NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF INFORMATICS AND COMPUTER TECHNOLOGY
COMPUTING SYSTEMS LABORATORY

Algorithms Design for the Parallelization of Nested Loops

DOCTORAL DISSERTATION

Florina-Monica Ciorba

Athens, February 2008

NATIONAL TECHNICAL UNIVERSITY OF ATHENS
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING
DEPARTMENT OF INFORMATICS AND COMPUTER TECHNOLOGY
COMPUTING SYSTEMS LABORATORY

Algorithms Design for the Parallelization of Nested Loops

DOCTORAL DISSERTATION

of

Florina-Monica Ciorba

Computer Engineer, University of Oradea, Romania (2001)

3 Member Advisory Committee: G. Papakonstantinou
P. Tsanakas
N. Koziris

Was approved by the 7 Member Examining Committee on February 29th, 2008.

....	....	....
G. Papakonstantinou	P. Tsanakas	N. Koziris
Professor NTUA	Professor NTUA	Assistant Professor NTUA
....	....	....
A. G. Stafylopatis	P. Constantinou	T. Varvarigou
Professor NTUA	Professor NTUA	Associate Professor NTUA

....
M. Bekakos
Professor Demokritus University of Thrace

Athens, February 2008

....

Florina M. Ciorba
School of Electrical and Computer Engineering, National Technical University of Athens.

Copyright © Florina M. Ciorba, 2008
All rights reserved

No part of this thesis may be reproduced, stored in retrieval systems, or transmitted in any form or by any means - electronic, mechanical, photocopying, or otherwise - for profit or commercial advantage. It may be reprinted, stored or distributed for a non-profit, educational or research purpose, given that its source of origin and this notice are retained. Any questions concerning the use of this thesis for profit or commercial advantage should be addressed to the author.
The opinions and conclusions stated in this thesis are expressing the author. They should not be considered as a pronouncement of the National Technical University of Athens.

EJNIKO METSOBIO POLUTEQNEIO

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΕΡΓΑΣΤΗΡΙΟ ΥΠΟΛΟΓΙΣΤΙΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

Σχεδίαση Αλγορίθμων για την Παραλληλοποίηση Φωλιασμένων Βρόχων

ΔΙΔΑΚΤΟΡΙΚΗ ΔΙΑΤΡΙΒΗ

της

Florina-Monica Ciorba

Διπλωματούχος Μηχανικός Υπολογιστών, Πανεπιστήμιο της Οράντεα, Ρουμανία (2001)

Η τριμελής συμβουλευτική Επιτροπή: Γ. Παπακωνσταντίνου
Π. Τσανάκας
Ν. Κοζύρης

Εγκρίθηκε από την επταμελή εξεταστική επιτροπή στις 29 Φεβρουαρίου 2008.

....	....	....
Γ. Παπακωνσταντίνου Καθηγητής Ε.Μ.Π.	Π. Τσανάκας Καθηγητής Ε.Μ.Π.	Ν. Κοζύρης Επίκουρος Καθηγητής Ε.Μ.Π.
....	....	....
Α.-Γ. Σταφυλοπάτης Καθηγητής Ε.Μ.Π.	Φ. Κωνσταντίνου Καθηγητής Ε.Μ.Π.	Θ. Βαρβαρίγου Αναπληρώτρια Καθηγήτρια Ε.Μ.Π.
	....	
	Μ. Μπεκάκος Καθηγητής Δημοκριτείου Πανεπιστημίου Θράκης	

Αθήνα, Φεβρουάριος 2008

Părinţilor mei Andrei şi Viorica,
surorii mele Adriana
şi bunicii mele Iuliana

To my beloved parents Andrei and Viorica,
my sister Adriana
and my grandmother Iuliana

Περίληψη

Η ανάγκη για παράλληλη επεξεργασία προκύπτει από την ύπαρξη χρονοβόρων εφαρμογών σε διάφορους τομείς, όπως η πρόβλεψη καιρού, οι εξομοιώσεις πυρηνικής ένωσης, η ανάλυση του DNA και των πρωτεϊνών, η υπολογιστική προσέγγιση της δύναμης των ρευστών, κ.α. Η παράλληλη επεξεργασία συμπεριλαμβάνει αλγόριθμους, αρχιτεκτονική υπολογιστών, παράλληλο προγραμματισμό και ανάλυση της αποδοτικότητας. Κατά τη βελτιστοποίηση της απόδοσης των ακολουθιακών επιστημονικών και τεχνολογικών προγραμμάτων, το μέγιστο κέρδος προέρχεται από την παραλληλοποίηση των φωλιασμένων βρόχων ή των επαναληπτικών διαδικασιών, όπου μεγάλα κομμάτια υπολογισμού εκτελούνται επανειλημμένα. Οι φωλιασμένοι βρόχοι χωρίς εξαρτήσεις ονομάζονται DOALL, ενώ αυτοί με εξαρτήσεις ονομάζονται DOACROSS. Η παραλληλοποίηση των DOACROSS βρόχων είναι πολύ πιο δύσκολη από την περίπτωση των DOALL βρόχων, διότι πρέπει να ικανοποιηθούν οι υπάρχουσες ανάμεσα στις επαναλήψεις εξαρτήσεις. Οι προκλήσεις που πρέπει να αντιμετωπιστούν για την παραλληλοποίηση χρονοβόρων προβλημάτων είναι: η ελαχιστοποίηση του συνολοκού χρόνου επεξεργασίας, η ελαχιστοποίηση του χρόνου επικοινωνίας μεταξύ επεξεργαστών (ιδιαίτερα στην περίπτωση των DOACROSS βρόχων), η εξισορρόπηση του υπολογιστικού φόρτου ανάμεσα στους επεξεργαστές, η αντιμετώπιση και η ανάκτηση από σφάλματα που μπορούν να προκύπτουν στο πρόγραμμα ή στο σύστημα, η διατήρηση των χρονικών περιορισμών, ή ένας συνδυασμός των παραπάνω. Η παρούσα διδακτορική διατριβή εστιάζεται στη παραλληλοποίηση εφαρμογών που περιέχουν DOACROSS βρόχους, αντιμετωπίζοντας μερικές από τις παραπάνω προκλήσεις. Συγκεκριμένα, προτάθηκαν και παρουσιάζονται τέσσερεις στατικές μεθόδοι και τρεις δυναμικές μεθόδοι δρομολόγησης, για διάφορες αρχιτεκτονικές υπολογιστών. Οι στατικές μεθόδοι σχεδιάστηκαν για ομογενή συστήματα, ενώ για ετερογενή συστήματα ή συστήματα με γρήγορο μεταβαλλόμενο φορτίο, σχεδιάστηκαν οι δυναμικές μεθόδοι. Μια από αυτές τις δυναμικές προσεγγίσεις ήταν βιβλιογραφικά η πρώτη προσπάθεια προς την παραλληλοποίηση των DOACROSS φωλιασμένων βρόχων, χρησιμοποιώντας μια χονδρόκοκκη (coarse grain) προσέγγιση και δυναμική δρομολόγηση, σε ετερογενή συστήματα υπολογιστών. Οι προτεινόμενοι αλγόριθμοι υλοποιήθηκαν, επαληθεύτηκαν και αξιολογήθηκαν με εξαντλητικά πειράματα σε διάφορες αρχιτεκτονικές υπολογιστικών συστημάτων.

Abstract

The need for parallel processing arises from the existence of time consuming applications in different areas, such as weather forecasting, nuclear fusion simulations, DNA and protein analysis, computational fluid dynamics, etc. Parallel processing comprises algorithms, computer architecture, parallel programming and performance analysis. In optimizing the performance of scientific and engineering sequential programs, the most gain comes from optimizing nested loops or recursive procedures, where major chunks of computation are performed repeatedly. Nested loops without dependencies are called DOALL, while those with dependencies are called DOACROSS loops. Parallelizing DOACROSS loops is much more challenging than parallelizing DOALL loops, because the existing dependencies between iterations of the loop nest much be satisfied. The challenges that must be addressed for the parallelization of time consuming applications are: minimizing the total execution time, minimizing the communication time between the processors (especially in the case of DOACROSS loops), load balancing the computational load among the processors, dealing with and recovering from failures that may occur either in the program or the system, meeting deadlines, or a combination of these. This doctoral dissertation focuses on parallelizing applications that contain nested DOACROSS loops, while trying to address some of the aforementioned challenges. In particular, it proposes and presents four static methods and three dynamic methods for scheduling nested DOACROSS loops on various architectures. The static scheduling methods were devised for homogeneous systems, while the dynamic scheduling methods were devised for heterogeneous systems or systems with rapidly varying loads. One of the dynamic approaches was bibliographically the first attempt towards the parallelization of nested DOACROSS loops using a coarse grain approach and dynamic scheduling, on heterogeneous systems. The proposed algorithms were implemented, verified and evaluated through extensive experiments on various computer systems architectures.

Contents

List of figures xiii

List of tables xv

Acknowledgements xvii

1 Introduction **1**

1.1 What is Parallel Processing? 1

1.1.1 Motivation 3

1.2 Overview of the Bibliography 4

1.2.1 What is missing? 6

1.3 Contribution of this Dissertation 7

1.4 Dissertation Overview 8

1.5 Publications 8

2 Preliminary Concepts - Background **11**

2.1 The Problem of Task Scheduling 11

2.1.1 Target Systems 13

2.2 Algorithmic Model - Nested DO Loops 15

2.2.1 Dependencies and DOACROSS Loops 15

2.2.2 Representation of DOACROSS Loops 16

3 Static scheduling algorithms **19**

3.1 What is Static Scheduling? 19

3.2 Searching for The Minimum Number of Processors 22

3.2.1 Computing the ECT and LCT Subsets 22

3.2.2 Overview of the Binary Search Decision Algorithm 23

3.2.3 Complexity of the BSDA Algorithm 27

3.3 Assigning Tasks on The Minimum Number of Processors 28

3.3.1 Computing the Lower and Upper Bounds of Processors 28

3.4 Static Scheduling by Exploiting the Geometric Properties of the Iteration Space 32

3.4.1 Adaptive Cyclic Scheduling - ACS 32

3.4.2 Chain Pattern Scheduling - CPS 37

4 Dynamic scheduling algorithms **43**
4.1 What is Dynamic Scheduling? . . . 43
4.2 Automatic Parallel Code Generation . . . 46
4.2.1 For *Scaled* GRIDs . . . 46
4.2.2 For General Loops using Cronus . . . 51
4.3 Self-Scheduling DOACROSS Loops . . . 57
4.3.1 Overview of Self-Scheduling Methods for DOALL Loops (CSS, TSS, DTSS) . . . 58
4.3.2 Motivation for Dynamic Scheduling of DOACROSS Loops . . . 60
4.3.3 The Inter-Slave Communication Scheme . . . 60
4.3.4 Dynamic Multi-Phase Scheduling . . . 61
4.4 Enhancing Self-Scheduling Algorithms via Synchronization and Weighting . . 65
4.4.1 Overview of Self-Scheduling Methods for DOALL Loops (GSS, FSS) . 66
4.4.2 The Synchronization Mechanism – $\mathcal{S}$ . . . 66
4.4.3 Empirical Determination of M . . . 69
4.4.4 The Weighting Mechanism – $\mathcal{W}$ . . . 72
4.4.5 The Combined Mechanisms – $\mathcal{SW}$ . . . 75
4.5 Finding the Optimal Inter-Processor Synchronization Frequency . . . 78
4.5.1 Communication Cost Model . . . 78
4.5.2 Computation Cost Model . . . 78
4.5.3 Optimal Inter-Processor Synchronization Frequency for CSS and $\mathcal{W}$-CSS 78

5 Implementation and Experimental Validation **85**
5.1 Test Cases . . . 85
5.1.1 Mandelbrot Set . . . 85
5.1.2 Full Search Block Matching Motion Estimation . . . 86
5.1.3 Heat Conduction . . . 87
5.1.4 Hydrodynamics . . . 87
5.1.5 Floyd-Steinberg Dithering . . . 88
5.1.6 Transitive Closure . . . 88
5.1.7 Artificial Code . . . 89
5.2 Static Methods . . . 89
5.2.1 Evaluation of BSDA . . . 89
5.2.2 Evaluation of DA . . . 90
5.2.3 Evaluation of CPS . . . 92
5.3 Dynamic Methods . . . 93
5.3.1 Evaluation of the Automatic Code Generation Tool for Scaled Grids . 94
5.3.2 Evaluation of Cronus for General Loops . . . 96
5.3.3 Evaluation of the DMPS($\mathcal{A}$) Scheme . . . 103
5.3.4 Evaluation of $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{SW}$ . . . 107
5.3.5 Verifying the Theoretical Model for Predicting the Optimal h for CSS and $\mathcal{W}$-CSS . . . 119

6 Conclusions **127**
6.1 Future Research Directions . . . 127

A Solving the general equation for 2D hyperplanes, computing minimum and successor points 131

A.1 Solving the general equation $a_1x_1 + a_2x_2 = k$ 131

A.2 Computing minimum points in n-dimensional hyperplanes 132

A.3 Computing successor points in n-dimensional hyperplanes 134

Bibliography 135

List of Figures

2.1 Algorithmic model 15
2.2 Task graph representation of an application with seven tasks and eight dependencies 16
2.3 Cartesian representation of a 2D DOACROSS loop with five uniform dependencies 17

3.1 Representation of ECT, LCT and CR sets for Example 3.2.1 23
3.2 Decision Algorithm 26
3.3 ECT sets partitioned into D_i^k sets for Example 3.3.1 30
3.4 Geometrical representation of the index space, the regions, the cones, the cone vectors, the communication vector, some chains and the optimal hyperplane of Example 3.4.1 33
3.5 ACS algorithm 35
3.6 ACS chain assignment on a homogeneous system with 5 slaves 36
3.7 Patterns, pattern vectors and chains for a general UDL with 4 dependence vectors 38
3.8 Scenario (1) Unbounded P – high communication ⇒ moderate performance: every chain is mapped to a different processor (unrealistic assumption with respect to the number of available processors) 39
3.9 Scenario (2) Fixed P – Moderate Communication (cyclic mapping): chains are mapped to the available processors in a cyclic fashion starting with chain C_0 41
3.10 Scenario (2) Fixed P – Moderate Communication (mapping with offset 3 along the x axis, and offset 2 along the y axis): chains are mapped to the available processors so as to minimize communication imposed by $\mathbf{d}_c$, $\mathbf{d}_1$ and $\mathbf{d}_3$ 41

4.1 Geometrical representation of the (a) iteration index space J and (b) auxiliary index space J_{aux} of Example 4.2.1 48
4.2 Optimal hyperplane for two different index spaces 53
4.3 Minimum and maximum points on hyperplanes 54
4.4 Organization of Cronus 56
4.5 Self-Scheduling algorithms for DOALL loops 58
4.6 Synchronization points 61
4.7 Chunks are formed along u_c and SP are introduced along u_s 62
4.8 State diagram of the slaves 62

4.9 $DMPS(\mathcal{A})$ Scheme . . . 63
4.10 $DMPS(\mathcal{A})$ Scheme (*continued*) . . . 64
4.11 Partitioning of a 2D DOACROSS loop into chunks, and placement of synchronization points . . . 67
4.12 Master-slave model with synchronization mechanism . . . 68
4.13 Pseudocode of the communication scheme implementation on the slave side . 70
4.14 Parallel execution in a pipelined fashion . . . 71
4.15 Master-slave model with weighting mechanism . . . 73
4.16 Master-slave model with the combination of synchronization and weighting mechanisms . . . 76
4.17 Partitioning a 2D loop into chunks and placement of synchronization points ($P = N$) . . . 79
4.18 Parallel execution on a homogeneous system with $P = N$ . . . 80
4.19 Parallel execution on a homogeneous system with $P < N$, k=3 . . . 82

5.1 Test results of scheduling 2D, 3D, 4D, 5D UDLs with $OP \in [LB, UB]$ . . . 90
5.2 Test results of scheduling 2D UDLs: left - without the LB_3, right - with the LB_3 . . . 91
5.3 Test results of scheduling 3D UDLs: left - without the LB_3, right - with the LB_3 . . . 91
5.4 Test results of scheduling 4D UDLs: left - without the LB_3, right - with the LB_3 . . . 91
5.5 Test results of scheduling 5D UDLs: left - without the LB_3, right - with the LB_3 . . . 92
5.6 Experimental results, CPS vs cyclic, on $NP = 5$ . . . 92
5.7 Experimental results, CPS vs cyclic, on $NP = 6$ . . . 93
5.8 Experimental results, CPS vs cyclic, on $NP = 7$ . . . 93
5.9 Experimental results, CPS vs cyclic, on $NP = 8$ . . . 93
5.10 Speedup results for 7 testing examples . . . 95
5.11 Speedup comparison for Floyd-Steinberg . . . 98
5.12 Communication vs. computation percentages for Floyd-Steinberg . . . 98
5.13 Speedup comparison for Transitive Closure . . . 98
5.14 Communication vs. computation percentages for Transitive Closure . . . 99
5.15 Speedup comparison for artificial code snippet . . . 99
5.16 Communication vs. computation percentages for artificial code snippet . . . 99
5.17 Speedup comparison for FSBM . . . 100
5.18 Communication vs. computation percentages for FSBM . . . 100
5.19 Real communication vs. scheduling overhead percentage for Floyd-Steinberg . 100
5.20 Real communication vs. scheduling overhead percentage for Transitive Closure 101
5.21 Real communication vs. scheduling overhead percentage for artificial code snippet . . . 101
5.22 Performance comparison of parallel codes generated by CRONUS and UPC . . 102
5.23 Dependence patterns for Heat Conduction and Floyd-Steinberg. . . . 105
5.24 Speedups for the heat equation on a dedicated & non-dedicated heterogeneous cluster . . . 107
5.25 Speedups for Floyd-Steinberg on a dedicated & non-dedicated heterogeneous cluster . . . 109

5.26 Impact of M on the parallel execution times for Floyd-Steinberg and Hydro case studies . . . 111
5.27 Parallel times of the synchronized–only algorithms for Floyd-Steinberg and Hydro case studies . . . 112
5.28 Parallel times of the weighted and non-weighted algorithms for Mandelbrot case study . . . 114
5.29 Parallel times of the synchronized–weighted and synchronized–only algorithms for Floyd-Steinberg case study, for three different problem sizes: 15000×5000, 15000×10000 and 15000×15000 . . . 116
5.30 Parallel times of the synchronized–weighted and synchronized–only algorithms for Hydro case study, for three different problem sizes $10000 \times 5 \times 5000$, $10000 \times 5 \times 7500$ and $10000 \times 5 \times 10000$ . . . 117
5.31 Theoretical vs. actual parallel times, homogeneous system, 10 workers, k=1, 4 and 8 . . . 122
5.32 Theoretical vs. actual parallel times, homogeneous system, 6 workers, k=1, 4 and 8. . . . 124
5.33 Theoretical vs. actual parallel times, heterogeneous system, 10 workers, k=1, 4 and 8 . . . 125

A.1 Pseudocode for the dynamic programming algorithm that computes the minimum point . . . 133
A.2 Pseudocode for finding the successor of index point `pointJ` that belongs to hyperplane `hPlane` and stores it in `successorJ` . . . 134

List of Tables

3.1 Optimal schedule with $N = LB = 20$ processors for the UDL of Example 3.2.1 27

4.1 Sample chunk sizes given for $|J| = 5000 \times 10000$ and $m = 10$. Chunks are formed along $u_c = 5000$ 60
4.2 Chunk sizes given by the original and weighted algorithms for the Mandelbrot set, index space size $|J| = 10000 \times 10000$ points and $m = 4$ slaves 74
4.3 Chunk sizes given by the synchronized–only and synchronized–weighted algorithms for the Floyd-Steinberg loop, index space size $|J| = 10000 \times 10000$ points and $m = 4$ slaves 77

5.1 Execution evaluation for 2D, 3D, 4D, 5D UDLs and the percentage of binary searching for OP between LB and UB 90
5.2 SGRID examples used to test the automatic code generation tool 94
5.3 Parallel execution times (sec) for Heat Conduction equation on a dedicated & non-dedicated heterogeneous cluster 106
5.4 Parallel execution times (sec) for Floyd-Steinberg on a dedicated & non-dedicated heterogeneous cluster 108
5.5 Problem sizes for Floyd-Steinberg and Hydro test cases 110
5.6 Speedups for Floyd-Steinberg and Hydro test cases 113
5.7 Gain of the weighted over the non-weighted algorithms for the Mandelbrot test case 114
5.8 Load balancing in terms of total number of iterations per slave and computation times per slave, GSS vs $\mathcal{W}$-GSS (Mandelbrot test case) 115
5.9 Gain of the synchronized–weighted over the synchronized–only algorithms for the Floyd-Steinberg and Hydro test cases 118
5.10 Load balancing in terms of total number of iterations per slave and computation times per slave, $\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS 119
5.11 Estimated computation and communication parameters 120
5.12 Theoretical and actual optimal synchronization intervals, actual time for both these intervals, their difference, and deviation from the minimum actual parallel time 123

Acknowledgements

This doctoral dissertation was supported by the Greek State Scholarships Foundation (Ίδρυμα Κρατικών Υποτροφιών, www.iky.gr) from November 2004 till February 2008. I am grateful to the Greek State Scholarships Foundation for this support.

There is a great number of people who supported and contributed to making this dissertation possible that I would like to express my thanks and appreciation to. First and foremost, my biggest thanks go to my advisor, Professor George Papakonstantinou. Without his support, patience, guidance and confidence in me, I would not be here today. He had faith in me when I lost faith in myself, he encouraged me when I was down, he guided me when I was lost, he was like a father to me during all these years away from my family. He is an inspiration to me on many levels: research, teaching, and life. For all these, I shall forever be indebted to him.

I would also like to sincerely thank the members of my advisory committee, Professors Panagiotis Tsanakas and Nektarios Koziris for their assistance and support to completing this dissertation. Also, my sincerest thanks go to Professors Andreas G. Stafylopatis, Philip Constantinou, Theodora Varvarigou and Manolis P. Bekakos for serving on my dissertation examination committee.

My deep appreciation and gratitude go to Professor Anthony T. Chronopoulos for his advice and collaboration the last three years of my studies.

Special thanks go to my mentor and friend, Theodore Andronikos, who initiated me to the mysteries and wonders of parallel processing, always believed in me, was always patient and had a good word for me. My warmest gratitude goes to my dear friends and colleagues Ioannis Riakiotakis, Ioannis Drositis, Dimitrios Kamenopoulos, Marios Kalathas, and Panagiotis Theodoropoulos for their invaluable help in implementing the methods described in this dissertation. I especially enjoyed the many constructive and challenging discussions with Ioannis Riakiotakis, whom I've been working closely with for the past three years.

Acknowledgements also go to all the people in the Computing Systems Laboratory that I have encountered throughout my time here: Eliza, Aneta, Ioannis P., Dimitrios K., Aris S., Georgios, Maria, Valia, Dora, Dimitrios Voudouris, Christos, Alexandros, Marinos, Vangelis, Nasia, Kornilios, Dimitrios Vergados, and outside of the laboratory: Ileana. First of all for your friendship and second for putting up with me these past few years, both professionally and on a personal level. Thanks for the good times. I have really enjoyed working with you all!

Most of all I am grateful to my parents, sister and grandmother, who supported and continue to support me in all my endeavors and always encouraged me to spread my wings and fly, even at the expense of my absence from their side.

Last, but not least, to *you*, who inspired and motivated me more than you know.

CHAPTER 1

Introduction

This chapter gives an introduction to parallel processing, the motivation for this work, it indicates what is missing from the field and how this dissertation makes a contribution.

1.1 What is Parallel Processing?

Parallel processing (or parallel computing) is a form of computing in which many instructions are carried out simultaneously [AG89]. The underlying principle of parallel processing is that large (or heavy) problems can almost always be divided into smaller (or lighter) ones, which may be carried out concurrently ("in parallel"). Parallel computing exists in several different forms: bit-level parallelism, instruction level parallelism, data parallelism, and task parallelism. Parallel computing has been applied mainly in high performance computing systems. High performance computing (HPC) usually refers to the use of a group of interconnected computers (supercomputers and computer clusters) in order to perform computations in parallel.

The need for parallel processing (hence, high performance computing) arises from the existence of time consuming applications in different areas, such as:

- computational fluid dynamics – all sorts of fluids
- molecular dynamics and astrodynamics – e.g. nuclear fusion simulations
- environmental modeling – atmosphere, land use, acid rain
- integrated complex simulations – e.g. weather forecasting, climate changes
- health and biological modeling – empirical models, DNA and protein analysis
- structural dynamics – civil and automotive

Parallel processing comprises algorithms, computer architecture, programming and performance analysis. The process of parallelization consists of analyzing sequential programs for parallelism and restructuring them to run efficiently on parallel systems. When optimizing the performance of scientific and engineering sequential programs, the most gain comes from optimizing nested loops or recursive procedures, where major chunks of computation are performed repeatedly. Nested loops without dependencies are called DOALL, while those with

dependencies are called DOACROSS loops. Parallelizing DOACROSS loops is much more challenging than parallelizing DOALL loops, because the existing dependencies between iterations of the loop nest much be satisfied. Generally, the challenges that must be addressed for the parallelization of time consuming applications are: minimizing the total execution time, minimizing the communication time between the processors (especially in the case of DOACROSS loops), load balancing the computational load among the processors, dealing with and recovering from failures that may occur either in the program or the system, meeting deadlines, or a combination of these.

The process of parallelization consists of three steps [Sin07]: (1) decomposing the application into tasks, (2) analyzing the dependencies between the decomposed tasks and (3) scheduling these tasks onto the target parallel or distributed system. A task can range from a simple statement to basic blocks, loops and sequences of these. In this work, we will refer to tasks as *one iteration of a nested DO loop*, unless otherwise stated.

(1) Decomposing an application into tasks, is called *task decomposition*, and is determined by the following factors:

Concurrency - The degree of concurrency of an application gives the number of tasks that can be executed concurrently. Applications in which all tasks can be executed concurrently are called *embarrassingly parallel*, whereas applications in which no two tasks can be executed concurrently are called *embarrassingly serial*. The embarrassingly parallel and the embarrassingly serial applications are the two extremes of the time consuming applications' spectrum. Obviously, there are application in which certain groups of tasks can be executed concurrently and they are the focus of this work.

Granularity - Expresses the size of tasks after the decomposition and therefore refers to their computational size. There are three types of granularity: *fine*, *medium* and *coarse*. Fine grain granularity expresses tasks that usually consist of a single operation or computational unit (e.g. an instance of a DO loop nest), whereas coarse grain granularity expresses tasks that consist of multiple operations or computational unit (e.g. may consist of general program statements and/or including repetitive statements).

Application type - Applications may consists of distinct steps, in which each step is viewed as a task and this may lead to a coarse grain decomposition. Alternatively, applications may consist of one iterative block of (regular or not) computations, for which a fine grain decomposition based on data decomposition is more suitable.

Target system - If the target system is a shared-memory architecture, a fine grain decomposition is usually performed, since the cost of communicating information between tasks is very small, whereas for a distributed-memory architecture a coarse gain decomposition may be applied to reduce such communication costs, at the expense of parallelism.

(2) Once the application is decomposed into tasks, the dependencies between tasks must be determined (if they exist). This is called *dependence analysis*. Generally, there are two types of dependencies: *data dependencies* and *control dependencies*. Data dependencies are caused by data transfer between tasks. There are three types of data dependencies:

- flow (true) dependencies - one task writes a variable and another tasks reads that variable, thus creating a precedence order for the execution of these two tasks.

- anti-dependencies - one task reads a variable and then another tasks writes that variable.
- output dependencies - both tasks write on that variable.

Flow dependencies are real dependencies between tasks and cannot be eliminated. Anti-dependencies and output dependencies can be eliminated by modifying programs (e.g. by using a new variable) [CDRV98], [Sin07].

Unlike data dependencies, control dependencies are not created by data transfer but describe the control structure of the program. Control dependencies can be transformed into data dependencies [BENP93] and then the techniques for analyzing data dependencies can be applied. In this work we deal with flow dependencies between tasks (i.e. iterations of nested DOACROSS loops), which are also called inter-iteration dependencies. We consider that all operations of a task are executed in sequential order, therefore we do not address intra-iteration dependencies.

(3) After the application is partitioned into tasks and the dependencies between tasks are determined, the last step of the parallelization process is scheduling these tasks in time and spatially assigning them to the processors of the target system. The time schedule (temporal assignment) refers to assigning a start time to each task. The spatial assignment (mapping) is the allocation of tasks to processors. The time schedule along with the spatial assignment are generally called *scheduling*.

Local scheduling is performed by the operating system of a processor and consists of the assignment of the processes to the time-slices of the processor. Global scheduling is the process of deciding where to execute a process in a multiprocessor system. It may be carried out by a single authority or it may be distributed among processing elements. In this work we focus on global scheduling, and we will refer to it simply as scheduling.

A scheduling algorithm must take into account all characteristics of the particular architecture. A key issue that must be carefully considered is minimizing the communication cost, in order to increase the performance of the parallel program. Communication may be required for every iteration of the loop (fine grain parallelism approach) or for groups of iterations (coarse grain parallelism approach).

1.1.1 Motivation

Why parallelize nested loops in general? Because they constitute the most computational intensive part of a time consuming problem, hence the most gain in performance comes from parallelizing nested loops. Parallelizing applications and scheduling tasks is a very complex problem and crucially determines the efficiency of the target system.

How does task scheduling relate to loop scheduling? Loop scheduling is a particular case of the more general case of task scheduling, in which each loop iteration is considered to be a task. Hence, DOACROSS loop scheduling refers to the problem of scheduling dependent tasks.

How easy is it to schedule tasks? Task scheduling is an NP-complete problem, meaning that an optimal general solution cannot be found in polynomial time. This has been the motivation behind the development of many heuristics for its near optimal solution. Heuristic methods rely on the rules of the thumb to guide the scheduling processes in the proper track for a near optimal solution. Static scheduling algorithms are in general easier to design and program. In static scheduling, the communication and computation structure of the program and the target parallel system must be completely known at compile

time. Dynamic scheduling algorithms may have a more complicated design and programming. However they are more flexible allowing for adaptation to the unforeseen application requirements at run-time or state of the target parallel system. Early heuristics focused on producing the optimal schedules did not consider the costs of communication. Alternatively, the heuristics which considered both the computation and communication costs, usually assumed unit computations and communication costs or an unlimited number of processors. Good scheduling heuristic (static and/or dynamic) should be based on realistic assumption, meaning to consider arbitrary computation and communication costs as well as a limited number of processors.

The motivation of this work is the need for such general scheduling algorithms, based on realistic assumptions, that focus on meeting the different challenges of parallelization (minimizing the total execution time, minimizing the communication time between the processors and load balancing the computational load among the processors).

1.2 Overview of the Bibliography

Traditionally, software has been written for serial computation. The serial program was to be run on a single computer having a single CPU, the problem was broken into a sequence of instructions, that were executed one after another, and only one instruction could be executed at any moment in time. In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem. The parallel program is to be run using multiple CPUs. The problem is broken into discrete parts that can be solved concurrently. Each part is further broken down to a series of instructions, and instructions from each part execute simultaneously on different CPUs.

The compute resources can include a single computer with multiple processors, an arbitrary number of computers connected by a network, or a combination of both. The computational problem usually demonstrates characteristics such as the ability to: be broken apart into discrete pieces of work that can be solved simultaneously; execute multiple program instructions at any moment in time; solved in less time with multiple compute resources than with a single compute resource. When the discrete pieces of work are independent, then no communication is required for the parallel execution.

A dependence exists between program statements when the order of statement execution affects the results of the program. A data dependence results from multiple use of the same storage location(s) by different tasks. Dependencies are of crucial importance to parallel programming because they are one of the primary inhibitors to parallelism. Dependencies must therefore be thoroughly analyzed in order to decide upon the best way to parallelize the application [Ban88] [Pug92] [Mol93].

Usually the most computationally intensive part of a program can be attributed to the nested loops it contains, since they iterate many times over the same statements. One of the most addressed challenges of efficient parallelization of nested loops programs is to minimize the makespan, i.e., the parallel execution time. The major issues of minimizing the makespan of a parallel program concern the parallel resources and the communication overhead. With regard to the former, parallelizing algorithms must be designed for a bounded number of resources, such that maximum resource utilization is achieved in most cases. As for the latter, the cost of communication between the existing resources must be minimized.

A very comprehensive historical review of scheduling is given in [Jam99]. It classifies important models found in the literature up to 1999 as static, dynamic or hybrid (static and

dynamic), according to the input model (arbitrary graphs, trees, precedence graphs, unrelated tasks, arbitrary task graphs and DAGs), to the task execution time (known or unknown) and finally according to the processors used by the model (homogeneous or heterogeneous).

The problem of scheduling DOACROSS loops with uniform dependencies is a very special case of scheduling *Directed Acyclic Graphs* (DAGs). The general DAG multiprocessor scheduling with precedence constraints is known to be NP-complete [GJ79] [Ull75] even when the number of processors is unbounded [PY88]. Many researchers have tackled the special cases of DAG scheduling [KPT96] [EFKR01] hoping to come up with efficient polynomial algorithms.

Many static methods have focused on accomplishing the optimal time scheduling, the first of which is the hyperplane method [Lam74]. The computations are organized into well defined distinct groups, called wavefronts or hyperplanes, using a linear transformation. The hyperplane method is applicable to all DOACROSS loops with uniform dependencies and all the points belonging to the same hyperplane can be concurrently executed. Darte [DKR91] proved that this method is nearly optimal. Moldovan, Shang, Darte and others applied the hyperplane method to find a linear optimal execution schedule using diophantine equations [MF86], linear programming in subspaces [SF91] and integer programming [DKR91]. The problem of finding a hypersurface that results in the minimum makespan was solved more efficiently in [PAD01]. Other attempts of scheduling uniform DOACROSS loops include the free scheduling method introduced in [KPT96]. One of the goals of this work is to schedule uniform DOACROSS loops so that the optimal parallel processing time is achieved using the minimum number of processors. All these approaches consider unit execution time for each iteration and zero communication for each communication step (UET model). It was assumed that each loop iteration requires exactly the same processing time and there is no communication penalty for the transmission of data among the processors. Another polynomial time algorithm for scheduling in- and out-forrests with unit time computation and communication delays (UET-UCT model) was given in [VRKL96].

In their efforts to minimize the communication cost for fine grained parallelism, several methods for grouping neighboring chains of iterations have been proposed in [SC95] [KN01], while maintaining the optimal linear scheduling vector [ST91] [DGK$^+$00] [TKP00]. The goal of partitioning the iteration space into chains of iterations is to annul the communication for iterations of the same chain and minimizing the inter-chains communication. Hence, some (neighboring or non-neighboring) chains may be grouped and computed by the same processor, thus reducing the overhead of inter-processor communication.

For coarse grained parallelism, many researchers use the tiling transformation to reduce the communication cost. Loop tiling (or loop blocking) is a loop optimization used by compilers to make the execution of certain types of loops more efficient. It was proposed by Irigoin and Triolet in [IT88]. Loop tiling for parallelism [Xue00] is a static transformation and the time complexity of tiling is exponential in the size of the iteration space. The major issues addressed in tiling are: minimizing the communication between tiles [WL91], finding the optimal tile shape [BDRR94] [RS92] [Xue97] and scheduling tiled iteration spaces onto parallel systems [DRR96] [RRP03].

A significant amount of work has been done for determining the optimal partitioning (tile size, block size, grain size) of nested loops for homogeneous systems ([FDR96], [AR97], [Low00], [Xue00], [XC02], [MSK04] and references therein). Desprez et al. presented in [FDR96] a method for overlapping communications on homogeneous systems for pipelined algorithms. He provided a general theoretical model to find the optimal packet size. Andonov

and Rajopadhye addressed in [AR97] the problem of finding the tile size that minimizes the total execution time, on homogeneous systems. Xue studied the problem of time minimal tiling in [Xue00]. Tiles are statically assigned in a block or block cyclic fashion to homogeneous processors. The optimal tile size is determined based on the critical path of the last processor. Xue and Cai presented in [XC02] a solution to the problem of finding the optimal tile size on homogeneous systems, when the rise is larger than zero. Lowenthal et al proposed in [Low00] a method for selecting block size at run-time in pipelined programs; they target problems with irregular workloads on homogeneous systems. Strout et al proposed in [MSK04] a run-time reordering transformation (full sparse tiling) that improves the data locality for stationary iterative methods. The problem of finding the optimal partitioning of iteration spaces for heterogeneous systems has not been given enough attention so far. Chen & Xue proposed in [CX99] a method for obtaining the optimal tile size on heterogeneous networks of workstations, in which the shape and sizes of tiles are statically determined and scheduled using a block distribution approach.

Traditionally, DOACROSS loops were parallelized using static methods that employ either fine grain or coarse grain parallelism. However, DOALL loops have been efficiently parallelized with the use of various dynamic schemes. Dynamic scheduling methods are suitable for dynamic applications that run on a static platform, for static applications that run on a dynamic platform or both.

An important class of dynamic scheduling algorithms for DOALL loops are the self-scheduling schemes: Chunk Self-Scheduling (CSS) [KW85], Guided Self-Scheduling (GSS) [PK87], Trapezoid Self-Scheduling (TSS) [TN93], Factoring Self-Scheduling (FSS) [HSF92]. These algorithms are coarse-grained and devised for DOALL loops without dependencies executed on homogenous systems. Self-scheduling algorithms divide the total number of tasks into chunks, which are then assigned to processors (slaves). In their original form, these algorithms cannot handle loops with dependencies and do not perform satisfactory on non-dedicated heterogeneous systems. A first attempt to make self-scheduling algorithms suitable for heterogeneous systems was Weighted Factoring (WF) proposed in [HSUW96]. WF differs from FSS in that the chunks sizes are weighted according to the processing powers of the slaves. However, with WF processor weights remain constant throughout the parallel execution. Banicescu and Liu proposed in [BL00] a method called Adaptive Factoring (AF), that adjusts the processor weights according to timing information reflecting variations of slaves computation power. This was designed for time-stepping scientific applications. Banicescu et al proposed in [IBV00] a method similar to AF, called Adaptive Weighted Factoring (AWF) scheme. AWF evolves FSS, and newly computed weights are not only based on the performance of particular processors during the previous iteration step, but also on their cumulative performance during all the previous iterations. Chronopoulos et al. extended in [CABG01] the TSS algorithm, proposing the Distributed TSS (DTSS) algorithm suitable for distributed systems. With DTSS the chunks sizes are weighted by the slaves relative power and the number of processes in their run-queue.

1.2.1 What is missing?

As described above, the problem of DOACROSS loop scheduling has been extensively studied in the future and there is a large number of proposed heuristics that aim at satisfying one or more of the parallelization challenges. In order to achieve optimality for a certain criterion, heuristics make simplifying assumptions, especially about the target architectures, which renders them less practical or efficient on current systems. Even though we assume that

problems containing DOACROSS loops are of static nature, the platform may be quite the opposite, i.e. the system may be of dynamic nature. Therefore, there is a need for methods to address: maximization of resource utilization, minimizing the inter-processor communication cost, dynamic scheduling and load balancing, fault tolerance, scalability and reliability.

1.3 Contribution of this Dissertation

This dissertation presents several innovative methods for the parallelization of DOACROSS loops, for various system architectures. Particularly, the contributions are summarized below:

Static methods

- An algorithm for the efficient assignment of computations that uses the least number of processors, which guarantees the optimal parallel execution time [AKC+03a].
- An algorithm for determining the least number of processors required for an optimal static scheduling along hyperplanes [AKC+03b].
- An adaptive cyclic scheduling method [CAP05] which exploits the geometric properties of the index space in order to reach an efficient geometric decomposition of the index space. This algorithm is also considered static.
- A static scheduling algorithm, called chain pattern scheduling [CAD+05], similar to the adaptive cyclic scheduling method, which exploits also the regularity of the index space of nested loops.

Dynamic methods

- A tool was implemented for the automatic parallel code generation, equivalent to the original sequential program with nested loops, such that it is executable on share-memory and distributed-memory systems [CAK+03] [ACT+04]. The proposed scheduling algorithms used by this tool are dynamic, and based on the hyperplane method and methods from computational geometry.
- A new method called dynamic multi-phase scheduling [CAR+06] [PRA+06], that extends several dynamic scheduling schemes originally devised for DOALL loops. This method was bibliographically the first attempt to parallelizing nested DOACROSS loops using a coarse grain approach, on heterogeneous systems and employing dynamic scheduling.
- Two general mechanisms for enhancing the performance of self-scheduling algorithms through synchronization and weighting [CRA+08].
- A theoretical model for determining the optimal synchronization/communication frequency between processors for the execution of DOACROSS loops on heterogeneous systems [CRA+07a].

1.4 Dissertation Overview

In Chapter 2 we describe the basic concepts and give some background on the problem addressed in this work. The static scheduling algorithms that this thesis contributes with are described in Chapter 3, whereas the dynamic scheduling algorithms are exposed in Chapter 4. The implementation, experimental validation and testing of all proposed algorithms are described in Chapter 5. Finally, we draw the conclusions and discuss future research directions in Chapter 6.

1.5 Publications

In this list journal papers are numbered using "J", journal papers under review/revision using "UR", journal papers in submission using "IS" and refereed conferences and workshops using "C".

International journals

J.1 Theodore Andronikos, Florina M. Ciorba, Panagiotis Theodoropoulos, Dimitrios Kamenopoulos and George Papakonstantinou, "Cronus: A platform for parallel code generation based on computational geometry methods", *Journal of Systems and Software*, 2008, Available online: http://dx.doi.org/10.1016/j.jss.2007.11.715.

J.2 Florina M. Ciorba, Ioannis Riakiotakis, Theodore Andronikos, George Papakonstantinou and Anthony T. Chronopoulos, "Enhancing self-scheduling algorithms via synchronization and weighting", *Journal of Parallel and Distributed Computing*, vol. 68, no. 2, pp. 246-264, 2008.

J.3 George Papakonstantinou, Ioannis Riakiotakis, Theodore Andronikos, Florina M. Ciorba and Anthony T. Chronopoulos, "Dynamic Scheduling for Dependence Loops on Heterogeneous Clusters", *Neural, Parallel & Scientific Computations*, vol. 14, no. 4, pp. 359-384, 2006.

J.4 Florina M. Ciorba, Theodore Andronikos and George Papakonstantinou, "Adaptive Cyclic Scheduling of Nested Loops", *HERMIS International Journal*, vol. 8, pp. 69-76, 2006.

International journals under review

UR.1 Ioannis Riakiotakis, Florina M. Ciorba, Theodore Andronikos, George Papakonstantinou and Anthony T. Chronopoulos, "Optimal Synchronization Frequency for Dynamic Pipelined Computations on Heterogeneous Systems", *Journal of Cluster Computing.*

UR.2 Florina M. Ciorba, Ioannis Riakiotakis, Theodore Andronikos, George Papakonstantinou and Anthony T. Chronopoulos, "Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems", *Journal of Performance Evaluation.*

International conferences

C.1 Florina M. Ciorba, Ioannis Riakiotakis, Theodore Andronikos, Anthony T. Chronopoulos, and George Papakonstantinou, "Optimal Synchronization Frequency for Dynamic Pipelined Computations on Heterogeneous Systems", *IEEE International Conference on Cluster Computing (CLUSTER 2007)*, Austin, TX USA, September 17-20, 2007.

C.2 Florina M. Ciorba, Ioannis Riakiotakis, Theodore Andronikos, Anthony T. Chronopoulos, and George Papakonstantinou, "Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems", *PACT '07: Proceedings of the 16th International Conference on Parallel Architectures and Compilation Techniques*, pp. 403, Brasov, Romania, September 15-19, 2007.

C.3 Ioannis Riakiotakis, Florina M. Ciorba, Theodore Andronikos, and George Papakonstantinou, "Self-Adapting Scheduling for Tasks with Dependencies in Stochastic Environments", *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER 2006), Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks (HeteroPar '06)*, pp. 1-8, September 25-28, Barcelona, Spain, 2006.

C.4 Florina M. Ciorba, Theodore Andronikos, Ioannis Riakiotakis, Anthony T. Chronopoulos, and George Papakonstantinou, "Dynamic Multi Phase Scheduling for Heterogeneous Clusters", *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 25-29, Rhodes Island, Greece, 2006.

C.5 Florina M. Ciorba, Theodore Andronikos, Ioannis Drositis, George Papakonstantinou, "Reducing Communication via Chain Pattern Scheduling", *Proceedings of the 4th IEEE Conference on Network Computing and Applications (NCA'05)*, Cambridge, MA, July, 2005.

C.6 Florina M. Ciorba, Theodore Andronikos, George Papakonstantinou, "Adaptive Cyclic Scheduling of Nested Loops", *Proceedings of the 7th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'05)*, Athens, Greece, September, 2005.

C.7 Theodore Andronikos, Florina M. Ciorba, Panagiotis Theodoropoulos, Dimitris Kamenopoulos and George Papakonstantinou, "Code Generation for General Loops Using Methods from Computational Geometry", *Proceedings of the IASTED Parallel and Distributed Computing and Systems Conference (PCDS 2004)*, pp. 348-353, Cambrige, MA USA, November 9-11, 2004.

C.8 Florina M. Ciorba, Theodore Andronikos, Dimitris Kamenopoulos, Panagiotis Theodoropoulos and George Papakonstantinou, "Simple Code Generation for Special UDLs", *In Proceedings of the 1st Balkan Conference in Informatics (BCI'03)*, pp. 466-475, Thessaloniki, Greece, November 21-23, 2003.

C.9 Theodore Andronikos, Marios Kalathas, Florina M. Ciorba, Panagiotis Theodoropoulos and George Papakonstantinou, "An Efficient Scheduling of Uniform Dependence Loops", *In Proceedings of the 6th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'03)*, Athens, Greece, September 25-27, 2003.

C.10 Theodore Andronikos, Marios Kalathas, Florina M. Ciorba, Panagiotis Theodoropoulos, George Papakonstantinou and Panagiotis Tsanakas, "Scheduling nested loops with

the least number of processors", *In Proceedings of the 21st IASTED International Conference on Applied Informatics (AI 2003)*, pp. 713-718, Innsbruck, Austria, February, 2003.

CHAPTER 2

Preliminary Concepts - Background

This chapter starts by defining the problem of task scheduling and enumerating the characteristics of tasks in general. The second step of the parallelization process is decomposing an application into tasks. One of the factors influencing task decomposition is the target system. We describe here the characteristics of the target systems and how they influence the task decomposition and the task scheduling decisions. Next, we describe the general model of nested DO loops and explain the different types of dependencies that may exist in DOACROSS loops. Finally, we present the two models used for representing DOACROSS loops (or tasks with dependencies, generally) and give an emphasis on the Cartesian space representation model.

2.1 The Problem of Task Scheduling

In the previous chapter (1.1) we described the three steps that characterize the parallelization process. The third step is *task scheduling* and the problem of task scheduling is described below.

Definition 2.1.1 (The task scheduling problem) *[Par99]: Given the set of tasks of a parallel computation, determine how the tasks can be assigned to processing resources (scheduled on them) to satisfy certain optimality criteria.*

After the application is partitioned into tasks and the dependencies between tasks are determined, the last step of the parallelization process is scheduling these tasks in time and spatially assigning them to the processors of the target system. The time schedule (temporal assignment) refers to assigning a start time to each task. The spatial assignment (mapping) is the allocation of tasks to processors. The time schedule along with the spatial assignment are generally called *scheduling*.

The optimality criteria may include minimizing execution time, maximizing resource utilization, minimizing inter-processor communication, load balancing computations, handling and/or recovering from failures, meeting deadlines, or a combination of these.

Associated with each task is a set of parameters or characteristics, including one or more of the following:

1. *Execution or running time* - We may be given the worst case, average case, or the probability distribution of a task's execution time.

2. *Creation* - We may be faced with a fixed set of tasks, known at compile time, or a probability distribution for the task creation times.

3. *Relationship with other tasks* - There are two types of tasks: independent tasks (embarrassingly parallel) and dependent tasks, for which the dependency relationship may stem from criticality, priority order, and/or data dependencies.

4. *Start or end time* - A task's release time is the time before which the task should not be executed. Also, a hard or soft deadline may be associated with each task. A hard deadline is specified when the results of a task become practically worthless if not obtained by a certain time. A soft deadline may penalize late results but does not render them totally worthless.

5. *Task granularity* - There are three types of granularities: fine (tasks consist of a single operation or computational unit, e.g. an instance of a nested loop), medium and coarse (tasks consist of multiple operations or computational unit, e.g. general program statements and/or repetitive statements).

According to *when* the scheduling decisions are performed, there are two types of task scheduling methods: static and dynamic. In *static scheduling*, the assignment of tasks to the processors is done before the program execution begins. Information regarding the task execution times and resource requirements are assumed to be known at compile time itself. Static scheduling methods are processor non-preemptive. The goal, usually, is to minimize the execution time of the concurrent program while minimizing the communication delay. Static methods aim at:

- Predicting the program execution behavior at compile time.

- Performing a grouping of smaller tasks into coarser grain processes to reduce communication delays.

- Allocating the coarser grain processes to processors.

Perhaps one of the most critical drawbacks of static scheduling is that generating optimal schedules is an NP-complete problem, hence only restricted solutions (heuristics) can be given. Heuristic methods rely on the rules of the thumb to guide the scheduling processes in the proper track for a near optimal solution. Static scheduling algorithms are in general easier to design and program.

Dynamic scheduling is based on redistribution of tasks at execution time. This is achieved either by transferring the tasks from heavily loaded processors to the lightly loaded ones called load balancing, or by adapting the assigned number of tasks to match the workload variation in the multiprocessor system. Dynamic scheduling is necessary in the following situations: when a static scheduling may result in a highly imbalanced distribution of work among processors; when the task-dependency graph itself is dynamic, thus precluding a static scheduling; or when the target system is not stable enough or if it evolves too fast. The primary reason for using a dynamic scheduling algorithm is balancing the workload among processors, therefore dynamic algorithms are often referred to as *dynamic load balancing algorithms*. Dynamic scheduling algorithms are usually more complicated, particularly in the message-passing programming paradigm.

According to *where* the scheduling decisions are performed, dynamic scheduling algorithms can be centralized or distributed (decentralized). In a *centralized* dynamic load balancing approach, all ready-to-be-executed tasks are maintained in a common central data structure or they are maintained by a special processor or a subset of processors. If a special processor is designated to manage the pool of available tasks, then it is often referred to as the *master* and the other processors that depend on the master to obtain work are referred to as *slaves* or *workers*. Whenever a processor has no work, it takes a portion of available work from the central data structure or the master processor. Whenever a new task is generated, it is added to this centralized data structure or reported to the master processor. This describes the *master-slave* model. Centralized load-balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability. As more and more processors are used, the large number of accesses to the common data structure or the master processor tends to become a bottleneck.

In a *distributed* dynamic load balancing approach, the set of ready-to-be-executed tasks are distributed among processors which exchange tasks at run time to balance work. Each processor can send work to or receive work from any other processor. These schemes do not suffer from the bottleneck associated with the centralized schemes. However, it is more difficult to pair up sending and receiving processors for an exchange, to decide whether the transfer is initiated by the sender (an overburdened processor) or the receiver (a lightly loaded processor), to decide how much work is transferred in each exchange and to decide how often a work transfer should be performed.

The main advantage of the dynamic approach over the static approach is the inherent flexibility allowing for adaptation to the unforeseen application requirements at run-time, though at the cost of communication overheads. The disadvantages are mainly due to communication delays, load information transfer and decision making overheads.

Unfortunately, most interesting task scheduling problems, some with as few as two processors, are NP-complete. This fundamental difficulty has given rise to research results on many special cases that lend themselves to analytical solutions and to great many heuristic procedures that work fine under appropriate circumstances or with tuning of their decision parameters. Stankovic et al. [SSDNB95] present a good overview of basic scheduling results and the boundary between easy and hard problems. El-Rewini et al. [ERAL95] provide an overview of task scheduling in multiprocessors.

Polynomial time optimal scheduling algorithms exist only for very limited classes of scheduling problems. Examples include scheduling tree-structured task graphs with any number of processors and scheduling arbitrary graphs of unit-time tasks on two processors [ERAL95]; scheduling in- and out-forrests for unit time computation and communication delays [VRKL96]; free scheduling of task graphs with unit time computation delays and no communication delays [KPT96]. Most practical scheduling problems are solved by applying heuristic algorithms.

2.1.1 Target Systems

The resources or processors on which tasks are to be scheduled are typically characterized by their ability to execute certain classes of tasks and by their performance or speed. To achieve performance on parallel systems, scheduling algorithms must take into account both parallelism and data locality. Data locality is influenced directly by the memory architecture of the target system. There are two main classes of memory architectures for parallel machines: distributed address space machines and shared address space machines. The *distributed ad-*

dress space machines are built from a group of processors connected via a communication network. Each processor has its own local memory that it can access directly; messages have to be sent across the network for a processor to access data stored in another processors's memory. The most popular example is a cluster of workstations (COW).

In contrast, the *shared address space* machines present the programmer with a single memory space that all processors can access. Many of these machines support hardware cache coherence to keep data consistent across the processors. Shared address space machines can either have a single shared memory that can be uniformly accessed by all the processors, or the memories may be physically distributed across the processors. The former class of machines are often called *centralized* shared address space machines, while the latter class of machines are called *distributed* shared address space machines. Distributing the memories across processors removes the need for a single shared bus going from all the processors' caches to the memory and thus makes it easier to scale to more processors. An example of distributed shared address space systems are the Silicon Graphics Origin commercial machines.

The problem of scheduling for distributed-memory systems is more challenging than for shared-memory systems due to the fact that the communication cost must be taken into account. Two things make the communication in multiprocessor systems more inefficient than in uniprocessors: long latencies due to inter-processor communication and multiprocessor-specific cache misses on machines with coherent caches. Long memory latencies imply that the amount of inter-processor communication in the program is a critical factor for performance. Thus it is important for computations to have good data locality. A computation has good data locality if it re-uses much of the data it has been accessing; programs with high data locality tend to require less communication. It is therefore important to take communication and data locality into consideration when deciding how to parallelize a program and how to assign the tasks to processors.

Apart from the memory architecture, the scheduling algorithms are influenced by the individual performance or speed of every processor in the system. Often uniform capabilities are assumed for all processors, either to make the scheduling problem tractable or because parallel systems of interest do in fact consist of identical processors, called *homogeneous* systems. Obviously, *heterogeneous* systems consist of non-identical processors, with either different processing speeds (processor heterogeneity) or different communication speeds (network heterogeneity), or both. *Processor* heterogeneity arises from having processors with a different architecture, i.e., different processor types, memory types etc. *Network* heterogeneity arises when the speed and bandwidth of communication links between different pairs of processors differs significantly. The problem of optimal distribution of tasks across a system with both processor and network heterogeneity is much more difficult that across a cluster of heterogeneous processors interconnected with a homogeneous high-performance communication network.

According to their delivered performance or speed, both homogeneous and heterogeneous systems can be either *dedicated*, meaning that processors are dedicated to running the program and no other loads are interposed during the execution, or *non-dedicated*, meaning the system may be used by other users as well, affecting the performance of the parallel program. Both cases are examined in this dissertation, for homogeneous systems, as well as for heterogeneous systems.

2.2 Algorithmic Model - Nested DO Loops

In order to parallelize a sequential program, the focus should be on loops. They are very attractive since they represent the most time consuming part of the program, hence the expected gain of parallelizing them is quite large.

The focus of this work are perfectly nested DO loops. They represent iterative (or cyclic) computations. Perfectly DO nested loops have the form shown in Fig. 2.1.

```
for (i1=l1; i1<=u1; i1++) {
        ...
        for (in=ln; in<=un; in++) {
Loop         S1(I);
Body         ...
             Sk(I);
        }
        ...
}
```

Figure 2.1: Algorithmic model

The depth of the loop nest,n, determines the dimension of the iteration index space $J = \{\mathbf{j} \in \mathbb{N}^n \mid l_r \leq i_r \leq u_r, 1 \leq r \leq n\}$. The lower and upper bounds of the loop indices are $\mathtt{l}_i$ and $\mathtt{u}_i \in \mathbb{Z}$, respectively. Each point of this n-dimensional index space is a distinct iteration of the loop and is represented by the vector $\mathbf{j} = (i_1, \ldots, i_n)$. An iteration $\mathbf{j}$ refers to all the statements of the loop that will be executed for the current value of the loop indices. The group of statements executed in one iteration is called loop body, and denoted $LB(\mathbf{j})$. When the loop body contains simple assignment statements, the loop nest is considered to be simple and the execution time of one iteration is identical (or regular) for all iterations. The loop body $LB(\mathbf{j})$, however, can contain general programs statements $(S_i(I))$, that may include assignment statements, conditional **if** statements and repetitions such as **for** or **while**. In this case, the execution time of one iteration is not necessarily the same for all iterations, and these are called irregular loops. $\mathbf{L} = (l_1, \ldots, l_n)$ and $\mathbf{U} = (u_1, \ldots, u_n)$ are the *initial* and *terminal* points of the index space. The cardinality of J, denoted $|J|$ is $\prod_{i=1}^{n}(u_i - l_i + 1)$.

If the iterations of nested DO loops are independent of one another, the loops are called DOALL loops (or parallel loops); otherwise, if there is a dependency relation between the iterations, the loops are called DOACROSS loops (or loops with dependencies).

2.2.1 Dependencies and DOACROSS Loops

Generally, there are two types of dependencies between tasks of an application: *data dependencies* and *control dependencies*. Data dependencies are caused by data transfer between tasks. There are three types of data dependencies:

- *flow (true) dependencies* - one task writes a variable and another tasks reads that variable, thus creating a precedence order for the execution of these two tasks.

- *anti-dependencies* - one task reads a variable and then another tasks writes that variable.
- *output dependencies* - both tasks write on that variable.

Flow dependencies are real dependencies between tasks and cannot be eliminated. Anti-dependencies and output dependencies can be eliminated by modifying programs (e.g. by using a new variable) [CDRV98], [Sin07].

Unlike data dependencies, control dependencies are not created by data transfer but describe the control structure of the program. Control dependencies can be transformed into data dependencies [BENP93] and then the techniques for analyzing data dependencies can be applied. In this work we deal with *flow dependencies* between tasks (i.e. iterations of nested DOACROSS loops), which are also called *inter-iteration* dependencies. We consider that all operations of a task are executed in sequential order, therefore we do not address intra-iteration dependencies in this work.

2.2.2 Representation of DOACROSS Loops

Once the application is decomposed into tasks, these tasks need to be represented somehow. Usually, the set of tasks is defined in the form of a directed acyclic graph (DAG), called *task graph*, with nodes specifying computational tasks and links corresponding to data dependencies or communications. A task graph (TG) differs from a dependence graph (DG) in that its links represent only flow dependencies between nodes (tasks), whereas a DG may contain all types of dependencies. Fig. 2.2 illustrate a TG with seven tasks and eight flow dependencies.

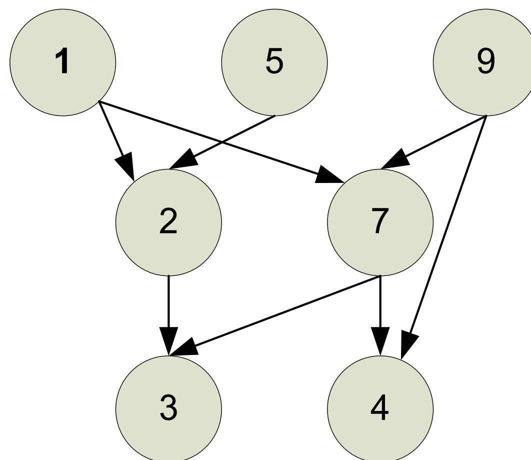


Figure 2.2: Task graph representation of an application with seven tasks and eight dependencies

If the number of tasks is large and/or if the dependencies between tasks are not uniform, the description of a decomposed application using the task graph representation can be very long and may not fit in the available memory, since each dependency must be stated explicitly.

Alternatively, a very popular representation of a nested loop is the Cartesian representation (or iteration dependence graph [Sin07]), in which points of the Cartesian space have coordinates and represent iterations of the DOACROSS loop (or tasks), and the directed vectors represent the inter-iteration dependencies, such as precedence. The description of a decomposed application using the Cartesian representation is succinct, due to the existence of coordinates, which makes it much easier to describe the dependencies between tasks implicitly.

An example of using the Cartesian representation is illustrated in Fig. 2.3. The initial and terminal points are highlighted and five flow dependencies are depicted. In this work we use the Cartesian representaion to model DOACROSS loops. We refer to the directed vectors as uniform dependence vectors. The set of the p dependence vectors is denoted $DS = \{\tilde{\mathbf{d}}_1, \ldots, \tilde{\mathbf{d}}_p\}$, $p \geq n$. The size of the directed vector represents the number of iterations between the two tasks instances forming a dependency relation. If the size is constant, then the dependence vector is uniform.

Definition 2.2.1 (Dependence vectors) *The dependence vectors are always* $>$ *than* $\mathbf{0}$*, where* $\mathbf{0} = (0, \ldots, 0)$ *and* $>$ *is the* lexicographic *ordering.*

Since loops often contain array variables, most dependencies arising in loops are uniform and in the form of array subscripts. In this dissertation, we assume that all dependence vectors are uniform, that is, they are independent of the array indices. In particular, we assume that any loop body $LB(\mathbf{j})$ instance depends on previous loop body instance(s) as follows:

$$LB(\mathbf{j}) = f(LB(\mathbf{j} - \vec{\mathbf{d}}_1), \ldots, LB(\mathbf{j} - \vec{\mathbf{d}}_p)) \tag{2.1}$$

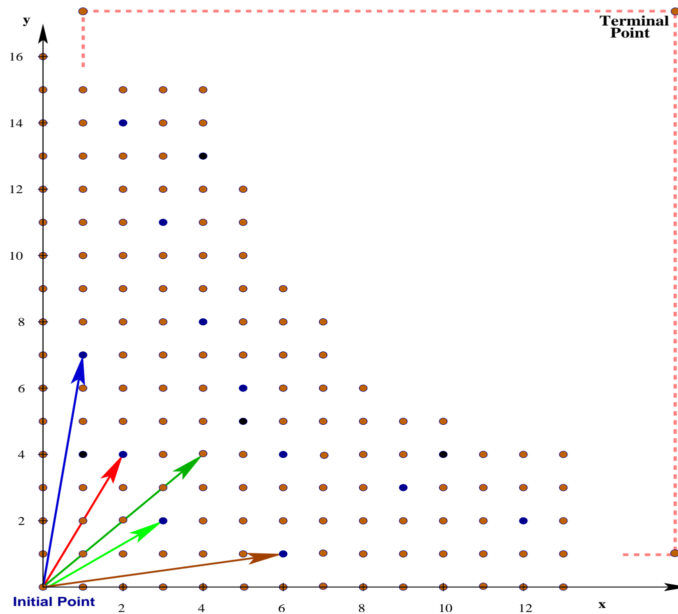


Figure 2.3: Cartesian representation of a 2D DOACROSS loop with five uniform dependencies

CHAPTER 3

Static scheduling algorithms

3.1 What is Static Scheduling?

The main objective of scheduling is to minimize the completion time of a parallel application, while minimizing the communication delay, by properly allocating the tasks to the processors. As mentioned in the Introduction, the scheduling can be static or dynamic, depending on *when* the tasks are assigned to the processors. Static scheduling involves assigning the tasks to processors before the execution of the problem, in a non-preemptive fashion. The characteristics of a parallel program (such as task processing times, communication, data dependencies, and synchronization requirements) are (assumably) known before program execution. Static methods aim at:

- Predicting the program execution behavior at compile time.
- Performing a grouping of smaller tasks into coarser grain processes to reduce communication delays.
- Allocating the coarser grain processes to processors.

Perhaps one of the most critical drawbacks of static scheduling is that generating optimal schedules is an NP-complete problem, hence only restricted solutions (heuristics) can be given. Heuristic methods rely on the rules of the thumb to guide the scheduling processes in the proper track for a near optimal solution. Static scheduling algorithms are in general easier to design and program.

This section gives a few static scheduling algorithms, devised to efficiently schedule DOACROSS loops. Common simplifying assumptions include uniform task execution times, zero inter-task communication times, contention-free communication, full connectivity of parallel processors, and availability of unlimited number of processors. These assumptions may not hold in practical situations for a number of reasons. The first two algorithms described in this section take a fine grained approach to the parallelizing problem, and give optimal time schedules that use the lower bound on the number of processors needed for scheduling. The last two algorithms, approach the problem again in a fine grain fashion, while striving at significantly reducing the communication overhead by enhancing data locality.

The following notations and definitions are used throughout this chapter:

- UDL – uniform dependence (DOACROSS) loop, a loop nest with uniform precedence constraints

- *GRIDs* – a special category of UDLs with unitary dependence vectors that reside on the axes
- PE – a processing element
- i – the current time step
- OET – the optimal execution time, or the least parallel time (assuming that a sufficient number of processors is available)
- LB – the lower bound on the number of processors
- UB – the upper bound on the number of processors
- P – a candidate for the number of processors
- OP – the optimal number of processors, the least number of processors required to achieve the OET, $OP \in [LB, UB]$
- ECT – the earliest computation time
- LCT – the latest computation time
- CR – the crucial points
- J – the iteration (or index) space of a UDL
- ECT_i and LCT_i – J is organized into two disjoint sequences of *time sets* denoted ECT_i and LCT_i, $1 \leq i \leq OET$. ECT_i contains the points of J whose earliest computation time is i whereas LCT_i contains the points of J whose latest computation time is i. $ECT(\mathbf{j})$ and $LCT(\mathbf{j})$ are the earliest and the latest computation time, respectively, of point $\mathbf{j} \in J$
- $CR_i = ECT_i \cap LCT_i$ – the set of *crucial* points that must be executed during time step i
- *Out degree* of $\mathbf{j}$ – the number of dependence vectors originating from $\mathbf{j}$, that end in points of the index space
- *Zero delay* – all points in ECT_i are grouped according to their delay resulting in the sequence D_i^k of points having delay k. We call *crucial* all the points with delay zero (denoted D_i^0). These points *must* be executed during time step i
- U_i – the set of points that are *unavailable* for execution at the beginning of time step i
- A_i – the set of points that are *available* for execution at the beginning of time step i
- NA_i – the set of points that are (or will *become*) *available* for execution at the end of time step i
- R_i – the set of points that were not executed during time step i due to lack of free processors, i.e., the points *remaining* from time step i

- E_h – the total number of points that must be scheduled at this time step (containing points of the current hyperplane h and any other points remained unexecuted from previous hyperplanes)
- *Cone* – the convex subspace formed by q vectors $\mathbf{d}_1, \ldots, \mathbf{d}_q \in \mathbb{N}^n$, $q \leq p$, and is denoted $Con(\mathbf{d}_1, \ldots, \mathbf{d}_q) = \{\mathbf{j} \in \mathbb{N}^n \mid \mathbf{j} = \lambda_1 \mathbf{d}_1 + \ldots + \lambda_q \mathbf{d}_q, \text{ where } \lambda_1, \ldots, \lambda_q \geq 0\}$
- *Non-trivial cones*, or just *cones* – the ones defined exclusively by dependence vectors
- *Trivial cones* – cones defined by dependence vectors and at least one unitary axe vector
- *Hyperplane* k (*wavefront*) – the set of points that can be executed concurrently during time step k. The set is determined by the existing dependence vectors
- R_k – J can be partitioned into disjoint *time subsets* called *regions* and denoted $\mathcal{R}_k$, $k \geq 0$, such that $\mathcal{R}_k$ (i.e., region k) contains the points whose earliest computation time is k, and designates the area between hyperplanes k and $k+1$ as well as the points of hyperplane k. $\mathcal{R}_0$ denotes the boundary (pre-computed[1]) points
- *Optimal hyperplane* - J can be further partitioned into cones, and from the hyperplane of every cone, an optimal hyperplane is determined using the `QuickHull` algorithm
- $\mathbf{d}_c$ – *communication vector*, usually chosen to be the dependence vector that incurs the largest amount of communication (in most cases the vector with the "smallest" absolute coordinates)
- *Pattern* – J can also be partitioned into disjoint *time subsets* denoted Pat_s, $r \geq 0$, such that Pat_s contains those points of J, whose earliest computation time is s
- Pat_s – the *pattern* corresponding to time step s. Pat_0 denotes the boundary (pre-computed) points
- Pat_1 – the initial pattern, i.e., the subset of index points that can be computed initially (during time step 1)
- *Pattern outline* – the upper boundary of each Pat_s, is denoted as $pat_s, s \geq 1$
- *Pattern points* – those index points necessary to define the polygon shape of the pattern
- *Pattern vectors* – those dependence vectors $\mathbf{d}_i$ whose end-points are the pattern points of the initial pattern Pat_1

We make the following assumptions in this chapter:

A.1 The computation begins always at time step 1, hence $OET = ECT(\mathbf{U})$.

A.2 The processors are homogeneous and dedicated to executing the tasks.

A.3 The communication network is homogeneous.

A.4 The loop body consists of simple assignment statements and the dependence vectors are constant.

[1]The *pre-computed* points designate iteration points that do not belong to the first quadrant of the index space, but represent initial values for specific problems.

Assumption **A.2** does not entirely hold for the CPS algorithm, because it also considers heterogeneous processors. Similarly, assumption **A.4** does not hold for CPS because the loop body of the loops considered consists of general program statements. The implementation and experimental validation of the methods presented in this chapter is given in chapter 5.

3.2 Searching for The Minimum Number of Processors

An algorithm for the efficient assignment of computations onto the minimum number of processing elements that guarantees the optimal makespan is described next. The algorithm is called binary search decision algorithm (BSDA) and was published in [AKC+03a]. BSDA is polynomial in the size of the index space and performs a binary search between a lower and an upper bound on the optimal number of processors. This algorithm aims at building an optimal schedule that achieves the optimal parallel processing time using the minimum number of processors. This algorithm follows a fine grained approach by assigning one iteration to a processor, in a centralized manner, by having all scheduling decisions be performed by one processor.

The problem we considered in this subsection can be stated as follows: given a DOACROSS loop that satisfies assumption A.4, we want to find a legal schedule that results in the minimum makespan using the least number of processors. A schedule is legal iff it includes all index points and does not violate any precedence constraint.

In [APT96] [KPT96], an optimal scheduling in terms of the *number of processors* that achieves the optimal makespan, is established. However, their approach is based on the unit execution–zero communication model (UET). As it has been shown in [PAD01], a UDL with unit execution–unit communication time (UET-UCT) can be reduced to an equivalent UDL with UET; hence, in that sense, their approach also applies to UDLs that consider the UET-UCT model. By ignoring the communication delays the criteria optimality are restricted to the total number of processing elements. A preliminary version of BSDA was proposed in [APT96] and [PKA+97]. Herein, we present how BSDA is employed for DOACROSS loops with arbitrary computation and communication costs.

3.2.1 Computing the ECT and LCT Subsets

Given a UDL, ECT_1 contains those points $\mathbf{j}$ that initially do not depend on other points. Every subsequent ECT_{i+1} set, $1 \leq i \leq OET - 1$, contains those points $\mathbf{j}$ that depend on points belonging to one of the previous ECT sets, at least one of which must be ECT_i. After the ECT subsets are computed with any of the known methods, e.g. from [DAK+01] [PAD01], the LCT of any index point $\mathbf{j}$ can also be computed using the following formula from [AKC+03b]:

$$LCT(\mathbf{j}) = OET - ECT(\mathbf{U} - \mathbf{j} + \mathbf{L}) + 1 \tag{3.1}$$

in one sweep of the index space. The CR subsets are formed by simply taking the intersection of the ECT and LCT sets.

Example 3.2.1 *Consider a 2D index space representing the following double DOACROSS loop in unit-increment steps:*

```
for (i=1; i<=10; i++) {
    for (j=1; j<=10; j++) {
```

```
            A[i,j]=A[i-3,j-1]+3*A[i-4,j-2]+1;
            B[i,j]=2*B[i-2,j-2]-1;
      }
}
```

The index space J in this case is $\{(i,j) \in \mathbb{N}^2 \mid 1 \leq i \leq 10, 1 \leq j \leq 10\}$, the initial point $\mathbf{L}$ *is* $(1,1)$*, the terminal point* $\mathbf{U}$ *is* $(10,10)$ *and the dependence vectors are* $\mathbf{d}_1 = (3,1), \mathbf{d}_2 = (4,2), \mathbf{d}_3 = (2,2)$*. The dependence vectors express the fact that iteration* $\mathbf{j}$ *depends on the* $\mathbf{j}-\mathbf{d}_1, \mathbf{j}-\mathbf{d}_2, \mathbf{j}-\mathbf{d}_3$ *previous iterations; for instance, to compute* `A[i,j]` *we must have already computed* `A[i-3,j-1]` *and* `A[i-4,j-2]`*. The ECT and LCT sets and their intersection (CR sets) are depicted in Figure 1. The minimum makespan OET is determined to be 5. All the points of ECT_1* can *be executed during time step 1. However, only those points that belong to CR_1* must *be executed during time step 1.* ◁

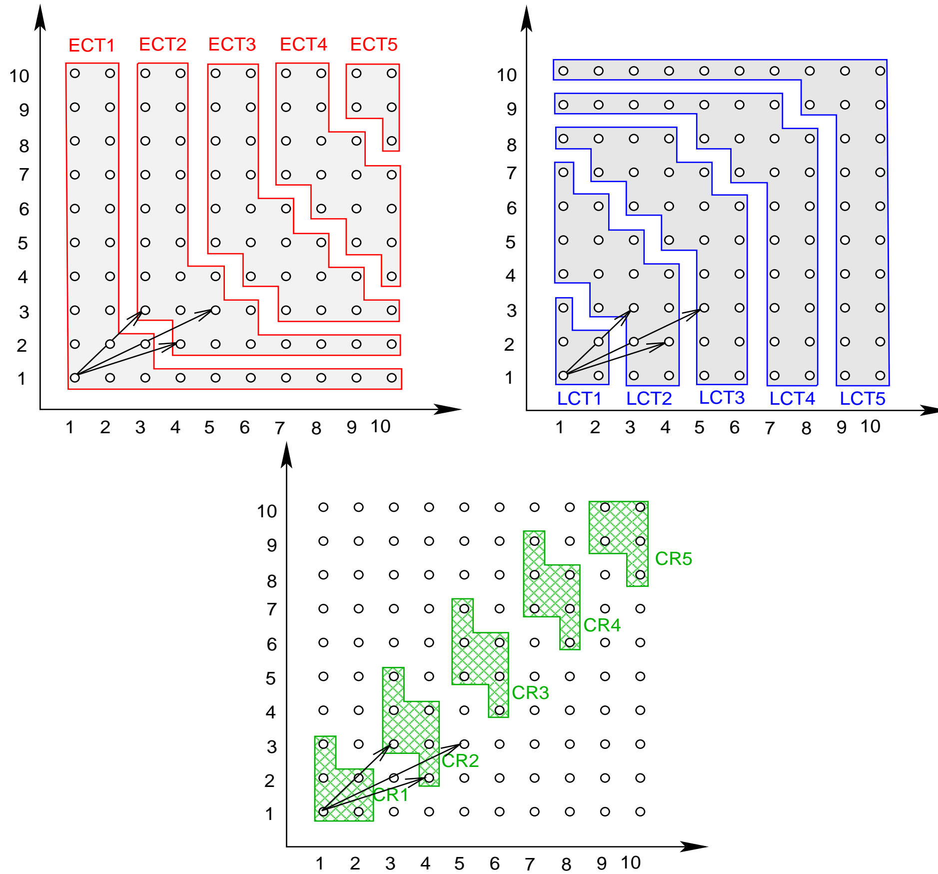


Figure 3.1: Representation of ECT, LCT and CR sets for Example 3.2.1

3.2.2 Overview of the Binary Search Decision Algorithm

For any given n-dimensional UDL, we construct the sequences ECT_i and LCT_i, $1 \leq i \leq OET$. Consider that i is the ECT for point $\mathbf{j}$, hence $\mathbf{j} \in ECT_i$. If $\mathbf{j} \in LCT_i$ too, then the latest time step it can be executed is i. Hence $\mathbf{j}$ is a crucial point and $\mathbf{j} \in CR_i$. This yields that no

delay is possible for this point and the algorithm will mandatorily execute **j** during time step i. However, if $\mathbf{j} \in LCT_{i+k}$, $k > 0$, then we could delay its execution until time step $i + k$, in which case **j** would be considered to have delay k.

The binary search decision algorithm (BSDA) always schedules the crucial points at their appropriate time steps, i.e., it schedules the points of CR_i during time step i. If there are *not* enough processors to schedule all the crucial points at any time step, then the algorithm will return **NO** indicating it is impossible to to produce a legal schedule in OET using this number of processors. Otherwise, considering there are enough processors to schedule all the crucial points, and still have some free processors, it continues to schedule points with delay 1, i.e., $k = 1$. If the points with delay 1 are more than the free processors, then, in order to establish some kind of priority, they are sorted in increasing order of their earliest computation time, and those with the same ECT value are sorted in decreasing order of their *out degree*. If some processors still remain free after all points with delay 1 have been assigned, we proceed to schedule the points with delay 2 on these processors, and so on.

I. The Binary Search Algorithm – Description

The binary search decision algorithm (BSDA) consists of two parts: the binary search algorithm (BSA) and the decision algorithm (DA). The BSA takes as input an algorithm $A(J, DS)$ and computes the ECT, LCT and CR sets. It determines the LB and UB and performs a binary search between LB and UB. For every candidate value `P` that results from this search, BSA calls DA to decide whether there exists a legal schedule with `P` processors. Hence, the smallest natural OP for which DA returns **YES** and the corresponding schedule are produced.

Searching Between the Lower and Upper Bounds

The lower bound on the number of processors is given by:

$$LB = \max\{\left\lceil \frac{|J|}{OET} \right\rceil,\ \max\{|CR_i|\}\},\quad 1 \leq i \leq OET \tag{3.2}$$

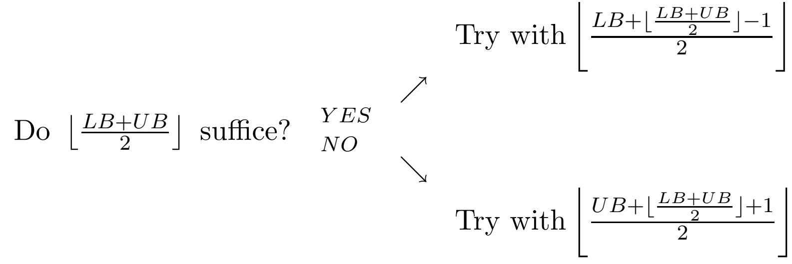
With fewer than LB processors there *is no* legal way of scheduling $|J|$ points in OET time steps.
The upper bound on the number of processors is given by:

$$UB = \max\{|ECT_1|, \ldots, |ECT_{OET}|\} \tag{3.3}$$

where $|ECT_i|$ is the cardinality of ECT_i. With UB processors *we can* always legally schedule a UDL achieving OET. However, the produced schedule might not be the optimal, if there is at least one idle processor at a given time step.

A binary search is performed between the lower and upper bounds in order to narrow down, as fast as possible, the search for the number of processors that suffice for the schedul-

ing. The search goes as follows:



$\left\lfloor \frac{LB+UB}{2} \right\rfloor$ processors are used in the beginning, and if a schedule is found, then the search proceeds with $\left\lfloor \frac{LB+\lfloor \frac{LB+UB}{2} \rfloor -1}{2} \right\rfloor$ processors, getting closer to LB. If no schedule can be found, the search proceeds with $\left\lfloor \frac{UB+\lfloor \frac{LB+UB}{2} \rfloor +1}{2} \right\rfloor$ processors, getting closer to UB, and so on, until the OP that suffice for the legal and optimal schedule is reached.

Example 3.2.2 *(continued from 3.2.1) In this example,* $|J| = 100$*, therefore* $\left\lceil \frac{|J|}{OET} \right\rceil = 20$. *Since* $\max\{|CR_i|\} = \max\{5, 6, 6, 6, 5\} = 6$*, then* $LB = \max\{20, 6\} = 20$*. Similarly,* $UB = \max\{29, 26, 23, 17, 5\} = 29$.

The binary search is performed between these bounds as follows: it starts with trying to find a legal schedule using $\left\lfloor \frac{20+29}{2} \right\rfloor = 24$ *processors. If a schedule is found, then the process goes on with trying with* $\left\lfloor \frac{20+24-1}{2} \right\rfloor = 21$ *processors, and if there is no legal schedule with 21 processors, it tries with* $\left\lfloor \frac{29+24+1}{2} \right\rfloor = 27$ *processors, and so on.* ◁

II. The Decision Algorithm – Description

The algorithm described below is basically a decision one, which takes as input a candidate number `P` of processors and *decides* whether there is a legal schedule with `P` processors can be produced in OET time steps. If the answer is **YES** it also outputs the corresponding schedule.

Example 3.2.3 *(continued from 3.2.2) We obtained earlier the* $LB = 20$ *and the* $UB = 29$*. We want to see if the DA can produce a legal schedule with LB processors. When DA starts, at the beginning of each time step we have 20 available processors. We initialize* $A_1 = ECT_1$ *containing 29 points (see Figure 3.2.1).*

Time step 1: The first points chosen for scheduling are $CR_1 = A_1 \cap LCT_1$*.* $|CR_1| = 5$*, so they are assigned to the first 5 processors and N becomes 15. Then the* 11 *points of* $A_1 \cap LCT_{1+1}$ *are chosen and assigned to the next 11 available processors, N becoming 4. Next, the* $|A_1 \cap LCT_{1+2}| = 5$ *points are chosen for scheduling, but because* $N < |A_1 \cap LCT_{1+2}|$*, they are sorted in increasing order of their ECT value, and those with the same ECT value in decreasing order of their out degree; the first 4 points are assigned to the last 4 free processors and N becomes -1, which means there is one point remaining unexecuted from this time step which is included in* R_1*.*

Time step 2: Since there is one unexecuted point from the previous time step, the points (at this time step) that depend on it form the U_2 *set, that contains 4 points. The set* $A_2 = (ECT_2 - U_2) \cup R_1$ *that contains 31 points is calculated, and the points available for execution at this time step are designated from this set. The first selected points are* $|A_2 \cap LCT_2| = 6$ *which are assigned to the first 6 free processors, N becoming 14. Next, the* $|A_2 \cap LCT_{2+1}| = 13$

Figure 3.2: Decision Algorithm

INPUT: ECT_i, LCT_i, $1 \leq i \leq OET$, P
OUTPUT: **YES** *(and a schedule) iff* P *PEs execute* $A(J, DS)$ *in* OET
Initialization: $A_1 = ECT_1$
for $(i = 1;\ i \leq OET;\ i++)$ **do**
  N = P;
  if $i == 1$ **then**
    Go to Step 2.
  end if
  if $i == 2$ **then** ▷ Step **1**
    $U_2 = \{\mathbf{j} \in ECT_2 \,|\, \exists\, \mathbf{i} \in R_1\ \exists\, \mathbf{d} \in DS\ \mathbf{j} = \mathbf{i} + \mathbf{d}\}$
  else
    $U_i = \{\mathbf{j} \in ECT_i \,|\, \exists\, \mathbf{i} \in (R_{i-1} \cup U_{i-1}) \exists\, \mathbf{d} \in DS\ \mathbf{j} = \mathbf{i} + \mathbf{d}\} \cup U_{i-1}$
  end if
  $A_i = (ECT_i - U_i) \cup R_{i-1}$
  if $\mathtt{N} < |A_i \cap LCT_i|$ **then** ▷ Step **2**
    Go to Step 6
  end if
  Assign a PE to each $\mathbf{j} \in A_i \cap LCT_i$
  $\mathtt{N} = \mathtt{N} - |A_i \cap LCT_i|$
  if $\mathtt{N} > 0 \wedge A_i \cap LCT_{i+k} \neq \emptyset$, for some k **then**
    Go to Step 3
  else
    Go to Step 4
  end if
  while $N > 0$ **do** ▷ Step **3**
    k_0 is the *smallest* natural k such that $A_i \cap LCT_{i+k} \neq \emptyset$
    if $|A_i \cap LCT_{i+k_0}| > N$ **then**
      Sort $A_i \cap LCT_{i+k_0}$ in order of increasing ECT & decreasing out-degree
    end if
    Assign as many PEs as possible to $\mathbf{j} \in A_i \cap LCT_{i+k_0}$
    $\mathtt{N} = \mathtt{N} - |A_i \cap LCT_{i+k_0}|$
  end while
  if $i \neq 1$ **then** ▷ Step **4**
    $NA_i = \{\mathbf{j} \in U_i | \forall\, \mathbf{d} \in DS, \mathbf{j} - \mathbf{d}$ was executed or is out of the index space$\}$
    $U_i = U_i - NA_i$
    $R_i = (A_i - \{$points executed during time step i $\}) \cup NA_i$
  else
    $R_1 = A_1 - \{$points executed at 1$\}$
  end if
end for
OUTPUT: **YES** and terminate.
OUTPUT: **NO** and terminate.

points are selected and assigned to the next 13 available processors and N becomes 1. Then the $|A_2 \cap LCT_{2+2}| = 7$ points are selected and, because $N < |A_2 \cap LCT_{2+2}|$, they are sorted as described in time step 1, and then the first point is scheduled to the last free processor and N becomes -6. The subset NA_2 containing 3 points is calculated, and those points initially belonging to U_2 that now *became available for execution are designated, because they do not depend anymore on any unexecuted points. The subset of unavailable points at this time step, U_2, is recalculated by subtracting the points of NA_2, resulting in one point. The 14 remaining points at this time step, included in R_2, are the points of A_2 that were not yet executed, along with the points of NA_2. The algorithm runs for each time step until $OET = 5$ is reached. The resulting schedule is shown in the Table 3.1.*

Conclusion. *For this example, DA produced a legal schedule using $N = LB = 20$ processors; since there are no idle processors at any time step, maximum processor utilization is achieved.* ◁

Table 3.1: Optimal schedule with $N = LB = 20$ processors for the UDL of Example 3.2.1

Time step	1	2	3	4	5
Processor id	Iterations				
0	(2,2)	(3,3)	(5,5)	(7,7)	(9,9)
1	(2,1)	(3,4)	(5,6)	(7,8)	(9,10)
2	(1,3)	(3,5)	(5,7)	(7,9)	(10,8)
3	(1,2)	(4,2)	(6,4)	(8,6)	(10,9)
4	(1,1)	(4,3)	(6,5)	(8,7)	(10,10)
5	(4,1)	(4,4)	(6,6)	(8,8)	(8,10)
6	(3,2)	(3,6)	(1,9)	(8,5)	(10,5)
7	(3,1)	(3,7)	(2,9)	(8,4)	(10,6)
8	(2,6)	(3,8)	(8,1)	(8,3)	(10,7)
9	(2,5)	(4,5)	(7,2)	(6,9)	(9,3)
10	(2,4)	(4,6)	(4,9)	(5,9)	(10,3)
11	(2,3)	(4,7)	(8,2)	(10,1)	(9,4)
12	(1,7)	(4,8)	(3,9)	(9,1)	(6,10)
13	(1,6)	(5,2)	(5,8)	(2,10)	(10,4)
14	(1,5)	(5,3)	(6,7)	(1,10)	(9,8)
15	(1,4)	(5,4)	(6,8)	(4,10)	(9,7)
16	(6,1)	(6,2)	(7,3)	(9,2)	(9,6)
17	(5,1)	(6,3)	(7,4)	(10,2)	(9,5)
18	(2,8)	(1,8)	(7,5)	(3,10)	(8,9)
19	(2,7)	(7,1)	(7,6)	(5,10)	(7,10)

3.2.3 Complexity of the BSDA Algorithm

BSDA is linear in the size of J. The most computationally expensive step is the calculation of the two sequences of ECT_i and LCT_i sets. Due to the uniformity of the index space this can be done in linear time in the size of J. Computing the ECT subsets requires that we traverse the index space only once, whereas the LCT subsets are computed using eq.(3.1). The CR subsets are computed by intersecting each ECT with the corresponding LCT subset.

Throughout the steps 1-6 of the DA, every point of the index space is examined at most twice. This examination depends only on the number of dependence vectors, p, and the dimension of the index space n which are independent of the size of the index space J. Therefore, we conclude that each run of DA can be done in $\Theta(|J|)$ time. A trivial upper bound on the number of DA runs is log UB, which is bounded above by $\log|J|$. Thus, the overall time complexity of BSDA is $O(|J|\log|J|)$.

3.3 Assigning Tasks on The Minimum Number of Processors

The problem of finding a legal schedule that results in the minimum makespan using the least number of processors is addressed again in this section, and a fine grain, centralized scheme is presented, published in [AKC+03b].

The approach described here is a different approach than the previous algorithm. A new dynamic lower bound on the number of processors is presented, and the decision algorithm (DA), described in the previous section, is employed to verify that all DOACROSS nested loops can be legally scheduled with this lower bound.

As before, the index space is organized into the sequences of disjoint time-subsets ECT_i and LCT_i, $i \geq 1$, such that ECT_i contains the points whose *earliest computation time* is i, and LCT_i contains those points whose *latest computation time* is i. The LCT of an index point is computed using eq. (3.1). If the ECT and the LCT of an index point are both i, then we say that the point belongs to ECT_i and has *zero delay*. On the other hand, if the ECT of an index point is i but its LCT is $i+1$, we say that this point belongs to ECT_i but has a delay of *one* time step. Following this direction, we form partitions of ECT sets according to the delay corresponding to each point, i.e., those that have zero delay in D_i^0, those that have delay 1 in D_i^1, and so on. The intersection of each ECT_i with the corresponding LCT_i is D_i^0 and we denote these points as *crucial*.

3.3.1 Computing the Lower and Upper Bounds of Processors

For any given UDL, depending on the cardinality of ECT_1 set, there may exist three major *lower bounds*. The most trivial one is $LB_1 = \left\lceil \frac{|J|}{OET} \right\rceil$. If $|ECT_1| > LB_1$, during the first time step all free processors are utilized and for most UDLs the OET is achieved using $OP = LB_1$, due to the geometry of the index space, that allows to obtain maximum processors utilization at each time step.

If $|ECT_1| < LB_1$ it is possible, however, that a legal schedule cannot be found with LB_1 processors in OET. On the other hand, at each time step there is the sequence of crucial points D_i^0. Therefore, another lower bound is $LB_2 = \max\{|D_i^0|\}$, since with fewer than LB_2 processors it is impossible to produce a schedule in OET. Unfortunately, in many cases even this does not suffice. LB_2 is a static lower bound, in the sense that it considers the number of crucial points as a constant for each time step, regardless of the number of processors, which is not an accurate assumption. For instance, if some of the points of D_i^1 are not scheduled during time step i, they become crucial during time step $i+1$, so the correct number of crucial points during time step $i+1$ is greater than the initially computed D_{i+1}^0 points, therefore it is not a constant number. This partitioning of the ECT sets into D_i^k sets is depicted in Fig. 3.3.1 for Example 3.3.1.

It becomes obvious that a better and improved lower bound is needed. The new lower bound (LB_3) described in this section takes into consideration the propagation of points with

delays, remained unexecuted from one time step to another. In order to determine LB_3 a sequence of linear inequalities of the form below must be solved:

> *min* $P_h \in \mathbb{N}$ *such that* $P_h \geq E_h$ *where* $1 \leq h \leq OET$, P_h *is the candidate number of processors required for hyperplane* h, *and* $LB_3 = \max\{P_h\}$

For the first hyperplane, the candidate number of processors to schedule the crucial points of ECT_1 must satisfy the inequality:

$$P_1 \geq E_1 = D_1^0 \tag{3.4}$$

For every subsequent hyperplane h, a check is performed to determine if the number of processors we have found so far P_{h-1} is sufficient. This is done by computing the total number of points that must be scheduled at this time step, using the formula:

$$E_h = D_h^0 + \sum_{k=1}^{h-1}(\sum_{i=0}^{h-k} D_k^i - P_{h-1}) \tag{3.5}$$

where h, $1 \leq h \leq OET$, is the current hyperplane. If $E_h \leq P_{h-1}$ then P_{h-1} processors suffice to schedule the crucial points of the first h ECT sets at this time step also, hence there is no need for more processors. In this case, P_h is set to $P_h = P_{h-1}$ and the method continues with the next hyperplane.

On the other hand, if $E_h > P_{h-1}$ it is obvious that more processors are needed and the exact number of processors, P_h, necessary for the current hyperplane, is computed as follows. The following condition must hold:

$$P_h \geq E_h = D_h^0 + \sum_{k=1}^{h-1}(\sum_{i=0}^{h-k} D_k^i - P_h) \tag{3.6}$$

In order to solve meaningfully the above inequality, h cases must be examined. Clearly $P_h \geq D_h^0$ and it has to be determined whether $P_h \leq \sum_{i=0}^{h-k} D_k^i$ or $P_h > \sum_{i=0}^{h-k} D_k^i$, where $1 \leq k \leq r-1$. Hence, the $r-1$ terms of $\sum_{i=0}^{h-k} D_k^i$ are computed, then ordered in increasing order along with D_h^0 (yielding a sequence of h terms $t_1 \leq \ldots \leq t_h$) and the inequality (3.6) is solved initially by assuming that $t_1 \leq P_h \leq t_2$; if no solution is found, it is assumed that $t_2 < P_h \leq t_3$, and so on, the first solution for (3.6) is found. The worst case scenario is when $P_h > t_h$.

We must stress the fact that in solving (3.6) the physical meaning of E_h must be taken into consideration. For instance if $\sum_{i=0}^{h-k} D_k^i < P_h$, then in E_h the term $\sum_{i=0}^{h-k} D_k^i - P_h$ is not taken as a negative integer, which would have no meaning, but is taken instead as zero, meaning that the index points belonging to this set were all scheduled and none remained.

In the end, LB_3 is the last dynamically computed P_h that suffices for legally scheduling all the index points of the DOACROSS loop nest. The *final* lower bound is taken to be: $LB = \max\{LB_1, LB_2, LB_3\}$.

Let us define $UB = \max\{|ECT_1|, \ldots, |ECT_{OET}|\}$ as the trivial *upper bound* of processors and let us say that with UB processors *we can* legally schedule any DOACROSS achieving the OET.

Example 3.3.1 *Consider the 2D index space that represents the following DOACROSS loop nest:*

```
for (i=1; i<=18; i++) {
    for (j=1; j<=18; j++) {
        A[i,j]=A[i-1,j-4]+3*A[i-4,j-1]+1;
    }
}
```

◁

The *ECT* sets divided into D_i^k subsets according to the delay value are depicted in Fig 3.3.1.

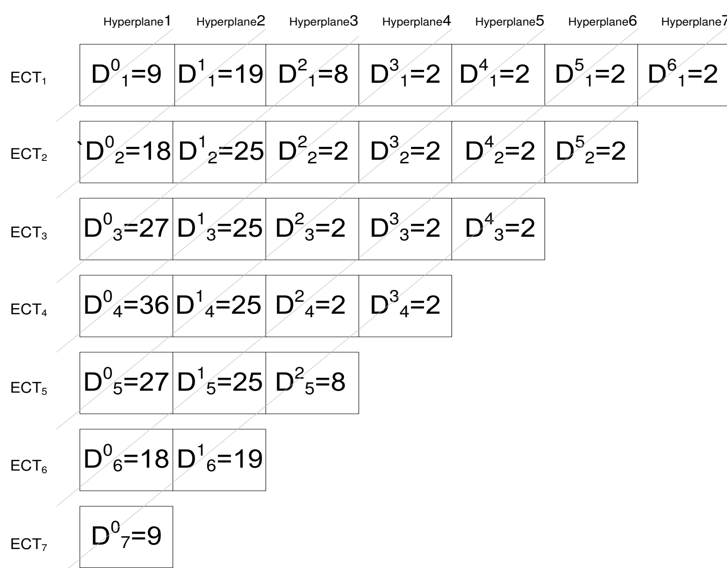


Figure 3.3: *ECT* sets partitioned into D_i^k sets for Example 3.3.1

For the first hyperplane, the inequality below holds:

$$P_1 \geq D_1^0 = 9$$

hence $P_1 = 9$. For the second hyperplane, a check is performed to determined if P_1 processors suffice using 3.5.

$$E_2 = D_2^0 + (D_1^0 + D_1^1 - P_1) = 37$$

hence $E_2 > P_1$, and therefore more processors are needed. The following condition must hold:

$$P_2 \geq D_2^0 + (D_1^0 + D_1^1 - P_2) \Rightarrow P_2 \geq 18 + (28 - P_2)$$

In order to solve the above inequality, the case where $18 \leq P_2 \leq 28$ must be examined and a solution is sought by solving the equation:

$$P_2 = 18 + (28 - P_2) \Rightarrow P_2 = 23$$

which satisfies the above condition. The method proceeds to the third hyperplane and checks if P_2 processors suffice for scheduling, using:

$$E_3 = D_3^0 + (D_1^0 + D_1^1 + D_1^2 - P_2) + (D_2^0 + D_2^1 - P_2) = 60$$

hence $E_3 > P_2$, and therefore more processors are needed again. The following condition is imposed:

$$P_3 \geq D_3^0 + (D_1^0 + D_1^1 + D_1^2 - P_3) + (D_2^0 + D_2^1 - P_3)$$
$$\Rightarrow P_3 \geq 27 + (36 - P_3) + (43 - P_3)$$

In order to solve the above inequality, the case where $27 \leq P_3 \leq 36$ must be examined and a solution is sought by solving the equation:

$$P_3 = 27 + (36 - P_3) + (43 - P_3) \Rightarrow P_3 = 36$$

which satisfies the previous condition. The method advances to the fourth hyperplane and checks if P_3 processors suffice.

$$E_4 = D_4^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 - P_3) + (D_2^0 + D_2^1 + D_2^2 - P_3) + (D_3^0 + D_3^1 - P_3) = 63$$

hence $E_4 > P_3$ and again more processors are needed. The following condition is imposed:

$$P_4 \geq D_4^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 - P_4) + (D_2^0 + D_2^1 + D_2^2 - P_4) + (D_3^0 + D_3^1 - P_4)$$
$$\Rightarrow P_4 \geq 36 + (38 - P_4) + (45 - P_4) + (52 - P_4)$$

Similarly, in order to solve the above inequality the case where $36 \leq P_4 \leq 38$ must be examined and a solution is sought by solving the equation:

$$P_4 = 36 + (38 - P_4) + (45 - P_4) + (52 - P_4) \Rightarrow P_4 = 43$$

which *does not* satisfy the last condition. Therefore the case where $38 < P_4 \leq 45$ must be examined and the same equation is solved as above, taking the first parenthesis as zero instead of a negative integer:

$$P_4 = 36 + (45 - P_4) + (52 - P_4) \Rightarrow P_4 = 45$$

which satisfies the condition. Next, the method advances to the fifth hyperplane and checks if P_4 processors suffice.

$$E_5 = D_5^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 + D_1^4 - P_4) + (D_2^0 + D_2^1 + D_2^2 + D_2^3 - P_4) + (D_3^0 + D_3^1 + D_3^2 - P_4) + (D_4^0 + D_4^1 - P_4) = 49$$

hence $E_5 > P_4$ and again more processors are needed. The condition below is imposed:

$$P_5 \geq D_5^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 + D_1^4 - P_5) + (D_2^0 + D_2^1 + D_2^2 + D_2^3 - P_5) + (D_3^0 + D_3^1 + D_3^2 - P_5) + (D_4^0 + D_4^1 - P_5)$$
$$\Rightarrow P_5 \geq 27 + (40 - P_5) + (47 - P_5) + (54 - P_5) + (61 - P_5)$$

and the case where $27 \leq P_5 \leq 40$ is examined first, and a solution is sought by solving the equation:

$$P_5 = 27 + (40 - P_5) + (47 - P_5) + (54 - P_5) + (61 - P_5) \Rightarrow P_5 = 46$$

which *does not* satisfy the latest condition. Therefore, the case where $40 < P_5 \leq 47$ is examined and the same equation is solved as above, taking the first parenthesis as zero:

$$P_5 = 27 + (47 - P_5) + (54 - P_5) + (61 - P_5) \Rightarrow P_5 = 48$$

which *does not* satisfy the condition. Again, the case when $47 < P_5 \leq 54$ must be examined and the same equation is solved as above, taking the first two parentheses as zero:

$$P_5 = 27 + (54 - P_5) + (61 - P_5) \Rightarrow P_5 = 48$$

which this time satisfies the latest condition. The method proceeds to the sixth hyperplane and checks whether P_5 processors suffice.

$$E_6 = D_6^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 + D_1^4 + D_1^5 - P_5) + (D_2^0 + D_2^1 + D_2^2 + D_2^3 + D_2^4 - P_5) + (D_3^0 + D_3^1 + D_3^2 + D_3^3 - P_5) + (D_4^0 + D_4^1 + D_4^2 - P_5) + (D_5^0 + D_5^1 - P_5) = 40$$

hence $E_6 < P_5$. Since all the points of this hyperplane can be scheduled with P_5 processors, there is no need to estimate a new value for the number of processors and P_6 is considered to be equal to P_5. On the last hyperplane, it checks if P_6 processors suffice.

$$E_7 = D_7^0 + (D_1^0 + D_1^1 + D_1^2 + D_1^3 + D_1^4 + D_1^5 + D_1^6 - P_6) + (D_2^0 + D_2^1 + D_2^2 + D_2^3 + D_2^4 + D_2^5 - P_6) + (D_3^0 + D_3^1 + D_3^2 + D_3^3 + D_3^4 - P_6) + (D_4^0 + D_4^1 + D_4^2 + D_4^3 - P_6) + (D_5^0 + D_5^1 + D_5^2 - P_6) + (D_6^0 + D_6^1 - P_6) = 36$$

hence $E_7 < P_6$. Since all the points of this hyperplane can be scheduled with P_6 processors, P_7 is considered to be equal to P_6 and OP required for the optimal schedule is the last computed P_h, hence $LB = P_7 = 48$.

3.4 Static Scheduling by Exploiting the Geometric Properties of the Iteration Space

In this section, two methods that perform static scheduling by exploiting the geometric properties of the iteration space are presented. The first is called adaptive cyclic scheduling (ACS), published in [CAP05], which by exploiting the geometric properties of the index space reaches an efficient geometric decomposition that reduces the communication cost imposed by certain dependence vectors. The second method is called chain pattern scheduling (CPS), published in [CAD+05], and is rather similar to the ACS method, however performs a different assignment to processors of the partitioned index space. Both methods aim at reducing the communication cost by assigning chains of computations (created by the a certain dependence vector) to the same processor, and mapping dependent chains on the same processor as much as possible.

3.4.1 Adaptive Cyclic Scheduling - ACS

As mentioned above, ACS exploits the geometric properties of the iterations space in order to reach an efficient geometric decomposition of the iterations space. This algorithm is also a centralized scheme with fine grained parallelism approach.

ACS is a based on [DAK+02] and [CAK+03]. In [DAK+02] a nearly optimal solution to the loop scheduling problem, considering communication costs is given, which attains a makespan with constant delay from the earliest possible time. In [CAK+03], the index space of a DOACROSS loop is replicated with patterns of independent points, and a new index space is obtained by mapping every such pattern to an iteration point of the new index space; the scheduling is performed along the hyperplanes of the new index space, in a round-robin fashion among the available processors. In contrast with the two works cited above, ACS accounts for the communication cost, while being suitable for the two most common system types: homogeneous and heterogeneous.

In this approach, the index space of a UDL is partitioned into regions. J is further divided into cones, according to the existing dependence vectors. Using the `QuickHull` algorithm [BDH96], from the hyperplane of every cone an optimal scheduling hyperplane is determined, that best exploits the available parallelism. These partitions are illustrated in Fig. 3.4 for the Example 3.4.1.

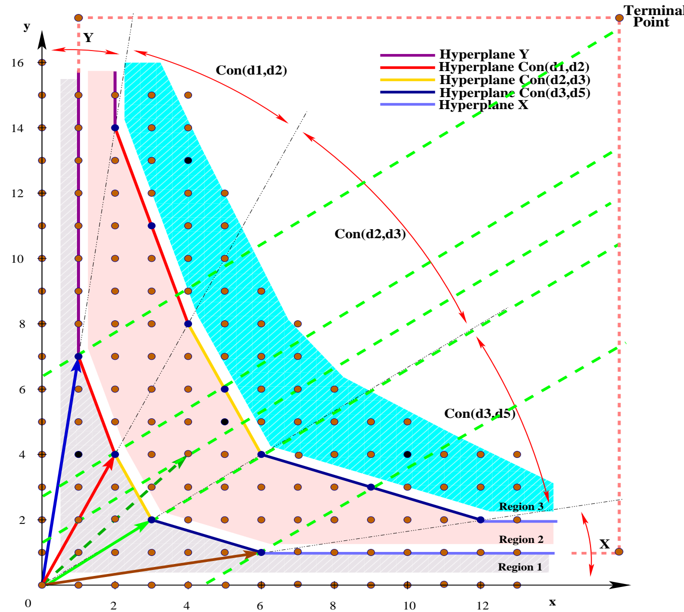


Figure 3.4: Geometrical representation of the index space, the regions, the cones, the cone vectors, the communication vector, some chains and the optimal hyperplane of Example 3.4.1

Example 3.4.1 *Consider the 2D index space of a UDL with the following dependence vectors:* $\mathbf{d}_1 = (1,7)$*,* $\mathbf{d}_2 = (2,4)$*,* $\mathbf{d}_3 = (3,2)$*,* $\mathbf{d}_4 = (4,4)$ *and* $\mathbf{d}_5 = (6,1)$*. Except* $\mathbf{d}_4$*, all other vectors belong to cone vector sequence:* $CV = \langle \mathbf{d}_1, \mathbf{d}_2, \mathbf{d}_3, \mathbf{d}_5 \rangle$*. The 5 cones that formed are colored respectively and the first 3* regions *of all cones are shown (see Fig. 3.4). The communication vector in this example is* $\mathbf{d}_c = \mathbf{d}_3$*.* ◁

Note that only *certain* dependence vectors contribute to the formation of every hyperplane in each cone. These vectors are called *cone vectors*. Partitioning the index space into regions is beneficial to the time scheduling. However, it is not sufficient for an efficient space scheduling, because it does not address the data locality issue. Therefore, an additional decomposition of the index space is necessary to enhance the data locality [MMS99][MSM04]. Usually, there are certain sequences of points computed by the same processor. Each such sequence can be viewed as a chain of computations, created by a certain dependence vector, as shown in Fig. 3.4. This dependence vector is called *communication vector*, denoted $\mathbf{d}_c$, and is usually chosen to be the dependence (cone) vector that incurs the largest amount of communication (in most cases $\mathbf{d}_c$ is the vector with the "smallest" absolute coordinates values). The goal here is to eliminate the communication cost incurred by $\mathbf{d}_c$. This means that the communication cost for executing the loop is hence attributed to the remaining k dependence vectors: $\mathbf{d}_1, \ldots, \mathbf{d}_k$. Hereafter, $k = p-1$, i.e., the number of dependence vectors, excluding $\mathbf{d}_c$.

The communication vector $\mathbf{d}_c$ defines the following family of lines in an n-dimensional

space: $\mathbf{j} = \mathbf{p}+\lambda\mathbf{d}_c$, where $\mathbf{p} \in \mathbb{N}^n$ and $\lambda \in \mathbb{R}$. This means that every single index point belongs to one of these lines. Therefore, by defining the chain $C_\mathbf{r}$ with *offset* $\mathbf{r}$ to be $\{\mathbf{j} \in J \mid \mathbf{j} = \mathbf{r}+\lambda\mathbf{d}_c$, for some $\lambda \in \mathbb{R}\}$, the index space J is partitioned into a set $\mathcal{C}$ of such chains. $|C_\mathbf{r}|$ and $|\mathcal{C}|$ are the cardinalities of $C_\mathbf{r}$ and $\mathcal{C}$, respectively, and C_M is the cardinality of a maximal chain of the set of chains $\mathcal{C}$. Generally, an index space may have multiple maximal chains. The points of $C_\mathbf{r}$ communicate via $\mathbf{d}_i$ (i designating every dependence vector except $\mathbf{d}_c$, i.e., $1 \leq i \leq k$) with the points of $C_{\mathbf{r}'}$.

Let $D^{in}_\mathbf{r}$ be the volume of the "incoming" data for $C_\mathbf{r}$, that is the number of index points on which the points of $C_\mathbf{r}$ depend on. Similarly, let $D^{out}_\mathbf{r}$ be the volume of the "outgoing" data for $C_\mathbf{r}$, that is the number of index points which depend on the points of $C_\mathbf{r}$. Then, the total communication associated with $C_\mathbf{r}$ is $D^{in}_\mathbf{r} + D^{out}_\mathbf{r}$. Hereafter, P denotes the number of available processors.

Adaptive Cyclic Scheduling – Description

ACS is designed to enhance the data locality of programs containing DOACROSS loops using the concept of chains, while taking advantage of the hyperplane scheduling method [DAK+02]. When mapping all points of a chain $C_\mathbf{r}$ to a single processor, the communication incurred by $\mathbf{d}_c$ is completely eliminated. Furthermore, assuming that $C_\mathbf{r}$ sends data to $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}$ due to dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_m$, by mapping $C_\mathbf{r}$ and $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}$ to the *same* processor, the communication incurred by $\mathbf{d}_1, \ldots, \mathbf{d}_m$ is also eliminated. Similarly, the chains from which $C_\mathbf{r}$ receives data, $C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_m}$, can also be mapped on the same processor, hence further decreasing the execution time of the parallel program. Usually, it is not be possible to map all these chains on the same processor; yet the important thing is the following: when assigning a new chain to the processor that executed $C_\mathbf{r}$, systematically pick one of $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_m}, C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_m}$ instead of choosing randomly or arbitrarily. This strategy is guaranteed to reduce the communication cost. Depending on the type of the target systems' architecture, the chains are assigned to processors as follows: on homogeneous systems – in a load balanced fashion, whereas on heterogeneous systems – they are assigned according to the available computational power of every processor. In both cases, an extended version of the master-slave model is used in which the master communicates with the slaves and the slaves communicate with each other whenever a data exchange is necessary. A coordinator processor (called *master*) computes the cones and the chains and then assigns them to the *slave* processors, who will execute them and communicate accordingly with other processors directly for sending or receiving the necessary data. The algorithm is outlined below:

ACS can be employed both on homogeneous and heterogeneous systems. In a homogeneous system, each slave is assigned the same number of chains and they are expected to finish together, assuming that each slave gets approximately the same number of iterations. In a heterogeneous system, the chains are assigned in a load balanced fashion; that is, slaves are assigned chains according to their delivered computational power (faster slaves get more chains), with the intent of having all slaves finish computing in the same time.

I. ACS on Homogeneous Systems

Consider a homogeneous system of 5 slaves and one master, and the example in Fig. 3.6. The dependence vectors are $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = (2, 2)$, $\mathbf{d}_3 = (4, 1)$ and the communication vector is $\mathbf{d}_c = \mathbf{d}_2$. The points of $C_{(0,0)}$ send data to the points of $C_{(0,0)} + \mathbf{d}_1 = C_{(0,2)}$, $C_{(0,0)} + \mathbf{d}_3 = C_{(3,0)}$. By mapping all these chains to the same slave, say S_1, one can see

Figure 3.5: ACS algorithm

```
procedure INPUT                                                        ▷ Master Input
    A DOACROSS nest with n loops, and terminal point U.
end procedure
```

```
 1: Master:                                                            ▷ Master code
 2: Determine the cone vectors.
 3: Compute the cones.
 4: Use QHull to find the optimal hyperplane.
 5: Choose the d_c.
 6: Form and count the chains.
 7: Compute the relative offsets between C_(0,0) and the k dependence vectors.
 8: Divide P so as to cover most successfully the relative offsets below as well as above d_c.
 9: if no dependence vector exists below (or above) d_c then
10:     choose the closest offset to P.
11: end if
12: Assign chains to slave processors in a cyclic fashion.
```

```
 1: Slave P_k:                                                         ▷ Slave code
 2: Send request for work to master.
 3: Wait for reply; when request served, store all chains and sort the points by the region
    they belong to.
 4: Compute points region by region, and along the optimal hyperplane. Communicate only
    when needed points are not locally computed.
```

```
procedure OUTPUT
    if no more point to assign then                                    ▷ Master Output
        notify all slaves, collect results and terminate
    end if
    if no more tasks come from master & notified by the master then   ▷ Slave Output
        terminate
    end if
end procedure
```

that the data locality is greatly enhanced. The relative offsets are: $r = 3$ (below $\mathbf{d}_2$) and $r = 2$ (above $\mathbf{d}_2$). However the system has 5 slaves and it must assign them chains in a cyclic fashion, so as to cover most successfully the relative offsets. In particular, the 5 slaves are divided as follows: slaves S_1,S_2, and S_3 are cyclicly assigned chains below $\mathbf{d}_c$, thus eliminating communication along $\mathbf{d}_3$, whereas the other 2 slaves, S_4 and S_5, are cyclicly assigned chains above $\mathbf{d}_c$, eliminating communication along $\mathbf{d}_1$ as well. This way all slaves are used in the most efficient way.

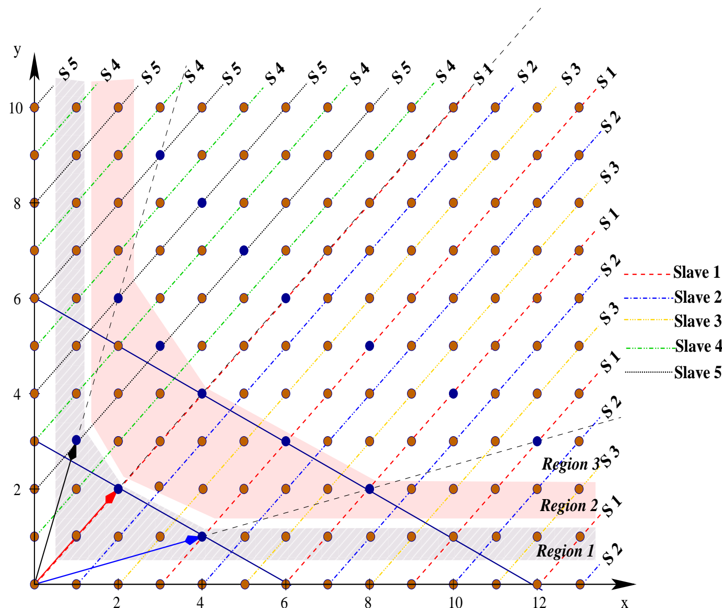


Figure 3.6: ACS chain assignment on a homogeneous system with 5 slaves

II. ACS on Heterogeneous Systems

When dealing with heterogeneous systems, one must take into account the available computing power and the actual load of the slave making the request to the master. It is assumed that every process running on the slave takes an equal share of its computing resources. The available virtual power (ACP_i) of a slave i is computed by dividing its virtual computing power (VCP_i) by the number of processes running in its queue (Q_i) [CABG01] [Kun91]. The total available computing power of the system (ACP) is the sum of the available (delivered) computing power of every slave. ACP_i is communicated to the master when slave i makes a request for work. In this case, the master will assign chains to slave i according to the ratio ACP/ACP_i. For instance, consider a heterogeneous system with 5 slaves and the example in Fig. 3.6. Suppose slave S_3 has less computational powers than the other slaves. Therefore it gets assigned only 4 chains, whereas the slaves S_1, S_2, S_4 and S_5 get assigned 5 chains each. Of course, this is an oversimplified example; in a larger heterogeneous system, and for bigger problems, the ACS algorithm would assign chains to slaves accurately reflecting the effect of the different ACP_i of every slave.

The adaptive cyclic scheduling described hereinbefore has some similarities with the static cyclic and static block scheduling methods described in [MA01]. The similarities are: the assignment of iterations to a processor is determined a priori and remains fixed – yielding reduced scheduling overhead; they require explicit synchronization between dependent chains

or tiles. However, the three scheduling methods are in the same time different, and the main differences are: iterations within a chain are independent, while within a tile they are not (this promotes the adaptability of ACS to different architectures, as mentioned above); ACS significantly enhances the data locality, hence is expected to outperform the static cyclic and block methods.

3.4.2 Chain Pattern Scheduling - CPS

This section describes a static scheduling algorithm similar to ACS, that also exploits the regularity of nested loops iterations spaces, named chain pattern scheduling (CPS) and published in [CAD+05]. The loops considered in this approach, however, have uniform flow dependencies and their loop body is complex, consisting of general program statements, such as assignments, conditions or repetitions. These loops are called *general nested loops*.

CPS is an extension of [DAK+] and [PAD01], in which a geometric method for the prediction of the pattern outline was presented for uniform dependence loops, but no space mapping method was presented. In [DAK+] and [PAD01] no communication cost was taken into account, and the unit-execution time (UET) model was assumed. In contrast, CPS accounts for the communication cost and a trade-off is given between different space-mapping schemes. CPS combines the method in [DAK+] with the static cyclic scheduling of [MA01], thus improving [DAK+] and [MA01] in that it enhances the data locality utilizing methods from computational geometry, to take advantage of the regularity of the index space. In particular, groups of iteration points, call *chains*, defining points connected via a specific dependence vector, called communication vector $\mathbf{d}_c$ (see [PAD01] and [CAP05]) have been identified. Specific chains are mapped to the same processor to enhance the data locality. It is a well-known fact that the communication overhead in most cases determines the quality and efficiency of the parallel code. The fundamental idea of CPS is that regardless of the underlying interconnection network (FastEthernet, GigabitEthernet, SCI, Myrinet), or of the number of processors within a node (for SMP systems), or of the system's homogeneity or heterogeneity, reducing the communication cost always yields enhanced performance. Given two scheduling policies that use the same number of processors, the one requiring less data exchange between the processors will almost certainly outperform the other.

Due to the existence of dependence vectors, only a certain set of iteration points of J can be executed at every time step [PAD01]. The geometric border of this set, forms a polygonal shape called *pattern*. CPS executes every index point at its *earliest computation time* (ECT), imposed by the existing dependence vectors. This policy guarantees the *optimal* execution time of the entire DOACROSS loop (assuming an unbounded number of processors). The index space is partitioned into disjoint *time subsets* denoted Pat_s, $r \geq 0$, such that Pat_s contains the set of points of J, whose earliest computation time is s. Fig. 3.7 illustrates the patterns, the pattern outline, the pattern points and the pattern vectors for a general UDL with the 4 dependence vectors $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = (2, 2)$, $\mathbf{d}_3 = (4, 1)$, and $\mathbf{d}_4 = (4, 3)$.

Each shaded polygon depicts a *pattern*, that can be executed at a certain time step. The first 3 patterns are showed. The dashed lines mark the border of each pattern, called *pattern outline*. Except $\mathbf{d}_4$, all other vectors contribute to the formation of the pattern outline, therefore are called *pattern vectors*. On the axes lie the pre-computed boundary points. The communication vector of the loop in Fig. 3.7 is $\mathbf{d}_c = \mathbf{d}_2$. The chain with origin $(0, 0)$, noted ($C_{(0,0)}$) is shown, as well as the chains it sends data to due to $\mathbf{d}_1$, $\mathbf{d}_4$ and $\mathbf{d}_3$, i.e., $C_{(0,2)}$, $C_{(1,0)}$ and $C_{(3,0)}$ respectively.

As with ACS, the goal of CPS is to eliminate the communication cost incurred by $\mathbf{d}_c$,

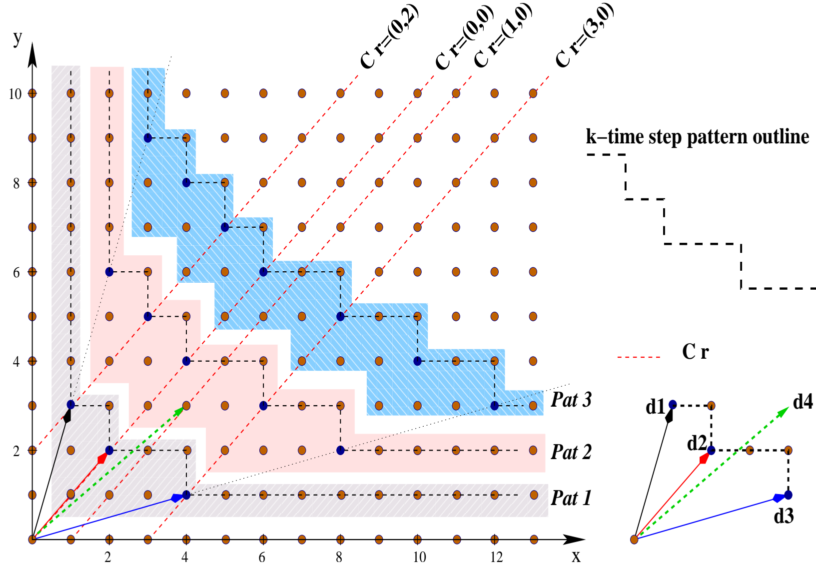


Figure 3.7: Patterns, pattern vectors and chains for a general UDL with 4 dependence vectors

and attributing the communication cost to the remaining k dependence vectors: $\mathbf{d}_1, \ldots, \mathbf{d}_k$, $k = p-1$. The communication vector $\mathbf{d}_c$ defines the following family of lines in n-dimensional space: $\mathbf{j} = \mathbf{p} + \lambda\mathbf{d}_c$, where $\mathbf{p} \in \mathbb{N}^n$ and $\lambda \in \mathbb{R}$. This way, every index point belongs to one such line. Thus, by defining the chain $C_{\mathbf{r}}$ with *offset* $\mathbf{r}$ to be $\{\mathbf{j} \in J \mid \mathbf{j} = \mathbf{r} + \lambda\mathbf{d}_c$, for some $\lambda \in \mathbb{R}\}$, we partition the index space J into a set $\mathcal{C}$ of such chains. The offset $\mathbf{r}$ is chosen so as to have *at least one* of its coordinates equal to 0, in other words $\mathbf{r}$ is a pre-computed point. $|C_{\mathbf{r}}|$ and $|\mathcal{C}|$ are the cardinalities of $C_{\mathbf{r}}$ and $\mathcal{C}$, respectively, and $|C_M|$ is the cardinality of a maximal chain of the set of chains $\mathcal{C}$. Generally, an index space may have multiple maximal chains. The points of $C_{\mathbf{r}}$ communicate via $\mathbf{d}_i$ (i designating any dependence vector except $\mathbf{d}_c$) with the points of $C_{\mathbf{r_i}'}$.

Let $D^{in}_{\mathbf{r}}$ be the volume of the "incoming" data for $C_{\mathbf{r}}$, i.e., the number of index points on which the points of $C_{\mathbf{r}}$ depend on. Similarly, let $D^{out}_{\mathbf{r}}$ is the volume of the "outgoing" data for $C_{\mathbf{r}}$, i.e., the number of index points which depend on the points of $C_{\mathbf{r}}$. Then, the total communication associated with $C_{\mathbf{r}}$ is $D^{in}_{\mathbf{r}} + D^{out}_{\mathbf{r}}$. Hereafter, P denotes the number of available processors.

Chain Pattern Scheduling – Description

The *chain pattern scheduling* algorithm is devised to enhance the data locality of programs with general nested loops using the concept of chains, and taking advantage of the optimality of the scheduling method based on patterns. By mapping all points of a chain $C_{\mathbf{r}}$ to a single processor, communication incurred by $\mathbf{d}_c$ for those points is completely eliminated. Furthermore, assuming that $C_{\mathbf{r}}$ sends data to $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_k}$ due to dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_k$, by mapping $C_{\mathbf{r}}$ and $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_k}$ to the *same* processor the communication incurred by $\mathbf{d}_1, \ldots, \mathbf{d}_k$ is also eliminated. Similarly, the chains from which $C_{\mathbf{r}}$ receives data, $C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_k}$, can also be mapped on the same processor, thus eliminating even more communication. Generally, it will not be possible to map all these chains on the same processor; yet the important thing is the following: when assigning a new chain to the processor that executed $C_{\mathbf{r}}$, systematically pick one of $C_{\mathbf{r}_1}, \ldots, C_{\mathbf{r}_k}, C_{\mathbf{r}'_1}, \ldots, C_{\mathbf{r}'_k}$ instead of choosing in a random or arbitrary manner. This strategy is guaranteed to lead to significant communication reduction.

To better illustrate our rationale, consider two different scenarios: (1) unbounded P – high communication $\Rightarrow$ moderate performance, and (2) fixed P – moderate communication $\Rightarrow$ good performance. A common feature of both scenarios is that chains are mapped to processors starting with chain $C_{(0,0)}$ and proceeding with the chains to its left (or above) and right (or below), like a fan spreading out. It is straightforward that by doing this more points become available for execution (are "released" from their dependencies) than in any other way.

Scenario 1: Unbounded P – High Communication

This is the case where there are enough available processors so that each chain is mapped to a different processor (see Fig. 3.8). Two similar examples are given: (a) A loop with three dependence vectors, $\mathbf{d}_1 = (1, 3)$, $\mathbf{d}_2 = \mathbf{d}_c = (2, 2)$ and $\mathbf{d}_3 = (4, 1)$ with chains formed along $\mathbf{d}_2$. (b) A loop with two dependence vectors, $\mathbf{d}_1 = (4, 1)$ and $\mathbf{d}_2 = \mathbf{d}_c = (2, 2)$, with chains also formed along $\mathbf{d}_2$. Note that in both cases, 24 chains are created. However this scenario is somewhat unrealistic because for larger index spaces the number of chains would be much greater and, therefore, the number of processors required for assigning one chain to one processor would be prohibitive.

On the other hand, this scenario does not support any kind of data locality (except for $\mathbf{d}_2$), requiring a large volume of communication between the processors. Basically, a chain $C_{\mathbf{r}}$ must receive data from k different chains and send data to k different chains. This implies that both $D^{in}_{\mathbf{r}}$ and $D^{out}_{\mathbf{r}}$ are bounded above by $k|C_M|$ (recall that $|C_M|$ is the cardinality of a maximal chain). The total volume of communication induced by this scheme, denoted $\mathcal{V}$, is then: $\mathcal{V} \approx 2k|C_M||\mathcal{C}|$. It is obvious that such a high volume of communication diminishes the overall performance of the parallel program.

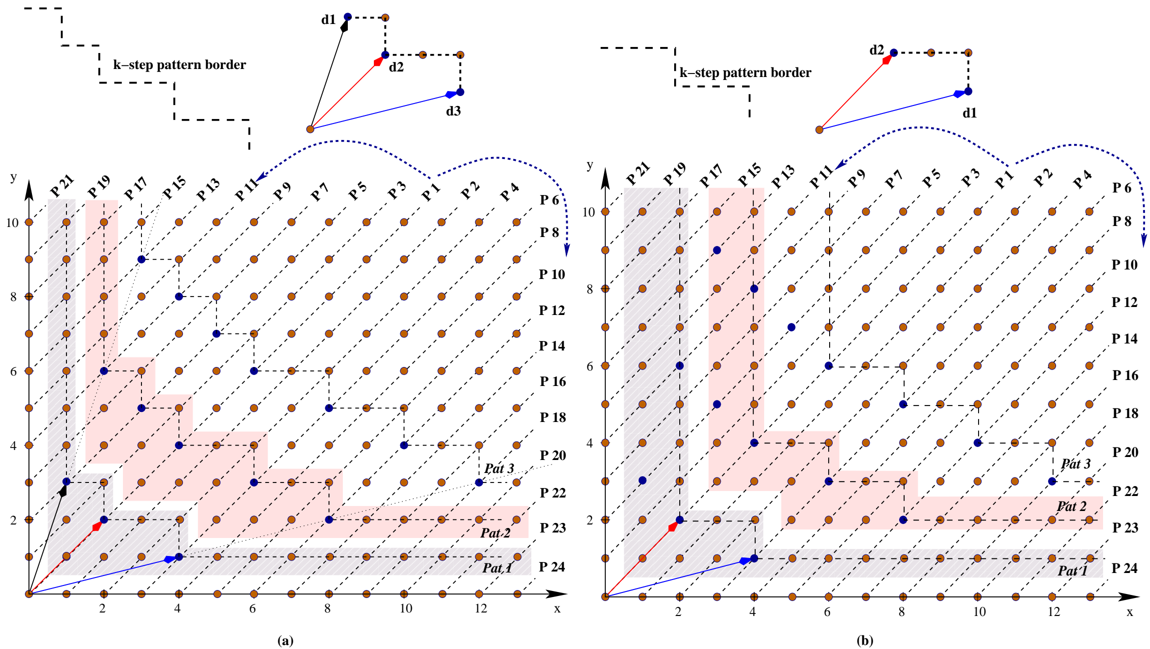


Figure 3.8: Scenario (1) Unbounded P – high communication $\Rightarrow$ moderate performance: every chain is mapped to a different processor (unrealistic assumption with respect to the number of available processors)

Scenario 2: Fixed P – Moderate Communication

This scenario is designed to minimize the communication and enhance the data locality, thus increasing the overall performance of the parallel program. For this scenario, two alternative mappings are considered, as shown in Fig. 3.9 and 3.10. In order to have moderate communication, and thus better performance than in scenario (1), an arbitrary P was considered (herein 5 processors). The first mapping (see Fig. 3.9(a)(b)) is an implementation of the well known cyclic mapping [MA01], where each chain from the pool of unassigned chains is mapped in a cyclic fashion, starting with $C_{(0,0)}$. This means, that the same processor will execute chains $C_{(0,0)}$, $C_{(5,0)}$, $C_{(10,0)}$ and so on. The communication volume for this mapping depends on P and on the chain-offsets $\mathbf{r}_1, \ldots, \mathbf{r}_k$ corresponding to dependence vectors $\mathbf{d}_1, \ldots, \mathbf{d}_k$. In particular, if P is *greater than the maximum coordinate* appearing in one of the offsets $\mathbf{r}_1, \ldots, \mathbf{r}_k$, then the volume of communication is prohibitive, basically being the same with the one in scenario (1), i.e., $\mathcal{V} \approx 2k|C_M||\mathcal{C}|$. Otherwise, if P is equal to a coordinate of an offset $\mathbf{r}_i$, the communication incurred by the corresponding dependence vector $\mathbf{d}_i$ is eliminated. Hence, let q be the number of offsets that have one of their coordinates equal to P; then the volume of communication is $\mathcal{V} \approx 2(k-q)|C_M||\mathcal{C}|$.

The second mapping, differs from the previous in that it intentionally zeroes the communication cost imposed by as many dependence vectors as possible. In particular, in Fig. 3.10(a) the first three processors execute the chains “below” $\mathbf{d}_c$ in a round-robin fashion, whereas the other two processors execute the chains above $\mathbf{d}_c$, again in a round-robin fashion. This way, the communication cost attributed to $\mathbf{d}_3$ is eliminated for the chains $C_{(0,0)}$ to $C_{(13,0)}$ and the communication cost attributed to $\mathbf{d}_1$ is eliminated for the chains $C_{(0,1)}$ to $C_{(0,10)}$. The difference from Fig. 3.9(a) is that in this case, the processors do not span the entire index space, but only a part of it (i.e., below or above $\mathbf{d}_c$). The benefits of doing so are that a more realistic P is assumed while still having acceptable communication needs, and performance is increased as a result. In Fig. 3.10(b) due to the fact that there is no dependence vector above $\mathbf{d}_c$, a *cyclic* mapping of chains is possible, starting with the chain $C_{(0,0)}$ and moving above and below, so as to incrementally release points for execution. This is similar to Fig. 3.9(b), with the difference that 5 processors were used instead of 3. The advantage of this scenario is that it does not limit the degree of parallelism because it uses *all the available* processors, hence being the most realistic of the two scenarios.

To estimate the volume of communication in this case we reason as follows: consider P as the sum of $q_1, \ldots, q_l$ (i.e., $P = q_1 + \ldots + q_l$ and, ideally, $l = k$) such that each q_i is a coordinate of offset $\mathbf{r}_i$. By proper assignment of groups of q_i chains to processors, the communication cost incurred by $\mathbf{d}_i$ is greatly diminished throughout the index space. Even if such a decomposition of P in groups of q_i processors is not possible, it is always possible to choose the appropriate integers $q_1, \ldots, q_l$ such that each q_i is a coordinate of $\mathbf{r}_i$ and $P > q_1 + \ldots + q_l$. Thus, by letting $q = q_1 + \ldots + q_l$, we conclude that the volume of communication is $\mathcal{V} \approx 2(k-q)C_M|\mathcal{C}|$.

For the sake of simplicity, in both scenarios every chain was assigned to *a single* processor. This is best suited for distributed-memory systems, that consist usually of single processor nodes. Note however, that each chain may be assigned to more than one processors such that no communication among them is required. This is more advantageous for multiprocessor nodes, where processors of a node can communicate through the locally (intra-node) shared-memory. Our approach is also suitable for heterogeneous networks, in which case a processor with higher computational power would be given either longer or more chains, whereas one with lesser computational power would be given either fewer or shorter chains. It is obvious that the ratio $\frac{\textit{communication time}}{\textit{processing time}}$ is critical for deciding which scenario and which architecture

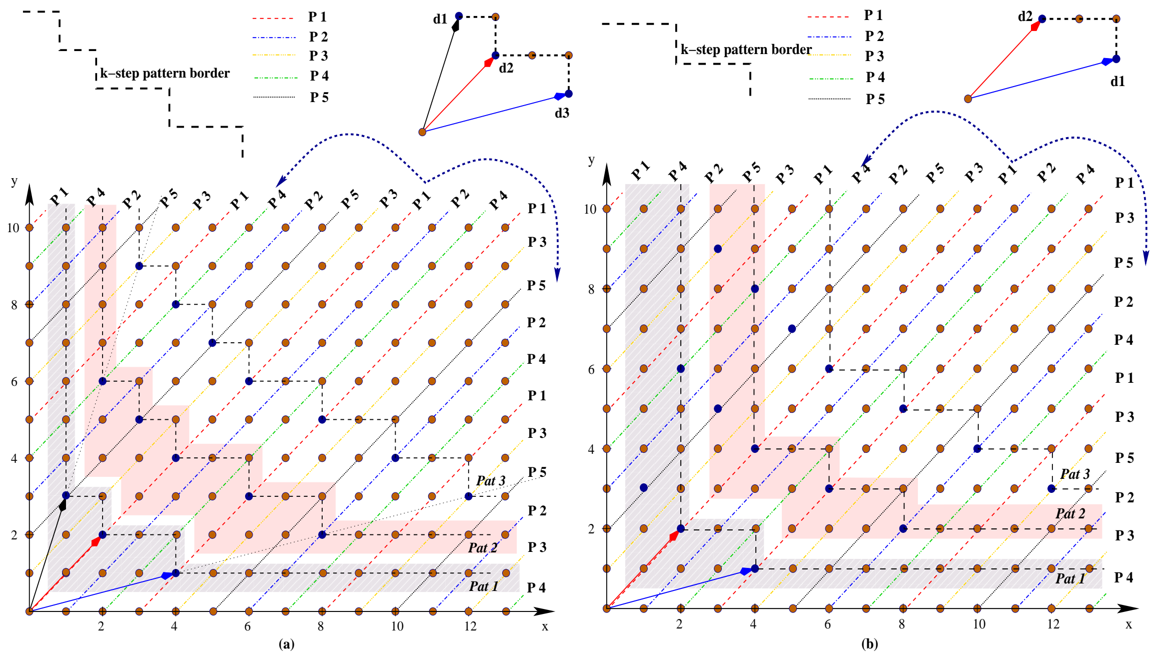


Figure 3.9: Scenario (2) Fixed P – Moderate Communication (cyclic mapping): chains are mapped to the available processors in a cyclic fashion starting with chain C_0

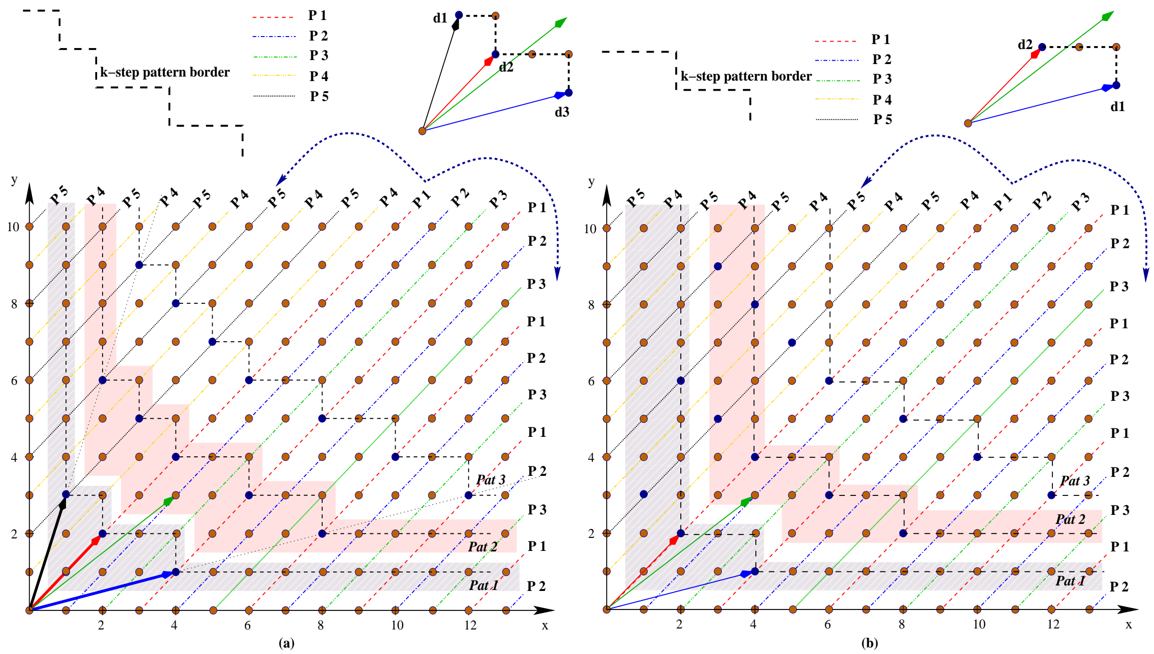


Figure 3.10: Scenario (2) Fixed P – Moderate Communication (mapping with offset 3 along the x axis, and offset 2 along the y axis): chains are mapped to the available processors so as to minimize communication imposed by $\mathbf{d}_c$, $\mathbf{d}_1$ and $\mathbf{d}_3$

suits best a specific application.

CHAPTER 4

Dynamic scheduling algorithms

4.1 What is Dynamic Scheduling?

In dynamic scheduling, only a few assumptions about the parallel program or the parallel system can be made before execution, and thus, scheduling decisions have to be made on-the-fly [AG91] [PLR+95]. The goal of a dynamic scheduling algorithm, as such, includes not only the minimization of the program completion time but also the minimization of the scheduling overhead which constitutes a significant portion of the cost paid for running the scheduler.

Dynamic scheduling is necessary in situations where static scheduling may result in a highly imbalanced distribution of work among processes or where the dependencies between the tasks are dynamic (e.g. due to changing system's behavior or changing application's behavior), thus precluding a static scheduling approach. It has been shown that load balancing is crucial to achieving adequate performance [ELJ86]. Dynamic scheduling algorithms [HBD] attempt to use the runtime state information of the system in order to make informative decisions for balancing the workload. Since the primary reason for using a dynamic scheduling is balancing the workload among processes, dynamic scheduling is often referred to as dynamic load balancing. Dynamic scheduling techniques are usually classified as either centralized or distributed.

In a *centralized* dynamic load balancing scheme, all executable tasks are maintained in a common 'pool' or they are maintained by a special processor or a subset of processors. If a special processor is designated to manage the pool of available tasks, then it is often referred to as the *master* and the other processors that depend on the master to obtain work are referred to as *slaves* (or workers). Whenever a slave processor has no work, it takes a portion of available work from the central data structure or the master process. Whenever a new task is generated, it is added to this centralized pool and/or reported to the master processor. Centralized load balancing schemes are usually easier to implement than distributed schemes, but may have limited scalability. As more and more processors are used, the large number of accesses to the pool of tasks or the master processor tends to become a bottleneck. The ultimate in automatic load balancing is a self-scheduling system that tries to keep all processing resources running at maximum efficiency. There may be a central location or authority (usually the master) to which processors refer for work and where they return their results. An idle processor requests that it be assigned new work by sending a message to the master and in return receives one or more tasks to perform. This works nicely for tasks with small contexts and/or relatively long running times. If the master is consulted too

often, or if it has to send (receive) large volumes of data to (from) processors, it can become a bottleneck.

In a *distributed* dynamic load balancing scheme, the set of executable tasks are distributed among processors which exchange tasks at run time to balance work. Each process can send work to or receive work from any other process. These methods do not suffer from the bottleneck associated with the centralized schemes. However, they are more difficult to implement.

The advantage of dynamic load balancing over static scheduling is that the system does not need to know the behavior of the application before execution (if the application in itself is dynamic and the system is static) and that the dynamic scheduler need not be aware of the system's behavior before execution (if the system's behavior changes and the application is static) or a combination of both. Dynamic load balancing is very useful in systems where the primary performance goal is maximizing processors utilization, rather than minimizing the runtime for individual jobs [SHK95]. This is often the case in networks of workstations.

This section describes several dynamic load balancing algorithms, devised to efficiently schedule DOACROSS loops on homogeneous and heterogeneous systems. The first two algorithms take a fine grain approach to parallelizing DOACROSS loops and focus on automatically generating the equivalent (but efficient) parallel code. The rest of the methods take a coarse grain approach and focus on load balancing DOACROSS loops on heterogeneous systems.

The notations used throughout this chapter are given below:

- UDL – uniform dependence loop, a loop nest with uniform precedence constraints
- GL – general UDL, nested loops of which the loop body consists of generic program statements (such as assignments, conditions and repetitions) and which exhibit uniform flow dependencies
- GRIDs – a special category of UDLs, with unitary dependence vectors that reside on the axes
- PE – a processing element
- $P_1, \ldots, P_m$ – the slave PEs; P_0 – the master PE
- J_{aux} – the auxiliary index space, obtained from the normal index space J by mapping groups of iterations from J to a point in J_{aux}
- SpectPat – the special pattern, containing s neighboring independent iteration points of J
- $\mathbf{L_{aux}}$ – the initial point of the auxiliary index space
- $\mathbf{U_{aux}}$ – the terminal point of the auxiliary index space
- *Convex Hull* ($\mathcal{CH}$) – the $\mathcal{CH}$ formed from the index points $\mathbf{j}_1, \ldots, \mathbf{j}_m$ is defined as: $\mathcal{CH} = \{\mathbf{j} \in \mathbb{N}^n \mid \mathbf{j} = \lambda_1 \mathbf{j}_1 + \ldots + \lambda_m \mathbf{j}_m,$ where $\lambda_1, \ldots, \lambda_m \geq 0$ and $\lambda_1 + \ldots + \lambda_m = 1\}$
- i - the current scheduling step
- N – the number of scheduling steps (total number of chunks), $i = 1, \ldots, N$

- u_c – the scheduling dimension and U_c the size of this dimension
- u_s – the synchronization dimension and U_s the size of this dimension
- C_i – the *chunk*; represents a few consecutive iterations of the loop; C_i is the chunk size at the i-th scheduling step
- V_i – the projection of C_i along scheduling dimension u_c
- $C_i = V_i \times \frac{\prod_{j=1}^{n} u_j}{u_c}$ – the size of a chunk
- M - the number of synchronization points inserted along u_s
- SP_j – the j-th synchronization point, $1 \leq j \leq M$
- h - the synchronization interval, i.e., the number of elements in the index space along u_s between two SPs; h is the same for every chunk
- *Synchronization frequency* – the number of synchronization points over the synchronization dimension
- *Current slave* – the slave assigned chunk C_i (master side)
- *Previous slave* – the slave assigned chunk C_{i-1} (master side)
- *Send-to slave*[1] – the slave id to which P_k must send computed data (slave side)
- *Receive-from slave* – the slave id from which P_k must receive computed data (slave side)
- VP_k – virtual computing power of slave P_k (relative speed of processor k when compared to the other $k-1$ processors)
- q_k – number of processes in the run-queue of slave P_k reflecting the total load of P_k
- $A_k = \lfloor \frac{VP_k}{q_k} \rfloor$ – available computing power of slave P_k (the delivered speed)
- $A = \sum_{m}^{i=1} A_k$ – total available computing power of the system
- $SC_{i,j}$ – the set of iterations of chunk i, between SP_{j-1} and SP_j
- $\mathcal{S}$ – the synchronization mechanism (completely independent of the scheduling algorithm and does not enhance the load balancing capability of the algorithm)
- $\mathcal{W}$ – the (chunk) weighting mechanism (completely independent of the scheduling algorithm and alone can be used for DOALL loops)
- p – the total number of pipelines (depends on the algorithm employed)
- t_c – the cost of communicating a message of size h between two slaves; $t_c = c_d + hc_c$
- c_d – the start-up cost (the time to send a zero-length message including the hardware/software overhead of sending the message)

[1]Master informs P_k about the *send-to*and *receive-from* ids based on the *current* and *previous* slaves ids

- c_c – the network throughput, defined as $\frac{1}{sustained\ bandwidth}$, where the *sustained bandwidth* is the ratio of the amount of data sent over the actual time measured at the application level
- t_p – the computation cost of a subchunk; $t_p = hV_ic_p$
- c_p – the computation time per iteration of the *slowest* worker

We make the following assumptions in this chapter:

A.1 The computation begins always at time step 1.

A.2 The processors are homogeneous and dedicated to executing the tasks (the first two algorithms).

A.3 The processors are heterogeneous and not necessarily dedicated to executing the tasks (all methods, except for the first two).

A.3 The communication network is homogeneous.

A.4 The loop body consists of general program statements and the dependence vectors are constant.

A.5 The execution and communication times are arbitrary.

A.6 The communication network is homogeneous and the cost of sending a message is equal to the cost of receiving a message.

A.7 The computation cost is a linear function of the computation cost per iteration times the number of iterations.

Note that assumption **A.4** does not hold for the first method presented below, because the loop body of the loops considered consists of simple assignment statements.

4.2 Automatic Parallel Code Generation

4.2.1 For *Scaled* GRIDs

A method for transforming sequential perfectly nested loops into their equivalent parallel form, is described next, published in [CAK+03]. The motivation for this method was devising a way to automatically generating the near optimal parallel code, which is equivalent to the original sequential code. The contribution of this new dynamic (fine grained) scheduling methodology is ensuring a high resource utilization and providing an automatic code generation tool.

This work is based on parallelizing the special category of uniform dependence loops (UDLs), called *Scaled* GRIDs and denoted SGRIDs. Every UDL in this category has the general structure of a GRID, i.e., all dependence vectors reside on the axes and the value of the non-zero coordinate is an arbitrary positive integer. GRIDs are a particular case of SGRIDs, where the dependence vectors are the unitary vectors along the axes.

The problem addressed below can be stated as follows: given a sequential program with a loop of SGRID UDLs structure, with a simple loop body, assuming that a fixed number of processors is available, produce its asymptotically optimal equivalent parallel program. The

method described here always produces the equivalent parallel code in polynomial time. If the targeted architecture is a shared memory system, the produced parallel code is the optimal time scheduling for the available number of processors. However, if the targeted architecture is a distributed memory system, the produced parallel code is an efficient scheduling for the specified number of processors, and the experimental results are very close to the ideal speedup.

Definition 4.2.1 (Scaled GRIDs) *The Scaled GRIDs are a subclass of the general class of UDLs in which the set DS is partitioned into two disjoint subsets: AXV and $DS - AXV$. The AXV (Axes Vectors) subset contains all the dependence vectors that lie along the axes, i.e., each dependence vector has the form $\mathbf{d} = \lambda \cdot \mathbf{e}_i$, where $\mathbf{e}_i$ is the unitary vector along dimension i and $\lambda \geq 1$ is a positive integer. Further, we require that for every i, $1 \leq i \leq n$, there exists at least one $\mathbf{d} = \lambda \cdot \mathbf{e}_i$.*

Definition 4.2.2 (SpecPat) *The special pattern (*SpecPat*) is a collection of s iteration points that are non-dependent on each other throughout the execution of the whole nest, obtained by considering the vectors with the smaller values from the AXV set, such that all other dependence vectors of $DS - AXV$ do not create any dependency between iteration points within the pattern.*

The well known GRIDs are a special case of SGRIDs in which $\lambda = 1$ for all the dependence vectors, and any resulting SpecPat contains only one iteration point, due to the unitary dependence vectors that characterize GRIDs.

Considering s to be the number of independent points within the pattern, replicating SpecPat throughout the whole iteration space, yields collections of s independent points, which obviously can be concurrently executed, provided there are available hardware resources. Therefore, the loop nest can be scheduled according to the available number of processors, that should preferably be a multiple of s. If the number of processors is not a multiple of s then, at some moment, several processors will be idle due to no available iterations eligible for execution. If the number of available resources is less than s (which is rarely the case) the results produced by our tool are not satisfactory. Therefore, hereafter, we assume that the number of available processors is a multiple of s.

Example 4.2.1 *Consider the 2D index space of an* S*GRID, representing the following DOACROSS loop:*

```
for (i=1; i<=10; i++)
    for (j=1; j<=10; j++) {
        A[i,j]=(A[i,j-2]*A[i-2,j])^20;
        B[i,j]=2*B[i,j-3]^10+B[i-4,j]^10-1;
    }
```

For this example, the initial point of the index space is $\mathbf{L} = (1, 1)$, the terminal point is $\mathbf{U} = (10, 10)$ and the cardinality of $|J| = 100$ points. The dependence vectors are $\mathbf{d}_1 = (0, 2)$, $\mathbf{d}_2 = (2, 0)$, $\mathbf{d}_3 = (0, 3)$ and $\mathbf{d}_4 = (4, 0)$. Due to the size of the smallest vectors on each axe, the SpecPat is a collection of 4 independent iteration points. The iteration index space with the replicated patten is shown in Figure 1(a), while the auxiliary space is depicted in Figure 1(b). Notice the initial point in the auxiliary space is $\mathbf{L_{aux}} = (0, 0)$ and contains

$(1,1)$ *(the initial iteration point)*, $(2,1)$, $(1,2)$ *and* $(2,2)$ *iteration points. The terminal point in the auxiliary space* $\mathbf{U_{aux}} = (4,4)$ *contains the iteration points:* $(9,9)$, $(9,10)$, $(10,9)$ *and* $(10,10)$ *(the terminal iteration point). The cardinality of the auxiliary space is* $|J_{aux}| = 25$ *patterns.* ◁

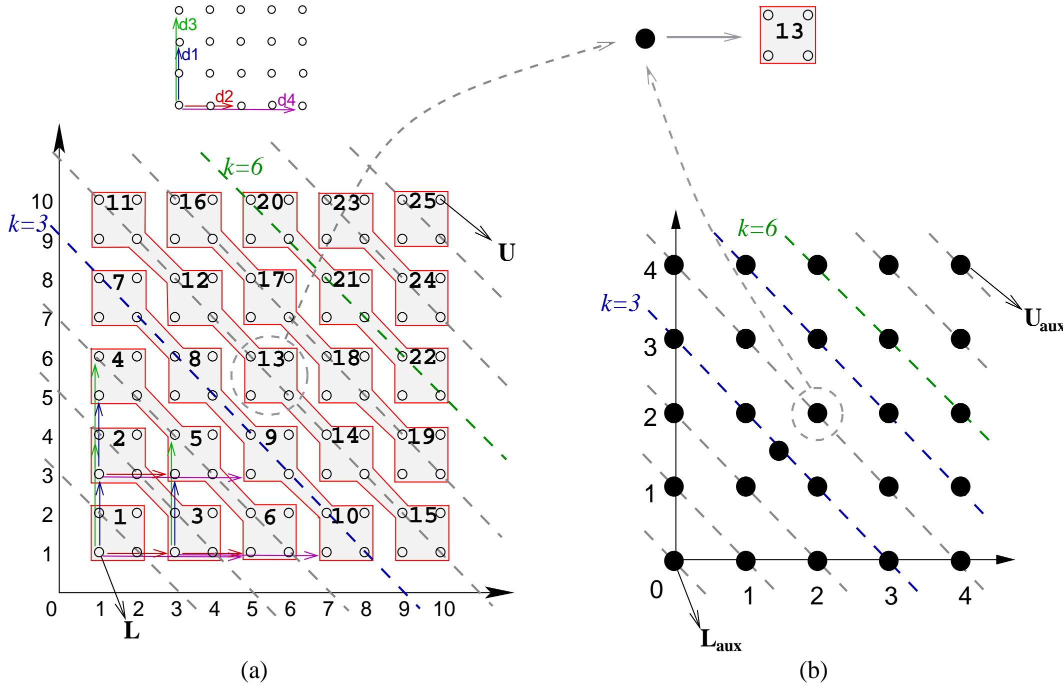


Figure 4.1: Geometrical representation of the (a) iteration index space J and (b) auxiliary index space J_{aux} of Example 4.2.1

The iteration index space J is partitioned into hyperplanes (or hypersurfaces) using the concept of *earliest computation time* [AKC⁺03b] [PAD01], that is, all the points that can be executed the earliest at time step k are said to belong to hyperplane k. As it can be seen in Figure 1, all the points that belong to hyperplane 3 in J, belong to the same hyperplane 3 in the auxiliary space J_{aux} too. Of course, this holds for every hyperplane of J.

I. Dynamic Scheduling Policy

We describe here the first of the dynamic scheduling methods, a distributed coarse grain approach scheme. This method schedules concurrently all the eligible iterations grouped in a pattern, because there is no need to exchange data between iteration points of the same pattern. Any iteration space replicated with the special pattern (SpecPat) is traversed pattern by pattern according to the lexicographic ordering of the resulting coordinates, as if each pattern were taken as a cartesian point in the *auxiliary index space* J_{aux}. Therefore, the patterns' execution order is the lexicographically traversal of the points on every hyperplane (zig-zag ordering) of the auxiliary space, as shown in Figure 1(b). In other words, all the patterns in the iteration space J are executed in a lexicographic order. At every moment, all the executable points are executed on the existing resources, in groups of s processors. The processors within each such group work concurrently, due to being assigned independent

iterations. It is obvious though, that two distinct groups of processors may exchange data, if there is any dependency between the iterations being executed by one group and the iterations executed by the other group (which correspond to dependencies between the two patterns).

Let us assume we have m available processors. They are arranged in groups of s processors. The execution flow goes as follows. Initially, the first group executes the first pattern (i.e., $\mathbf{L_{aux}}$), the second executes the second pattern, and so on, until the last group executes the m/s-th pattern. Once the initial pattern assignment is done, every subsequent pattern of each group is determined by skipping m/s positions from the currently assigned pattern. In other words, the first group gets the $\mathbf{L_{aux}} + m/s$-th pattern, the second group gets the $\mathbf{L_{aux}} + m/s + 1$-th pattern, and so on, until the last group gets the $\mathbf{L_{aux}} + 2 \times m/s$-th pattern. Notice that, when the processor groups are assigned patterns belonging to the same hyperplane, they can all compute concurrently. The scheduling proceeds this way, until reaching and executing the pattern containing the terminal point $\mathbf{U}$ of J.

II. Automatic Parallel Code Generation

The automatic code generation tool performs the following tasks:

Input: The sequential C code, P and the target system (shared-memory or distributed-memory)

Extracts: n, $|J|$, m, SpecPat, s

Performs: transforms J into J_{aux}; forms processor groups with ranks ranging $1, \ldots, \lfloor \frac{P}{m} \rfloor$

Outputs: The near-optimal equivalent parallel C+MPI code

Consider the loop nest of Example 4.2.1, written in a high level programming language (such as C). Since loops have a simple loop body, in the general case, a conventional parser can be used to extract the necessary information about the program. In this case, we assume that the parsing has been performed and the following information is extracted: depth of the loop nest $n = 2$, size of the iteration index space $|J| = 100$, number of dependence vectors $k = 4$, and their coordinates $\mathbf{d}_1 = (0, 2)$, $\mathbf{d}_2 = (2, 0)$, $\mathbf{d}_3 = (0, 3)$, $\mathbf{d}_4 = (4, 0)$.

Following the parsing and dependence analysis, the number of independent iterations that form a pattern is determined to be $s = 4$. Before the scheduling begins, the tool requires as input from the user the number of available hardware resources m on which the sequential code is to be parallelized. Assuming $m = 12$ available processors, they are arranged in three groups of four processors each. According to the scheduling policy, the first such group (G_1) is assigned the pattern of the first hyperplane (hyperplane 0), the second (G_2) is assigned the next lexicographic pattern (hyperplane 1) and the third (G_3) is assigned the next lexicographic pattern (also on hyperplane 1). After the initial assignment of patterns to groups, the next pattern for each group is the one found three (that is $m/p - 1$) positions further in the lexicographic ordering. In other words, the second pattern for G_1 is found by skipping three patterns from the initial pattern position (0,0) in J_{aux}, i.e., the first pattern on hyperplane 2, (0,2). The second pattern for G_2 is the pattern at (1,1), three position further from the initial pattern (0,1), and G_3 gets (2,0), three position further from the initial pattern (1,0). The *skipping method* for traversing the index space is used in order to avoid the re-execution of some pattern(s) (or iterations) and constitutes the main benefit of using multiple groups of processors.

We make the following observations: (1) All groups assigned auxiliary space points of the same hyperplane compute *concurrently*; (2) The scheduling policy is *decentralized*, i.e. there is no master processor.

The loop body of the sequential program is taken as a 'black box' (which contains all assignment statements) and is passed on to each processor to be executed for the appropriate loop indices. It is worth mentioning that the only way to distinguish processors in MPI is to reference them by their rank. A rank is a positive integer assigned to each processor in the system, which designates the *identification number* of the processor throughout the parallel execution. Therefore, in order to know from which processors to receive necessary data and/or to send computed data to, the available processors are assigned ranks as follows: processors from first group get ranks between 0 to 3, processors from second group get ranks between 4 to 7 and the ones of third group get ranks between 8 and 11. The parallel code is generated upon the completion of the above phases and is described next.

III. Parallel Code Generation

This section gives the parallel code that was generated by the tool for Example 4.2.1. The pseudocode for the equivalent parallel version of the program is given below:

```
forall (available processors)
   /* get coordinates of initial pattern and processor rank */
   current_pattern = get_initial_pattern(rank);
   while (current_pattern!=out_of_auxiliary_space)
   {
     /* we know the current pattern and the rank, find the */
     /* iteration point to be executed by each processor with rank */
     current_iteration_position=find_position(current_pattern, rank);
     /* receive data from all points the current point depends on */
     MPI_Recv(from_all_dependencies);
     /* execute the loop body associated to the current iteration */
     execute(Loop_Body, current_iteration_position);
     /* send data to all points that depend on current point */
     MPI_Send(to_all_dependencies);
     /* move on to the next pattern */
     current_pattern=Skip_x_patterns(current_pattern);
   }
endforall
```

The function `get_initial_pattern(rank)` is the following:

```
get_initial_pattern(rank){
    if (0 <= rank <=3) return L_aux
    else if (4 <= rank <=7) return Skip_1_pattern(L_aux)
    else if (8 <= rank <=11) return Skip_2_patterns(L_aux)
}
```

Herein, `L_aux` represents the initial point of the auxiliary space, i.e., $\mathbf{L_{aux}}$. This function returns the initial pattern coordinates for every of the three groups of $s = 4$ processors using

the skipping method and according to their rank values. The generated file containing MPI is ready to be executed onto the available processors.

The conclusions that can be drawn regarding dynamic scheduling for simple DOACROSS loops are:

- Making scheduling decisions at run time for DOACROSS loops is possible and efficient.
- Using blocking communication has the advantage of accuracy of the data exchanged.

4.2.2 For General Loops using Cronus

This section describes CRONUS, a platform for parallelizing general DOACROSS loops, published in [ACT$^+$04] and to appear in [ACT$^+$08]. CRONUS uses geometric methods, particularly the QuickHull algorithm (which can be freely downloaded from www.qhull.org) to determine the optimal hyperplane, by computing the convex hull of the end points of the dependence vectors (starting in origin) and the terminal point of the iteration space. For scheduling, CRONUS employs a distributed, fine-grain dynamic scheme, called *successive dynamic scheduling* (SDS). SDS sweeps the index space of DOACROSS loop along the optimal hyperplane, leading to the optimal makespan. CRONUS generates the parallel code so that each processor executes its apportioned computation and communication explicitly. Finally, the parallel code is run on the target machine.

The problem addressed by SDS and CRONUS is defined as follows: given a sequential general UDL (GL), produce the equivalent high performance parallel program using a bounded (but arbitrary) number of processors. The approach to the above problem consists of three steps:

Step 1 *(Preprocessing)* Determining the optimal family of hyperplanes using the QuickHull algorithm [BDH96]. Computing the *minimum* and *maximum* points of every hyperplane in order to follow the lexicographic ordering.

Step 2 Scheduling the iterations along the optimal hyperplane *on-the-fly* using the successive dynamic scheduling algorithm (SDS).

Step 3 Generating portable parallel code with CRONUS.

Note: A hyperplane is optimal if no other hyperplane leads to a smaller makespan.
Step 1 is performed at compilation, whereas steps 2 and 3 are carried out at runtime. Step 2 is performed at runtime because we can exploit the uniformity of the index space and make the most of the SDS adaptive rule.

Definition 4.2.3 (General loops) General loops *(GL) are nested DOACROSS loops that contain in their loop body generic program statements (such as assignments, conditions and repetitions) and which exhibit uniform flow dependencies.*

I. Finding the optimal scheduling hyperplane using convex hulls

The hyperplane (or wavefront) method [Lam74] was one of the first methods for parallelizing uniform dependence loops, and as such it formed the basis of most heuristics algorithms. A *hyperplane* is an n-dimensional plane that consists of a subset of *independent* iteration points ([Lam74], [Mol93, p.75]). These points can be executed in parallel, which results

in the significant reduction of the total makespan. Formally, a *hyperplane* $\Pi_k(a_1, \ldots, a_n)$, where $a_i, k \in \mathbb{N}$, consists of the index points $\mathbf{j} = (j_1, \ldots, j_n) \in J$ that satisfy the equation $a_1 j_1 + \ldots + a_n j_n = k$. The number of points of hyperplane Π_k is called the cardinality of Π_k and is denoted $|\Pi_k|$. For all index points $\mathbf{j} = (j_1, \ldots, j_n)$ lying on hyperplane $\Pi_k(a_1, \ldots, a_n)$ the sum $\sum_{r=1}^{n} a_r j_r$ has the same value k. In particular, the terminal point $\mathbf{U} = (u_s, \ldots, u_n)$ lies on hyperplane $k = a_1 u_s + \ldots + a_n u_n$.

Example 4.2.2 *Consider a 2D index space and a family of hyperplanes defined by the equation* $x_1 + x_2 = k$. *For* $k = 1$, *the index points lying on hyperplane* Π_1 *are* $(0,1)$ *and* $(1,0)$, *and the cardinality of this hyperplane is* $|\Pi_1| = 2$. *The index points lying on hyperplane* Π_2 *are* $(0,2),(1,1)$ *and* $(2,0)$, *and the cardinality of this hyperplane is* $|\Pi_2| = 3$. $\lhd$

As we have already mentioned, most methods for finding the optimal hyperplane use exponential time algorithms. In [PAD01] and [DAK+02] the problem of finding the optimal hyperplane for uniform dependence loops was reduced to the problem of computing the convex hull of the dependence vectors and the terminal point. The convex hull is a concept from computational geometry, defined as follows:

Definition 4.2.4 (Convex Hull) *The convex hull formed from the points* $\mathbf{j}_1, \ldots, \mathbf{j}_m$ *is defined as:* $\mathcal{CH} = \{\mathbf{j} \in \mathbb{N}^n \mid \mathbf{j} = \lambda_1 \mathbf{j}_1 + \ldots + \lambda_m \mathbf{j}_m,$ *where* $\lambda_1, \ldots, \lambda_m \geq 0$ *and* $\lambda_1 + \ldots + \lambda_m = 1\}$.

This means that algorithms from computational geometry that compute the convex hull can be used to find the optimal hyperplane. Typically, such algorithms produce as output a sequence of hyperplanes (in the context of computational geometry they are called “facets”) that define the convex hull. The hyperplanes defined exclusively by the endpoints of dependence vectors are the candidate optimal hyperplanes. Which one is the optimal depends on the **terminal point** of the loop. In [PAD01] and [DAK+02] it was shown that the optimal hyperplane is the one defined by the endpoints of the dependence vectors with the following property: the terminal point can be written as a linear combination of these dependence vectors with non-negative coefficients. One of the best algorithms for computing the convex hull is QuickHull [BDH96], which has lower complexity because it determines the hyperplanes for up to 3D index spaces in polynomial time as opposed to other methods (e.g. diophantine equations [MF86], linear programming in subspaces [SF91], and integer programming [DKR91]) which have above polynomial time complexity. This technique is explained in the following example:

Example 4.2.3 *In Fig. 4.2 the dependence vectors are* $\tilde{\mathbf{d}}_1 = (1,8)$, $\tilde{\mathbf{d}}_2 = (2,5)$, $\tilde{\mathbf{d}}_3 = (3,3)$, $\tilde{\mathbf{d}}_4 = (6,2)$ *and* $\tilde{\mathbf{d}}_5 = (8,1)$. *Consider the following two cases:*
(**1**) $N_1 = 75$, $N_2 = 90$, *resulting in the terminal point* $\mathbf{U}_1 = (75,90)$ *(Fig. 4.2(a))*
(**2**) $N_1' = 105$, $N_2' = 90$, *resulting in the terminal point* $\mathbf{U}_2 = (105,90)$ *(Fig. 4.2(b)).*

As shown in Fig. 4.2, in the first case the convex hull is the polygon $ABCEU_1$ *and in the second case the convex hull is the polygon* $ABCEU_2$. *The terminal point* $\mathbf{U}_1$ *belongs to the* cone *defined by the dependence vectors* $\tilde{\mathbf{d}}_2$ *and* $\tilde{\mathbf{d}}_3$ *(see Fig. 4.2(a)). This means that* $\mathbf{U}_1$ *can be written as* $\lambda \mathbf{d}_2 + \lambda' \mathbf{d}_3$ *with* λ *and* $\lambda' \geq 0$. *Similarly,* $\mathbf{U}_2$ *belongs to the* cone *defined by* $\tilde{\mathbf{d}}_3$ *and* $\tilde{\mathbf{d}}_5$ *(see Fig. 4.2(b)), i.e., it can be written as* $\rho \mathbf{d}_3 + \rho' \mathbf{d}_5$ *with* ρ *and* $\rho' \geq 0$. *The optimal hyperplane is defined by the endpoints of the dependence vectors that* form the cone containing the terminal point *([PAD01] and [DAK+02]). Thus, the optimal hyperplane in the first case is the line defined by the endpoints of* $(\tilde{\mathbf{d}}_2, \tilde{\mathbf{d}}_3)$, *whereas, in the second case, it is the line defined by the endpoints of* $(\tilde{\mathbf{d}}_3, \tilde{\mathbf{d}}_5)$. *Note that* $\tilde{\mathbf{d}}_4$ *is an interior point of the convex hull*

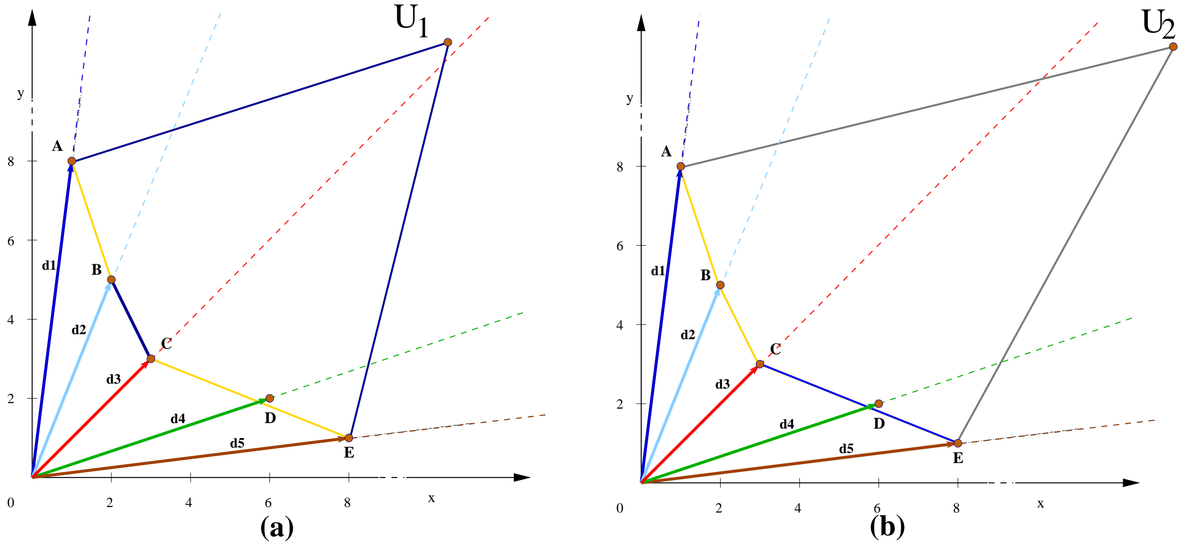


Figure 4.2: Optimal hyperplane for two different index spaces

and as such it plays no role in defining the optimal hyperplane. The equations of the lines passing through points $(2, 5)$ *and* $(3, 3)$ *and points points* $(3, 3)$ *and* $(8, 1)$ *are* $2x_1 + x_2 = 9$ *and* $2x_1 + 5x_2 = 21$*, respectively.* $\triangleleft$

II. Lexicographic ordering on hyperplanes

An important issue in dynamic scheduling is finding an adaptive rule for instructing every processor what to do at runtime rather than explicitly specifying at compile time. The adaptive rule determines the next-to-be-executed point and the required-already-executed point for any loop instance. To efficiently define such a rule, a total ordering (or *serialization*) of the index points is necessary. We shall define a serialization that facilitates the scheduling strategy we present.

Definition 4.2.5 (Lexicographic ordering) *Suppose that* $\mathbf{i} = (i_1, \ldots, i_n)$ *and* $\mathbf{j} = (j_1, \ldots, j_n)$ *are two index points. We say that* $\mathbf{i}$ *is less than* $\mathbf{j}$ *according to the* lexicographic *ordering and we write* $\mathbf{i} < \mathbf{j}$*, if* $i_1 = j_1 \wedge \ldots \wedge i_{r-1} = j_{r-1}$ *and* $i_r < j_r$ *for some* $r, 1 \leq r \leq n$*.*

In the rest of this paper, we use the lexicographic ordering among the index points, i.e., we write $\mathbf{i} < \mathbf{j}$ when $\mathbf{i}$ is lexicographically less than $\mathbf{j}$. The index space can be traversed along hyperplanes, yielding a zigzag traversal in 2D index spaces, or a spiral traversal in 3D (or higher) spaces. Given a specific hyperplane, the lexicographic ordering induces a *total* ordering of its points.

Definition 4.2.6 (Minimum of a hyperplane) *Index point* $\mathbf{j}$ *belonging to hyperplane* Π_k *is the* minimum *point of* Π_k*, if* Π_k *contains no other point* $\mathbf{i}$ *such that* $\mathbf{i} < \mathbf{j}$*. Similarly, point* $\mathbf{j}$ *of* Π_k *is the* maximum *point of* Π_k*, if* Π_k *contains no other point* $\mathbf{i}$ *such that* $\mathbf{j} < \mathbf{i}$*.*

Finding the minimum point of a given n-dimensional hyperplane is an optimization problem. Moreover, it exhibits the *optimal substructure* property, i.e., the optimal solution to the n-dimensional case contains within it the optimal solution to the $(n-1)$-dimensional

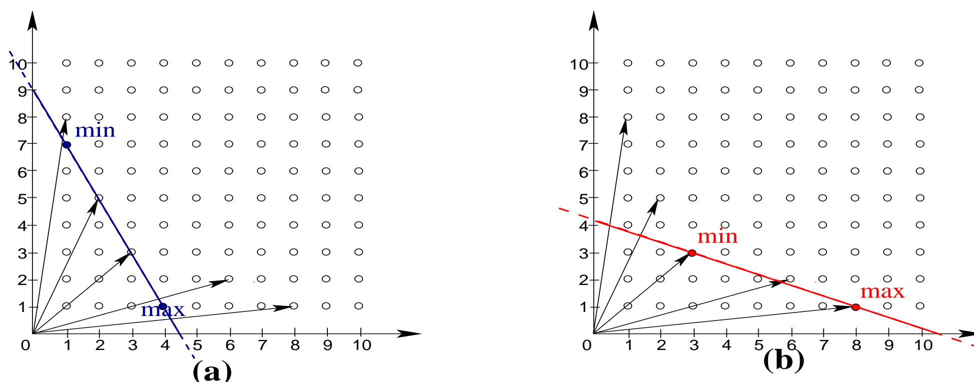


Figure 4.3: Minimum and maximum points on hyperplanes

case. Consequently, the minimum point can be computed by a simple dynamic programming algorithm. The pseudocode for this algorithm is given in the Appendix (Fig. A.1). In a symmetrical way, one can compute the maximum points of n-dimensional hyperplanes.

However, when dealing with 2D nested loops we can use mathematical techniques to speed up the computation of the minimum point of a 2D hyperplane. Before we explain what these techniques are, it is perhaps necessary to argue that although special, the 2D case is indeed very important for two reasons:
1. Most practical cases (most examples in the literature) fall into this category.
2. Even in higher dimensional cases, we may use, if necessary, loop transformation techniques (such as loop interchange, see [Mol93] for more details) in order to treat the higher dimensional loop as a 2D loop. This is not only possible (our platform Cronus can handle complex loop bodies) but it is also advantageous because it leads to heavier loop bodies and, thus enabling a more coarse grain parallelization approach.

Consider the equation $a_1x_1 + a_2x_2 = k$, where $a_1, a_2, x_1, x_2, k \in \mathbb{Z}$. A solution of this equation is an ordered couple (j_1, j_2), where $j_1, j_2 \in \mathbb{Z}$, such that $a_1j_1 + a_2j_2 = k$. Assuming that $g = gcd(a_1, a_2)$ and that not both a_1 and a_2 are equal to 0, then this equation has infinitely many solutions if and only if g divides k. It has no solution if g does not divide k (see [NZH91] and [Sil01]). Furthermore, if (j_1, j_2) is a solution, then all other solutions are of the form $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$, where $l \in \mathbb{Z}$. Finding the solutions, assuming of course that they exist, can be done easily and efficiently using the tabular algorithm presented in [NZH91], which is a modification of the Gaussian elimination method. We explain the details of this method in the Appendix. We introduce now two new concepts: the *successor* and the *predecessor* of an index point.

Definition 4.2.7 (Successor & Predecessor) *Suppose now that* $\mathbf{i}$ *and* $\mathbf{j}$ *are two index points of the* same *hyperplane* Π_k. *We say that* $\mathbf{j}$ *is the* successor *of* $\mathbf{i}$ *(equivalently,* $\mathbf{i}$ *is the* predecessor *of* $\mathbf{j}$*) according to the* lexicographic *ordering, if* $\mathbf{i} < \mathbf{j}$ *and for no other point* $\mathbf{j}'$ *of the same hyperplane does it hold that* $\mathbf{i} < \mathbf{j}' < \mathbf{j}$.

The maximum point of a hyperplane has no successor by definition (otherwise, it would not be the maximum point). The computation of all minima and maxima is done during Step 1 (preprocessing), i.e., when we compute the convex hull. It makes sense to perform these computations offline (at compilation time), because once the minima and maxima are computed they are stored in arrays, which are then transmitted to all slaves, so as to be used

in the process of finding the successor and predecessor at runtime.

Finding the successor of an index point is straightforward once the minimum points are known. The pseudocode of the algorithm for computing the successor in the general n-dimensional case is given in the Appendix (Fig. A.2). However, for the case of 2D loops, computing the successor of a point is trivial. Suppose that $\mathbf{i}$ and $\mathbf{j}$ belong to the same hyperplane $\Pi_k(a_1, a_2)$. In this case both $\mathbf{i}$ and $\mathbf{j}$ can be derived from the general form $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$ for different values of l. Assuming that l_i and l_j are the values corresponding to $\mathbf{i}$ and $\mathbf{j}$, we immediately deduce that $l_i < l_j \Leftrightarrow \mathbf{i} < \mathbf{j}$. Hence, $\mathbf{j}$ is the successor of $\mathbf{i}$ if and only if $l_j = l_i + 1$.

III. Successive Dynamic Scheduling (SDS)

Our scheduling scheme is based on the concept of the successor. Using the successor the index space can be traversed hyperplane by hyperplane and each hyperplane lexicographically, yielding a zig-zag/spiral traversal. This ensures its *validity* and its *optimality*. The former because the points of the current hyperplane depend only on points of previous hyperplanes, and the latter because by following the optimal hyperplane all inherent parallelism is exploited. We advocate the use of the successor function to iterate along the optimal hyperplane as an adaptive dynamic rule. The use of the successor concept is efficient because it induces a negligible overhead, and it does not incur any additional communication cost in distributed memory platforms. For this purpose we define the following function:

$$Succ(\mathbf{j}, r) = \begin{cases} \mathbf{j}, & \text{if } \; r = 0; \; /* \, the\ same\ point\ \mathbf{j} * / \\ \mathbf{successor}(\mathbf{j}), & \text{if } \; r = 1; \; /* \, the\ immediate\ successor\ of\ point\ \mathbf{j} * / \\ \mathbf{successor}(Succ(\mathbf{j}, r-1)), & \text{if } \; r > 1. \; /* \, the\ r-th\ successor\ of\ point\ \mathbf{j} * / \end{cases}$$

The above function provides: (1) the means by which each processor decides what iteration to execute next, and (2) an efficient algorithm enabling each processor to determine from which processor(s) to receive their computed data and to which processor(s) to send locally computed data.

The decentralized scheduling policy is the following: assuming there are NP available processors (P_1, ..., P_{NP}), P_1 executes the initial index point $\mathbf{L}$, P_2 executes $Succ(\mathbf{L}, 1)$, P_3 executes $Succ(\mathbf{L}, 2)$, and so on, until all processors are employed in execution for the first time. Upon completion of $\mathbf{L}$, P_1 executes the next executable point, found by skipping NP points on the hyperplane. The coordinates of this point are obtained by applying the $Succ$ function NP times to the point currently executed by P_1: $Succ(\mathbf{L}, NP)$. Similarly, upon completion of its current point $\mathbf{j}$, P_2 executes the point $Succ(\mathbf{j}, NP)$ and so on until exhausting all index points. SDS ends when the terminal point $\mathbf{U}$ has been executed.

The communication scheme employed by CRONUS is capable of identifying *on the fly* the processor(s) to receive data from and send data to. Every processor must receive data from other processors before commencing its computation and must, afterwards, send data to other processors.

To explain how this works in CRONUS, let us assume that processor P_i is assigned iteration $\mathbf{j}$. P_i must go through the following steps:

(1) Determine the iterations on which $\mathbf{j}$ depends on, according to the dependence vectors.

(2) Find which processors executed these iterations; in MPI this means finding their rank.

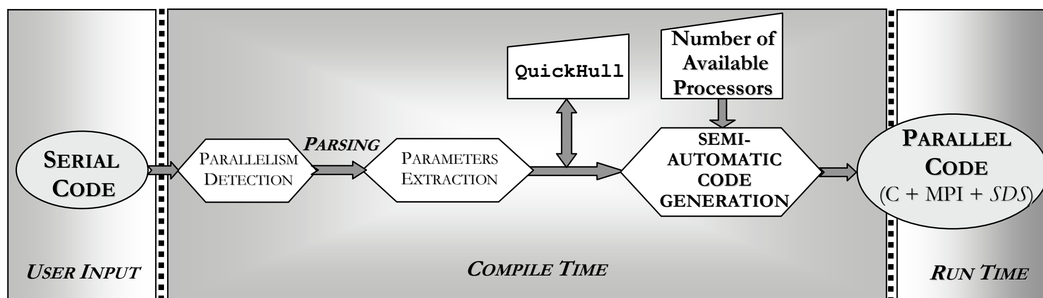


Figure 4.4: Organization of CRONUS

(3) Initiate appropriate MPI receive calls, using these ranks.

(4) Upon receiving data, compute iteration **j**.

(5) Upon computation, determine the iterations which depend on **j**.

(6) Find the ranks of the processors that are assigned these iterations[2].

(7) Initiate appropriate MPI send calls for these ranks.

(8) Determine the next executable iteration by calling the *Succ* function with parameters **j** and NP.

As mentioned earlier, employing a fine grain parallelization scheme implies that the loop body must be computationally intensive, otherwise the performance gain is canceled out. In particular, when such schemes are implemented on message passing architectures, the cost incurred by the communication routines should be less than the cost of computing the respective iteration. Therefore, generating MPI calls for each iteration is not prohibitive as long as computing the loop body surpasses this cost. The test cases presented in Chapter 5 have different loop bodies, leading to different granularities, i.e., different computational sizes of the iterations.

Another aspect of our approach is the decentralization of the scheduling overhead. This means that each processor is responsible for making the proper scheduling decisions. The decentralized scheduling approach is better suited for fine grain parallelism than the centralized one. This is because if the scheduling were left to a master processor that would have to communicate with multiple slave processors after every iteration, the resulting overhead would be excessive.

IV. Overview of CRONUS

CRONUS is an existing semi-automatic parallelization platform[3] (see Fig. 4.4). In the first stage (USER INPUT) the user inputs a serial program. The next stage (COMPILE TIME) consists of the loop nest detection phase (Parallelism Detection). If no loop nest can be

[2]In order to find the rank of the processor one needs to receive from or the rank of the processor one needs to send to, we use the simple successor computation technique of the previous Section when dealing with 2D cases, or the general successor algorithm, given in the Appendix for higher dimensional cases.

[3]A detailed description of the platform is contained in the “Cronus User’s Guide” that can be found at `www.cslab.ece.ntua.gr/∼cflorina/` .

found in the sequential program, CRONUS stops. Throughout the second phase (Parameters Extraction), the program is parsed and the following essential parameters are extracted: depth of the loop nest (n), size of the index space ($|J|$) and dependence vectors set (DS). Once the necessary parameters are available, the program calls the QuickHull algorithm (which we freely downloaded from www.qhull.org), and use it as an external routine, which takes as input the endpoints of the dependence vectors and the terminal point of the problem; finally it returns the coordinates of the optimal hyperplane. At this point, the available number of processors (NP) is required as input. A semi-automatic code generator produces the appropriate parallel code (in the Semi-automatic Code Generation phase) for the given NP. This is achieved with the help of a Perl script that operates on a configuration file, which contains all the required information.

In the configuration file the user must also define a startup function in C (automatically called by the parallel code) to perform data initialization on every processor. The parallel code is written in C and contains runtime routines for SDS and MPI primitives for data communication; it is eligible for compilation and execution on the multicomputer at hand (in the RUN TIME stage).

The conclusions that can be drawn regarding dynamic scheduling for general DOACROSS loops are:

- Making scheduling decisions at run time for DOACROSS loops is now possible and efficient.
- Using blocking communication has the advantage of accuracy of the data exchanged.

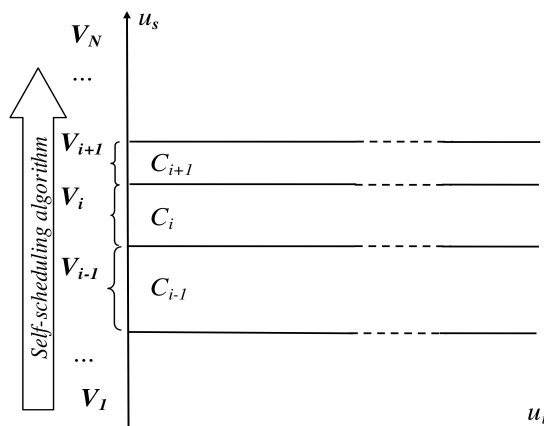
4.3 Self-Scheduling DOACROSS Loops

This section addresses the problem of scheduling DOACROSS loops in the more general case, that of heterogeneous distributed systems, with and without permanent extra load (dedicated and non-dedicated mode). We extended three well known dynamic schemes: Chunk self-scheduling (CSS), Trapezoid self-scheduling (TSS) and Distributed TSS (DTSS) by introducing synchronization points at certain intervals so that processors compute in wavefront fashion. Our scheme, published in [CAR+06] and extended in [PRA+06], is called dynamic multi-phase scheduling $DMPS(\mathcal{A})$, where $\mathcal{A}$ stands for one of the three algorithms, taken as input parameter to $DMPS$. Our scheme is stable with respect to internal parameters (such as chosen chunk size and synchronization interval) but sensitive to external parameters (such as cluster configuration and workload). $DMPS(\mathcal{A})$ is a distributed coarse-grain approach that is applied to general loops with uniform flow dependencies.

Most systems nowadays are heterogeneous and non-dedicated to specific users, yielding a system with variable workload. When static schemes are applied to heterogeneous systems with variable workload the performance is severely deteriorated. Dynamic scheduling algorithms adapt the assigned work to match the workload variation of both homogeneous and heterogeneous systems. An important class of dynamic scheduling algorithms are the self-scheduling schemes (such as CSS [KW85], GSS [PK87], TSS [TN93], Factoring [BL00], and others [MJ94]). On distributed systems these schemes are implemented using the classical master-slave model.

Since distributed systems are characterized by heterogeneity, to offer load balancing loop scheduling schemes must take into account the processing power of each computer in the system. The processing power depends on CPU speed, memory, cache structure and even the

Figure 4.5: Self-Scheduling algorithms for DOALL loops



program type. Furthermore, the processing power depends on the workload of the computer throughout the execution of the problem. Therefore, load balancing methods adapted to distributed environments take into account the relative powers of the computers. These relative computing powers are used as weights that scale the size of the sub-problem assigned to each processor. This significantly improves the total execution time when a non-dedicated heterogeneous computing environment is used. Such algorithms were presented in [HBFF00] and [HSUW96]. A recent algorithm that improves TSS by taking into account the processing powers of a non-dedicated heterogeneous system is DTSS (Distributed TSS) [CABG01]. To the best of our knowledge, all dynamic schemes proposed so far apply only to parallel loops (also called DOALL loops).

When DOALL loops are parallelized using dynamic schemes, the index space is partitioned into chunks, and the master processor assigns these chunks to slave processors upon request. Throughout the parallel execution, every slave works independently and upon executing the chunk sends the results back to the master. Obviously, this approach is not suitable for DOACROSS loops because, due to dependencies, iterations in one chunk depend on iterations in other chunks. Hence, slaves need to communicate. Inter-processor communication is the foremost important reason for performance deterioration when parallelizing DOACROSS loops. No study of dynamic algorithms for loops with dependencies on homogeneous or heterogeneous clusters has been reported so far.

In order to parallelize nested DOACROSS loops, after partitioning the index space into chunks (using one of the three schemes), we introduce synchronization points at certain intervals so that processors compute chunks in wavefront fashion. Synchronization points are carefully placed so that the volume of data exchange is reduced and the wavefront parallelism is improved.

4.3.1 Overview of Self-Scheduling Methods for DOALL Loops (CSS, TSS, DTSS)

Before explaining how the self-scheduling schemes are applied to DOACROSS loops, we must describe the methods extended in this section, as they were originally devised for DOALL loops. Recall that for DOALL loops the classical master-slave model is employed, where the communication takes place only between the master and slave(s).

Self-scheduling algorithms work by partitioning the iteration space into chunks along one of its dimensions, called *chunk* dimension and denoted u_c. This creates a pool of tasks which are then dynamically assigned to available workers upon request. The simplest self-scheduling algorithm, called *pure self-scheduling* (PSS) assigns one iteration to each worker per request. To reduce the synchronization overhead of PSS, the *chunk self-scheduling* (CSS) algorithm assigns a fixed size chunk of iterations per request.

Chunk Self-Scheduling (CSS) [KW85] assigns constant size chunks to each slave, i.e., $C_i = constant$. The chunk size is chosen by the user. If $C_i = 1$ then CSS is the so-called (pure) Self-Scheduling. A large chunk size reduces scheduling overhead, but also increases the chance of load imbalance, due to the difficulty to predict an optimal chunk size. As a compromise between load imbalance and scheduling overhead, other schemes start with large chunk sizes in order to reduce the scheduling overhead and reduce the chunk sizes throughout the execution to improve load balancing. These schemes are known as reducing chunk size algorithms and their difference lies in the choice of the first chunk and the computation of the decrement.

The Trapezoid Self-Scheduling (TSS) [TN93] scheme linearly decreases the chunk size C_i. The first and last (assigned) chunk size pair (F, L) may be set by the programmer. A conservative selection for the (F, L) pair is: $F = \frac{|U_c|}{2*m}$ and $L = 1$, where m is the number of slaves. This ensures that the load of the first chunk is less than $1/m$ of the total load in most loop distributions and reduces the chance of imbalance due to a large first chunk. Still many synchronizations may occur. One can improve this by choosing $L > 1$. The proposed number of steps needed for the scheduling process is $N = \frac{2\times|U_c|}{(F+L)}$. Thus, the decrement between consecutive chunks is $D = (F - L)/(N - 1)$, and the chunk sizes are $C_1 = F, C_2 = F - D, C_3 = F - 2 \times D, \ldots, C_N = F - (N - 1) \times D$.

Distributed TSS (DTSS) [CABG01] improves on TSS by selecting the chunk sizes according to the computational power of the slaves. DTSS uses a model that includes the number of processes in the run-queue of each slave. Every process running on a slave is assumed to take an equal share of its computing resources. The application programmer may determine the pair (F, L) according to TSS, or the following formula may be used in the conservative selection approach: $F = \frac{|U_c|}{2\times A}$ and $L = 1$ (assuming that the loop which is scheduled by DTSS is the r-th loop). The total number of steps is $N = \frac{2\times|U_c|}{(F+L)}$ and the chunk decrement is $D = (F-L)/(N-1)$. The size of a chunk in this case is $C_i = A_k \times (F - D \times (S_{k-1} + (A_k - 1)/2))$, where: $S_{k-1} = A_1 + \ldots + A_{k-1}$. When all slaves are dedicated to a single user job then $A_k = VP_k$. Also, when all slaves have the same speed, then $VP_k = 1$ and the tasks assigned in DTSS are the same as in TSS. The important difference between DTSS and TSS is that in DTSS the next chunk is allocated according to the slave's available computing power. Hence, faster slaves get more loop iterations than slower ones. In contrast, TSS simply treats all slaves in the same way.

Table 4.2 shows the chunk sizes computed with CSS, TSS and DTSS for a DOALL loop with an index space of 5000×10000 iterations and and $m = 10$ slaves. Since, CSS and TSS were devised for homogeneous system, they are insensitive to the system's load. Hence, CSS and TSS obtain the same chunk sizes in the dedicated cluster as in the non-dedicated cluster. On the contrary, DTSS improves on TSS for heterogeneous systems, because it adapts the chunk size to match the different computational powers of slaves. As mentioned before, all these algorithms have been evaluated for DOALL loops [CABG01] and it has been established that the DTSS algorithm improves on the TSS, which in turn outperforms CSS.

Table 4.1: Sample chunk sizes given for $|J| = 5000 \times 10000$ and $m = 10$. Chunks are formed along $u_c = 5000$

Algorithm	Chunk sizes
CSS	300 300 300 300 300 300 300 300 300 300 300 300 300 300 300 300 200
TSS	277 270 263 256 249 242 235 228 221 214 207 200 193 186 179 172 165 158 151 144 137 130 123 116 109 102 73
DTSS (dedicated)	392 253 368 237 344 221 108 211 103 300 192 276 176 176 252 160 77 149 72 207 130 183 114 159 98 46 87 41 44
DTSS (non-dedicated)	263 383 369 355 229 112 219 107 209 203 293 279 265 169 33 96 46 89 86 83 80 77 74 24 69 66 31 59 56 53 50 47 44 20 39 20 33 30 27 24 21 20 20 20 20 20 20 20 20 8

4.3.2 Motivation for Dynamic Scheduling of DOACROSS Loops

Existing dynamic scheduling algorithms cannot cope with DOACROSS loops. Consider, for instance, the heat equation, and its pseudocode below:

```
/* Heat equation */ for (l=1; l<loop; l++) {
    for (i=1; i<width; i++){
        for (j=1; j<height; j++){
                A[i][j] = 1/4*(A[i-1][j] + A[i][j-1]
                       + A'[i+1][j] + A'[i][j+1]);
        }
    }
}
```

When dynamic schemes are applied to parallelize this problem, the index space is partitioned into chunks, that are consequently assigned to slaves. These slaves then work independently. But due to the presence of dependencies, the slaves *have to* communicate with each other. However, existing dynamic schemes do not support inter-slave communication, only master-to-slave(s) communication. Therefore, in order to apply dynamic schemes to DOACROSS loops, one must provide an inter-slave communication scheme, such that problem's dependencies are not violated or ignored.

4.3.3 The Inter-Slave Communication Scheme

In this work we bring the existing dynamic scheduling schemes into the field of scheduling DOACROSS loops. We propose an inter-slave communication scheme for three well known dynamic schemes: CSS [KW85], TSS [TN93] and DTSS [CABG01]. In all cases, after the master assigns chunks to slaves, the slaves communicate with each other by means of synchronization points. This provides the slaves with a unified communication scheme. This is

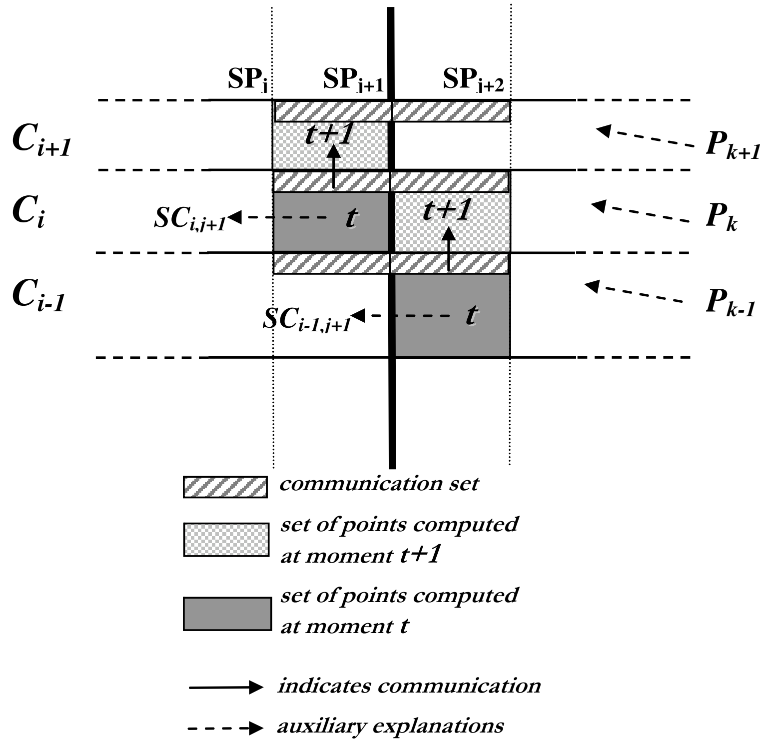


Figure 4.6: Synchronization points

depicted in Fig. 4.6 and 4.11, where chunks $i-1, i, i+1$ are assigned to slaves P_{k-1}, P_k, P_{k+1}, respectively. The shaded areas denote sets of iterations that are computed concurrently by different PEs. When P_k reaches the synchronization point SP_{j+1} (i.e. after computing $SC_{i,j+1}$) it sends P_{k+1} only the data P_{k+1} requires to begin execution of $SC_{i+1,j+1}$. The data sent to P_{k+1} designates only those iterations of $SC_{i,j+1}$ imposed by the dependence vectors, on which the iterations of $SC_{i+1,j+1}$ depend on. Similarly, P_k receives from P_{k-1} the data P_k requires to proceed with the execution of $SC_{i,j+2}$. Note that slaves do not reach a synchronization point at the same time. For instance, P_k reaches SP_{j+1} earlier than P_{k+1} and later than P_{k-1}. The existence of synchronization points leads to pipelined execution, as shown in Fig. 4.6 by the shaded areas.

4.3.4 Dynamic Multi-Phase Scheduling

For the sake of simplicity we consider a 2D DOACROSS loop with $\mathbf{U} = (u_s, u_c)$, and $u_s \geq u_c$. The index space of this loop is partitioned into chunks along the scheduling dimension u_c (using one of the self-scheduling algorithms). Synchronization points are inserted along the synchronization dimension u_s, at equal intervals. The interval length, h, is carefully chosen by the programmer and yields the number of synchronization points, $M = \frac{U_s}{h}$.

The following notation is essential for the inter-slave communication scheme: the master *always* names the slave assigned with the latest chunk (C_i) as *current* and the slave assigned with the chunk C_{i-1} as *previous*. Whenever a new chunk is computed and assigned, the *current* slave becomes the *(new) previous* slave, whereas the new slave is named *(new) current*. Fig. 4.8 below shows the state diagram related to the *(new) current* – *(new) previous* slaves.

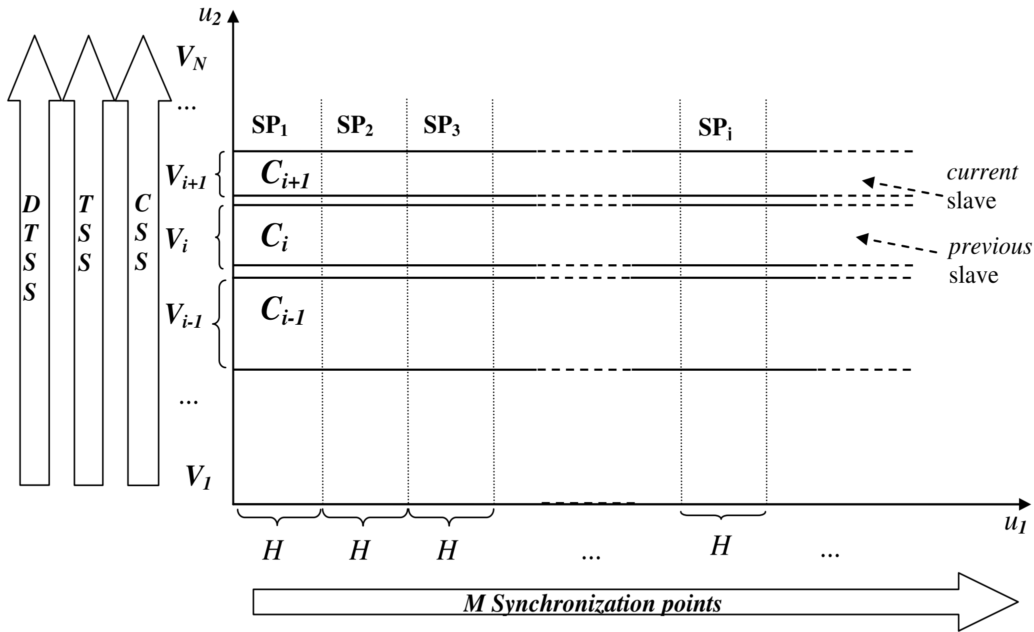


Figure 4.7: Chunks are formed along u_c and SP are introduced along u_s

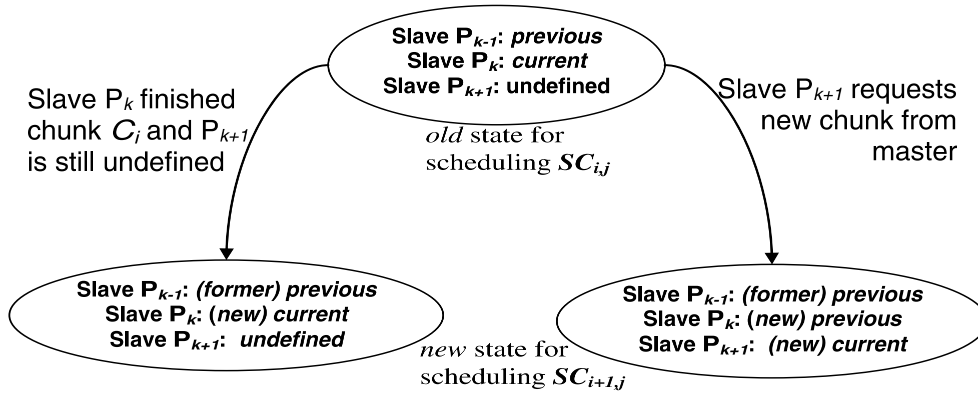


Figure 4.8: State diagram of the slaves

The state transitions are triggered by new requests for chunks to the master.

The Dynamic Multi-Phase Scheduling scheme consists of the following two phases:

Phase 1 Apply self-scheduling algorithms to the scheduling dimension.

Phase 2 Insert synchronization points along the synchronization dimension.

The Dynamic Multi-Phase Scheduling scheme $DMPS(\mathcal{A})$, which employs the extended master-slave model (i.e. with inter-slave communication) is described next.

Remark: (1) Note that the synchronization intervals are the same for all chunks. For remarks (2)–(5) below refer to Fig. 4.8 for an illustration. (2) Upon completion of $SC_{i,0}$, slave P_k requests from the master the identity of the *send-to* slave. If no reply is received, then P_k

Figure 4.9: $DMPS(\mathcal{A})$ Scheme

```
procedure INPUT                                                      ▷ Master Input
    An n-dimensional DOACROSS loop, with terminal point U
    The choice of algorithm A ∈ [CSS, TSS, DTSS]
    if A =CSS then
        chunk size is C_i
    end if
    The synchronization interval h
    The number of slaves m
    if A =DTSS then
        also the virtual power V_k of every slave
    end if
end procedure
```

```
 1: Master:                                                          ▷ Master code
 2: procedure INITIALIZATION:
 3:     Register slaves
 4:     if A =DTSS then
 5:         slaves report their A_k
 6:     end if
 7:     Calculate F, L, N, D for TSS and DTSS; for CSS use the given C_i
 8: end procedure
 9: while there are unassigned iterations do
10:     if a request arrives then
11:         put it in the queue
12:     end if
13:     Pick a request from the queue, and compute the next chunk size using CSS, TSS or
    DTSS
14:     Update the current and previous slave ids
15:     Send the id of the current slave to the previous one
16: end while
```

Figure 4.10: $DMPS(\mathcal{A})$ Scheme (*continued*)

```
1:  Slave P_k:                                                      ▷ Slave code
2:  procedure INITIALIZATION:
3:      Register with the master
4:      if A =DTSS then
5:          report A_k
6:      end if
7:      Compute M according to the given h
8:  end procedure
9:  Send request to the master
10: Wait for reply.
11: if received chunk from master then
12:     go to step S.3
13: else
14:     goto OUTPUT
15: end if
16: while the next SP is not reached do
17:     compute chunk i
18: end while
19: if id of the send-to slave is known then
20:     goto step S.5
21: else
22:     goto step S.6
23: end if
24: Send computed data to send-to slave
25: Receive data from the receive-from slave and goto step S.3
```

```
procedure OUTPUT
    if no more chunks to be assigned then                       ▷ Master Output
        terminate
    end if
    if no more tasks come from master then                      ▷ Slave Output
        terminate
    end if
end procedure
```

is *still* the current slave, and it proceeds to receive data from the previous slave P_{k-1}, and then it begins $SC_{i,1}$. (3) Slave P_k keeps requesting the identity of the *send-to* slave, at the end of every $SC_{i,j}$ until either a *(new) current* slave has been appointed by the master or P_k has finished chunk i. (4) If slave P_k has already executed $SC_{i,0}, \ldots, SC_{i,j}$ by the time it is informed by the master about the identity of the *send-to* slave, it sends all computed data from $SC_{i,0}, \ldots, S_{i,j}$. (5) If no *send-to* slave has been appointed by the time slave P_k finishes chunk i, then all computed data is kept in the local memory of slave P_k. Then P_k makes a new request to the master to become the *(new) current* slave.

The advantages of the proposed Dynamic Multi-Phase Scheduling scheme $DMPS(\mathcal{A})$ are outline below:

✓ Can take as input *any* self-scheduling algorithm, without requiring any modifications

✓ **Phase 2** is independent of **Phase 1**

✓ **Phase 1** deals with the heterogeneity and load variation in the system

✓ **Phase 2** deals with minimizing the inter-slave communication cost

✓ Suitable for any type of heterogeneous systems

The conclusion regarding the proposed dynamic schemes is that DOACROSS loops can *now* be dynamically scheduled on heterogeneous dedicated and non-dedicated systems. However, there are still a few things missing. And they are:

- A *generic add-on* to other self-scheduling algorithms, such that they can all handle DOACROSS loops and account for system's heterogeneity without any modifications

- A *model* for predicting the optimal synchronization interval h and minimizing the communication

4.4 Enhancing Self-Scheduling Algorithms via Synchronization and Weighting

In this section we introduce two mechanisms that were devised to enable other self-scheduling algorithms to handle general DOACROSS loops and account for system's heterogeneity without any modifications, published in [CRA+08].

To compensate for the fact that existing self-scheduling algorithm cannot handle DOACROSS loops (or tasks with dependencies), we introduce here a *synchronization mechanism* that provides the inter-processor communication, thus, enabling self-scheduling algorithms to handle efficiently nested DOACROSS loops. We also present in what follows a *weighting mechanism* that significantly improves the performance of dynamic self-scheduling algorithms. These algorithms divide the total number of tasks into chunks and assign them to processors. The weighting mechanism adapts the chunk sizes to the computing power and current run-queue state of the processors. The synchronization and weighting mechanisms are orthogonal, in the sense that they can simultaneously be applied to DOACROSS loops. Hence, they broaden the applications' spectrum of dynamic self-scheduling algorithms and improve their performance. We expect that the synchronization and weighting mechanisms to be quite efficient for DOACROSS loops on heterogeneous systems, and this is verified in chapter 5.

This section extends and generalizes the work in the previous section by constructing a general *synchronization* mechanism $\mathcal{S}$, which applies to all loop self-scheduling schemes. When this mechanism is applied to a self-scheduling algorithm, it enables it to handle efficiently DOACROSS loops. The synchronization mechanism $\mathcal{S}$ inserts synchronization points in the execution flow so that slaves perform the appropriate data exchanges. This mechanism is not incorporated within the self-scheduling algorithm, but it is an additional stand-alone component, applicable without further modifications. Given a self-scheduling algorithm $\mathcal{A}$, its synchronized version is denoted $\mathcal{S}$-$\mathcal{A}$.

In addition, motivated by the results in [SHW96] [BL00] [CABG01], we define a *weighting* mechanism $\mathcal{W}$, aimed at improving the load balancing and, thus, the performance of all non-adaptive self-scheduling algorithms on non-dedicated heterogeneous systems. This mechanism is inspired from the approach used in [CABG01], i.e., it uses the relative powers of the slaves combined with information regarding their run-queues to compute chunks. However, in contrast to previous approaches to chunk weighting, this mechanism is not embedded within the self-scheduling algorithm, but it is an external stand-alone component applicable to any dynamic algorithm without modifications. Given any self-scheduling algorithm $\mathcal{A}$, its weighted version will be called $\mathcal{W}$-$\mathcal{A}$.

4.4.1 Overview of Self-Scheduling Methods for DOALL Loops (GSS, FSS)

Before we describe the two mechanisms, we will describe the self-scheduling algorithms that we considered for testing in chapter 5. In section 4.3.1 we described the CSS, TSS and DTSS self-scheduling schemes. In this section, we will describe two other self-scheduling schemes: Guided Self-Scheduling (GSS) and Factoring Self-Scheduling (FSS), as they were originally devised for DOALL loops. Recall that for DOALL loops the classical master-slave model is employed, where the communication takes place only between the master and slave(s).

In the Guided Self-Scheduling (GSS) [PK87] scheme, each slave is assigned a chunk given by the number of remaining iterations divided by the number of slaves, i.e., $C_i = R_i/m$, where R_i is the number of remaining iterations. Assuming that the loop which is scheduled with GSS is the r-th loop, then R_0 is the total number of iterations, i.e., $|U_r|$, and $R_{i+1} = R_i - C_i$, where $C_i = \lceil R_i/m \rceil = \lceil (1 - \frac{1}{m})^i \cdot \frac{|U_r|}{m} \rceil$. This scheme initially assigns large chunks, which implies reduced communication/scheduling overheads in the beginning. At the last steps small chunks are assigned to improve the load balancing, at the expense of increased communication/scheduling overhead.

The Factoring Self-Scheduling (FSS) [BL00] scheme schedules iterations in batches of m equal chunks. In each batch, a slave is assigned a chunk size given by a subset of the remaining iterations (usually half) divided by the number of slaves. The chunk size in this case is $C_i = \lceil \frac{R_i}{\alpha * m} \rceil$ and $R_{i+1} = R_i - (m \times C_i)$, where the parameter α is computed (by a probability distribution) or is sub-optimally chosen $\alpha = 2$. The weakness of this scheme is the difficulty to determine the optimal parameters. However, tests show [BL00] improvement on previous adaptive schemes (possibly) due to fewer adaptations of the chunk-size.

4.4.2 The Synchronization Mechanism – $\mathcal{S}$

The purpose of the synchronization mechanism is to enable self-scheduling algorithms to handle general DOACROSS loops. In all cases the master assigns chunks to slaves, which synchronize with each other at every synchronization point SP. We must emphasize that the synchronization mechanism is completely *independent* of the self-scheduling algorithm and

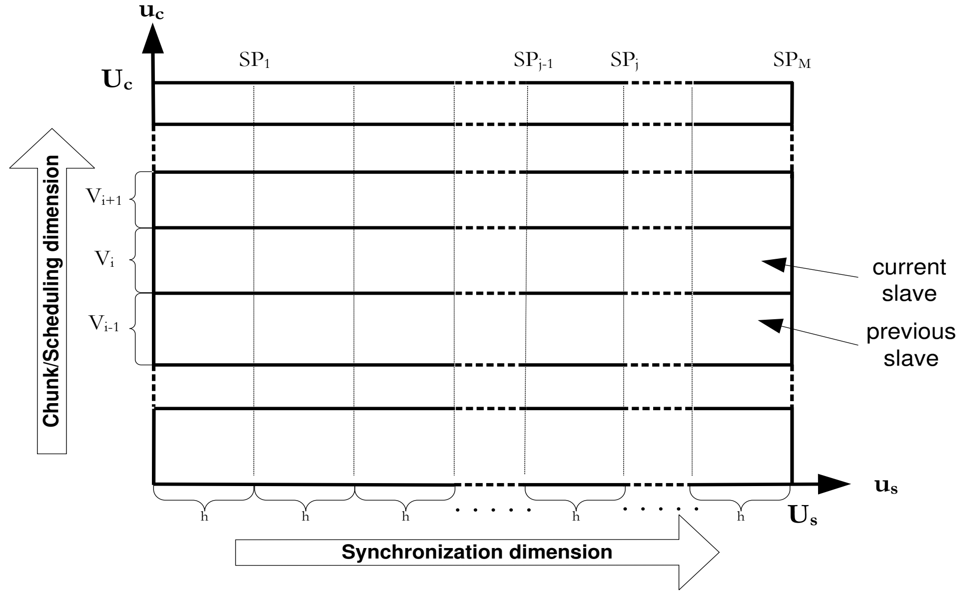


Figure 4.11: Partitioning of a 2D DOACROSS loop into chunks, and placement of synchronization points

does not enhance the load balancing capability of the algorithm. Therefore, synchronized self-scheduling algorithms perform well on heterogeneous systems only if the self-scheduling algorithm itself explicitly takes into account the heterogeneity. The synchronization overhead is compensated by the increase of parallelism resulting in an overall performance improvement.

The synchronization mechanism $\mathcal{S}$ provides the synchronization interval along u_s:

$$h = \lceil \frac{U_s}{M} \rceil \tag{4.1}$$

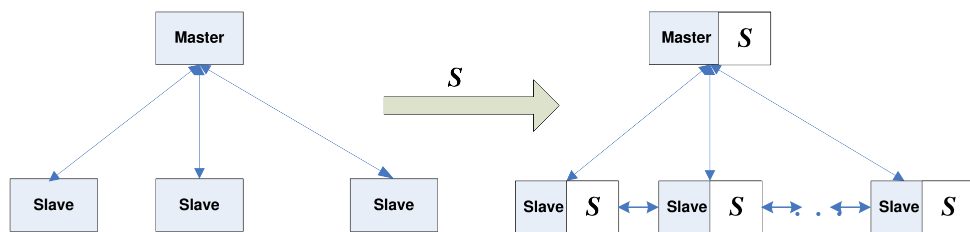
and the framework for inter-slave communication. Figure 4.11 illustrates V_i and h. The horizontal strip sections are assigned to single slaves. Synchronization points are placed in the u_s dimension so that other slaves can start computing as soon as possible. Note that in this example, C_i is the number of loop iterations in the horizontal strip, i.e., $C_i = V_i \times U_s$.

The synchronization mechanism $\mathcal{S}$ extends the classical master slave model, by enabling slaves to exchange data directly with each other, and communicate with the master only to request and receive work. To accomplish this, the synchronization mechanism $\mathcal{S}$ adds three components to the original algorithm $\mathcal{A}$ (see Fig. 4.12):

(1) **transaction accounting** (master side) - according to slaves' requests, the master extracts information and decides upon the identity of the slaves participating in a data exchange (*previous* and *current* slaves)

(2) **receive part** (slave side) - uses the information from (1) to receive the corresponding communication sets

(3) **transmit part** (slave side) - uses the information from (1) to send the corresponding communication sets.



(a) Classic master-slave model

(b) Master-slave model with the stand alone synchronization component and inter-slaves communication links

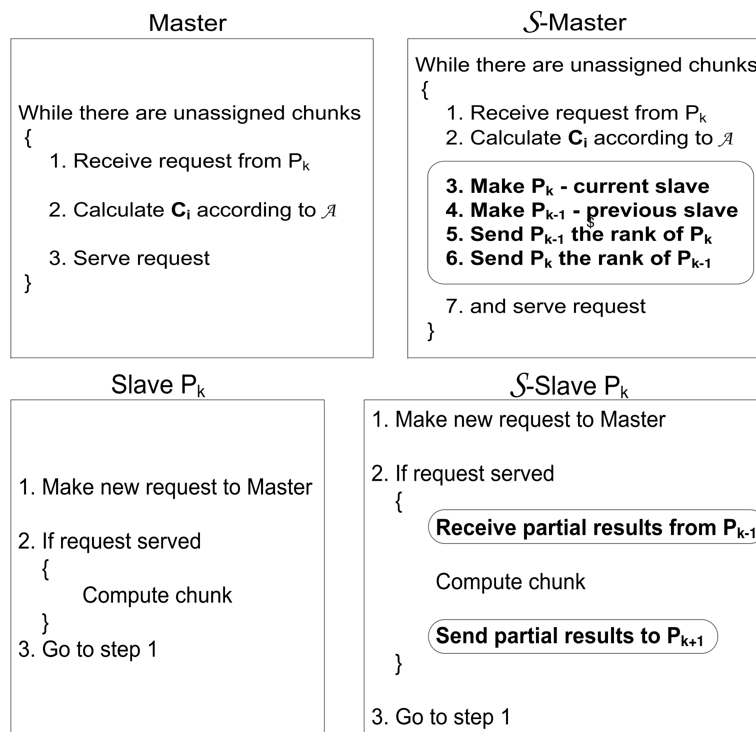


Figure 4.12: Master-slave model with synchronization mechanism

Details regarding components (2) and (3) are given in the pseudocode algorithm 4.4.2 below. We must specify that the synchronization interval h is determined empirically or selected by the user, and must be a tradeoff between synchronization overhead and parallelism.

The flexibility of the synchronization mechanism is twofold:

(1) P_k checks for the identity of the *send-to* slave at every synchronization point. Suppose that P_k first learns the identity of the *send-to* slave P_{k+1} at synchronization point SP_j. In this case, P_k sends to P_{k+1} all the locally stored data that P_{k+1} requires, in one step (sending one large message is more efficient than sending the same amount of data in smaller consecutive messages). Hence, P_{k+1} is not delayed at any synchronization point up to SP_j. Moreover, the fact that P_k is j synchronization points ahead of P_{k+1}, where $2 \leq j \leq M$, means that P_{k+1} receives from P_k the data it needs since they have already been computed.

(2) In the extreme case where no *send-to* is designated by the time P_k reaches the last synchronization point, P_k stores all the data it should have sent to the *send-to* slave in a local buffer. When the master designates the *send-to* slave P_{k+1}, then P_k sends to P_{k+1} all stored data. Note that in this case P_k could also be the *send-to* slave.

For the implementation of the synchronization mechanism, the following issues must be addressed: (a) the placement of synchronization points along dimension u_s and (b) the transmission of data to the adjacent slave. The synchronization points are computed as follows: $SP_1 = h$ and $SP_{i+1} = SP_i + h$. The synchronization interval h depends on M, which is determined empirically, as demonstrated in the experiments of Chapter 5, or selected by the user. The pseudocode of Algorithm 4.4.2 provides the basis for two C-code blocks that handle the transmission part. The code blocks are inserted in the slave code, in the positions indicated in Fig. 4.12. Before "Compute chunk" we insert the block responsible for handling the reception part of the communication, and after "Compute chunk" we insert the block that handles the send part. We have assumed that the outer dimension of the loop is the synchronization dimension. This implies that the receive and send code blocks are inserted between the loop iterating over the synchronization dimension and before the loop iterating over the chunk dimension. Whenever the index of synchronization dimension is iterated h times, these blocks are activated in order for the data exchange event to occur. On the master side, a code block is inserted so as whenever a new slave makes a request, it is registered as the *current slave*, and the last registered slave is renamed as the *previous slave*. This information is then transmitted to the last two registered slaves. A preprocessor could be implemented for automatic parsing, detecting and inserting the appropriate code blocks. In our experiments we did this manually. In either case the initial code does not have to be rewritten. This is also applicable to the implementation of the $\mathcal{W}$ and the $\mathcal{SW}$ mechanisms, as will be described next.

4.4.3 Empirical Determination of M

In order to understand the impact on performance of the choice of M, we give in Fig. 4.14 the parallel execution on $m = 4$ slaves of a hypothetical example. For simplicity, assume that each slave is assigned only two chunks and that all chunks are of the same size. The slave request order, i.e., P_1, P_2, P_3, P_4, and the synchronization points, $SP_1, \ldots, SP_{12}$, inserted by the synchronization mechanism, are also shown. The numbers in each subchunk $SC_{i,j}$ indicate the time step in which it is executed. The flow of execution follows a wavefront

Figure 4.13: Pseudocode of the communication scheme implementation on the slave side

```
1: procedure Receive part                                ▷ Receive partial results from P_{k-1}
2:     in synchronization point SP_i check for partial results from previous slave P_{k-1}
3:     if current and previous slaves are the same, that is, P_k = P_{k-1} then
4:         all partial results exist in local memory
5:         proceed to the computation without blocking in any SP for the completion of
   current chunk
6:     else
7:         receive partial results from P_{k-1}
8:         check the number of communication sets b of partial results received
9:         if b > 1 then
10:            skip the next b SPs
11:        end if
12:    end if
13:    proceed to computation
14: end procedure
```

```
1: procedure Transmit part                                ▷ Send partial results to P_{k+1}
2:     in synchronization point SP_i
3:     if SP reached is SP_1 then
4:         make a non-blocking request to master for the rank of the next slave P_{k+1}
5:         if the rank of slave P_{k+1} is not yet known to master then
6:             store partial results in local memory
7:             proceed to receive part
8:         else
9:             get the rank of P_{k+1}
10:            send it partial results
11:        end if
12:    else
13:        if the rank of P_{k+1} is already known by P_k then
14:            send P_{k+1} the partial results
15:        else
16:            if a reply has been received by the master for the rank of the next slave P_{k+1}
   then
17:                send P_{k+1} all previous partial results in a single packet
18:                proceed to receive part
19:            else
20:                store partial results in local memory
21:                proceed to receive part
22:            end if
23:        end if
24:    end if
25: end procedure
```

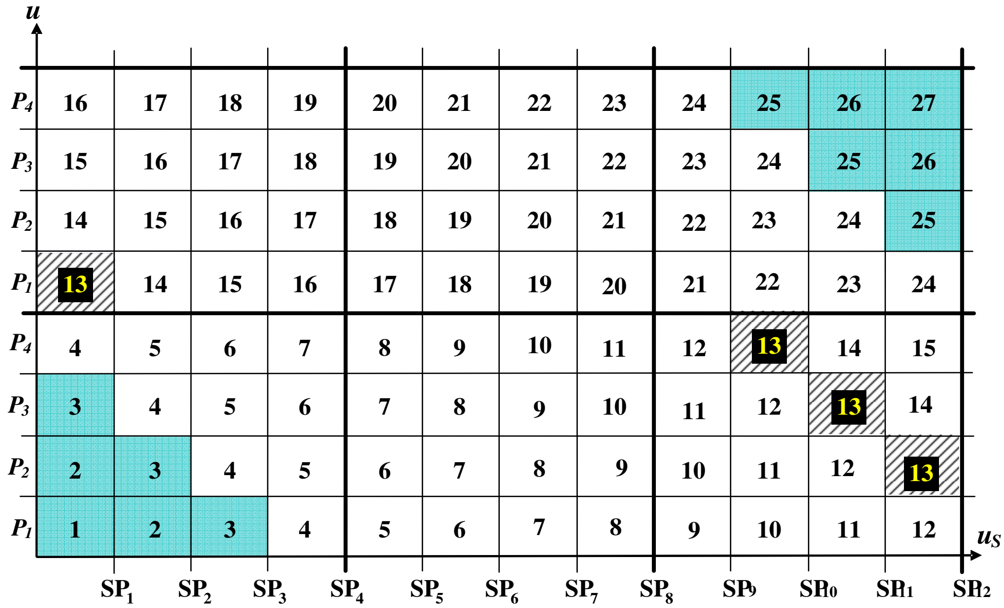


Figure 4.14: Parallel execution in a pipelined fashion

fashion. From time steps 1 to 3 (*initial time steps*) and 25 to 27 (*final time steps*), denoted by the gray areas, one can see that not all slaves are active, whereas from 4 to 24 (*intermediate time steps*) all slaves are active. At the end of the 12-th time step, slave P_1 has completed its first chunk and it is then assigned a second chunk by the master. It begins computing the second chunk at time step 13, since it has all the necessary data from slave P_4. In the same time step, slaves P_2, P_3, P_4 are still computing their first chunk. As soon as slave P_2 completes its first chunk (at the end of time step 13), it proceeds with its next chunk at time step 14. The transition to the second chunk for P_3 and P_4 takes place at time steps 14 and 15, respectively. In other words, except for the initial and the final time steps, the execution proceeds with no delays, apart from the synchronization between slaves, as imposed by the synchronization points. The *#initial steps* is $(m-1)$ and is equal to the *#final steps*, while *#intermediate steps* = *#total steps* - $2(m-1)$, as illustrated in Fig. 4.14. The number of total steps depends on the number of SPs and the number of chunks produced by the scheduling algorithm. Since the number of chunks is algorithm dependent, the choice of the number of SPs should maximize the percentage of intermediate time steps over the total time steps.

The selection of the number of SPs is a tradeoff between synchronization overhead and parallelism. A choice of a large M incurs too frequent data exchanges and a high synchronization overhead, whereas a small M restricts the inherent parallelism. We believe that the optimal selection of the synchronization interval depends on many factors, such as: the dependencies of the loop, the characteristics of the underlying communication network and of the processors, and the self-scheduling algorithm used. Extensive experimental runs for various test cases and self-scheduling algorithms (contained in Section 6) show that a good, albeit arbitrary, choice is $M \geq 3 * m$, where m is the number of slaves. In our example (Fig. 4.14), $M = 12$, which yields that *#total steps* = 27, *#intermediate steps* = 21, and a percentage of 77% of execution without idle times.

4.4.4 The Weighting Mechanism – $\mathcal{W}$

The purpose of the weighting mechanism $\mathcal{W}$ is to enable the self-scheduling algorithms to handle load variations and system heterogeneity. $\mathcal{W}$ adjusts the amount of work (chunk size) given by the original algorithm $\mathcal{A}$ according to the current load of a processor and its nominal computational power. It must be mentioned that $\mathcal{W}$ is a completely independent mechanism of the self-scheduling algorithm and alone (without $\mathcal{S}$ that is) can be used for DOALL loops. The weighting overhead is insignificant (a $\star$ and a / operation).

Let us consider that the self-scheduling algorithm $\mathcal{A}$ is used to parallelize a nested loop on a heterogenous system with m slaves, each with $VP_1, \dots, VP_m$ virtual computational power. Furthermore, let us assume that during the i-th scheduling step, slave P_k has q_k processes in its run-queue. The weighting mechanism $\mathcal{W}$, calculates the chunk $\widehat{C_i}$ assigned to P_k, using the following formula:

$$\widehat{C_i} = C_i \times \frac{VP_k}{q_k} \qquad (4.2)$$

In the above formula VP_k and q_k are the virtual power and number of processes in the run-queue of slave P_k, respectively, and C_i is the chunk size given by the original self-scheduling algorithm $\mathcal{A}$. Hence, $\widehat{C_i}$ is the "weighted" chunk size, given the current load conditions of P_k. In most cases the addition of the $\mathcal{W}$ mechanism improves the performance. However, when the loop is run on a dedicated homogeneous cluster, the $\mathcal{W}$ mechanism does not improve the performance and could be omitted.

The effects of the weighting mechanism on a self-scheduling algorithm $\mathcal{A}$ are shown in Fig. 4.15. The mechanism adds two components to the original algorithm:

(1) **chunk weighting** (master side) - the master adjusts the chunk size based on the slave's load information and computational power

(2) **run-queue monitor** (slave side) - it keeps track of the number of processes that require CPU time, updates q_k and informs the master of its current load.

Table 4.2 shows the chunk sizes given by the original and weighted self-scheduling algorithms. These chunks were obtained for a parallel loop (Mandelbrot computation[4]) of 10000×10000 points. Four slaves were used, having virtual computing powers $VP_1 = 1, VP_2 = 0.8, VP_3 = 1$ and $VP_4 = 0.8$. The two slowest slaves were loaded with an extra process, i.e., $q_2 = 2, q_4 = 2$, and their available computing power halved: $A_2 = 0.4$ and $A_4 = 0.4$.

Table 4.2 also shows the order in which slaves requested work from the master, which differs from algorithm to algorithm. Self-scheduling algorithms were devised for homogenous systems and they tend to assign large initial chunks to all slaves. They make the assumption that all slaves compute their assigned chunk in roughly the same time and advance to the next chunk simultaneously, as it is explained in [MJ94]. This assumption is not valid for heterogenous systems. Slower slaves may fall behind faster ones because they need more time to compute chunks of equal size. In most cases, slower slaves requested work only once throughout the whole execution. The weighting mechanism compensates for this deficiency as shown by the request orders in Table 4.2. The gain of the weighted algorithms over the non-weighted ones is also demonstrated by the times of the parallel execution in the same table.

[4]The Mandelbrot test case is described in Section 5.1.

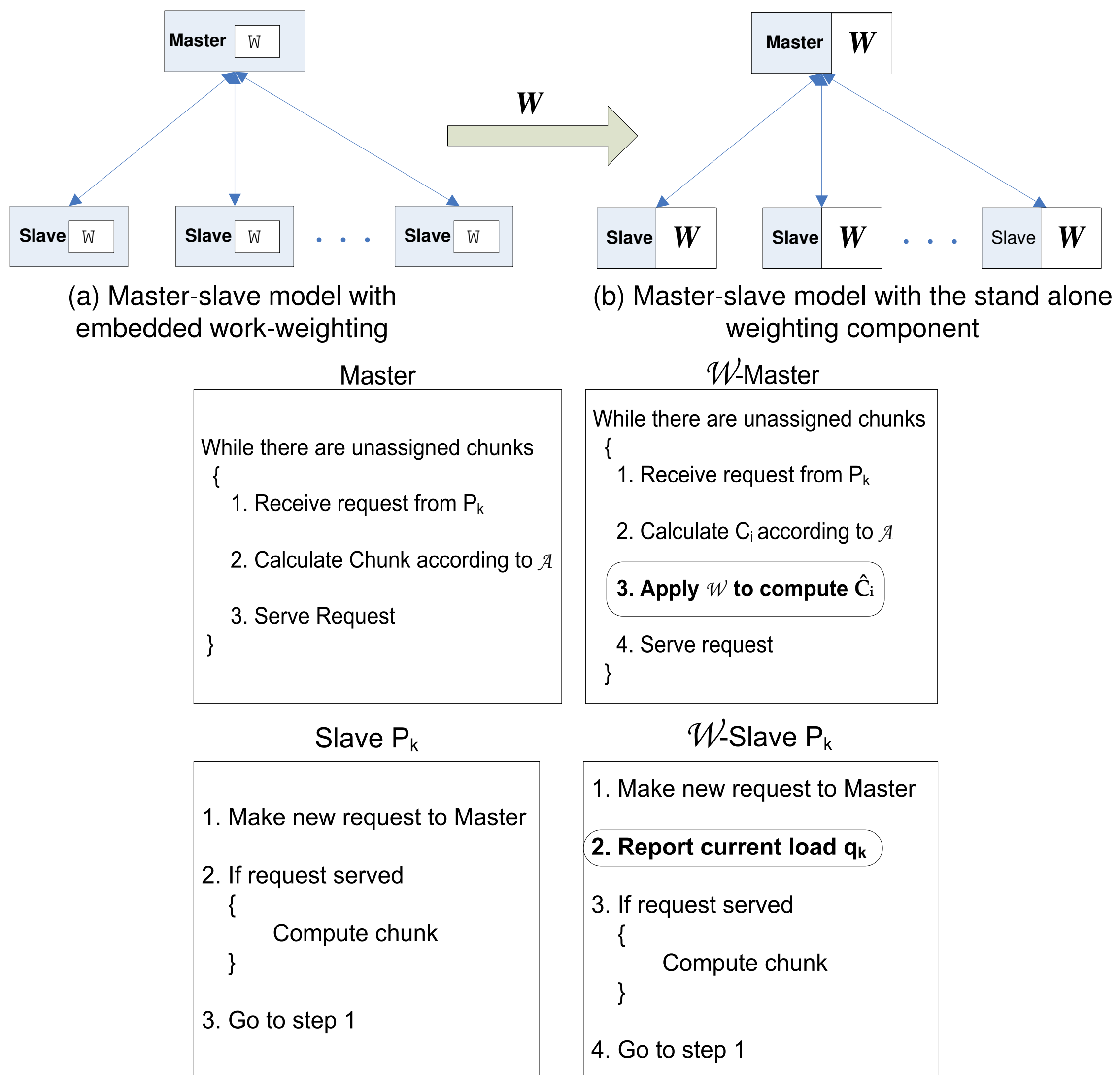


Figure 4.15: Master-slave model with weighting mechanism

Table 4.2: Chunk sizes given by the original and weighted algorithms for the Mandelbrot set, index space size $|J| = 10000 \times 10000$ points and $m = 4$ slaves

$\mathcal{A}$	Chunk sizes with $\mathcal{A}$ with respect to the processors' request order	Chunk sizes with $\mathcal{W}$-$\mathcal{A}$ with respect to the processors' request order	Par. time for $\mathcal{A}$	Par. time for $\mathcal{W}$-$\mathcal{A}$
CSS	$1250(P_1)$ $1250(P_2)$ $1250(P_3)$ $1250(P_4)$ $1250(P_3)$ $1250(P_1)$ $1250(P_3)$ $1250(P_1)$	$1250(P_1)$ $1250(P_3)$ $500(P_4)$ $500(P_2)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $1250(P_1)$ $1250(P_3)$ $500(P_4)$ $1250(P_1)$	$120.775s$	$66.077s$
FSS	$1250(P_1)$ $1250(P_3)$ $1250(P_2)$ $1250(P_4)$ $625(P_3)$ $625(P_3)$ $625(P_1)$ $625(P_3)$ $390(P_1)$ $390(P_1)$ $390(P_3)$ $390(P_1)$ $244(P_3)$ $244(P_4)$ $244(P_1)$ $208(P_3)$	$1250(P_1)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $812(P_3)$ $324(P_2)$ $324(P_4)$ $324(P_1)$ $324(P_3)$ $812(P_3)$ $630(P_1)$ $630(P_1)$ $630(P_4)$ $252(P_3)$ $176(P_1)$ $441(P_4)$ $441(P_2)$ $176(P_3)$ $123(P_1)$ $308(P_2)$ $308(P_4)$ $113(P_1)$	$120.849s$	$56.461s$
GSS	$2500(P_1)$ $1875(P_2)$ $1406(P_3)$ $1054(P_4)$ $791(P_3)$ $593(P_3)$ $445(P_3)$ $334(P_1)$ $250(P_3)$ $188(P_1)$ $141(P_3)$ $105(P_1)$ $80(P_3)$ $80(P_1)$ $80(P_3)$ $78(P_1)$	$2500(P_1)$ $1875(P_3)$ $562(P_2)$ $506(P_4)$ $455(P_4)$ $410(P_2)$ $923(P_3)$ $692(P_3)$ $519(P_1)$ $155(P_4)$ $140(P_2)$ $315(P_3)$ $94(P_4)$ $213(P_1)$ $160(P_3)$ $120(P_1)$ $90(P_3)$ $80(P_2)$ $80(P_1)$ $80(P_3)$ $31(P_1)$	$145.943s$	$58.391s$
TSS	$1250(P_1)$ $1172(P_3)$ $1094(P_2)$ $1016(P_4)$ $938(P_3)$ $860(P_1)$ $782(P_3)$ $704(P_1)$ $626(P_3)$ $548(P_4)$ $470(P_2)$ $392(P_1)$ $148(P_3)$	$1250(P_1)$ $1172(P_3)$ $446(P_2)$ $433(P_4)$ $1027(P_3)$ $388(P_4)$ $375(P_2)$ $882(P_1)$ $804(P_3)$ $299(P_4)$ $286(P_2)$ $660(P_1)$ $582(P_3)$ $504(P_1)$ $179(P_4)$ $392(P_3)$ $134(P_2)$ $187(P_1)$	$89.189s$	$63.974s$

The code blocks that implement the weighting mechanism are much shorter than those of the synchronization mechanism, as it can be seen from Fig. 4.15. In particular, we insert a block in the slave's code that monitors the current load of the slave at the time it makes a new request for work to the master. This load is then reported to the master along with the new request. On the master side, the block code implementing $\mathcal{W}$ performs a multiplication of the chunk size produced by the original self-scheduling algorithm according to the slave's reported virtual power and current load, using formula (4.2). Note that the performance gain of $\mathcal{W}$-$\mathcal{A}$ over $\mathcal{A}$ is quite significant.

4.4.5 The Combined Mechanisms – $\mathcal{SW}$

In Section 4.4.2 we applied the synchronization mechanism $\mathcal{S}$ to DOACROSS loops whereas in Section 4.4.4 we applied the weighting mechanism $\mathcal{W}$ to DOALL loops. In this section we combine the two mechanisms and demonstrate their effectiveness. The synchronization mechanism, while necessary to parallelize DOACROSS loops, does not provide any sort of load balancing. This makes advantageous the simultaneous use of the weighting mechanism in order to improve the overall performance of the self-scheduling schemes. It is obvious that a self-scheduling algorithm with both synchronization and weighting will outperform the same self-scheduling algorithm without weighting in a heterogeneous system.

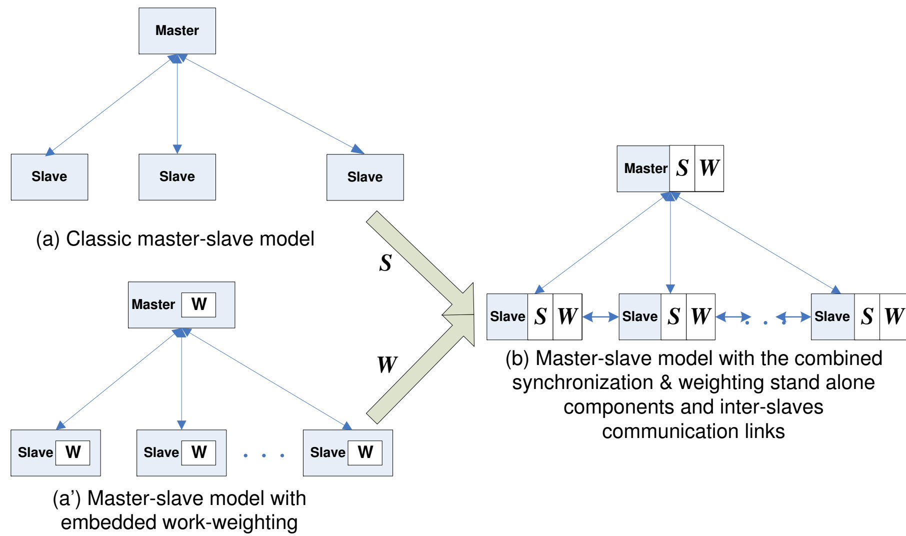
Figure 4.16 shows the effects of both synchronization and weighting mechanisms for the scheduling of dependence loops on heterogenous systems. As with the previous cases, the combined $\mathcal{SW}$ mechanisms add two components to the master: (1) **chunk weighting** and (2) **transaction accounting**, and three components to the slave: (3) **run-queue monitor**, (4) **receive part** and (5) **transmit part**. Component (1) (master-side) along with component (3) (slave-side) are related to the weighting mechanism, whereas all other components (both from master and slave) belong to the synchronization mechanism.

The chunk sizes of the synchronized–only and synchronized–weighted algorithms for a DOACROSS loop (Floyd-Steinberg computation[5]) of 10000×10000 points are shown in Table 4.3. Notice that due to the existing dependencies and synchronization, the slaves request order stays the same for a particular algorithm. The chunk sizes differ from the ones in Table 4.2, where no synchronization was used and the slave request order was random. Again, from the parallel times in Table 4.3 one can see that the synchronized–weighted algorithms perform better than the synchronized–only ones. It is worth mentioning that the synchronization and weighting overheads are compensated by the performance gain.

The implementation of the combination of the two mechanisms is actually the insertion of all code blocks associated with each of the mechanisms, both in the code of the slave and the code of the master, as it can be seen in Fig. 4.16.

The conclusions that can be drawn from the proposed mechanisms is that DOACROSS loops can be dynamically scheduled using $\mathcal{S}$, whereas self-scheduling algorithms are quite efficient on heterogeneous dedicated and non-dedicated systems using $\mathcal{W}$. Throughout this work, we identified the importance of the synchronization frequency and the need for a model to predict the optimal synchronization interval h which will minimize the communication. This is the focus of the next section.

[5] The Floyd-Steinberg test case is described in Section 5.1.



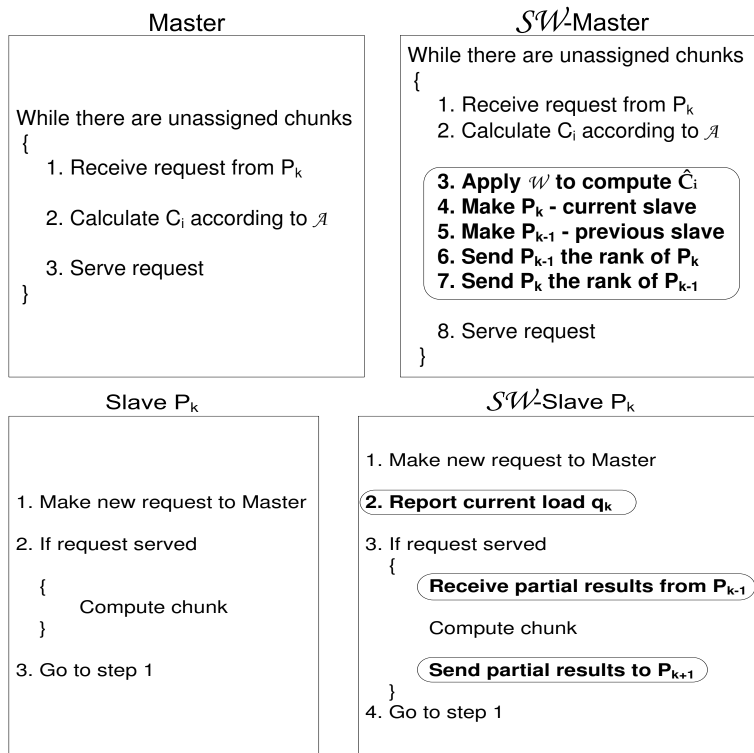


Figure 4.16: Master-slave model with the combination of synchronization and weighting mechanisms

Table 4.3: Chunk sizes given by the synchronized–only and synchronized–weighted algorithms for the Floyd-Steinberg loop, index space size $|J| = 10000 \times 10000$ points and $m = 4$ slaves

$\mathcal{A}$	Chunk sizes with $\mathcal{S}$-$\mathcal{A}$ with respect to the processors' request order	Chunk sizes with $\mathcal{SW}$-$\mathcal{A}$ with respect to the processors' request order	Par. time for $\mathcal{S}$-$\mathcal{A}$	Par. time for $\mathcal{SW}$-$\mathcal{A}$
CSS	$1250(P_1)$ $1250(P_3)$ $1250(P_2)$ $1250(P_4)$ $1250(P_1)$ $1250(P_3)$ $1250(P_2)$ $1250(P_4)$	$1250(P_1)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $1250(P_1)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $1250(P_1)$ $1250(P_3)$ $500(P_2)$	$27.335s$	$16.582s$
FSS	$1250(P_1)$ $1250(P_3)$ $1250(P_2)$ $1250(P_4)$ $625(P_1)$ $625(P_3)$ $625(P_2)$ $625(P_4)$ $390(P_1)$ $390(P_3)$ $390(P_2)$ $390(P_4)$ $244(P_1)$ $244(P_3)$ $244(P_2)$ $208(P_4)$	$1250(P_1)$ $1250(P_3)$ $500(P_2)$ $500(P_4)$ $812(P_1)$ $812(P_3)$ $324(P_2)$ $324(P_4)$ $630(P_1)$ $630(P_3)$ $252(P_2)$ $252(P_4)$ $488(P_1)$ $488(P_3)$ $195(P_2)$ $195(P_4)$ $378(P_1)$ $378(P_3)$ $151(P_2)$ $151(P_4)$ $40(P_1)$	$27.667s$	$16.556s$
GSS	$2500(P_1)$ $1875(P_3)$ $1406(P_2)$ $1054(P_4)$ $791(P_1)$ $593(P_3)$ $445(P_2)$ $334(P_4)$ $250(P_1)$ $188(P_3)$ $141(P_2)$ $105(P_4)$ $80(P_1)$ $80(P_3)$ $80(P_2)$ $78(P_4)$	$2500(P_1)$ $1875(P_3)$ $562(P_2)$ $506(P_4)$ $1139(P_1)$ $854(P_3)$ $256(P_2)$ $230(P_4)$ $519(P_1)$ $389(P_3)$ $116(P_2)$ $105(P_4)$ $237(P_1)$ $178(P_3)$ $80(P_2)$ $80(P_4)$ $108(P_1)$ $81(P_3)$ $80(P_2)$ $80(P_4)$ $25(P_1)$	$28.526s$	$18.569s$
TSS	$1250(P_1)$ $1172(P_2)$ $1094(P_3)$ $1016(P_4)$ $938(P_1)$ $860(P_2)$ $782(P_3)$ $704(P_4)$ $626(P_1)$ $548(P_2)$ $470(P_3)$ $392(P_4)$ $148(P_1)$	$509(P_2)$ $1217(P_1)$ $464(P_4)$ $1105(P_3)$ $420(P_2)$ $995(P_1)$ $376(P_4)$ $885(P_3)$ $332(P_2)$ $775(P_1)$ $288(P_4)$ $665(P_3)$ $244(P_2)$ $555(P_1)$ $200(P_4)$ $445(P_3)$ $156(P_2)$ $335(P_1)$ $34(P_4)$	$25.587s$	$14.309s$

4.5 Finding the Optimal Inter-Processor Synchronization Frequency

In this section we study the problem of finding the optimal synchronization frequency. We propose a theoretical model that estimates the parallel time of tasks in DOACROSS loops on heterogeneous systems. We show that the minimum parallel time is obtained theoretically with a synchronization frequency that minimizes the function giving the parallel time.

We describe next the communication and computation cost models we used to estimate the parallel time in a distributed (heterogeneous) system.

4.5.1 Communication Cost Model

We use a linear model for estimating the communication cost between workers. The cost of communicating a message between two workers is assumed to be the sum of two factors: the *start-up* cost c_d, representing the time to send a zero-length message including the hardware/software overhead of sending the message, and the *transmission cost* of the message. The transmission cost varies according to the size and type of data. The transmitted data can be of different type, e.g., float, double or any other user-defined type. The transmitted message consists of β rows of h data elements each. In this paper we assume that the dependencies create one row messages, that is $\beta = 1$. The transmission cost per unit and type of data is denoted c_c. For simplicity of explanation, we assume that the cost t_s of sending a message is equal to the cost t_r of receiving a message, as we explain in Chapter 5, Section 5.3.5. This is a simple but realistic assumption, since in most cases the send and receive operations are executed in pairs between communicating workers. Therefore, the cost t_r or t_s of communicating a message of size h data units, of data is:

$$t_r = t_s = c_d + hc_c \tag{4.3}$$

where c_d is the start-up cost and c_c is the network throughput, defined as $\frac{1}{sustained\ bandwidth}$, where *sustained bandwidth* is the ratio of the amount of data sent over the actual time measured at the application level.

4.5.2 Computation Cost Model

We define the computation cost as a linear function of the computation cost per iteration times the number of iterations. The computation cost per iteration is application- and processor-dependent. In this work we consider DOACROSS loops with regular loop bodies. The computation time per iteration, c_p, differs from processor to processor in heterogeneous systems, as is shown later in the paper (see Chapter 5, Section 5.3.5). Hence, the computation cost of a subchunk (i.e., the number of iterations between two successive synchronization points), is:

$$t_p = hV_i c_p \tag{4.4}$$

4.5.3 Optimal Inter-Processor Synchronization Frequency for CSS and $\mathcal{W}$-CSS

A few methods that focused on determining the optimal partitioning (tile size, block size, grain size) of DOACROSS nested loops were described in Section 1.2. The theoretical model

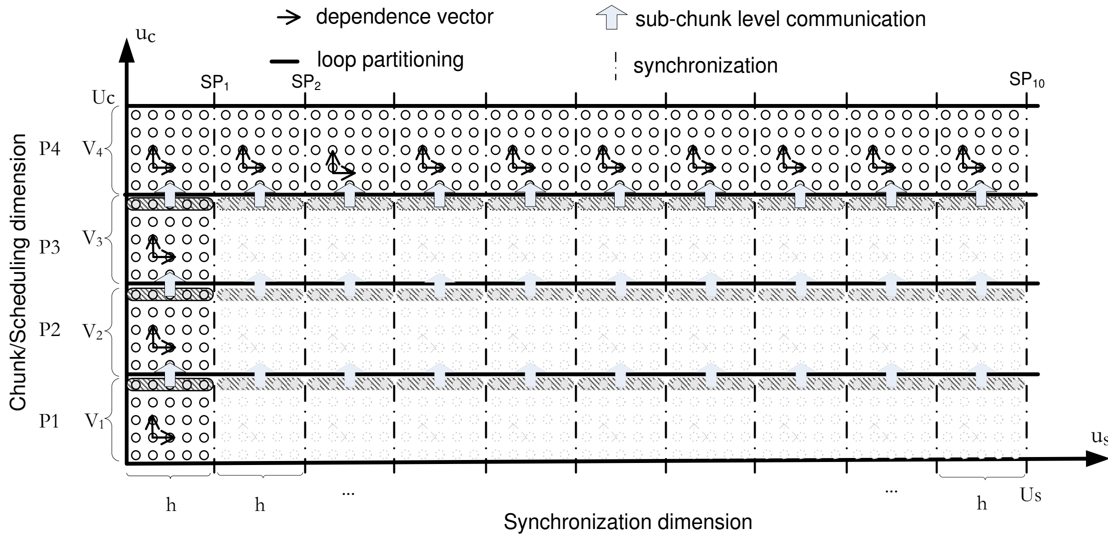


Figure 4.17: Partitioning a 2D loop into chunks and placement of synchronization points ($P = N$)

described in this section is one of the first attempts to determine the optimal synchronization frequency for DOACROSS loops on heterogeneous systems. It appeared in [CRA$^+$07a] and its extended version [CRA$^+$07b] is currently under review.

To find the optimal synchronization frequency, we first build a theoretical model for predicting the minimum parallel time for the simpler case of homogeneous dedicated systems. For the CSS algorithm with synchronization points [CAR$^+$06] [CRA$^+$08] we show that the minimum *actual* parallel time for homogeneous systems is achieved with a synchronization interval that is within +0.0750% from the theoretical optimal interval. Next, we build a theoretical model for heterogeneous dedicated systems, in which workers have different speeds. Again using the CSS algorithm, we show that the minimum actual parallel time for the particular heterogeneous system is achieved for a synchronization interval within +0.1750% of the theoretical optimal interval. For the experiments on the heterogeneous clusters, we use the weighting mechanism presented earlier (see 4.4.4) and apply it to the CSS algorithm such that the chunk assigned to a worker is *weighted* according to its computational power.

Fig. 4.17 depicts U_c, U_s, h, SP_j and V_i, for $i = 1, \ldots, 4$. In this work we assume a master-worker model with P workers, which are assigned the iterations to execute by the master. Fig. 4.18 depicts the parallel execution with one master and four workers. Workers are considered to be homogeneous and each worker is assigned one chunk.

We investigate next the impact of the synchronization interval h on the parallel execution time. We construct a mathematical model that estimates the parallel time, both for homogeneous and heterogeneous systems (which are dedicated to our application), as a function of the synchronization interval. The proposed model is valid if the following two reasonable premises are satisfied:

(1) The time to compute a subchunk is the same in all cases and for all worker types. This is a reasonable assumption because the size of the subchunk can be adjusted according to the computational power of a worker.

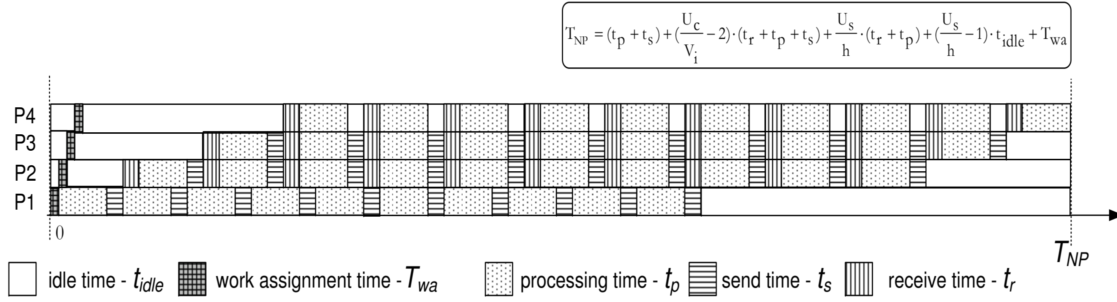


Figure 4.18: Parallel execution on a homogeneous system with $P = N$

(2) The communication time (sending and/or receiving data) for every subchunk is the same.

The special case when $P > N$ is not investigated, since it can be reduced to the case when $P = N$ by dropping the extra processors. Thus we have two cases: $P = N$ and $P < N$ to be studied and analyzed next.

I. Parallel Time Estimation for Homogeneous Systems

The case of homogeneous systems is presented in order to facilitate the understanding of the heterogeneous systems case (analyzed in Section 4.5.3).

Case $P = N$: This is the special case when there are as many processors available as the *chunks* of tasks, so that each processor is assigned exactly *one* chunk. Note that this is different from the case where there are as many processors available as tasks to be executed (i.e., fine grain parallelism). Fig. 4.17 illustrates a small scale index space partitioned into four chunks (horizontal segments) assigned to four workers. Ten synchronization points are inserted in each chunk. The parallelization strategy for this case is given in Fig. 4.18.

All four workers start by requesting work from the master. The master receives the first request, from P_1, it calculates the size of the first executable chunk and then it assigns it to P_1. The master continues to serve the other incoming work requests in the same fashion. On the workers' side, P_1 begins computing its assigned chunk. Due to the dependencies P_2 can start receiving and computing its assigned chunk only after P_1 has computed its first subchunk and sent the necessary data to P_2. This takes place at the first synchronization point SP_1. Similarly, P_3 can begin computations only after P_2 has sent the initially required data, again at the first synchronization point, but at a later time. This is the idle time t_{idle} shown in Fig. 4.18 by the horizontal white strip. The same holds for P_4, except for the sending part. P_4 does not need to send any data since it is the last worker. However, it must wait for P_3 to send the necessary data, thus introducing an expected idle time, as depicted in Fig. 4.18 by the white small boxes between computing and receiving.

Since V_i and c_p are the same for all four processors, it follows that the time needed to compute the iterations in a subchunk $t_p = hV_ic_p$, is the same for all processors. The communication between two workers at the subchunk level consists of two parts: the sending part and the receiving part. The send operation takes $t_s = c_d + hc_c$, and is equal to the receive operation, according to our communication model.

For the case when $P = N$, the total number of chunks is P and the size of each chunk is $V_i = U_c/P$. The theoretical parallel time for this situation, denoted T_P, is the completion time of the last subchunk of the problem (in our example this is the last subchunk of P4).

This can be estimated as the completion time of the highlighted subchunks in Fig. 4.17. We must clarify that for every chunk, except for the first and last chunk, a worker has to *receive*, *compute* and *send* data. For the first chunk however a worker needs only to *compute* and *send* data. Therefore, the time to *compute* each subchunk and *send* the necessary data to the next worker is: $t_p + t_s$. Likewise, for the last chunk a worker needs only to *receive* and *compute* data, i.e., the time required to received the necessary data and compute every subchunk is $t_r + t_p$. The time needed for the first subchunk of the chunks $2, 3, \ldots, P-1$ is: $(P-2)(t_r + t_p + t_s)$, where P can be written as $P = \frac{U_c}{V_i}$, since $P = N$ (see Fig. 4.17).

Proposition 4.5.1 *The time required to compute the last chunk is the product of the time to compute any one of its subchunks, i.e.,* $t_r + t_p$*, multiplied by the number of subchunks, i.e.,* $\frac{U_s}{h}$*, including the expected total idle time spent on waiting for the required data to be sent from the previous worker, i.e.,* $(\frac{U_s}{h} - 1)t_{idle}$*. We assume* t_{idle} *is approximately equal to* t_s*, as illustrated in Fig. 4.18.*

It is easy to see that this claim holds since, due to the dependencies, workers cannot work in parallel. A worker starts computing only after it receives the partial data from the previous worker. Therefore, the total parallel time is the completion time of the worker that computes the last chunk (and subchunk) of the problem.

Since we use the master-worker model, the work assignment time is $T_{wa} = t_r + c_{sch} + t_s$, where t_r is the transmission time needed for the work request to reach the master and t_s is the time needed for the master's reply to reach the worker; c_{sch} is the time needed for the master to compute the next executable chunk size, called *scheduling overhead*. Therefore, the total parallel time in this case is:

$$T_P = (t_p + t_s) + (\frac{U_c}{V_i} - 2)(t_r + t_p + t_s) + \frac{U_s}{h}(t_r + t_p) + (\frac{U_s}{h} - 1)t_{idle} + T_{wa} \quad (4.5)$$

Note that T_{wa} is taken only once because the work assignment time for every chunk is overlapped with the worker's computation or communication operations, except for the first chunk of the problem.

Fig. 4.18 gives a timing diagram which clarifies formula (3). Using the above formula in conjunction with formulas (1) and (2), one can determine the theoretically optimal value of h for which T_P is minimized. This is done by differentiating T_P with respect to h, which yields:

$$h_P = \sqrt{\frac{2c_d U_s}{(U_c - V_i)c_p + (2\frac{U_c}{V_i} - 4)c_c}} \quad (4.6)$$

Case $P < N$: In this case there are more chunks of tasks than the available processors and each processor is assigned *more than one* chunk. Fig. 4.19 depicts such a situation, in which there are four workers and 12 chunks, and every worker is assigned three chunks. For simplicity, in this analysis we assume that all workers are assigned the same number of chunks.

We address this case by assuming that a problem of size $U_c \times U_s$ with $P < N$ processors can be decomposed into k subproblems of size $U_c^\star \times U_s$, where $U_c{}^\star = \frac{U_c}{k}$, and $V_i = \frac{U_c{}^\star}{P}$, with $P = N/k$ processors. These subproblems are inter-dependent in the sense that part of the data produced by one subproblem are consumed by the next subproblem. Therefore, the computation of a subproblem cannot start until the previous subproblem is completed. Upon completion, the processor assigned the last chunk of the previous subproblem transmits

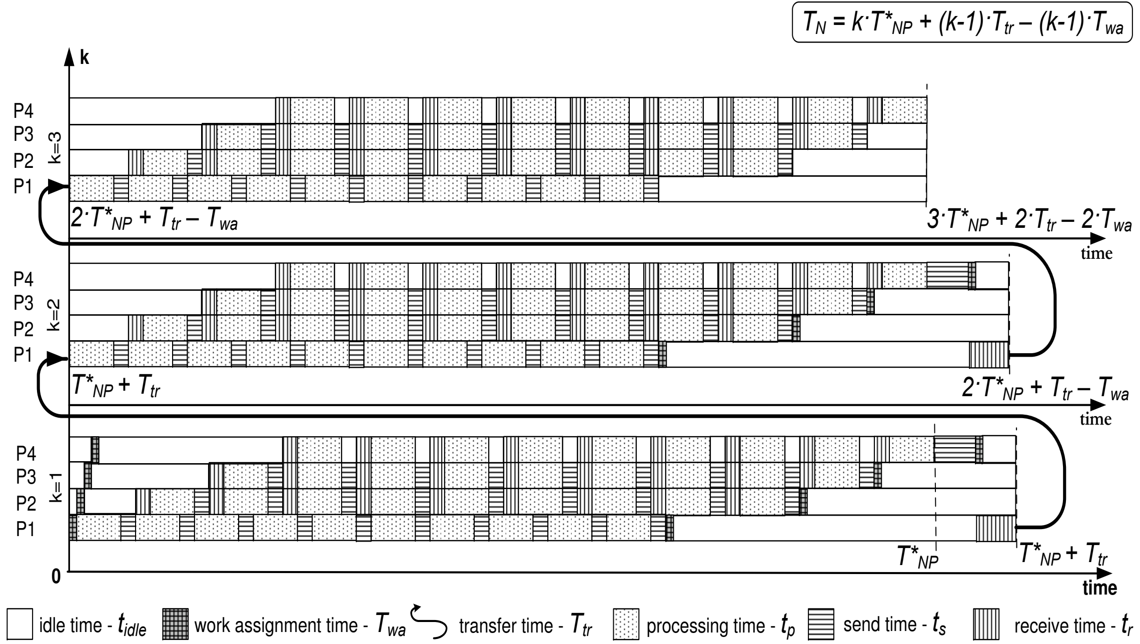


Figure 4.19: Parallel execution on a homogeneous system with $P < N$, k=3

(in a single message) all necessary data to the processor assigned the first chunk of the next subproblem. The time to complete this data transfer is called T_{tr} and is given by $T_{tr} = 2(c_d + U_s c_c)$; it designates the time to send and receive a data packet of size equal to the size of the scheduling dimension.

Fig. 4.19 illustrates a problem when $P < N$ problem, which is divided into 3 subproblems with $P = N/3$ and the resulting flow of parallel execution. The same set of processors is used for each subproblem.

In the general case, assuming k subproblems, the parallel time for the completion of the *first* subproblem is $T_P^\star$, which is the time required to compute a subproblem of size $U_c{}^\star \times U_s$. The parallel time for the completion of every subsequent subproblem is $T_P^\star - T_{wa}$ because the work assignment time for every chunk is overlapped with other communication or communication operations, except for the chunks of the first subproblem. Finally, the time to transfer the necessary data between all the successive subproblems is $(k-1)T_{tr}$.

Proposition 4.5.2 *The total parallel time for the case when $P < N$, denoted T_N, is:*

$$T_N = kT_P^\star + (k-1)T_{tr} - (k-1)T_{wa} \tag{4.7}$$

It is easy to see that this claim holds because, due to the dependencies, the execution of each subproblem can only start after it receives the partial data from the previous subproblem. Therefore, the total parallel time is the completion time of the subproblem that contains the last chunk of the original problem, plus the time required for the transfer of partial results from one subproblem to the next.

As in the previous case, one can determine the theoretical optimal value of h for which T_N is minimized by differentiating T_N with respect to h. Hence,

$$(T_N)' = k(T_P)' + ((k-1)T_{tr})' + ((k-1)T_{wa})' \tag{4.8}$$

where $T'_{tr} = 0$ and $T'_{wa} = 0$. This yields:

$$h_N = \sqrt{\frac{2c_d U_s}{(U_c{}^\star - V_i)c_p + (2\frac{U_c{}^\star}{V_i} - 4)c_c}} \tag{4.9}$$

II. Parallel Time Estimation for Heterogeneous Systems

To describe our theoretical model for a dedicated heterogeneous system, we use the notations of the previous section and the following additional notations:

- $\alpha_1, \ldots, \alpha_\xi$ – the worker types of the heterogeneous system.
- $NP^{\alpha_j} \geq 1$ – the # processors of type α_j, $j = 1, \ldots, \xi$.
- VP^{α_j} – the virtual computing power[6] of a worker of type α_j.
- $V_i^{\alpha_j}$ – the size of the projection of chunk i on u_c, assigned to processor of type α_j and weighted according to its virtual power.
- $c_p^{\alpha_j}$ – the computation time per iteration on a worker of type α_j.
- $t_p^{\alpha_j}$ – the computation time of $hV_i^{\alpha_j}$ iterations on a worker of type α_j.

In order to derive a formula for the theoretical parallel time on a heterogeneous system, we follow the homogeneous case approach and devise first a formula for the case when $P = N$, which we use to infer the formula for the case $P < N$. We assume that workers are assigned work in decreasing order of their computational power. This means that the last chunk is assigned to the slowest worker. In order to satisfy the first premise, i.e., the time to compute a subchunk is the same in all cases and for all worker types, the size of the chunk must be *weighted* according to the computational power of each worker type. We assume that workers are grouped according to their type, and that the group of workers having the greatest computational power per worker, gets assigned work first.
Case $P = N$: As with the homogeneous $P = N$ case, the parallel time is given by the completion time of the last subchunk of the problem. The completion time of the first subchunk assigned to the *last* worker is the total time required for the completion of the first subchunk of all previous chunks. In this scheme all workers are grouped according to their type and are assigned only one chunk, so, we can compute the total time for the completion of the first subchunks in each group. For all workers of type α_j, the total time for the completion of all corresponding *first* subchunks is $P^{\alpha_j}(t_r + t_p^{\alpha_j} + t_s)$. Therefore, the total time for all worker types is given by $\sum_{j=1}^{\xi}(P^{\alpha_j})(t_r + t_p^{\alpha_j} + t_s)$. From this sum we subtract t_r since there is no receive operation for the first subchunk of the problem, and t_s since there is no send operation for the last subchunk of the problem.

The time required for the computation of the remaining subchunks of the last chunk is $(\frac{U_s}{h} - 1)(t_r + t_p^{\alpha_\xi} + t_{idle})$ and it corresponds to a worker of type α_ξ which is the slowest type of workers of the heterogeneous systems.

Reasoning as in Proposition 4.5.2, we can prove the following Proposition:

[6] The virtual power for each machine type can be established as the normalized execution time of the same test program on each machine type.

Proposition 4.5.3 *The theoretical parallel time on a heterogeneous system when $P = N$ is:*

$$T_P = \sum_{j=1}^{\xi}(P^{\alpha_j})(t_r + t_p^{\alpha_j} + t_s) - t_r - t_s + (\frac{U_s}{h} - 1)(t_r + t_p^{\alpha_\xi} + t_{idle}) + T_{wa} \quad (4.10)$$

Differentiating T_P with respect to h to determine the optimal h we obtain:

$$h_P = \sqrt{\frac{2c_d U_s}{\sum_{j=1}^{\xi}(NP^{\alpha_j})(V_i^{\alpha_j} c_p^{\alpha_j} + 2c_c) - V_i^{\alpha_\xi} c_p^{\alpha_\xi} - 4c_c}} \quad (4.11)$$

Case $P < N$: In this case there are more chunks of tasks than the available processors and each processor is assigned *more than one* chunk. Chunks are again *weighted* according to the virtual power of a node.

Proposition 4.5.4 *The theoretical parallel time for the heterogenous case when $P < N$ is obtained similarly to the theoretical parallel time of the homogeneous case when $P < N$:*

$$T_N = kT_P^{\star} + (k-1)T_{tr} - (k-1)T_{wa} \quad (4.12)$$

where, $T_P^{\star}$ is the parallel time of the heterogeneous case when $P = N$ for a subproblem of size $U_c^{\star} \times U_s$, where $U_c^{\star} = \frac{U_c}{k}$, and $T_{tr} = 2(c_d + U_s c_c)$.

The proof is the same with that of Proposition 4.5.2.

As in the previous case, one can determine the theoretical optimal value of h for which T_N is minimized by differentiating T_N with respect to h. Hence,

$$(T_N)' = k(T_P)' + ((k-1)T_{tr})' + ((k-1)T_{wa})' \quad (4.13)$$

Taking into account that $T'_{tr} = 0$ and $T'_{wa} = 0$, we derive the optimal h:

$$h_N = \sqrt{\frac{2c_d U_s}{\sum_{j=1}^{\xi}(P^{\alpha_j})(V_i^{\alpha_j} c_p^{\alpha_j} + 2c_c) - V_i^{\alpha_\xi} c_p^{\alpha_\xi} - 4c_c}} \quad (4.14)$$

Formula (4.14) is similar to formula (4.11), only now the values of $V_i^{\alpha_j}$ are different because each worker is assigned more than one chunk.

CHAPTER 5

Implementation and Experimental Validation

5.1 Test Cases

To validate the usefulness of the various proposed algorithms, we experimented with several case studies. They are described below, along with their pseudocodes.

5.1.1 Mandelbrot Set

The Mandelbrot set generator [Man88] is a type of fractal model generator. Fractal models are used in many supercomputing applications. They are useful for predicting systems that demonstrate chaotic behavior. The Mandelbrot set is obtained from the quadratic recurrence equation $z_{n+1} = z_n^2 + C$, with $z_0 = C$, where points C in the complex plane for which the orbit of z_n does not tend to infinity are in the set. Setting z_0 equal to any point in the set gives the same result. The Mandelbrot set was originally called a μ molecule by Mandelbrot [Man88]. This application is a real life example that has no loop iteration dependencies (i.e., is a DOALL nested loop) but was considered herein because the loop iterations tasks are highly irregular in size. The pseudocode of the Mandelbrot fractal computation is given below:

```
/* Mandelbrot set */
    for (hy=1; hy<=hyres; hy++)  { /* scheduling dimension */
        for (hx=1; hx<=hxres; hx++)  {
            cx = (((float)hx)/((float)hxres)-0.5)/magnify*3.0-0.7;
            cy = (((float)hy)/((float)hyres)-0.5)/magnify*3.0;
            x = 0.0; y = 0.0;
            for (iteration=1; iteration<itermax; iteration++)  {
                xx = x*x-y*y+cx;
                y = 2.0*x*y+cy;
                x = xx;
                if (x*x+y*y>100.0)  iteration = 999999;
            }
            if (iteration<99999)  color(0,255,255);
```

```
            else color(180,0,0);
        }
    }
```

5.1.2 Full Search Block Matching Motion Estimation

Block motion estimation in video coding standards such as MPEG-1, 2, 4 and H.261 is perhaps one of the most computation-intensive multimedia operations. Hence is also the most implemented algorithm. The block matching algorithm is an essential element in video compression to remove the temporal redundancy among adjacent frames. The motion compensated frame is reconstructed from motion estimated blocks of pixels. Every pixel in each block is assumed to displace with the same 2D displacement called *motion vector*, obtained with the Block ME algorithm.

The Full-Search-Block-Matching Motion-Estimation Algorithm (FSBM ME) [YH95] is a block matching method, for which every pixel in the search area is tested in order to find the best matching block. Therefore, this algorithm offers the best match, at an extremely high computation cost. Assuming a *current* video frame is divided into $N_h \times N_v$ blocks in the horizontal and vertical directions, respectively, with each block containing $N \times N$ pixels, the most popular similarity criterion is the mean absolute distortion (MAD), defined as:

$$MAD(m,n) = \frac{1}{N^2} \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |x(i,j) - y(i+m, j+n)| \tag{5.1}$$

where $x(i,j)$ and $y(i+m, j+n)$ are the pixels of the current and previous frames. The motion vector (MV) corresponding to the minimum MAD within the search area is given by:

$$MV = arg\{minMAD(m,n)\}, -p \leq m,n \leq p, \tag{5.2}$$

where p is the search range parameter. The algorithm focuses on the situation where the search area is a region in the reference frame consisting of $(2p+1)^2$ pixels. In FSBM, MAD differences between the current block and all $(2p+1)^2$ candidate blocks are to be computed. The displacement that yields the minimum MAD among these $(2p+1)^2$ positions is chosen as the motion vector corresponding to the present block. For the entire video frame, this highly regular FSBM can be described as a six-level nested loop algorithm, as shown in the pseudocode below:

```
/* FSBM ME */
for (h=0; h<=Nh; h++)  {
    for (v=0; v<=Nv; v++)  {
         MV[h][v] = {0,0};
         Dmin[h][v] = maxint;
         for (m=0; m<=2*p; m++)  {
             for (n=0; n<=2*p; n++)  {
                 MAD[m][n]=0;
                 for (i=h*N; i<=h*N+N; i++)  {
                      for (j=v*N; j<=v*N+N; j++)  {
                           MAD[m][n] += fabsf(x[i][j] - y[i+m][j+n]);
```

```
                    } // end j
                } // end i
                if (Dmin[m][n] > MAD[m][n]) {
                    Dmin[m][n] = MAD[m][n];
                    MV[h][v] = {m,n};
                } // end if
            } // end n
        } // end m
    } // end v
} // end h
```

Note that the loop body itself contains nested DOACROSS loops. However it has no dependencies with respect to the two outer loops and the entire loop body is executed sequentially. This means that the dependencies inside the loop body are satisfied. This is acceptable because our tool is designed to handle such complex loop bodies. Moreover, it is desirable to have such computationally intensive loop bodies, in order to maximize the performance gain of the parallel program.

5.1.3 Heat Conduction

The heat conduction computation is one of the most widely used case studies in the literature, and its loop body is similar to the majority of the numerical methods used for solving partial differential equations. It computes the temperature in each point of a grid, based on two values of the current time step (`A[i-1][j], A[i][j-1]`) and two values from the previous time step (`A'[i+1][j], A'[i][j+1]`), over a number of `loop` time steps. The pseudocode is given below:

```
/* Heat Conduction */
for (l=1; l<loop; l++) {
    for (i=1; i<width; i++){  /* synchronization dimension */
        for (j=1; j<height; j++){ /* scheduling dimension */
            A[i][j] = 1/4*(A[i-1][j] + A[i][j-1]
                   + A'[i+1][j] + A'[i][j+1]);
        }
    }
}
```

5.1.4 Hydrodynamics

Hydro is a modified version of the Livermore kernel 23 (Implicit Hydrodynamics fragment) [McM86], widely used in hydrodynamics. It is a 3-dimensional DOACROSS loop, which we modified in order to explicitly show the 3-dimensional dependencies among iterations for the array `za`. The pseudocode is given below:

```
/* hydro */
for (l=1; l<=loop; l++) { /* synchronization dimension */
    for (j=1; j<5; j++)  {
        for (k=1; k<n; k++){ /* scheduling dimension */
```

```
                qa = za[l-1][j+1][k]*zr[j][k] + za[l][j-1][k]*zb[j][k] +
                     za[l-1][j][k+1]*zu[j][k] + za[l][j][k-1]*zv[j][k] +
                     zz[j][k];
                za[l][j][k] += 0.175 * (qa - za[l][j][k] );
            }
        }
    }
```

5.1.5 Floyd-Steinberg Dithering

The Floyd-Steinberg computation [FS76] is an image processing algorithm used for the error-diffusion dithering of a *width* by *height* grayscale image. The boundary conditions are ignored. The pseudocode is given below:

```
/* Floyd-Steinberg */
for (i=1; i<width; i++){ /* synchronization dimension */
    for (j=1; j<height; j++){ /* scheduling dimension */
            I[i][j] = trunc(J[i][j]) + 0.5;
            err = J[i][j] - I[i][j]*255;
            J[i-1][j] += err*(7/16);
            J[i-1][j-1] += err*(3/16);
            J[i][j-1] += err*(5/16);
            J[i-1][j+1] += err*(1/16);
    }
}
```

5.1.6 Transitive Closure

Transitive closure is used to find if any two vertices in a graph are connected. Formally, if $G = (V, E)$ is a graph, then the transitive closure of G is defined as the graph $G^* = (V, E^*)$, where $E^* = \{(i, j)|$ there is a path from i to j in $G\}$. The transitive closure of a graph is determined by computing the connectivity matrix P; $P[i][j] = 1$ if there is a path from i to j, and 0 otherwise. Matrix P can be obtained using Warshall's algorithm given below. In this algorithm A is the adjacency matrix of G, that is $A[i][j] = 1$ iff $(i, j) \in E$.

```
/* Transitive Closure */
for (i=0; i<N; i++){
    for (j=0; j<N; j++){
    /* There is a path if there is an edge */
        P[i][j] = A[i][j];
    } // end j
} // end i
for (k=0; k<N; k++){
    for (i=0; i<N; i++){
        for (j=0; j<N; j++){
            if (!P[i][j])
                P[i][j] = P[i][k] && P[k][j];
        } // end j
```

```
        } // end i
    } // end k
```

5.1.7 Artificial Code

The artificial code is a 2-dimensional nested loop with a general loop body (which includes a third loop). It has five 2-dimensional dependence vectors, given by the two outer loops.

```
/* Artificial Code */
for (i=0; i<N1; i++){
    for (j=0; j<N2; j++){
        A[i][j] = 2*A[i-1][j-8]+A[i-8][j-1]+3*B[i-2[j-5];
        B[i][j] = 0;
        for (k=0; k<j; k++){
            B[i][j] += C[i][k]*C[i][j];
        } // end k
        B[i][j] += B[i-3][j-3]+B[i-6][j-2];
    } // end j
} // end i
```

5.2 Static Methods

In this section we describe the experimental evaluation and validation of the static methods described in Chapter 3. The first evaluated algorithm is BSDA (Binary Scheduling Decision Algorithm), the second one is DA (Decision Algorithm) and the last one is CPS (Chain Pattern Scheduling). ACS (Adaptive Cyclic Scheduling) was not implemented and evaluated.

5.2.1 Evaluation of BSDA

We developed a tool, called BSDA-Tool, that generates random UDLs and then determines the optimal schedules. We ran experiments with 2D, 3D, 4D and 5D random UDLs, with the number of dependence vectors ranging from 3-6 and 7-10. The results given in Table 5.1 and Fig. 5.2.1, show the percentage of UDLs that have $OP = LB$, the percentage that have OP within 10% of the LB, etc.

While generating random UDLs, a special subclass of UDLs was identified: the scaled grids (SGRIDs). It includes those UDLs of which the dependence vectors reside on the axes and the value of the non-zero coordinate is an arbitrary positive integer, i.e., these dependence vectors are non-unitary. We observed that SGRIDs always require UB processors for an optimal schedule, *if* the size of the index space in each dimension is *divisible* with the corresponding coordinate of the dependence vector of this dimension. This means that the size of the index space in a certain dimension is a multiple of the size of the dependence vector residing on that axe. In order to produce the optimal schedule, the DA algorithm must be used.

The statistical analysis of the experimental results lead us to the following conclusions:

(1) UB processors must be used for SGRIDs.

(2) The vast majority of UDLs (55% to more than 80%, in some cases), needs an OP within 10% of the LB.

Table 5.1: Execution evaluation for 2D, 3D, 4D, 5D UDLs and the percentage of binary searching for OP between LB and UB

dim.#	*dep.#*	LB	1-10 %	10-50 %	50-99 %	UB
2	3-6	63.1%	9.2%	19.5%	6.9%	1.2%
dim.	7-10	46.1%	7.9%	26.1%	17.1%	2.9%
3	3-6	71.9%	6.3%	10.6%	8.8%	2.4%
dim.	7-10	64.4%	10.3%	16.1%	7.3%	1.9%
4	3-6	75.3%	6.6%	8.7%	7.4%	2.0%
dim.	7-10	65.6%	12.4%	13.9%	6.1%	1.9%
5	3-6	77.2%	7.6%	6.6%	6.9%	1.7%
dim.	7-10	68.3%	7.9%	15.2%	6.5%	2.1%

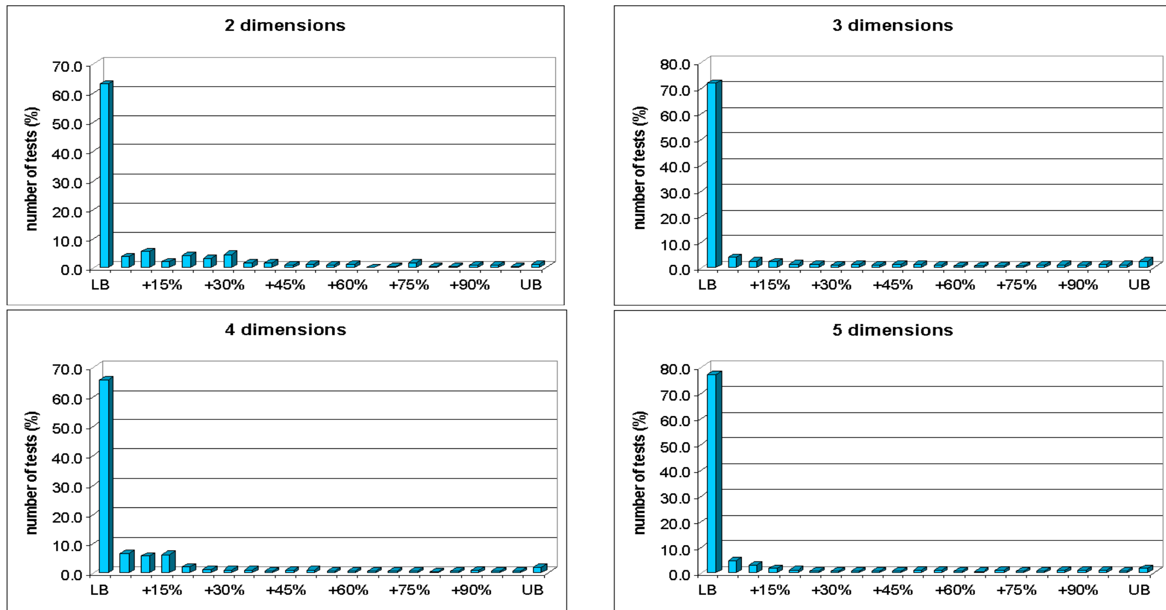


Figure 5.1: Test results of scheduling 2D, 3D, 4D, 5D UDLs with $OP \in [LB, UB]$

5.2.2 Evaluation of DA

In order to verify that all UDLs can achieve the proposed dynamic lower bound, $LB = max\{LB_1, LB_2, LB_3\}$, we use the DA described in Section 3.2.2. The BSDA-Tool is used to generate random UDLs. The goal is to determine if an optimal schedule can be found for each UDL using the candidate number of processors. We experimented with 2D, 3D, 4D and 5D random UDLs. In each case, we compare the results obtained using $LB = max\{LB_1, LB_2\}$ with the one obtained using $LB = max\{LB_1, LB_2, LB_3\}$. The results given in Fig. 5.2.2–5.2.2, show the percentage of UDLs that have $OP = LB$, the percentage that have OP within 10% of the LB, etc.

The results indicate that *all* UDLs are scheduled using $LB = max\{LB_1, LB_2, LB_3\}$ processors, in comparison with the results before introducing LB_3. The statistical analysis of the experimental results lead us to the conclusion that we can schedule every random UDL using the proposed lower bound. This means we can know *a priori* the number of processors

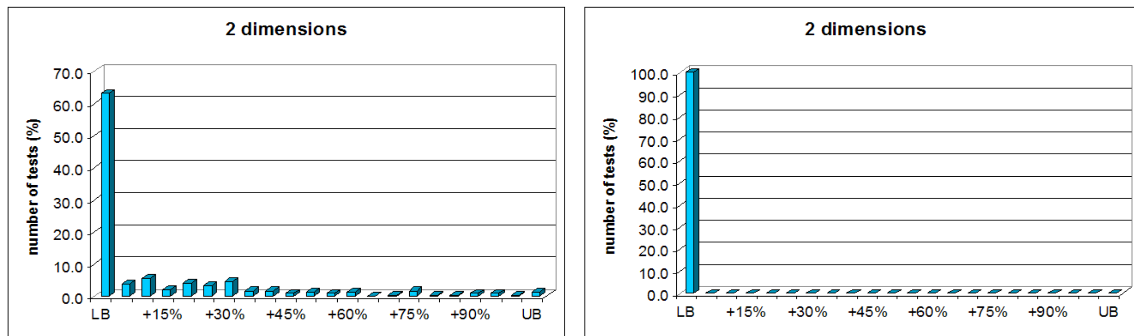


Figure 5.2: Test results of scheduling 2D UDLs: left - without the LB_3, right - with the LB_3

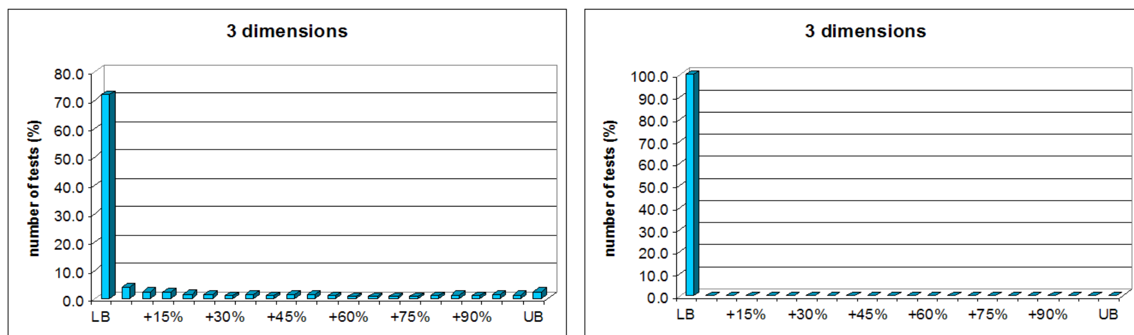


Figure 5.3: Test results of scheduling 3D UDLs: left - without the LB_3, right - with the LB_3

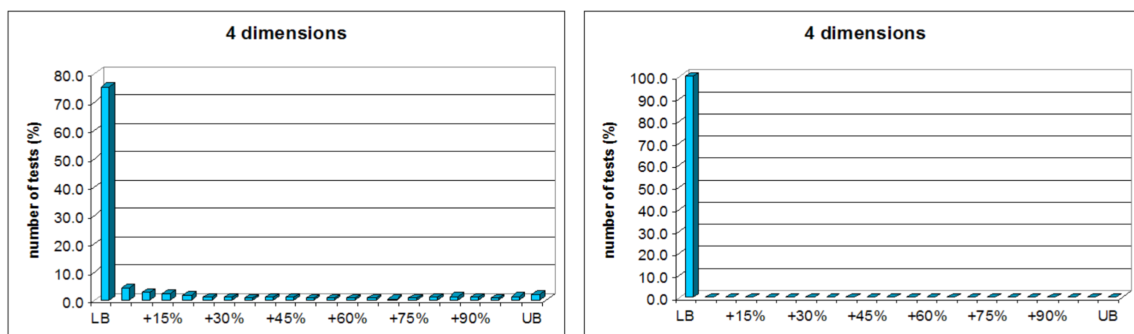


Figure 5.4: Test results of scheduling 4D UDLs: left - without the LB_3, right - with the LB_3

needed for the optimal schedule and we can start building the schedule using the appropriate lower bound, hence reducing the hardware cost. Also, we conclude that for SGRIDs, UB actually coincides with LB_3. Moreover, if the size of the index space in each dimension of an SGRID UDL is *divisible* with the corresponding coordinate then every index point is crucial and $LB_2 = UB$.

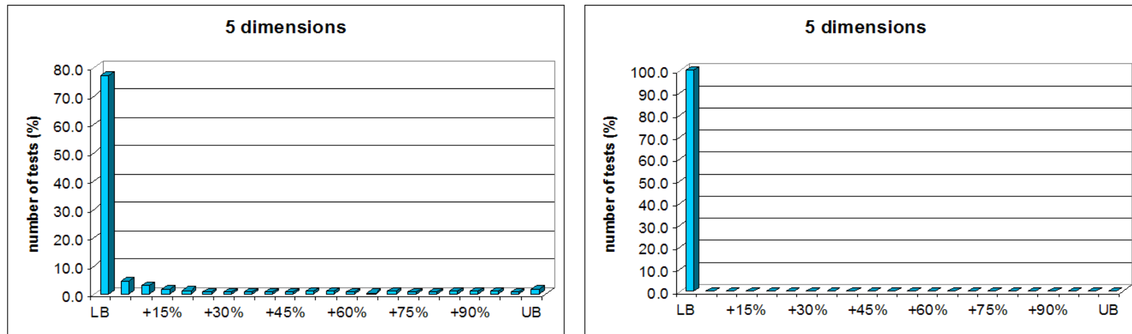


Figure 5.5: Test results of scheduling 5D UDLs: left - without the LB_3, right - with the LB_3

5.2.3 Evaluation of CPS

To validate the methodology proposed in Section 3.4.2, a program written in C++, which emulates the distributed-memory systems model, was used. This program implements both the cyclic scheduling method [MA01], and the chain pattern scheduling. We experimented with index spaces ranging from 10×10 to 1000×1000 index points. For all index spaces, the four dependence vectors of the loop nest given in Fig. 3.7 and the communication vector (2,2) were considered. Fig. 5.6–5.9 give the simulation results when NP ranges from 5 to 8 processors. Note that in every case, and for all index space sizes, the chain pattern mapping performs better than the classic cyclic mapping. In particular, the communication reduction achieved with the chain pattern mapping ranges from 15% - 35%.

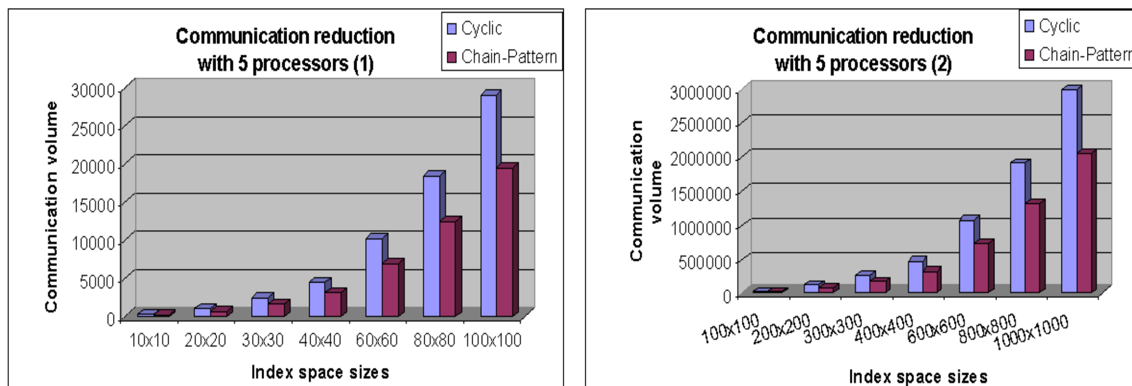


Figure 5.6: Experimental results, CPS vs cyclic, on $NP = 5$

The conclusions that can be drawn regarding CPS are outlined below:

- The total communication cost can be significantly reduced if the communication incurred by certain dependence vectors is eliminated
- The chain pattern mapping outperforms other mapping schemes (cyclic mapping) by enhancing the data locality

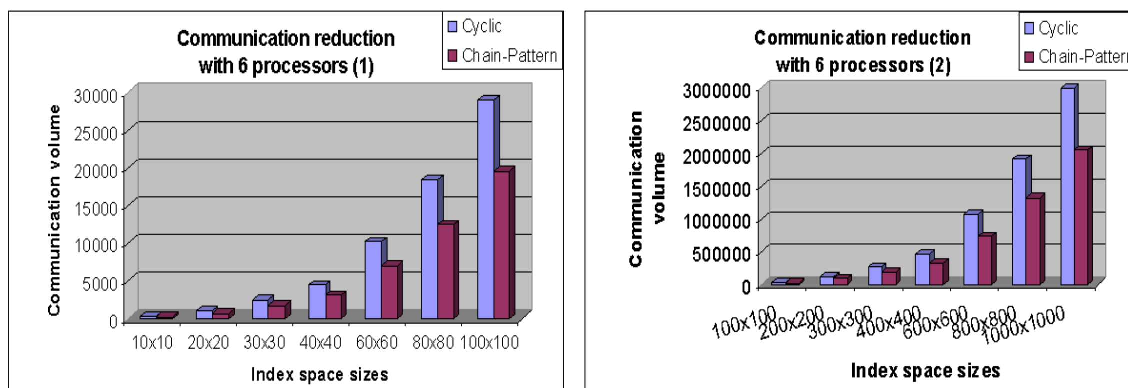


Figure 5.7: Experimental results, CPS vs cyclic, on $NP = 6$

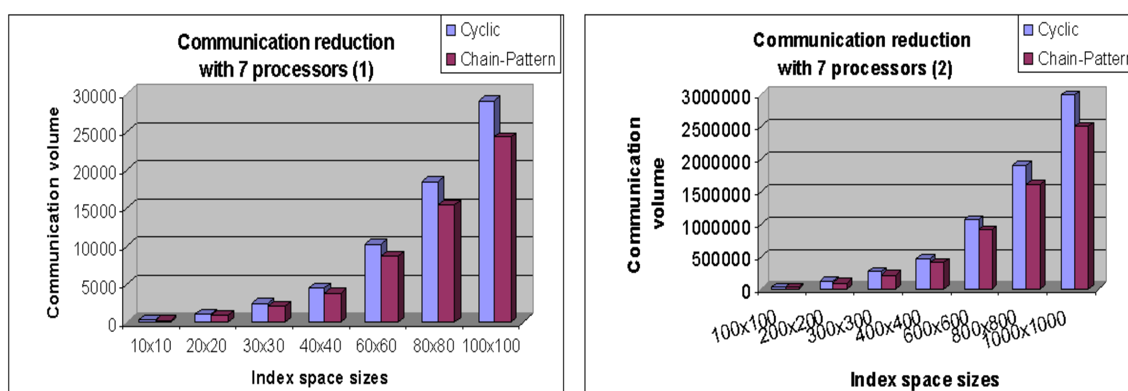


Figure 5.8: Experimental results, CPS vs cyclic, on $NP = 7$

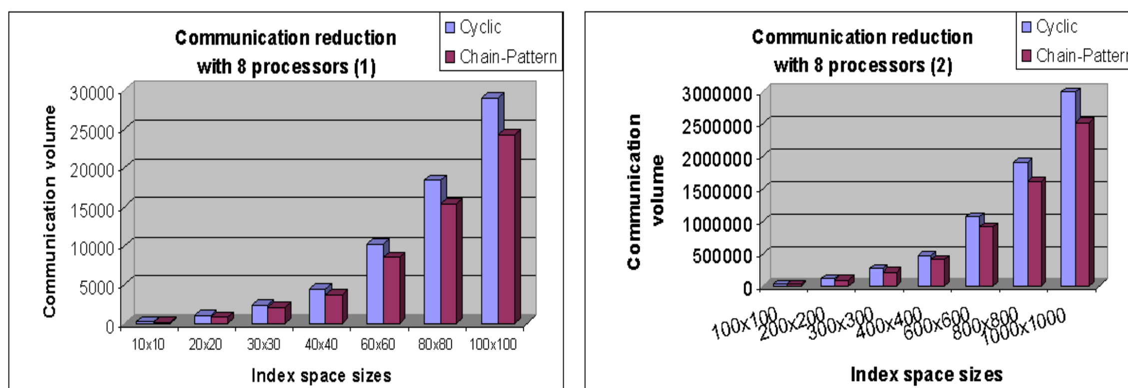


Figure 5.9: Experimental results, CPS vs cyclic, on $NP = 8$

5.3 Dynamic Methods

In this section we describe the experimental evaluation and validation of all dynamic methods described in Chapter 4. We first describe the parallel code generation tool devised for SGRIDS, then the automatic code generation tool, Cronus, devised for general loops (GLs). Next, we describe ...

5.3.1 Evaluation of the Automatic Code Generation Tool for Scaled Grids

Recall that the automatic code generation tool performs the following tasks:

Input: The sequential C code, P and the target system (shared-memory or distributed-memory)

Extracts: n, $|J|$, m, SpecPat, s

Performs: transforms J into J_{aux}; forms processor groups with ranks ranging $1, \ldots, \lfloor \frac{P}{m} \rfloor$

Outputs: The near-optimal equivalent parallel C+MPI code

The tool was implemented in C with MPI primitives for the parallel execution and data exchange. The resulting MPI code uses point-to-point communication and standard blocking send and receive MPI operations, meaning that a send or receive operation blocks the current process until resources used for the operation can be reutilized. The experiments were performed on a cluster with 16 identical 500MHz Pentium nodes. Each node has 256MB of RAM and 10GB hard drive and runs Linux with 2.4.20 kernel version. MPICH was used to run the experiments FastEthernet.

We evaluated the performance of the tool by means of randomly generated SGRID UDLs, described in Table 5.2. The speedup of the generated parallel code over the sequential program was measured. It is expected that the tool schedules and achieves speedups that scale with the number of available hardware resources. As validated by the experimental results (Figure 5.3.1), the proposed methodology obtained near optimal results for seven reasonably sized randomly generated applications, given in Table 5.2. The tool adapts well to shared memory systems by simply replacing the MPI communication routines with read and write calls from and to the shared memory of the multicomputer system.

Table 5.2: SGRID examples used to test the automatic code generation tool

Test #	# of dim, n	Size of index space $\|J\|$	Dependencies
1	2	400 points	$d_1 = (0,2)$, $d_2 = (1,0)$ and $d_3 = (1,2)$
2	2	600 points	$d_1 = (0,1)$, $d_2 = (1,0)$ and $d_3 = (1,1)$
3	2	1024 points	$d_1 = (0,4)$, $d_2 = (2,0)$ and $d_3 = (4,2)$
4	3	1000 points	$d_1 = (0,0,2)$, $d_2 = (0,2,0)$, $d_3 = (2,0,0)$ and $d_4 = (2,2,2)$
5	3	2744 points	$d_1 = (0,0,2)$, $d_2 = (0,2,0)$, $d_3 = (1,0,0)$ and $d_4 = (1,2,2)$
6	3	2744 points	$d_1 = (0,0,2)$, $d_2 = (0,1,0)$, $d_3 = (1,0,0)$ and $d_4 = (1,1,2)$
7	3	3375 points	$d_1 = (0,0,1)$, $d_2 = (0,1,0)$, $d_3 = (1,0,0)$ and $d_4 = (1,1,1)$

The experimental results proved to be very close to the ideal speedup. However, by using standard blocking send/receive communication primitives, the tool introduces a communication overhead which can slightly reduce the speedup when the number of utilized processors exceeds an optimal threshold, i.e., when the communication to computation ratio becomes higher than one. On the other hand, the choice of using blocking communication primitives brings the advantage of accuracy regarding the appropriate data exchanged between processors at execution, and the experimental results proved to be near optimal even in this case.

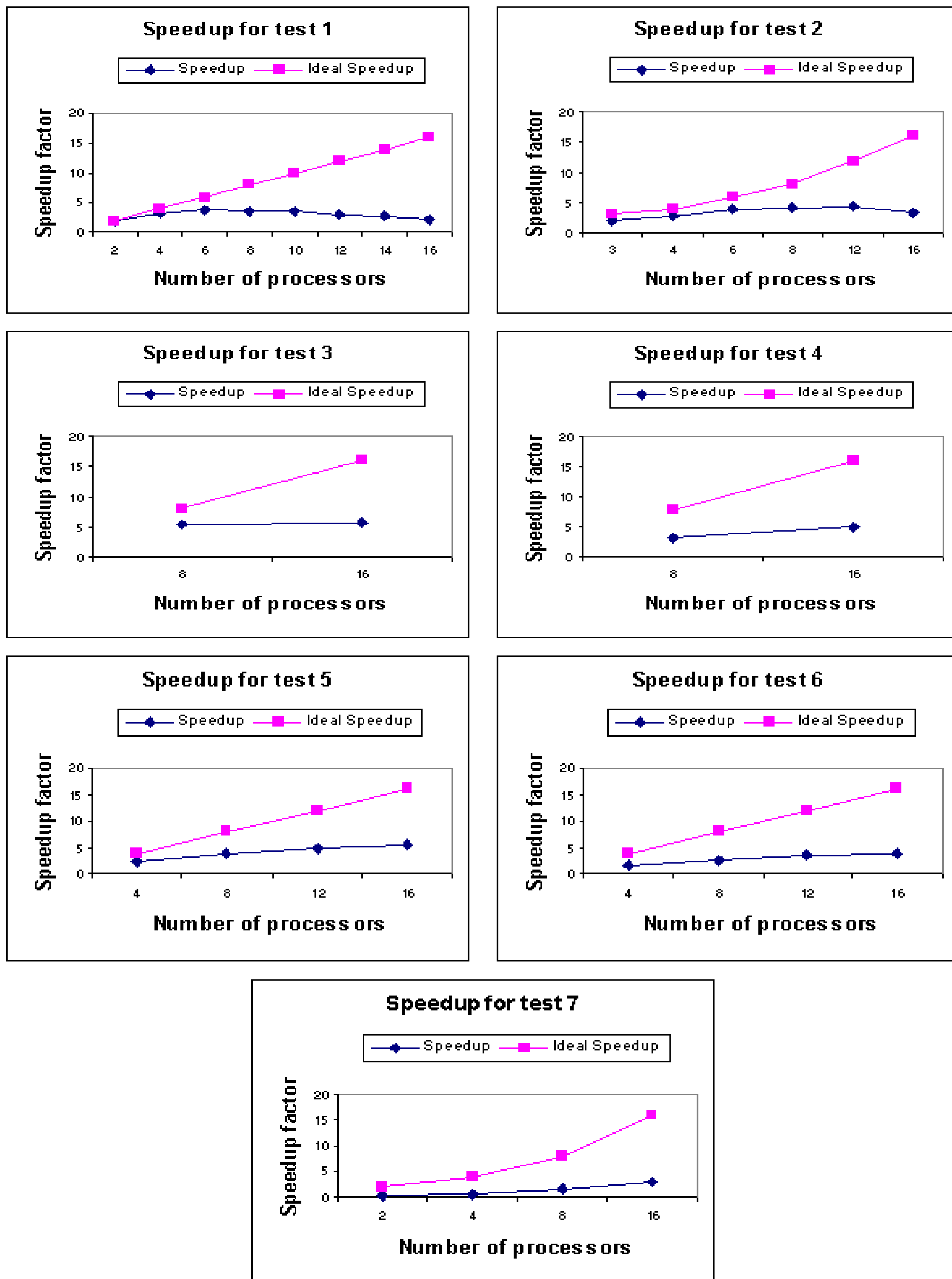


Figure 5.10: Speedup results for 7 testing examples

5.3.2 Evaluation of CRONUS for General Loops

CRONUS was coded in C, except for the code generator written in Perl. The parallel code produced by CRONUS uses point-to-point, synchronous send and receive MPI calls when required. The experiments were conducted on a cluster with 16 identical 500MHz Pentium III nodes. Each node has 256MB of RAM and 10GB hard drive and runs Linux with 2.4.23 kernel version. MPI (MPICH) was used to run the experiments over the FastEthernet interconnection network.

The experiments were conducted on (a) an SGI ORIGIN 2000 shared memory system, running IRIX 6.5, with 16 R10000 processors, total memory 4GB RAM, and 300 GB hard drives connected through raid 5 CXFS system; and (b) on a cluster with 16 identical 500MHz Pentium III nodes, each node having 256MB RAM and 10 GB hard drive and runs Linux with 2.4.23 kernel version. MPI (MPICH) was used to run the experiments on the shared memory system (a) and over the FastEthernet interconnection network of the cluster (b).

The MPI standard was chosen for the inter-processor communication due to the fact that it is widely portable. Also, the MPI implementation used on the shared memory system is a shared memory optimized implementation that supports the MPI 1.2 standard, as documented by the MPI Forum [For02]. In addition, certain MPI-2 features are also supported. On this system, MPI uses the first interconnect it can detect and configure correctly. By default, if MPI is run across multiple hosts, or if multiple binaries are specified on the `mpirun` command, the software searches for interconnects in the following order (on IRIX systems): XPMEM, GSN, MYRINET, TCP/IP. There is only one interconnect configured for the entire MPI job, with the exception of XPMEM. If XPMEM is found on some hosts, but not on others, one additional interconnect is selected. However, this was not the case in our examples as the MPI had access only to a single 16-CPU host, so all communication utilized shared memory buffers.

The **parallel execution time** is the sum of *communication time*, *computation time* and *idle time* (time spend by a processor waiting for data to become available or points to become eligible for execution). The computation time is the run time of the algorithm itself, whereas the communication time contains both the time of the actual communication between processors as well as the additional time needed by the dynamic scheduling algorithm. This additional time is a known overhead of every dynamic scheduling technique so we need to keep it to a minimum. We have measured it in our test cases in order to get the percentage of this time over the full communication time. We found this to vary according to the example in question, but in all cases it was between 7% to 30% of the total (measured) communication time. Specifically, for case study I (Floyd-Steinberg) this was between 15%-25%, for case study II (Transitive Closure) around 7%-9% and for case study III (artificial snippet) around 20%-30%. We believe these values are acceptable since we employ a dynamic decentralized fine grain scheduling algorithm. We have not measured this overhead for case study IV (FSBM) because in this case the absence of dependencies results in a negligible overall communication percentage (see Fig. 5.18 for details).

The **speedup**, denoted S_n, is defined as the sequential execution time (T_1) over the parallel execution time (T_n): $S_n = \frac{T_1}{T_n}$. Figures 5.11, 5.13, 5.15 and 5.17 depict the obtained speedup vs. the number of processors, both for the shared-memory system and for the distributed-memory system described above. Note that the parallel time is the average of tens of runs for each test case.

I. Cronus vs. Berkeley UPC

Berkeley UPC is an open-source implementation of the language Unified Parallel C [LB]. Unified Parallel C (UPC) is an extension of the C programming language designed for high performance computing on large-scale multiprocessors, PC clusters, and clusters of shared memory systems. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. The UPC execution model is similar to that used by the message passing style of programming (MPI or PVM). This model is called Single Program Multiple Data (SPMD) and it is an explicitly parallel model in which the amount of parallelism is fixed at program startup time. It works by translating UPC code to equivalent C-only code that, in a way similar to Cronus, relies on a runtime library for communication and data allocation primitives. The underlying communication library is GASNet [Bon], which supports an extensive range of target architectures and communication methods (MPI, native SMP, UPD over Ethernet etc.). Internally, communication in GASNet is heavily based on the Active Messages Paradigm [vECGS92].

Clearly, Berkeley UPC is a more generic system than Cronus and its runtime library is designed to handle a much broader range of problems. On the other hand Cronus' runtime library has been optimized exclusively for general loops, which implies that it should perform much better than a generic solution. For testing this hypothesis, we compared the performance of Cronus' generated program to the performance of a UPC-generated equivalent for the same test cases on a distributed memory cluster of workstations. We used the SDS scheduling algorithm for both programs, so in fact the parts of Cronus that deal with SDS had to be ported to UPC. This was a purposeful decision, since what we wanted to measure was how the two platforms perform in terms of communication and data locality handling. We used the same cluster as for the other case studies. The results are shown in Fig. 5.22.

II. Experiments and Results

In every experiment, both the shared-memory and the distributed memory systems were dedicated. This means that no other load existed in the systems except for our own experiment, and allowed us to achieve the best performance on these systems, using our platform. Obviously, if there are other loads in the system, the results will not be as good as the ones presented here.

In Fig. 5.11 we show the speedup with respect to the number of processors for the Floyd-Steinberg case. One can see that the speedup of the parallel code ranges from 3.95 (out of the ideal 4) to 15.70 (out of the ideal 16) in the shared-memory case and 1.18 (out of 4) to 4.21 (out of 16) in the distributed-memory case. In Fig. 5.11 (last chart) we compare the speedup obtained on the shared-memory system with the one obtained on the distributed-memory system, for a 2000×2000 index space. This confirms our prediction that Cronus performs much better in shared-memory systems, when the application is communication-intensive. This is depicted in Fig. 5.12 where one can see that the percentage of the parallel time used for communication is much higher on a distributed system. In Fig. 5.19 we show how much of the measured communication time from Fig. 5.12 is spent on real communication (i.e., MPI primitives) and how much is attributed to the scheduling overhead.

Fig. 5.13 gives the results for the Transitive Closure algorithm; the speedup of the parallel program ranges from 3.40 (out of the ideal 4) to 13.74 (out of the ideal 16) in the shared-

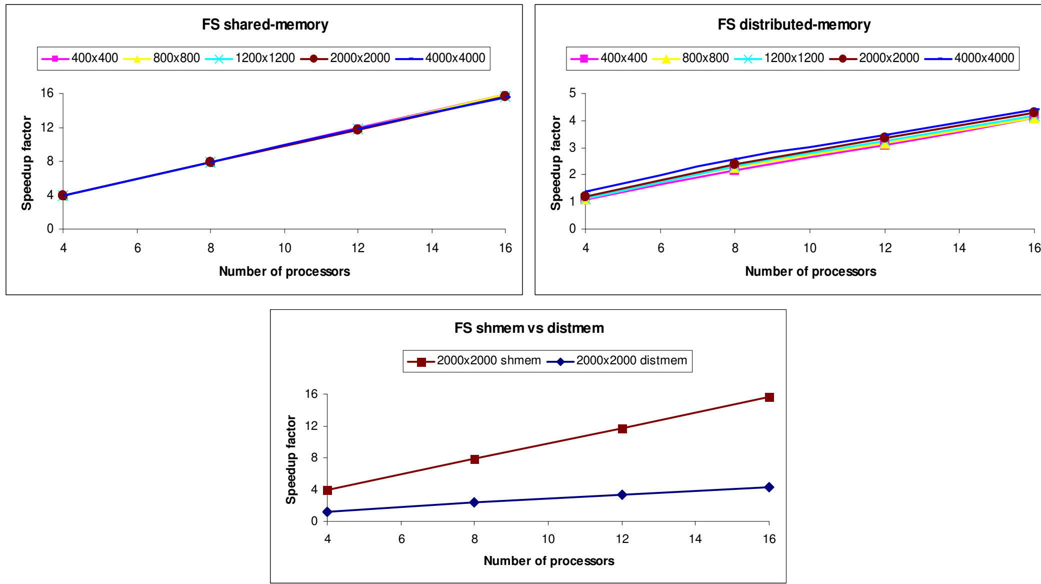


Figure 5.11: Speedup comparison for Floyd-Steinberg

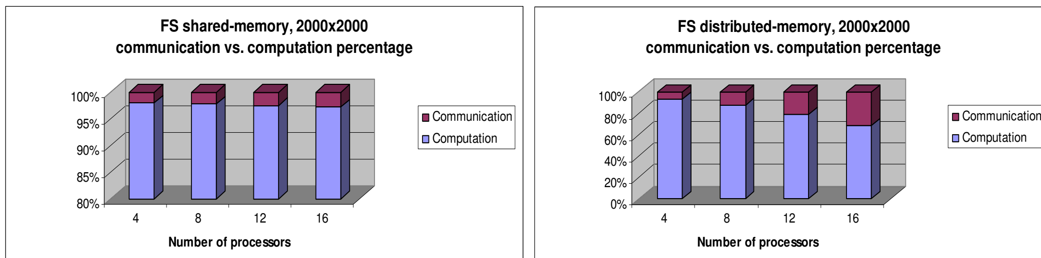


Figure 5.12: Communication vs. computation percentages for Floyd-Steinberg

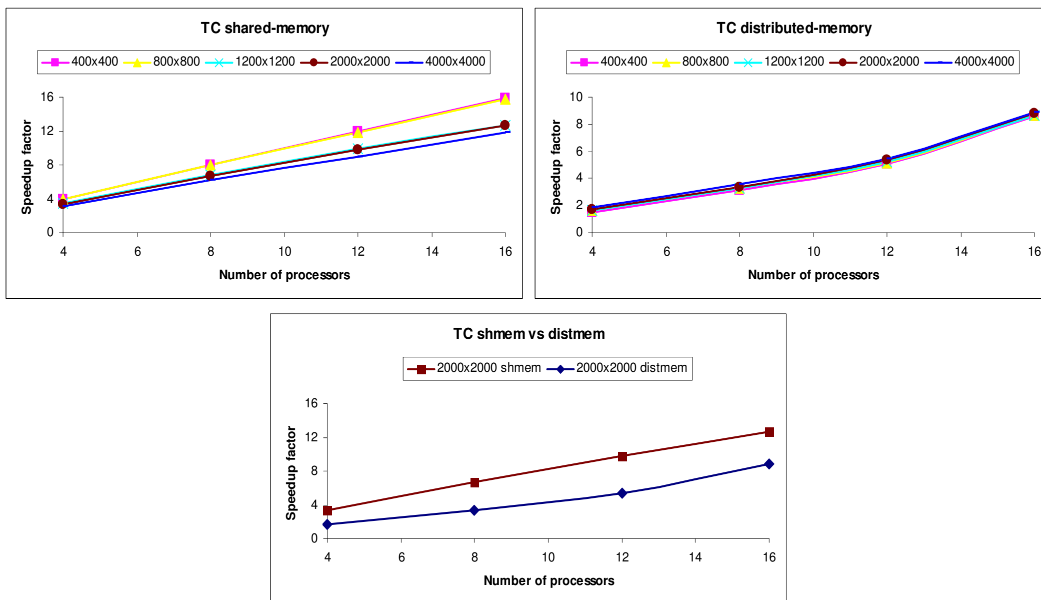


Figure 5.13: Speedup comparison for Transitive Closure

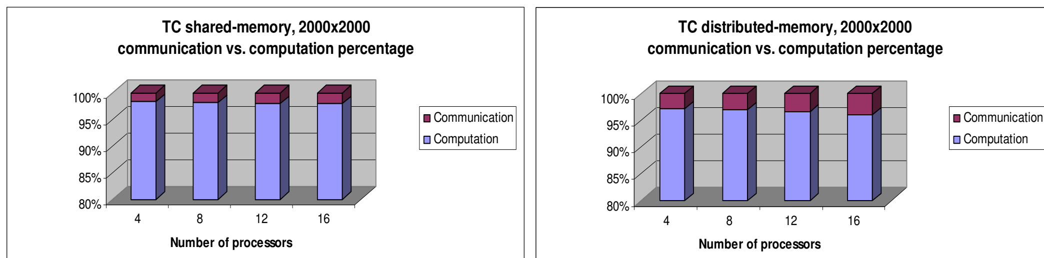


Figure 5.14: Communication vs. computation percentages for Transitive Closure

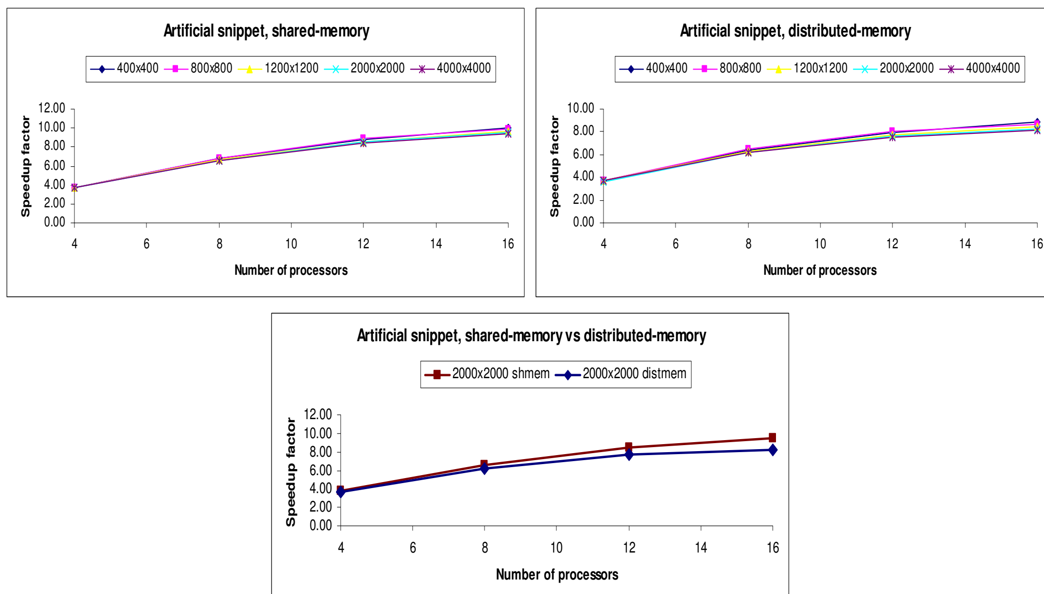


Figure 5.15: Speedup comparison for artificial code snippet

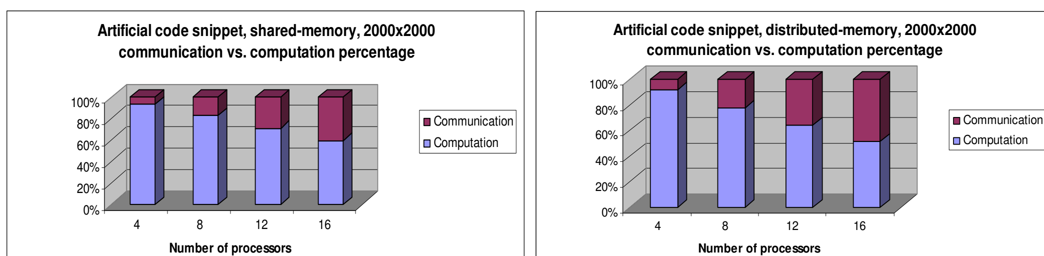


Figure 5.16: Communication vs. computation percentages for artificial code snippet

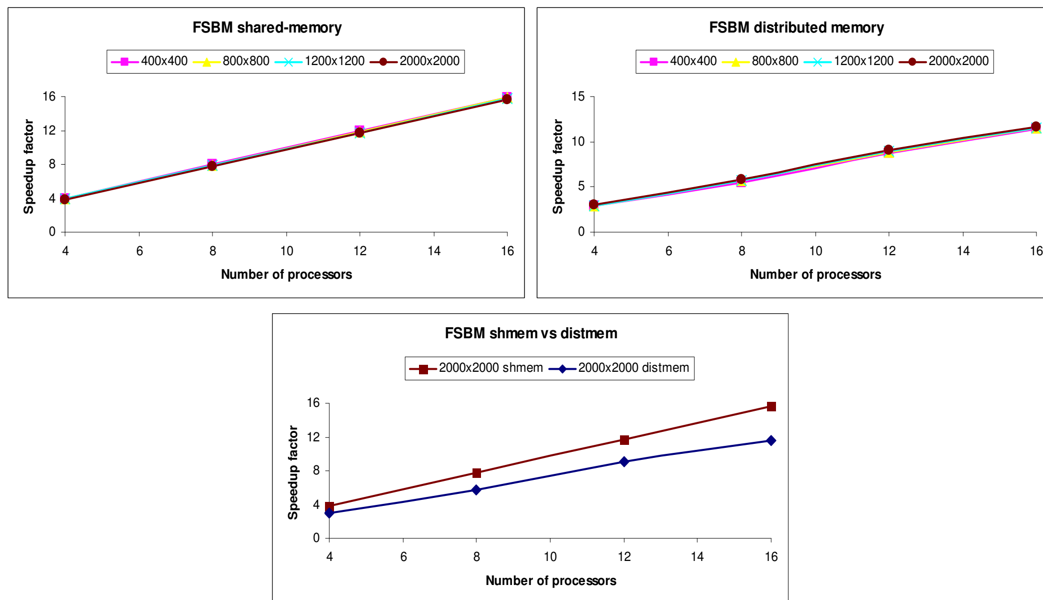


Figure 5.17: Speedup comparison for FSBM

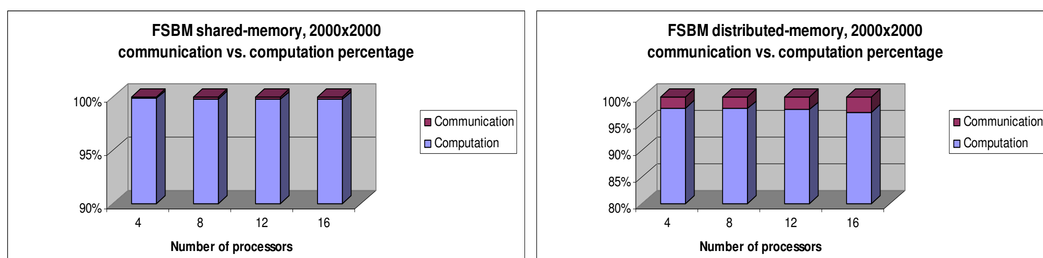


Figure 5.18: Communication vs. computation percentages for FSBM

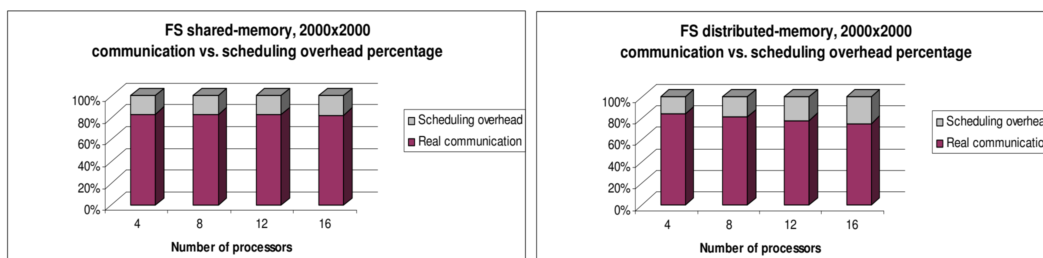


Figure 5.19: Real communication vs. scheduling overhead percentage for Floyd-Steinberg

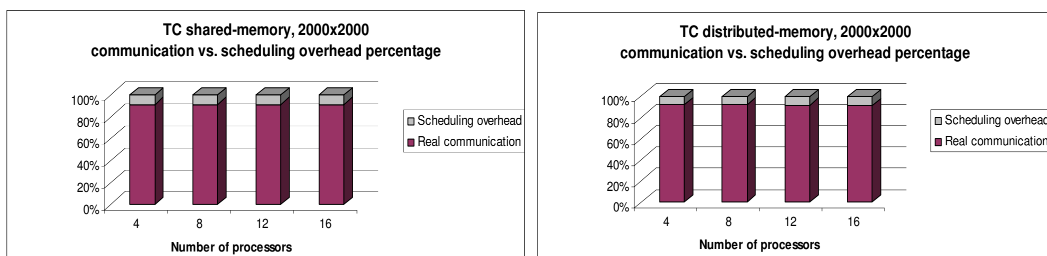


Figure 5.20: Real communication vs. scheduling overhead percentage for Transitive Closure

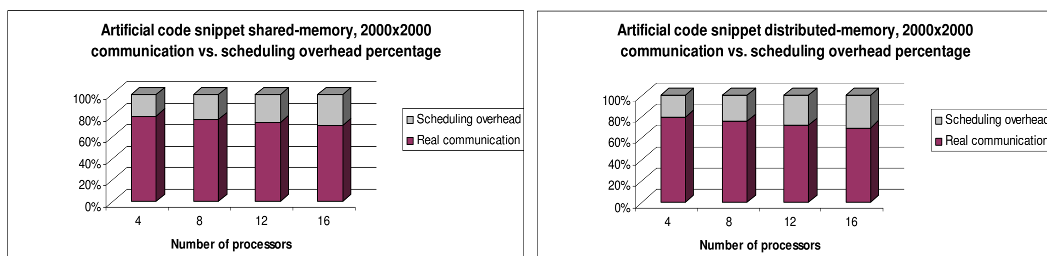


Figure 5.21: Real communication vs. scheduling overhead percentage for artificial code snippet

memory system, and 1.64 (out of 4) to 8.70 (out of 16) in the distributed-memory system. As before, the speedup obtained on the shared-memory system is plotted against the speedup obtained on the distributed-memory system, for a 2000×2000 index space (Fig. 5.13, last chart). Fig. 5.14 shows the communication and computation percentages of this example on both systems. One can see that the improvement in communication percentage of the shared-memory system over the distributed-memory system is not as obvious as in the Floyd-Steinberg case. This is due to the nature of the problem, which is more computationally-intensive. In Fig. 5.20 we show how much of the measured communication time from Fig. 5.14 is spent on real communication (i.e., MPI primitives) and how much is attributed to the scheduling overhead.

Fig. 5.15 gives the results for the artificial code snippet; the speedup of the parallel program ranges from 3.74 (out of the ideal 4) to 9.54 (out of the ideal 16) in the shared-memory system, and 3.66 (out of 4) to 8.27 (out of 16) in the distributed-memory system. As before, the speedup obtained on the shared-memory system is plotted against the speedup obtained on the distributed-memory system, for a 2000×2000 index space (Fig. 5.15, last chart). Fig. 5.16 shows the communication and computation percentages of this example on both systems. One can see that the improvement in communication percentage of the shared-memory system over the distributed-memory system is fairly obvious. This is due to the fact that the artificial code snippet has more dependence vectors than the rest of the test cases, and benefits more when executed on a shared-memory system. In Fig. 5.21 we show how much of the measured communication time from Fig. 5.14 is spent on real communication (i.e., MPI primitives) and how much is attributed to the scheduling overhead.

Fig. 5.17 depicts the speedup of the FSBM parallel code, which results in speedup of 3.92 (out of 4) to 15.79 (out of 16) in the shared-memory case, and 2.93 (out of 4) to 11.59 (out of 16) in the distributed-memory case. Again, the results for the 2000×2000 index space are compared (Fig. 5.17, last chart). One can see there is medium performance gain between the shared- and distributed-memory systems for this particular test case. This is due to the

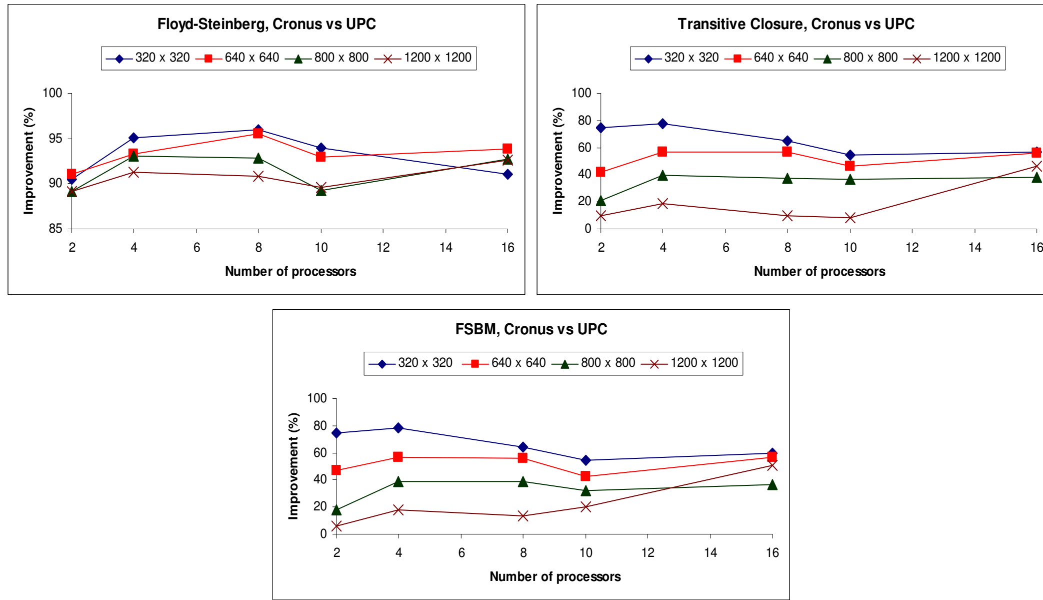


Figure 5.22: Performance comparison of parallel codes generated by CRONUS and UPC

fact that FSBM does not have any loop-carried dependencies. The difference in performance is caused mainly by the characteristics of the system, and not only by the interconnection network. This can be seen in Fig. 5.18.

Comparative results of CRONUS vs. UPC for FS, TC and FSBM, respectively, are summarized in Fig. 5.22. All test cases were tested on a distributed memory cluster of workstations. We measured the improvement of CRONUS over UPC according to the following formula:

$$\frac{T_{UPC} - T_{CRONUS}}{T_{UPC}} \tag{5.3}$$

where T_{CRONUS} and T_{UPC} are the parallel times measured with CRONUS and UPC, respectively.

Fig. 5.22 gives the improvement percentage vs. the number of processors used. It is clear that CRONUS outperforms UPC in all test cases. This figure shows that a common pattern emerges in every test case: as the number of processors increases, the improvement percentage converges to a constant number. In particular, it converges to 90% for Floyd-Steinberg, to 50% for Transitive Closure and to 50% for FSBM.

III. Interpretation of the Results

For every test case we experimented with different index space sizes and different number of processors. The results confirm our prediction that the shared-memory system outperforms in all cases the distributed-memory systems (by a factor ranging from 6% to 60%). This is justified by the fact that the communication overhead is minimized on shared-memory systems. Another conclusion that can be drawn from the results is that the weight of the problem's loop body plays a major role in the performance of the parallel code, regardless of the architecture. The superior performance of the FSBM parallel code is due to the fact that it has the heaviest loop body of the three case studies, whereas the performance of the Floyd-

Steinberg parallel code is the poorest of all three due to its very light loop body. Moreover, the Floyd-Steinberg parallel code with UPC support performed much worse than the other two test cases, which is again attributed to its light loop body and intensive communication.

Our philosophy is that simplicity and efficiency are the key factors for minimizing the runtime of the parallel program. In our approach, the compilation time is kept to a minimum because the index space is not traversed and the issue of what iteration to compute next is solved at runtime. We decided to make this trade-off because the successor concept is very simple and efficient, and it does not incur a large time penalty. This penalty is significantly small when compared to the computationally intensive loop bodies we target.

The conclusions that can be drawn regarding dynamic scheduling are:

- Making scheduling decisions at run time for loops with dependencies is possible and efficient
- Even with blocking send/receive communication primitives, the results are very close to the ideal speedup
- Using blocking communication has the advantage of accuracy of the data exchanged

5.3.3 Evaluation of the DMPS($\mathcal{A}$) Scheme

DMPS($\mathcal{A}$) is implemented in C and relies on the distributed programming framework offered by the `mpich 1.2.6` implementation of the Message Passing Interface (MPI) [Pac97] for exchanging messages.

We used a heterogeneous distributed system that consists of 10 computers, and selected one of them as *master*. In particular, we used: (a) 4 Intel Pentiums III 1266MHz with 1GB RAM (called '*zealots*'), assumed to have $VP_k = 1.5$ (one chosen to be master); and (b) 6 Intel Pentiums III 500MHz with 512MB RAM (called '*kids*'), assumed to have $VP_k = 0.5$. The virtual power for each machine type was determined as a ratio of processing times established by timing a test program on each machine type. The machines are interconnected by a Fast Ethernet, with a bandwidth of 100 Mbits/sec.

We present two cases, *dedicated* and *non-dedicated*. In the first case, processors are dedicated to running the program and no other loads are interposed during the execution. We perform measurements with up to 9 slaves. In the second case, at the beginning of the execution of the program, we start a resource expensive process on some of the slaves. Due to the fact that scheduling algorithms for DOACROSS loops are usually static and no other dynamic algorithms have been reported so far, we are not able to compare with similar algorithms. We perform three series of experiments for the dedicated and non-dedicated case: (1) $DMPS(CSS)$, (2) $DMPS(TSS)$, and (3) $DMPS(DTSS)$ and compare the results for two real-life case studies. We run the above series for $m = 3, 4, 5, 6, 7, 8, 9$ slaves in order to compute the speedup. We compute the **speedup** according to the following equation:

$$S_p = \frac{min\{T_{P_1}, T_{P_2}, \ldots, T_{P_m}\}}{T_{PAR}} \tag{5.4}$$

where T_{P_i} is the serial execution time on slave P_i, $1 \leq i \leq m$, and T_{PAR} is the parallel execution time (on m slaves). Note that in the plotting of S_p, we use VP instead of m on the x-axis.

I. Experiments and Results

We used the Heat Conduction computation for a domain of 5000×10000 grid points, and the Floyd-Steinberg error diffusion computation for a image of 10000×20000 pixels, on a system consisting of 9 heterogeneous slave machines and one master, with the following configuration: zealot1 (master), zealot2, kid1, zealot3, kid2, zealot4, kid3, kid4, kid5, kid6. For instance, when using 6 slaves, the machines used are: zealot1 (master), zealot2, *kid1*, zealot3, *kid2*, zealot4, *kid3*. The slaves in italics are the ones loaded in the non-dedicated case. As mentioned previously, by starting a resource expensive process on these slaves, their ACP is halved.

An illustration of the dependence patterns for the Heat Conduction and Floyd-Steinberg applications is given in Fig. 5.23. The iterations in a chunk are executed in the order imposed by the dependencies of each application. Whenever a synchronization point is reached, data is exchanged between the processors executing neighboring chunks.

Table 5.3 shows comparative results we obtained for the heat equation, for the three series of experiments: $DMPS(CSS)$, $DMPS(TSS)$ and $DMPS(DTSS)$, on a dedicated and a non-dedicated heterogeneous cluster. The values represent the parallel times (in seconds) for different number of slaves. Three synchronization intervals were chosen by the user, and the total ACP ranged according to the number of slaves from 3.5–7.5.

Fig. 5.24 presents the speedups for the heat conduction equation on a grid of 5000×10000 points, for one time step (i.e. `loop`=1), for chunks sizes computed with CSS, TSS and DTSS and synchronization interval 150, on a dedicated cluster and a non-dedicated cluster.

Comparative results for the Floyd-Steinberg case study on a dedicated and a non-dedicated heterogeneous cluster are given in Table 5.4. The values represent the parallel times (in seconds) for different number of slaves. Three synchronization intervals were chosen by the user, and the total ACP ranged according to the number of slaves from 3.5–7.5.

Fig. 5.25 presents the speedup results of the Floyd-Steinberg algorithm, for the three variations. The size of the 2D frame was 10000×20000 pixels. Chunks sizes were computed with CSS, TSS and DTSS and synchronization interval chosen to be 100, on a dedicated cluster and a non-dedicated cluster.

II. Interpretation of the Results

As expected, the results for the dedicated cluster are much better for both case studies. In particular, $DMPS(TSS)$ seems to perform slightly better than $DMPS(CSS)$. This was expected since TSS provides better load balancing than CSS for simple parallel loops without dependencies. In addition, $DMPS(DTSS)$ outperforms both algorithms. This is because it explicitly accounts for the heterogeneity of the slaves. For the non-dedicated case, one can see that $DMPS(CSS)$ and $DMPS(TSS)$ cannot handle workload variations as effectively as $DMPS(DTSS)$. This is shown in Fig. 5.24. The speedup for $DMPS(CSS)$ and $DMPS(TSS)$ decreases as loaded slaves are added, whereas for $DMPS(DTSS)$ it increases even when slaves are loaded. In the non-dedicated approach, our choice was to load the slow processors, so as to incur large differences between the processing power of the two machine types. Even in this case, $DMPS(DTSS)$ achieved good results.

The ratio $\frac{commputation}{communication}$ along with the selection of the synchronization interval play a key role in the overall performance of our scheme. A rule of thumb is to maintain this ratio ≥ 1 at all times. The choice of the (fixed) synchronization interval has a crucial impact on the performance, and it is dependent on the concrete problem. h must be chosen so as to ensure

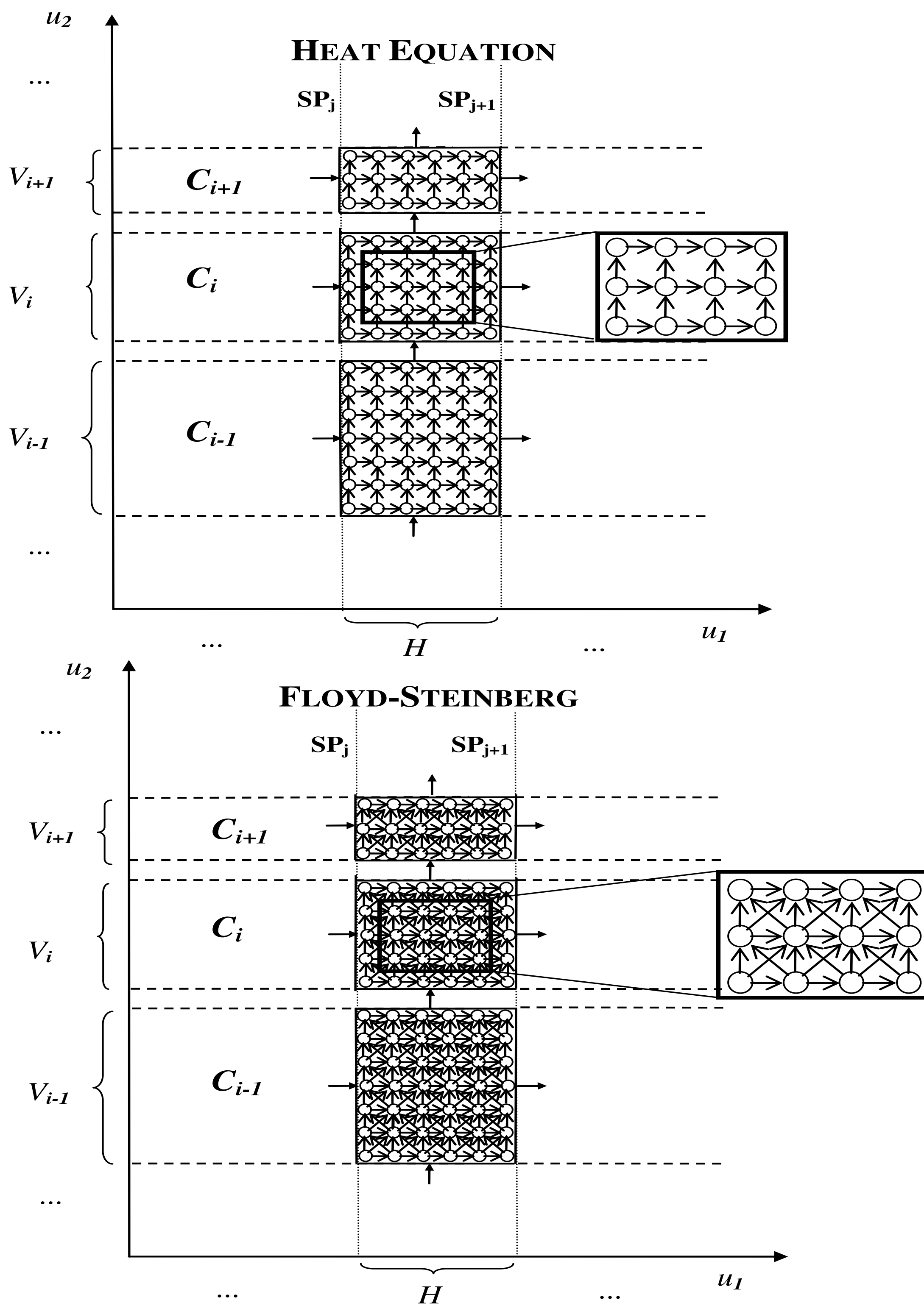


Figure 5.23: Dependence patterns for Heat Conduction and Floyd-Steinberg.

Table 5.3: Parallel execution times (sec) for Heat Conduction equation on a dedicated & non-dedicated heterogeneous cluster

Sync. interval	Dedicated	Number of slaves						
		3	4	5	6	7	8	9
100	DMPS(CS)	2.32	1.75	1.73	1.23	1.21	1.21	1.18
	DMPS(TSS)	2.20	1.73	1.56	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.14	1.00	0.95	0.91	0.85	0.78
150	DMPS(CS)	2.31	1.74	1.71	1.21	1.22	1.21	1.18
	DMPS(TSS)	2.18	1.72	1.54	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.13	0.99	0.93	0.90	0.84	0.78
200	DMPS(CS)	2.30	1.74	1.73	1.22	1.23	1.22	1.19
	DMPS(TSS)	2.21	1.74	1.55	1.38	1.25	1.14	1.02
	DMPS(DTSS)	1.42	1.13	0.99	0.94	0.90	0.83	0.78

Sync. interval	Non-dedicated	Number of slaves						
		3	4	5	6	7	8	9
100	DMPS(CS)	2.33	1.76	1.73	2.46	2.45	2.38	2.06
	DMPS(TSS)	2.20	1.74	1.56	2.52	2.56	2.18	2.10
	DMPS(DTSS)	1.95	1.45	1.30	1.31	1.33	1.38	1.25
150	DMPS(CS)	2.33	1.74	1.72	2.46	2.49	2.43	2.05
	DMPS(TSS)	2.19	1.72	1.54	2.42	2.23	2.31	2.06
	DMPS(DTSS)	1.94	1.47	1.30	1.30	1.28	1.36	1.23
200	DMPS(CS)	2.30	1.74	1.73	2.39	2.36	2.38	2.10
	DMPS(TSS)	2.22	1.75	1.56	1.79	2.32	2.10	2.02
	DMPS(DTSS)	1.96	1.44	1.29	1.29	1.27	1.32	1.21

the ratio $\frac{commputation}{communication}$ is maintained above 1, even when V_i decreases at every scheduling step. Assuming that for a certain h, $\frac{commputation}{communication} \geq 1$ is satisfied, small changes in the value of h do not significantly alter the overall performance. Notice that the best synchronization interval for the heat equation was $h = 150$, whereas for the Floyd-Steinberg better results were obtained for $h = 100$. The performance differences for interval sizes close to the ones

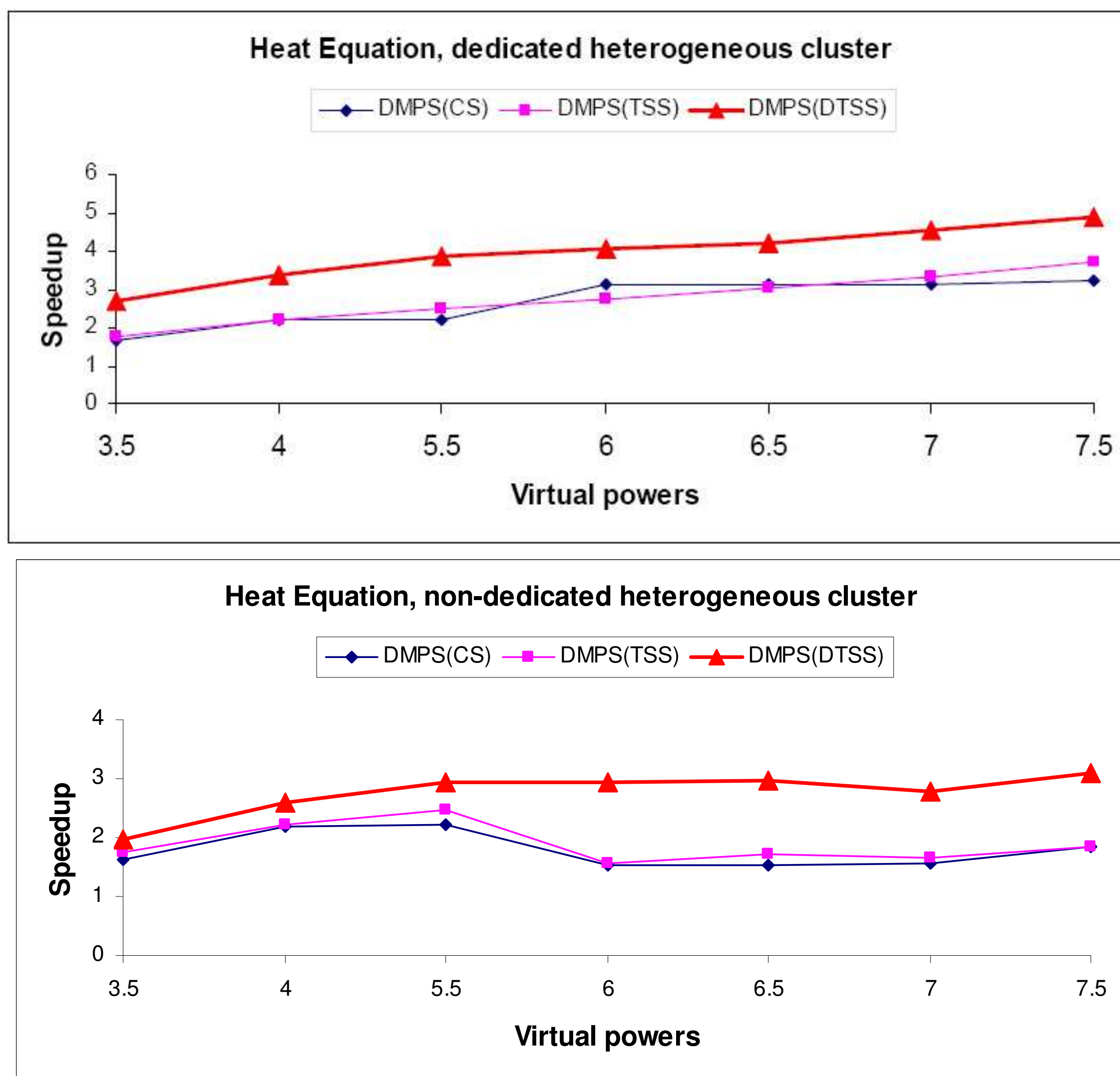


Figure 5.24: Speedups for the heat equation on a dedicated & non-dedicated heterogeneous cluster

depicted in Fig. 5.24 and 5.25 are small.

The main contribution of this section is extending three previous schemes by taking into account the existing iteration dependencies of the problem, and hence providing a scheme for inter-slave communication. The method was tested with two real-life applications: Heat Conduction equation and Floyd-Steinberg dithering. The results demonstrate that this new scheme is effective for distributed applications with dependence loops.

5.3.4 Evaluation of $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{SW}$

The self-scheduling schemes, along with the $\mathcal{S}$, $\mathcal{W}$ and $\mathcal{SW}$ mechanisms were implemented in C, and the communication between the master and slaves, as well as the inter-slave communication were implemented using MPI. The experiments were performed on a heterogeneous Linux cluster of 13 nodes (1 master and 12 slaves). The cluster consists of two machine types: (a) 7 Intel Pentiums III 800MHz with 256MB RAM (called '*twins*'), with virtual

Table 5.4: Parallel execution times (sec) for Floyd-Steinberg on a dedicated & non-dedicated heterogeneous cluster

Sync. interval	Dedicated	Number of slaves						
		3	4	5	6	7	8	9
50	DMPS(CS)	27.79	22.14	16.78	16.69	16.53	11.38	11.36
	DMPS(TSS)	25.32	19.77	17.30	15.41	13.80	12.43	11.40
	DMPS(DTSS)	19.63	14.87	13.28	12.72	11.57	11.45	10.73
100	DMPS(CS)	27.52	22.01	16.70	16.65	16.43	11.34	11.33
	DMPS(TSS)	25.22	19.70	17.24	15.35	13.75	12.38	11.38
	DMPS(DTSS)	19.63	14.80	13.21	12.66	11.52	11.34	10.64
150	DMPS(CS)	27.58	22.03	16.75	16.70	16.44	11.43	11.43
	DMPS(TSS)	25.22	19.70	17.22	15.34	13.75	12.39	11.38
	DMPS(DTSS)	19.62	14.82	13.24	12.67	11.53	11.34	10.65

Sync. interval	Non-dedicated	Number of slaves						
		3	4	5	6	7	8	9
50	DMPS(CS)	27.72	22.13	16.76	23.81	22.32	22.47	22.44
	DMPS(TSS)	25.18	19.72	17.24	22.34	24.14	22.26	20.95
	DMPS(DTSS)	21.88	16.06	14.38	13.74	13.26	13.02	11.71
100	DMPS(CS)	27.49	21.99	16.67	22.61	22.42	22.59	22.35
	DMPS(TSS)	25.18	19.66	17.17	19.23	24.15	22.24	20.88
	DMPS(DTSS)	21.85	15.96	14.32	13.65	13.11	12.80	11.58
150	DMPS(CS)	27.57	22.01	16.74	22.49	22.48	22.32	22.46
	DMPS(TSS)	25.17	19.65	17.20	26.20	24.14	22.02	20.82
	DMPS(DTSS)	21.86	15.96	14.31	13.58	13.18	12.80	11.59

power $VP_k = 1$; and (b) 6 Intel Pentiums III 500MHz with 512MB RAM (called '*kids*'), with virtual power $VP_k = 0.8$. In order to obtain the virtual power of each slave we ran 10 times a test problem (which involved nested loops with floating point operations) serially on each computer and averaged the measured execution times. Although this is a simple model, it is appropriate for the type of applications we study, namely nested loops with floating point operations. The machines are interconnected by a 100Mbits/sec Fast Ethernet network.

We experiment on the *non-dedicated* cluster. In particular, at the beginning of execution, we start a resource expensive process on some of the slaves, which halves their ACP. We ran three series of experiments:

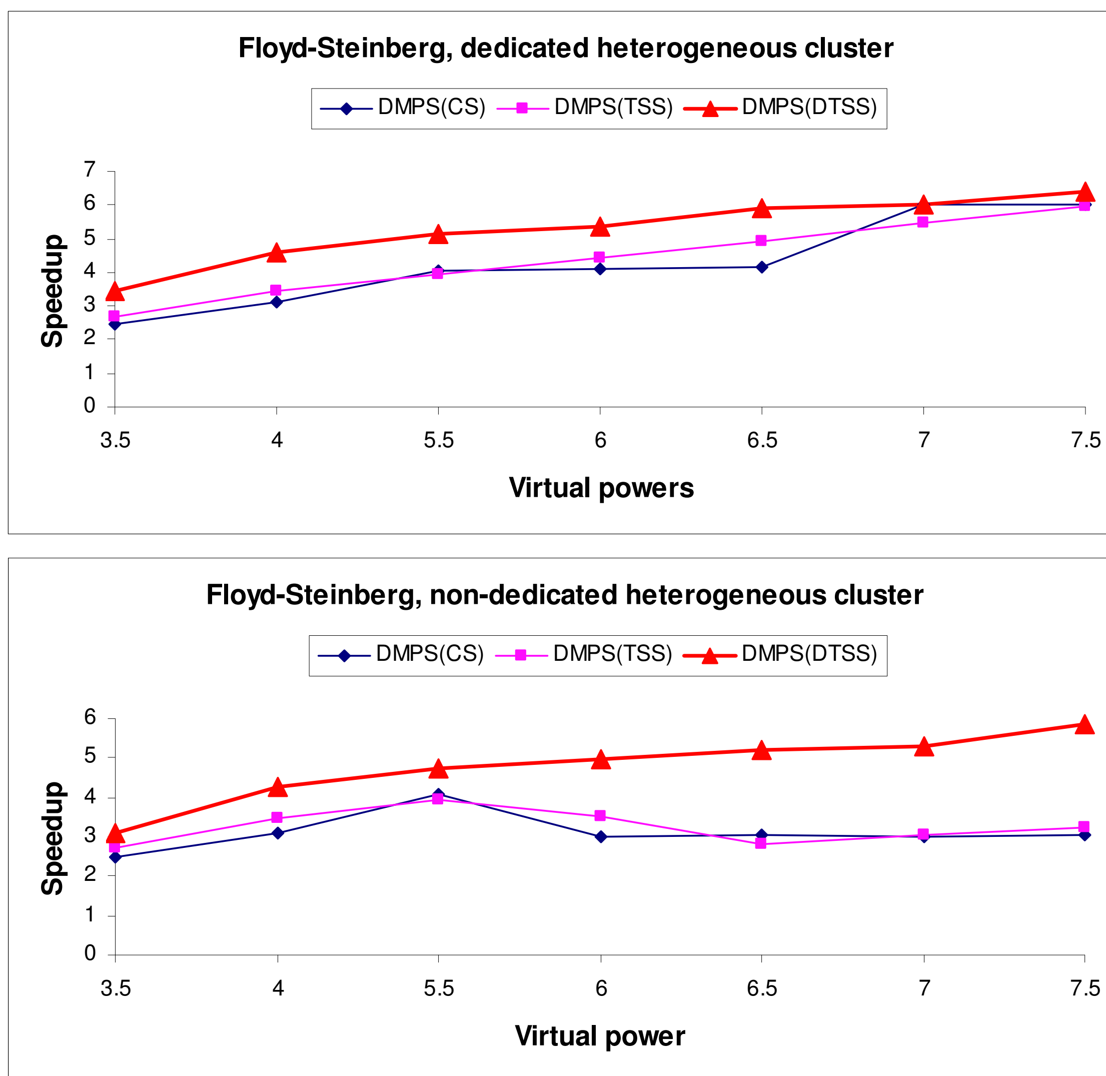


Figure 5.25: Speedups for Floyd-Steinberg on a dedicated & non-dedicated heterogeneous cluster

#1: for the synchronization mechanism ($\mathcal{S}$)

#2: for the weighting mechanism ($\mathcal{W}$)

#3: for the combined mechanisms ($\mathcal{SW}$).

We ran the above series for $m = 4, 6, 8, 10, 12$ slaves. The results given in the following subsections are the average of 10 runs for each experiment. We used the following machines: twin1(master), twin2, **kid1**, twin3, **kid2**, twin4, **kid3**, twin5, **kid4**, twin6, **kid5**, twin7, **kid6**. In all cases, the overloaded machines were the **kids** (written in boldface). When $m = 4$, the first four machines are used from the above list, when $m = 8$ the first eight machines are used, and so on.

Table 5.5: Problem sizes for Floyd-Steinberg and Hydro test cases

Problem size	small	medium	large
Floyd-Steinberg	5000×15000	10000×15000	15000×15000
Upper/lower threshold	500/10	750/10	1000/10
Hydro	$5000 \times 5 \times 10000$	$7500 \times 5 \times 10000$	$10000 \times 5 \times 10000$
Upper/lower threshold	500/10	750/10	1000/10

Experiment #1

For the first series, we experimented on two real-life applications: Floyd-Steinberg and Hydro. For each application we used three problem sizes, given in Table 5.5, in order to perform a scalability analysis with respect to the problem size. In each case, U_s was taken as the largest dimension, whereas U_c was taken as the second largest dimension. For each of these applications we compared the parallel execution times against the serial execution times.

We scheduled each test case using the five self-scheduling algorithms, i.e., CSS, FSS, GSS, TSS and $\mathcal{W}$-TSS, to which we applied the synchronization mechanism $\mathcal{S}$. The algorithms were implemented as described in Section 4.4.2. For CSS we used the chunk size $\frac{U_c}{2*m}$ in order to ensure that slaves receive work twice. To avoid excessive synchronization overheads and large idle times we used lower and upper bounds (see Table 5.5) on the size of the chunks given by the self-scheduling algorithms.

The synchronization interval h for both test cases is given by formula (4.1), where M $= 3 * m$ was used. In order to show that M must be at least $3 * m$ we ran multiple tests with different values for M, ranging from $1*m$ to $10*m$, in order to assess the impact of the chosen M on the parallel execution times. The results for both applications with dependencies are given in Figure 5.26. One can see that a good performance can be obtained if M$\geq 3 * m$.

Both the Floyd-Steinberg and Hydro applications have unitary dependence vectors, i.e., unitary projection lengths along the chunk dimension u_c. This yields a relatively small volume of communication, making easy to maintain the ratio $\frac{communication}{computation}$ well below 1. The results for all problem sizes are shown in Fig. 5.27.

We plotted the parallel and serial times vs the virtual power of the cluster. With 4 slaves VP is 3.6, with 6 slaves VP is 5.4, with 8 slaves is 7.2, with 10 slaves VP is 9 and with 12 slaves is 10.8. The serial time was measured on the fastest slave type, i.e., twin.

The results show that the synchronization mechanism can be applied to all existing self-scheduling algorithms, yielding their synchronized–only versions, which can efficiently parallelize real-life DOACROSS loops. One can notice that $\mathcal{S}$-CSS, $\mathcal{S}$-FSS, $\mathcal{S}$-GSS, and $\mathcal{S}$-TSS give significant speedups over the serial execution (see Table 5.6), proving the efficiency of the transformed algorithms. In all cases, $\mathcal{SW}$-TSS, which explicitly accounts for system's heterogeneity, shows an even greater speedup over all synchronized–only algorithms. The two charts at the bottom of Fig. 5.27 illustrate the serial and parallel times on 12 processors for both applications. The serial times increase faster than the parallel times as the problem size increases in both cases. This implies that the larger the problem size, the more processors can be effectively employed in the parallel execution. This yields larger speedups for larger problem sizes. This was anticipated since with larger index spaces and for the same task granularity, a greater degree of parallelism becomes available.

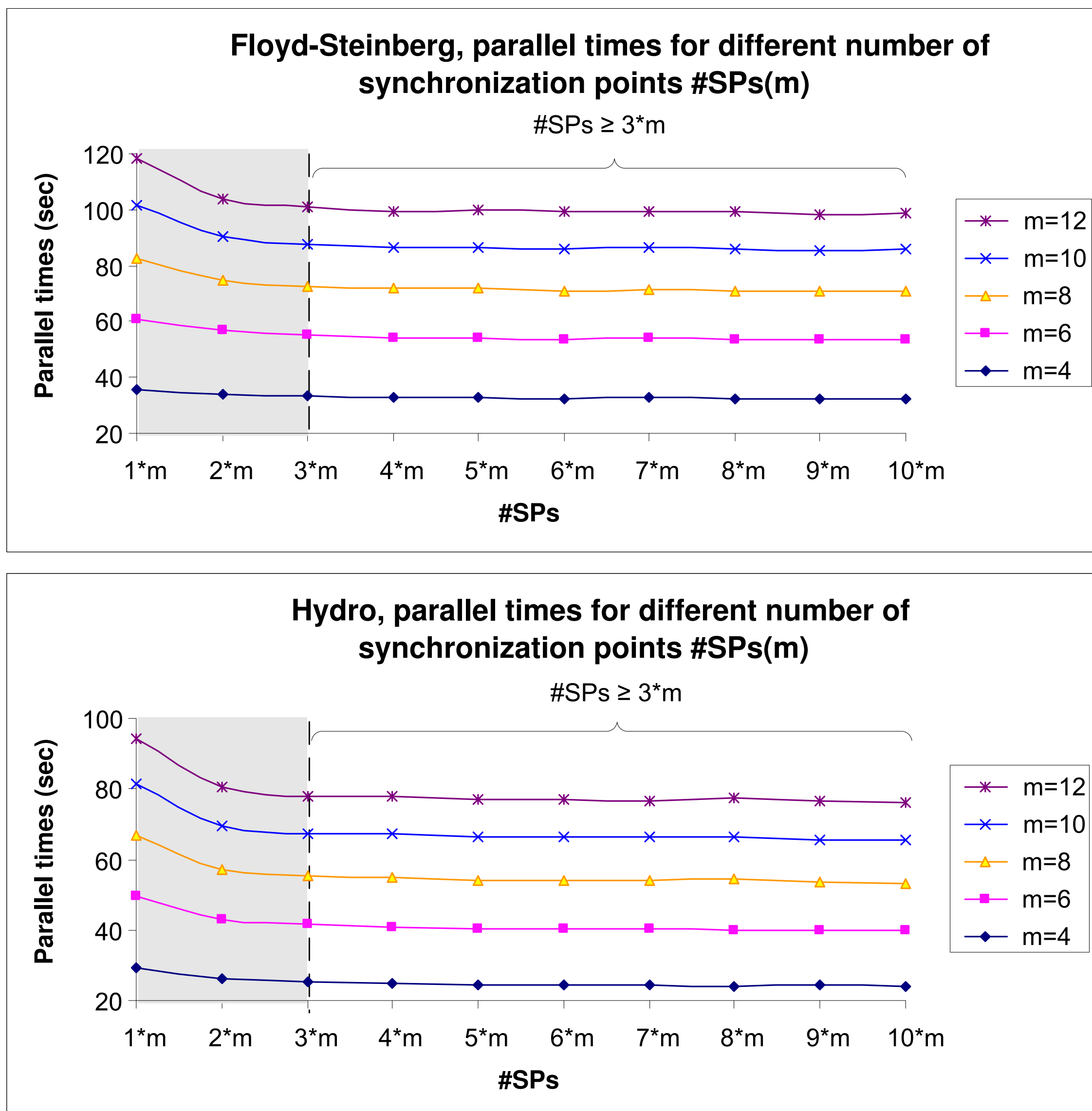


Figure 5.26: Impact of M on the parallel execution times for Floyd-Steinberg and Hydro case studies

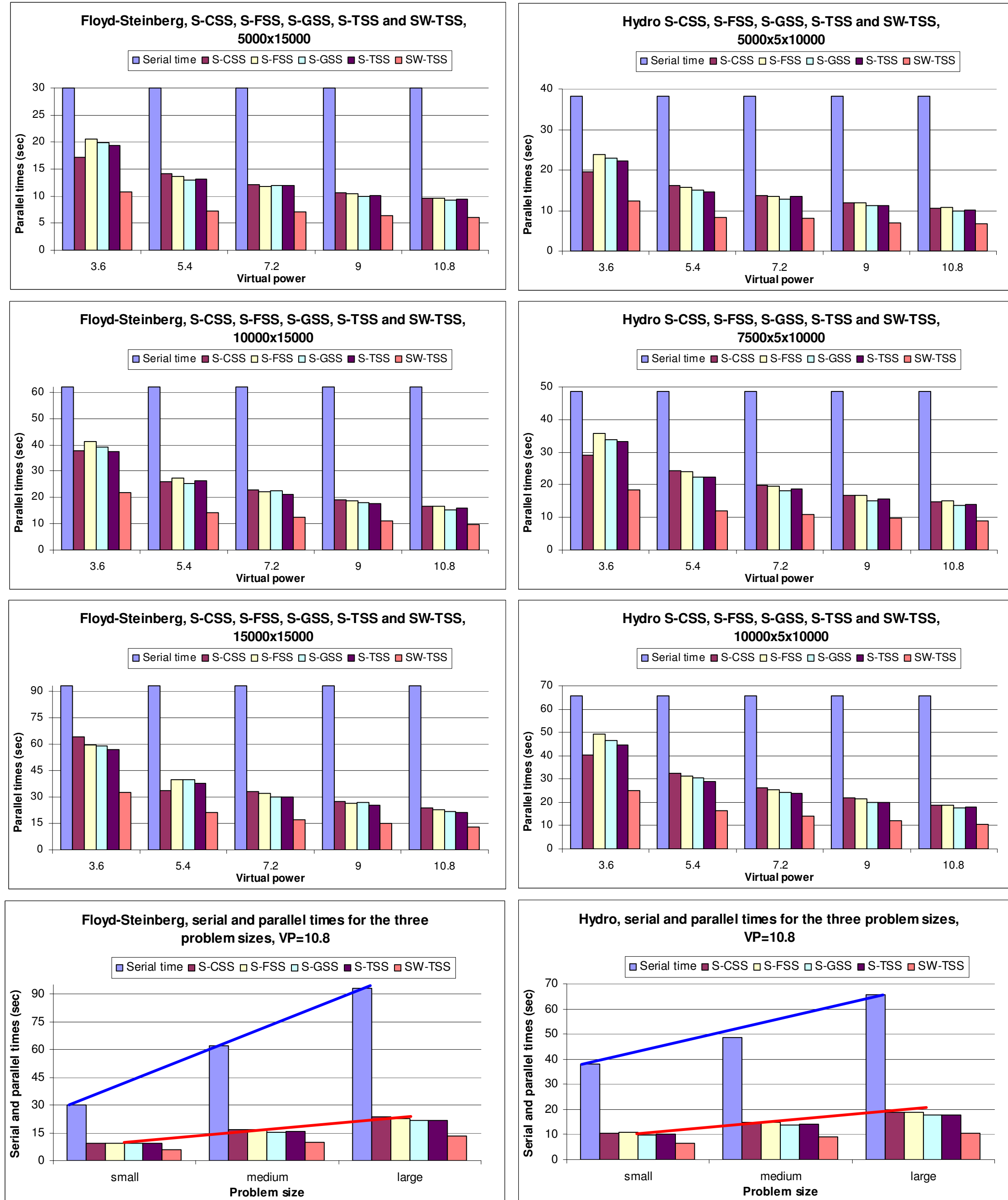


Figure 5.27: Parallel times of the synchronized–only algorithms for Floyd-Steinberg and Hydro case studies

Table 5.6: Speedups for Floyd-Steinberg and Hydro test cases

Test case	VP	$\mathcal{S}$-CSS	$\mathcal{S}$-FSS	$\mathcal{S}$-GSS	$\mathcal{S}$-TSS	$\mathcal{SW}$-TSS
Floyd-Steinberg	**3.6**	1.45	1.57	1.59	1.63	2.86
	5.4	2.76	2.35	2.33	2.47	4.35
	7.2	2.81	2.92	3.09	3.10	5.39
	9	3.41	3.50	3.49	3.70	6.27
	10.8	3.95	4.07	4.27	4.34	7.09
Hydro	**3.6**	1.64	1.33	1.42	1.47	2.61
	5.4	2.02	2.10	2.16	2.28	4.04
	7.2	2.53	2.57	2.72	2.75	4.73
	9	3.01	3.07	3.33	3.29	5.43
	10.8	3.49	3.49	3.72	3.69	6.16

Experiment #2

For the second series of experiments, we used the Mandelbrot fractal computation algorithm (a DOALL loop), on the domain [-2.0, 1.25] $\times$ [-1.25, 1.25], for different window sizes: 7500×10000, 10000×10000 and 12500×12500. The computation involves unpredictably irregular loop tasks. We scheduled the Mandelbrot set with CSS, FSS, GSS, TSS and compared their performance against their weighted versions $\mathcal{W}$-CSS, $\mathcal{W}$-FSS, $\mathcal{W}$-GSS, $\mathcal{W}$-TSS. In this series, the weighting mechanism is expected to facilitate the scheduling algorithms to distribute the work to slaves more evenly and to improve the overall performance. This is confirmed by the experimental results, depicted in Fig. 5.28. One can see that in all cases the weighted algorithm clearly outperforms the non-weighted algorithm.

The **gain** of the weighted over non-weighted algorithms is also given in Table 5.7, computed as $\frac{T_{\mathcal{A}}-T_{\mathcal{W}-\mathcal{A}}}{T_{\mathcal{A}}}$, where $T_{\mathcal{A}}$ is the parallel time of the non-weighted algorithm $\mathcal{A}$ and $T_{\mathcal{W}-\mathcal{A}}$ is the parallel time of the weighted algorithm $\mathcal{W}$-$\mathcal{A}$. We are also interested in establishing an adequate confidence interval for the performance gain in each case. As it is common in practice, we consider 95% confidence. The algorithm with the least parallel time is $\mathcal{SW}$-FSS for all problem sizes; the performance gain of $\mathcal{SW}$-FSS over FSS ranges from 18% to 53%. With 0.95 probability the performance gain lies in the interval $42 \pm 8\%$. However, the algorithm with the best overall performance gain is GSS, ranging from 50% to 57%. With 0.95 probability the gain lies in the interval $53 \pm 6\%$. As shown in Fig. 5.28, the difference in performance of the weighted algorithms is much smaller than the performance difference between their non-weighted versions.

In order to examine the effect of the weighting mechanism on load balancing, Table 5.8 provides the total computation times of each slave together with the total number of iterations it was assigned, for the computation of the Mandelbrot test-case. The total computation time of each slave is the time it spends performing actual work. In the ideal case, all slaves should have exactly the same computation time, and a large divergence shows great load imbalance.

The data in Table 5.8 correspond to Fig. 5.28 (GSS vs $\mathcal{W}$-GSS, the parallel times obtained for four slaves with total $VP = 3.6$), and shows the difference between the non-weighted and the weighted algorithm. In particular, Table 5.8 analyzes the parallel times of GSS and $\mathcal{W}$-GSS on each of the four slaves and the number of iterations computed by each slave. Note that the parallel time plotted in Fig. 5.28 is close to the computation time of the slowest slave. It is clear that with the non-weighted algorithm the computation times of each slave vary

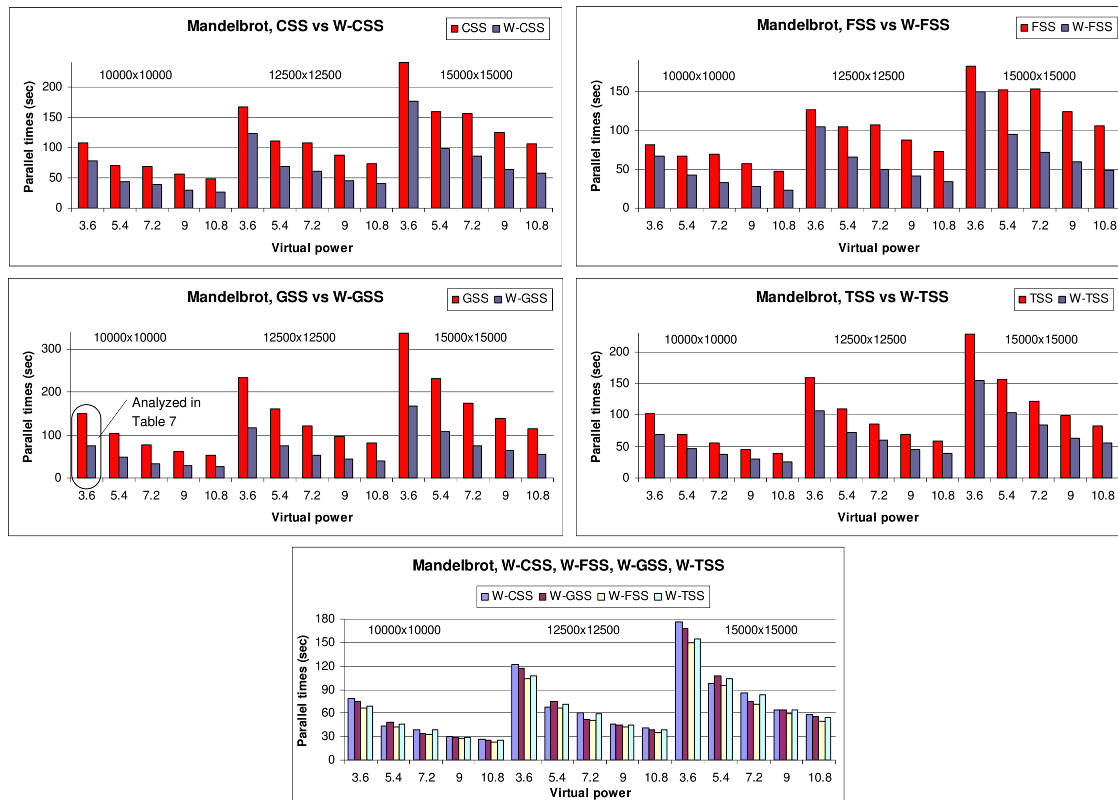


Figure 5.28: Parallel times of the weighted and non-weighted algorithms for Mandelbrot case study

Table 5.7: Gain of the weighted over the non-weighted algorithms for the Mandelbrot test case

Test case	Problem size	VP	$\mathcal{S}$-CSS vs $\mathcal{SW}$-CSS	$\mathcal{S}$-GSS vs $\mathcal{SW}$-GSS	$\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS	$\mathcal{S}$-TSS vs $\mathcal{SW}$-TSS
Mandelbrot	10000 × 10000	**3.6**	27%	50%	18%	33%
		5.4	38%	54%	37%	34%
		7.2	43%	57%	52%	32%
		9	48%	53%	52%	35%
		10.8	43%	52%	52%	34%
	12500 × 12500	**3.6**	27%	50%	18%	33%
		5.4	38%	54%	37%	34%
		7.2	44%	57%	53%	30%
		9	47%	54%	52%	35%
		10.8	44%	52%	53%	34%
	15000 × 15000	**3.6**	27%	50%	18%	33%
		5.4	38%	54%	37%	34%
		7.2	45%	57%	53%	31%
		9	49%	54%	52%	35%
		10.8	46%	52%	54%	33%
Confidence interval (95%)	Overall 42 ± 3 %		40 ± 6 %	53 ± 6 %	42 ± 8 %	33 ± 4 %

Table 5.8: Load balancing in terms of total number of iterations per slave and computation times per slave, GSS vs $\mathcal{W}$-GSS (Mandelbrot test case)

Slave	GSS	GSS	$\mathcal{W}$-GSS	$\mathcal{W}$-GSS
	# Iterations (10^6)	Comp. time (sec)	# Iterations (10^6)	Comp. time (sec)
twin2	56.434	34.63	55.494	62.54
kid1	18.738	138.40	15.528	62.12
twin3	10.528	39.37	15.178	74.63
kid2	14.048	150.23	13.448	61.92

in relation to the computation power and load of the slave, i.e., slow and overloaded slaves have larger total computation times. In the case of the weighted algorithm this variation is reduced significantly. Also, with GSS, even though kid2 and twin2 were assigned 14.048×10^6 and 56.434×10^6 loop iterations respectively, twin2 required 34.63 sec to compute these loop iterations, in contrast with 150.23 sec required by kid2 (this is roughly the parallel time for this case). This led to a huge load imbalance, which deteriorated significantly the algorithm's performance. Unlike GSS, $\mathcal{W}$-GSS execution times for all slaves are about the same which confirms that $\mathcal{W}$-GSS indeed achieves good load balancing.

Experiment #3

For the third series of experiments we repeat the first series, applying now the $\mathcal{W}$ mechanism to all synchronized–only algorithms. In particular, we schedule the Floyd-Steinberg and Hydro test cases with the following synchronized–weighted algorithms: $\mathcal{SW}$-CSS, $\mathcal{SW}$-FSS, $\mathcal{SW}$-GSS and $\mathcal{SW}$-TSS. The results in Fig. 5.29 and 5.30 show that in all cases the synchronized–weighted algorithms clearly outperform their synchronized–only counterparts. One can notice that all $\mathcal{SW}$ algorithms give comparable parallel times.

The above results are also illustrated in Table 5.9, which shows the gain of $\mathcal{SW}$-$\mathcal{A}$ over $\mathcal{S}$-$\mathcal{A}$, computed as $\frac{T_{\mathcal{S}-\mathcal{A}} - T_{\mathcal{SW}-\mathcal{A}}}{T_{\mathcal{S}-\mathcal{A}}}$, where $T_{\mathcal{S}-\mathcal{A}}$ is the parallel time of the synchronized–only algorithm $\mathcal{A}$ and $T_{\mathcal{SW}-\mathcal{A}}$ is the parallel time of the synchronized–weighted algorithm $\mathcal{SW}$-$\mathcal{A}$. Confidence intervals are also given in the same Table, both with respect to every algorithm and overall confidence intervals per test case.

In Table 5.9, we show the algorithm with the highest gain for each application. Subsequently, in Table 5.10 we analyze the gain of $\mathcal{SW}$-CSS over $\mathcal{S}$-CSS for Floyd-Steinberg and of $\mathcal{SW}$-GSS over $\mathcal{S}$-GSS for Hydro. This gain is attributed to better load balancing, which is expressed in terms of the total computation time per slave. In particular: $\mathcal{S}$-CSS and $\mathcal{S}$-GSS, respectively, assigned approximately the same number of iterations to all slaves; this led to larger computation times for the slower slaves (i.e., kids) in comparison to faster slaves (i.e., twins). With $\mathcal{SW}$-CSS and $\mathcal{SW}$-GSS, respectively, the difference in the slaves' computation time is reduced because the number of iterations assigned to each slave has been adjusted according to their available power.

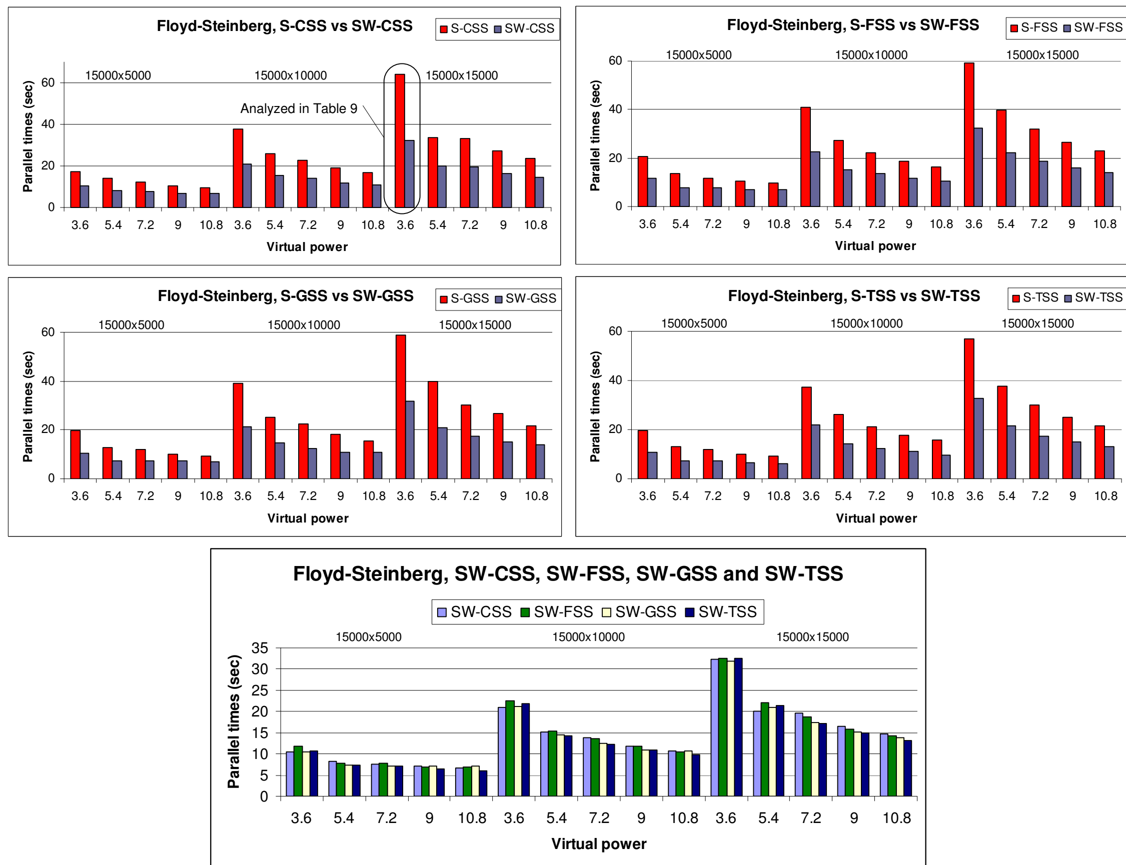


Figure 5.29: Parallel times of the synchronized–weighted and synchronized–only algorithms for Floyd-Steinberg case study, for three different problem sizes: 15000×5000, 15000×10000 and 15000×15000

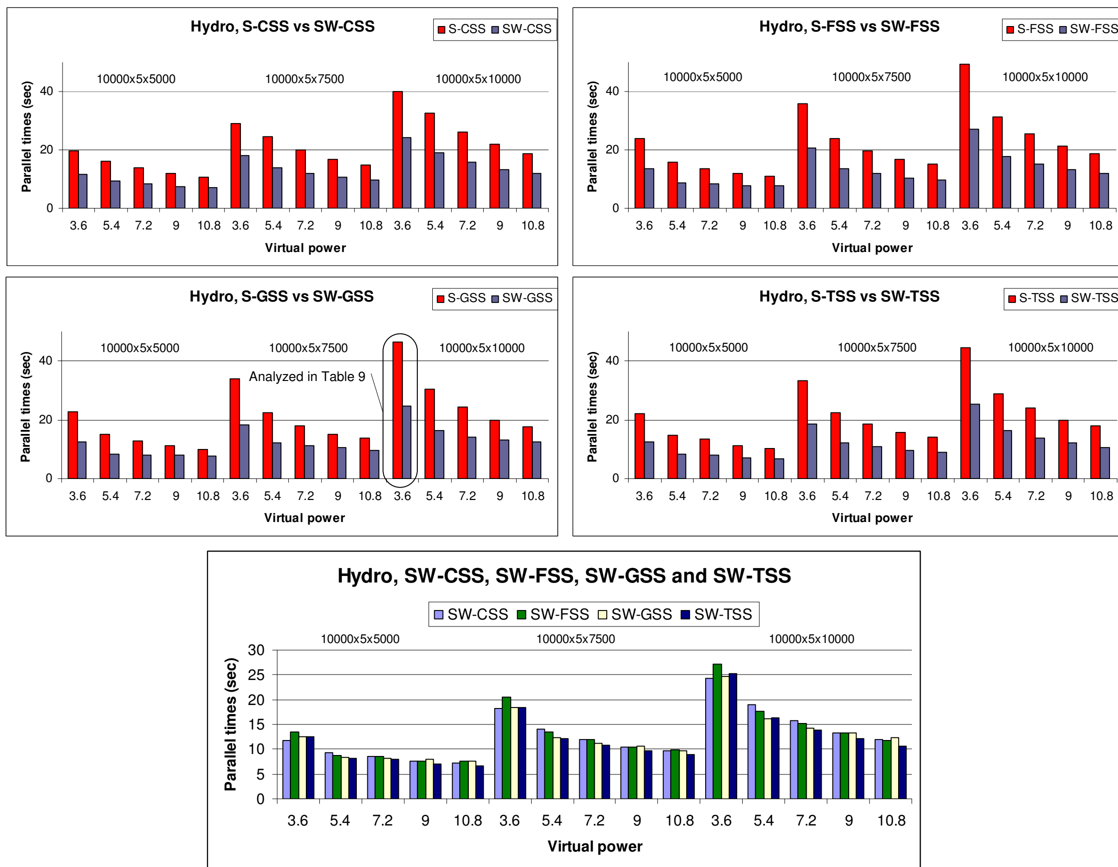


Figure 5.30: Parallel times of the synchronized–weighted and synchronized–only algorithms for Hydro case study, for three different problem sizes $10000 \times 5 \times 5000$, $10000 \times 5 \times 7500$ and $10000 \times 5 \times 10000$

Table 5.9: Gain of the synchronized–weighted over the synchronized–only algorithms for the Floyd-Steinberg and Hydro test cases

Test case	Problem size	**VP**	$\mathcal{S}$-CSS vs $\mathcal{SW}$-CSS	$\mathcal{S}$-GSS vs $\mathcal{SW}$-GSS	$\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS	$\mathcal{S}$-TSS vs $\mathcal{SW}$-TSS
Floyd-Steinberg	15000 × 5000	**3.6**	39%	47%	43%	45%
		5.4	42%	43%	44%	44%
		7.2	37%	40%	35%	40%
		9	34%	27%	34%	36%
		10.8	31%	23%	28%	35%
	15000 × 7500	**3.6**	44%	46%	45%	42%
		5.4	41%	42%	44%	45%
		7.2	39%	45%	38%	42%
		9	37%	40%	37%	38%
		10.8	36%	30%	36%	38%
	15000 × 10000	**3.6**	**50%**	46%	45%	43%
		5.4	41%	48%	44%	43%
		7.2	41%	42%	41%	42%
		9	39%	43%	40%	41%
		10.8	38%	36%	38%	39%
Confidence interval (95%)	Overall 40 ± 1 %		39 ± 2 %	40 ± 3 %	40 ± 2 %	41 ± 2 %
Hydro	10000 × 5 × 5000	**3.6**	40%	46%	43%	44%
		5.4	43%	44%	44%	44%
		7.2	39%	37%	37%	41%
		9	37%	29%	36%	38%
		10.8	32%	23%	29%	34%
	10000 × 5 × 7500	**3.6**	38%	46%	43%	44%
		5.4	43%	45%	44%	46%
		7.2	40%	37%	39%	42%
		9	37%	29%	38%	38%
		10.8	35%	30%	35%	36%
	10000 × 5 × 10000	**3.6**	40%	**47%**	45%	44%
		5.4	42%	47%	44%	43%
		7.2	40%	41%	41%	42%
		9	39%	33%	38%	39%
		10.8	37%	30%	37%	40%
Confidence interval (95%)	Overall 39 ± 1 %		39 ± 2 %	38 ± 4 %	40 ± 2 %	41 ± 2 %

Table 5.10: Load balancing in terms of total number of iterations per slave and computation times per slave, $\mathcal{S}$-FSS vs $\mathcal{SW}$-FSS

Test	Slave	# Iterations (10^6)	Comp. time (sec)	# Iterations (10^6)	Comp. time (sec)
		$\mathcal{S}$-CSS	$\mathcal{S}$-CSS	$\mathcal{SW}$-CSS	$\mathcal{SW}$-CSS
Floyd-Steinberg	twin2	59.93	19.25	89.90	28.88
	kid1	59.93	62.22	29.92	30.86
	twin3	59.93	19.24	74.92	24.06
	kid2	44.95	46.30	29.92	29.08
		$\mathcal{S}$-GSS	$\mathcal{S}$-GSS	$\mathcal{SW}$-GSS	$\mathcal{SW}$-GSS
Hydro	twin2	84.50	15.32	117.94	21.39
	kid1	78.38	42.60	38.03	22.49
	twin3	62.69	17.44	106.48	20.75
	kid2	73.58	33.72	36.41	19.46

Conclusions

In this section we addressed the problem of load balancing and scheduling DOACROSS (or DOALL) loops on heterogeneous non-dedicated distributed systems. We studied existing self-scheduling schemes and proposed two mechanisms to improve their performance using an extended master-slave model. Firstly, we tested the synchronization mechanism, which enables the application of existing self-scheduling algorithms to loops with dependencies. Secondly, we tested the weighting mechanism, which improves the load balancing capability of these algorithms. We ran experiments from practical applications involving DOACROSS loops and also a test with a DOALL loop with uneven tasks. The results show that the synchronization mechanism enables the scheduling algorithms to obtain significant speedups for the dependence loops. Furthermore, the weighting mechanism makes the existing algorithms most suitable for heterogenous non-dedicated systems since significant gains were obtained over the algorithms without weighting.

Although in this article we study the problem of scheduling DOACROSS loops with uniform dependencies we expect that our results can be extended to apply to DOACROSS loops with non-uniform dependencies. In the future we plan to study the scheduling problem for non-uniform dependencies DOACROSS loops in distributed systems. We also plan to further investigate the synchronization mechanism in terms of the number of synchronization points required for achieving the best performance.

5.3.5 Verifying the Theoretical Model for Predicting the Optimal h for CSS and $\mathcal{W}$-CSS

In this section, we present the practical verification of the theoretical model we proposed, for predicting the optimal synchronization frequency for DOACROSS loops. This model holds for the simple version of the CSS self-scheduling algorithm, as well as for its weighted version $\mathcal{W}$-CSS.

I. Experimental environment

We implemented our model in C using the MPI message-passing interface. The experiments were performed on two homogeneous clusters and one heterogeneous cluster:

Table 5.11: Estimated computation and communication parameters

Exp.	c_d $[\mu s]$	c_c $[\mu s]$	c_p *kids* $[\mu s]$	c_p *twins* $[\mu s]$	c_{sched} $[\mu s]$	t_{m-s} $[\mu s]$
#1	99	0.69	0.526	-	500	1.253
#2	99	0.65	-	0.319	500	1.25
#3	99	0.69	0.526	0.319	650	1.25

#1: a homogeneous cluster of 11 Intel Pentiums III machines called *'kids'* with 500MHz, 512MB RAM, with virtual power $VP^{kid} = 0.63$.

#2: a homogeneous cluster of 7 Intel Pentiums III machines called *'twins'* with 800MHz, 256MB RAM, with virtual power $VP^{twin} = 1$.

#3: a heterogeneous cluster of 5 *'kids'* and 6 *'twins'* (one *'twin'* was used as master).

We measured the virtual computational power of each worker by running a test problem 10 times (which involved nested loops with floating point operations) serially on each computer and averaging the measured execution times. The machines are interconnected by a 100Mbits/sec Fast Ethernet network. The results given in the following subsections are the average of 10 runs for each experiment. We used as a test case the Floyd-Steinberg algorithm.

We compare our model for the case when $P = N$ with the model of Chen & Xue [CX99]. We chose to compare with their work because, to the best of our knowledge, they are the only ones to address this problem for heterogeneous systems. They only do block assignment of tiles, both on a homogeneous and a heterogeneous system, which corresponds only to our $P = N$ cases.

II. Estimation of Computation and Communication Parameters

To quantify the *communication* parameters we developed a benchmark program, that simulates a small scale master-worker model with inter-node communication. It performs send and receive calls between all pairs of workers for different sized messages. We chose to perform data exchanges between all pairs of workers in order to simulate levels of network contention similar to those of the actual application. We measured the average round-trip time for all data exchanges. We then halve this averaged time and assume that the send (t_s) and receive (t_r) times are equal. This is a simple but realistic assumption, since in most cases the send and receive operations are executed in pairs between communicating workers. This allows us to estimate the start-up time c_d and transmission cost per unit and type of data c_c to the values are given in Table 5.3.5.

The computation cost per iteration is application- and processor-dependent. In our model we assume that it is the same for every iteration of the same application. To accurately quantify the *computation per iteration* cost we ran a small subset of the Floyd-Steinberg application, on each processor type. We divided the total computation time with the total number of iterations to obtain the c_p for this application. The values for each machine type (*kids* and *twins*) are given in Table 5.3.5.

III. Experiments and Results

In the following series of experiments we find the actual optimal synchronization interval which gives the minimum actual parallel time, denoted $h_{m,a}$. We then compare $h_{m,a}$ with the theoretical optimal value, denoted $h_{m,t}$, which is obtained from formulas (4.6), (4.9), (4.11) and (4.14), depending on the case. The maximum possible value of h is actually the size of the synchronization dimension, i.e., U_s. If $h = U_s$ it means there is no synchronization between workers which leads to a serial execution. The difference between $h_{m,t}$ and $h_{m,a}$ is measured as a percentage of U_s, since this is the maximum value h could take.

Experiment #1. We used the homogeneous cluster of 10+1 *kids*. We ran the Floyd-Steinberg application for an image size of 10k × 20k pixels.

(a) Initially, we examined the case when $P = N$ and compared our results with the results of Chen & Xue [CX99]. To compute V_i we used the following formula:

$$V_i = \frac{U_c}{P} \tag{5.5}$$

Chen & Xue computed the vertical tile size using formula (4) in [CX99]: $n_1^{opt} = N_1/P$. This can be written in our terms as: $V_i^{kids} = U_c/P$. Similarly, they computed the horizontal tile size as $n_2^{opt} = \sqrt{\frac{P(\alpha_s+\alpha_r+\gamma(P-1))N_2}{(P-1)(N_1 tc+\beta_s+\beta_r)}}$, which translates in our terms as $h_P = \sqrt{\frac{P(c_d+c_d)U_s}{(P-1)(U_c c_p^{kids}+2c_c)}}$. For the Floyd-Steinberg application on $10k \times 20k$ pixels, their formulas gave a vertical tile size equal to $V_i^{kids} = 1000$, a horizontal tile size equal to $h_P = 29$ and a parallel time of $T_P = 10.8359s$. The values obtained with our model in this case are: chunk size equal to $V_i^{kid} = 1000$, synchronization interval equal to $h_P = 30$ and parallel time of $T_P = 10.8361s$. It is obvious that both Chen & Xue's and our model predict that the minimum execution time is obtained for approximately the same synchronization interval.

(b) We then studied the case $P < N$. Each worker was assigned k chunks, where k assumed the values of 4 and 8. In this case V_i was computed according to following formula:

$$V_i = \frac{U_c}{kP} \tag{5.6}$$

In Fig. 5.31 we plotted the theoretical vs. the actual parallel times, on the *kids* homogeneous cluster. The theoretical time was obtained using the constants from Table 1 (#1). The curve representing the theoretical parallel time has a global minimum (i.e., the parallel time is minimized for $h_{m,t}$) and for every value of h greater than $h_{m,t}$, the parallel time increases approximately linearly, with a (very) small slope. Respectively, for every value of h less than $h_{m,t}$ excessive synchronization occurs which results in performance degradation. As it can be seen in Fig. 5.31, the actual parallel time confirms the theoretical pattern. The actual parallel time is always greater than the corresponding theoretical time. The global minimum of the actual parallel time is to the right of the theoretical global minimum. In all cases the difference between these two values, $h_{m,a}$ and $h_{m,t}$, and the actual time for each of these intervals can be seen in Table 5.3.5.

The differences in Table 5.3.5 show that one can now run the parallel application using the theoretical optimal synchronization interval $h_{m,t}$, which in practice gives a parallel time very close to the actual minimum parallel time. By using the synchronization interval given by eq. (4.9), one can expect to deviate from the optimal actual parallel time less than +0.0069% as demonstrated in Table 5.3.5, column 7.

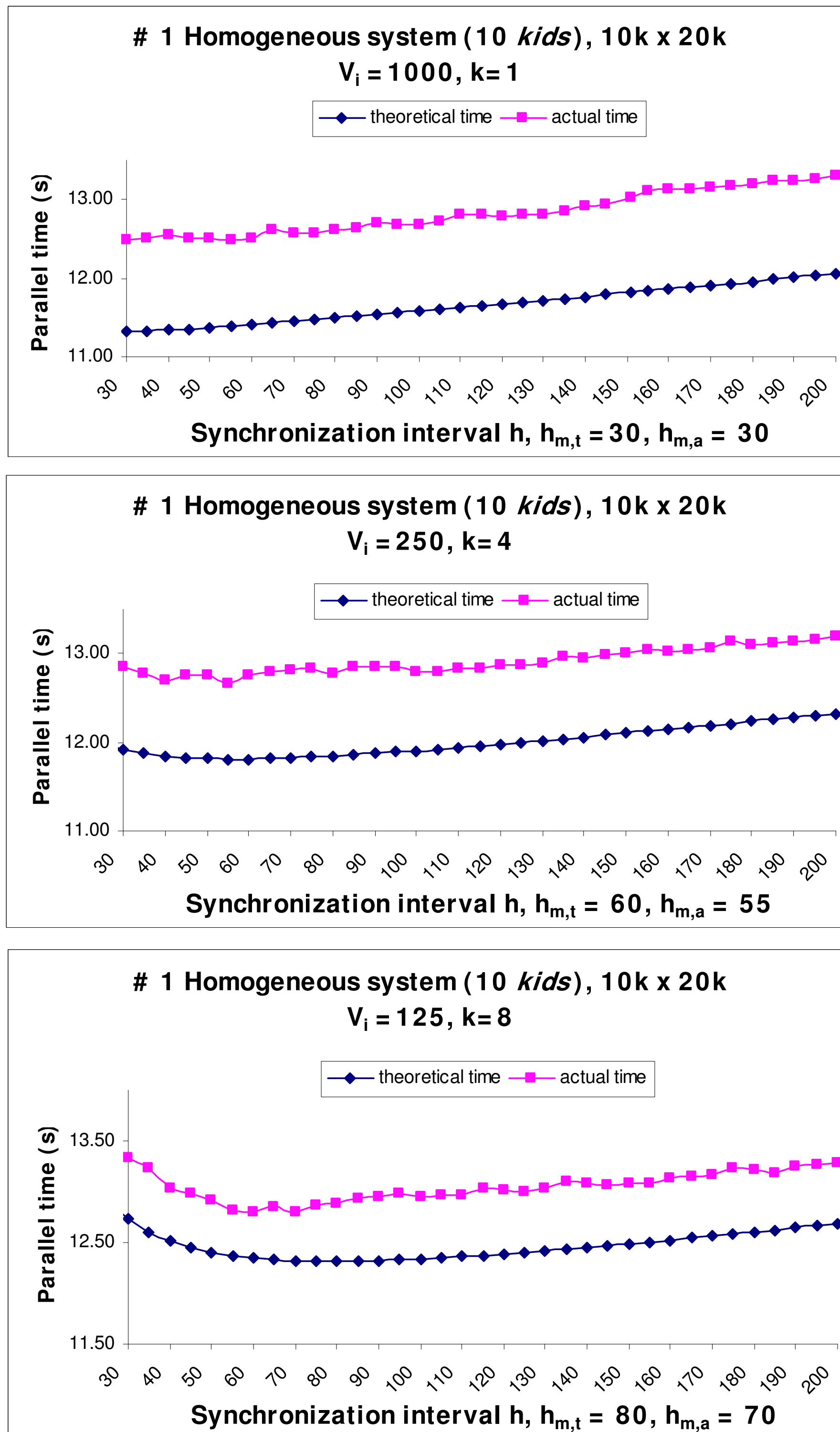


Figure 5.31: Theoretical vs. actual parallel times, homogeneous system, 10 workers, k=1, 4 and 8

Experiment #2. We wanted to test our model in a second homogeneous cluster, with different characteristics (i.e., processor speed, memory size), in order to confirm the validity

Table 5.12: Theoretical and actual optimal synchronization intervals, actual time for both these intervals, their difference, and deviation from the minimum actual parallel time

k	$h_{m,t}$	$h_{m,a}$	$T_a(h_{m,t})$ [s]	$T_a(h_{m,a})$ [s]	$\lvert T_a(h_{m,t}) - T_a(h_{m,a})\rvert$ [s]	$\frac{\lvert T_a(h_{m,t})-T_a(h_{m,a})\rvert}{T_a(h_{m,a})}$	$\lvert h_{m,t} - h_{m,a}\rvert$	$\frac{\lvert h_{m,t}-h_{m,a}\rvert}{U_s}100$
Homogeneous cluster with 10+1 *kids*								
1	30	30	12.4787	12.4787	0.00000	0.0000%	0	0.0000%
4	60	55	12.7500	12.6627	0.08730	0.0069%	5	0.0250%
8	80	70	12.8900	12.7982	0.09180	0.0072%	10	0.0500%
Homogeneous cluster with 6+1 *twins*								
1	40	55	7.04723	7.03397	0.01326	0.0019%	15	0.0750%
4	75	70	7.18130	7.15514	0.02616	0.0037%	5	0.0250%
8	110	100	7.29295	7.28574	0.00721	0.0010%	10	0.0500%
Heterogeneous cluster with 5 *kids* and 5+1 *twins*								
1	35	35	10.2356	10.2356	0.0000	0.0000%	0	0.0000%
4	65	100	10.6341	10.5301	0.1040	0.0099%	35	0.1750%
8	90	105	10.8348	10.7911	0.0437	0.0040%	15	0.0750%

of our theoretical model. Therefore, for this experiment, we repeated the measurements of the first experiment, but on the 6+1 *twins* homogeneous cluster, using the Floyd-Steinberg application for $8k \times 16k$ pixels.

(a) Again we began with the case $P = N$ and compared our results with the results of Chen & Xue [CX99]. They computed the vertical and horizontal tile size [CX99] using these formulas: $n_1^{opt} = N_1/P$ and $n_2^{opt} = \sqrt{\frac{P(\alpha_s+\alpha_r+\gamma(P-1))N_2}{(P-1)(N_1tc+\beta_s+\beta_r)}}$, respectively. Using our notation the above formulas become: $V_i^{twin} = U_c/P$ and $h_P = \sqrt{\frac{P(c_d+c_d)U_s}{(P-1)(U_c c_p^{twin}+2c_c)}}$, respectively. For the Floyd-Steinberg application on $8k \times 16k$ pixels, their formulas gave a vertical tile size equal to $V_i^{twins} = 1333$, a horizontal tile size equal to $h_P = 39$ and the parallel time of $T_P = 6.9987s$. Our values in this case were: chunk size $V_i^{twin} = 1334$, synchronization interval $h_P = 40$ and parallel time $T_P = 6.9988s$. Again, the results obtained by Chen & Xue are very similar to our results.

(b) For the case $P < N$ each worker was assigned k chunks, where k assumed again the values of 4 and 8, and V_i was computed according to formula (5.6). Fig. 5.32 shows the theoretical vs the actual parallel time. As with the previous case, the theoretical time was obtained using the constants from Table 5.3.5 (#2). Note that the curve of the theoretical parallel time follows the curve of the actual parallel time. The differences between $h_{m,t}$ and $h_{m,a}$ and the differences between their corresponding actual parallel times are given in Table 5.3.5. Again, using as synchronization interval the value given by eq. (4.9), one can expect to deviate from the minimum actual parallel time less than 0.0063% as demonstrated in Table 5.3.5, column 7.

Experiment #3. In this experiment, we tested our model in a heterogeneous cluster of 5 *kids* and 5+1 *twins*. We ran the Floyd-Steinberg application on an image size of $10k \times 20k$ pixels.

(a) For the heterogeneous case $P = N$ the value of V_i differed for each node type. Chunks were *weighted* according to the virtual power of a node and as a result *twins* workers got larger chunks than *kids* workers. The chunks were weighted as follows:

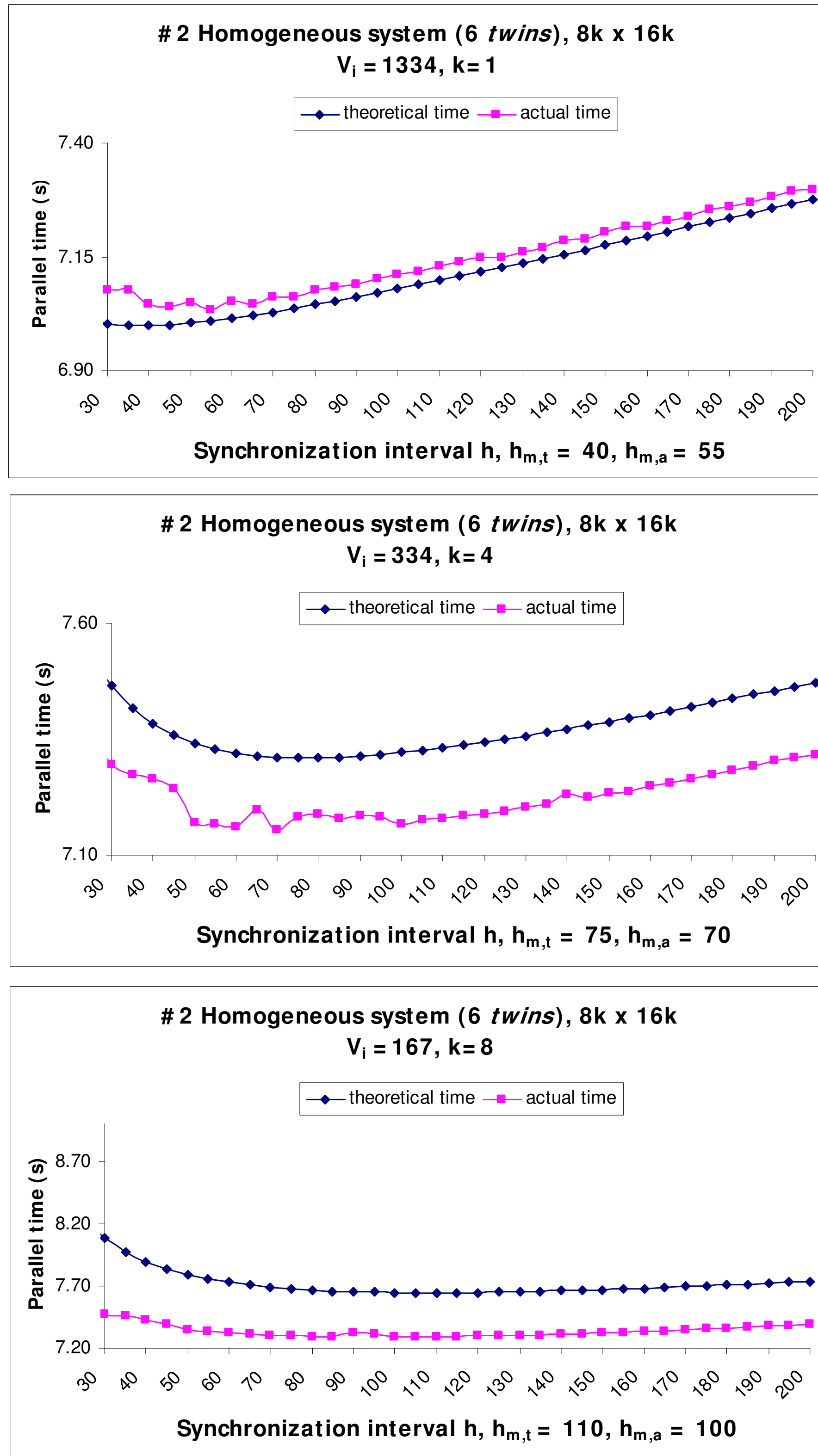


Figure 5.32: Theoretical vs. actual parallel times, homogeneous system, 6 workers, k=1, 4 and 8.

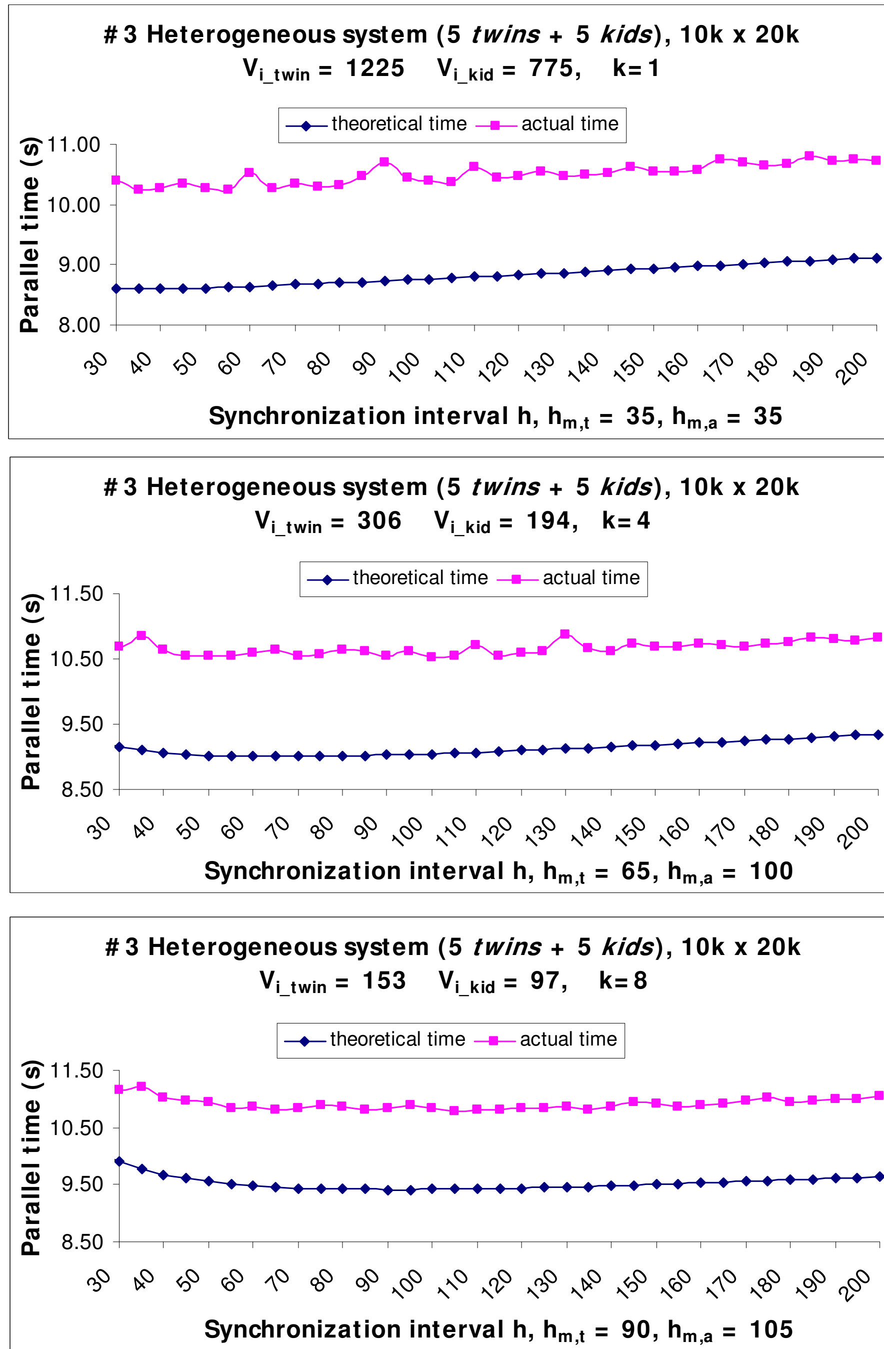


Figure 5.33: Theoretical vs. actual parallel times, heterogeneous system, 10 workers, k=1, 4 and 8

$$V_i^{kid} = \frac{U_c}{P} VP^{kid} \tag{5.7}$$

$$V_i^{twin} = \frac{U_c}{P} VP^{twin} \tag{5.8}$$

In this case Chen & Xue compute the vertical tile size using formula (8) from [CX99]: $n_1^i =$

$N_1 \frac{C_i}{\sum_P^{i=1} C_i}$. This formula, when written into our notation, becomes: $V_i^{\alpha_j} = U_c \frac{V_i P^{\alpha_j}}{\sum_\xi^{j=1}(VP^{\alpha_j} P^{\alpha_j})}$. The horizontal tile size is given by the formula $n_2^i = n_2^{opt} = \sqrt{\frac{P(\overline{\alpha_s}+\overline{\alpha_r}+\overline{\gamma}(P-1))N_2}{(P-1)(N_1\overline{tc}+\overline{\beta_s}+\overline{\beta_r})}}$, which in our notation becomes $h_P = \sqrt{\frac{\sum_\xi^{j=1} P^{\alpha_j}(c_d+c_d)U_s}{(\sum_\xi^{j=1} P^{\alpha_j}-1)(U_c\overline{c_p^{\alpha_j}}+2c_c)}}$. For the Floyd-Steinberg application and an image size of $10k \times 20k$, the vertical tile size for a *twin* worker is $V_i^{twin} = 1223$, and for a *kid* worker $V_i^{kid} = 776$; the horizontal tile size is $h_{P=N} = 162$ and the parallel time $T_P^{ht} = 8.8273s$. For the V_i^{twin}, V_i^{kid} and h values obtained with Chen & Xue's model, our model gives a theoretical parallel time of $T_P = 8.8316s$. However, the optimal theoretical parallel time predicted by our model is *smaller* than the above parallel time. Particularly, it is $T_P = 8.4501s$ and is obtained for a synchronization interval of $h_P = 35$. As can be seen in Table 2, our model gives a better estimate of $h_{m,t}$ since the value obtained with our model coincides with the experimentally computed value of $h_{m,a} = 35$.

(b) For the case $P < N$ each worker was assigned k chunks, where k assumed the values of 4 and 8. The value of V_i differed for each node type because chunks were *weighted* according to the virtual power of their node. The chunks were computed as follows:

$$V_i^{kid} = \frac{U_c}{kP} VP^{kid} \tag{5.9}$$

$$V_i^{twin} = \frac{U_c}{kP} VP^{twin} \tag{5.10}$$

The theoretical and actual parallel times for this experiment are plotted in Fig. 5.33. The theoretical time was obtained using the constants from Table 5.3.5 (#3). As it can be seen, the curve of the theoretical parallel time follows the curve of the actual parallel time, and the differences between $h_{m,t}$ and $h_{m,a}$, and between their corresponding actual parallel times are given in Table 5.3.5. The values in Table 5.3.5, column 7, show that using the synchronization interval given by eq. (4.14), one can expect to deviate from the optimal parallel time less than 0.0062%.

IV. Conclusions

In this section we presented the practical verification of the proposed theoretical model for determining the optimal synchronization frequency for DOACROSS loops on homogeneous and heterogeneous clusters. The accuracy of the proposed model is confirmed in all cases by experimental results. The main contribution of this work is the fact that the formulas (4.6), (4.9), (4.11) and (4.14) provide the means for calculating the optimal synchronization interval. In every experiment, the theoretical optimal h is very close to the actual optimal h. The performance loss due to the difference between the actual and the theoretical optimal h is very small, in the range 5-10% for all our experiments. Moreover, the cost of finding the optimal synchronization interval through extensive testing is clearly prohibitive and a poor choice of the synchronization interval leads to increased performance penalty. Our model improves on a previously existing model [CX99] on heterogeneous systems, whereas it gives similar results on homogeneous systems.

CHAPTER 6

Conclusions

This thesis presented several innovative static and dynamic scheduling algorithms for DOACROSS nested loops with uniform data dependencies. All these algorithms were designed to parallelize loops on high performance computing systems.

We show that computations can be efficiently assigned onto the minimum number of processing elements, guaranteeing the optimal makespan [AKC+03a] (see §3.2). Then, we show how to determine the minimum number of processors required for an optimal static schedule [AKC+03b] (see §3.3). Subsequent efforts to minimize the communication cost, with the aim of reducing the makespan, yield two static methods: an adaptive cyclic scheduling method (ACS) [CAP05], that exploits the geometric properties of the iteration space in order to arrive at an efficient geometric decomposition of the iteration space (see §3.4.1); and a chain pattern scheduling (CPS) [CAD+05], similar to ACS, but performs a different assignment to processors of the partitioned index space. Both methods aim at reducing the communication cost by assigning chains of computations (created by the a certain dependence vector) to the same processor, and mapping dependent chains on the same processor as much as possible.

We show how to automatically produce the equivalent parallel code from a given sequential program with nested DOACROSS loops, for shared and distributed memory architectures [CAK+03] [ACT+04] [ACT+08] (see §4.2.1 and §4.2.2). The algorithms used for scheduling are dynamic, based on the hyperplane method and on computational geometry methods. We then show how a new dynamic multi-phase scheduling scheme [CAR+06] [PRA+06] is employed as a first effort towards scheduling DOACROSS nested loops with a coarse grained approach, on heterogeneous dedicated & non-dedicated clusters (see §4.3). Next, we propose two mechanisms (synchronization and weighting) to improve the performance of self-scheduling algorithms for nested loop, on heterogeneous systems [CRA+08] (see §4.4). Finally, we show how to predict the optimal synchronization frequency for parallelizing DOACROSS loops using self-scheduling algorithms, using our proposed parallel estimation model [CRA+07a] [CRA+07b] (see §4.5).

6.1 Future Research Directions

1. Scheduling and Load Balancing for Grid Computing Environments

The main assumption in previous work on scheduling and load balancing schemes is that the systems are clusters of heterogeneous workstations. However, grid computing environments can be used for high performance computations. Grid technologies enable sharing of

computing resources, which may be heterogeneous computers on different sites. Scheduling and load balancing of non-independent tasks in such systems is a serious challenge. Recently, some scheduling and load balancing methods for independent tasks have been implemented on grid environments. The exploration of new and general methods that efficiently harness the power of computational grids is an open and active area of research, especially for applications with *non-independent tasks*. In the future, I would like to modify the existing schemes and devise new scheduling and load balancing methods for non-independent tasks with irregular workloads to run efficiently in grid environments. An immediate modification would be developing hierarchical versions of the dynamic schemes presented in Chapter 4, such that each grid site has one master, and inter-site communication is done through the masters. Another aspect that must be considered in grid computing is heterogeneity of communication links. Hence, another immediate modification to the dynamic schemes would be to assign highly communicating tasks to nodes with fast(est) links and ideally, also with fast processors. Depending on the nature and structure of the application's tasks, task replication could also be performed to avoid idle time and reduce (or save) the communication costs.

2. Resource Allocation and Scheduling for Large Scale Computations on Computational Grids

The need for large-scale computations, whether they originate as scientific or engineering applications, or for supporting large data-intensive calculations, leads to utilizing multiple (clusters of) computers at different sites distributed across the Internet. Computational grids are collections of distributed, possibly heterogeneous, resources which can be used as ensembles to execute large-scale applications. Such a large scale computational system requires two types of schedulers: global and local. Global schedulers distribute the tasks among the different sites, whereas local schedulers distribute the tasks among the computational nodes of a particular site. Local schedulers employ scheduling and load balancing algorithms usually devised for clusters. Global schedulers must be scalable and therefore should employ decentralized scheduling and load balancing algorithms that take into account the differences in network and processing speeds of the different sites, the number of sites involved in the computation, and the dynamic characteristics of the resources (availability, capability). In the future, I intend to modify existing algorithms devised for cluster environments to be used as decentralized global scheduling algorithms both for independent and non-independent tasks.

3. Fault Tolerant Scheduling in Distributed Systems

Distributed systems are mainly characterized by different processing powers. Another feature of interest for scheduling and load balancing is reliability, that is support for failure detection and management. Failures include node faults (nodes go down, reboot, etc) and network faults (links go down). A solution to the application-level fault tolerance problem must meet the following requirements: (i) efficiency, without compromising performance; (ii) the reliability level must be user controlled – greater reliability incurs a higher cost; and (iii) minimal code changes in the application. Scheduling algorithms that detect failures and are able to manage them are called fault-tolerant. The most common strategy is task replication. In the future, I intend to investigate the existing fault tolerance strategies for scheduling independent-tasks and apply them or devise new strategies for fault tolerant scheduling of non-independent tasks in distributed systems.

4. Scheduling and Load Balancing Divisible Loads

Divisible load theory is a methodology involving the linear and continuous modeling of partitionable computation and communication loads for parallel processing. It adequately represents an important class of problems with applications in parallel and distributed system

scheduling, various types of data processing, scientific and engineering computation, and sensor networks. The loads that must be processed (processing load) can be indivisible or divisible. The *divisible loads* can be split and assigned to many processors. But, the manner in which this partitioning (or load division) can be done depends on its *divisibility property*, that is, the property which determines whether a load can be decomposed into a set of smaller loads or not. If loads are *indivisible*, they are independent, of different sizes, and cannot be further subdivided. Thus, they have to be processed in their entirety in a single processor. These loads do not have any precedence relations and, in the context of static/deterministic scheduling, they give rise to bin-packing problems[1] that are known to be NP-complete and hence amenable only to heuristic algorithms that yield sub-optimal solutions. In the context of dynamic/stochastic scheduling, these loads arrive at random time instants and have to be assigned to processing nodes based on the state of the system.

Alternatively, a load may be *modularly divisible* in which case it is a priori subdivided into smaller modules based on some characteristics of the load or the system. The processing of a load is complete when all its modules are processed. Further, the processing of these modules may be subject to *precedence relations*. Usually such loads are represented as task interaction graphs whose vertices correspond to the modules, and whose edges represent interaction between these modules and perhaps also the precedence relationships. Finally, a load may be *arbitrarily divisible* which has the property that all elements in the load demand an identical type of processing. These loads have the characteristic that they can be arbitrarily partitioned into any number of load fractions. These load fractions may or may not have precedence relations.

In the future, I intend to apply existing scheduling algorithms or devise new ones to schedule and load balance applications with divisible loads (modularly divisible and/or arbitrarily divisible), which constitute tasks with dependencies.

[1]The bin-packing problem is stated as follows: objects of different volumes must be packed into a finite number of bins of capacity V in a way that minimizes the number of bins used.

APPENDIX A

Solving the general equation for 2D hyperplanes, computing minimum and successor points

A.1 Solving the general equation $a_1x_1 + a_2x_2 = k$

We show here how we can find the solutions of equation $a_1x_1+a_2x_2 = k$, assuming they exist. This can be done easily and efficiently using the tabular algorithm presented in [NZH91], which is a modification of the Gaussian elimination method. We explain this technique using the following example.

Example A.1.1 *Suppose that we want to find the solutions of the equation $2x_1 + x_2 = 9$ (recall Fig. 4.3(a)). For this we write:*

$$\begin{array}{ccc} 2 & 1 & 9 \\ 1 & 0 & \\ 0 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 1 & 1 & 9 \\ 1 & 0 & \\ -1 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 1 & 0 & 9 \\ 1 & -1 & \\ -1 & 2 & \end{array}$$

If the variables that are implicit in the array are u and v, where $u, v \in \mathbb{Z}$, we get that $u = 9$. The general solutions are given by $x_1 = u - v = 9 - v$ and $x_2 = -u + 2v = 2v - 9$. Similarly, we can find the solutions of the equation $2x_1 + 5x_2 = 21$ (recall Fig. 4.3(b)).

$$\begin{array}{ccc} 2 & 5 & 21 \\ 1 & 0 & \\ 0 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 2 & 3 & 21 \\ 1 & -1 & \\ 0 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 2 & 1 & 21 \\ 1 & -2 & \\ 0 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 1 & 1 & 21 \\ 3 & -2 & \\ -1 & 1 & \end{array} \Rightarrow \begin{array}{ccc} 1 & 0 & 21 \\ 3 & -5 & \\ -1 & 2 & \end{array}$$

Assuming that the variables that are implicit in the last array are $u, v \in \mathbb{Z}$, we see that $u = 21$. Consequently, the general solutions are $x_1 = 3u - 5v = 3 \times 21 - 3v = 63 - 5v$ and $x_2 = -u + 2v = 2v - 21$.

In our case, we are only interested in 2D points that lie in the first quadrant, i.e., that have both of their coordinates nonnegative. In view of this fact, the general solution $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$ gives that $j_1 + \frac{la_2}{g} \geq 0$ and $j_2 - \frac{la_1}{g} \geq 0$, which in turn implies that $la_2 + gj_1 \geq 0$ and $-la_1 + gj_2 \geq 0$ and finally that

$$-\frac{gj_1}{a_2} \leq l \leq \frac{gj_2}{a_1} \tag{A.1}$$

The above equation is very useful because the number of integers that lie in this interval gives the cardinality of the hyperplane $\Pi_k(a_1, a_2)$. Furthermore, the minimum index point of $\Pi_k(a_1, a_2)$ is computed by substituting the minimum value of integer l that satisfies inequality (A.1). Symmetrically, the maximum index point of $\Pi_k(a_1, a_2)$ is computed by substituting the maximum value of integer l that satisfies inequality (A.1).

Example A.1.2 *Continuing Example A.1.1, we find the number of nonnegative solutions of the equation* $2x_1 + x_2 = 9$*, which amounts to finding the number of index points of the hyperplane* $\Pi_9(2, 1)$ *(Fig. 4.3(a)). From equation (A.1) we get* $-\frac{j_1}{1} \leq l \leq \frac{j_2}{2}$*. Recall that* (j_1, j_2) *is a solution of* $2x_1 + x_2 = 9$*. From the previous example we know that* $x_1 = 9 - v$ *and* $x_2 = 2v - 9$*, where* $v \in \mathbb{Z}$*. We can use the value* $v = 1$*, thus, getting* $j_1 = 8$ *and* $j_2 = -7$*. Substituting these specific values for* j_1 *and* j_2 *we derive that* $-8 \leq l \leq \frac{-7}{2} \Rightarrow -8 \leq l \leq -3.5 \Rightarrow l = -8$ *or* $l = -7$ *or* $l = -6$ *or* $l = -5$ *or* $l = -4$*. This means that there are 5 index points that lie on the hyperplane* $2x_1 + 5x_2 = 21$*. From the general solution* $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$ *we find that the minimum point, corresponding to the minimum value of* $l = -8$*, is* $(0, 9)$ *and the the maximum point, corresponding to the maximum value of* $l = -4$*, is* $(4, 1)$*.*

Finally, we find the number of nonnegative index points of the line $2x_1 + 5x_2 = 21$ *(Fig. 4.3(b)). From equation (A.1) we get* $-\frac{j_1}{5} \leq l \leq \frac{j_2}{2}$*. Recall that* (j_1, j_2) *is a solution of* $2x_1 + 5x_2 = 21$*. From the previous example we know that* $x_1 = 63 - 5v$ *and* $x_2 = 2v - 21$*, where* $v \in \mathbb{Z}$*. We can use the value* $v = 1$*, thus, getting* $j_1 = 58$ *and* $j_2 = -19$*. Substituting these specific values for* j_1 *and* j_2 *we derive that* $-\frac{58}{5} \leq l \leq \frac{-19}{2} \Rightarrow -11.6 \leq l \leq -9.5 \Rightarrow l = -11$ *or* $l = -10$*. This means that there are two nonnegative index points that lie on* $2x_1 + 5x_2 = 21$*. From the general solution* $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$ *we find that the minimum point, corresponding to the minimum value of* $l = -11$*, is* $(3, 3)$ *and the maximum point, corresponding to the maximum value of* $l = -10$*, is* $(8, 1)$*.*

This formulation also facilitates the easy calculation of the successor in 2D index spaces. In the above example, we have found that hyperplane $\Pi_9(2, 1)$ contains 5 index points that lie on the hyperplane $2x_1 + 5x_2 = 21$. These points can be computed from the general solution $(j_1 + \frac{la_2}{g}, j_2 - \frac{la_1}{g})$ by substituting $j_1 = 8$ and $j_2 = -7$ and letting l take the values $-8, -7, -6, -5, -4$. Index point $(2, 5)$ corresponds to $l = -6$. Therefore, the successor of $(2, 5)$ is the index point corresponding to $l = -5$. Substituting $l = -5$ in the general solution gives the index point $(3, 3)$, which is indeed the successor of $(2, 5)$.

A.2 Computing minimum points in n-dimensional hyperplanes

Finding the minimum point of a given hyperplane is an optimization problem. Moreover, it exhibits the *optimal substructure* property, i.e., the optimal solution to the n-dimensional case contains within it the optimal solution to the $(n-1)$-dimensional case. Consequently, the minimum point can be computed by a simple dynamic programming algorithm. The pseudocode for this algorithm is given in Fig. A.1. Vector $\mathbf{A}$ contains the coefficients of the family of hyperplanes and vector $\mathbf{U}$ contains the coordinates of the terminal point. For instance, referring to Fig. 4.3(b) of Example A.2.1, $\mathbf{A} = \{2, 5\}$ and $\mathbf{U} = \{105, 90\}$. Array **isMin** is a 2-dimensional boolean array such that $\mathbf{isMin}[k][n]$ is `true` if the n-dimensional hyperplane k has a minimum point. Array **minPoint** is a 2-dimensional array that stores the minimum points, i.e., $\mathbf{minPoint}[k][n]$ contains the minimum point of the n-dimensional hyperplane k. The concatenation of vectors is denoted by $\circ$, e.g., $(3) \circ (3)$ gives $(3, 3)$. In a symmetrical way, one can compute the maximum points of n-dimensional hyperplanes. In

```
for (hPlane = 0; hPlane ≤ maxHPlane; hPlane++) {
     posValue = min (hPlane / A[n], U[n]);
     isMin[hPlane][1] = (A[n] * posValue == hPlane);
     if (isMin[hPlane][1])
          minPoint[hPlane][1] = (posValue);
}

for (dim = 2; dim ≤ n; dim++)
     for (hPlane = 0; hPlane ≤ maxHPlane; hPlane++)
          for (value = hPlane; value ≥ 0; value--)
               if (isMin[value][dim-1]) {
                    remPoints = hPlane - value;
                    posValue = min (remPoints / A[n-dim], U[n-dim]);
                    if (A[n-dim]*posValue == remPoints) {
                         isMin[hPlane][dim] = true;
                         minPoint[hPlane][dim] = (posValue) ∘ minPoint[value][dim-1];
                         break;
                    }
               }
```

Figure A.1: Pseudocode for the dynamic programming algorithm that computes the minimum point

typical dynamic programming fashion, the algorithm in Fig. A.1 works bottom-up. This means that given an n-dimensional index space, the algorithm begins by first computing the minimum points for 1D hyperplanes moving up one dimension at a time until it computes the minimum points for the n-dimensional hyperplanes of the given index space. The details are explained in the following example.

Example A.2.1 *Revisiting Example A.1.2, we show in Fig. 4.3 the minimum and the maximum points of the hyperplanes* $\Pi_9(2,1) : 2x_1 + x_2 = 9$ *and* $\Pi_{21}(2,5) : 2x_1 + 5x_2 = 21$. *The former hyperplane (Fig. 4.3(a)) contains the following five index points (lexicographically ordered):* $(0,9), (1,7) < (2,5) < (3,3) < (4,1)$*; the minimum is* $(0,9)$ *and maximum is* $(4,1)$. *The latter hyperplane (Fig. 4.3(b)) contains two index points:* $(3,3)$*, which is the minimum, and* $(8,1)$*, which is the maximum.*

We show now how the Algorithm of Fig. A.1 finds the minimum points. In order to find the minimum points for the hyperplanes $2x_1 + 5x_2 = k,\ 0 \leq k \leq 21$*, we begin by considering the family of 1D hyperplanes* $\Pi^1_k(5)$*, which is defined by the equation* $5x_1 = k,\ 0 \leq k \leq 21$. *A 1D hyperplane has a minimum point if and only if it contains an index point. So, in this case finding the minimum point of a hyperplane (or establishing that the hyperplane contains no index points) is trivial. For instance, the minimum point of* $\Pi^1_0(5)$ *is* (0)*, but* $\Pi^1_1(5)$ *has no minimum point because there is no integer* l *that satisfies the equation* $5 \cdot l = 1$. *In this way, we can easily find the minimum points (if they exist) for the hyperplanes* $\Pi^1_k(5),\ 0 \leq k \leq 21$.

Suppose that we want to compute the minimum point $\mathbf{j}_m = (j_1, j_2)$ *of hyperplane* $\Pi_{21}(2,5)$. *The fact that* $\mathbf{j}_m$ *belongs to* $\Pi_{21}(2,5)$ *means that* $2j_1 + 5j_2 = 21$. *The lexicographic ordering implies that* (j_2) *is the minimum point of* $\Pi^1_r(5)$*, for some* $r,\ 0 \leq r \leq 21$. *The algorithm utilizes the knowledge of the previously computed minimum points for the 1D hyperplanes* $\Pi^1_k(5)$ *in order to determine the value of* r. *This is done by starting from* 21 *and decrementing by one until we find a candidate value for* r. *The first such possible value is* 20*, corresponding to the 1D hyperplane* $\Pi^1_{20}(5)$ *with minimum point* (4). *This value is dismissed because there is no integer* l *that satisfies the equation* $2 \cdot l = 21 - 20 = 1$. *So, we proceed to find the next candidate value, which is* 15*, corresponding to to the 1D hyperplane* $\Pi^1_{15}(5)$ *with minimum point* (3). *This value is acceptable because there exists an integer* l *that satisfies the equation*

```
for (dim = (n-1); dim ≥ 1; dim--)
     if (pointJ[ dim ] < U[ dim ]) {
          for (i = 1; i < dim; i++)
               successorJ [ i ] = pointJ [ i ];
          for (value = pointJ[ dim ] + 1; value ≤ U[ dim ]; value++) {
               successorJ[ dim ] = value;
               usedPoints = 0;
               for (i = 1; i ≤ dim; i++)
                    usedPoints += A[i] * successorJ[i];
               remPoints = hPlane - usedPoints;
               if ( remPoints ≥ 0 && isMin [ remPoints ] [ n-dim ] ) {
                    for (i = dim + 1; i ≤ n; i++)
                         successorJ [ i ] = minPoint[ remPoints ] [ n-dim ];
                    break;
               }
          }
     }
```

Figure A.2: Pseudocode for finding the successor of index point `pointJ` that belongs to hyperplane `hPlane` and stores it in `successorJ`

$2 \cdot l = 21 - 15 = 6$. *Hence, the value of the first coordinate is* 3 *and the resulting minimum point is the concatenation of* (3) *with the minimum point of* $\Pi^1_{15}(5)$, *that is* $(3, 3)$.

A.3 Computing successor points in n-dimensional hyperplanes

Finding the successor of an index point is straightforward once the minimum points are known. The pseudocode of the algorithm for computing the successor in the general n-dimensional case is given in Fig. A.2. It finds the successor of index point `pointJ` that belongs to hyperplane `hPlane` and stores it in `successorJ`. Vectors **A** and **U** contain the coefficients of the family of hyperplanes and the coordinates of the terminal point, respectively. Array **isMin** is a 2-dimensional boolean array such that **isMin**$[k][n]$ is `true` if the n-dimensional hyperplane k has a minimum point. Array **minPoint** is a 2-dimensional array that stores the minimum points, i.e., **minPoint**$[k][n]$ contains the minimum point of the n-dimensional hyperplane k. Let $\mathbf{i} = (i_1, \ldots, i_n)$ be an index point of hyperplane $\Pi_k(a_1, \ldots, a_n)$, other than the maximum, and let $\mathbf{j} = (j_1, \ldots, j_n)$ be its successor. The fact that **i** and **j** belong to $\Pi_k(a_1, \ldots, a_n)$ means that $a_1 i_1 + \ldots + a_n i_n = a_1 j_1 + \ldots + a_n j_n = k$. Moreover, $\mathbf{i} < \mathbf{j}$, meaning that there exists an integer r, $1 \le r \le n$ such that $i_1 = j_1, \ldots, i_{r-1} = j_{r-1}$ and $i_r < j_r$. So, in order to construct **j** we begin by finding those coordinates of **i** that can be increased, leading to points greater than **i**, but still in hyperplane k. This is trivial because coordinate r of **i** can be increased if $i_r < u_r$, where u_r is the r-th coordinate of the terminal point $\mathbf{U} = (u_1, \ldots, u_n)$. Notice that it is pointless to increase the n-th coordinate (the last one) and we use this observation in the code. From all the possible coordinates we can increase, we use the maximum because any other candidate coordinate less than the maximum would result in an index point greater than the successor of **i**. That is the reason we begin our search from right to left, that is from $n - 1$ to 1. After we pick the candidate coordinate, call it r, we try to increase it by 1[1], until we find the successor. The remaining $k - (a_1 j_1 + \ldots + a_r j_r)$ points must be distributed in the last $(n - r)$ dimensions in a valid manner. The crucial

[1]Any increase greater than 1 may result in a point greater than the successor.

observation here is that in order to construct the successor we must use the minimum point of the $(n-r)$-dimensional hyperplane $k-(a_1j_1+\ldots+a_rj_r)$. Any other point will result in $\mathbf{j}$ being greater that the successor.

Example A.3.1 *Continuing the previous example, we explain how to find the successor of point* $(3,3)$ *of hyperplane* $\Pi_{21}(2,5)$. *Using the Algorithm of Fig. A.2 we try to increment the first coordinate of* $(3,3)$. *We begin the search for the proper value from* $4=3+1$. *This leaves* $21-2\cdot 4=13$ *points for the second coordinate. Unfortunately, hyperplane* $\Pi^1_{13}(5)$ *has no minimum point, so we must reject candidate value* 4. *Similarly, candidate values* $5,6$ *and* 7 *are rejected. Let us examine now how the algorithm works when we try value* 8. *In this case, we check whether* $\Pi^1_5(5)$ *(*$5=21-2\cdot 8$*) has a minimum point. The answer is positive and the minimum point is* (1). *Hence, the successor of* $(3,3)$ *is* $(8,1)$.

Bibliography

[ACT+04] T. Andronikos, F. M. Ciorba, P. Theodoropoulos, Kamenopoulos D., and G. Papakonstantinou. Code Generation For General Loops Using Methods From Computational Geometry. In *IASTED Parallel and Distributed Computing and Systems Conference (PDCS'04)*, pages 348–353, 2004.

[ACT+08] T. Andronikos, F.M. Ciorba, P. Theodoropoulos, D. Kamenopoulos, and G. Papakonstantinou. Cronus: A platform for parallel code generation based on computational geometry methods. *J. of Systems and Software*, available online 2008.

[AG89] G. S. Almasi and A. Gottlieb. *Highly Parallel Computing.* Benjamin-Cummings publishers, Redwood city, CA, 1989.

[AG91] I. Ahmad and A. Ghafoor. Semi-distributed load balancing for massively parallel multicomputer systems. *IEEE Transactions on Software Engineering*, 17(10):987–1004, 1991.

[AKC+03a] T. Andronikos, M. Kalathas, F. M. Ciorba, P. Theodoropoulos, and G. Papakonstantinou. An efficient scheduling of uniform dependence loops. In *6th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'03)*, September 2003.

[AKC+03b] T. Andronikos, M. Kalathas, F.M. Ciorba, P. Theodoropoulos, and G. Papakonstantinou. Scheduling nested loops with the least number of processors. In *Proceedings of the 21st IASTED Int'l Conference APPLIED INFORMATICS*, 2003.

[APT96] T. Andronikos, G. Papakonstantinou, and P. Tsanakas. Optimal scheduling of uniform dependence loops in polynomial time. In *2nd ECPD International Conference on Advanced Robotics, Intelligent Automation and Active Systems*, pages 605–610, Vienna, Austria, 1996.

[AR97] R. Andonov and S. Rajopadhye. Optimal orthogonal tiling of 2-d iterations. *J. Parallel Distrib. Comput.*, 45(2):159–165, 1997.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[BDH96] C. Bradford Barber, David P. Dobkin, and Hannu Huhdanpaa. The Quickhull Algorithm for Convex Hulls. *ACM Transactions on Mathematical Software*, 22(4):469–483, 1996.

[BDRR94] P. Boulet, A. Darte, T. Risset, and Y. Robert. (pen)-ultimate tiling. *Integration, the VLSI Journal*, 17:33–51, 1994.

[BENP93] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua. Automatic program parallelization. *IEEE*, 81(2):211–243, 1993.

[BL00] I. Banicescu and Z. Liu. Adaptive factoring: A dynamic scheduling method tuned to the rate of weight changes. In *Proc. of the High Performance Computing Symposium 2000*, pages 122–129, Washington, USA, 2000.

[Bon] Dan Bonachea. Gasnet communication system. http://gasnet.cs.berkeley.edu/.

[CABG01] A. T. Chronopoulos, R. Andonie, M. Benche, and D. Grosu. A class of distributed self-scheduling schemes for heterogeneous clusters. In *roc. of the 3rd IEEE International Conference on Cluster Computing (CLUSTER 2001)*, Newport Beach, CA USA, 2001.

[CAD+05] F. M. Ciorba, T. Andronikos, I. Drositis, G. Papakonstantinou, and P. Tsanakas. Reducing the communication cost via chain pattern scheduling. In *4th IEEE Conference on Network Computing and Applications (NCA'05)*, July 2005.

[CAK+03] F. M. Ciorba, T. Andronikos, D. Kamenopoulos, P. Theodoropoulos, and G. Papakonstantinou. Simple code generation for special UDLs. In *1st Balkan Conference in Informatics (BCI'03)*, pages 466–475, 2003.

[CAP05] F. M. Ciorba, T. Andronikos, and G. Papakonstantinou. Adaptive cyclic scheduling of nested loops. In *7th Hellenic European Research on Computer Mathematics and its Applications (HERCMA'05)*, September 2005.

[CAR+06] F. M. Ciorba, T. Andronikos, I. Riakiotakis, A. T. Chronopoulos, and G. Papakonstantinou. Dynamic multi phase scheduling for heterogeneous clusters. In *20th International Parallel and Distributed Processing Symposium (IPDPS'06)*, April 2006.

[CDRV98] P. Y. Calland, A. Darte, Y. Robert, and F. Vivien. On the removal of anti and output dependencies. *Int. Journal of Parallel Programming*, 26(2):285–312, 1998.

[CRA+07a] F. M. Ciorba, I. Riakiotakis, T. Andronikos, A. T. Chronopoulos, and G. Papakonstantinou. Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous sytems. In *16th International Conference on Parallel Architectures and Compilations Techniques (PACT '07)*, pages 403–403, September 2007.

[CRA+07b] F.M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos. Studying the impact of synchronization frequency on scheduling tasks with dependencies in heterogeneous systems. *J. of Performance Evaluation*, Under review 2007.

[CRA+08] F. M. Ciorba, I. Riakiotakis, T. Andronikos, G. Papakonstantinou, and A. T. Chronopoulos. Enhancing self-scheduling algorithms via synchronization and weighting. *J. Parallel Distrib. Comput.*, 68(2):246–264, 2008.

[CX99] S. Chen and J. Xue. Partitioning and scheduling loops on nows. *Comp. Comm. J.*, 22:1017–1033, 1999.

[DAK+] I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric pattern prediction and scheduling of uniform dependence loops. In *5th Hellenic Eur. Conf. on Comp. Math. and its Appl. - HERCMA 2001.*

[DAK+01] I. Drositis, T. Andronikos, A. Kokorogiannis, G. Papakonstantinou, and N. Koziris. Geometric scheduling of 2-D uniform dependence loops. In *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS 2001*, pages 259–264, Korea, 2001.

[DAK+02] I. Drositis, T. Andronikos, M. Kalathas, G. Papakonstantinou, and N. Koziris. Optimal loop parallelization in n-dimensional index spaces. In *Proc. of the 2002 Int'l Conf. on Par. and Dist. Proc. Techn. and Appl. (PDPTA'02)*, 2002.

[DGK+00] I. Drositis, G. Goumas, N. Koziris, P. Tsanakas, and Papakonstantinou. Evaluation of loop grouping methods based on orthogonal projection spaces. In *Proceedings of the 2000 International Conference on Parallel Processing ICPP-2000*, pages 469–476, Toronto, Canada, 2000. IEEE Press.

[DKR91] A. Darte, L. Khachiyan, and Y. Robert. Linear scheduling is nearly optimal. *Par. Proc. Letters*, 1.2:73–81, 1991.

[DRR96] M. Dion, T. Risset, and Y. Robert. Resource-constrained scheduling of partitioned algorithms on processor arrays. *Integration, the VLSI Journal*, 20(2):139–159, 1996.

[EFKR01] D.W. Engels, J. Feldman, D.R. Karger, and M. Ruhl. Parallel processor scheduling with delay constraints. In *12th Annual Symp. on Discrete algorithms*, pages 577–585, NY, USA, 2001. ACM Press.

[ELJ86] D. L. Eager, E. D. Lazowska, and Zahorjan J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Engineering*, SE-12(5):622–675, 1986.

[ERAL95] H. El-Rewini, H. H. Ali, and T. Lewis. Task scheduling in multiprocessing systems. *IEEE Computer*, 28(12):27–37, 1995.

[FDR96] P. Ramet F. Desprez and J. Roman. Optimal grain size computation for pipelined algorithms. In *Euro-Par*, pages 165–172, 1996.

[For02] MPI Forum. Message Passing Interface Standard. http://www-unix.mcs.anl.gov/mpi/indexold.html, 2002.

[FS76] R.W. Floyd and L. Steinberg. An adaptive algorithm for spatial grey scale. In *Proc. Soc. Inf. Display*, volume 17, pages 75–77, 1976.

[GJ79] M.R. Garey and D.S. Johnson. *Comp. and Intractability, a Guide to the Theory of NP-completeness*. W.H. Freeman & Co., New York, 1979.

[HBD] M. Harchol-Balter and A.B. Downey.

[HBFF00] D. J. Hancock, J. M. Bull, R. W. Ford, and T. L. Freeman. An investigation of feedback guided dynamic scheduling of nested loops. In *Proc. of the IEEE International Workshops on Parallel Processing*, pages 315–321, 2000.

[HSF92] S. F. Hummel, E. Schonberg, and L.E. Flynn. Factoring: A method for scheduling parallel loops. *Communications of the ACM*, 35(8):90–101, 1992.

[HSUW96] S.F. Hummel, J. Schmidt, R.N. Uma, and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *Proc. of 8th Annual Symp. on Parallel Algorithms and Architectures*, pages 318–328, Padua, Italy, 1996.

[IBV00] S.Ghafoor I. Banicescu, P. Soni and V. Velusamy. Effectiveness of adaptive weighted factoring in computational field simulation on unstructured grids. In *Proc. of the High Performance Computing Symposium 2000*, pages 168–177, Washington, USA, 2000.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Proceedings of the 15th Annual ACM SIG ACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 319–329, January 1988.

[Jam99] Heath A. James. *Scheduling in Metacomputing Systems*. PhD thesis, University of Adelaide, Australia, 1999.

[KN01] Chou W.-H. King, C.-T. and L. Ni. Pipelined Data-Parallel Algorithms: Part II Design. *IEEE Trans. on Parallel and Distributed Systems*, 2(4):430–439, 2001.

[KPT96] N. Koziris, G. Papakonstantinou, and P. Tsanakas. Optimal time and efficient space free scheduling for nested loops. *The Computer Journal*, 39(5):439–448, 1996.

[Kun91] T. Kunz. The influence of different workload descriptions on a heuristic load balancing scheme. *IEEE Trans. on Soft. Eng.*, pages 725–730, 1991.

[KW85] C. P. Kruskal and A. Weiss. Allocating independent subtasks on parallel processors. *IEEE Transactions on Software Engineering*, 11(10):1001–1016, January 1985.

[Lam74] L. Lamport. The Parallel Execution of DO Loops. *Comm. of the ACM*, 37(2):83–93, Feb. 1974.

[LB] LBNL and UC Berkeley. *Berkeley Unified Parallel C*.

[Low00] D. K. Lowenthal. Accurately Selecting Block Size at Run Time in Pipelined Parallel Programs. *Int. J. of Par. Progr.*, 28(3):245–274, 2000.

[MA01] N. Manjikian and T.S. Abdelrahman. Exploiting Wavefront Parallelism on Large-Scale Shared-Memory Multiprocessors. *IEEE Trans. on Par. and Dist. Systems*, 12(3):259–271, 2001.

[Man88] B.B. Mandelbrot. *Fractal Geometry of Nature*. W. H. Freeman & Co, 1988.

[McM86] F.H. McMahon. The livermore fortran kernels: A computer test of the numerical performance range. *Lawrence Livermore National Laboratory*, UCRL-53745, 1986.

[MF86] D. I. Moldovan and J. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–11, 1986.

[MJ94] E. P. Markatos and LeBlanc T. J. Using processor affinity in loop scheduling on shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):379–400, 1994.

[MMS99] B. L. Massingill, T. G. Mattson, and B. A. Sanders. Patterns for parallel application programs. In *Proc. of the 6th Pattern Lang. of Prog. (PLoP '99)*, 1999.

[Mol93] D. I. Moldovan. *Parallel Processing: From Applications to Systems*. M. Kaufmann, California, 1993.

[MSK04] J. Ferrante M.M. Strout, L. Carter and B. Kreaseck. Sparse tiling for stationary iterative methods. *Int'l J. of High Perf. Computing Applications*, 18(1):95–113, 2004.

[MSM04] T. G. Mattson, B. A. Sanders, and B. L. Massingill. *Patterns for Parallel Programming*. Addison-Wesley, 1st edition, 2004.

[NZH91] I. Niven, H. Zuckerman, and Montgomery H. *An Introduction to the Theory of Numbers*. John Wiley & Sons, 5th edition, 1991.

[Pac97] Peter Pachecho. *Parallel Programming with MPI*. Morgan Kauffman, 1997.

[PAD01] G. Papakonstantinou, T. Andronikos, and I. Drositis. On the parallelization of UET/UET-UCT loops. *Neural, Parallel & Scientific Computations*, December 2001.

[Par99] Behrooz Parhami. *Introduction to Parallel Processing: Algorithms and Architectures*. Kluwer Academic Publishers, Norwell, MA, USA, 1999.

[PK87] C. D. Polychronopoulos and D. J. Kuck. Guided self-scheduling: A practical self-scheduling scheme for parallel supercomputers. *IEEE Transactions on Computer*, C-36(12):1425–1439, 1987.

[PKA+97] A. Pappas, N. Koziris, T. Andronikos, G. Papakonstantinou, and P. Tsanakas. Nested loop parallelization over a distributed memory environment. In *6th Hellenic Conference on Informatics*, Athens, 1997.

[PLR+95] M. A. Palis, J.-C. Liou, S. Rajasekaran, S. Shende, and D. S. L. Wei. Online scheduling of dynamic trees. *Parallel Processing Letters*, 5(4):635–646, 1995.

[PRA+06] G. Papakonstantinou, I. Riakiotakis, T. Andronikos, F. M. Ciorba, and A. T. Chronopoulos. Dynamic scheduling for dependence loops on heterogeneous clusters. *Neural, Parallel & Scientific Computations*, 14(4):359–384, 2006.

[Pug92] W. Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, 1992.

[PY88] C. Papadimitriou and M. Yannakakis. Toward an architecture-independent analysis of parallel algorithms. *SIAM J. of Comp., Ext. Abstract in Proc. STOC 1988*, 19:322–328, 1988.

[RRP03] Fabrice Rastello, Amit Rao, and Santosh Pande. Optimal task scheduling at run time to exploit intra-tile parallelism. *Parallel Comput.*, 29(2):209–239, 2003.

[RS92] J. Ramanujam and P. Sadayappan. Tiling multidimensional iteration spaces for multicomputers. *Journal of Parallel and Distributed Computing*, 16(2):108–230, 1992.

[SC95] J.-P. Sheu and T.-S. Chen. Partitioning and Mapping Nested Loops for Linear Array Multicomputers. *Journal of Supercomputing*, 9:183–202, 1995.

[SF91] W. Shang and J.A.B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6):723–742, 1991.

[SHK95] B. A. Shirazi, A. R. Husson, and K. M. Kavi. Scheduling and load balancing in parallel and distributed systems, chapter introduction to scheduling and load balancing. *IEEE Computer Society Press*, 1995.

[SHW96] R.N. Uma S.F. Hummel, J. Schmidt and J. Wein. Load-sharing in heterogeneous systems via weighted factoring. In *8th Annual ACM Symp. on Parallel Algorithms and Architectures*, 1996.

[Sil01] J. Silverman. *A Friendly Introduction to Number Theory*. Prentice Hall, 2nd edition, 2001.

[Sin07] Oliver Sinnen. *Task Scheduling for Parallel Systems*. Wiley, 2007.

[SSDNB95] J. A. Stankovic, M. Spuri, M. Di Natale, and G. C. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25, 1995.

[ST91] J. Sheu and T. Tai. Partitioning and mapping nested loops on multiprocessors systems. *IEEE Transactions on Parallel and Distributed Systems*, 24:430–439, 1991.

[TKP00] P. Tsanakas, N. Koziris, and G. Papakonstantinou. Chain Grouping: A method for partitioning loops onto mesh-connected processor arrays. *IEEE Transactions on Parallel and Distributed Systems*, 11(9):941–955, September 2000.

[TN93] T. H. Tzen and L. M. Ni. Trapezoid self-scheduling: A practical scheduling scheme for parallel compilers. *IEEE Transactions on Parallel and Distributed Systems*, 4(1):87–98, 1993.

[Ull75] J. Ullman. NP-complete scheduling problems. *J. of Comp. and System Sciences*, 10:384–393, 1975.

[vECGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: A mechanism for integrated communication and computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[VRKL96] T. Varvarigou, V. P. Roychowdhury, T. Kailath, and E. Lawler. Scheduling in and out forrests in the presence of communication delays. *IEEE Trans. of Par. and Dist. Comp.*, 7(10):1065–1074, 1996.

[WL91] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 30–44, New York, NY, USA, 1991. ACM Press.

[XC02] J. Xue and W. Cai. Time-minimal tiling when rise is larger than zero. *Parallel Computing*, 28(6):915–939, 2002.

[Xue97] Jingling Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

[Xue00] J. Xue. *Loop Tiling for Parallelism.* Kluwer Academic Publishers, 2000.

[YH95] H. Yee and Yu Hen Hu. A novel modular systolic array architecture for full-search block matching motion estimation. *IEEE Transactions on Circuits and Systems for Video Technology*, 5(5):407–416, October 1995.

Index

*S*GRIDs, 47
CRONUS, 51

algorithm
  BSA, 24
  BSDA, 22
  DA, 24, 25

benchmarks
  Artificial, 89
  Floyd-Steinberg, 88
  FSBM ME, 86
  Heat Conduction, 87
  Hydro, 87
  Mandelbrot Set, 85
  Transitive Closure, 88
binary search, 24

Cartesian
  coordinates, 16
  representation, 16
cluster
  dedicated, 104
  non-dedicated, 104
code generation, 50
  automatic, 49
communication cost, 45, 78
computation cost, 46, 78
computing power
  available, 45
  virtual, 45
Convex Hull, 52

data locality, 13, 19, 38
dependence
  vector, 17
    uniform, 17
dependence vectors, 17

ECT or earliest computation time, 20

General loops, 44, 51
GRIDs, 20

heterogeneity
  network, 14
  processor, 14, 57
high performance computing, 1
hyperplane, 21, 29

index space, 15, 32
  auxiliary, 48
iterations space, 16, 32, 37

LCT or latest computation time, 20
Lexicographic ordering, 53
load balancing
  dynamic, 43
    centralized, 43
    distributed, 44
loop scheduling, 3

makespan, 22
master-slave model, 57
maximum point, 53, 54
mechanism
  synchronization, 45, 66
  synchronization and weighting, 75
  weighting, 45, 72
minimum point, 53, 55

nested loops, 1
  perfectly, 15

parallel processing, 1
parallelism
  coarse grain, 3
  fine grain, 3

precedence constraints, 16
predecessor, 54
program
  parallel, 46
  sequential, 46

Scaled GRIDs, 47
scheduling
  dynamic, 12, 43, 48
    $DMPS(\mathcal{A})$, 61
    centralized, 13
    de-centralized (distributed), 13
    SDS, 55
    self-scheduling, 58
    self-scheduling schemes, 57
  global, 3
  local, 3
  self-scheduling schemes, 66
    CSS, 57, 59
    DTSS, 57, 59
    FSS, 66
    PSS, 59
    TSS, 57, 59
  self-scheduling-schemes
    GSS, 66
  static, 12, 19
    ACS, 32
    CPS, 37
SGRIDs, 91
successor, 54
synchronization
  frequency, 45, 78, 79
  interval, 45, 62, 79
  points, 45, 57, 69
systems
  dedicated, 14
  heterogeneous, 14
  homogeneous, 14
  non-dedicated, 14

task scheduling, 3, 11, 13
tasks
  dependent, 3, 12
  independent, 12

UCT or unit communication time, 22
UDL or uniform dependence loop, 19
UET or unit execution time, 22

wavefront, 21