798

To Maria Teresa and Pier Luigi

Preface

P

1 Introduction

- 1.1 The book's organization
- 1.2 External and internal structure
- 1.3 The book's objectives
- 1.4 Acknowledgements

2 Problems and models in parallel algorithms

- 2.1 A simple parallel algorithm
- 2.2 A more complex algorithm
- 2.3 Speedup and efficiency
- 2.4 The parallel processing model
- 2.5 The parallel processing model
- 2.6 The parallel processing model
- 2.7 The parallel processing model
- 2.8 The parallel processing model
- 2.9 The parallel processing model
- 2.10 The parallel processing model

3 Basic parallel paradigms

- 3.1 The divide-and-conquer paradigm
- 3.2 The greedy paradigm
- 3.3 The dynamic programming paradigm
- 3.4 The branch-and-bound paradigm
- 3.5 The heuristic paradigm

4 Mapping and scheduling in graph theoretical models

- 4.1 The reduction of the graph to a tree
- 4.2 The reduction of the graph to a tree
- 4.3 The reduction of the graph to a tree
- 4.4 The reduction of the graph to a tree

Foreword



Structure and abstraction are the essence of good computer science. Parallel computing has the pursuit of absolute performance as its *raison d'être*. In what are still, from any sensible perspective, the early days of parallelism, there has been an understandable tendency to ignore issues of higher-level principle, particularly in the area of programming model and language design, in the rush for the improved speed-up curve. This book is representative of a research area of growing interest which seeks to impose structural discipline on the parallel programmer's task, offering in return conceptual abstraction and its concomitant simplicity, portability and predictability, while remaining sensitive to the need for good performance.

The key observation is that real parallel programs are rarely random collections of processes interacting in unpredictable ways, but are in fact well structured in concept (if not in concrete presentation) and adhere to a small collection of more or less regular operational patterns. The challenge is then to embed our understanding of these patterns into the design of programming systems, both at the level of language constructs and in the implementation process.

The book is not simply a research monograph, destined for the library shelves or the desks of a few closely related researchers. Care has been taken to place the ideas within the wider field of parallel computing, making it an appropriate vehicle through which to introduce the subject to aspiring postgraduates and even appropriately focused senior undergraduates. The opening chapters are a properly contextualised manifesto for the structured approach and one might go so far as to suggest that this is the perspective from which parallelism ought to be taught in general. Certainly, there is plenty of food for thought here for anyone who is serious about the development of parallel programming as a well-founded discipline.

It is apt that such a book should emanate from Pisa. Susanna and her colleagues are eminent in the field and their efforts in addressing underlying issues, building concrete implementations and introducing the methodology to industrial partners are praiseworthy.

It is a pleasure to be able to offer these few words of introduction to what should become recognised as one of the foundational texts in the area.

MURRAY COLE
Edinburgh

Preface



"Would you tell me, please, which way I ought to go from here?"

"That depends a good deal on where you want to get to", said the Cat.

"I don't much care where—" said Alice.

"Then it doesn't matter which way you go," said the Cat.

"—so long as I get somewhere," Alice added as an explanation.

"Oh, you're sure to do that," said the Cat, "if you only walk long enough." L. CARROLL

Parallelism has always been considered important, but in the past decade interest in it has grown enormously. The reason for this growth is that parallel and distributed architectures have become readily available as commercial products and parallelism promises to allow the solution of challenging frontier problems. However, from the very beginning the main limit on parallelism has been the ability to write parallel programs at a reasonable cost. Coordinating and managing execution of parallel tasks is too difficult to be left as the responsibility of the programmer. If parallelism is to have an impact in the real world, parallel machines need to be programmed using high-level languages and most of the parallel machine complexity must be implicitly dealt with by compilers.

There is currently a wide variety of ways in which parallel software is developed, but there is no widely accepted software development methodology able to free programmers from parallelism complexity and at the same time able to achieve high degrees of performance on different architectures.

The key problem when developing such a methodology is the intrinsic difficulty of the problems to be solved automatically by the language compiler/support. Data decomposition, process mapping, task scheduling and grain optimization have a tremendous impact on the actual performance achieved on a parallel architecture. Moreover, the performance achieved by different solutions to these problems needs to be predicted to take sensible optimization decisions in the compiling process. Unfortunately, for general parallel program structures the decomposition mapping and scheduling problems are intractable and such that the performance of a given solution cannot be estimated accurately.

The purpose of this book is to show how notoriously difficult problems such as mapping/scheduling can be made tractable by restricting the structure of parallel programs and to propose a methodology for the design of parallel software which is high-level, portable and able to achieve high performance figures.

The methodology uses a small number of parallel forms as building blocks for par-

allel applications and allows the coding of parallel programs in a high-level machine-independent notation which can be automatically reorganized by the compiler. The parallel forms are provided as primitive constructs of a structured parallel language and are the only way in which the application parallel structure can be expressed. The class of applications which can be coded in this way is extremely large, and includes many data parallel and task parallel examples, or applications exploiting a mixture of the two. The restriction imposed by the language allows an innovative organization of the compiler to be adopted, which results in a number of benefits. In particular, machine-dependent problems can be solved effectively and performance prediction can be achieved at all decision levels.

The book is organized as follows. It first analyzes the existing systems for parallel software production, present in the literature, and outlines the intrinsic limitations of these different approaches. Then, it details an innovative methodology for the development of parallel programs which provides the programmer with a small set of parallel forms (skeletons) as building blocks for parallel applications. Finally, it describes P3L, a structured parallel programming language based on skeletons, discusses the organization of the P3L compiler, and gives examples of structured parallel program development.

The book is targeted at researchers, both in academic institutions and in progressive commercial companies, and at aspiring research students in the area of parallel and distributed processing.

A number of people have made important contributions to the development of the ideas, methodology and tools discussed in this book. I owe much to Bruno Bacci, Marco Danelutto, Salvatore Orlando and Marco Vanneschi for their extensive work on the P3L project, first at the Hewlett Packard Pisa Science Center and then at the Department of Computer Science of the University of Pisa. In particular, Bruno Bacci and Salvatore Orlando also developed most of the first prototype compiler of P3L. Marco Danelutto proofread part of the book and provided some of the examples in Chapter 10. Special thanks go to Milon Mackey, who developed the initial front-end and participated enthusiastically in the P3L project in his spare time. Fabio Piazzai, Fabrizio Pasqualetti, Francesco Chiaravalloti, Barbara Cantalupo and Nicola Guerrini contributed to the template development for both the Meiko CS1 and the PVM versions of the compiler. A number of undergraduate students participated in the project to build a set of applications using the P3L compiler: Domenica Barresi, Giacomo Giunti, Stefano Milana, Paolo Pesciullesi, Paola Criscione, Gianni De Giorgi, Antonio Bisio, Alessia Conserva, Stefano Bordin, Davide Pasetto and Maria Gabriella Brodi. Roberto Ravazzolo and Alessandro Riaudo implemented the OCR application discussed in Chapter 10. I would like to thank Peter Thanish for the many helpful suggestions on the material and the presentation of Chapter 4. Very special thanks go to Paul Kelly, who is definitely responsible for convincing me to write this book. I am grateful to Murray Cole, whose suggestions greatly contributed to the improvement of the manuscript and to David Skillicorn for his encouragement.

Finally, I am indebted to Bruno, who participated in the book in more ways than I can mention.

SUSANNA PELAGATTI

*University of Pisa
Pisa, Italy*

susanna@di.unipi.it

Contents



| | |
|--|----------------|
| <i>Foreword</i> | <i>page xi</i> |
| <i>Preface</i> | <i>xiii</i> |
| 1 Introduction | 1 |
| 1.1 Effective sequential programming | 1 |
| 1.2 Defining a suitable methodology for parallel programming | 2 |
| 1.3 Overview of the book | 4 |
| 1.4 Notation and terminology | 5 |
| 2 Problems and models in parallel computation | 9 |
| 2.1 Solving a problem in parallel | 9 |
| 2.2 A simple example | 12 |
| 2.3 Spectrum of solutions in the literature | 19 |
| 2.4 Implicitly parallel models | 21 |
| 2.5 Completely abstract models | 23 |
| 2.6 High-level partly abstract models | 24 |
| 2.7 Low-level partly abstract models | 26 |
| 2.8 Machine-dependent models | 28 |
| 2.9 Discussing the different classes: the whole picture | 28 |
| 2.10 Summary | 32 |
| 3 Basic parallel paradigms | 33 |
| 3.1 Parallelizing the computation of a function | 33 |
| 3.2 Stream parallelism | 34 |
| 3.3 Data parallelism | 39 |
| 3.4 Composition of the basic paradigms | 46 |
| 3.5 Summary | 48 |
| 4 Mapping and scheduling in graph-based systems | 49 |
| 4.1 Introduction: the mapping terminology | 49 |
| 4.2 Modeling the mapping problem | 51 |
| 4.3 Solving the mapping problem | 59 |
| 4.4 The survey | 60 |

| | | |
|----------|---|-----|
| 4.5 | Directed Acyclic Graph models | 61 |
| 4.6 | Synchronous Phase models | 64 |
| 4.7 | Heuristic models using cost functions | 66 |
| 4.8 | Heuristic models constraining the feasible mappings | 68 |
| 4.9 | An automatic mapping tool | 70 |
| 4.10 | Summary | 72 |
| 5 | Template-based systems | 73 |
| 5.1 | Graph-based and template-based systems | 73 |
| 5.2 | Basic structure of a template-based system | 74 |
| 5.3 | Cole's Algorithmic Skeletons | 77 |
| 5.4 | Darlington <i>et al.</i> skeleton library | 79 |
| 5.5 | Skeleton composition | 83 |
| 5.6 | The SCL approach | 84 |
| 5.7 | Summary and other recent skeleton proposals | 89 |
| 6 | A structured methodology for parallel programming | 91 |
| 6.1 | Setting the motivations for structured parallel programming | 91 |
| 6.2 | A methodology based on structured parallel programming | 93 |
| 6.3 | Summary | 103 |
| 7 | P3L, a structured parallel programming language | 105 |
| 7.1 | P3L overview | 105 |
| 7.2 | Data types | 106 |
| 7.3 | The sequential construct | 107 |
| 7.4 | The farm construct | 108 |
| 7.5 | The pipe construct | 109 |
| 7.6 | The loop construct | 110 |
| 7.7 | The map construct | 112 |
| 7.8 | The reduce construct | 115 |
| 7.9 | The comp construct | 116 |
| 7.10 | Iterating data-parallel computations | 119 |
| 7.11 | Functional semantics of the parallel constructs | 120 |
| 7.12 | Summary | 121 |
| 8 | The P3L compiler | 123 |
| 8.1 | The abstract machine | 123 |
| 8.2 | Prototypes and ongoing work | 124 |
| 8.3 | Implementation templates | 124 |
| 8.4 | The structure of the compiler | 125 |
| 8.5 | Basic data structures | 126 |
| 8.6 | Libraries | 128 |
| 8.7 | Front-end | 131 |
| 8.8 | Middle-end | 133 |
| 8.9 | Back-end | 139 |
| 8.10 | A simple compilation example | 141 |
| 8.11 | Summary | 145 |

| | |
|--|-----|
| 9 Template development | 147 |
| 9.1 Issues in template design | 147 |
| 9.2 A map template for full topology | 149 |
| 9.3 A map template for mesh topology | 153 |
| 9.4 Summary | 157 |
| 10 Structured parallel programs in P3L | 159 |
| 10.1 Parallel program development in P3L | 159 |
| 10.2 Matrix multiplication | 161 |
| 10.3 A case study: Optical Character Recognition | 167 |
| 10.4 Summary | 172 |
| 11 Conclusions | 175 |
| 11.1 Assessing the methodology | 175 |
| 11.2 Structured parallel programming: background | 176 |
| 11.3 P3L development | 177 |
| 11.4 Current status of research | 178 |
| References | 179 |

List of Tables



| | | |
|-----|--|-----|
| 1.1 | Miscellaneous mathematical notations used within the book. | 6 |
| 1.2 | Symbols used within the book (except Chapter 4). | 7 |
| 2.1 | Parameters of the MP cost model. | 15 |
| 2.2 | Cost of each operation on the dmDLX machine. | 18 |
| 2.3 | Characteristics of the parallel computational model classes. | 29 |
| 4.1 | Summary of the parameters used in the mapping survey. | 52 |
| 9.1 | Parameters used in the map templates. | 150 |
| 9.2 | Performance formulae of the map template for full topology. | 153 |
| 9.3 | Performance formulae of the map template for mesh topology. | 156 |

List of Illustrations

| | | |
|-----|--|----|
| | The dag of a parallel algorithm for computing the inner product of two vectors a and b with eight elements each. | 12 |
| | A process network implementing the algorithm of Figure 2.1 onto the abstract MP architecture (process graphs Γ_1 and Γ_2). Distribution phase (a) and binary tree sum computation (b). | 13 |
| | Completion time $T_2(p, n)$ of Γ_2 for $n = 10000, 40000, 100000$ and different numbers of processes. | 16 |
| | Absolute speedup of $T_2(p, n)$ for different numbers of processes: S_2 is the real speedup and S_{2wd} is the speedup achieved without paying the distribution cost. | 16 |
| | A possible Γ_2 mapping on dmDLX. | 18 |
| 6 | The Γ_3 process graph: scattering the input data (a) and gathering the partial sums (b). | 18 |
| 7 | Exploiting optimized collective operations on dmDLX: Sc_2 is the speedup achieved by Γ_2 using point-to-point communications and Sc_3 is the speedup achieved by Γ_3 employing collective operations. | 19 |
| 3.1 | A process graph Γ_p exploiting farm parallelism in the implementation of M | 35 |
| 3.2 | A process graph Γ_p implementing M according to the pipeline paradigm. | 37 |
| 3.3 | A process graph Γ_{si} implementing M according to the stream-iterative paradigm. Γ_{si} emulates the unbounded pipeline typical of the stream-iterative paradigm with a static chain of processes. | 38 |
| 3.4 | A process graph Γ_b implementing M according to the basic data-parallel paradigm. | 40 |
| 3.5 | An optimized process graph for the Map&Reduce paradigm. | 42 |
| 3.6 | A process graph Γ_c for implementing the composed data-parallel paradigm. The workers communicate using a completely interconnected channel structure. The picture illustrates the communication channels between W_9 and the rest of the workers. Each worker employs a similar set of channels. | 43 |
| 3.7 | A process graph for fine grain matrix multiplication. The algorithm exploits the basic data-parallel paradigm both for the computation of each element of C (inner product) and between all the elements of C | 46 |
| 5.1 | A possible implementation template for the PIPE skeleton. | 75 |
| 6.1 | The construct tree of the example function composition. | 95 |

| | | |
|------|---|-----|
| 6.2 | A flat implementation template for the FARM construct on the MP model. | 96 |
| 6.3 | The temporal behavior of the flat template. | 98 |
| 6.4 | A composed implementation template for the PIPE construct. | 98 |
| 6.5 | The implementation generated for the construct tree in Figure 6.1. | 101 |
| 6.6 | An absolute optimization rule. | 102 |
| 6.7 | A timed optimization rule. | 102 |
| 7.1 | A sequential module computing a function f on a stream of input data and producing a stream of results. | 106 |
| 7.2 | Declaration of a sequential module accepting in input an integer vector and computing the sum of all the vector elements. | 107 |
| 7.3 | A parallel module exploiting farm parallelism in the computation of the sum of the elements of a stream of vectors. | 108 |
| 7.4 | Declaration of a parallel module exploiting pipeline parallelism between two stages. The first stage computes the sum of the elements of an integer vector and the second stage computes the square root of the result. | 109 |
| 7.5 | Declaration of a parallel module exploiting the stream-iterative paradigm in the computation of A^m . This is expressed by a definite loop. | 110 |
| 7.6 | An indefinite loop instance computing a^{2^k} until it exceeds a threshold N . The corresponding module accepts a stream of integer values and exploits stream-iterative parallelism. | 111 |
| 7.7 | Definition of a parallel module exploiting basic data parallelism in matrix-by-matrix multiplication. | 113 |
| 7.8 | Different overlapping multicast: a column slice of a (a); a square slice of b (b) and a group of columns of c (c). | 114 |
| 7.9 | Definition of a data-parallel module computing a single iteration of the game of life. | 115 |
| 7.10 | Reducing a vector and a matrix with binary user-defined operators. f is a binary operator working on integers and f_vect is a binary associative and commutative operator working on vectors. | 116 |
| 7.11 | A parallel module summing all the rows of a matrix in parallel. The parallel evaluation of all the reductions is achieved by nesting map and reduce. | 117 |
| 7.12 | A parallel module computing the inner product according to the Map&Reduce paradigm. | 118 |
| 7.13 | A module computing A^4 with two data-parallel multiplications in cascade. | 118 |
| 7.14 | A module computing A^m where $m = 2^h$ according to the composed data-parallel paradigm. | 119 |
| 7.15 | Definition of a parallel module computing the game of life. | 120 |
| 8.1 | Outline of the P3L compiler. | 125 |
| 8.2 | A <i>pipe-flattening</i> optimization. | 130 |
| 8.3 | Transformation of two pipeline stages to a notable Map&Reduce composition. | 130 |
| 8.4 | Transformation of a simple sequential module by the front-end. | 131 |
| 8.5 | Construct tree Prolog facts for a simple P3L program. | 133 |
| 8.6 | Labeling of a simple construct tree: name and kind of nodes (a) and an eval&map labeling achieving the minimum service time (b). | 135 |
| 8.7 | A reduction process involving the farm construct. (a) Abstract process graph structure defined by the labeling of Figure 8.6b. (b) Abstract process graph structure after the first reduction step. | 137 |

| | | |
|-------|--|-----|
| 8.8 | A reduction process involving the farm construct. (a) Abstract process graph using only 4 nodes (after the second reduction step). Further reductions with a constraint of having at most 3 processes: the farm construct is collapsed to a single sequential node (c) and the corresponding process graph uses 2 nodes (b). | 138 |
| 8.9 | The abstract process graph corresponding to the labeled tree of Figure 8.6. Each node is identified by a unique name (a number from 0 to 5 inside the node) and is labeled with the information needed to generate the correct process template instance. | 139 |
| 8.10 | A C function computing the color of a complex point (z_r, z_i). | 140 |
| 8.11 | A P3L program computing the Mandelbrot set. | 140 |
| 8.12 | Definition of the main pipeline for the Mandelbrot P3L program. | 141 |
| 8.13 | The logical structure of the Mandelbrot program. | 142 |
| 8.14 | The abstract process graph generated for a Meiko CS1 with 40 nodes available. | 143 |
| 8.15 | Options of the current p3l Unix command. | 144 |
| 8.16 | Experimental results. The service time T_s for Mandelbrot set computation with 300×300 points with resolution 300 ($T_{s300(p)}$) and 400×400 points with resolution 500 ($T_{s400(p)}$) (left) and the corresponding speedups (right). | 144 |
| 8.17 | The abstract process graph generated for PVM. | 145 |
| 9.1 | The distribution and collection interactions between the workers of the map template on the abstract machine. | 151 |
| 9.2 | The process graph of the square template (a) and the worker numbering (1:p)(b). | 154 |
| 9.3 | A scattering strategy taking $O(m)$ communication steps. | 155 |
| 10.1 | Developing and tuning a parallel program with P3L. | 160 |
| 10.2 | A P3L program computing matrix multiplication. All the columns of the result matrix C are computed in parallel (MM_1). | 162 |
| 10.3 | Service time (left) and absolute speedup (right) of the matrix multiplication module MM_1 on Meiko CS1. | 162 |
| 10.4 | Service time (left) and absolute speedup (right) of the matrix multiplication module computing all the elements in parallel (MM_2) on Meiko CS1. | 163 |
| 10.5 | Service time (left) and absolute speedup (right) for MM_1 on a Cray T3D using MPI. | 163 |
| 10.6 | A P3L program computing matrix multiplication on a stream of pairs of matrices (sMM). The parallel algorithm exploits farm parallelism between the computation of different matrix products and uses the module mul of MM_1 to exploit map parallelism in a single matrix product. | 164 |
| 10.7 | Predicted and measured service time values for sMM on the Meiko CS1. Each curve fixes the number of workers in the outer farm template and varies the workers in the inner map. | 165 |
| 10.8 | Estimated and measured service times for sMM on Cray T3D and MPI. Each curve fixes the number of workers of the inner map (4,6,8,16) and varies the number of workers in the farm template. | 166 |
| 10.9 | Service time of the intermediate implementations of sMM generated by the LR algorithm for a Cray T3D with 128 nodes. | 166 |
| 10.10 | The parallel structure of the first solution (OCR_1). | 167 |
| 10.11 | P3L coding for the first solution OCR_1. | 168 |

| | | |
|-------|---|-----|
| 10.12 | A solution exploiting map parallelism (a) and its refinement OCR_2 (b) taking pipe imbalance into account. | 169 |
| 10.13 | The OCR_3 solution exploiting geometric data parallelism. | 169 |
| 10.14 | Parallel implementation of the segmentation stage emulating the Divide&Conquer paradigm with a nesting of map and loop. | 170 |

Introduction

The main obstacle to the widespread diffusion of parallel computing is its complexity and the cost of the associated software development process.

What is really needed is a general purpose methodology for the development of parallel programs and their support able to ensure:

- **programmability**, it should be possible to write and modify parallel programs easily, and to prove their correctness against specifications;
- **portability**, the programs written should be portable across a broad range of architectures,
- **performance**, the support should be able to translate a program for different target architectures achieving good performance figures and optimized resource utilization.

In this book, we first analyze the main problems to be solved to achieve these goals, and then propose a methodology that seems able to overcome most of the problems outlined.

1.1 EFFECTIVE SEQUENTIAL PROGRAMMING

A brief analysis of the way in which programmability, portability and performance have been achieved in imperative sequential programming may be useful to illustrate the main issues to be addressed.

In the sequential world, the above-mentioned goals have been achieved by providing

- a high-level language in which a programmer can easily express sequential algorithms, without dealing with machine-dependent features,
- a performance calculus, which can be used to predict the performance of a sequential algorithm (program),
- efficient compiling tools, able to translate and optimize the source code automatically (and effectively) against the target architecture.

Given a problem P to be solved, the programmer first analyzes P in order to find a “good” sequential algorithm \mathcal{A}_P to solve it. The choice of \mathcal{A}_P is guided by a performance calculus allowing the programmer to optimize the algorithm structure *without specifically taking into account the target machine features*. Then, the algorithm is coded in a suitable language and is optimized and translated by the compiler according to the underlying architecture (e.g. superscalar, vector, VLIW). If the underlying architecture changes, it is sufficient to re-compile the program for the new target and all the advantages are maintained.