

PARALLELISM REVELATION MODEL

— APPLICATION TO PARALLEL PROGRAMMING —

I hereby state that this thesis contains only my own original work
except where explicit reference has been made to the work of others,
and has not been submitted for any other degree.

Pin Chen


Pin Chen

14th Feb 1996

Date

A thesis submitted for the degree of
Doctor of Philosophy
of The Australian National University


©Pin Chen
February 1996

Dedicated

Statement

to my parents, Gengren Chen and Ling Xie

I hereby state that this thesis contains only my own original work except where explicit reference has been made to the work of others, and has not been submitted for any other degree.



Pin Chen

14th Feb. 1996

Date

Dedicated

to my parents, *Gengren Chen and Ling Xie,*

and

to my wife, *Jian Yang*

This version of the thesis has been revised carefully and reprinted after taking into account the criticisms and suggestions made by the panel of examiners.

The author expresses his gratitude to all the examiners for their valuable time and suggestions provided.

Date of original version: 1994 Feb. 1994

Date of revised version: 1996 July 1996



Dedicated

to my parents, Gertrude Chen and Tang Xie

and

to my wife, Jean Yang



Acknowledgments

Particular thanks are due to my supervisor, Prof. E. V. Krishnamurtay, for his continued support over the previous three years keeping me on track and providing an additional source of ideas and suggestions have been instrumental in the elaboration of my ideas. I am especially grateful for his most helpful comments on my written work. Also, to the rest of my thesis committee, Prof. Richard Brent and Dr.

Revised Version

This version of the thesis has been revised carefully and reprinted after taking into account the criticisms and suggestions made by the panel of examiners.

The author expresses his gratitude to all the examiners for their valuable time and suggestions provided.

Date of original version: 14th Feb. 1996

Date of revised version: 18th July 1996

Revised Version

This version of the thesis has been revised carefully and reprinted after taking into account the criticisms and suggestions made by the panel of examiners.

The author expresses his gratitude to all the examiners for their valuable time and suggestions provided.

Date of original version: 10th Feb 1995

Date of revised version: 18th July 1996



Acknowledgments

Particular thanks are due to my supervisor, Prof. E. V. Krishnamurthy, for his continued support over the previous three years keeping me on track and providing an additional source of ideas and feedback. His critical questions have been instrumental in the elaboration of my ideas. I am especially grateful for his most helpful comments on my written work. Also, to the rest of my thesis committee, Prof. Richard Brent and Dr. Xin Yao, for their input towards the progress and direction of my research work.

I am also indebted to Prof. Terry Bossomaier and Drs. Brendan McKay, Iain Macleod, and Bing Bing Zhou for their continuous and friendly support and encouragement which were indispensable to the development of this research.

I would like to thank again Prof. Richard Brent for fighting on my behalf, with bureaucracies of various sorts; also for supporting my attendance at a number of international conferences. So allowing me to encounter other researcher in the area of parallel computing and exchange ideas. I also acknowledge the help received from Michelle Moravec during the write-up of this thesis.

I am grateful to the fellow researchers I have been in contact with in Australia and abroad, in the exchange of ideas with whom has greatly contributed to my own understanding of many problems.

Finally, I would like to thank my family and friends, whose confidence in me always seemed greater than my own; my special thank goes to my wife Jian Yang, for her perpetual love, constant support and encouragement and being there with me.

This research has been funded by the PhD Scholarship from The Australian National University. I gratefully acknowledge the financial support received both from the University and from the Computer Sciences Laboratory.

Acknowledgments

Particular thanks are due to my supervisor, Prof. E. V. Krishnamurthy, for his continued support over the previous three years, helping me on track and providing an additional source of ideas and feedback. His critical questions have been instrumental in the elaboration of my ideas. I am especially grateful for his most helpful comments on my written work. Also, to the rest of my thesis committee, Prof. Richard Brent and Dr. Xin Yang, for their input towards the progress and direction of my research work.

I am also indebted to Prof. Jerry Rosenblum and Dr. Braden Krag for their comments, and Bing Hong Zhou for their continuous and friendly support and encouragement, which were indispensable to the development of this research.

I would like to thank again Prof. Richard Brent for fighting on my behalf, with persistence of various sorts, also for supporting my attendance at a number of international conferences. So allowing me to encounter other researchers in the area of parallel computing and exchange ideas. I also acknowledge the help received from Michelle Motzoc during the write-up of this thesis.

I am grateful to the fellow researchers I have been in contact with in Australia and abroad, in the exchange of ideas with whom has greatly contributed to my own understanding of many problems.

Finally, I would like to thank my family and friends, whose confidence in me always seemed greater than my own; my special thanks goes to my wife, Jan Yang, for her perpetual love, constant support and encouragement and being there with me.

This research has been funded by the PhD Scholarship from The Australian National University. I gratefully acknowledge the financial support received both from the University and from the Computer Science Laboratory.

Abstract

The massive computational power provided by parallel computers can be used with great benefit to many application areas; however, the lack of adequate programming tools for the development of parallel software prevents the efficient utilisation of parallel computers. Also, the performance failures of parallel implementation arise due to great freedom available in exploiting parallelism and due to the subjective view of the programmer.

This thesis proposes a new model of parallel computation, ABCOM (ABSTRACT Computational tuple space Model). Unlike most existing models or programming languages which support expressing parallelism in a program in terms of the subjective knowledge of a programmer, ABCOM is developed with a primary goal of revealing parallelism of a given problem in a programmer-view independent manner.

We introduce our model by describing its notation and properties, and comparing it with other practical or theoretical models. The characteristic features of ABCOM are demonstrated through applications to parallelism inference, optimisation, abstraction, profiling, speculation and scalability analysis. Based on ABCOM, the spatial structure and temporal logic of solving a problem can be fully exhibited in an abstract computational space; an initially expressed solution of a problem can be optimised until all computations involved are exploited in a dataflow computation fashion. The motivation of our research is to improve parallel programming methodologies by providing a new model that enhances the existing techniques and tools. An important aspect of this research is to separate the parallelism investigation task as a relatively independent one from that of mapping of the problem into a particular physical architecture. This investigation is carried out to establish a general knowledge of parallel properties of a real world problem. Such a knowledge serves as a common basis for various tasks involved in parallel programming.

The main contributions of this thesis are: i) introduction of new concepts for parallel computing – such as: *subjective parallelism*, *objective parallelism* and *scalability of application domain parallelism*; ii) development of ABCOM as a parallelism revelation model; iii) new approaches to detecting exact data dependence and parallelising program solutions; and iv) construction of a foundation for an integrated parallel programming platform.

Also this research throws new light on the current state of art in parallel computing and enables one to reevaluate our current views on parallel computing — in particular, some fundamental issues in programming philosophy and methodologies.

Abstract

The massive computational power provided by parallel computers can be used with great benefit to many application areas; however, the lack of adequate programming tools for the development of parallel software prevents the efficient utilization of parallel computers. Also, the performance failure of parallel implementation arises due to the great freedom available in exploiting parallelism and due to the subjective view of the programmer.

This thesis proposes a new model of parallel computation, ABDOM (Abstract Computational Tuple Space Model). Unlike most existing models or programming languages which support expressing parallelism in a program in terms of the subjective knowledge of a programmer, ABDOM is developed with a primary goal of revealing parallelism of a given problem in a programmer-view independent manner.

We introduce our model by describing its notation and properties, and comparing it with other practical or theoretical models. The characteristic features of ABDOM are demonstrated through applications to parallelism inference, optimization, abstraction, profiling, specification and scalability analysis. Based on ABDOM, the parallel structure and temporal logic of solving a problem can be fully exhibited in an abstract computational space; an initially expressed solution of a problem can be optimized until all computations involved are explained in a database computation fashion. The motivation of our research is to improve parallel programming methodology by providing a new model that enhances the existing techniques and tools. An important aspect of this research is to separate the parallelism investigation task as a relatively independent one from that of mapping of the problem into a particular physical architecture. This investigation is carried out to establish a general knowledge of parallel properties of a real world problem. Such a knowledge serves as a common basis for various tasks involved in parallel programming.

The main contributions of this thesis are: i) introduction of new concepts for parallel computing - such as: subjective parallelism, objective parallelism and scalability; ii) development of ABDOM as a parallelism level-of-application domain paradigm; iii) new approaches to detecting exact data dependence and parallelism; and iv) construction of a foundation for an integrated parallel programming platform.

Also this research throws new light on the current state of art in parallel computing and enables one to reevaluate our current view on parallel computing - in particular some fundamental issues in programming philosophy and methodology.

Contents

Glossary

ABCOM	Abstract COmputational tuple space Model	4
ADP	Application Domain Parallelism	140
BMF	Bird-Meertens Formalism	121
CDOAD	Completely specified DOAG	36
DAG	Directed Acyclic Graph	45
DDG	Data Dependence Graph	25
DOAG	Data-Operation-Associated Graph	35
EDOAG	Elementary Data-Operation-Associated Graph	35
EP	Execution Pointer	41
GAMMA	Γ programming model	23
HPF	High Performance FORTRAN	23
PPM	Parallel Programming Methodologies	7
PRAM	Parallel Random Access Machine	10
SADP	Scalability of ADP	140
uDDG	<i>unconstrained</i> Dynamic Dependence Graph	25

Glossary

4	Abstract Computational Tuple space Model	ARCOM
140	Application Domain Parallelism	ADP
131	Bird-Mestens Formalism	BMF
38	Completely parallel DOAG	CDOAG
42	Directed Acyclic Graph	DAG
38	Data Dependence Graph	DDG
48	Data-Operation-Associated Graph	DOAG
42	Elementary Data-Operation-Associated Graph	EDOAG
44	Exit Point	EP
38	T programming model	GAMMA
33	High Performance FORTRAN	HPF
7	Parallel Programming Methodologies	PPM
10	Parallel Random Access Machine	PRAM
140	Stability of ADP	SADP
32	Unconstrained Dynamic Dependence Graph	UDDG

Contents

1 Introduction	1
1.1 Motivation	1
1.2 Why a Model for Parallelism Revelation	3
1.3 Structure of the thesis	5
2 Background, Motivation and Objectives	7
2.1 The Development of PPM	7
2.1.1 Parallel computation models	8
2.1.2 Parallelising compilers	12
2.1.3 Functional programming	13
2.1.4 Programming paradigms	15
2.2 Problems	17
2.3 Related research	21
2.4 Thesis Objectives	25
2.4.1 The Proposed improvement in PPM	25
2.4.2 Parallelism revelation models	26
2.5 Summary	27
3 ABCOM— A Parallelism Revelation Model	29
3.1 A Puzzle — Parallelism in a Problem	29
3.2 Definitions and Properties	33
3.3 Program Solutions in ABCOM	40
3.4 Features of ABCOM	43
3.4.1 The ABCOM (virtual) machine	43

3.4.2	ABCOM and other models — a comparison	44
3.5	Summary	48
4	Expressive Power and Transformation	49
4.1	Expressive Power	49
4.1.1	Syntax of ABCOM	50
4.1.2	Examples	53
4.2	Solution Transformation	54
4.2.1	Principles of translation	56
4.2.2	Trace-driven code generation	60
4.2.3	Branching statements transformation	61
4.2.4	Loop transformation	67
4.2.5	While--Do transformation	70
4.3	Summary	73
5	Parallel Computational Inference	75
5.1	Domain Representation Issues	75
5.1.1	Requirements for conducting parallelism analysis	76
5.1.2	ABCOM tuplebase	78
5.2	Relation-Based Computing Inference and Analysis	81
5.2.1	Time-based inference	83
5.2.2	Data-based inference	86
5.2.3	Operation-based inference	89
5.2.4	CDOAG relations inference	89
5.2.5	Nondeterministic computation analysis	90
5.2.6	Element-state based inference	95
5.3	Summary	97
6	Solution Parallelisation in ABCOM	99
6.1	Overview of Optimising Compilers	99
6.1.1	Existing studies	100
6.1.2	Problems	101

6.2	ABCOM-Based Data Dependence Tests	104
6.2.1	Dependence representation	104
6.2.2	Features of ABCOM-based detection	105
6.3	Parallelisation in ABCOM	107
6.3.1	CDOAG optimisation	108
6.3.2	Solution parallelisation	111
6.3.3	Observation of nondeterministic computation	118
6.4	Summary	119
7	Parallel Computing Platform	121
7.1	The Notation of Bird-Meertens Formalism	121
7.2	Parallelism Profiling	124
7.2.1	Data parallelism profiling	124
7.2.2	Control parallelism profiling	125
7.3	Computation Pattern Testing	126
7.3.1	Normalising <i>CDOAGs</i>	127
7.3.2	<i>CDOAG</i> structure optimisation	130
7.3.3	Patterns represented in a loop	131
7.4	Size-Based Parallelism Speculation	131
7.5	Scalable Performance Analysis	137
7.5.1	Scalable parallel computing	137
7.5.2	Scalability of application domain parallelism	140
7.6	Other Applications	146
7.6.1	Integrating with a cost system	146
7.6.2	Solution Derivation Support	148
7.6.3	ABCOM-based programming paradigm	150
7.7	Summary	152
8	Conclusions	153
8.1	Thesis Summary	153
8.2	Limitations	155
8.3	Future Work	157

A	161
A.1	161
A.2	164
Bibliography	167
1.1	167
1.2	167
1.3	167
1.4	167
1.5	167
1.6	167
1.7	167
1.8	167
1.9	167
1.10	167
1.11	167
1.12	167
1.13	167
1.14	167
1.15	167
1.16	167
1.17	167
1.18	167
1.19	167
1.20	167
1.21	167
1.22	167
1.23	167
1.24	167
1.25	167
1.26	167
1.27	167
1.28	167
1.29	167
1.30	167
1.31	167
1.32	167
1.33	167
1.34	167
1.35	167
1.36	167
1.37	167
1.38	167
1.39	167
1.40	167
1.41	167
1.42	167
1.43	167
1.44	167
1.45	167
1.46	167
1.47	167
1.48	167
1.49	167
1.50	167
1.51	167
1.52	167
1.53	167
1.54	167
1.55	167
1.56	167
1.57	167
1.58	167
1.59	167
1.60	167
1.61	167
1.62	167
1.63	167
1.64	167
1.65	167
1.66	167
1.67	167
1.68	167
1.69	167
1.70	167
1.71	167
1.72	167
1.73	167
1.74	167
1.75	167
1.76	167
1.77	167
1.78	167
1.79	167
1.80	167
1.81	167
1.82	167
1.83	167
1.84	167
1.85	167
1.86	167
1.87	167
1.88	167
1.89	167
1.90	167
1.91	167
1.92	167
1.93	167
1.94	167
1.95	167
1.96	167
1.97	167
1.98	167
1.99	167
2.00	167

List of Tables

3.1	Comparison of <i>CDOAG</i> with other graph representations used	47
4.1	Example of Quadruples	56
4.2	A merging-point processing table	65
7.1	Operator precedence for pattern normal forms	129
7.2	SADP analysis of Example 5, 6 and 7.	145

List of Figures

2.1	The paradigms of Parallel computing	9
2.2	Different programming solutions lead to different performance.	19
3.1	An parallel algorithm for matrix multiplication	30
3.2	Cannon's systolic algorithm for matrix multiplication	31
3.3	Computation graph for c_{ij}	31
3.4	<i>CDOAGs</i> of Example 1	36
3.5	Examples of relations between <i>CDOAGs</i>	38
3.6	ABCOM platform for parallel processing	46
4.1	The Syntax of ABCOM	50
4.2	A composed element	52
4.3	Overview of the ABCOM compiler	55
4.4	Syntax tree	56
4.5	The grammar for assignments	56
4.6	Syntax-directed semantic rules to produce ABCOM code for assignments	58
4.7	The grammar for conditional statements	59
4.8	The syntax-directed semantic rules to translate conditional statements	60
4.9	Algorithm for assignment transformation	62
4.10	The table of operation precedence relations	63
4.11	A general case of the branching and merging points	63
4.12	A program flow chart including conditional statements	63
4.13	A general example of the conditional statements	65
4.14	An ABCOM code of a conditional statement with two branch flows	66
4.15	Algorithm for Do--loop	68
4.16	The transformed code of Example 5	69

4.17	The transformed code of Example 6	69
4.18	A accuracy-control iteration of <code>While-----Do</code>	71
4.19	The transformed result of <code>While--do</code>	71
5.1	The conceptual schema in ABCOM tuplebase	80
5.2	Algorithm 3 for CDOAG generation	82
5.3	Algorithm 4 to obtain $t_{lower_{u_i}}$	84
5.4	Algorithm 5 and 6 for detecting data flow relations.	85
5.5	A data access pattern abstracted from Table 5.1	87
5.6	Algorithm 7 for testing a contained relation between <i>CDOAGs</i>	90
5.7	Algorithm 8 for testing $CDOAG_{u_i} \bowtie CDOAG_{u_j}$	91
5.8	Algorithm 9 for testing $CDOAG_{u_i} \parallel_c CDOAG_{u_j}$	92
5.9	Part of the ABCOM code of Example 8.	93
5.10	Uncertain relations among some elements of Example 8	94
5.11	The procedure of element migration	96
6.1	Two access patterns based on the same data object	108
6.2	Algorithm optimising a <i>CDOAG</i>	110
6.3	The <i>CDOAGs</i> of Example 6.	112
6.4	Two access patterns of data object <i>a</i>	113
6.5	The optimised solution of Example 5	116
6.6	The optimised solution of Example 6	117
7.1	Two different <i>CDOAGs</i> of the same set of elements.	128
7.2	The normal form of $CDOAG_{u_{11} 1}$ and $CDOAG_{u_{11} 2}$	130
7.3	The iteration increment in a nested loop	135
7.4	Scalable computing	138
7.5	The revised scalability metrics.	142
7.6	ABCOM-based parallel programming paradigm	151

Chapter 1

Introduction

.....for art and science are a single gift, called science inasmuch as art refashions the mind, and called art inasmuch as by science the world is refashioned.

Santayana, *Dialogues in Limbo*

1.1 Motivation

Massively parallel computers are attractive tools for solving many computationally intensive problems. While parallel programming practice is still considered as an art that demonstrates personal experience, skills and knowledge, the main challenge lies in writing programs that fully exploit the power of parallelism for a given problem and architecture. Current parallel programming methodologies are empirical or *ad hoc* since a number of different solutions to a given problem can be arrived at by different people even if they are allowed to use the same computer architecture. The performance of a solution can be quite different for each possible solution due to the differing experiences in programming and understanding of the problem acquired by each programmer. In most cases, the first workable program solution is likely to be adopted. If the chosen solution does not turn out to be efficient one, then it will be an expensive choice in the long run. As a consequence, a natural question asked by programmers is: why a developed solution is better than others that have not been tried yet and/or why a

particular architecture is more suitable for a given problem than other architectures. If this question is not addressed properly, parallel computing will not achieve its potential.

The diversity of research in this discipline which can be broadly termed 'parallel computing' is large and ever growing. This discipline ranges from the highly abstract and theoretical formalisms to very specific and practical implementations. However, sophisticated programming methodologies for parallel computing are yet to be developed [Pan91], [KN93]. It is also hard to find research and techniques that provide general and useful guidelines as to what extent the parallelism of a problem should be exploited, though there are many research papers outlining techniques that describe how parallelism can be achieved.

We should nevertheless realise that it is the domain knowledge of parallelism that is vital in answering the question such as '*to what extent*' parallelism is realisable. In order to reduce the risk of performance failures of parallel programming, therefore, the scope of research and development in parallel computing should be enlarged. Parallel programming methodologies should pay more attention to the foundations. One of the important issues to be considered is how a developer can be assisted in building up a sound domain knowledge of parallelism, or how users can share the same background of domain knowledge. This background should provide a general view on the problem, and should be relatively free of subjective factors.

Parallelism in solving a problem results from two different, yet interrelated aspects that are *objective* and *subjective* respectively. Objective aspects restrict certain computation tasks to execute sequentially (for instance, operations in a dataflow relationship), while some other tasks can be parallelised if certain execution conditions are met. The awareness of subjective aspects arises when a number of different program solutions with different parallel properties are developed. In such a case individual parallel properties may be or may not be selected to be used in a solution. The selection of parallel properties in a physical implementation results from a subjective decision of the programmer and is constrained by the tools and architectures used. Therefore, expressing parallelism in programming is based on subjectivity. The subjective aspects of parallelism play an important role in achieving good performance in parallel computing. The study and techniques to deal with objective aspects of parallelism have not yet

been paid much attention in the literature except the concept of dataflow computing, perhaps due to the fact that we have yet not realised their importance because of our cognitive limitations. The aim of this thesis is to examine the role of the objectivity in parallel computing.

1.2 Why a Model for Parallelism Revelation

A model called a *parallelism revelation model* is developed with the primary goal of revealing parallelism. Unlike the conventional computation models that are developed for designing architectures and languages, or as a tool to express parallel properties of solving a problem that are known to a developer, a parallelism revelation model plays quite a different role in parallel computing. From a methodology point of view, first, this model assists a programmer to build up a sound domain knowledge of parallel properties before a real implementation commences. Secondly, the method of revealing parallelism differs from that for expressing parallelism in the following respects:

1. The parallelism expressed in a language is somehow constrained by a number of subjective factors — such as subjective views of programmers and constraints of the language and the architecture, if architecture dependent. The expressed parallelism therefore contains the features of subjectivity. The parallelism revealed by a parallelism revelation model should be much less constrained so that the objectivity of parallelism can be studied.
2. The result of expressing parallelism leads directly to a specific implementation with certain properties of parallelism subjectively selected by the programmer. The parallelism revelation results in a knowledge about parallel properties of solving a problem. This knowledge can serve as a common basis for parallelism analysis, solution derivation and further mapping the problem onto a specific architecture.

Conventional programming practice starts with expressing a solution in a particular language in the light of a subjective understanding. In a parallelism revelation model, one starts with building up a background of parallel properties of solving a problem.

Consequently, a parallelism revelation model can complement many existing techniques and tools of parallel computing.

This thesis is devoted to the development of such a parallelism revelation model and a programming platform that relates this model to other research aspects and issues in parallel computing. The model developed is called an *abstract computational tuple space model* (ABCOM) [CK95c], [CK95a]. It is an intermediate representation into which a program solution expressed in a conventional programming language can be converted. The initial version of the solution converted from a source code preserves all execution features of computation designed in the code. As a parallelism revelation model, ABCOM supports parallelism analysis and inference by providing for a set of relation-based rules [CK95b]. Moreover, all the dependencies can be exactly detected [CKY95], and the memory-based dependencies are removable from the solution. Thus, an initial version of a solution can be optimised until we get a new solution having the maximum parallelism. The optimisation can then be carried out in a machine independent fashion.

In such an optimised solution that is executable in an ideal machine, all the dataflow computation features contained in solving the problem are exploited by removing all the constraints introduced in the source code. For any two solutions to the same problem, the difference in performance between their optimised solutions is equal to the difference between the two longest paths of dataflow computation in these two solutions respectively. It is also demonstrated that this optimisation which reveals parallelism is a relatively independent task from physically mapping a real world problem to a particular architecture. Based on the optimised solution, the objective features of parallel properties of the problem can be inferred, analysed, abstracted and profiled. ABCOM can serve as a platform to support parallelism speculation, scalable performance analysis, solution derivation and performance prediction when a target architecture is selected for implementation.

To study parallelism as a function of the size of the problem, we introduce the concept of scalability of application domain parallelism. This concept is important in programming as well as selecting a suitable architecture for a given problem. By examining the applications of ABCOM, we show that a parallelism revelation model

can be used in an integrated parallel programming environment in which a number of techniques and tools of parallel computing can be cooperatively applied and developed.

The main contributions of this work are: i) proposing and developing a parallelism revelation model that enhances the existing techniques and tools; ii) introducing the concepts of subjective parallelism and objective parallelism that can tell us as 'to what extent' parallelism can be exploited; iii) developing techniques to reach an optimised solution with objective parallelism for a given problem; iv) introducing the concept of scalability of application domain parallelism based on which scalable performance analysis can be carried out to support program design and architecture selection, in particular when the problem size is variable; and v) presenting a foundation of a parallel programming platform that can be used cooperatively by many techniques and tools used in parallel computing.

1.3 Structure of the thesis

The work reported in this thesis focuses on investigating the nature of parallelism inherent in a problem and developing techniques that support this study. Unlike other studies, our study of parallelism is considered as a relatively independent task from any physical realisation.

In the following chapter we provide a background of the state of art of parallel programming, identify problems that have not been addressed properly and discuss the relations between existing research (including techniques and tools) and issues concerned in this thesis. By examining the main problems, we conclude that some new techniques or tools should be developed to enhance the existing techniques. The features of and requirement for the new techniques are highlighted.

To address the issues raised, *an abstract computational tuple space model* (ABCOM) is introduced as a parallelism revelation model in Chapter 3. ABCOM's notation and properties are provided as a theoretical foundation of the model. A comparison between ABCOM and conventional languages and computation models is presented to characterise the potential application of ABCOM.

Chapter 4 examines the expressive power of ABCOM and the main transformation

techniques to generate ABCOM code from a FORTRAN-like sequential source code. Since ABCOM is an intermediate representation, it is not suitable to directly express a solution in it by hand in the sense of programming.

The features of ABCOM are further explored in Chapter 5 to show the advantages of such a model in supporting computation and parallelism inference required by performing more complex tasks in parallel computing. A relation-based programming database is suggested for practical implementation of the inference techniques.

In Chapter 6 an approach to ABCOM based solution parallelisation is presented. Using this approach a given solution that usually is a **control flow program** is optimised into **data flow computation** where the resultant parallelism is objective.

To pursue our goals of developing such a model, in Chapter 7, we discuss ABCOM as a parallel computing platform to support parallelism profiling, speculation and scalable performance analysis. The connection between ABCOM and other techniques of parallel computing are also examined.

Finally, Chapter 8 contains a summary of the contributions and limitations of the present thesis. Also we draw some conclusions, and point out avenues for future research.

Chapter 2

Background, Motivation and Objectives

In the first section of this chapter we briefly examine the state of the art of parallel programming methodologies (PPM); in particular, the main concepts and techniques used for exploiting parallelism. These techniques are not necessarily the most popular or practically successful ones, but are relevant to this dissertation due to the significance of the issues they address. The major problem considered in this thesis is presented in Section 2.2 with a review of current related research to the problem in Section 2.3. The thesis objectives are stated in Section 2.4.

2.1 The Development of PPM

The state of the art of parallel programming methodologies is influenced by three main factors:

(i) Sequential programming concepts: The main framework of PPM is based on conventional sequential programming, which includes a variety of aspects such as languages, computation models, compilers, debugging and portability of programs. Therefore, parallel programming inherits most of the problems encountered in sequential programming.

(ii) Different architectures: Different architectures require PPM to deal with different models of computation and communication for achieving high performance. Due to the development of diverse parallel system architectures, a number of special areas of research have emerged and became part of the PPM. Examples of these are message-passing, synchronising, load-balancing and performance prediction. Without providing

adequate system support for these aspects, a new architecture cannot be successful commercially. In order to utilise this system support, there is an increasing demand for programmers with adequate skills and knowledge. Also the expertise in programming a particular architecture becomes necessary for successful parallel implementation.

(iii) Programming paradigms: Finally, to cope with the difficulty and complexity of parallel programming, certain programming paradigms have been developed — such as coordination languages and skeleton programming. These paradigms strongly influence parallel programming styles.

Parallel computing can be achieved by using an *explicit* or an *implicit* approach, as shown in Fig. 2.1.

In the explicit approach a mapping procedure from a real world problem to a target parallel computer system is carried out. The parallelism properties for solving a problem are recognised, exploited and represented in a programming language by programmers; then a compiler transforms the given program with associated parallelism into an executable code on a specific architecture. The present generation of languages requires programmers to be aware of, and explicitly handle either the parallelism, communication, or both. The awareness of parallelism requires not only the characteristics of the architecture, but also the features of problem domain.

In the implicit approach a parallelising compiler first detects parallelism in a sequential or functional program, and then converts the program into executable code for a specific architecture.

The key issue in both approaches is how to fully exploit parallelism for a given problem based on a specific architecture. To achieve this goal, a number of concepts and techniques have been developed. There are a number of important research papers on exploiting parallelism, bringing significant progress for PPM. We briefly review the relevant research aspects in the following subsections.

2.1.1 Parallel computation models

The von Neumann model is universal and serves as a bridge between programs and machines for sequential computation. In parallel computation, however, the existing parallel programming languages are tied to some particular parallel computation model,

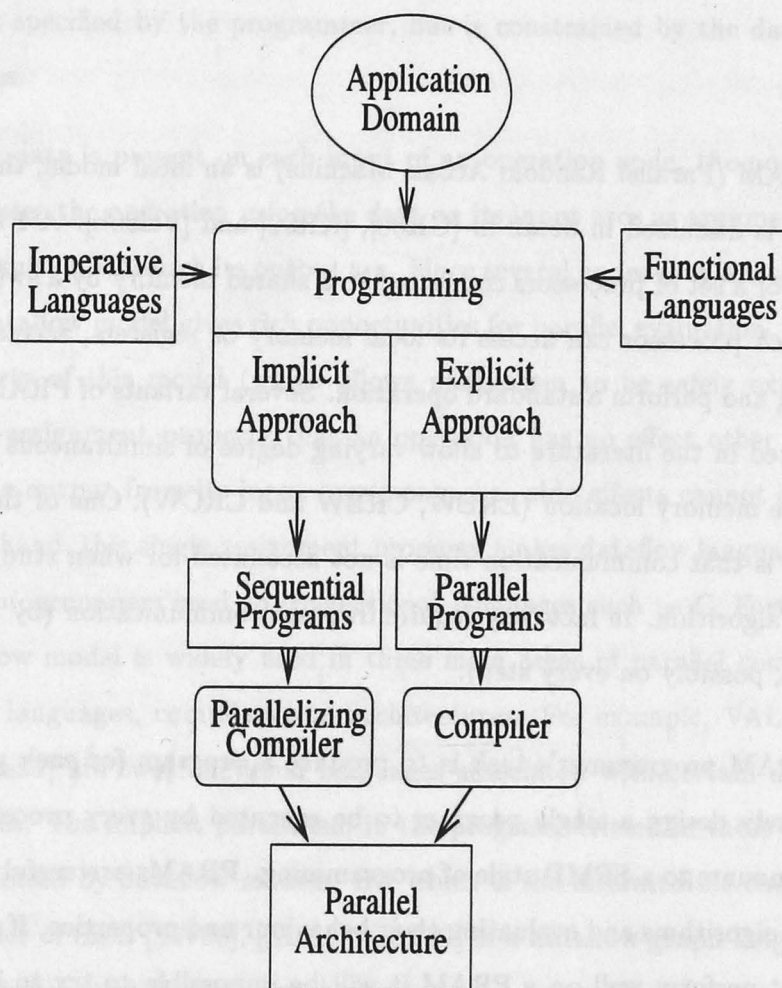


Figure 2.1: The paradigms of Parallel computing

either theoretical or physical. In the last three decades, to improve the expressive power of parallelism, great efforts have been made to develop new models and languages. The requirements for exploiting parallelism, studying the complexity of algorithms and achieving portability of programming, have resulted in a significant advance in this area due to the introduction of *machine-independent parallel computing* [Lew94], [Ski90], [FS92], [Val90a].

- **PRAM**

The PRAM (Parallel Random Access Machine) is an ideal model, that is widely used. It is discussed in detail in [GR88], [KR90] and [Val90b]. A PRAM model consists of a set of processors connected to a shared memory by a switch. In unit time, each processor can access its local memory or registers, access the shared memory, and perform a standard operation. Several variants of PRAM have been introduced in the literature to allow varying degree of simultaneous reference to the same memory location (EREW, CREW and CRCW). One of their common features is that communication time is not accounted for when studying a given parallel algorithm. In fact they require frequent communication (by using shared memory, possibly on every step).

The PRAM programmer's task is to produce a program for each processor, or more likely design a single program to be executed by every processor in what often amounts to a SPMD style of programming. PRAMs are useful for studying parallel algorithms and evaluating their behaviour and properties. If an algorithm does not perform well on a PRAM it will be impossible to try to implement it on a realistic, but weaker, parallel architecture. The suitability of PRAM as a universal model was been examined by Skillicorn [Ski91]. He has concluded that the PRAM model is universal over the classes of tightly coupled and hypercuboid multiprocessors, but not universal over the constant-valence topology multiprocessors and SIMD computers. The problem with efficient implementation of the PRAM model on these systems is that the amount of communication generated at each step can easily overrun the bandwidth provided by the topology.

- **Dataflow model**

The dataflow model of computing gets around the problems encountered in introducing parallelism in the traditional control flow model, by using a different viewpoint of the process of computation. Computation is represented by a directed dataflow graph [DK82] [Sha85] with the nodes as operations and the arcs as paths carrying data tokens. In a dataflow program the ordering of operations is not specified by the programmer, but is constrained by the data interdependencies.

When data is present on each input of an operation node, the node *fires*, i.e. it computes the operation using the data on its input arcs as arguments and passes the result out through its output arc. Since several nodes may fire simultaneously, the dataflow model gives rich opportunities for parallel evaluation. An important property of this model (which allows parallelism to be safely exploited) is the single-assignment property that an operation has no effect other than computing the output from its input arguments, i.e. side effects cannot occur. On the other hand, this single-assignment property makes dataflow languages unpopular with programmers used to conventional languages such as C, Fortran, etc. The dataflow model is widely used in three main areas of parallel computing: functional languages, compilers and architectures. For example, VAL [McG82] and Id [Eka91] are two functional languages associated with certain dataflow architectures. The implicit parallelism in the program written in these two languages is exploited by dataflow models. IF1 which is the intermediate code used by the compiler of Sisal [SW85], [Ske91] [CBF91] is a dataflow graph language. In fact, the dataflow technique is popular and is used by various compilers for optimisation.

- **Bird-Meertens Formalism**

The Bird-Meertens Formalism (BMF) is an approach to software development and computation based on categorical data types and associated operations. This theory is initially based on the theory of lists [Bir87] and developed in [Spi89], [Ski93] and other papers. The theory of lists adds a number of the second-order functions

to the base algebra, which includes *map* (\times), *reduce* (\cdot), *directed reduce* (\nrightarrow), *prefix* (\cdot), *filter* (\triangleleft) and so forth. BMF encourages software development by equational transformation which can be applied for optimisation, or regarded as rewrite rules [Mal90]. BMF does not directly express low-level parallelism physically; it is the compiler's task to implement the operations in parallel. Communication in this model is restricted to a set of functions, each of which encapsulates a particular communication pattern requiring only a constant size of neighbourhood locality [Ski91]. Both parallelism and communication are thus hidden from the direct concern of the programmer. A strategy for building cost calculi for skeleton-based programming languages based on the Bird-Meertens formalism is presented in [SC94] so that trade-offs in software design can be explored before implementation. A major drawback of BMF is that it is applicable to only data-parallel algorithms.

2.1.2 Parallelising compilers

In the implicit approach, also called the *conversional approach*, a parallelising compiler first detects parallelism in a sequential or functional program, and then converts the program into executable code for a specific parallel architecture. Recent research has underlined the importance of exploiting both control and data parallelism in a single compiler framework that can map a single source program in many different ways onto a given parallel machine. One of the most difficult problems for parallelising compiler techniques is how to find parallelisable execution code based on efficient and exact data dependence analysis [Pug92], [Lam74], [Ban90], [Lil94], [Mea91]. Despite some progress in the last two decades, a really sophisticated parallelising compiler is unlikely to be developed in the near future. One of the main reasons for this is the lack of sophisticated techniques and tools to fully exploit parallelism. Hence, at present, program parallelisation is only based on an incomplete knowledge background of parallel properties of application domains. An important question is whether the research frontiers in parallelising compilers are currently pushing the limits of traditional data dependence analysis. There are several complex tradeoff factors between control and data parallelism, depending on the nature of the program to be executed and the

performance parameters of the target parallel machine. This makes it difficult for a compiler to select a good mapping for a control and data parallel program, because any such rational selection has to be based on the performance evaluation of different solutions. Further discussion on the state of art of data dependence testing is given in Chapter 6.

A survey on compiler transformations for high-performance computing by Bacon, Graham, *et al* [BGet al94] shows that the current parallelising compilers lack an organising principle that allows them to choose how and when the transformations should be applied. In particular, due to the absence of a strategy for unifying transformations on parallel architecture, most high-performance applications currently rely on the programmer's skills rather than the compiler to manage parallelism. Since efforts to automatically parallelise sequential languages have not been very successful (as people have expected), the focus of research has shifted to compiling other non-traditional languages, such as functional or parallel programming languages, where the programmer needs to express directly or indirectly the parallelism needed [MPC90].

2.1.3 Functional programming

Functional programming has attracted research attention for more than thirty years. Its clean semantics make it an attractive vehicle for investigating various programming language concepts. Church's lambda-calculus [Chu46], [Bar81] is the formal basis of all functional programming languages.

An expression in the pure lambda-calculus is composed solely from three syntactic objects: function abstractions, function applications and identifiers. An application is reduced by replacing occurrences of the function's formal parameter with copies of its argument (β -reduction). In *applicative order* reduction, the argument in a function application is reduced prior to doing the β -reduction. In *normal order* reduction the β -reduction is performed directly with the unevaluated argument. Regardless of which reduction order is used for evaluating lambda-expression, the result remains identical. This important property of the lambda-calculus implies that a lambda-expression can be evaluated using any order of reductions. In principle, performing reduction in parallel is allowed. One of the main features of parallel functional programming is that

the programmer is able to view a program as a collection of high level units ignoring computational details. The lambda-calculus has a natural parallel semantics or meaning, since no particular execution order is enforced. Thus, functional programs contain implicit parallelism at all levels.

It is claimed in [Szy91] that functional programming is a convenient basis for the development of the parallel programming languages and the compilers in designing parallel programs. In practice, functional languages are used for parallel processing in two different approaches.

- The **purely implicit approach**, where an ordinary functional language with no parallel additions whatsoever is implemented on a parallel architecture. To exploit parallelism in such a program, a compiler needs to abstract *useful* parallelism and organise all computation units effectively on a target architecture. The typical techniques used for these compilers are the *dataflow model* and *parallel graph reduction*. The main problem with this approach is that compilers seem to have difficulties in deciding when a parallel evaluation is worthwhile, and when a standard sequential evaluation is preferable. To efficiently exploit parallelism, the dataflow model has been extended to support threads of appropriate grain size, allowing hybrid dataflow and control flow evaluation [GGB93]. While simple and sound, there are doubts as to whether the extended multithreading of a dataflow system is as attractive as originally thought. Culler *et al* point out two fundamental limitations [CSE93]: latency tolerance is limited in practice and local scheduling policies are inadequate. Many parallel graph reduction based systems can be considered to be not quite purely implicit since they rely on various degree of programmer annotations to identify the useful parallelism.
- The **purely explicit approach**, where a functional language is given extra syntactical constructs through which the programmer can instruct the compiler that parallel evaluation should take place. Here the burden of explicit parallel programming is put back onto the programmer, whose skills and knowledge is instrumental to the performance of implementation. As a result, the programmer is required to indicate opportunities for fruitful parallel evaluation with various

annotations and also to specify how it is to be performed on certain architectures if necessary.

Although functional languages provide abstractions, determinacy, succinctness and ease of expression, they are not commercially popular since their efficiency does not match the imperative languages. Moreover, there is a trade off between *expressiveness* and *parallelism* because if all the parallelism is exposed in the program, the program tends to become cumbersome and less succinct, particularly for large applications. Some compromise between *expressiveness* and *parallelism* is necessary for efficiency. In summary, it is certain that the fruits of functional languages, however attractive they may appear, cannot be reaped until definitive efficiency comparisons with the conventional computing are shown.

2.1.4 Programming paradigms

Regardless of the target parallel architecture, parallel programs must harmoniously coordinate two or more program segments to assure correctness, as well as high speed. This is the challenge of parallel programming. Exactly how parallelism is achieved is largely determined by the particular *paradigm* used by the programmer and programming language used. With the development of PPM, in order to reduce the difficulty of complexity management in parallel programming, including expressing parallelism, partitioning, message passing and synchronising, a number of special programming paradigms have been developed, which have different programming styles with different programming philosophies.

1. UNITY

UNITY is introduced as a foundation of parallel programming design [CM88]. A UNITY program describes *what* should be done in the sense that it specifies the initial state and the state transformation (i.e., the assignments). A UNITY program does not specify precisely *when* an assignment should be executed — the only restriction is a rather weak fairness constraint: Every assignment is executed infinitely often. Neither does a UNITY program specify *where* (i.e., on which processor in a multiprocessor system) an assignment is to be executed, nor to which process an assignment belongs. A UNITY program does not specify *how* assignments are to be executed or

how an implementation may halt a program execution. UNITY separates concerns between *what* on the one hand, and *when*, *where*, and *how* on the other. The *what* is specified in a program, whereas the *when*, *where*, and *how* are specified in a mapping. By separating concerns in this way, a simple programming notation is obtained that is applicable for a wide variety of architectures. Of course, this simplicity is achieved at the expense of making mappings immensely more important and more complex than they are in traditional programs. In this approach no explicit control of scheduling or communications is required. UNITY is viewed as a language for reasoning about computation rather than executing computation. The departure point of UNITY from the conventional view of programming is to attempt to decouple a program from its implementation. This decoupling leads that the correctness of a program is independent of the target architecture and the manner in which the program is executed, hence, a mapping becomes a description of how programs are to be executed on the target machine. The philosophy of UNITY shows the impetus of developing a different theoretical foundation for parallel programming from the conventional utilisation of von Neumann architectures.

2. Linda

Linda [CG89],[CG90] (a coordination language) is based on a shared, associative object memory—a *tuple space*. This tuple space contains an unordered collection of ‘tuples’, where each tuple contains an ordered collection of data fields. A tuple in the space is either *active* or *passive*. An active tuple is a process, destined to turn into an ordinary passive tuple upon completion. Tuples are removed using an associative matching protocol resembling the *select* operation in a relational database.

Linda provides a radically uncoupled model of parallel computing. As a language, it places *simplicity* uppermost so that *uncoupling* has a space-wise and time-wise aspect, that is, processes may communicate in Linda although they are mutually anonymous and their lifetimes are disjoint. Linda requires the explicit expression of parallelism and communication (by accessing tuple space) but abstracts from synchronisation.

3. Skeleton-Based Programming

It is observed that parallel programs written in explicitly parallel languages consist of two different kinds of codes, *task specific code* implementing the individual steps

of the algorithm, and *code for structuring* the program into patterns of computation and communication associated with specific architectures for parallel execution. It is typically only in the latter kind of code that there is a need to deal with low-level machine dependent design problems. Moreover, the typical parallel infrastructures of programs have been classified into a few well-known parallel paradigms (patterns), each determining the operation and coworking of groups of processors.

In [Col89],[Det a193] the use of patterns is suggested as is a fundamental concept in parallel programming. From the programmer's perspective the skeleton is a high level semantic description of an algorithmic operation with gaps left for problem specific procedures and declarations. To use such a skeleton, the programmer must fill these gaps with parameters. The implementation of a skeleton is, however, completely hidden from the programmer. By using skeletons there will be parallelism implicit in the program which can be potentially be exploited on a parallel architecture by a compiler. Consequently, the portability of the program is enhanced in this paradigm.

In summary, during the last three decades, parallel programming techniques have been developed with great effort, providing various advanced vehicles to let programmers make use of their personal knowledge and skills because parallelising compilers have not been so successful as expected. Due to the different strengths and weaknesses of these techniques and approaches, we cannot predict which one of these will dominate. The attention of our research is limited to examining the current state of the art in the literature, identifying some inadequate components in PPM and possibly providing improved techniques for PPM.

2.2 Problems

Parallel computers are used almost exclusively for very specific areas of computing. It is widely believed that the principal reason why parallel computing has not had a major commercial breakthrough is due to the lack of adequate software development methodologies and tools [Col92], [Pan91], [KN93]. To expand the use of parallel computers, a fundamental requirement is that these computers are easy to use like uniprocessor computers. In fact, it is not difficult to write a program with cer-

tain parallel properties, but it is difficult to develop a parallel program with a good performance.

Nobody wants parallelism. What we want is performance. It is the fact that going to parallelism is the only way to continue to enhance performance that makes parallelism a necessity [Pan91].

Ken Neves of Boeing Computer Services

Because the von Neumann architecture is universal for sequential computation, an implementation of an algorithm on one manufacturer's uniprocessor will differ in speed by no more than a constant factor from that on another's [Ski90]. In parallel programming, however, the situation is quite different. Whether an implementation of parallel computing is successful should be decided by both correctness of computation and the performance achieved. An implementation with certain parallel features does not mean a success if some more significant parallelism is missed out. In other words, it is possible that there could be another solution making use of that parallelism to achieve a better performance. In such a situation, we believe that current parallel programming methodologies are empirical or *ad hoc* since a number of different solutions to a given problem, with widely different performances, can be arrived at by different people, even if based on the same architecture.

In most cases, the first workable solution is likely to be adopted. If the chosen solution is poor, this will not be a cost-effective choice in the long run. Hence, programming methodologies should provide a guideline to convince programmers why a chosen solution is better than others, or why an architecture used is better suited for a problem than others. Unfortunately, most current research papers and techniques in the literature do not deal with such practical guidelines. The problem discussed here, actually, is how to reduce the risk of performance failures using proper methodologies. In the explicit approach most languages and models are used to express parallelism in a problem. Thus, the parallelism must be known to the programmer before it is expressed. In such a case, it is found that a successful implementation of parallel computing is determined by two main characteristics of the programmer, namely, *programming skills* and *knowledge of parallel properties* in the problem. If a programmer

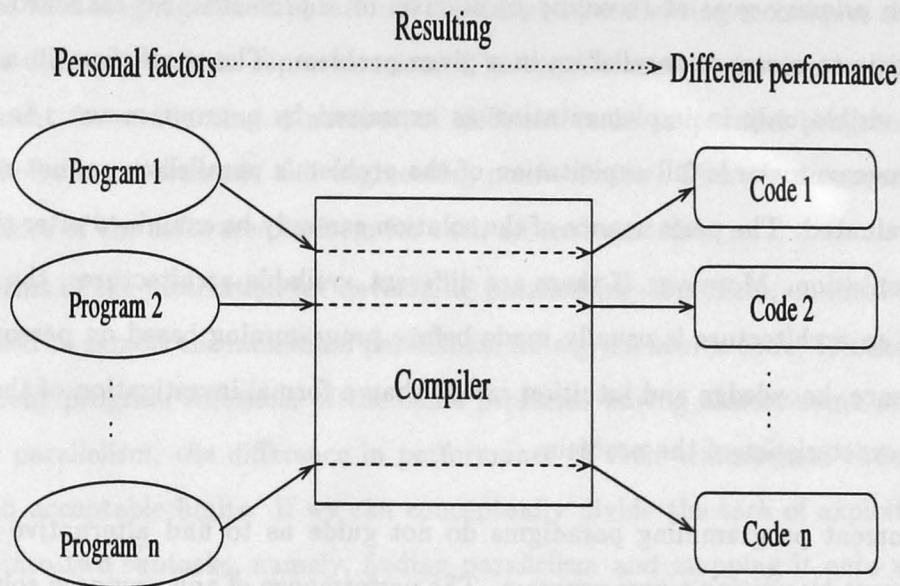


Figure 2.2: Different programming solutions lead to different performance.

lacks excellent programming skills or understanding of the parallelism in the problem, the risk of performance failures always exists. This situation is illustrated in Fig. 2.2. Different programmers produce diverse solutions in which personal experience is heavily involved. If no special optimisation is applied in transformation by the compiler, the difference in performance of the two solutions proposed by programmers is unlikely to be improved in the compiled codes.

The main reasons for this situation can be summarised by the following four points:

- Programming languages permit to express an algorithm to solve a problem; parallel programming languages are no exception. The parallelism specified in a parallel program is known to programmers and is constrained by using certain languages. The same is true of most parallel computation models and functional parallel languages, no matter whether they are machine-dependent or machine-independent. In the diagram shown in Fig. 2.2, the constraints from the techniques used in programming and personal experience of programmer are introduced at the programming stage and finally contribute to the performance.

- It is observed that none of those languages and models has been developed with the primary goal of *revealing parallelism* in a problem. No standardised scale exists to measure parallelism in a given problem. The parallelism in a problem is visible only in implementation as expressed by a programmer. As a result, progress towards full exploitation of the problem's parallelism cannot readily be evaluated. The performance of the solution can only be examined after the implementation. Moreover, if there are different available architectures, the selection of an architecture is usually made before programming based on personal experience, knowledge and intuition rather than a formal investigation of the parallel characteristics of the problem.
- Current programming paradigms do not guide us to find alternative solutions except by writing a new program. The performance of any proposed solution can only be examined after implementation.
- Using existing concepts and terminology, we are not able to explain some of our concerns. The concepts, *explicit parallelism* and *implicit parallelism*, are suitable to indicate the state of individual parallel properties in a program rather than the general feature of the parallelism because
 - Explicit parallelism does not explain the differences between parallel features in two solutions;
 - Implicit parallelism cannot determine to what extent a given solution can be optimised.

As a result, it is difficult to introduce proper measures to check if a parallelising compiler can successfully exploit parallelism for a given source code.

Machine-independent parallel programming is important to make the programming task easy and address the portability issues. It does not help to eliminate the subjective aspects that affect the performance.

In order to obtain suitable methodologies for parallel computing, attention has been paid to topics, such as parallelism profiling, program derivation and performance

prediction. It is not clear whether these topics could be naturally and successfully combined with languages, computational models and parallelising techniques since they use quite different tools in representation. The requirement to develop an integrated environment for conducting a number of different tasks in parallel programming is inevitable from a systematic and engineering point of view. There is no consensus yet on structure of the software platform for such an environment.

In terms of the motivation for developing parallelising compilers, an ideal compiler is supposed to exploit the maximum parallelism for a given source code. In other words, for different program solutions of the same problem, having almost same amount of inherent parallelism, the difference in performance of their transformed codes should be within acceptable limits. If we can conceptually divide the task of exploiting parallelism into two subtasks, namely, finding parallelism and mapping it onto a specific architecture, then whether a compiler could find out a certain amount of parallelism for both solutions becomes a key issue. The survey by Bacon *et al* [BGet al94], shows that the research progress in this field has not indicated the availability of such a smarter parallelising compiler in the near future.

2.3 Related research

The problem we have raised is related to a number of fundamental issues in parallel programming, like languages, computational models, parallelisation and so on. Some issues have been recognised and studied by several researchers around the world.

1. Pomsets approach

To model concurrency with partial orders, Vaughan Pratt [Pra86] introduces a single hybrid approach having a rich language that mixes algebra and logic and having a natural class of models of concurrent processes. The language is extracted from three existing approaches, that is, formal languages, partial orders and temporal logic. The heart of this approach is a notation of partial string derived from the view of a string as a linearly ordered multiset by relaxing the linearity constraint, thereby permitting partially ordered multisets or *pomsets*.

As a formal representation of processes, a *pomset* is the isomorphism class of an *lpo*

(labelled partial order), denoted $[V, \Sigma \leq, \mu]$ by Gischer [Gis84]. Unlike the 'operational' approach using reductions between expressions advocated by Milner [Mil83], Pratt's approach is *denotational* or *extensional* in the sense that it uses a concrete mathematical model of a computational behaviour, along with operations on behaviours that yield a particular algebra of behaviours.

The above approach has the advantages of being straightforward, involving fewer artificial constructs than many computing models of concurrency, and is applicable to a wider range of types of systems [Pra86]. As a theoretical demonstration, this approach shows how to use abstract concepts of partial ordering and multisets to exploit parallelism.

In the recent work of Pratt all relational structures between time and information are mathematically realized in the phase spaces of the Chu space [Pra94]. It is observed, theoretically, that the passage from sequential non-branching computation to concurrent and branching computation can be understood as the relaxing of the linear structure to looser (weaker) spaces from both temporal and spatial points of view.

2. Complex systems

In the last ten years Fox's groups at the Caltech Concurrent Computation Program (C3P), and more recently the Northeast Parallel Architectures Centre (NPAC) at Syracuse University have made great efforts in improving the understanding, concepts and techniques of parallel computing [Fox90], [Fox92]. They consider parallel computing as the mapping of one complex system — typically a model of the world — into another complex system — the parallel computer. Thus, the use of parallel computers can help improve our understanding of complex systems, and the converse is also true — we can apply techniques used for the study of complex systems to improve our understanding of parallel computing.

Fox's mapping theory of parallel computing is based on the *space-time picture* of parallel computing, that is, *spatial properties* and *temporal properties*. The spatial properties of the problem are determined by the concepts like *system size*, *geometry* and *information dimension*, and mapping decisions like *problem decomposition* and *allocation*. The temporal properties include static and dynamic *scheduling*, *synchro-*

nising and other dynamic factors. In terms of these properties a qualitative theory of the architectures of problems is developed, which is analogous to Flynn's well-known classification of parallel architectures. Fox discusses how the spatial (data) parallelism of the problem becomes the temporal structure in software. He concludes that the failure of most parallelising compilers is caused by missing the point that the parallelism comes from spatial and not control (time) structure [Fox92].

Most languages do not express and preserve space time structure. Consequently, the efforts being made in NPAC are to develop languages that can express better the problem structures. High Performance Fortran (HPF) is one of the languages developed for this improvement. With HPF certain structure features of the problem can be expressed by *data distribution directives* (TEMPLATE, ALIGN, PROCESSORS, DISTRIBUTE, DYNAMIC and REDISTRIBUTE), *parallel statements* (INDEPENDENT and FORALL) and *intrinsic functions* and the *standard library* [Kea94].

3. GAMMA programming model

The GAMMA formalism [BM90] [BM93] in which programs are described as multiset transformation was introduced to support a systematic program derivation method in parallel computing. The main feature of GAMMA model is the function:

$$\Gamma((R_1, A_1), \dots, (R_m, A_m))(M) = \text{oneof}(\Gamma^s((R_1, A_1), \dots, (R_m, A_m))(M))$$

where

$$\begin{aligned} &\Gamma^s((R_1, A_1), \dots, (R_m, A_m))(M) = \\ &\quad \text{if } \forall i \in [1, m], \forall x_1, \dots, x_n \in M, \neg R_i(x_1, \dots, x_n) \\ &\quad \text{then } \{M\} \\ &\quad \text{else } \{M' \mid \exists x_1, \dots, x_n \in M, \exists i \in [1, m] \text{ such that} \\ &\quad \quad R_i(x_1, \dots, x_n) \text{ and} \\ &\quad \quad M' \in \Gamma^s((R_1, A_1), \dots, (R_m, A_m)) \\ &\quad \quad ((M - \{x_1, \dots, x_n\}) + A_i(x_1, \dots, x_n))\} \end{aligned}$$

The function R is the reaction condition (or condition text); it is a boolean function indicating when some of the elements of the multiset M can react. The function

A (action text) describes the result of this reaction. Hence, if one or several reaction conditions hold for several non-disjoint subset at the same time, the choice made among them is nondeterministic. This aspect of GAMMA provides for *competitive parallelism* [CG89], [MK95]. However, if the reaction condition holds for several disjoint subsets at the same time, the reactions can take place independently and simultaneously; this aspect provides for *cooperative parallelism*.

The motivation of GAMMA is to express *logical parallelism* of a problem before an implementation in which *physical parallelism* is achieved. The confusion between *logical parallelism* and *physical parallelism* is part of the heritage of several decades of imperative programming. It is suggested that in parallel programming we should be able to build in the first place an abstract version of the program that should be free of artificial sequentiality [BM93].

4. Other earlier studies

Also there are a number of previous studies on instruction-level parallelism, involving a wide variety of machine models and applications, to measure the limits of parallelism which may be exploited in a program [KMC72], [NF84], [Kum88], [Wal91], [LW92], [TGH92b], [AS93]. Lam and Wilson studied the limits of control flow on parallelism. They demonstrate that substantially higher parallelism can be achieved by relaxing the constraints imposed by control flow using control-dependence analysis and speculative execution [LW92]. Theoblad, *et al* developed an experimental testbed to examine the limits of parallelism in a program and its smoothability on a given model. The result of their experiments shows that some applications intuitively seem to have much more potential parallelism than found by current techniques. Thus it is suggested that, in many cases, large-scale parallelism will not be achieved merely by transliterating existing imperative-language programs [TGH92a], [TGH92b]. One of the common features of these studies is to measure or examine performance and potential parallelism in programs as they are executed. The concept of *upper bounds* for potential parallelism is associated with the limitations of models and architectures.

Kumar has developed a software tool (COMET) that measures parallelism quantitatively when a FORTRAN code is executed on an ideal parallel machine, and found the potential for higher levels of parallelism in scientific numerical programs [Kum88].

The measurements obtained from COMET would aid the evaluation of various parallel processing systems by providing the right set of assumptions for the extent of parallelism presented in applications. Also Kumar found that the characterisation of parallelism is more difficult and is handled at a cursory level.

Tetra [AS93] is a multi-platform instruction trace analyser developed by Todd Austin. As a tool for evaluating serial program performance under the resource and control constraints of fine-grain parallel processors, *Tetra's* primary goal is to quickly generate performance metrics for yet-to-be designed architectures. The core of this tool is dynamic dependence analysis through the *unconstrained dynamic dependence graph* (we use *uDDG* to distinguish another widely used concept *DDG* for *data dependence graph*). The *uDDG* is also called *dynamic dataflow graph* because construction of such a graph is based on the data dependencies realized in the analysed program's execution. This approach has significant difference from other studies which are mainly based on the context of programs. According to architecture features specified, *Tetra* produces an execution graph that is finally analysed to evaluate the serial program's performance under the specified architecture model.

2.4 Thesis Objectives

2.4.1 The Proposed improvement in PPM

The central problem raised in the last section is about the deficiency that exists in the current representation techniques used for parallel computing. A possible solution to the problem is to develop techniques to exploit parallelism before or when a program is developed. These techniques can be either improvement to the existing representation tools or development of a new representation that will have a special facility to reveal parallelism and can be easily integrated with other parallel programming tools.

A language or a computational model in programming is typically used as a tool to express a real world problem in a particular form for computation. The parallel properties of solving a problem are exploited and expressed in a specification or a program. While two specified solutions of the problem may be quite different in performance. If there is no smart compiler that could optimise two solution of the same problem

so that the performance difference between them could be reduced to an acceptable range, then the issue raised would remain unsolvable with current techniques. Both imperative languages and functional languages have already been designed for their special features. The experience with language development has shown that it is very difficult to make a language meet many different requirements of parallel computing, since they may be contradictory.

Previous studies on machine-independent parallel computing, logical parallelism [BM93] and the limits of parallelism in a program [TGH92a], show that parallelism can be investigated in different ways. The task of finding parallelism is typically carried out by a compiler or a program. The methods used to exploit parallelism is always described in a traditional way that one can use the methods to express parallelism. While, it has never been explained whether or how the methods can ensure that for a given problem parallelism is expressed correctly from a good performance point of view. The problem here is that the success of using these methods is determined by the person who uses them. In other words, the method itself cannot guarantee a success when it is applied.

In order to improve this deficiency, we propose to develop some new techniques to study parallelism in solving a problem as a relatively independent task from mapping a real world problem onto a specific architecture. Such a technique should have a special power to reveal parallelism in a non-traditional manner. We believe that a *programmer-view independent* manner to express parallelism should be advocated such that at the programming stage parallelism of a given problem can be revealed without any constraints. This goal is not achievable by using conventional languages, thus, it is necessary to define what this new technique is. Moreover, the relation between the new technique and those existing concepts or tools should be set up properly. The effort we are making is to enhance relevant techniques in PPM rather than developing a stand alone one that would be better than others.

2.4.2 Parallelism revelation models

As discussed earlier, in conventional parallel programming approaches [CM88], [CG89], [BM93], [Sab88], [Ski91], [CG90], the parallelism achieved by a particular

program is based on: (i) the models, architectures and techniques used as guidelines; and (ii) the subjective view of programmers who produce different solutions using different parallelism features. Therefore, we call this kind of parallelism **programming solution-based parallelism** or **subjective parallelism**. The word *subjective* itself does not mean anything on whether the performance of a solution is good or not, since it only indicates that the solution is designed subject to certain constraints. Different constraints introduced in program design lead to different performances.

To improve PPM, we believe exploiting parallelism needs some kind of support that is based on a special model, called a *parallelism revelation model*. The most important feature of this model is the capability to reveal parallelism in a *programmer-view independent* manner. In addition to the key properties possessed by a parallel computation model suggested in [Ski90], the special criteria for this kind of model are as follows:

- It should be developed with a primary goal to reveal parallelism of a problem.
- Its representation should be grain-dependent since parallelism analysis is based on fine-grain representation where most data parallelism can be found, we should then move on to a medium-grain or coarse-grain level for control parallelism.
- It should support parallelism inference.
- It should support optimisation of a solution.
- It should provide mechanisms to reconstruct solutions.
- It should have potential applications as a kernel or foundation for parallel programming so that certain related techniques and tools can be integrated.

2.5 Summary

In summary, we quote from Chandy and Misra [CM88]:

The basic problem in programming is managing complexity. We cannot address that problem as long as we lump together concerns about the core

problem to be solved, the language in which the program is to be written, and the hardware on which the program is executed. Program development should begin by focusing attention on the problem to be solved and postponing considerations of architecture and language constructs.

To achieve this goal, in this dissertation, the discussion and study of parallelism of a given problem (see Chapter 3 to 6) are separated from architectures and implementation (a brief discussion of mappings is included in Chapter 7). The parallelism revelation model is proposed as a new tool to exploit inherent parallelism that is independent of the programmer's view.

Chapter 3

ABCOM— A Parallelism Revelation Model

Presented in this chapter are a parallelism revelation model, called ABCOM, and its properties.

3.1 A Puzzle — Parallelism in a Problem

Parallelism is realisable in various forms — lookahead, pipelining, vectorisation, concurrency, simultaneity, data parallelism, partitioning, interleaving, overlapping, multiplicity, replication, time sharing, space sharing, multitasking, multiprogramming, multithreading, and distributed computing at different processing levels. No matter how they differ in their properties, a common feature of all is to solve a real world problem on a particular architecture with an appropriate parallelism.

Parallelism is a qualitative concept when we say a problem is solved in parallel. However, parallelism has also a quantitative aspect when the problem is expressed in a particular program with parallel properties, which determine the performance of the program. Parallelism is inherent in a definition of the problem, remains dormant until it can be expressed in a notational form (such as a program or specifications). A definition of a real world problem contains a data domain and operations associated with the data. Parallelism characteristics for solving the problem are available based on a combination of these two concepts and time. The objectivity of parallelism means the parallelism in a given problem is inherent. But it can be partially and subjectively expressed in a specific representation tool (for instance, a language). Hence the amount of the parallelism expressed can be quite different from one solution to another due to

```

real A(n × n), B(n × n) and C(n × n)
real temp(n)
for i=1 to n do
  for j=1 to n do
    temp(1:n) = A(i, 1:n) × B(1:n, j)
    c(i, j) = Sum(temp(1:n))
  end
end
end

```

Figure 3.1: An parallel algorithm for matrix multiplication

subjective factors and technical constraints involved. Thus, to answer to what extent the parallelism of a problem can be exploited we have to know whether those factors and constraints could be avoided or eliminated from programmer's expression as far as possible.

To understand the subjective parallelism, let us start with the example of a matrix multiplication $C = A \times B$, where

$$c_{ik} = \sum_j a_{ij} b_{jk}.$$

Here the data domain is composed of the elements of three matrices. Using a SIMD architecture, one can solve this problem using different parallel algorithms. One possible solution is shown in Fig. 3.1 where data parallelism is achieved in a vector-wise manner. Another solution [Can69], [Cor90] is illustrated in Fig. 3.2. It can be speeded up by using a systolic array for more data parallelism. In addition, we can decompose [FJea88] the whole matrix into a number of sub-matrices for parallel computing. The (subjective) parallelism exploited in each of these algorithms is different, and results the difference in performance due to the subjective selection of parallel properties of the problem.

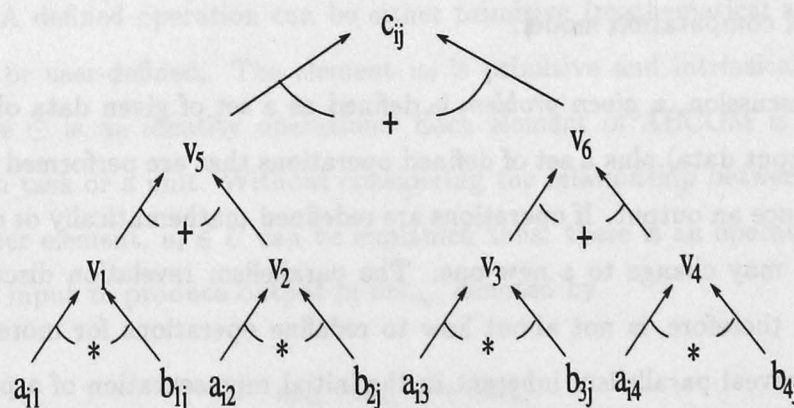
A computation task can be illustrated using computation graphs, where considerations on how to handle data manipulation in a program and storage on a particular architecture are ignored. The matrix multiplication of $A(3 \times 4) \times B(4 \times 4)$ is shown in such a graph in Fig. 3.3 to compute an element c_{ij} of $C = A \times B$.

```

real A(n × n), B(n × n) C(n × n)
real a1(n × n), b1(n × n)
integer skew(n)
/*Set up skewing vector: [0, 1, 2, ..., n - 1] */
for i = 1 to n do
    skew(i) = i - 1
end
/* Perform the initial skewing */
a1 = cshift(A, 2, skew)      /*skewing along dimension 2 */
b1 = cshift(B, 1, skew)     /*skewing along dimension 1 */
/* Loop to accumulate the dot product at each iteration */
C(n × n) = 0
for i = 1 to n do
    C(n × n) = C(n × n) + a1(n × n) * b1(n × n)
    a1 = cshift(a1, 2, 1)    /*skewing for next iteration*/
    b1 = cshift(b1, 1, 1)   /*skewing for next iteration*/
end

```

Figure 3.2: Cannon's systolic algorithm for matrix multiplication

Figure 3.3: Computation graph for c_{ij}

There are totally 3×4 computation graphs like this for this example. They are computationally independent, yet share certain input data. Using this representation, in a general case, if $C(m \times p) = A(m \times n) \times B(n \times p)$, it is seen that all $m * n * p$ multiplications required could be performed in parallel and all $(m * (n - 1) * p)$ additions can be done in $\log_2 n$ steps (by using a CREW PRAM based model with $m * n * p$ processors). Note computation here is executed in a dataflow style with maximal parallelism. The above example reveals the following:

1. Each element a_{ij} of A or b_{jk} of B is used p or m times in multiplication. These multiplications are independent and can be carried out in parallel if there is no data-access conflict. The computational graph in Fig. 3.3 provides not only a machine-independent but also program-solution independent approach to express a computational schema.
2. The parallelism in a problem is objectively determined by the spatial structure (data domain) and associated temporal logic of computation. Using parallel computation models and architectures with different topological structures, we can develop a number of different solutions for the same problem. The subjective parallelism achieved in a program is extracted from and is also constrained within the objective parallelism. Current parallel programming always expresses a subjective solution with the parallelism realised by a programmer based on a particular parallel computation model.

In our discussion, a *given problem* is defined as a set of given data objects (input data and output data) plus a set of defined operations that are performed on the input data to produce an output. If operations are redefined mathematically or conceptually, the problem may change to a new one. The parallelism revelation discussed in this dissertation, therefore, is not about how to redefine operations for more parallelism. Our aim to reveal parallelism inherent in the initial representation of a problem. The difference between the objective parallelism of a problem and subjective parallelism realised in a program provides a measure for the performance of a program.

Also in this thesis, by the term '*solution*' we mean any form of representation to express an algorithm to solve a problem. A program is a special solution, while, a

solution need not be a program.

3.2 Definitions and Properties

The **abstract computational tuple-space model** (ABCOM) is defined as a finite symbolic data space in which computational information is represented into a set of computation units, called **elements** of the space. ABCOM is formally defined thus:

Definition 3.1 *The abstract computational tuple-space U consists of a set of quadruples, called computation units (or elements) of the tuple space, that is, $U = \{u_0, u_1, u_2, \dots, u_n\}$ with $u_0 \vdash (\odot, \{\emptyset\}, \{\emptyset\}, 0)$ and $u_i \vdash (op_{u_i}, in_{u_i}, out_{u_i}, ex_{u_i})$ for $i = 1, 2, \dots, n$, where*

- op_{u_i} is a defined operation;
- in_{u_i} is a set of input data objects;
- out_{u_i} is a set of output data objects;
- ex_{u_i} is a logical execution time.

Here ‘ \vdash ’ means ‘perform’; for sake of simplicity, we usually use ‘.’ instead of ‘ \vdash ’ in discussion. A defined operation can be either primitive (mathematical and relational operations) or user-defined. The element u_0 is primitive and intrinsically contained by U , where \odot is an identity operation. Each element of ABCOM is defined as a computation task or a unit. Without considering the relationship between an element and any other element, $u_i \in U$ can be explained thus: there is an operation op_{u_i} that uses in_{u_i} as input to produce output in out_{u_i} , denoted by

$$op_{u_i} : in_{u_i} \mapsto out_{u_i}.$$

For a given data object x if there is $x \in in_{u_i}$, it means u_i has a ‘read’ access to x ; while if $x \in out_{u_i}$, then a ‘write’ access to x . The order of data objects in in_{u_i} and out_{u_i} is determined by the relations associated with op_{u_i} .

The execution order of elements in ABCOM is controlled by a *logical clock* for counting logical steps of the computation; each element takes a unit of logical time for execution. It means that $ex_{u_i} = 1, 2, \dots, n$ if we define the start time of the clock is equal to ex_{u_o} . When ex_{u_i} is equal to current value of the clock, op_{u_i} is executed, we say u_i is **performed**. If there is a partial order $ex_{u_i} < ex_{u_j} < ex_{u_k}$ with $ex_{u_i} = 2$, $ex_{u_j} = 3$ and $ex_{u_k} = 7$, for example, we say u_i , u_j and u_k are performed at timesteps 2, 3 and 7 respectively. If $ex_{u_i} = ex_{u_j}$, it means that u_i and u_j are performed in parallel.

If there are two or more elements in U , the relationship between element u_i and other elements is decided thus: If there is a data object $x \in in_{u_i}$ or $x \in out_{u_i}$, and also appears in another element u_j , then these two elements are computationally related. In this case, there may be an intentionally specified partial order between them to perform a computation task that sequentially combines u_i with u_j . That is, u_i is designed to be executed at a particular time (ex_{u_i}):

$$op_{u_i} |_{ex_{u_i}} : in_{u_i} \mapsto out_{u_i}$$

An example is given as follows:

Example 1

$$\begin{array}{ll} S_1 : A = B + C & u_1 : (+, \{B, C\}, \{A\}, 1) \\ S_2 : E = 2 \times F & u_2 : (\times, \{2, F\}, \{E\}, 2) \\ S_3 : Q = A - E & u_3 : (-, \{A, E\}, \{Q\}, 3) \end{array}$$

where the program is expressed in $U = \{u_1, u_2, u_3\}$, and statements S_1 , S_2 and S_3 are transformed to elements u_1 , u_2 and u_3 respectively.

Definition 3.2 For $\exists u_i \in U$ and $x \in in_{u_i}$, x is said to be **specified** for u_i at a timestep t if and only if:

- (1) x is a constant; or,
- (2) $(\exists u_j \in U, ex_{u_j} < t < ex_{u_i} \wedge x \in out_{u_j}) \wedge (\nexists u_k \in U, t < ex_{u_k} < ex_{u_i} \wedge x \in out_{u_k})$.

Definition 3.3 For $\exists u_i \in U$, if $\forall x \in in_{u_i}$ are specified, then u_i is said to be **ready** for execution, otherwise, u_i is **not ready**.

In Example 1, if we assume variables B , C , and F are assigned certain values before S_1 , then B and C in u_1 , and F in u_2 are specified at the beginning, accordingly, u_1 and u_2 are ready for execution. After performing u_1 and u_2 , A and E become specified and then u_3 becomes ready for execution.

Definition 3.4 For $\{u_i, u_j\} \in U$, a partial order $ex_{u_i} < ex_{u_j}$ is a **successive partial order** if and only if $ex_{u_j} = ex_{u_i} + 1$.

According to Definition 3.1, op_{u_i} is a primitive operation or composed operation abstracted from a number of the operations. Therefore, the representation in U is said to be grain-dependent.

Definition 3.5 Let k_i denote the number of data objects in out_{u_i} , and $u_i \in U$. If $\forall u_i$ for $i = 1, 2, \dots, n$, $k_i = 1$ and op_{u_i} is a primitive operation, then U is said to have a **fine-grain representation**, denoted by U^f .

In this thesis our discussion is mainly focussed on the properties of U^f and its applications to parallel programming. Therefore, we simply just refer to U^f as U in discussion without indication.

Definition 3.6 For $\exists u_i \in U$, there is an **elementary data-operation-associated graph (EDOAG)** of data $x \in out_{u_i}$, denoted by $EDOAG_{u_i}$ which is a directed acyclic graph where the vertex $x \in out_{u_i}$ is called a successor of the data objects listed in in_{u_i} and there is an edge from each of them to x .

Definition 3.7 For $\{u_i, u_j\} \in U$, if $ex_{u_i} < ex_{u_j}$ and $out_{u_i} \cap in_{u_j} \supseteq \{x\}$, and there is no $u_k \in U$ in which $out_{u_k} \supseteq \{x\}$ and $ex_{u_i} < ex_{u_k} < ex_{u_j}$, then $EDOAG_{u_i}$ and $EDOAG_{u_j}$ are merged or involved in a dataflow relation from u_i to u_j .

Definition 3.8 For $\exists u_i \in U$, a **data-operation-associated graph (DOAG)** of data $x \in out_{u_i}$ is a composition of a number of EDOAGs for u_j, u_k, \dots, u_l , which has the following properties:

1. $DOAG_{u_i}$ has a vertex $x \in out_{u_i}$ with outdegree zero;

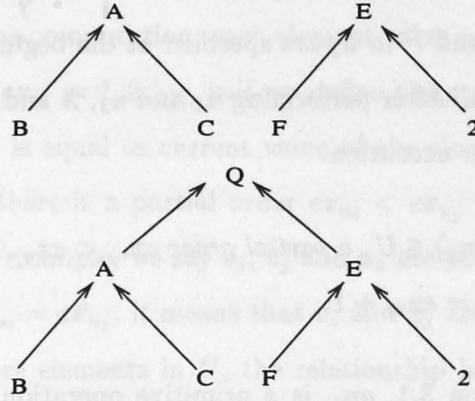


Figure 3.4: CDOAGs of Example 1

2. $EDOAG_{u_j}, EDOAG_{u_k}, \dots, EDOAG_{u_i}$ are involved in dataflow relations in a pair-wise manner.

Definition 3.9 For $\exists u_i \in U$, if there is a $DOAG_{u_i(x)}$ and all vertices with indegree zero in $DOAG_{u_i(x)}$ are specified, then $DOAG_{u_i(x)}$ is **completely specified**, denoted as $CDOAG_{u_i(x)}$.

In terms of Definition 3.9, it is seen that there are three $CDOAG$ s in Example 1, which are illustrated in Fig 3.4, if $EDOAG_{u_1}$ and $EDOAG_{u_2}$ are viewed as two special $CDOAG$ s that contain no other $EDOAG$ except themselves. Thus, one important property of $CDOAG$ can be expressed in Lemma 3.1.

Lemma 3.1 If $EDOAG_{u_j(y)}$ is a subgraph in $CDOAG_{u_i(x)}$, then $CDOAG_{u_j(y)}$ is also a subgraph of $CDOAG_{u_i(x)}$.

It is observed that in any $CDOAG_{u_i}$ there are a number of paths which emanate from those vertices with indegree zero towards the root $x \in out_{u_i}$. The number of edges contained in the longest path in $CDOAG_{u_i}$ is called the *depth* of $CDOAG_{u_i}$. Let $U_{CDOAG_{u_i}}$ denote a sub-tuple-space which contains all elements of $CDOAG_{u_i}$, h_{u_i} denote the depth of $CDOAG_{u_i}$, and $\xi_{CDOAG_{u_i}}$ be the length of **critical path** of computation logic for $CDOAG_{u_i}$, then one of the properties regarding to h_{u_i} and $\xi_{CDOAG_{u_i}}$ can be described as follows:

Lemma 3.2 *If there is $U_{CDOAG_{u_i}} = \{u_i, \dots, u_k\}$, then*

$$h_{u_i} \leq \xi_{CDOAG_{u_i}} = \text{Max}\{ex_{u_i}, \dots, ex_{u_k}\} = ex_{u_i}.$$

The *CDOAG* is a time-dependent concept; at each new logical step certain data objects that are not specified earlier become specified. Hence, the size and depth of a *CDOAG* are reduced gradually during the execution of computation. This feature of *CDOAG* can be flexibly used to consider a *DOAG* $_{u_i}$ as a *CDOAG* by assuming that certain vertices of *DOAG* $_{u_i}$ are specified.

The relationships between any two *CDOAG*s are classified under the following four categories:

- **Contained** ($CDOAG_{u_i(x)} \sqsubset CDOAG_{u_j(y)}$)

If $CDOAG_{u_i(x)}$ is a subgraph of $CDOAG_{u_j(y)}$, then $CDOAG_{u_i(x)}$ is properly contained by $CDOAG_{u_j(y)}$;

- **Overlapping** ($CDOAG_{u_i(x)} \bowtie CDOAG_{u_j(y)}$)

If there is $CDOAG_{u_k(z)}$ which is contained by both $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$, but $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are not contained each other, then $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are overlapping;

- **Completely independent** ($CDOAG_{u_i(x)} \parallel CDOAG_{u_j(y)}$)

Let Σ_{u_i} indicate a set of all named vertices of $CDOAG_{u_i}$. If $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are neither contained by each other nor overlapping and $\Sigma_{u_i} \cap \Sigma_{u_j} = \{\emptyset\}$, then $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are said to be completely independent;

- **Conditionally independent** ($CDOAG_{u_i(x)} \parallel_c CDOAG_{u_j(y)}$)

Let $\Sigma_{in_{u_i}}$ indicate a set of all named vertices with indegree zero of $CDOAG_{u_i}$. If $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are neither contained by each other nor overlapping, but $\Sigma_{u_i} \cap \Sigma_{u_j} - (\Sigma_{in_{u_i}} \cap \Sigma_{in_{u_j}}) \neq \{\emptyset\}$, then $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are said to be conditionally independent due to collision of naming variables. (In other words, there are data objects that are shared by $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ to hold computation result.)

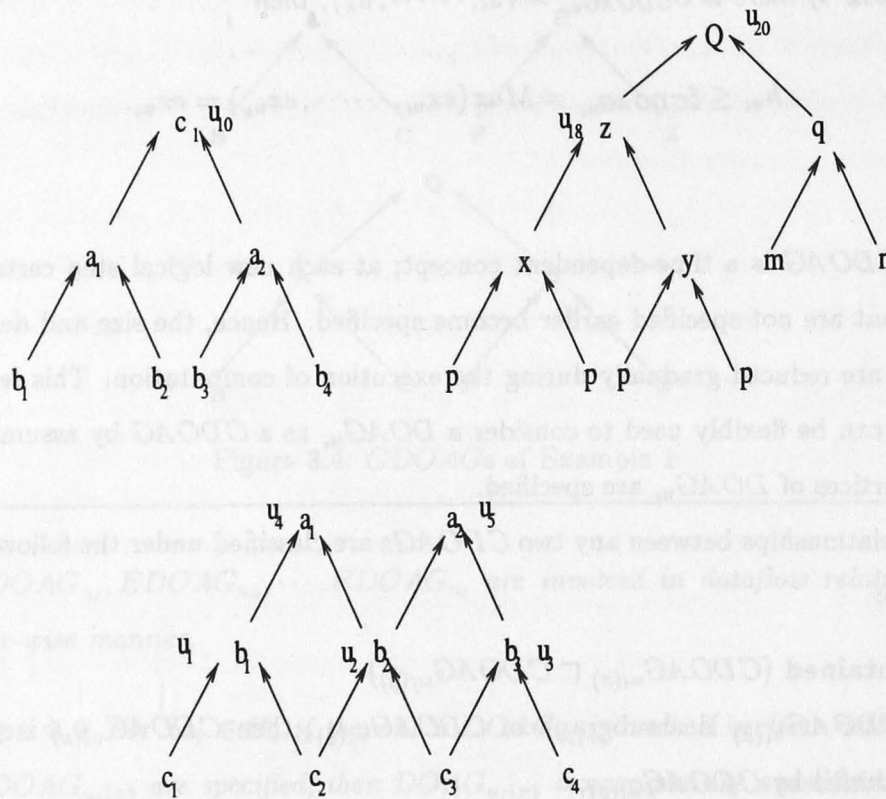


Figure 3.5: Examples of relations between *CDOAGs*

In Fig. 3.5, for instance, $CDOAG_{u_{18}}$ is contained by $CDOAG_{u_{20}}$; $CDOAG_{u_4}$ and $CDOAG_{u_5}$ are overlapping; $CDOAG_{u_4}$ and $CDOAG_{u_{10}}$ are conditionally independent because of variable reuse; and $CDOAG_{u_{10}}$ and $CDOAG_{u_{20}}$ are completely independent.

Lemma 3.3 *If $CDOAG_{u_i(x)}$ and $CDOAG_{u_j(y)}$ are conditionally independent, a shared object z ($z \in \Sigma_{u_i}$ and $z \in \Sigma_{u_j}$) can be removed by substitution of a new object for z in one of them without affecting the correctness of computation.*

From a semantic point of view, a *CDOAG* contains information of a complete computation task that is performed. All vertices with indegree zero of a *CDOAG* are given as input data; the vertex with outdegree zero is the output. The spatial properties of the computation are defined by the set of objects used. The computational logic is explicitly determined by the partial orders of the elements in the *CDOAG*. There are certain dataflows among the elements, which start from the elements that are ready

initially and end by producing the output of the *CDOAG*.

Note that due to the explicit specification of the execution order of elements in U , there is a partial order ($<$) between any two elements u_i and u_j . The partial order here that is expressed is of two types:

(i) Essential: A partial order is essential for a completion of a certain computation task, thus, we say this partial order is **necessary**;

(ii) Non-essential: A partial order introduced may be non-essential for the correct completion of the task, hence, this partial order is **unnecessary** for correct computation.

For example, there are three partial order relationships among the elements in Example 1, that is, $ex_{u_1} < ex_{u_2}$, $ex_{u_1} < ex_{u_3}$ and $ex_{u_2} < ex_{u_3}$. In order to keep the correct result of computation, partial orders $ex_{u_1} < ex_{u_3}$ and $ex_{u_2} < ex_{u_3}$ are necessary, but $ex_{u_1} < ex_{u_2}$ is not necessary because if we let $ex_{u_1} = 2$ and $ex_{u_2} = 1$, the result is same. It means the partial order between two elements that are contained in two completely independent *CDOAGs* is unnecessary.

In Definition 3.7, the formula $ex_{u_i} < ex_{u_j}$ for elements u_i and u_j which are involved in dataflow relation determines relation of execution orders between the elements, which can be expressed as Lemma 3.4.

Lemma 3.4 *If there is an element $u_i \in U$ and two subsets of elements $\{u_k, u_l, \dots, u_p\}$ and $\{u_j, u_h, \dots, u_q\}$ which are directly merged with u_i , and $in_{u_i} = out_{u_k} \cup out_{u_l} \cup \dots \cup out_{u_p}$ and $out_{u_i} \subseteq in_{u_j} \cup in_{u_h} \cup \dots \cup in_{u_q}$, then*

$$ex_{u_i} \geq t_{lower_{ex_{u_i}}} = \text{Max}\{ex_{u_k}, ex_{u_l}, \dots, ex_{u_p}\} + 1 \quad (3.1)$$

and

$$ex_{u_i} \leq t_{upper_{ex_{u_i}}} = \text{Min}\{ex_{u_j}, ex_{u_h}, \dots, ex_{u_q}\} - 1. \quad (3.2)$$

Here $t_{lower_{ex_{u_i}}}$ and $t_{upper_{ex_{u_i}}}$ are the **lower bound** and **upper bound** which indicate respectively the earliest and the latest execution time of ex_{u_i} . Let $\Delta_{ex_{u_i}}$

denote the difference between the lower bound and the upper bound, given by

$$\Delta_{ex_{u_i}} = [t_{upper_{ex_{u_i}}}, t_{lower_{ex_{u_i}}}] \quad (3.3)$$

$$= [Min\{ex_{u_j}, ex_{u_h}, \dots, ex_{u_q}\} - 1, Max\{ex_{u_k}, ex_{u_l}, \dots, ex_{u_p} + 1\}]. \quad (3.4)$$

The period of time covered by $\Delta_{ex_{u_i}}$ is called **legal execution zone** of u_i . It means that without losing correctness of computation u_i can be designed to be performed at any one of logical timesteps within $\Delta_{ex_{u_i}}$. How the execution time of u_i can be legally changed within $\Delta_{ex_{u_i}}$ will be discussed in Chapter 5.

Based on the concept of partial ordering, some important properties of computation can be observed:

- In performing a certain computation task, a set of computation elements of U can be put into a partial order for proper execution. A partial order between any two elements is necessary, if and only if they are related to a data flow.
- A particular execution order for a computation task is designed in a particular solution. By changing certain execution order, one can change one solution into another without losing correctness. Given necessary partial orders $ex_{u_i} < ex_{u_j}$ and $ex_{u_i} < ex_{u_k}$, and an unnecessary partial order $ex_{u_j} < ex_{u_k}$ (or $ex_{u_k} < ex_{u_j}$), under certain conditions, we can change the unnecessary partial order to $ex_{u_k} < ex_{u_j}$ (or $ex_{u_j} < ex_{u_k}$), or let them be executed *in parallel* ($ex_{u_j} = ex_{u_k}$).
- The range for changing an unnecessary partial order is determined by its corresponding legal execution zone $\Delta_{ex_{u_i}}$. If the execution order ex_{u_i} of is equal to the lower bound of $\Delta_{ex_{u_i}}$, u_i is said to be **successively executed**.

3.3 Program Solutions in ABCOM

By Definition 3.1, ABCOM is a representation in which computation is expressed as quadruples. To enable a set of elements to perform a particular computation task, these elements must be designed into a *program solution*.

Definition 3.10 Let $\mathcal{P} = \{u_0, u_1, \dots, u_n\}$ be a subset of U , and EP (execution pointer) be the current value of a logical clock with $EP = 0$ initially. \mathcal{P} is said to be a **solution** if and only if when EP starts to count, \mathcal{P} always meets the following conditions:

1. $\exists u_i, ex_{u_i} = 1$ and u_i is ready for $i = 1, 2, \dots, n$;
2. There is a successive partial order $ex_{u_l} \leq ex_{u_l} \leq \dots \leq ex_{u_k}$ for $1 \leq l, k \leq n$;
3. $\forall u_i \in \mathcal{P}$, u_i must be ready when $ex_{u_i} = EP$ and $\forall u_j \in \mathcal{P}$ with $ex_{u_j} < EP$ should have been performed.

When $ex_{u_i} = EP$, according to Definition 3.10, $u_i \in \mathcal{P}$ is executed. Thus, EP is particularly set to require that a solution should be successively executed as required by the condition 2 from the logical point of view. The condition 2 also implies that it is possible to let more than one elements be executed in parallel.

Definition 3.11 Let $u_i \in \mathcal{P}$, u_i is **currently executable** if and only if u_i is ready and $ex_{u_i} = EP$.

In ABCOM, the solution to a given problem is classified under three categories (according to their execution features):

- **Category 1** (Sequential solution)

Let \mathcal{P} be a solution; if there is no equation in the successive partial order of \mathcal{P} , then \mathcal{P} is a *sequential solution*.

- **Category 2** (Parallel solution with latencies)

Let \mathcal{P} be a solution; if there are equations in the the successive partial order of \mathcal{P} ; but at a given logical step (t) there is $u_i \in \mathcal{P}$ which is ready for execution but not currently executable ($ex_{u_i} > t$), then \mathcal{P} is said to be a parallel solution with latencies. The parallelism achieved by such a solution is *subjective* since the parallelism obtained here is determined by the constraints introduced by the special design of the solution.

- **Category 3** (Parallel solution without latencies)

Let \mathcal{P} be a solution; if there are equations in the successive partial order of \mathcal{P} ,

and at any given step (t) all those elements must be currently executable if they are ready for execution, then \mathcal{P} is said to be a parallel solution without latencies. The parallelism reached here is *objective* because all computation are performed in a dataflow fashion.

Programming experience shows that there are varieties of solutions in both Category 1 and 2 for a given problem. Any solution in Category 1 is executable on a sequential architecture. While a parallel solution in Category 2 can only be performed on a certain architecture that has enough processors to support all data accesses required. However, people have relatively little experience in the solution in the third category since it is difficult and not practical to manually design a program in that manner. Physical implementation of parallel programs is likely in Category 2 rather than in Category 3. A parallel program solution belongs to Category 3 if and only if all computation elements are executed as data-driven dataflow computation [Sha85].

If a given problem has deterministic computation in both data and operations, then there is a unique solution in Category 3. One may question whether two solutions not in Category 3 to the same problem can eventually be optimised to reach a same solution that is of the Category 3. The answer is yes if they have the same data domains and operations. A practical way to check the effectiveness of this optimisation is to compare the difference in performance of two optimised solutions to the same problem. This will be addressed in Chapter 5.

Lemma 3.5 *A program solution \mathcal{P} contains at least one $CDOAG_{u_i(x)}$ for $u_i \in \mathcal{P}$.*

If a solution $\mathcal{P} = \{u_0, u_1, u_2, \dots, u_n\}$ consists of a number of $CDOAG$ s, i.e. $\{CDOAG_{u_i}, \dots, CDOAG_{u_k}\}$, a **critical path** of computation logic for the solution, denoted as $\xi_{\mathcal{P}}$, can be expressed as:

$$\xi_{\mathcal{P}} = \text{Max}\{ex_{u_1}, ex_{u_2}, \dots, ex_{u_n}\}. \quad (3.5)$$

Lemma 3.6 *If a program solution with $\{CDOAG_{u_i}, \dots, CDOAG_{u_k}\}$ is of the third category, then:*

$$h_{u_i} = \xi_{CDOAG_{u_i}} \quad (3.6)$$

for $l = i, \dots, k$ and

$$\xi_P = \text{Max}\{h_{u_i}, \dots, h_{u_k}\} \quad (3.7)$$

3.4 Features of ABCOM

3.4.1 The ABCOM (virtual) machine

Based on discussion of different categories of solutions, we define an ABCOM virtual machine where the three categories of solutions are executable. The special requirement for this machine is to have an ability to accommodate any possible parallelism in computation. This was also considered by D. A. Padua's [PP90] on machine-independent evaluation of parallelising compilers. In the work reported in this thesis, we assume that in an ideal machine:

- there are unlimited number of processors which can exploit an unbounded amount of parallelism;
- there is a logical clock;
- the memory is based on a shared CREW PRAM model such that storage management and allocation of data among processors can be ignored;
- each defined operation consumes one unit of time, no matter what grain sizes they have;
- all other activities — including forking and synchronisation overhead, memory reads and I/O — are free.

Using such a machine, a machine-independent representation is ensured. The assumption of a CREW PRAM memory makes the representation free of memory constraints in a physical architecture.

The communication cost is introduced in a particular implementation based on a selected architecture. The communication issues are not discussed here because ABCOM is an ideal abstract machine for parallel execution of a solution with no special requirements on communication.

The execution of a given solution \mathcal{P} in the ABCOM machine can be explained as the following procedure:

1. Initialise $EP=0$;
2. Let $EP=EP+1$;
3. If $\exists u_i \in \mathcal{P} \wedge ex_{u_i} = EP$ then *terminate*;
4. Perform all u_i with $ex_{u_i} = EP$;
5. Back to 2.

3.4.2 ABCOM and other models — a comparison

The definitions and general properties of ABCOM can show certain features that are not usually presented in a conventional language (or computational model).

- **Usage**

ABCOM has a special representation structure (*tuple*) with three fundamental concepts, *operation*, *data* and *execution time*, and characterise computation in both spatial structure (data domain) and temporal properties. Based on this abstract space, we require: 1) the complex tasks of exploiting parallelism can be carried out independently in systematic methods; 2) relevant techniques involved at different stages of programming can be integrated into a practical framework. Parallel computation inference is expected to be introduced on this framework. In short, ABCOM is an intermediate representation to investigate and reveal parallelism.

- **Combination of three concepts**

The key feature of ABCOM is the combination of *tuple space*, *CDOAG* and *partial ordering*. Each of these play different roles in achieving the goals. Tuple space and partial ordering have been used separately by many researchers. Linda [CG89] programming uses tuple space as a virtual, associative and logically-shared memory. ABCOM uses it as an abstract computation space in which each element is viewed as a discrete computation unit performed at a given logical time point.

Under certain conditions that will be discussed in Chapter 6, a given computation unit can become schedulable such that optimisation can be carried out. The computation unit is grain-dependent. As far as an operation is defined, the unit can be of any size in a range from fine-grain to coarse-grain.

Partial order approach was used in Greif's thesis [Gre75] as an early appearance. The partial ordering and logical clocks are combined in the classical work of Leslie Lamport on a distributed system [Lam78], where the potential of the partial ordering to help one to understand the basic problems of multiprocessing independently of the mechanisms used to solve them has been demonstrated. Petri advocated this view of computation. Winskel's theory of event structures [Win80], [Win84] concerns partial orders on events in Petri net models. Pinter and Wolper consider partial orders as a model of temporal logic [PW84]. Pratt introduces *pomsets* [Pra86] [Pra94] by using partial orders in combination with formal languages and temporal logic. ABCOM uses the partial ordering to link related elements and ensure that computation can be performed correctly. The partial ordering expressed in a program is divided into two classes, that is, *necessary partial order* and *unnecessary partial order*. ABCOM uses this notion to find out where parallelism is inherent in a program solution.

The difference between *CDOAG* and other graphic representation techniques can be observed in many respects. We compare *CDOAG* with the following three types of graphs:

1. The *data dependency graph* (DDG) is widely used in parallelising compiler studies [WB87] [MPC90]. The DDG represents graphically the data dependency at a *statement level* or *statement instance level*. Most studies in the literature are based on DDG at the statement level.
2. The *directed acyclic graph* (DAG) used in [ASU86], [Ell86], [KR90] is also a statement-based graph. In the papers by [CBF91], [AE88], [Ske91], a dataflow model is used to represent a loop, instead of a DAG. In a dataflow model, the nodes are operations; and different outputs will be produced when different data elements enter the input ports although the same operations

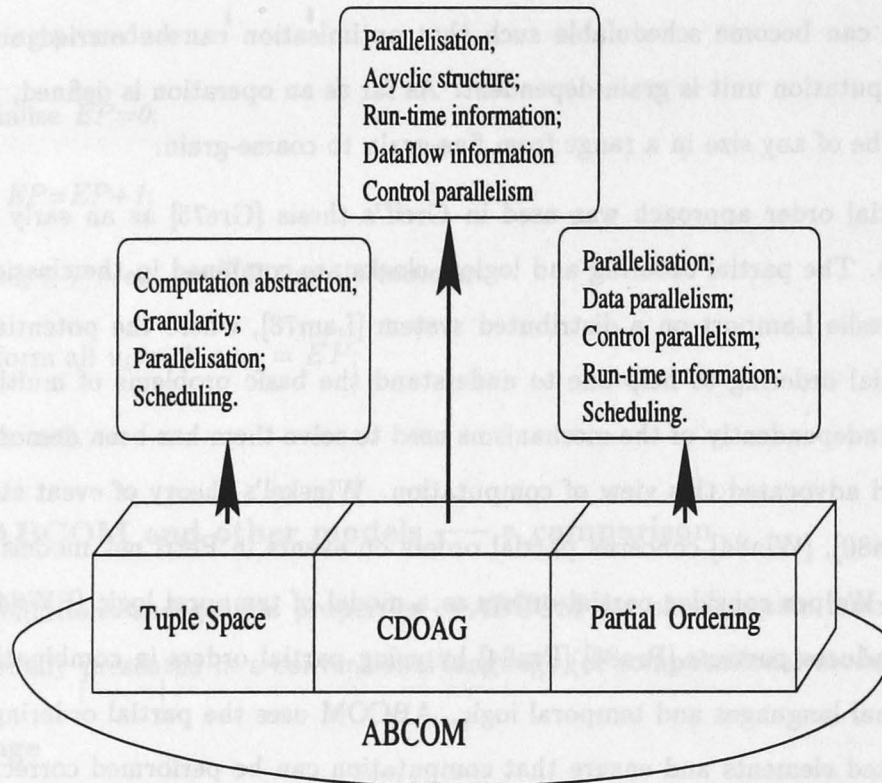


Figure 3.6: ABCOM platform for parallel processing

are performed. Thus, the dataflow model corresponds to a *function-* or a *process-oriented* graph.

3. The unconstrained *dynamic dependence graph* (*uDDG*) used in Tetra [AS93] is similar to *CDOAG*. The *uDDG* abstracts computation patterns from the execution of the program rather than from the context of the program.

A comparison of *CDOAG* with these techniques is given in Table 3.1 in terms of the graph attributes used to express the computation features. *CDOAGs* provide effective means to separate a complete procedure of data generation from irrelevant constraints in a program. The benefits of using *CDOAGs* are illustrated in the examination of applications of ABCOM. The combination of the three concepts mentioned, namely, *tuple-space*, *partial ordering* and *CDOAG*, makes ABCOM satisfy the criteria proposed for a new model (in Chapter 1).

Graph Attribute possessed	<i>CDOAG</i>	DDG	DAG(DFD)	<i>uDDG</i>
Representation level	operation	statement	statement	operation
Node	data	statement	statement (operation)	operation
Cyclic structure	no	yes	no	no
Spatial structure exploitation	yes	no	no	
Visualisation of data generation	yes	no	no	yes
Size of optimisation space provided	large $O(D)^*$	small $O(T)^*$	small $O(T)^*$	
Exploitation of dataflow computation	complete	partial	partial	
Instruction-level information	yes	no	no	no

*Note: 1) D is the size of data domain.

2) T is the size of the text of a loop body.

Table 3.1: Comparison of *CDOAG* with other graph representations used

- **Computation inference support**

Using ABCOM to express solutions, computation inference of parallelism can be carried out based on the concepts, like time, data and operations. Meanwhile, the data-access patterns of a solution can be determined by using input-output relationship among elements such that localising data can be used to detecting data dependences. The parallelism of computing through iterative structures, like loops, can be detected after transforming the cyclic structures into the acyclic structures in ABCOM. Computation inference based on ABCOM is presented in detail in Chapter 5.

The combination of the three concepts forms a unified platform of knowledge representation. A wide range of research interests and development issues in parallel computing can be supported by using different parts of the platform, as shown in Fig. 3.6.

3.5 Summary

Three concepts, *tuple space*, *CDOAGs* and *partial ordering*, are used in ABCOM for exploiting different computation features. The spatial structures and temporal properties of a problem which are related to many different issues in parallel computing, thus, can all be studied using a common basis. With the definitions and properties presented in this chapter, we will show in later chapters which and where parallelism is inherent and to what extent the parallelism can be achieved in a given solution. Also, the applications of ABCOM platform will be described in association with other programming tasks.

Chapter 4

Expressive Power and Transformation

In this chapter, we describe the expressive power of ABCOM and how to transform a source code into ABCOM. To demonstrate the expressive power of ABCOM, we compare it with a Fortran-like language with assignments, branch statements and loops. The transformation is also considered from a sequential structured code of such a language to a fine-grain form in ABCOM.

4.1 Expressive Power

ABCOM differs from conventional languages in the following respects. (i) It has an operational structure (element of the tuple-space) to express any computation. The granularity of representation in ABCOM can express computation at different abstract levels. Various data structures can be used in ABCOM. In a fine-grain representation data objects are mainly the variables and elements of arrays. For a medium or coarse grain representation data objects can be any general data structure e.g., lists and arrays. (ii) The representation form of ABCOM is not suitable for one to apply it manually, but it supports computation analysis, once a solution is converted into ABCOM.

Because of the above characteristics ABCOM is more like an intermediate language of a compiler (in both representation and translation). We do not claim that ABCOM is better than any programming language to express parallelism. What we are interested in is to describe its features to improve parallel programming.

```

 $ABCOM ::= \{u_0, u_1, \dots, u_i, \dots, u_n\}$ 
 $u_i ::= ('op_{u_i}, in_{u_i}, out_{u_i}, ex_{u_i}')$ 
 $op_{u_i} ::= math\_op | rel\_op | u\_op | \odot | con\_op$ 
 $in_{u_i} ::= \{'parameter\_list'\}$ 
 $out_{u_i} ::= \{'parameter\_list'\} | \{id | id'\}$ 
 $ex_{u_i} ::= integer | exp$ 
 $math\_op ::= + | - | \times | / | \dots$ 
 $rel\_op ::= > | < | = | \leq | \geq | \neq | \neg | \dots$ 
 $u\_op ::= < user\_symbol >$ 
 $con\_op ::= if - inten | while$ 
 $parameter\_list ::= parameter | parameter\_list, parameter$ 
 $exp ::= parameter + exp | parameter + integer$ 
 $parameter ::= id | integer$ 
 $i ::= integer$ 

```

Figure 4.1: The Syntax of ABCOM

4.1.1 Syntax of ABCOM

The syntax of ABCOM is shown in Fig. 4.1. Here, *user_op* is an operation defined by users, which can be a new primitive operation or a compound operation that consists of a number of relevant defined operations. Semantics of *user_op* symbols is defined in a *user_op-table*. If *user_op* is a compound operation, then the intensional semantics of the operation are explained by a subset of the tuple space, which is pointed by the indicator in *user_op-table*. The semantics of computation is given by intensional computation logic. Abstracting the operations of a subset of elements into a compound operation enables us to construct a medium-grain or coarse-grain representation. Accordingly, the data objects in *in* or *out* can be the names of data structures.

Our discussion here is focused on the fine-grain tuple space. A compound element is also considered to contain only one data object in *out*. This will make our discussion and techniques suitable as well to these kinds of compound elements. Occasionally, this restriction is removed when a general representation of an element with more than one data objects in *out* is needed.

If a set of related elements that compute certain data objects is defined as a compound element, then the input data for the resulting macro operation of the compound

element includes all those vertices (data objects) whose indegrees are zero in the specially defined *CDOAGs*; the output data is defined as a set of all vertices (data objects) with connections to other vertices which are not contained in these *CDOAGs*; and the execution order is assigned by the timestep at which all input data should have become specified.

The semantic abstraction of a compound element $user_op_{u_i}$ can be explained as a special $user_op$:

$$user_op_{u_i} : in_{u_i} \mapsto out_{u_i}.$$

If all primitive operations are binary in a compound element, a $CDOAG_{u_i}$ can be abstracted as a $user_op$ denoted by \otimes_i in terms of the following grammar:

$$\otimes := L$$

$$L := \Theta | \Theta L | (L, L) | \epsilon$$

$$\Theta := + | - | \times | \{ \text{primitiveoperator} \}.$$

Example 2

We have a subset $U_1 = \{u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9\}$ for computing

$$X = ((A - B) / ((C + D) + (E \times F))) \times ((G - H) + (I/J))$$

where

$$u_1 : (-, \{A, B\}, \{v_1\}, 1) \quad u_2 : (+, \{C, D\}, \{v_2\}, 2)$$

$$u_3 : (\times, \{E, F\}, \{v_3\}, 3) \quad u_4 : (+, \{v_2, v_3\}, \{v_4\}, 4)$$

$$u_5 : (-, \{G, H\}, \{v_5\}, 5) \quad u_6 : (/ , \{I, J\}, \{v_6\}, 6)$$

$$u_7 : (/ , \{v_1, v_4\}, \{v_7\}, 7) \quad u_8 : (+, \{v_5, v_6\}, \{v_8\}, 8)$$

$$u_9 : (\times, \{v_7, v_8\}, \{X\}, 9)$$

The corresponding $CDOAG_{u_9}$ is shown in Fig. 4.2. The compound operation of $CDOAG_{u_9}$ can be expressed by

$$\otimes_1 := \times (/ (-, + (+, \times)), + (-, /)),$$

and accordingly,

$$u_c : (\otimes_1, \{A, B, C, D, E, F, G, H, I, J\}, \{X\}, 1).$$

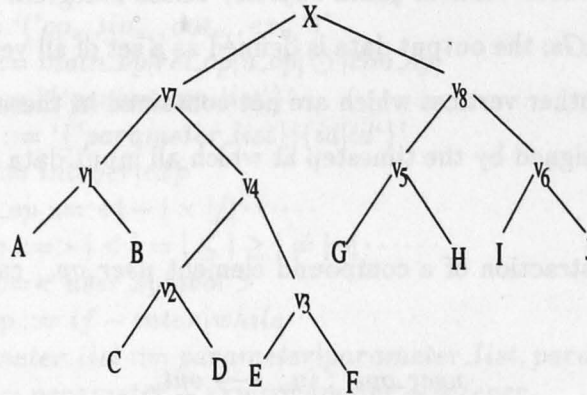


Figure 4.2: A composed element

Here we let $ex_{\otimes_1} = 1$ since all input are specified if this is a independent *CDOAG*. Note this form does not tell the intensional computation logic of \otimes_1 .

For a given subset of elements, the abstraction form of the compound operation may not be unique due to the properties of operations in the elements. The normalisation of the representation of compound operations is discussed in Chapter 7. For technical convenience we assume an identity element with a special operation symbol ' \odot '. Both *in* and *out* of an identity element are presented by an empty set $\{\emptyset\}$. In Definition 3.1, u_0 is such an element. An identity element does no computation but takes a unit of logical time, and is independent from other elements.

Based on the grammar and the definitions in Chapter 3, a computation unit is said to be valid in ABCOM if it does not belong to the following two cases.

- *Case 1* where $op_{u_i} : in_{u_i} \rightarrow out_{u_i}$ cannot be carried out due to inconsistency among given op_{u_i} , in_{u_i} and out_{u_i} . For example, element $(+, \{a, b, c\}, \{d\}, e_1)$ is invalid because '+' is a binary operation, but there are more than two variables in in_{u_i} .
- *Case 2* where for $u_i, u_j \in U$ there are $x \subseteq \{out_{u_i} \cap out_{u_j}\}$ and $ex_{u_i} = ex_{u_j}$. Here variable x is uncertain since concurrent writing occurs, which is not allowed in terms of the assumption of CREW PRAM memory used in ABCOM.

4.1.2 Examples

To illustrate the expressive power of ABCOM, we convert some statements of a Fortran-like language (such as assignments, conditional statements and loops) into the form of ABCOM with the assumption of $EP = 1$.

- Assignments

The conversion of a simple assignment is illustrated in Example 2 where v_1, v_2, \dots, v_8 are temporary working variables to store intermediate results of computation. As a special case, an assignment $x := y$ is converted to $u : (=, \{x\}, \{y\}, k)$. It would be noted that, unlike the *N-ADDRESS CODE*(NADDR)[NF84] and the intermediate codes used by a compiler [ASU86] in which execution semantics of computation is implied, ABCOM requires explicit execution specification for each element.

- Conditional statements

A simple conditional statement is expressed using three related elements.

Example 3 The statement:

$$\text{if } a < b \text{ then } x = y + z \text{ else } x = y - z,$$

is realised by:

$$u_1 : (<_b, \{a, b, 2\}, \{e_1 \mid e_2\}, 1),$$

$$u_2 : (+, \{y, z\}, \{x\}, e_1),$$

$$u_3 : (-, \{y, z\}, \{x\}, e_2).$$

Here u_1 with a special boolean operation $<_b$ can be explained thus: if $a < b$ then $e_1 = 2$, else $e_2 = 2$. It means that the result of performing u_1 is to assign the current execution control to either e_1 or e_2 so that one of u_2 or u_3 can be executed after u_1 , while $e_1 = e_2 = 0$ before u_1 and after performing u_2 or u_3 . This is a simple case of condition statements.

Example 4. A statement

$$\text{if } a < b \text{ then } x = y + z,$$

is realised by

$$u_1 : (<, \{a, b, 2\}, \{e_1 \mid e_2\}, 1),$$

$$u_2 : (+, \{y, z\}, \{x\}, e_1),$$

$$u_3 : (\odot, \{\emptyset\}, \{\emptyset\}, e_2).$$

A conditional branching statement is sometime followed by two groups of successive computations; then one is chosen by test condition. How to express this kind of computation is discussed in the next section.

- **Loops**

A loop consists of a set of well-organised statements to carry out iteration. A task performed in a loop is expressed in ABCOM by interpreting each statement during execution using the methods described above, and generating a *total order* of execution for all elements involved in the loop. That is, a loop is transformed into linear structure by trace generation. This is one of main differences between ABCOM and other intermediate codes. Both **Do--Loop** and **While--Do** can be translated into ABCOM. This will be discussed in the next section.

4.2 Solution Transformation

Transforming a solution from a FORTRAN-like language into ABCOM is similar to compilation, and techniques described in [ASU86], [Ell86] are needed. The transformation described here is different from a traditional compiler.

- The first difference is the target code of transformation. A compiler produces a machine-executable code for a target architecture when a source code is transformed. ABCOM is machine-independent but virtually executable in an abstract computation space (an ideal machine). Therefore, the transformation process related to the run-time environment is greatly simplified.
- The transformation performed here is more like interpretation of program execution by using a discrete form of tuples. It can be thought as an application of trace-driven techniques. Hence, the sequence of computations in a program is preserved in the transformed code. In this sense, the ABCOM compiler performs a combination of parsing, translation and trace generation.

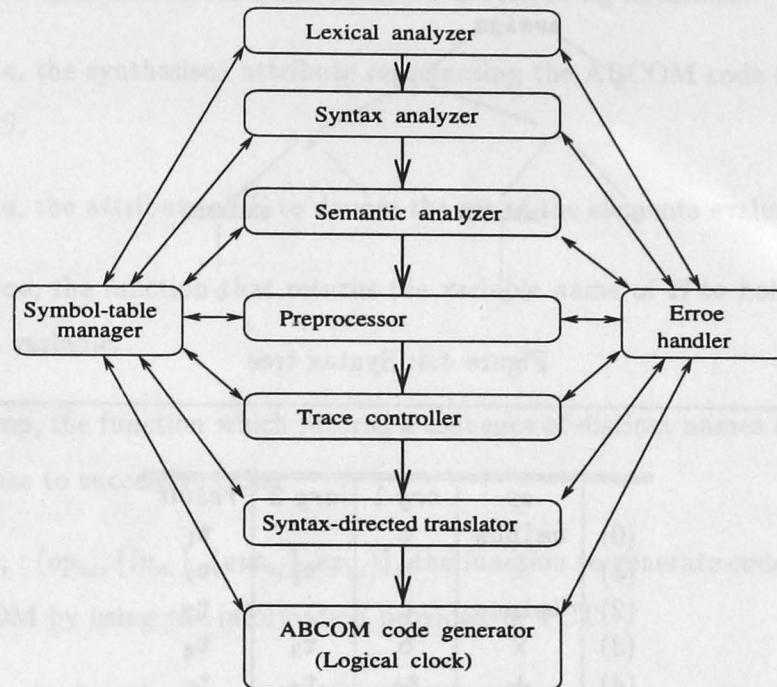


Figure 4.3: Overview of the ABCOM compiler

ABCOM transformation can be divided into several phases as shown in Fig. 4.3. All parsing techniques, like lexical analysis and syntax analysis can use the front end of the standard Fortran-compiler.

Unlike a traditional compiler, an ABCOM compiler has three special components.

(1) A *preprocessor* to perform preparation for transformation, including *branching-merging point analysis* (described in Section 4.2.3); and substitution of function-based reference of the element of an array (discussed in the Section 4.2.1).

(2) A *logical clock* to provide execution specification when each element is generated.

(3) A *trace controller* to keep the current values of all execution-related variables (e.g. loop-control variables) using a *trace-control table* (TCT). The *trace controller* points out where program execution heads for in a source code during the trace generation.

In the last two phases the transformation process uses modified techniques introduced in the translation of the intermediate code (*Quadruples*) in [ASU86]. One of the special features of our translation is to assign explicit execution specifications to each element generated. As an example demonstrating traditional intermediate code gener-

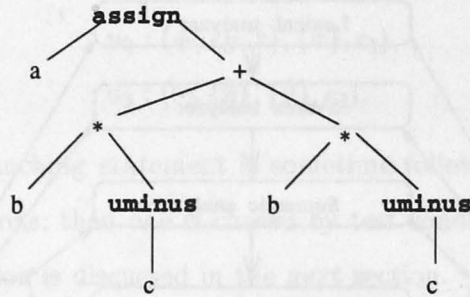


Figure 4.4: Syntax tree

	<i>op</i>	<i>arg 1</i>	<i>arg 2</i>	<i>result</i>
(0)	uminus	c		t ₁
(1)	×	b	t ₁	t ₂
(2)	uminus	c		t ₃
(3)	×	b	t ₃	t ₄
(4)	+	t ₂	t ₄	t ₅
(6)	:=	t ₅		a

Table 4.1: Example of Quadruples

ation, a syntax tree for the assignment statement $a = b \times -c + b \times -c$ and its quadruple representation of three-address statements are illustrated in Fig. 4.4 and Table 4.1.

4.2.1 Principles of translation

Our discussion will focus on ABCOM code generation. ABCOM code requires the logical execution order for elements that is critical for transformation. Our approach is discussed first using the assignment statements in Fig. 4.5.

$$\begin{aligned}
 S &\rightarrow \text{id} := E \\
 E &\rightarrow E_1 + E_2 \\
 E &\rightarrow E_1 E_2 \\
 E &\rightarrow E_1 / E_2 \\
 E &\rightarrow -E_1 \\
 E &\rightarrow (E_1) \\
 E &\rightarrow \text{id}
 \end{aligned}$$

Figure 4.5: The grammar for assignments

In order to generate an ABCOM code, we use following notations.

- **S.code**, the synthesised attribute representing the ABCOM code for the assignment S .
- **E.code**, the attribute of E to denote the set of the elements evaluating E .
- **E.place**, the function that returns the variable name of E to hold the value as E is a variable.
- **newtemp**, the function which returns a sequence of distinct names v_1, v_2, v_3, \dots in response to successive calls.
- **gen** $[u_i : (op_{u_i}, \{in_{u_i}\}, \{out_{u_i}\}, ex_{u_i})]$, the function to generate code (elements) of ABCOM by using the information provided in TCT.
- **T(EP)**, the function to provide current execution order according to the logical clock (EP). Each call of the function leads $EP = EP + 1$.

When ABCOM code is generated, temporary variables are created for holding intermediate results. The syntax-directed rules in Fig. 4.6 generate ABCOM code for assignment statements. For the moment, we create a new name every time a temporary is needed. The techniques for reusing temporaries in ABCOM transformation are the same as those described in [ASU86].

The function **E.place** can return three different kinds of variable names in terms of different methods used for reference.

- (1) If the variable is a singleton data object, the name of the object is returned.
- (2) If the variable is the element of an array and referenced with an index controlled by the iteration control variable with no function, a particular element with a fixed index determined by current value of the iteration control variable is returned. An example is presented in Section 4.2.4.
- (3) If an assignment contains a variable that is an element of an array of which the index is referenced with a function. For this kind of statement, a substitution is introduced by the preprocessor of the compiler if the variable is involved in an operation with other data variables. The basic idea is to replace these elements of an array with

Production	Semantic Rules
$S \rightarrow \text{id} := E$	$S.\text{code} := E.\text{code} \parallel \text{gen}[u_i: (=, \{E.\text{place}\}, \{\text{id}\}, T(\text{EP}))]$
$E \rightarrow E_1 + E_2$	$E.\text{place} := \text{newtemp}$ $E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel$ $\text{gen}[u_i: (+, \{E_1.\text{place}, E_2.\text{place}\}, \{E.\text{place}\}, T(\text{EP}))]$
$E \rightarrow E_1 \times E_2$	$E.\text{place} := \text{newtemp}$ $E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel$ $\text{gen}[u_i: (\times, \{E_1.\text{place}, E_2.\text{place}\}, \{E.\text{place}\}, T(\text{EP}))]$
$E \rightarrow E_1 / E_2$	$E.\text{place} := \text{newtemp}$ $E.\text{code} := E_1.\text{code} \parallel E_2.\text{code} \parallel$ $\text{gen}[u_i: (/ , \{E_1.\text{place}, E_2.\text{place}\}, \{E.\text{place}\}, T(\text{EP}))]$
$E \rightarrow -E_1$	$E.\text{place} := \text{newtemp}$ $E.\text{code} := E_1.\text{code} \parallel$ $\text{gen}[u_i: (\text{'uminus'}, \{E_1.\text{place}, E_2.\text{place}\}, \{E.\text{place}\}, T(\text{EP}))]$
$E \rightarrow (E_1)$	$E.\text{place} := E_1.\text{place}$ $E.\text{code} := E_1.\text{code}$
$E \rightarrow \text{id}$	$E.\text{place} := \text{id.place}$ $E.\text{code} := \text{' '}$

Figure 4.6: Syntax-directed semantic rules to produce ABCOM code for assignments

$$S \rightarrow \text{if } E \text{ then } S_1$$

$$S \rightarrow \text{if } E \text{ then } S_1 \text{ else } S_2$$

Figure 4.7: The grammar for conditional statements

a generic element with a subscript x . Thus, for a functionally indexed variable the function $E.\text{place}$ returns the name of an array with with a subscript x .

To translate a conditional statement with the grammar given in Fig. 4.7, we need to introduce more notation and functions. We assume that, for each branching point, its corresponding merging point is found before translation. The technique to find a merging point is described in next subsection.

- **newlabel**: the function to produce a sequence of labels, e_1, e_2, \dots , as unknown execution orders (called *conditional execution label variables* or *CEL-variables*) for elements generated from transforming conditional statements. The CEL-variables are assigned certain values when the condition in the corresponding branching statement is tested during execution.
- **relop**: a generic relational operation which is replaced by the relational operation in a condition statement as it is translated. The boolean expression E in the grammar has the form of $\text{id}_1 \text{ relop id}_2$.
- in_S : a set of input data which are used for performing a branch flow of a conditional statement.
- out_S : a set of output data produced by a branch flow of a conditional statement.
- $\text{if-inten}(e_i.\text{place})$: a special compound operation performing the operations of statements between the branching point and its associated merging point. $e_i.\text{place}$ points to where the intensional computation is performed.
- $\text{Branch}(\{\text{branch flow list}\}, \{\text{CEL-variable list}\})$: the function to translate branch flows of a conditional statement with the return of in_S and out_S of

Production: $S \rightarrow \text{if } E \text{ then } S1$ Semantic Rules: $S.code := e1.place := \text{newlabel} \parallel e2.place := \text{newlabel} \parallel$ $\text{gen}[u_i : (\text{relop}, \{id_1, id_2, EP + 1\}, \{e1.place e2.place\}, T'(EP))] \parallel$ $\text{Branch}(\{S1\}, \{e1, e2\}) \parallel$ $\text{gen}[u_i : (\text{if-inten}(e1.place), \{in_{S1}\}, \{out_{S1}\}, e1.place)] \parallel$ $\text{gen}[u_i : (\emptyset, \{\emptyset\}, \{\emptyset\}, e2.place)]$
Production: $S \rightarrow \text{if } E \text{ then } S1 \text{ else } S2$ Semantic Rules: $S.code := e1.place := \text{newlabel} \parallel e2.place := \text{newlabel} \parallel$ $\text{gen}[u_i : (\text{relop}, \{id_1, id_2, EP + 1\}, \{e1.place e2.place\}, T'(EP))] \parallel$ $\text{Branch}(\{S1, S2\}, \{e1, e2\}) \parallel$ $\text{gen}[u_i : (\text{if-inten}(e1.place), \{in_{S1}\}, \{out_{S1}\}, e1.place)] \parallel$ $\text{gen}[u_i : (\text{if-inten}(e2.place), \{in_{S2}\}, \{out_{S2}\}, e2.place)]$

Figure 4.8: The syntax-directed semantic rules to translate conditional statements

each branch flow between a pair of branching and merging points. The implementation of this function is described in the Section 4.2.3.

- $T'(EP)$: the same function as $T(EP)$ except here $EP = EP + 2$ after each call.

As a conditional statement with two branch flows is translated, three relevant elements are generated. One of them is to test the conditions, the other two are called the *head* of branch flows. The corresponding syntax-directed rules are illustrated in Fig. 4.8.

4.2.2 Trace-driven code generation

Trace-driven techniques are widely used in performance analysis [Lar90], [Wet al94], timing simulation, interactive debugging [MC91] [MPK91], and programming visualisation [Hea91] [KN91]. In these applications, trace generation facility fulfils trace event generation with no modification on computation of a source code. Trace generation treats a computation unit as a trace event, and records the elements of ABCOM. The timing record is created as logical timesteps of execution order that is controlled by *logical clock* (EP) (to indicate the logical step at which the event happens).

The trace generation converts a given source code into a stream of elements that correspond to the execution of a program. Transforming a source code corresponds to interpreting the computation performed at each logical step.

The transformation of expression-based assignment statements was described in the last subsection. When a complete source code is transformed, we combine this approach with trace generation and the parsing techniques [ASU86], such as *shift-reduce parsing* and *operator-precedence parsing*. In the transformed code of ABCOM, hence, the original execution order of operations is retained and specified explicitly. Such a procedure of the trace-driven transformation can be explained in the algorithm illustrated in Fig. 4.9. The operation-precedence relations shown in Table 4.10 are described in [ASU86].

4.2.3 Branching statements transformation

In programming languages, a branching statement contains an *expression* to compute a predicate that alters the flow of control. As a result, a branching statement determines whether an operation will be executed or not depending upon the test result. The basic idea of representing such a structure in ABCOM is given in the Section 4.1.2. We also combine this idea with trace generation techniques to transform conditional statements in a source code. Consider a general example in a basic block shown in Fig. 4.11. Here the statement S_1 is a branching statement (called *branching point*) and the statement S_n is called a *merging point* of the branching statement.

It is assumed that the *branching* and *merging* points appear in a pair-wise manner in a structured source code. To transform such a structure, the preprocessor has to do *control flow analysis* before transformation. We need to know which statement is the *merging point* of the corresponding *branching point* previously executed. After a branching point, the statements in each branch flow will all depend on the condition of this branch statement. If a merging point is encountered (that is, every flow branching from the same branching statement finally comes into this statement), then the statements following it will no longer depend on the same condition as the flow's. Instead, these statements will now depend on the condition on which the branching statement depends.

Input : An input string w of an assignment, logical clock EP and a table of operation precedence relations shown as Fig. 4.10.

Output : if w is well formed, a sequence of ABCOM code; otherwise, an error indication.

Initial : A stack contains \$ and an input buffer containing the string w .

Algorithm1

- (1) set ip to point to the first symbol of w ;
- (2) **repeat forever**
- (3) **if** \$ is on the top of the stack and ip point to \$ **then**
- (4) **return**
- else begin**
- (5) let a be the topmost terminal symbol on the stack
and let b be the symbol pointed to by ip ;
- (6) **if** $a < \cdot b$ or $a \cdot = b$ **then begin**
- (7) push b onto the stack;
- (8) advance ip to the next input symbol;
- end;**
- (9) **else if** $a \cdot > b$ **then** /* translation and code generation*/
- (10) **repeat**
- (11) pop the stack
- (12) and apply the syntax-directed translation rules
to generate ABCOM code with the current EP ,
and let the symbol in *out* of the new element be
in the position in the string pointed to by ip ;
- (13) **until** the top stack terminal is related by
 $< \cdot$ to the terminal most recently popped
- (14) **else error()**
- end**
- end**

Figure 4.9: Algorithm for assignment transformation

	+	-	×	/	=	id	()	\$
+	·>	·>	<·	<·	·>	<·	<·	·>	·>
-	·>	·>	<·	<·	·>	<·	<·	·>	·>
×	·>	·>	·>	·>	·>	<·	<·	·>	·>
/	·>	·>	·>	·>	·>	<·	<·	·>	·>
=	<·	<·	<·	<·		<·	<·		
id	·>	·>	·>	·>	·>			·>	·>
(<·	<·	<·	<·	<·	<·	<·	=	
)	·>	·>	·>	·>				·>	·>
\$	<·	<·	<·	<·	<·	<·	<·		

Figure 4.10: The table of operation precedence relations

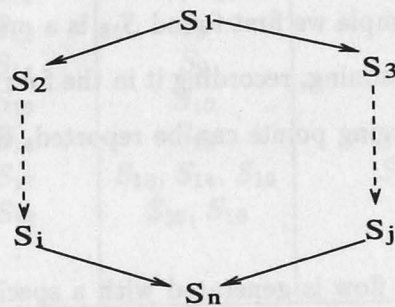


Figure 4.11: A general case of the branching and merging points

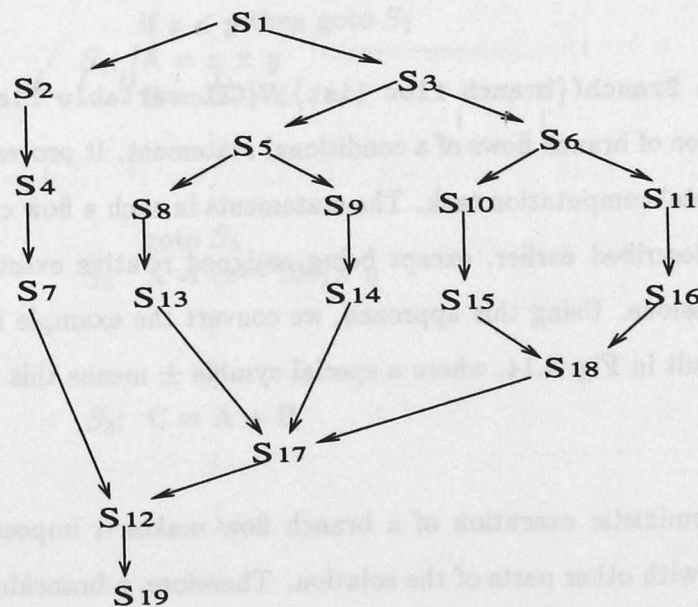


Figure 4.12: A program flow chart including conditional statements

In the control flow analysis [Che85], to identify such a point in a source code, we record how many flows from a *branching point* are flowing through a statement, and how many flows are required to make this statement a *merging point* for the *branching point*. The relation between a branching point S_k and its merging point S_l is denoted as $S_k \leftrightarrow S_l$. The idea can be implemented by using a *merging-point processing table* shown in Table 4.2. Using this table, we show how those merging points in Fig. 4.12 are found. All statements encountered are listed in the order of execution in the second column. For each of these statements the corresponding logically successive statements are recorded in the third column. A merging point can be found by using backward reasoning when there are two or more successive statements that follow the same statement. In this example we first found S_{18} is a merging point of S_6 by taking S_{15} and S_{16} for backward reasoning, recording it in the fourth column. Similarly, three other pairs of branching-merging points can be reported, that is, $S_5 \leftrightarrow S_{17}$, $S_3 \leftrightarrow S_{17}$ and $S_1 \leftrightarrow S_{12}$.

The head of a branching flow is generated with a special operation $\text{if-inten}(e_i)$. Here e_i is associated with a set of elements that correspond to the branching flow. These elements are assigned the expressions with symbols of \pm as the relative execution orders to e_i .

The function $\text{Branch}(\{\text{branch flow list}\}, \{\text{CEL-variable list}\})$ carries out the transformation of branch flows of a conditional statement. It processes each branch flow as a 'complete' computation task. The statements in such a flow can be converted in the method described earlier, except being assigned relative execution orders expressed in expressions. Using this approach, we convert the example in Fig. 4.13 and illustrate the result in Fig. 4.14, where a special symbol \pm means this value is relative to e_1 (or e_2).

The nondeterministic execution of a branch flow makes it impossible to achieve data parallelism with other parts of the solution. Therefore, a branching flow could be treated as 'complete' computation task, and its computation features can be investigated independently.

	Statement label	followed by	Corresponding branching points
	S_1		
	S_2	S_1	
	S_3	S_1	
	S_4	S_2	
	S_5	S_3	
	S_6	S_3	
	S_7	S_4	
	S_8	S_5	
	S_9	S_5	
	S_{10}	S_6	
	S_{11}	S_6	
	S_{12}	S_7, S_{17}	S_1
	S_{13}	S_8	
	S_{14}	S_9	
	S_{15}	S_{10}	
	S_{16}	S_{11}	
	S_{17}	S_{13}, S_{14}, S_{18}	S_5, S_3
→	S_{18}	S_{15}, S_{16}	S_6
	⋮		
	⋮		

Table 4.2: A merging-point processing table

```

    if  $x < y$  then goto  $S_2$ 
 $S_1$ :  $A = x \times y$ 
       $B = D - W$ 
      ⋮
      ⋮
      goto  $S_3$ 
 $S_2$   $A = (x - 100) \times y$ 
      ⋮
      ⋮
 $S_3$ :  $C = A + B$ 
      ⋮
      ⋮
  
```

Figure 4.13: A general example of the conditional statements

4.2.4 Loop transformation

In each iteration the statements of a loop body are executed as a basic block with certain current iteration variables. To transform a loop, we carry out trace generation along with the execution of the loop. As a result, its cyclic structures are transformed into linear structure.

In [Ell86] Ellis described how the Bulldog compiler unrolls the bodies of inner loops immediately after parsing the source code into intermediate code. The *loop unwinding transformation* using the combining DAG technique was introduced by Kramer [KGS94]. Unlike these approaches, our trace generation produces a linear structure for a whole iteration space. For a nested loop an inner loop structure is unrolled completely in each iteration of its outer loop. The semantics (data flow information) of a source code are preserved by the generation (that is, what is created in the tuple space is what is performed in the program). Such a transformation is performed by the algorithm in Fig. 4.15.

Example 5. For the following nested loops

```

for i = 1 to n do
  for j = 1 to n do
    a(i, j) = (a(i, j - 1) + a(i - 1, j))/2
  enddo
enddo

```

the first 21 generated elements of ABCOM are presented in Fig. 4.17 where $n = 10$.

Example 6. There is a nested loop

```

for i = 1 to N do
  s = 0
  for j = 1 to i - 1 do
    s = s + aij × bj
    bi = bi - s
  enddo
enddo

```

Input: a Do—loop statement, current *EP* and a loop-control stack *LPC*.

Output: ABCOM code.

Algorithm 2

1. Push current loop-control information onto the *LPC*;
2. **If** the current loop has been completed, **then** pop *LPC* and **exit**;
3. **If** the current statement is a Do---loop,
 then recursively call *Algorithm 2*;
4. Translate the current statement (an assignment
 or a conditional statement) into ABCOM with current *EP*;
5. **If** there is next statement, **then** let it be
 the current statement and **back to 3**;
6. Update the loop-control variable, make the first statement
 in the current loop be the current statement, **back to 2**.

Figure 4.15: Algorithm for Do--loop

for $n = 10$ its ABCOM code is shown in Fig. 4.16.

Example 7. Consider a sequential code for Gaussian Elimination (without pivoting).

```

for  $k = 1$  to  $n$ 
  for  $i = k + 1$  to  $n$ 
     $a(i, k) = a(i, k) / a(k, k)$ 
    for  $j = k + 1$  to  $n$ 
       $a(i, j) = a(i, j) - a(k, j) \times a(i, k)$ 
    enddo
  enddo
enddo

```

Let $n = 6$, a transformed code of this solution is shown in Appendix A.1.

Algorithm 2 in Fig. 4.15 demonstrates the principle of how the source code is directly transformed to ABCOM. The transformation rules described in 4.2.1 and 4.2.3 are used for generating each element of ABCOM from the source code. In fact, for a good performance, there is an alternative resolution for the transformation in which the source code is first transformed into an intermediate code (for instance, *Quadruples*);

$u_1 : (+, \{a_{10}, a_{01}\}, \{v_1\}, 1)$	$u_{21} : (+, \{a_{20}, a_{11}\}, \{v_{11}\}, 21) \quad \vdots$
$u_2 : (, \{v_1, 2\}, \{a_{11}\}, 2)$	$u_{22} : (, \{v_{11}, 2\}, \{a_{21}\}, 22) \quad \vdots$
$u_3 : (+, \{a_{11}, a_{02}\}, \{v_2\}, 3)$	$u_3 : (+, \{a_{21}, a_{12}\}, \{v_{12}\}, 23) \quad \vdots$
$u_4 : (, \{v_2, 2\}, \{a_{12}\}, 4)$	$u_{24} : (, \{v_{12}, 2\}, \{a_{22}\}, 24) \quad \vdots$
$u_5 : (+, \{a_{12}, a_{03}\}, \{v_3\}, 5)$	$u_{25} : (+, \{a_{22}, a_{13}\}, \{v_{13}\}, 25) \quad \vdots$
$u_6 : (, \{v_3, 2\}, \{a_{13}\}, 6)$	$u_{26} : (, \{v_{13}, 2\}, \{a_{23}\}, 26) \quad \vdots$
$u_7 : (+, \{a_{13}, a_{04}\}, \{v_4\}, 7)$	$u_{27} : (+, \{a_{23}, a_{14}\}, \{v_{14}\}, 27) \quad \vdots$
$u_8 : (, \{v_4, 2\}, \{a_{14}\}, 8)$	$u_{28} : (, \{v_{14}, 2\}, \{a_{24}\}, 28) \quad \vdots$
$u_9 : (+, \{a_{14}, a_{05}\}, \{v_5\}, 9)$	$u_{29} : (+, \{a_{24}, a_{15}\}, \{v_{15}\}, 29) \quad \vdots$
$u_{10} : (, \{v_5, 2\}, \{a_{15}\}, 10)$	$u_{30} : (, \{v_{15}, 2\}, \{a_{25}\}, 30) \quad \vdots$
$u_{11} : (+, \{a_{15}, a_{06}\}, \{v_6\}, 11)$	$u_{31} : (+, \{a_{25}, a_{16}\}, \{v_{16}\}, 31) \quad \vdots$
$u_{12} : (, \{v_6, 2\}, \{a_{16}\}, 12)$	$u_{32} : (, \{v_{16}, 2\}, \{a_{26}\}, 32) \quad \vdots$
\vdots	\vdots
\vdots	\vdots
\vdots	\vdots
\vdots	\vdots

Figure 4.16: The transformed code of Example 5

$u_1 : (=, \{0\}, \{s\}, 1)$	$u_{12} : (=, \{0\}, \{s\}, 12)$
$u_2 : (\times, \{a_{21}, b_1\}, \{v_1\}, 2)$	$u_{13} : ((\times, \{a_{41}, b_1\}, \{v_4\}, 13)$
$u_3 : (+, \{s, v_1\}, \{s\}, 3)$	$u_{14} : (+, \{s, v_4\}, \{s\}, 14)$
$u_4 : (-, \{b_2, s\}, \{b_2\}, 4)$	$u_{15} : (-, \{b_4, s\}, \{b_4\}, 15)$
$u_5 : (=, \{0\}, \{s\}, 5)$	$u_{16} : (\times, \{a_{42}, b_2\}, \{v_5\}, 16)$
$u_6 : (\times, \{a_{31}, b_1\}, \{v_2\}, 6)$	$u_{17} : (+, \{s, v_5\}, \{s\}, 17)$
$u_7 : (+, \{s, v_2\}, \{s\}, 7)$	$u_{18} : (-, \{b_4, s\}, \{b_4\}, 18)$
$u_8 : (-, \{b_3, s\}, \{b_3\}, 8)$	$u_{19} : (\times, \{a_{43}, b_3\}, \{v_6\}, 19)$
$u_9 : (\times, \{a_{32}, b_2\}, \{v_3\}, 9)$	$u_{20} : (+, \{s, v_6\}, \{s\}, 20)$
$u_{10} : (+, \{s, v_3\}, \{s\}, 10)$	$u_{21} : (-, \{b_4, s\}, \{b_4\}, 21)$
$u_{11} : (-, \{b_3, s\}, \{b_3\}, 11)$	\vdots
	\vdots

Figure 4.17: The transformed code of Example 6

the trace generation is then carried out such that the transformation from source code to the intermediate code does not need to be repeated for each element of ABCOM.

4.2.5 While--Do transformation

The **While--Do** loop statement has an indefinite number of iterations. It is observed that the **While--Do** structure can be divided into two groups based on the control mechanism of iteration and the features of processing data within the loop.

In the first group, the same operations are executed based on the same set of data objects in all iterations in which the objects may until a particular condition is satisfied by the computation result of the last iteration. We call this loop *value-control* or *fixed-data-domain* iteration.

The second group is called *size-control* or *variable-data-(sub)domain* (for each iteration) loop since, for a given data domain for each call of the loop, the operations in the loop body are executed to process different subsets of the domain in each iteration.

Precisely, the distinction between these two groups is whether the same data objects are repeatedly processed in each iteration of loop execution. Different methods will be used to transform them respectively.

In the case of a value-control **While--Do** loop, the iteration stops if the condition is satisfied. Our method to transform such a structure is to treat it as the combination of a loop and a special condition statement. The (head) statement of **While--Do** is transformed in a similar approach to converting a branching statement, which generates three relevant elements. To transform the "satisfied branch flow", that is the body of the loop, we introduce another special three elements at the end of the execution of the "flow". The execution condition is thus checked at the end of each iteration, to decide whether another iteration is needed. As an example, Fig. 4.18 is transformed into a code in Fig. 4.19.

The size-control iteration performs iterations with a special value given to define the size the iteration space at each call. The termination condition here is not determined by the computation result of each iteration. To transform such a structure, theoretically, we need to generate an ABCOM code for the whole iteration space. This structure is therefore similar to a **Do--loop** statement except that its loop bound is

```

:
:
While  $\delta < 0.0005$  do
   $Z = (y - x) \times D$ 
:
:
 $\delta = Z/1000$ 
enddo
:
:

```

Figure 4.18: A accuracy-control iteration of While-----Do

```

:
:
:
 $u_l: (<_b, \{\delta, 0.0005, k+1\}, \{e_1 | e_2\}, k)$ 
 $u_{l+1}: (\text{While}, \{\Omega\}, \{\Omega\}, e_1)$ 
 $u_{l+2}: (\text{While}, \{\Omega\}, \{\Omega\}, e_2)$ 
:
:
:
 $u_i: (-, \{y, x\}, \{Z\}, e_1 \pm 1)$ 
 $u_{i+1}: (\times, \{v_1, D\}, \{Z\}, e_1 \pm 2)$ 
:
:
:
 $u_{i+q}: (/ , \{Z, 1000\}, \{\delta\}, e_1 \pm m)$ 
 $u_j: (<_b, \{\delta, 0.0005, e_1 \pm (m+2)\}, \{e_1 | e_2\}, e_1 \pm (m+1))$ 
 $u_{j+1}: (\odot, \{\emptyset\}, \{\emptyset\}, e_1)$ 
 $u_{j+2}: (\odot, \{\emptyset\}, \{\emptyset\}, e_2)$ 

```

Figure 4.19: The transformed result of While--do

defined just before the statement is called rather than defined in programming. Hence, we can transform this statement by using Do--loop transformation techniques if a suitable size-control value that reflects the general characteristics of computation can be provided. Or, at least, a *lower bound* and a *upper bound* of the size could be used to help reveal general features about inherent parallel properties in the loop.

The size of a tuple space transformed from a source code containing loops is mainly determined by the iteration space. The size of a tuple space may become unmanageable when a computationally expensive program (with a large number of iterations) is transformed. To handle such a huge amount of data is not economical and sometimes impossible. But the relation between a tuple-space size and parallelism exploitation discussed in Chapter 7 shows that there are certain strategies and methods to help us to use a reasonable size of a given problem to investigate of parallelism characteristics of a general situation. The main idea of these strategies is based on:

1. Superblock-based parallelism revelation

The most computationally expensive part in a program are various loops, called *superblocks*. The superblock-based strategy is used to cope with individual loops as a number of subproblems. These subproblems can be investigated independently in some degree. The general features of the problem can then be obtained by synthesizing the results of investigation on these superblocks.

2. Computation pattern abstraction

A real world problem can usually be abstracted into certain computation patterns in the form of iteration in programming. The same problem may be abstracted into different patterns with different computation features. After transforming a source code into ABCOM and analysing it, thus, we expect to re-abstract it into new patterns with better performance.

3. Size-based parallelism speculation

Parallelism analysis is started with a reasonable size of a problem. Initially ABCOM uses a suitable size of tuple-space to reveal and speculate parallelism (described in Chapter 7).

4.3 Summary

In ABCOM an executable source code is transformed into another representation that is executable on an ideal machine rather than any physical architecture. In this representation, computation tasks designed in the source code are consistently rewritten by using the tuples of ABCOM. The execution semantics implied in the original code become explicit specification in the transformation under the assumption that each element of ABCOM takes a unit of logical time. In other words, the notation of a source code with implicit execution specification is represented in a set of units where the execution order is linear. This change of representation is the first step in our effort towards the goal of revealing parallelism of a problem.

As assumed in the beginning of this chapter, for the purpose of demonstration, we use a sequential source code. After describing the characteristics of the transformation, however, we see this assumption is not restrictive to generate ABCOM code from other forms of solution representation. We believe that any form of representation, with certain execution features specified implicitly or explicitly, can be considered for this transformation by adequately modifying trace-generation strategies and developing relevant syntax-directed translation rules. The choice of using a sequential source code for transformation does have the following advantages: 1) clear execution semantics that provides convenience for the trace generation; 2) free of communications that are usually required when mapping a problem into a specific architecture.; and 3) easy understanding of transformation techniques since they have certain similarity with conventional compilers.

...in which the original code is transformed into a form that is more suitable for the target machine. This process is often referred to as code generation or code transformation.

The first step in the transformation process is to analyze the source code. This involves identifying the basic blocks of the program and the control flow between them. The next step is to generate the target code, which may involve inserting instructions to handle the differences between the source and target architectures.

One of the key challenges in code transformation is maintaining the semantics of the original code. This means that the transformed code must execute the same operations in the same order as the original code. To ensure this, the transformation process must be carefully designed and tested.

Another important aspect of code transformation is optimization. This involves making changes to the code that improve its performance without changing its semantics. Common optimizations include basic block merging, constant propagation, and dead code elimination.

The final step in the transformation process is to verify the transformed code. This involves checking that the code is syntactically correct and that it executes the same operations as the original code. This is often done using a compiler driver that runs the transformed code on a test suite.

Chapter 5

Parallel Computational Inference

The question as to whether parallel programming can be carried out systematically can be rephrased into two associated questions: whether *parallelism revelation* and whether *static scheduling* can be achieved using systematic methods. Answers to these two related questions require not only a well-organised domain representation as a basis, but also a set of inference rules to analyse, abstract, reason and modify the solution. In this chapter, we first discuss the requirements for a representation form to be used for computation inference; then show that ABCOM serves as such a representation. Based on it we introduce a set of inference techniques to conduct parallelism analysis. This representation is further designed into the conceptual schema of a special programming database that it is used as an operational platform of representation of parallel programming. This permits the inference using relational algebra-like rules.

5.1 Domain Representation Issues

Programming languages and computation models have been developed for specifying application domain knowledge using formal rules. A good language is certainly important for parallel computation. The traditional programming philosophy is not suitable for parallel programming due to the issues raised in Chapter 2, since most languages passively implement subjective understanding of a problem from a programmer, with no function of revealing parallelism. Therefore, it is necessary to identify a set of the criteria required for a representation to support parallelism revelation.

5.1.1 Requirements for conducting parallelism analysis

A given problem solution is completely specified using specification languages, functional languages, high-level programming languages, an intermediate language or a target code. One of the critical issues involved in the investigation of parallelism in a problem is what kind of domain representation is needed for computation inference and analysis. The requirements for a domain representation to be a basis for parallelism analysis are as below:

- **Expressiveness**

The expressiveness of the domain representation should contain computational features of both data and control flows. The computation of a problem can be expressed in either machine dependent or independent manner. Of course, a machine-independent representation will simplify the problem statement without involving the details of implementation on a particular architecture.

- **Granularity**

The granularity of representation plays an important role in parallelism, e.g. to investigate data parallelism a fine-grained representation is necessary; but control parallelism requires investigation on a medium-grained or coarse-grained level.

- **Visibility**

The parallelism can be revealed if the independence among computations is made visible in the representation. The independence among computations can be in different forms and at different levels.

- **Consistency**

If the representation is more complex in its structures, the investigation of parallelism will be more difficult. As a result, a simple and consistent representation is desirable. Also the consistency of representations for dataflow, control flow and independence will be advantageous for abstraction and optimisation if it is available at different granular levels.

- **Temporal aspects**

The temporal aspect of computation is critical to parallel programs, and can

be specified completely or incompletely in an implicit or explicit manner in the representation forms. To fully investigate parallelism of a given problem (especially for inference) a complete and explicit specification of the temporality of computation will be needed.

- **Reconstructivity**

Parallelising compilers optimise a source code and transform it into a target code. In this method, usually, two different solutions with different performances are compiled when two source codes to the same problem are given. Note that performance properties from a source code cannot be eliminated by optimising computation. The reconstructivity of solution representation differs from compilability of languages. Reconstructing a solution means the reabstraction of computation from programming point of view. A reconstructed solution can be expressed in a different representation form or in the original form. The reconstruction is implemented using certain reconstruction rules after the original is optimised.

- **Operability**

Exploiting parallelism involves many different tasks — such as analysis, detection, optimisation, profiling, scheduling, and performance prediction. Hence, all the properties of representation mentioned above must be in a well-organised form and easy to process, abstract, reason, group, optimise and reconstruct.

The suitability of a programming language for parallelism investigation is examined in terms of the above properties. The basic expression unit of computation expressed in languages is the statement which have various forms in terms of syntax. The independence of computation can only be studied between statements. As a result, parallelism exploitation is considered at a statement level [JP93], [Bet al94b], [Bet al94a]. Though rich semantics of statements of a language brings people a lot of convenience to express computation (including parallelism), the consistency, granularity and operability of representation are limited. The temporal aspect of computation is usually specified completely in an implicit manner. Moreover, programming a solution needs to fit a real world into a specific architecture when a machine-dependent programming language is used. These issues complicate the expression of a problem. Computation inference

and analysis are therefore difficult to achieve in a programming language. Functional languages and specification languages have similar situations to some degree. In this respect, we cannot resist the temptation to quote from E.W. Dijkstra 20 years ago [Dij72]:

Another lesson we should have learned from recent past is that the development of 'richer' or 'more powerful' programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally.

Based on the definitions and properties described in Chapter 2, it is observed that ABCOM is designed particularly with intention to meet the requirements. ABCOM ensures independence and consistency in representation by unifying the computation unit into quadruples, i.e., elements of the tuple space. Abstracting a number of relevant elements of a *CDOAG* into a compound element changes the representation from the fine-grained to medium- or coarse-grained. The granularity of representation can be decided in terms of the size of the grain in ABCOM. The temporality of computation in ABCOM is completely and explicitly specified. To carry out inference, an organisation structure is needed to support manipulations, abstraction and modification on elements in ABCOM. In practice, a special programming database is a good environment to intensively support these tasks that cover different phases of parallel programming.

5.1.2 ABCOM tuplebase

Unlike an intermediate language of a compiler, ABCOM is represented as an information base where elements of a solution are expressed as record units of the tuplebase (called ABCOM *tuplebase*). Hence, the ABCOM tuplebase can be processed using relational database techniques.

The main schema contains five basic fields: four of these correspond to components of the element and the fifth denotes the *identifier* (ID) of an element in the tuplebase. Using such a schema, we can present the example in Fig. 4.15 into a tuplebase in Fig. 5.1.

A tuplebase for a given problem is created when a source code is transformed. Such a tuplebase consists of a number of database files created during transformation. These files are divided into two sub-tuplebases, that is, *Superblock_base* and *Branching_base*. A *Superblock_base* contains all elements generated when transforming a superblock excluding those elements which are logically contained in condition branching flows. Those converted elements arising from the statements between a branching statement and the associated merging statement are stored in a *Branching_base*. Thus the study of relations among data, operations and time can be carried out using relational algebra. To collect information and process data for computation inference in ABCOM, two important SQL-like functions are used.

1. **select** < *attributs* > **from** < *filename* > **where** < *condition* >

This function identifies the elements which meet certain conditions. Usually, a group of elements can be selected as they have the same logical execution time, or the same data object as one of their input or output, and so on.

2. **modify** < *filename* > **with** < *assignment* > **where** < *condition* >

Here < *assignment* > can be either an expression on **EX** or a substitution of a data object in **IN** or **OUT**. In general, any modification in a file is permitted if (and only if) it can be guaranteed that will not affect the correctness of the solution.

The relational algebraic primitive functions on the tuplebase can be combined with the definitions and properties of ABCOM, for computation inference, parallelising solutions, collecting profile information on parallelism, and predicting performance. Using relational techniques, thus, a parallel programming platform can be developed.

Generating *CDOAGs*

As previously mentioned, *CDOAG* plays an important role in ABCOM. Each element in ABCOM corresponds to an associated *CDOAG* that may be contained in another *CDOAG* belonging to other elements. Thus, which *CDOAGs* are to be generated and whether *CDOAGs* can be processed easily are critical to use ABCOM. We introduce two methods to construct *CDOAGs* from a given tuple space.

ID	OP	IN	OUT	EX
u_1	=	0	s	1
u_2	×	a_{21}, b_1	v_1	2
u_3	+	s, v_1	s	3
u_4	-	b_2, s	b_2	4
u_5	=	0	s	5
u_6	×	a_{31}, b_1	v_2	6
u_7	+	s, v_2	s	7
u_8	-	b_3, s	b_3	8
u_9	×	a_{32}, b_2	v_3	9
u_{10}	+	s, v_3	s	10
u_{11}	-	b_3, s	b_3	11
u_{12}	=	0	s	12
u_{13}	×	a_{41}, b_1	v_4	13
u_{14}	+	s, v_4	s	14
u_{15}	-	b_4, s	b_4	15
u_{16}	×	a_{42}, b_2	v_5	16
u_{17}	+	s, v_5	s	17
u_{18}	-	b_4, s	b_4	18
u_{19}	×	a_{43}, b_3	v_6	19
u_{20}	+	s, v_6	s	20
u_{21}	-	b_4, s	b_4	21

Figure 5.1: The conceptual schema in ABCOM tuplebase

- *Method 1*: Top-down strategy: Given a particular u_i , $CDOAG_{u_i}$ is generated by identifying the producers (elements) of its input and recursively finding producers of the input of those elements until it is found that all input data of new producers are specified. To generate $CDOAGs$ from a tuple space, only those elements that produce the output of the superblock are considered. The generation procedure is made efficient by starting from the elements having earlier execution orders.
- *Method 2*: Bottom-up strategy: Along with the original partial order generated in the trace generation of transformation, each element u_i determines an associated $CDOAG_{u_i}$ by combining existing $CDOAGs$ as its subgraphs. A special case is that if the input of u_i is specified, then $EDOAG_{u_i}$ itself is a specific $CDOAG$. Here generation of $CDOAGs$ is controlled by defining a number of output data we are interested in. Once all $CDOAGs$ corresponding to the data are constructed, the generation stops.

The $gen_CDOAG < x >$ is a function based on Method 1. The implementation of function $gen_CDOAG < x >$ using relational `select` queries is illustrated in Fig. 5.2. All elements found in a generated $CDOAG_{u_i}$ are stored in a `CDOAG_base` that has one more field than the schema of the main tuplebase, called `CDOAG_id` to keep the identifier of u_i .

Remark

To generate a $CDOAG$ we combine the individual dataflow relations between elements. Once a $CDOAG$ associated with a particular data object is generated, the dataflow computation feature relevant to computing this data is abstracted. Although there is an associated $CDOAG$ for each element in \mathcal{P} , it is not necessary to generate all of them since some of them are contained in others.

5.2 Relation-Based Computing Inference and Analysis

The efficiency of parallel computing depends on how to achieve the reduction in complexity of relations among *time*, *data* and *operations* so that high-performance can be obtained on a parallel architecture. In the explicit programming approach, inference of parallel computation for a given problem is done by a programmer. The results of

Input: an element u_i and a tuplebase U .

Output: all elements included in $CDOAG_{u_i}$ are found and stored in $CDOAG_base$

Temporal variables: $id1$, $op1$, $in1$, $out1$, $ex1$ and $cdoag1$.

Algorithm 3:

```

1.  if CDOAG_base does not exist then
2.      create CDOAG_base;
3.  put  $u_i$  into CDOAG_base (with its CDOAG_id= null);
4.  let %cdoag1='u_i'
5.  while  $\exists u_j \in CDOAG\_base \wedge$  its CDOAG_id is null then do
6.      for  $\forall x \in in_{u_j}$  do
7.          select into %id1, %op1, %in1, %out1, %ex1
              from U
              where EX =
8.              select Max(EX) from U
              where OUT like x and EX <  $ex_{u_j}$ ;
9.          if %id1=null then back to 6;
              / a vertices with indegree zero is reached. /
10.         select CDOAG_id into %id2
              from CDOAG_base
              where ID= %id1
11.         if %id2  $\neq$  null then do
12.             strcat %id3=(%id1,%id2)
13.             update CDOAG_base
              set CDOAG_id = %id3
              where ID= %id1
14.             back to 6
15.         insert %id1, %op1, %in1, %out1, %ex1,
16.         into CDOAG_base (ID,OP,IN,OUT,EX);
17.     endfor
18.     update CDOAG_base
19.     set CDOAG_id = 'u_i'
20.     where ID= 'u_j'
21. endwhiledo

```

Figure 5.2: Algorithm 3 for CDOAG generation

this inference are directly expressed in a program. In the implicit approach, inferences performed by a parallelising compiler, i.e. data dependence testing and parallelising programs, are limited to studying the relationship between statements. In this section we will show how the computation inference based on different aspects is achieved in ABCOM.

5.2.1 Time-based inference

Parallelism exploitation is based on the logical properties among computations (or the logical relation of computation execution). In programming, a programmer designs and implements an underlying computation in terms of certain programming logic. No substantial guideline can be derived from time-based computation inference to support parallelism analysis. Three typical relations between statements in a program are *sequencing*, *multi-tasking* and *synchronisation*. These three relations reflect the properties of control-flow. Data parallelism is expressed by a special statement with programmer's personal knowledge. Unfortunately, there are no effective tools for computation inference that systematically reveal computational characteristics of the problem. Although the statement-based DDG is used in data dependence testing, it does not characterise time-based computation logic.

The properties of ABCOM, such as temporality and consistency, enable time-based inference to be carried out.

(1) Computation latency analysis

Each element u_i in a solution \mathcal{P} has a legal-execution zone where parallelism and speedup can be obtained. The computation latencies are caused by subjective programming decisions. For a given element u_i its $t_{lower_{u_i}}$ can be obtained using the algorithm in Fig. 5.3. Similarly, $t_{upper_{u_i}}$ and the legal-execution zone Δ_{u_i} can be obtained.

Using the property expressed in Lemma 3.2, we find that computing a *CDOAG* can be speeded up by reordering certain execution orders of some elements and removing unnecessary partial orders. This can lead an optimised solution. We will discuss this optimisation using ABCOM in Chapter 6.

(2) Dataflow relation testing

Input: a given element u_i and tuplebase U .

Output: $t_lower_{u_i}$

Algorithm 4

- (1) $t_lower = 0$
- (2) **for** $\forall x \in in_{u_i}$ **do**
- (3) **select** EX **into** $\%t1$ **from** U
 where $EX =$
- (4) **select** $Max(EX)$ **from** U
 where OUT **like** x **and** $EX < ex_{u_i}$;
- (5) **if** $t_lower > \%t1$ **then** $t_lower = \%t1$
- (6) **enddo**

Figure 5.3: Algorithm 4 to obtain $t_lower_{u_i}$

In a dataflow computation the ordering of operations is determined by data interdependencies and availability of resources [Sha85]. The data flow relation, that exists between certain elements in ABCOM, is defined below.

Definition 5.1 Let $\{u_i, u_j\} \subset \mathcal{P}$; if data x is produced by u_i and later used as input data of u_j before it is modified, then there is a direct data-flow between u_i and u_j . We denote this by $u_i \xrightarrow{x} u_j$.

Note that if there are $u_i \xrightarrow{x} u_j$ and $u_i \xrightarrow{x} u_k$, then it means that there two data flows from the same element to different elements, denoted as

$$u_i \xrightarrow{x} \{u_j, u_k\}.$$

The data flow relation can be identified by using the following rule:

$$\begin{aligned} \text{Rule1.} \quad & \text{if } (\exists x, \{x\} = out_{u_i} \cap in_{u_j}) \wedge \\ & (\nexists u_k \in CDOAG_{u_j}) \wedge (x \in out_{u_k}) \wedge \\ & (ex_{u_i} < ex_{u_k} < ex_{u_j}) \\ & \text{then } u_i \xrightarrow{x} u_j; \end{aligned}$$

The dataflow inference detects data dependence. Two kinds of dataflow relations are

Input: a given element u_i and tuplebase U .

Output: elements which have input-dataflow relations associated with u_i .

Algorithm 5

- (1) **for** each data object $x \in in_{u_i}$ **do**
- (2) **select** id **from** U
 where $EX =$
- (3) **select** $Max(EX)$ **from** U
 where OUT like x and $EX < ex_{u_i}$;
- (4) **enddo**

Input: a given element u_i and tuplebase U .

Output: elements which have input-dataflow relations

Algorithm 6

- (1) **for** each data object $x \in out_{u_i}$ **do**
- (2) **select** id **from** U
 where $EX =$
- (3) **select** $Mim(EX)$ **from** U
 where IN like x and $EX > ex_{u_i}$;
- (4) **enddo**

Figure 5.4: Algorithm 5 and 6 for detecting data flow relations.

associated with a given element u_i in light of in_{u_i} and out_{u_i} respectively, which are called *input-dataflow relation* and *output-dataflow relation*. The algorithms for these are presented in Fig. 5.4 based on ABCOM tuplebase.

(3) Data-parallelism checking

In ABCOM, checking data parallelism is achievable in a step-wise manner if there are elements that meet the condition for data parallelism. However, data parallelism is not directly testable in an initial version of a solution converted from a sequential code. Thus, we do not discuss data parallelism inference and abstraction in ABCOM until we are able to parallelise a given solution.

(4) Static computation scheduling

Static computation scheduling is one of important issues in mapping an algorithm to a particular architecture [AS93], [Lil93], [Fea94], [NN94]. Static scheduling can be divided into two subtasks: *identifying candidates* and *making selection*. The degree

of parallelism achieved in a program is determined by the amount of parallelism found during static computation scheduling.

The static computation scheduling is grain-dependent and architecture-dependent. Using ABCOM, at a given timestep, static computation scheduling can use the following rule to identify those elements in the tuplebase which are ready for execution:

Lemma 5.1 *If $\exists u_i$ and $\forall x_k \in in_{u_i}$ have been specified at a given timestep $t_1 \leq ex_{u_i}$, and $\exists u_j$ in which $x_k \in out_{u_j}$ and $t_1 \leq ex_{u_j} \leq ex_{u_i}$, then element u_i is a candidate that is ready for scheduling.*

The above rule is general but only can address one of the two subtasks, and cannot be used efficiently in practice. To improve the efficiency, the rule can be implemented in different ways, in particular after a solution is optimised (see Section 7.6.2).

The static computation scheduling is grain-dependent and architecture-dependent. The scheduling at a fine-grain level looks into data parallelism, and at a coarse-grain level control parallelism is mainly considered. Using granularity and execution specification of ABCOM, we expect that the techniques of static scheduling can be benefited from time-based and dataflow related computation inference.

5.2.2 Data-based inference

Parallel programming experience shows that the decision made on data manipulation in programming can affect the performance. Different architectures (especially memory and interconnection structure used) require different data manipulation schemes. The ABCOM representation provides the following features to support data-based inference.

- **Data-access-pattern inference**

The data-access-pattern records how a data object is read or written for operations. The data-access-pattern is time-dependent because variables are usually reused. An access pattern of data x contains only one *write* access and all *read* accesses which are performed after this *write* and before next *write*. Because of the assumption of a CREW PRAM memory in ABCOM in Section 3.4.1, there

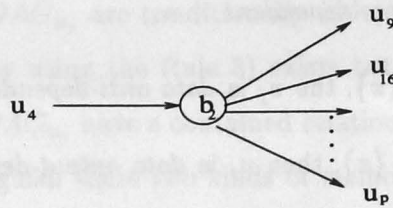


Figure 5.5: A data access pattern abstracted from Table 5.1

are two basic types of data access patterns. In the first type, the data a is only read by operation u_j after written by u_i and before the next 'write'. In the second type, the data b is read by a number of operations after written. It is often seen that there are a number of data-access patterns based on the same data object in a solution due to the reuse of variables. In ABCOM, a data-access pattern can be abstracted by checking all elements that perform 'read' accesses to a particular data between two 'write' accesses.

This method provides information of life cycles of data-access patterns, if all the patterns to the same data are abstracted along with logical execution of computation. If there is no *read* access between two *write* access to the same data, then the first *write* access is useless. Hence, programming errors can be detected. The life-cycle information of data-access patterns is very useful for variable reuse in programming. A data-access pattern illustrates the dataflow relations between the element of *write* access and the elements with *read* accesses. This detects data dependence among elements.

Using data-access-pattern inference, we abstract an access pattern of data object b_2 from the tuplebase in Table. 5.1. This pattern is represented in Fig. 5.5.

• Data-dependence testing

Except detecting dataflow relation, other kinds of data dependence are to be tested for computation analysis. Let \mathcal{P} be a solution converted from a sequential program. If any two elements $u_i, u_j \in \mathcal{P}$ are said to be data dependent on data object x , then there are following cases[Wol89] where $ex_{u_i} < ex_{u_j}$:

1. if $out_{u_i} \cap in_{u_j} = \{x\}$ and $\exists u_k$ in which $x \notin out_{u_j}$ and $ex_{u_i} < ex_{u_k} < ex_{u_j}$, then u_j is *data flow-dependent* on u_i ;
2. if $in_{u_i} \cap out_{u_j} = \{x\}$, the u_j is *data anti-dependent* on u_i ;
3. if $out_{u_i} \cap out_{u_j} = \{x\}$, then u_j is *data output-dependent* on u_i .

Both data anti-dependence and data output-dependence are called *memory-based dependence*. We denote the memory based dependence between u_i and u_j by

$$u_i \xrightarrow{x} u_j.$$

Using the concept of *CDOAG*, such a data dependence can be tested using Rule 2 and 3 respectively.

Case 1.

$$\begin{aligned}
 \text{Rule 2.} \quad & \text{if } (CDOAG_{u_i} \parallel_c CDOAG_{u_j} \vee \\
 & CDOAG_{u_i} \bowtie CDOAG_{u_j}) \wedge \\
 & ((in_{u_i} \cap out_{u_j} = \{x\}) \vee \\
 & ((out_{u_i} \cap in_{u_j} = \{x\}) \vee \\
 & (out_{u_i} \cap out_{u_j} = \{x\}))) \\
 & \text{then } u_i \xrightarrow{x} u_j.
 \end{aligned}$$

Case 2.

$$\begin{aligned}
 \text{Rule 3.} \quad & \text{if } (CDOAG_{u_i} \sqsubset CDOAG_{u_j} \vee \\
 & CDOAG_{u_j} \sqsubset CDOAG_{u_i}) \wedge \\
 & ((in_{u_i} \cap out_{u_j} = \{x\}) \vee \\
 & ((out_{u_i} \cap in_{u_j} = \{x\}) \vee \\
 & (out_{u_i} \cap out_{u_j} = \{x\}))) \wedge \\
 & \neg((u_i \rightarrow u_j) \vee (u_j \rightarrow u_i)) \\
 & \text{then } u_i \xrightarrow{x} u_j.
 \end{aligned}$$

The Rule 2 tests for a memory-based dependence between two elements whose $CDOAG_{u_i}$ and $CDOAG_{u_j}$ are conditionally independent or overlapping. The dependence (tested by using the Rule 3) exists between two elements of which $CDOAG_{u_i}$ and $CDOAG_{u_j}$ have a contained relation but not a dataflow relation. The reason we distinguish these two kinds of memory-based dependence is that in the second case there is actually an *indirect* data-flow relation between the elements. These two situations are treated in different ways when a solution is parallelised. The implementations of Rule 1, 2 and 3 are not complicated since the relations between $CDOAG$ s can be identified. $CDOAG$ -based computation inference described in Section 5.2.4 will address how to check relations between two $CDOAG$ s.

5.2.3 Operation-based inference

Operation-based inference is also useful to parallel computing. It has been shown in the last two subsections that computation inference based on either time or data could only reveal various logical and dependent relations between elements. We use operation-based inference in a similar way to assist computational analysis.

Data structures are mainly determined by a specific solution to a problem. To properly redefine or reconstruct data structures for solution optimisation and reconstruction, operation-based inference is required, including computation pattern tests and some special optimisation against a $CDOAG$. We discuss these techniques in Section 7.3.

The computation inference based on time, data and operations are fundamental and easy to understand in ABCOM. Various applications of this inference can be developed according to different interests. Some main applications of the inference are presented in Chapter 6 and 7.

5.2.4 $CDOAG$ relations inference

Based on dataflow relation inference, we can compose individual dataflow relations into a computation dataflow associated with related elements. The discussion presented in Chapter 2 and Section 5.1.2 shows that a $CDOAG$ contains all information required

Initial: given $CDOAG_{u_i}$ and $CDOAG_{u_j}$ in $CDOAG_base$.

Output: reporting a contained relation between $CDOAG_{u_i}$ and $CDOAG_{u_j}$.

Algorithm 7:

```

1.  %id1='u_i', %id2='u_j';
2.  select CDOAG_id into %cdoag_id
      from CDOAG_base where ID=%id1;
3.  if %cdoag_id=%id2 then
4.    writeln('CDOAG_{u_i}  $\sqsubset$  CDOAG_{u_j}'), exit;
5.  select CDOAG_id into %cdoag_id
      from CDOAG_base where ID=%id2;
6.  if %cdoag_id=%id1 then
7.    writeln('CDOAG_{u_j}  $\sqsubset$  CDOAG_{u_i}'), exit;
8.  exit

```

Figure 5.6: Algorithm 7 for testing a contained relation between $CDOAGs$

to complete a computation procedure and carry out computation inference. The four categories of the relations between two $CDOAGs$ are determined by the relevant computation features of them. According to the definitions of these categories, the relation between $CDOAG_{u_i}$ and $CDOAG_{u_j}$ is tested using the algorithms in Figs. 5.6, 5.7 and 5.8.

5.2.5 Nondeterministic computation analysis

Determinacy is important to exploit parallelism. As shown, in a $CDOAG$, deterministic computations are easily abstracted and represented using dataflow relation between elements. Inference to deterministic computations is thus developed with no difficulty. However, situation is different for nondeterministic computations.

A simple nondeterministic computation is expressed by a conditional statement in a source code. If a loop contains conditional statements, then computation inference becomes complicated. Consider the sequential code of *sorting*.

Example 8. A sequential sorting program is expressed as:

Initial: given $CDOAG_{u_i}$ and $CDOAG_{u_j}$ in $CDOAG_base$.

Output: checking whether $CDOAG_{u_i}$ and $CDOAG_{u_j}$ are overlapping.

Algorithm 8:

```

1.   %id1='u_i', %id2='u_j';
2.   declare CDOAG_overlapping cursor for
3.     select ID from CDOAG_base
       where CDOAG_id like %id1 and CDOAG_id like %id2;
4.   open CDOAG_overlapping ;
5.   fetch CDOAG_overlapping into %id3;
6.   if %id3 ≠ 'null' then
7.     writeln('CDOAG_{u_j} ⊆ CDOAG_{u_i}'),
       close cursor CDOAG_overlapping and exit
8.   close cursor CDOAG_overlapping and exit;

```

Figure 5.7: Algorithm 8 for testing $CDOAG_{u_i} \subseteq CDOAG_{u_j}$.

```

for i = 1 to n - 1 do
  for j = 1 to n - 1 do
    if a(j) > a(j + 1) then do
      t = a(j); a(j) = a(j + 1); a(j + 1) = t.

```

Here we use $n - 1$ instead of $n - i$ as the bound of the internal loop so that a complete computation space can be observed. We transform the above code into ABCOM when $n = 6$. In Fig. 5.9 the some elements generated from Example 8 are given. These elements are not directly performing *exchange* for the three assignment statements (called *threesort*) after each *if*, but they illustrate certain computation features of the loops.

Each element of subset $\{u_1, u_4, u_7, u_{10}, \dots, u_{73}\}$ performs a condition test to decide whether an associated element that carried out *exchange* of *threesort* should be executed with the a special compound operation (*if-inten*(e_i)). We illustrate the possible relations that exist among the elements with operation of *if-inten*(e_i) in Fig. 5.10.

The edge labelled by a cycle in Fig. 5.10 indicates a possible computation relation

Initial: given $CDOAG_{u_i}$ and $CDOAG_{u_j}$ in $CDOAG_base$ and they have a neither contained nor overlapping relation.

Output: reporting a conditionally independent relation or a completely independent relation between $CDOAG_{u_i}$ and $CDOAG_{u_j}$.

Algorithm 9:

```

1.   %id1='u_i', %id2='u_j';
2.   declare CDOAG_cond cursor for
3.     select IN from CDOAG_base
       where CDOAG_id = %id1;
   open CDOAG_cond;
   While CDOAG_cond is not empty do
   begin
4.   fetch CDOAG_cond into %input;
5.     for each data object x in %input do
6.       declare shared_var cursor for
7.         select IN from CDOAG_base
           where CDOAG_id = %id2 AND (IN like x OR OUT like x);
       open CDOAG_base;
8.       fetch shared_var into %sharedobject
9.       if %sharedobject ≠ 'null' then
10.        writeln('CDOAG_{u_i} ||_c CDOAG_{u_j}'), exit;
11.     repeat 5.
   end
12.  declare CDOAG_cond cursor for
13.    select OUT from CDOAG_base
       where CDOAG_id = %id1;
   open CDOAG_cond;
   While CDOAG_cond is not empty do
   begin
14.  fetch CDOAG_cond into %output;
15.    for each data object x in %output do
16.      declare shared_var cursor for
17.        select IN from CDOAG_base
           where CDOAG_id = %id2 AND (IN like x OR OUT like x);
       open CDOAG_base;
18.      fetch shared_var into %sharedobject
19.      if %sharedobject ≠ 'null' then
20.        writeln('CDOAG_{u_i} ||_c CDOAG_{u_j}'), exit;
21.    repeat 15
   end
22.  writeln('CDOAG_{u_i} || CDOAG_{u_j}');
22.  exit;

```

Figure 5.8: Algorithm 9 for testing $CDOAG_{u_i} ||_c CDOAG_{u_j}$.

$u_1 : (<b, \{a_1, a_2, 2\}, \{e_1 e_2, 1)$ $u_2 : (if - inten(e_1), \{a_1, a_2\}, \{a_1, a_2\}, e_1)$ $u_3 : (\odot, \{\emptyset\}, \{\emptyset\}, e_2)$ $u_4 : (<b, \{a_2, a_3, 4\}, \{e_3 e_4, 3)$ $u_5 : (if - inten(e_3), \{a_2, a_3\}, \{a_2, a_3\}, e_3)$ $u_6 : (\odot, \{\emptyset\}, \{\emptyset\}, e_4)$ $u_7 : (<b, \{a_3, a_4, 6\}, \{e_5 e_6, 5)$ $u_8 : (if - inten(e_5), \{a_3, a_4\}, \{a_3, a_4\}, e_5)$ $u_9 : (\odot, \{\emptyset\}, \{\emptyset\}, e_6)$ $u_{10} : (<b, \{a_4, a_5, 8\}, \{e_7 e_8, 7)$ $u_{11} : (if - inten(e_7), \{a_4, a_5\}, \{a_4, a_5\}, e_7)$ $u_{12} : (\odot, \{\emptyset\}, \{\emptyset\}, e_8)$ $u_{13} : (<b, \{a_5, a_6, 10\}, \{e_9 e_{10}, 9)$ $u_{14} : (if - inten(e_9), \{a_5, a_6\}, \{a_5, a_6\}, e_9)$ $u_{15} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{10})$	$u_{16} : (<b, \{a_1, a_2, 12\}, \{e_{11} e_{12}, 11)$ $u_{17} : (if - inten(e_{11}), \{a_1, a_2\}, \{a_1, a_2\}, e_{11})$ $u_{18} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{12})$ $u_{19} : (<b, \{a_2, a_3, 14\}, \{e_{13} e_{14}, 13)$ $u_{20} : (if - inten(e_{13}), \{a_2, a_3\}, \{a_2, a_3\}, e_{13})$ $u_{21} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{14})$ $u_{22} : (<b, \{a_3, a_4, 16\}, \{e_{15} e_{16}, 15)$ $u_{23} : (if - inten(e_{15}), \{a_3, a_4\}, \{a_3, a_4\}, e_{15})$ $u_{24} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{16})$ $u_{25} : (<b, \{a_4, a_5, 18\}, \{e_{17} e_{18}, 17)$ $u_{26} : (if - inten(e_{17}), \{a_4, a_5\}, \{a_4, a_5\}, e_{17})$ $u_{27} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{18})$ $u_{28} : (<b, \{a_5, a_6, 20\}, \{e_{19} e_{20}, 19)$ $u_{29} : (if - inten(e_{19}), \{a_5, a_6\}, \{a_5, a_6\}, e_{19})$ $u_{30} : (\odot, \{\emptyset\}, \{\emptyset\}, e_{20})$
--	--

Figure 5.9: Part of the ABCOM code of Example 8.

between two elements subject to the result of an associated condition test. Because there is an uncertainty of the relation, computation inference and abstraction of these elements require more investigation on what the real computational relation exists among these elements.

Though it is certain that there is some parallelism in data movement of Fig. 5.10, the inference techniques described in the previous subsections are not applicable due to the absence of explicit and deterministic specification of time. In order to exploit parallelism of nondeterministic computation, thus, one should be able to address the following questions:

- Whether an element is computationally independent from others?
- What is affected if the execution order is changed?
- Which elements can be parallelised according to the answers of the above questions and basic requirements of parallelism?
- How can all those possible dataflow relations be correctly performed in parallel?

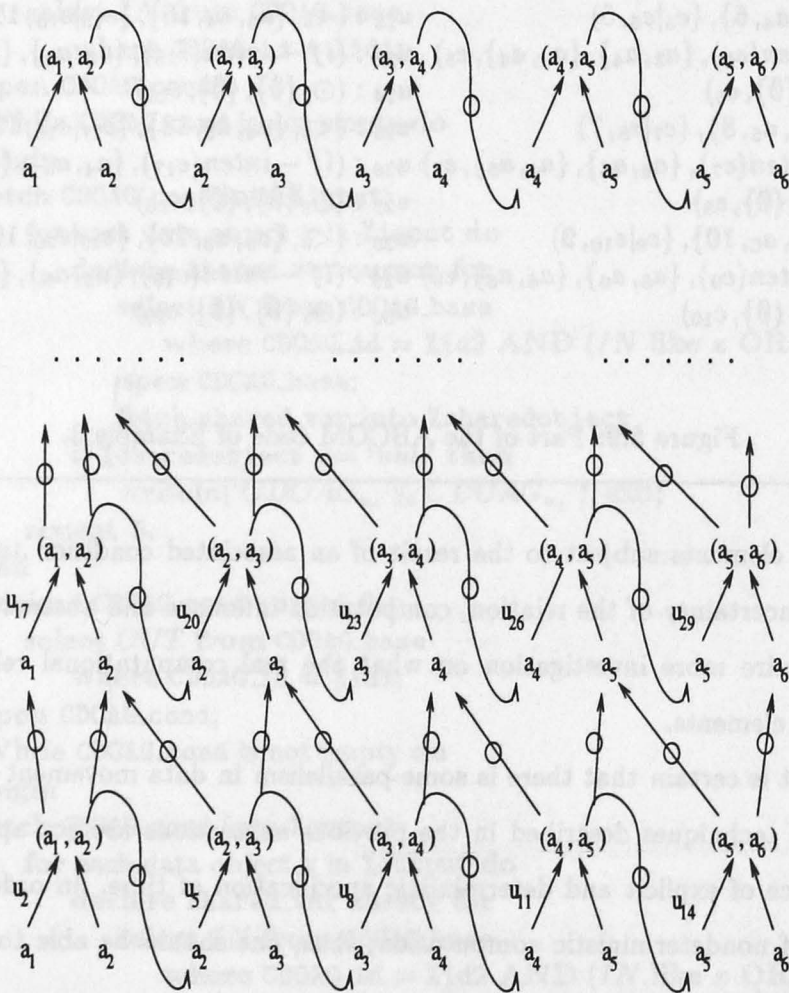


Figure 5.10: Uncertain relations among some elements of Example 8

5.2.6 Element-state based inference

Another important feature of computation execution in ABCOM is the evolution of element status along with logical execution of computation. Each element of ABCOM can only be found in one of the following groups:

1. *Waiting_group* (W_g)

If $u_i \in \mathcal{P} \wedge ex_{u_i} > EP$, and the input data objects of in_{u_i} have not been completely specified, then u_i is in W_g ;

2. *Ready_group* (R_g)

If $u_i \in \mathcal{P} \wedge ex_{u_i} > EP$, and the input data objects of u_i have been specified, then u_i is in R_g ;

3. *Execution_group* (E_g)

If $u_i \in \mathcal{P} \wedge ex_{u_i} = EP$, and all input data objects have been specified, then u_i is in E_g ;

4. *Conditional_execution_group* (C_g)

If $u_i \in \mathcal{P}$ and ex_{u_i} is an expression with the operation ' \pm ', then u_i is in C_g ;

5. *Post_execution_group* (P_g)

If $u_i \in \mathcal{P} \wedge ex_{u_i} < EP$, then u_i is in P_g .

During the execution of computation, an element may migrate from one group to another as certain execution conditions are met. The migration procedure of elements is outlined in Fig. 5.11.

The element migration between different groups is driven by certain conditions. The migration of u_i is denoted as:

$$(\text{Departuregroup}) \xrightarrow{u_i} (\text{Destinationgroup}).$$

In terms of Definition 3.10, the initial distribution of elements among these groups is determined by the decision of programming. When $EP = 0$, initially, all elements are distributed in W_g , R_g and C_g except $u_0 \in E_g$. The driven conditions can be explained in three categories:

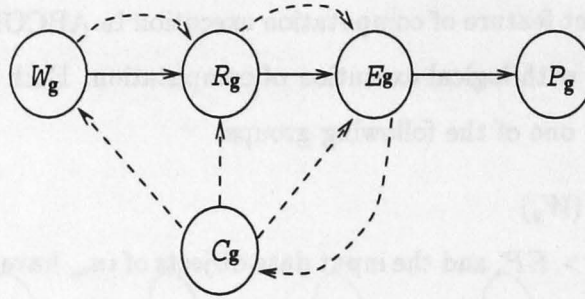


Figure 5.11: The procedure of element migration

1) **EP driven**

At each logical step some elements become currently executable and some become performed. We refer these changes of element states as the migration driven by EP . This migration happens between R_g and E_g , or E_g and P_g when $EP = EP + 1$, which are stated as:

- Rule 4.

$$\frac{(\forall u_i \in R_g) \wedge (ex_{u_i} = EP)}{R_g \xrightarrow{u_i} E_g};$$

- Rule 5.

$$\frac{\forall u_i \in E_g}{E_g \xrightarrow{u_i} P_g};$$

2) **Dataflow driven**

The element migration between W_g and R_g is driven by the dataflow.

- Rule 6.

$$\frac{(E_g \xrightarrow{u_i} P_g) \wedge (\exists u_j \in W_g) \wedge (u_i \xrightarrow{x} u_j) \wedge (\forall x_l \in in_{u_j}, \exists u_k, u_k \xrightarrow{x_l} u_j \wedge ex_{u_k} < ex_{u_i})}{W_g \xrightarrow{u_j} R_g}.$$

Remark

It is possible that some elements are driven by both dataflow and EP at a particular

timestep so that they can migrate from W_g directly to E_g in one logical step. In this case, these elements are executed in a dataflow computation fashion.

3) Conditional-control driven

It cannot be determined whether the elements in C_g are executed until the required conditions are tested during the execution. These elements would migrate to R_g and E_g only when the conditions are satisfied. To be consistent with the Definition 3.1, only the *head* element (a compound element of the elements converted from the branching flow) would migrate to other groups when the branching flow is determined to be executed. In this special unit of time when the head element is in E_g , all the elements (being in C_g) associated with the *head* element would complete their migrations in a '*hidden*' mode using its own relatively logical clock (intensional logic). The element migration in a '*hidden*' mode is shown by dot lines in Fig. 5.11.

The main restriction for the application of relation-based computation inference is that in order to perform inference, there must be certain kinds of relations that exist among elements with respect to data, operation or logical execution time. Using element-state based inference, we can carry out inference at a higher level for computation scheduling, balancing, simulation and performance prediction.

5.3 Summary

By developing an ABCOM programming database, we have shown how parallel computation analysis and inference can be performed using relational algebra, and rules. The inference features demonstrated in this chapter are based on three key factors of parallel computing, namely, *time*, *data* and *operations*.

...to that they are migrated from W_i directly to E_i in one logical step. In this

case, these elements are executed in a distributed computation fashion.



It cannot be determined whether the elements in C_i are executed with the required

conditions are tested during the execution. These elements would migrate to E_i and E_i

only when the conditions are satisfied. To be consistent with the Definition 3.1, only the

head element (a compound element of the elements converted from the branching flow)

would migrate to other nodes when the branching flow is determined to be executed.

In this special case of time when the head element is in E_i , all the elements (body

in C_i) associated with the head element would complete their migration in a similar

mode using its own relatively logical clock (internal logic). The element migration

is independent of the migration of other elements in the system.

The migration of the head element is independent of the migration of other elements in

the system. The migration of the head element is independent of the migration of other

elements in the system with respect to data, operation or logical execution time. Using

element-level based inference, we can carry out inference at a higher level for complex

inference scheduling, balancing, simulation and performance prediction.

$$R_i \rightarrow E_i$$

5.3 Summary

In developing an ABDM processing database, we have shown how parallel

computation analysis and inference can be performed using relational algebra and

rules. The inference process demonstrated in this chapter is based on three key

factors of parallel computing: namely, data and operations.

$$\frac{(E_i \rightarrow R_i) \wedge (R_i \rightarrow W_i) \wedge (W_i \rightarrow E_i) \wedge (E_i \rightarrow W_i) \wedge (W_i \rightarrow R_i) \wedge (R_i \rightarrow E_i)}{W_i \rightarrow R_i}$$

Chapter 6

Solution Parallelisation in ABCOM

A parallelising compiler detects potential parallelism in a source code, and implements it by a transformation. The result of the transformation is an optimised parallel program that can be executed in a specific architecture. The amount of parallelism achieved in the optimised program is dependent upon the amount of parallelism can be exploited by the compiler and how that can be realised in the architecture. We call such a procedure as “*program parallelisation*”. After a source code is transformed into ABCOM, the program takes a new form preserving the original execution semantics. How we can optimise this solution in ABCOM and to what extent the parallelism can be exploited are described in this chapter. To distinguish this optimisation from program parallelisation, we call it “*solution parallelisation*” since this optimised solution is executable on ABCOM machine and but may not be physically implementable. The purpose of optimisation is to reveal parallelism in a programmer-view independent manner.

6.1 Overview of Optimising Compilers

In the last decade optimising compilers have become an essential component of high-performance computer systems. The survey by Bacon *et al* [BGet al94] provides the state of the art in this area. Developing a framework that unifies the transformations is important in this area of research. We briefly review studies relevant to parallelisation.

6.1.1 Existing studies

1. Data Dependence Analysis

Dependence analysis and tests have been conducted in [Lam74], [WB87], [Ban88], [Pug92], [PP92], [JP93], [Lil94]. The definitions of data dependence relations given by Wolfe and Banerjee in [WB87] are stated below:

Given two statement S_v and S_w , the following *data dependence relations* may hold true or the statements may be data independent.

1. If some item $X \in OUT(S_v)$ and $X \in IN(S_w)$ and S_w is to use the value of X computed in S_v , then we say that S_w is *data flow-dependent* on S_v .
2. If some item $X \in IN(S_v)$ and $X \in OUT(S_w)$, but S_v is to use the value of X before it is changed by S_w , then we say that S_w is *data anti-dependent* on S_v .
3. IF $X \in OUT(S_v)$ and $X \in OUT(S_w)$ and the value computed by S_w is to be stored after the value computed by S_v , we say S_w is *data output-dependent* on S_v .

Here the data dependence is defined between statements (and differs from the dependence we have defined in Chapter 5). In traditional data dependence test a *Data Dependence Graph*(DDG) is used. This is a *statement-based* dependence graph. A statement S_v in a loop is designed for performing a number of instances in the iteration space of the loop. The dependence between instances of different statements in different iterations, the *dependence distance vector* and *dependence direction vector* are introduced [Wol89]. The dependence behaviour of a loop is described by the set of dependence vectors for each pair of possibly conflicting references. Determining data dependences is equivalent to testing whether there exists an integer solution to a set of linear equalities and inequalities It is an NP-complete problem [Pug92]. If the dependence information is inexact, the compiler must act conservatively, rejecting some transformations because they violate a constraint that may or may not be real [Pug92], [PW94], that is, some *false dependences* may be reported.

In addition, there are a number of *exact* tests that exploit some subscript characteristics to determine whether a particular type of dependence exists [Ban88], [Gea91], [Lea90], [Mea91], [Pug92], [Wol89], [WT92], [JP93].

2. Transformations

(1) Data-flow-based loop optimisation

A number of loop optimisations based on data-flow analysis are summarised in the 'Red Dragon' book by Aho et al [ASU86]. These include *loop-based strength reduction*, *induction variable elimination*, *loop-invariant code motion* and *loop unswitching*, which are used to optimise the computation cost of loops.

(2) Loop reordering

Loop reordering changes the relative order of execution of the iterations of a loop nest or nests. Such a transformation exposes parallelism and improves memory locality. Whether a loop can be parallelised is determined by the test result of data dependence. The loop reordering can be done using different methods, such as *loop interchange*, *loop skewing*, *loop reversal*, *strip mining*, *cycle shrinking*, *loop tiling*, *loop distribution* and *loop fusion*.

(3) Loop restructuring

Loop restructuring changes the structure of the loop, but leaves the computations performed by an iteration of the loop body with their relative order unchanged. The main approaches to loop restructuring are *loop unrolling*, *software pipelining*, *loop coalescing*, *loop collapsing*, *loop peeling*, *loop normalisation* and *loop spreading*.

Since these transformations are based on the relations between statements, the space for optimisation is limited by the context of program. The computation space (spatial structure) of a problem is not exhibited. That is the reason why various attempts have been made to explore certain individual parallel properties which can be detected by some tests.

6.1.2 Problems

The success of the applications of various transformations relies on the data dependence testing. To compute dependence information among the iterations of a loop,

we need to understand the use of arrays referenced in the loop. The complete information on a data dependence relation includes three aspects: (i) *data* that is the carrier of the dependence; (ii) *operation* which determines the nature of the dependence in the combination with the another aspect, and (iii) *time*. Dependence vectors describe dependence among *iterations* (conveying only information about the time), but not the precise information on *data objects* and *operations*. In other words, the data dependence exists among the operations on data objects, but people use only the dependences among iterations (or statements) as the abstract description (distance or direction vectors) for data dependence tests. In this case, testing dataflow relation or data dependence relation becomes complicated since each statement corresponds to a number of instances in different iterations.

The concept of data flow is directly or indirectly used by all those transformations. For individual transformation, however, only a certain part (sometime only a small part) of dataflow features inherent in computation is exploited. All dataflow features of a program could not be exploited completely in a certain transformation.

An important feature of using a loop is to let a certain computation pattern (body of a loop) to be repeated properly over a data domain of any size, as long as the loop control variable is defined. It is often seen that a loop body, which is "*small*" in the size of text, processes a data domain which is much much "*larger*" than the loop in size. Any optimisation is always developed against a particular object, called *optimisation space*. In conventional compilers, the optimisation space used is the context of a program, i.e, statements of loop rather than a space associated with the data domain. Selecting such an optimisation space has created certain difficulties in exploiting parallelism. Fox points out [Fox92] that:

The spatial (data) parallelism of the problem becomes purely temporal in the software, which implements this as a *Do loop*. Somewhat perversely, a parallelising compiler tries to convert the temporal structure of a *Do loop* back into spatial structure to achieve data parallelism on a spatial array of computers. Often parallelising compilers produce poor results as the original map of the problem into sequential Fortran 77 has 'thrown away' information necessary to reverse this map and recover unambiguously

the spatial structure. The first (and some ongoing) efforts in parallelising compilers tried to directly 'parallelise the *Do-loops*'. This seems doomed to failure in general as it does not recognise that in nearly all cases the parallelism comes from spatial and not in control (time) structure.

Consequently, the challenge is how to find a suitable method to recover the information on the spatial structure from a sequential program, or how to recover the information necessary to do a reverse mapping from the temporal to the spatial aspect unambiguously.

Before ABCOM-based solution parallelisation is discussed, we recall the definition of *transformation* given in [BGet al94]:

A Transformation is *legal* if, for all semantically correct program executions, the original and the transformed programs produce exactly the same output for identical executions.

The transformation techniques of ABCOM described in Chapter 4 preserve the original execution semantics of a source code using a trace-generation strategy. The total order is generated by sequential execution. *CDOAG* provides the complete information to compute a particular data object from both topological and temporal points of view. Abstracting all *CDOAGs* associated with the output of a problem (loops), we can clearly obtain the spatial structure of the problem. And partial ordering in *CDOAG* tells us which computation element is executed at each logical step in this particular code.

Therefore, the problem Fox pointed out can be solved using ABCOM transformation and associated techniques. That is, the information on the spatial structure that was thrown away in the sequential code, can be recovered in an abstract computation tuple space. How we can effectively use this information to parallelise a solution is discussed below.

6.2 ABCOM-Based Data Dependence Tests

6.2.1 Dependence representation

Data dependence relations exist between elements (that contain information of operation, data and time) of the tuple space. Let us recall the definition of data dependence given in Chapter 4.

If \mathcal{P} is a solution and any two elements $u_i, u_j \in \mathcal{P}$ are said to be dependent on data object x , then there are following cases that correspond to the definitions in [WB87], where if $ex_{u_i} < ex_{u_j}$ and $\exists u_k$ in which $x \in out_{u_j}$ and $ex_{u_k} \leq ex_{u_j}$:

1. $out_{u_i} \cap in_{u_j} = \{x\} \rightarrow$ data flow-dependence, denoted by $u_i \xrightarrow{x} u_j$;
2. $in_{u_i} \cap out_{u_j} = \{x\} \rightarrow$ data anti-dependence, denoted by $u_i \xleftarrow{x} u_j$;
3. $out_{u_i} \cap out_{u_j} = \{x\} \rightarrow$ data output-dependence, denoted by $u_i \xleftrightarrow{x} u_j$.

Data dependence relations can be detected using the inference rules presented in Chapter 5. As each data object is read and written in a certain access pattern, there are m dataflow dependence relations from an element to m different elements, called 1-to- m dataflow dependence.

The dependence relations discussed above are based on those variables that are either singletons or elements with fixed indexes of arrays. If dependent relation is related to a variable that has a functional index, then that relation is relevant to certain elements which are covered by the index function. It is a dynamic relation and cannot be tested exactly before execution. Therefore, it is necessary to relate all elements covered by the function, or the whole array. To detect these dependence relations, we modify the rules 1, 2 and 3 described in Chapter 4 as follows:

$$\begin{aligned}
 \text{Rule 4.} \quad & \text{if } (\exists u_j, a_x \in out_{u_j}) \wedge \\
 & (\forall, a^* \in out_{u_i}) \wedge \\
 & ((\exists u_k \in CDOAG_{u_j}) \wedge (a^* \in out_{u_k}) \wedge \\
 & (ex_{u_i} < ex_{u_k} < ex_{u_j})) \\
 & \text{then } u_i \xrightarrow{a^*} u_j;
 \end{aligned}$$

Rule 5. if $((\exists u_j, a_x \in out_{u_j}) \wedge (\forall u_i, a_\star \in in_{u_i})) \vee$
 $((\exists u_i, a_x \in out_{u_i}) \wedge (\forall u_j, a_\star \in out_{u_j})) \wedge$
 $((CDOAG_{u_j} \parallel_c CDOAG_{u_i}) \vee$
 $(CDOAG_{u_j} \bowtie CDOAG_{u_i}))$

then $u_j \xrightarrow{a_\star} u_i$;

Rule 6. if $((\exists u_j, a_x \in in_{u_j}) \wedge (\forall u_i, a_\star \in out_{u_i})) \vee$
 $((\exists u_i, a_x \in out_{u_i}) \wedge (\forall u_j, a_\star \in out_{u_j})) \wedge$
 $((CDOAG_{u_j} \sqsubset CDOAG_{u_i}) \vee$
 $(CDOAG_{u_i} \sqsubset CDOAG_{u_j})) \wedge$

$\neg(u_i \xrightarrow{a_\star} u_j)$

then $u_i \xrightarrow{a_\star} u_j$;

Here a_x is a element referenced by a function; a_\star stands for any element of the array related to a_x . According to the nature of the index function, it is possible to divide an array into two parts. One of them is related to the function, the other is not. For instance, if there is a functionally indexed variable $a(Q(i))$ with $Q(x, i) = 2 \times x \times i$, assuming x be an integer variable, then all elements with the indexes of even values are functionally related, while, those that have indexes of odd values are not.

6.2.2 Features of ABCOM-based detection

ABCOM-based detection of data dependence provides exact results in terms of the discussion in Section 5.2. The reason for the simplicity in testing is due using the spatial structure of a problem. This can be stated as follows:

1. From iteration abstraction to trace generation

The computation space of a loop (not value-control While--Do) is divided into a number

of subspaces that are processed in different iterations. The mathematical abstraction used by many tests can only indicate reference relations of data variables among iterations. The temporal or topological characteristics of the computation on the data domain cannot be properly described since the information on data dependence relations between statements or between iterations turns out to be vague and often incomplete.

Using trace generation, the ABCOM transformation converts a loop into a tuple space in which the computation is organised in a partial order. Moreover, the data dependence relations are detected between elements of ABCOM. In other words, a “*smaller space*” with cyclic structure (loop) is replaced by “*larger space*” that can be abstracted with acyclic structures (*CDOAG*). The result of this replacement is to obtain a clear description on data dependence relation and a much larger space to exploit parallel properties of computation.

2. *CDOAG* privatisation

The reuse of variables (in programming) reduces resource cost of computation. The data dependence relations caused by reusing variables, called memory-based dependence, can be removed if they can be tested. According to the rules of data dependency testing, the concept of *CDOAG* is critical for testing memory-based dependence effectively. In fact, if there are two elements u_i and u_j in which there is a shared data object as input or output, and $CDOAG_{u_i}$ and $CDOAG_{u_j}$ are not in the relation of the contained, then there is a memory-based dependence relation that needs to be eliminated for optimisation. This can be stated by the following theorem.

Lemma 6.1 *The data dependences that can be eliminated between two elements for optimisation are only those for which the corresponding *CDOAG*s of the elements are not contained within each other.*

Proof

As described in Chapter 3, there are four categories of relations between any two *CDOAG*s. If two *CDOAG*s are completely independent, there is no data dependence between them. To test for a memory-based dependence between two elements, one needs to consider the other three categories. In fact, we can exclude the two situations:

(i) there is a direct data-flow relation; or (ii) there is an indirect data-flow relation that is a memory-based dependence, but is not concerned for elimination since the partial order between the elements is necessary. These two situations can be tested by checking whether the two corresponding *CDOAGs* of them have a contained relation by using the Rule 3 in Chapter 5. In other words, if two $CDOAG_{u_i}$ and $CDOAG_{u_j}$ are in the contained relation, then there must be a direct or indirect dataflow relation between u_i and u_j , namely, the partial ordering of execution is necessary. Consequently, the data dependence caused by a shared data object needs to be removed for optimisation if and only if these two *CDOAGs* are overlapping or conditionally independent. \square

3. Dependence test by using data-access patterns

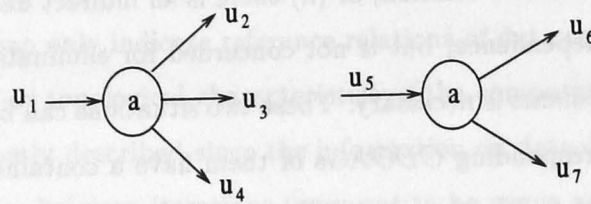
As shown in Chapter 5, in ABCOM, the history of reuse of variables is easy to be abstracted by data-based inference. The memory-based dependence between two access patterns to the same data object can be of four types:

1. $write1 - -write2$;
2. $write1 - -read2$;
3. $read1 - -write2$ and
4. $read1 - -read2$.

This is illustrated in Fig. 6.1 where 1) and 6) are of dataflow relation; 2), 3) and 4) are considered as being of memory-based dependence; and but 5) means that two groups $\{u_2, u_3, u_4\}$ and $\{u_6, u_7\}$ are exclusive each other, denoted by $\leftarrow\rightarrow$ and there must be a *write* access to the data between $\{u_2, u_3, u_4\}$ and $\{u_6, u_7\}$.

6.3 Parallelisation in ABCOM

As described in Section 3.2, the execution order of an element u_i can be legally modified within a certain range. By carrying out such modification for certain elements of a solution, the performance of the solution can be improved. The elements can be executed in parallel as long as their execution conditions are met.



- 1) $write1 - -read1: u_1 \xrightarrow{x} \{u_2, u_3, u_4\}$
- 2) $write1 - -write2: u_1 \xrightarrow{x} u_5$
- 3) $write1 - -read2: u_1 \xrightarrow{x} u_5$
- 4) $read1 - -write2: \{u_2, u_3, u_4\} \xrightarrow{x} u_5$
- 5) $read1 - -read2: \{u_2, u_3, u_4\} \xrightarrow{x} \{u_6, u_7\}$
- 6) $write2 - -read2: u_5 \xrightarrow{x} \{u_6, u_7\}$

Figure 6.1: Two access patterns based on the same data object

6.3.1 CDOAG optimisation

The partial order among elements may or may not be necessary. For a given source code the trace generation produces an ABCOM code without changing any execution order of computation. Under the driven condition of dataflow, a certain number of elements become ready for execution at a given timestep. In light of the driven condition of EP , while, only some of these elements, whose execution orders are equal to the current value of EP , become currently executable. This shows where possible speedup can be made. Let us start with the situation within a $CDOAG_{u_i}$. Assume that out_{u_0} be viewed as the input data for all vertices with indegree zero of $CDOAG_{u_i}$. If $CDOAG_{u_i}$ is composed of $ex_{u_i}, ex_{u_j}, \dots, ex_{u_q}$ and completely independent from other $CDOAGs$, then we define $T_{u_i} = Min\{ex_{u_i}, ex_{u_j}, \dots, ex_{u_q}\}$ as the lower bound of execution of $CDOAG_{u_i}$. According to the expression 3.2 in Lemma 3.4 we can write

$$\begin{aligned} \Delta_{T_{u_i}} &= Min\{ex_{u_i}, ex_{u_j}, \dots, ex_{u_q}\} - ex_{u_0} \\ &= T_{u_i} \end{aligned}$$

That is, the lower bound of executing $CDOAG_{u_i}$ can be at any timestep within the legal-execution zone of $[1, T_{u_i}]$. In order to parallelise execution of the elements in $CDOAG_{u_i}$, one can change certain execution orders of elements within their legal-execution zones so that all elements can be performed as early as possible. This procedure can start from those elements that are ready at beginning (when $EP = 0$), and can be continued along with the directions of data flows until u_i is processed.

Theorem 6.1 *If there is a $CDOAG_{u_i} \in \mathcal{P}$ in which all sub- $CDOAG$ s have no memory-based dependence, and for $\forall u_q \in \mathcal{P}$, $CDOAG_{u_q} \not\subseteq CDOAG_{u_i}$ and $CDOAG_{u_q} \parallel CDOAG_{u_i}$, then $CDOAG_{u_i}$ can be optimised until $\forall u_k \in CDOAG_{u_i}$ can be executed at their lower bounds, and $T_{u_i} = 1$.*

Proof (A sketch)

- $CDOAG_{u_i}$ can be optimised without losing correctness since it is a completely independent computation task according to the conditions that for $\forall u_q \in \mathcal{P}$, $CDOAG_{u_q} \not\subseteq CDOAG_{u_i}$ and $CDOAG_{u_q} \parallel CDOAG_{u_i}$.
- Lemma 3.3 indicates that there is a safety-execution zone for ex_{u_j} of $u_j \in CDOAG_{u_i}$. In order to optimise computations, an element could be executed at its lower bound if there is no memory-based dependence with other elements, that is, $ex_{u_j} = \text{Max}\{ex_{u_k}, ex_{u_l}, \dots, ex_{u_p}\} + 1$ where in_{u_j} is provided by the output of u_k, u_l, \dots, u_p . Let this optimisation procedure start from the elements in which all input data objects are those vertices with indegree zero, and be repeated to all their successive elements until u_i is processed.
- If all input of $u_j, \dots, u_q \in CDOAG_{u_i}$ are those vertices with indegree zero, namely, they have been specified when $EP = 0$, then $ex_{u_j}, \dots, ex_{u_q}$ can be reduced to 1 by the optimisation described above since it is defined that out_{u_0} is considered as the precedence of them. As a consequence, $T_{u_i} = 1$ can be reached in terms of the definition of T_{u_i} .

Using the approach described by Theorem 6.2, a given $CDOAG_{u_i}$ can be optimised until a special solution to $CDOAG_{u_i}$ is reached. In this solution the parallelism is

-
- Input: Given a set $B1$ of all elements of $CDOAG_{u_i}$ in a tuplebase, a temporal set $B3$ of elements which contains u_o initially.
- Output: An optimised solution to $CDOAG_{u_i}$ stored in $B2$.
- Algorithm 10:
1. Select an element u_j which has the smallest value of EX in $B1$;
 2. For $\forall x \in in_{u_j}$ do
 - Select u_l in which ex_{u_l} has the maximum value of EX among those elements producing x in $B2$;
 - Put u_l into $B3$
 - / if selection fails, then x is a vertex with indegree zero./
 - end
 3. Let $t_{lower} = 1 + Max\{\forall ex_{u_k}, u_k \in B3\}$;
 4. Remove all elements from $B3$ except u_o ;
 5. Modify ex_{u_j} by using t_{lower} ;
 6. move u_j from $B1$ to $B2$;
 7. If u_j is not u_i then back to 1;
 8. Exit.

Figure 6.2: Algorithm optimising a $CDOAG$.

objective, and all elements of $CDOAG_{u_i}$ are performed in a dataflow computation fashion. The procedure of this optimisation is done by Algorithm 10 in Fig. 6.2.

A critical path of a solution represents the sequence activities in a program that takes the longest time to execute. Using this concept, we can check our speedup of the parallelisation for a given $CDOAG_{u_i}$. Assume there be n elements contained by $CDOAG_{u_i}$. It takes n logical steps to sequentially compute $CDOAG_{u_i}$. The depth of $CDOAG_{u_i}$, h_{u_i} defined in Chapter 3, is in the relation of $h_{u_i} < ex_{u_i} = n$ initially. If $CDOAG_{u_i}$ is optimised by using the approach stated above, then we will get a new critical path $ex_{u_i}^o$ and a speedup of computation which can be described as the following theorem.

Theorem 6.2 *If there are n elements in $CDOAG_{u_i}$ that is optimised by using the approach of Theorem 6.2, then $ex_{u_i}^o = h_{u_i}$ holds, and the speedup is*

$$S_{CDOAG} = \frac{ex_{u_i}}{ex_{u_i}^o} = \frac{n}{h_{u_i}}. \quad (6.1)$$

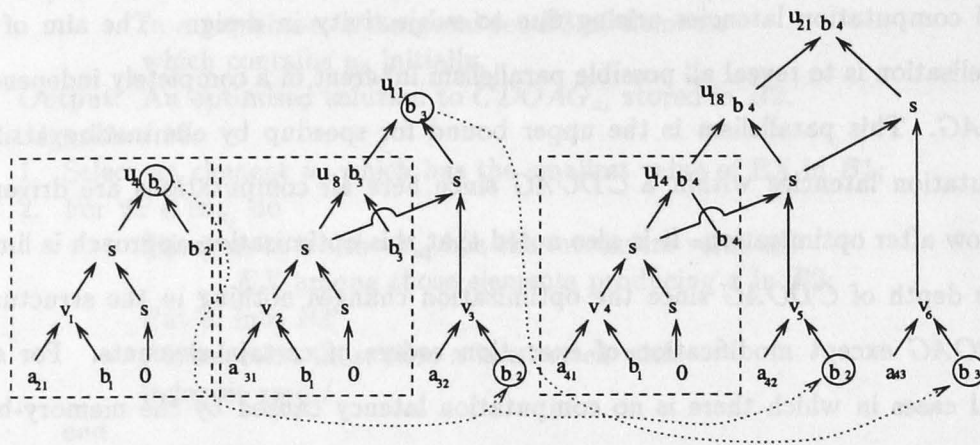
The parallelisation techniques for a *CDOAG* described so far are generally suitable for all computation latencies arising due to subjectivity in design. The aim of this parallelisation is to reveal all possible parallelism inherent in a completely independent *CDOAG*. This parallelism is the upper bound for speedup by eliminating artificial computation latencies within a *CDOAG* since here all computations are driven by dataflow after optimisation. It is also noted that this optimisation approach is limited to the depth of *CDOAG* since the optimisation changes nothing in the structure of a *CDOAG* except modification of execution orders of certain elements. For some special cases in which there is no computation latency caused by the memory-based data dependence, however, one may still be able to optimise a *CDOAG* by using other approaches. A typical example is a sequential *Sum* computation. We will discuss certain optimising techniques for such cases in the next chapter.

6.3.2 Solution parallelisation

The parallelisation achieved by the above approach is suitable to a *CDOAG* that is completely independent from other *CDOAGs* and does not have any two sub-*CDOAGs* having a memory-based dependence relation. In practice, it is often seen that there is memory-based dependence between two *CDOAGs* or two sub-*CDOAGs* in a *CDOAG*. To optimise these *CDOAGs*, eliminating the memory-based dependence is necessary.

As one of the main techniques of optimising compilers, *variable renaming* [Ell86], [PKL80] is widely used to remove the memory-based dependence. In order to achieve as much parallelism as possible, new names are introduced for disjoint uses of the same variable. The approach of variable renaming in ABCOM can be based on *CDOAGs* and the history of data-access patterns.

For example, three *CDOAGs* shown in Fig. 6.3, which contain the first twenty-one elements of Example 4 in Fig. 4.17, are not completely independent because the left one is contained in the other two through b_2 indicated as the dot lines, and the middle one is similarly contained in the right one through b_3 . But three sub-*CDOAGs* marked by boxes are conditionally independent due to a shared data object s . If all s occurring in the middle and right *CDOAGs* can be replaced by new data objects s_1

Figure 6.3: The *CDOAGs* of Example 6.

and s_2 respectively, we can find following facts:

1. $CDOAG_{u_4} \parallel CDOAG_{u_8} \parallel CDOAG_{u_{15}} \parallel \dots;$
2. $CDOAG_{u_{11}} \parallel CDOAG_{u_{18}} \parallel \dots;$
3. $CDOAG_{u_{21}} \parallel \dots;$
-

Using data-access patterns, variable renaming can be easily carried out since each pattern contains all elements that are needed to be renamed if the data accessed by this pattern is selected for renaming. The relation that exists between two patterns of the same data object corresponds to three different cases:

In the first case, there is a dataflow relation between two patterns. For instance, two patterns in Fig. 6.4 are related since there is u_4 in which $\{a\} \subset in_{u_4} \cap out_{u_4}$. In the converted code there must be $ex_{u_2} < ex_{u_4}$ and $ex_{u_3} < ex_{u_4}$ due to the anti-output dependence. To eliminate the dependences between u_2 and u_4 , or u_3 and u_4 of data a , a new variable could be introduced to replace a in $in_{u_5}, in_{u_6}, in_{u_7}$ and out_{u_4} . This kind of renaming is not necessary if the first pattern is not of 1-to- m form of dependence

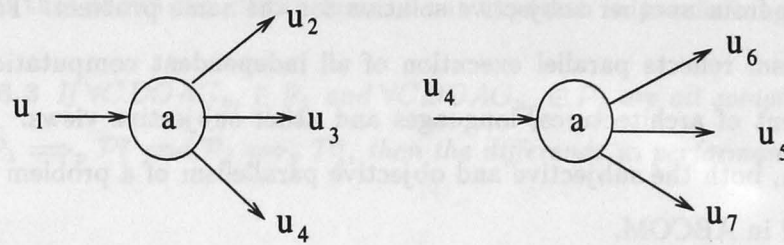


Figure 6.4: Two access patterns of data object a .

since it is caused by an expression of the form $a = a + x_i$.

In the second case there is an indirect dataflow relation between two patterns. It means that the *CDOAG* of the element having a 'write' access in the pattern 2 contains at least one of the elements having a 'read' access in the pattern 1. Thus, certain execution orders appearing in the indirect dataflow relations among the elements of the two patterns are necessary. Renaming variables in this case will not lead to give more parallelism to improve performance.

In the third case, two *CDOAGs* of the elements having 'write' accesses in two patterns are conditionally independent. To parallelise such two *CDOAGs*, the two patterns should not share a variable. Renaming will help to remove the dependence caused by the shared variable.

If we parallelise all independent computation tasks (superblocks or nested loops) in a given solution by modifying unnecessary partial orders, a special solution free of artificial sequentiality is got. The parallelism in this solution is not constrained by any computation model, architecture or subjective view of programming. Consequently, this parallelism is *objective*. Applying Theorem 6.1 to all *CDOAGs* of a solution is equal to carrying out a procedure in which a given solution in Category 1 or Category 2 described in section 2.2 is optimised until an equivalent and optimised solution that belongs to Category 3 is arrived at. The properties of the objective parallelism and the subjective parallelism can be stated thus:

- An important feature of subjective parallelism is that it arises in a physical im-

plementation. Hence the performance of one subjective solution can be quite different from another subjective solution for the same problem. The objective parallelism reflects parallel execution of all independent computation tasks independent of architectures, languages and other subjective views. For a given problem, both the subjective and objective parallelism of a problem can be represented in ABCOM.

- The degree of the objective parallelism turns out to be better than that of the subjective parallelism. The performance of an objective solution is better than the performance of a subjective solution. The difference between the objective parallelism and the subjective parallelism of an implementation can be used to examine whether more parallelism should be considered. This helps us to evaluate whether the implementation is developed successfully.

One may ask whether a unique solution with objective parallelism can be got if two different subjective solutions are optimised. The answer is yes if two solutions have exactly the same data domain and the same set of operations. In fact, this requirement is not necessary from the point of view of checking the effectiveness of this optimisation. Let us consider a general situation.

Let $\mathcal{P}_1 \Rightarrow_p \mathcal{P}_1^o$ denote that an optimised solution \mathcal{P}_1^o is obtained by parallelising \mathcal{P}_1 . For a solution \mathcal{P} if there are $CDOAG_{u_i}, CDOAG_{u_j}, \dots, CDOAG_{u_k}$ that are completely independent, and $h_{u_i}, h_{u_j}, \dots, h_{u_k}$ are the depths of $CDOAG_{u_i}, CDOAG_{u_j}, \dots, CDOAG_{u_k}$, then we define the critical path of a solution \mathcal{P} in ABCOM as the maximum value of ex among the all elements contained in \mathcal{P} , denoted as $H_{\mathcal{P}} = \text{Max}\{ex_{u_i} \mid \forall u_i \in \mathcal{P}\}$. Theorem 6.2 can directly lead to the following three corollaries.

Corollary 6.1 *If $\mathcal{P}_1 \Rightarrow_p \mathcal{P}_1^o$, then for $\forall u_i \in \mathcal{P}_1$ there is $ex_{u_i} = h_{u_i}$ for $CDOAG_{u_i}$.*

Corollary 6.2 *If $\mathcal{P}_1 \Rightarrow_p \mathcal{P}_1^o$, then $H_{\mathcal{P}^o} = \text{Max}\{h_{u_i} \mid \forall u_i \in \mathcal{P}^o\}$.*

Corollary 6.3 *If there are totally N elements in \mathcal{P}_1 and $\mathcal{P}_1 \Rightarrow_p \mathcal{P}_1^o$, then a speedup is achieved as*

$$S_{ABCOM} = \frac{N}{H_{\mathcal{P}^o}}. \quad (6.2)$$

With these three corollaries, it is not difficult to prove an interesting fact stated in Theorem 6.3 that shows the effectiveness of our approach in parallelising solutions.

Theorem 6.3 *If $\forall CDOAG_{u_i} \in \mathcal{P}_1$ and $\forall CDOAG_{u_j} \in \mathcal{P}_2$ are all completely independent, and $\mathcal{P}_1 \Rightarrow_p \mathcal{P}_1^o$ and $\mathcal{P}_2 \Rightarrow_p \mathcal{P}_2^o$, then the difference in performance between \mathcal{P}_1^o and \mathcal{P}_2^o is*

$$\xi_{\mathcal{P}_1^o - \mathcal{P}_2^o} = | \text{Max}\{h_{u_i} \mid \forall CDOAG_{u_i} \in \mathcal{P}_1^o\} - \text{Max}\{h_{u_j} \mid \forall CDOAG_{u_j} \in \mathcal{P}_2^o\} |$$

Theorem 6.3 shows that the difference in the critical paths between two optimised solutions to the same problem is equal to the difference in the depths between two *CDOAGs* that are the deepest in the two optimised solutions respectively.

Remark

1) In Theorem 6.3 the concept of $\xi_{\mathcal{P}_1^o - \mathcal{P}_2^o}$ is only suitable to demonstrate that in a general case any program or representation of solution can be parallelised. 2) Only the logical steps of computation involved in critical paths are considered and the optimisation is based on the ABCOM machine. This does not tell the real difference of computation costs of two solutions. 3) It should also be noted that these two solutions are represented at the same granularity level.

To ascertain the real cost benefit in physical implementation of two given solutions, further comparisons between the two solutions in the number and sizes of *CDOAGs* required to compute the same output should be conducted. In this thesis, no further discussion on this comparison is provided.

An important application of Theorem 6.3 is to compare parallel properties of two algorithms that express the same problem in different mathematical or conceptual methods. This is achieved using their sequential representation instead of their subjective parallel implementation where parallelism has not been exploited.

Fig. 6.5 and Fig. 6.6 show the optimised solutions of Examples 5 and 6. The transformed solution of Gaussian Elimination in Appendix A.1 can also be optimised

$$\begin{array}{ll}
 u_1 : (=, \{0\}, \{s\}, 1) & u_5 : (=, \{0\}, \{s_1\}, 1) \\
 u_2 : (\times, \{a_{21}, b_1\}, \{v_1\}, 1) & u_6 : (\times, \{a_{31}, b_1\}, \{v_2\}, 1) \\
 u_3 : (+, \{s, v_1\}, \{s\}, 2) & u_7 : (+, \{s_1, v_2\}, \{s_1\}, 2) \\
 u_4 : (-, \{b_2, s\}, \{b_2\}, 3) & u_8 : (-, \{b_3, s_1\}, \{b_3\}, 3) \\
 & u_9 : (\times, \{a_{32}, b_2\}, \{v_3\}, 4) \\
 & u_{10} : (+, \{s_1, v_3\}, \{s_1\}, 5) \\
 & u_{11} : (-, \{b_3, s_1\}, \{b_3\}, 6) \\
 \\
 u_{12} : (=, \{0\}, \{s_2\}, 1) & u_{22} : (=, \{0\}, \{s_3\}, 1) \\
 u_{13} : (\times, \{a_{41}, b_1\}, \{v_4\}, 1) & \vdots \\
 u_{14} : (+, \{s_2, v_4\}, \{s_2\}, 2) & \vdots \\
 u_{15} : (-, \{b_4, s_2\}, \{b_4\}, 3) & \vdots \\
 u_{16} : (\times, \{a_{42}, b_2\}, \{v_5\}, 4) & \vdots \\
 u_{17} : (+, \{s_2, v_5\}, \{s_2\}, 5) & \\
 u_{18} : (-, \{b_4, s_2\}, \{b_4\}, 6) & \\
 u_{19} : (\times, \{a_{43}, b_3\}, \{v_6\}, 7) & \\
 u_{20} : (+, \{s_2, v_6\}, \{s_2\}, 8) & \\
 u_{21} : (-, \{b_4, s_2\}, \{b_4\}, 9) &
 \end{array}$$

Figure 6.6: The optimised solution of Example 6

superblocks).

If there is no direct or indirect dataflow relation between two loops, these two superblocks can be executed in parallel. However, if there exist dataflow relations between the two loops, then the respective *CDOAGs* of the two superblocks can be merged. When the *CDOAGs* are merged the value of T_{u_i} , which we assumed is modified to reflect the successive execution orders. Such a modification achieves dataflow computation globally. This can be done using a similar approach used by the Stanford SUIF compiler [Hea93],[HMA95] for interprocedural parallelisation analysis.

6.3.3 Observation of nondeterministic computation

The difficulties in computation inference caused by conditional statements in a loop make finding the objective parallelism impossible since there are the following reasons:

- There is a nondeterministic computational logic for the problem.
- Because of the above reason, the set of operations being performed is unfixed. Thus, the data manipulation which would be actually performed by the operations is unknown though there is a defined data domain.

However, the computational tuple-space obtained from trace-generation-based transformation provides other opportunities for one to study parallel properties that are exhibited in such a space. The basic idea is to use the concept of *speculative parallelism*, often associated with logic programming but also significant in (for example) parallel algorithms for heuristic search (e.g. parallel alpha-beta search on game tree [MC82]).

In Example 8 (*Sorting*), the elements ($\{u_1, u_4, u_7, \dots\}$) that perform condition tests are definitely executed. But the elements ($\{u_2, u_5, u_8, \dots\}$) presented in Fig. 5.9 are conditionally performed; the same operation may be repeated by different elements (e.g., the exchange of *threesort* between a_3 and a_4 might be computed by u_8, u_{23}, \dots). Thus, the parallelism in this problem can only be found among the elements in the same row in Fig. 5.9. We also eliminate the possibility to perform the elements that require the same data object as input (for instance, a_2 is used as input of both u_2 and u_5) in parallel. In this problem, thus, we can only parallelise the elements that are in

the same row and have no overlapping input. This result is consistent with the logical parallelism of *Sorting* presented in [BM93].

6.4 Summary

An initial solution in ABCOM transformed from a source code performs computation in exactly same manner as a programmer has designed (in both execution sequence and data manipulation). All subjective control features of design are preserved in this solution. Using data dependence testing and parallelisation techniques described in this chapter, the initial solution can be optimised to obtain a solution with objective parallelism. This optimisation makes all computations be executed in a dataflow computation fashion.

Comparing with the techniques and results of traditional data dependence tests, ABCOM data dependence detection is simpler, and yields a better result. The reason for this is ABCOM provides effective support for exploiting dataflow computation features in solutions.

the same row and have no overlapping input. This results in consistent with the parallelism.

If there is no overlap in the input, the parallelism is consistent. However, if there are dependencies between the two loops, then the respective CDC of the two loops is not consistent.

When the CDC is merged the value of T , which is known to be the number of processors, is used. An initial solution is found by a program that is designed to be executed on a single processor. This program is then modified to be executed on multiple processors. All activities control features of data are preserved in this process. Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

This optimization makes all computations be executed in a dataflow graph. This optimization makes all computations be executed in a dataflow graph.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

Using data dependence testing and parallelization techniques described in this chapter, the initial solution can be optimized to obtain a solution with order $O(N/P)$.

However, the computational space obtained from trace-generation based transformation provides other opportunities for use to study parallel properties that are exhibited in such a space. The basic idea is to use the concept of *parallelism* (rather than *parallelism*) to measure the amount of parallelism in a program. This is done by using a heuristic search algorithm for heuristic search (e.g., parallel alpha-beta search on game trees [MCM]).

In Example 2 (Sorting), the elements $\{a_1, a_2, a_3, \dots, a_n\}$ that perform parallel tests are defined. But the elements $\{a_1, a_2, a_3, \dots, a_n\}$ presented in Fig. 3.3 are not necessarily performed; the elements may be executed by different elements (e.g., the execution of element a_2 and a_3 might be completed by a_1, a_2, \dots). Thus, the parallelism in this problem can only be found using the elements in the same row in Fig. 3.3. We also eliminate the possibility to perform the elements that require the same data object as input (for instance, a_2 is used as input to a_3 and a_4) in parallel. In this manner, then, we can only parallelize the elements that are

Chapter 7

Parallel Computing Platform

One important consideration in developing ABCOM is to combine it with existing techniques and tools rather than to let it as a stand-alone tool working in an independent manner. This chapter discusses the use of ABCOM as a parallel computing platform to support parallelism analysis, speculation, profiling, scalable performance analysis, and program solution reconstruction. For this purpose we first introduce the main features of Bird-Meertens Formalism (BMF) since it will be used for abstracting parallelism and evaluating performance based on ABCOM. Then, we describe the relation between ABCOM and some main techniques required by parallel programming.

7.1 The Notation of Bird-Meertens Formalism

To support explicit approach of parallel programming, an optimised solution needs to be rewritten into a new program having particular parallel properties derived from objective parallelism. The expression of the new solution can be machine dependent or independent. Since the optimised solution is based on ABCOM, a machine-independent expression of the new program can be achieved using a suitable language. As described in Chapter 2, BMF supports a machine-independent approach to expressing parallelism. The advantages of using BMF is discussed in [Ski90]. To illustrate the power of its expression, we describe some operations of BMF on lists [Bir89], [Ski93],[Jay95].

1. Elementary operations

- **Length**

The length of a finite list is the number of elements it contains. Denote this by the operator $\#$. Thus,

$$\#[a_1, a_2, \dots, a_n] = n.$$

- **Concatenation**

Two lists can be concatenated together to form one longer list. Denote this by the operator $\#$. Thus,

$$[a_1, a_2, \dots, a_n] \# [b_1, b_2, \dots, b_m] = [a_1, a_2, \dots, a_n, b_1, b_2, \dots, b_m].$$

- **Map**

The operator $*$ applies a function to each element of a list. We have

$$f * [a_1, a_2, \dots, a_n] = [fa_1, fa_2, \dots, fa_n].$$

- **Filter**

The operator \triangleleft takes a predicate p and a list x and returns the list of elements which satisfy p . For example,

$$\text{even} \triangleleft [1, 2, 3, \dots, 10] = [2, 4, 6, 8, 10].$$

- **Prefix**

The operator \oplus , given a list of values, returns a list of prefixes of these values by applying an associative operator \oplus :

$$\oplus[a_1, a_2, \dots, a_n] = [a_1, a_1 \oplus a_2, \dots, a_1 \oplus a_2 \oplus \dots \oplus a_n].$$

- **Inits**

The operator *inits* generates all of the initial segments of its argument list:

$$\text{inits}[a_1, a_2, \dots, a_n] = [a_1, [a_1, a_2], \dots, [a_1, a_2, \dots, a_n]].$$

- Zip

The operation Υ_{\oplus} combines two lists of the same length by applying \oplus to the pair with one element from the first list argument and the other from the second:

$$[a_1, a_2, \dots, a_n] \Upsilon_{\oplus} [b_1, b_2, \dots, b_n] = [a_1 \oplus b_1, a_2 \oplus b_2, \dots, a_n \oplus b_n].$$

2. Reduction

The operations introduced above transform lists into other lists. The reduction operator to be described is more general. It can convert a list into other kinds of values. The reduction operator, written $/$, takes an operator \oplus on the left and a list x on the right. Its effect is to insert \oplus between adjacent elements of x . Thus,

$$\oplus/[a_1, a_2, \dots, a_n] = [a_1 \oplus a_2 \oplus \dots \oplus a_n].$$

Here the operator \oplus must be associative. Some simple cases of reduction are given in the following definitions:

sum: $+/$

product: $\times/$

flatten: $\# /$

min: $\downarrow /$

max: $\uparrow /$

BMF theories have been built for *bag*, *cons* lists [Bir87] and other data types (like *cat* lists, trees and arrays).

The BMF's features of parallelism abstraction can be used to develop ABCOM-based techniques for parallelism profiling and speculation. This provides a bridge between ABCOM and other related techniques.

7.2 Parallelism Profiling

In parallel computing, the parallelism is exploited at the programming stage and performance is measured after implementation. Measuring the performance is usually carried out by monitoring the execution of a particular program on a selected architecture. This measurement can help to tune the performance of implementation and use the system resources more efficiently. Unfortunately, traditional parallel programming does not provide practical means of parallelism analysis and reasoning. The techniques of relation-based parallelism inference described in Chapter 5 show how ABCOM can provide practical methods for parallelism profiling based on concepts of time, data, operations and *CDOAGs*. When an optimised solution with an objective parallelism is got in ABCOM, we can collect the parallelism profiling information for use in implementation.

7.2.1 Data parallelism profiling

Using time-based parallelism inference, a step-wise abstraction method can be introduced to abstract data parallelism from an optimised solution. In Fig. 6.6, for instance, there are two different kinds of operations that can be performed at step 1. The elements that perform these operations are $\{u_1, u_5, u_{12}, \dots\}$ for '=' and $\{u_2, u_6, u_{13}, \dots\}$ for '×'. Similarly, we can find data parallelism at step 2, 3 and so forth. Let '⊨' and '⇒' stand for 'perform' and 'produce output to' respectively. To exhibit data parallelism in this example, we use BMF as below (where the function f_1 is an assignment):

$$\begin{array}{l}
 \text{step 1.} \quad \{u_1, u_5, u_{12}, \dots\} \quad \models \\
 \qquad \qquad \qquad f_1 * [0, 0, \dots, 0] \Rightarrow [s, s_1, s_2, \dots, s_{n-1}] \\
 \qquad \qquad \qquad \{u_2, u_6, u_{13}, \dots\} \quad \models \\
 \qquad \qquad \qquad [a_{21}, a_{31}, \dots, a_{n-1,1}] \overset{Y}{\times} [b_1, b_1, \dots, b_1] \Rightarrow [v_1, v_2, v_4, \dots] \\
 \text{step 2.} \quad \{u_3, u_7, u_{14}, \dots\} \quad \models \\
 \qquad \qquad \qquad [s, s_1, s_2, \dots, s_{n-1}] \overset{Y}{+} [v_1, v_2, v_4, \dots] \Rightarrow [s, s_1, s_2, \dots, s_{n-1}] \\
 \text{step 3.} \quad \{u_4, u_8, u_{15}, \dots, \} \quad \models \\
 \qquad \qquad \qquad [b_2, b_3, \dots, b_n] \overset{Y}{-} [s, s_1, s_2, \dots, s_{n-1}] \Rightarrow [b_2, b_3, \dots, b_n] \\
 \text{step 4.} \quad \{u_9, u_{16}, \dots\} \quad \models
 \end{array}$$

$$[a_{32}, a_{42}, \dots, a_{n2}] \dot{Y}_x [b_2, b_2, \dots] \Rightarrow [v_3, v_5, \dots]$$

step 5.

⋮

⋮

We can also abstract data parallelism from the optimised solution of Gaussian Elimination. That is

$$\text{step 1. } \{u_1, u_{12}, u_{23}, u_{34}, u_{45}\} \models [a_{21}, a_{31}, a_{41}, a_{51}, a_{61}] \dot{Y}_/ [a_1, a_{11}, a_{11}, a_{11}, a_{11}] \Rightarrow [a_{21}, a_{31}, a_{41}, a_{51}, a_{61}];$$

$$\text{step 2. } \{u_2, u_4, u_6, u_8, u_{10}\} \models [a_{12}, a_{13}, a_{14}, a_{15}, a_{16}] \dot{Y}_x [a_{21,21}, a_{21}, a_{21}, a_{21}] \Rightarrow [v_1, v_2, v_3, v_4, v_5]$$

$$\{u_{13}, u_{15}, u_{17}, u_{19}, u_{21}\} \models [a_{12}, a_{13}, a_{14}, a_{15}, a_{16}] \dot{Y}_x [a_{31,31}, a_{31}, a_{31}, a_{31}] \Rightarrow [v_6, v_7, v_8, v_9, v_{10}]$$

$$\{u_{24}, u_{26}, u_{28}, u_{30}, u_{32}\} \models [a_{12}, a_{13}, a_{14}, a_{15}, a_{16}] \dot{Y}_x [a_{41,41}, a_{41}, a_{41}, a_{41}] \Rightarrow [v_{11}, v_{12}, v_{13}, v_{14}, v_{15}]$$

.....

$$\text{step 3. } \{u_3, u_5, u_7, u_9, u_{11}\} \models [a_{22}, a_{23}, a_{24}, a_{25}, a_{26}] \dot{Y}_- [v_1, v_2, v_3, v_4, v_5] \Rightarrow [a_{22}, a_{23}, a_{24}, a_{25}, a_{26}]$$

$$\{u_{14}, u_{16}, u_{18}, u_{20}, u_{22}\} \models [a_{32}, a_{33}, a_{34}, a_{35}, a_{36}] \dot{Y}_- [v_6, v_7, v_8, v_9, v_{10}] \Rightarrow [a_{32}, a_{33}, a_{34}, a_{35}, a_{36}]$$

.....

$$\text{step 4. } \{u_{56}, u_{65}, u_{74}, u_{83}\} \models [a_{32}, a_{42}, a_{52}, a_{62}] \dot{Y}_/ [a_{22}, a_{22}, a_{22}, a_{22}] \Rightarrow [a_{32}, a_{42}, a_{52}, a_{62}]$$

.....

.....

(Here $V = \{v_1, v_2, v_3, \dots, v_n\}$)

7.2.2 Control parallelism profiling

So far we discussed only instruction-level parallelism or data parallelism at a fine-grain level. Nevertheless, control parallelism has to be dealt with at a coarse-grain level. To profile control parallelism in ABCOM, we consider the following aspects.

1. The control parallelism exists between any two independent superblocks of a solution or between any two independent *CDOAGs* in a superblock. This is *inherent* control parallelism. The *CDOAG*-based parallelisation method can also be used to parallelise two superblocks that have *memory-based* data dependence.
2. If there is a superblock of which all *CDOAGs* merge into one *CDOAG*, and it is big enough to be divided into smaller pieces (groups of sub-*CDOAGs*), then control parallelism arises. The division (or partition) is subjective and can be done in different ways. This becomes an implementation issue.
3. Theoretically, all computation relations between superblocks (designed in an algorithm) are due to dataflow relations (though they can be implemented using different methods). The division (or partition) of a problem in an implementation makes control parallelism possible. The communication is required when the decision of the partition is made in the association with a particular architecture.

7.3 Computation Pattern Testing

When a solution is transformed into ABCOM, the spatial structure of a problem can be recovered by converting a cyclic structure (a loop) into a number of acyclic structures (*CDOAGs*) to reveal parallelism inherent in the problem. This kind of parallelism revelation can help a programmer or a compiler to reach a suitable solution based on a specific architecture. To achieve this goal, solution reconstruction needs to be carried out to map the optimised solution into a specific architecture. An important technique for reconstruction and derivation is by using the computation patterns.

Because optimised solutions are expressed at a fine-grained level, to program these computation elements based on a target architecture, we need to express them at a medium or coarse-grained level. In many instances certain operations occur repeatedly in a regular form with different input and output data, and may be executed in a partial order or in parallel. Such a regular computation form is called a *computation pattern*.

A computation pattern (simply called *pattern*) has a set of related operations organised in a certain partial order for execution using a number of data objects as input,

output and working variables. The size of a pattern can be prefixed or left open.

In ABCOM a computation pattern is associated with a number of subsets of elements. These subsets have the same number of elements. Their computation features are graphically represented by *DOAGs* or *CDOAGs*. It is possible that two sets of elements (which perform the same operations or the same pattern) have different shaped *CDOAG* representation, especially when they contain *commutative* operations. For instance, consider the following set of elements:

$$\begin{aligned}
 u_1 &: (+, \{a_1, b_1\}, \{v_1\}, 1) & u_7 &: (\times, \{b_1, d_1\}, \{v_7\}, 7) \\
 u_2 &: (\times, \{c_1, d_1\}, \{v_2\}, 2) & u_8 &: (+, \{v_6, v_7\}, \{v_8\}, 8) \\
 u_3 &: (-, \{v_1, v_2\}, \{v_3\}, 3) & u_9 &: (\times, \{v_8, 2\}, \{v_9\}, 9) \\
 u_4 &: (\times, \{I_1, J_1\}, \{v_4\}, 4) & u_{10} &: (\times, \{v_5, v_9\}, \{v_{10}\}, 10) \\
 u_5 &: (/ , \{v_3, v_4\}, \{v_5\}, 5) & u_{11} &: (+, \{v_{10}, 100\}, \{Y_1\}, 11) \\
 u_6 &: (-, \{a_1, 1_1\}, \{v_6\}, 6)
 \end{aligned}$$

These elements can be graphically represented in different shapes of *CDOAG*. Two different *CDOAGs* containing these elements are shown in Fig. 7.1.

To detect whether the two subsets of elements carry out the identical computation, we need to check not only operations involved but also the computational logic of the operations in terms of the definition of computation patterns.

7.3.1 Normalising *CDOAGs*

To abstract a number of operations into a macro computation pattern, we define the pattern as a special operation \otimes . A macro operation can be abstracted by the grammar given in Section 4.1.1.

Using the pattern grammar, the operations of *CDOAG*_{u₁₁} 1 in a) of Fig. 7.1 can be abstracted into a *macro operation* \otimes_1 , here

$$\otimes_1 := +(\times(/(-(+, \times), \times), \times(+(-, \times), \odot), \odot).$$

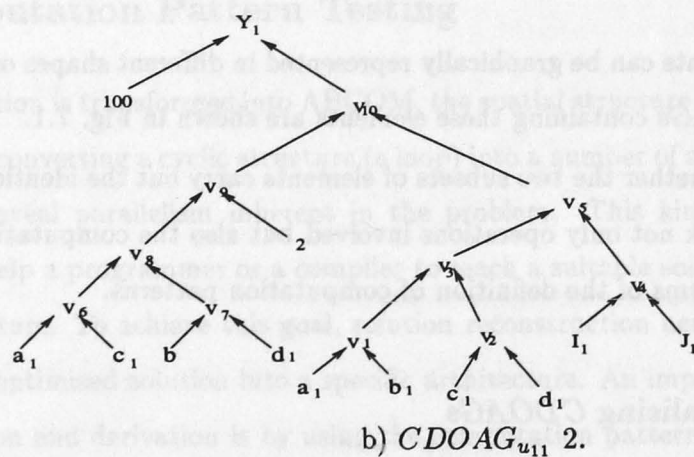
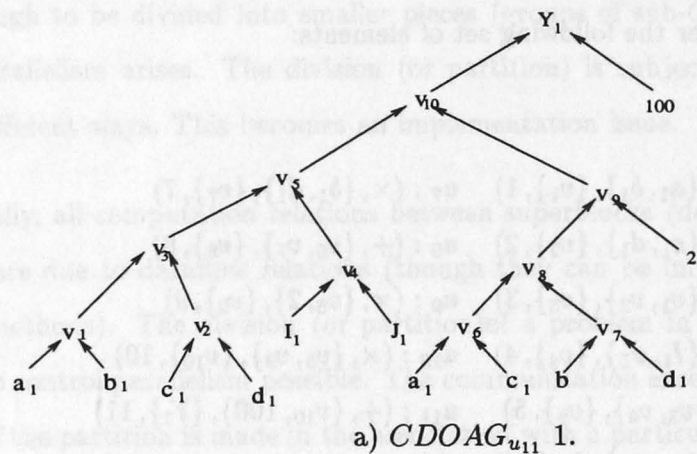


Figure 7.1: Two different $CDOAGs$ of the same set of elements.

Precedence	1	2	3	4	5	...	n
Operation	\odot	\times	$/$	$+$	$-$...	

Table 7.1: Operator precedence for pattern normal forms

Similarly, for the $CDOAG_{u_{11}2}$ we get

$$\otimes_2 := +(\odot, \times(\times(+(-, \times), \odot), /(-(+, \times), \times))).$$

To check whether two $CDOAG$ representations (or sets of elements) correspond to an identical computation pattern, the concept of *normal form* of a $CDOAG$ is useful. For this purpose we define a precedence among the all operators as shown in Table 7.1. We assume that the operator ' \odot ' has the highest precedence $p_{\odot} = 1$.

Operator precedence guides the construction of $CDOAG$ s so that those $CDOAG$ s having the same pattern can be constructed consistently. The basic idea here is that if $CDOAG_{u_i}$ has a commutative operation op_{u_i} and has two subgraphs $CDOAG_{u_j}$ and $CDOAG_{u_k}$ for $in_{u_i} \subset (out_{u_j} \cup out_{u_k})$ and $p_{op_{u_j}} < p_{op_{u_k}}$, then let $CDOAG_{u_j}$ be the left subgraph and $CDOAG_{u_k}$ the right one; if $p_{op_{u_j}} = p_{op_{u_k}}$, then check the operations of the elements at the next lower level in the subgraphs until difference is found. If no difference is found, it means the operations contained in this $CDOAG$ are commutative, and can be optimised into a unique representation.

Definition 7.1 For a given $CDOAG_{u_i}$ if its all sub- $CDOAG$ s are represented in terms of the precedence of operators at each level of the graph, then $CDOAG_{u_i}$ is represented in a normal form.

This definition can be used to construct $CDOAG$ s (or abstract pattern operations) in the normal form or to normalise a $CDOAG$. Using this method, \otimes_1 and \otimes_2 are normalised as below:

$$\otimes_{u_i} := +(\odot, \times(\times(\odot, +(\times, -)), /(-(+, \times), \times)))$$

and its corresponding $CDOAG$ is shown in Fig. 7.2.

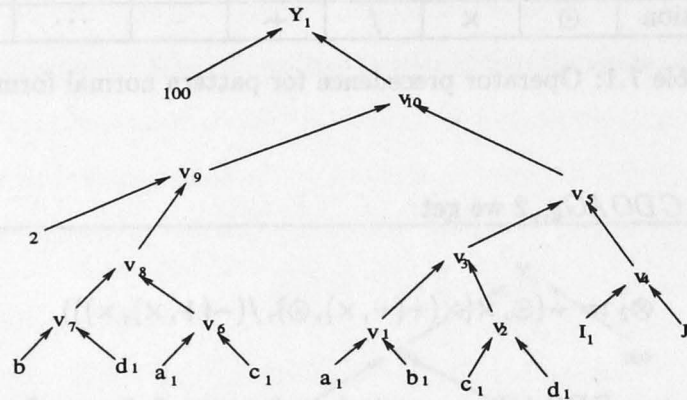


Figure 7.2: The normal form of $CDOAG_{u_{11}1}$ and $CDOAG_{u_{11}2}$.

Lemma 7.1 *If there are two subsets of elements and their $CDOAG$ s have identical normal forms, then they have the same computation pattern and the two $CDOAG$ s are isomorphic.*

7.3.2 $CDOAG$ structure optimisation

The optimisation approach described in the last chapter reveals parallelism without changing the structure of the $CDOAG$. The depth of a $CDOAG$ limits optimisation. However, further optimisation is possible due to the specific nature of operations in the $CDOAG$. For example, a sequential 'Sum' computation

for $i = 1$ to n

$$Sum = Sum + a(i)$$

can be represented as a $CDOAG$. If there are n data elements, the depth $h_{CDOAG_{u_8}}$ is $n - 1$.

It is known that the parallel computation time of Sum with n data elements is $O(\lg n)$. As a result, an optimisation should be applied to such a $CDOAG$. As an extension of the definition of the normal form, we introduce a special case where if a $CDOAG_{u_i}$ contains $n - 1$ operations that are same, associative and commutative, then

the normal form of $CDOAG_{u_i}$ is represented when its depth equals to $\lg n$.

7.3.3 Patterns represented in a loop

The discussion above on computation patterns is mainly based on the abstraction of $CDOAGs$. There are many different ways to abstract or represent a computation pattern in a program. A loop in a source code can be directly abstracted into certain forms of patterns. They are illustrated using dataflow computation models [CBF91], [Ske91].

As pointed out earlier, each statement in a loop corresponds to a number of instances in different iterations; and the body of the loop (which can be seen as a computation pattern) is repeated until the the whole computation space is performed. Using those dataflow models, the parallelism exploited is only what exists between different statements within the pattern. This limits the amount of parallelism. However, parallelism crossing different iterations can be revealed by using our approach and expressed in BMF (as described in the last section).

The information collected from parallelism profiling shows that, based on an optimised solution, new computation patterns can be abstracted for solution reconstruction. For example, profiling information of the example in Fig. 6.6 can be expressed in the following form:

For $i = 1$ to n do

$$[a_{(i+1)i}, a_{(i+2)i}, \dots, a_{ni}] \Upsilon_{\times} [b_i, b_i, \dots, b_i] \Rightarrow [v_i, v_{i+1}, \dots, v_{n-1}]$$

$$[s_i, s_{i+1}, \dots, s_{n-1}] \Upsilon_{+} [v_i, v_{i+1}, \dots, v_{n-1}] \Rightarrow [s_i, s_{i+1}, \dots, s_{n-1}]$$

$$[b_{i+1}, b_{i+2}, \dots, b_n] \Upsilon_{-} [s_i, s_{i+1}, \dots, s_{n-1}] \Rightarrow [b_{i+1}, b_{i+2}, \dots, b_n]$$

7.4 Size-Based Parallelism Speculation

We mentioned in Chapter 3 that for a loop with a large number of iterations, ABCOM code will not be generated for the whole iteration space by trace generation. To prove that it is possible to use a suitable and smaller sized iteration space instead

the real one for parallelism analysis, we must ensure that the parallelism revealed from a smaller size can be used to speculate the parallelism for a larger size problem.

A program processes certain data structures (such as arrays, tables and lists). As discussed in Chapter 3, the set of operations contained in a loop is iteratively executed under certain control mechanisms in the loop. The loop is classified into two types depending upon the relation between the data domain and the iteration:

(i) In the first type, each iteration processes exactly the same data sets of both input and output; i.e, *value-control* iteration. An important feature of this loop is that the iteration-control variable is not referred to as the index of any data object processed in the loop. In other words, the number of iterations is not associated with the size of data domain.

(ii) In the second type, i.e, *size-control* iteration, each iteration of loop deals with different subsets of the data domain. The iteration-control variable is related to an index of data being processed in the current iteration. This means the size of iteration space depends on the size of data.

The parallelism revealed in the first type corresponds to *pipeline* computations since each iteration requires the result of the previous one. In the second type, parallelism is directly proportional to the size of computation space where the size of data is related to the number of iterations. Consider a loop as a given problem; a general method to speculate computation features of parallelism (when the loop bound increases) is introduced in this section.

To study the relation between parallelism and size of computation space, consider a single loop A described by

X : the total number of the iterations;

x : x th iteration;

Q : the number of operations performed in each iteration of a loop;

$y(x)$: the critical path of the x th iteration;

$P_{s_i}(x)$: the total number of operations performed at the timestep $i = 1, \dots, y(x)$ when performing the x th iteration.

After optimising A using the approach described in Chapter 5, we have

$$\text{step 1: } Q = \sum_1^{y_1} P_{s_i}(1);$$

$$\text{step 2: } 2Q = \sum_1^{y_2} P_{s_i}(2);$$

$$\vdots$$

$$\vdots$$

Thus, for $x = k - 1$ and $x = k$ we obtain

$$(k - 1)Q = \sum_1^{y(k-1)} P_{s_i}(k - 1)$$

and

$$kQ = \sum_1^{y(k)} P_{s_i}(k).$$

When the number of the iteration increases by one, the increment of computation (denoted as ΔQ) is equal to Q ; that is

$$\begin{aligned} \Delta Q &= Q \\ &= \sum_1^{y(k-1)} (P_{s_i}(k) - P_{s_i}(k - 1)) + \sum_{y(k-1)+1}^{y(k)} P_{s_j}(k) \\ &= \Delta Q_p + \Delta Q_s \end{aligned}$$

where $\Delta Q_p = \sum_1^{y(k-1)} (P_{s_i}(k) - P_{s_i}(k - 1))$ and $\Delta Q_s = \sum_{y(k-1)+1}^{y(k)} P_{s_j}(k)$. It shows that ΔQ is divided into two parts, namely, *sequential increment* ΔQ_s and *parallel increment* ΔQ_p . To perform ΔQ_p with unlimited processors, due to parallelism, there is no need of additional computation time. But the *sequential increment* ΔQ_s does not require more processors but requires additional time:

$$\Delta y(i) = y(i) - y(i - 1).$$

For a single loop ΔQ is fixed for $x = 1, 2, \dots, X$; the ratios of $\Delta Q_p/\Delta Q$ and $\Delta Q_s/\Delta Q$ are determined by the dataflow relations that exist cross iterations. If there is no dataflow relation (that is $\Delta y(i) = 0$), then $\Delta Q_s = 0$. All iterations can be

parallelised. Note that in ΔQ_s , it is possible that there may be certain parallelism.

Consider a general situation of a nested loop having the following form:

```

for  $l = 1$  to  $X_1$  do
  For  $h = 1$  to  $X_2$  do
    :
    the body of the loop
    :
  end
end

```

X_1, X_2 : the bounds of the inner and outer loops;
 and $y(x_1, x_2)$: the critical path when $l = x_1$ and $h = x_2$;
 $P_{s_i}(x_1, x_2)$: the number of operations performed when $l = x_1$ and
 $h = x_2$.

When the iterative control variable of the internal loop increases, the computation increment is the same as in a single loop. Using a similar approach, we can express the change caused by the iterative control variable of the outer loop by

$$\begin{aligned}
 \Delta Q &= X_2 \times Q \\
 &= \sum_1^{y(k-1, X_2)} (P_{s_i}(k, X_2) - P_{s_i}(k-1, X_2)) + \sum_{y(k-1, X_2)+1}^{y(k, X_2)} P_{s_j}(k, X_2) \\
 &= \Delta Q_p + \Delta Q_s
 \end{aligned}$$

Unlike a single loop, a nested loop has a ΔQ that can be either fixed or left open when the iteration number of the outer loop increases. This depends on whether the defined range of h is related to l . If the range of h is defined as a function of l , then ΔQ is left open; otherwise it is prefixed. Fig. 7.3 shows three cases when $X_2 = M$ (ΔQ is prefixed); $X_2 = l$ and for $h = l$ to M (ΔQ is left open). In Fig. 6.3, for example, there are three *CDOAGs* that correspond to three iterations. It is seen that there ΔQ is left open due to ΔQ_p is changed in each iteration though ΔQ_s is fixed.

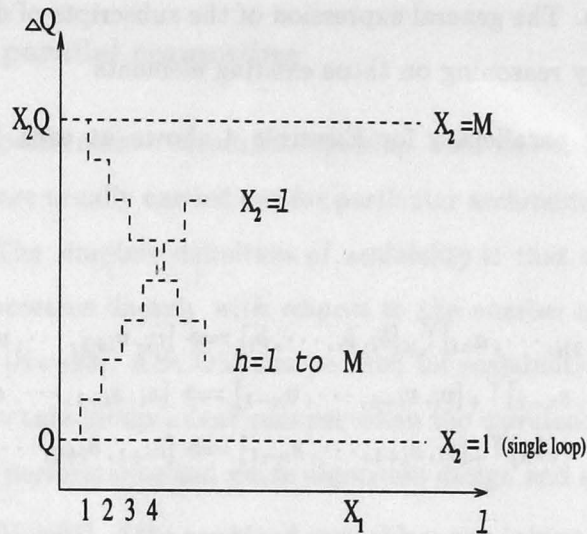


Figure 7.3: The iteration increment in a nested loop

The expression of ΔQ described above can be further generalised:

$$\begin{aligned} \Delta Q &= X_2 X_3 \cdots X_m Q \\ &= \sum_{y(k-1, X_2, \dots, X_m)}^{y(k, X_2, \dots, X_m)} (P_{s_i}(k, X_2, \dots, X_m) - P_{s_i}(k-1, \dots, X_m)) \\ &\quad + \sum_{y(k-1, X_2, \dots, X_m)+1}^{y(k, X_2, \dots, X_m)} P_{s_j}(k-1, \dots, X_m) \end{aligned}$$

Thus, no matter what kind of loop, the cost of increment in computation can be divided into two parts (ΔQ_s and ΔQ_p) when the iteration space increases. The purpose in distinguishing these two parts of computation is twofold. First, we need to speculate parallelism (parallel part) when the size of computation space increases. Secondly, if the increment of computation in the sequential part is consistently proportional to the size of the computation space, then abstracting the computation pattern that is repeated sequentially can help reconstruct a new solution (as discussed in Section 7.3.3).

In the parallel part, parallelism speculation is carried out based on the BMF representation of the optimised solution. For each vector-wise operation, the sizes of vectors

involved in the operation can be determined accordingly in terms of the bounds of the loop control variables. The general expression of the subscripts of data elements in the vectors is derivable by reasoning on those existing elements.

For example, the parallelism for Example 4 shown at each logical step can be speculated as below:

$$\begin{aligned} [a_{(i+1)i}, a_{(i+2)i}, \dots, a_{ni}] \Upsilon_{\times} [b_i, b_i, \dots, b_i] &\Rightarrow [v_i, v_{i+1}, \dots, v_{n-1}] \\ [s_i, s_{i+1}, \dots, s_{n-1}] \Upsilon_{+} [v_i, v_{i+1}, \dots, v_{n-1}] &\Rightarrow [s_i, s_{i+1}, \dots, s_{n-1}] \\ [b_{i+1}, b_{i+2}, \dots, b_n] \Upsilon_{-} [s_i, s_{i+1}, \dots, s_{n-1}] &\Rightarrow [b_{i+1}, b_{i+2}, \dots, b_n] \end{aligned}$$

In Example 5 (Gaussian Elimination) we can also speculate upon parallelism and construct a solution as follows (here we change working variables from a vector into a array):

For $i = 1$ to $n - 1$ do (in sequential)

$$[a_{i+1,i}, a_{i+2,i}, \dots, a_{ni}] \Upsilon_{/} [a_{ii}, a_{ii}, \dots, a_{ii}] \Rightarrow [a_{i+1,i}, a_{i+2,i}, \dots, a_{ni}]$$

For $j = i$ to $n - 1$ do in parallel

$$\begin{aligned} [a_{i,i+1}, a_{i,j+2}, \dots, a_{in}] \Upsilon_{\times} [a_{j+1,i}, a_{j+1,i}, \dots, a_{j+1,i}] &\Rightarrow \\ [v_{j+1,i+1}, v_{j+1,i+2}, \dots, v_{j+1,n}] & \end{aligned}$$

For $k = i$ to $n - 1$ do in parallel

$$\begin{aligned} [a_{k+1,i+1}, a_{k+1,i+2}, \dots, a_{k+1,n}] \Upsilon_{-} [v_{k+1,i+1}, v_{k+1,i+2}, \dots, v_{k+1,n}] &\Rightarrow \\ [a_{k+1,i+1}, a_{k+1,i+2}, \dots, a_{k+1,n}] & \end{aligned}$$

Using computation pattern tests and parallelism speculation, we can express a solution without any restriction on the initial size of the problem used (for trace generation). This shows that a manageable size of tuple space generated from a source code can be used to reveal parallelism in a real problem.

7.5 Scalable Performance Analysis

7.5.1 Scalable parallel computing

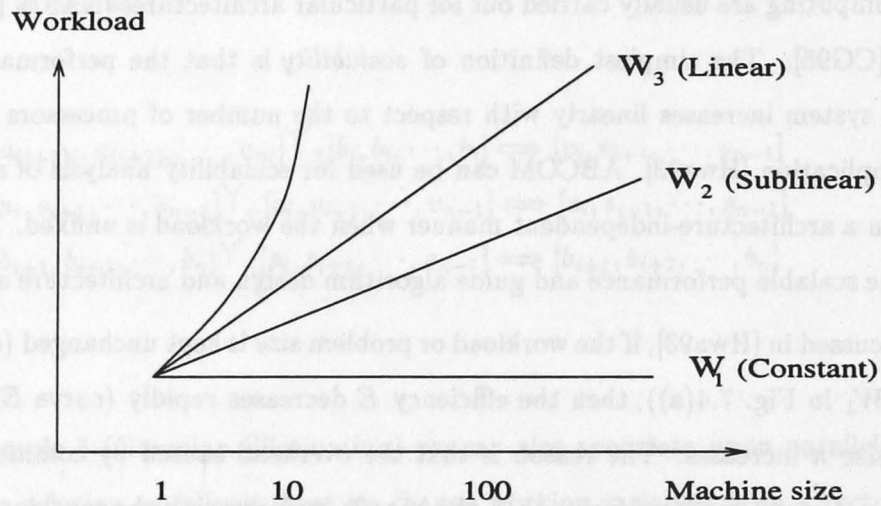
The studies of performance measures, speedup laws and scalability principles of parallel computing are usually carried out for particular architectures [NA91], [Hwa93], [Lew94], [CG95]. The simplest definition of *scalability* is that the performance of a computer system increases linearly with respect to the number of processors used for a given application [Hwa93]. ABCOM can be used for scalability analysis of solving a problem in a architecture-independent manner when the workload is unfixed. This can predict the scalable performance and guide algorithm design and architecture selection.

As discussed in [Hwa93], if the workload or problem size is kept unchanged (as shown by curve W_1 in Fig. 7.4(a)), then the efficiency E decreases rapidly (curve E_1) as the machine size n increases. The reason is that the overhead caused by communication between processors increases faster than the benefit by increasing the machine size. To maintain the efficiency at a desired level, scalability requires that both the machine size and the problem size increase proportionally. Such a system is known as a *scalable computer for solving scaled problems*. As shown in Fig. 7.4 (a), the ideal situation is to keep both the machine size and the problem size increasing linearly (curve W_3 in Fig. 7.4(a)). If the linear curve is not achievable, people will try to obtain a sublinear scalability as close to linearity as possible (as illustrated by curve W_2 in Fig. 7.4(a)).

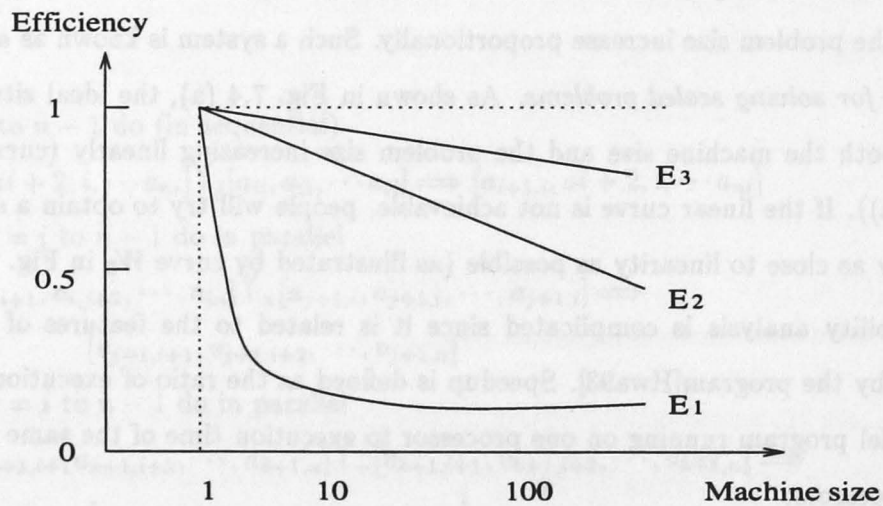
Scalability analysis is complicated since it is related to the features of speedup achieved by the program [Hwa93]. Speedup is defined as the ratio of execution time of the parallel program running on one processor to execution time of the same program on N processors:

$$\text{Speedup} = \frac{T_1}{T_N} \quad (7.1)$$

where T_1 is the execution time of the program running on 1 processor and T_N is the execution time of the same program running on N processors. In Amdahl's law (1967) it is assumed the time to run a parallel program on N processors depends on the *fraction* of program, α , that is inherently serial, and the remaining fraction $(1 - \alpha)$



(a) Four workload growth patterns



(b) Corresponding efficiency curves

Figure 7.4: Scalable computing

that is inherently parallel. That is, $T_N = \alpha T_1 + (T_1(1 - \alpha))/N$. Substituting into the formula for speedup, we get

$$S_A = \frac{N}{\alpha N + (1 - \alpha)}.$$

According to Amdahl's law, it is found that $S \rightarrow 1/\alpha$ as $N \rightarrow \infty$. In other words, under the above assumption, the best speedup one can expect is upper-bounded by $1/\alpha$ regardless of how many processors are employed. The interpretation of Amdahl's law is that, given a prefixed workload, the speedup will not improve much if the number of processors is increased.

Using the expression 7.1, we discuss the situation where the workload is *left open*. If let $N \rightarrow \infty$, it means there is an unlimited number of processors. It would be too pessimistic to use Amdahl's law in many cases if we assume α a constant when workload increases. The reason is because α is likely to be a function of the workload (size of the problem). In the last section we have shown that the parallelism of a program is determined by the size of computation space, or, the size of data domain (especially for data parallelism).

In 1988 John Gustafson and Ed Barsis proposed a *fixed-time* concept which led to a scaled speedup model. In the Gustafson-Barsis equation assume the time to compute data-parallel problem using N processors is normalised to unity, e.g. $T_N = 1$, then accordingly $T_1 = \alpha + (1 - \alpha)N$. Substituting into the expression 7.1, we get

$$S_{GB} = T_1 = \alpha + (1 - \alpha)N.$$

Our discussion on parallelism speculation shows that the increment of computation cost consists of two parts. Performing ΔQ_s requires additional time. Therefore, the assumption of Gustafson-Barsis is only one of the possible situations in which there is no increment in the sequential part, e.g. $\Delta Q_s = 0$ or $\Delta y = 0$ for any increment of the workload. This assumption is too optimistic.

Under the assumptions of Amdahl's law and the Gustafson-Barsis equation, if we let $N \rightarrow \infty$, then both S_A and S_{GB} should reflect the relationship of a scaled workload and the speedup of the program. It has often been observed that the problem size is the most significant factor of data-parallelism. Consequently, α becomes a critical

factor in explaining the relationship. In practice, for a subjective program there are various constraints introduced in design, related to scalable performance. Therefore, it is difficult to define clearly the function α against the workload. This is the major restriction in using speedup performance laws to predict the scalable performance when the workload increases.

7.5.2 Scalability of application domain parallelism

In terms of the discussion above, we believe it is necessary to further investigate the nature of scalability in parallel computing, especially the parallel properties of a scaled problem (or application domain parallelism (ADP)). If we see a loop as a problem, it is observed that there are two different situations for the increment of workload: 1) workload (problem size) increase when data size increases; 2) workload increases when total computation cost increases, while data size remains unchanged (for example, a *value-control While--Do* loop).

Usually, in scalability analysis, the relationship between α and the size of a problem is simplified by assuming that the parallelism achieved in a program is proportional to the size of problem. This assumption is not suitable in the following cases:

- (i) The objective parallelism is not scalable when the problem size increases.
- (ii) The objective parallelism increases much faster than the subjective parallelism achieved in an implementation when the problem size increases.

In (i) it is impossible to obtain scalable performance. In (ii), however, the subjective parallelism could be improved for better performance. Hence, one should study the nature of the fraction of parallel parts of the problem in scalability analysis rather than merely using the concept of the workload. That is, the scalability of a computer system (program and architecture) should be studied for an open-workload problem after we know whether the application domain parallelism is scalable. For this purpose, we introduce a concept, called *scalability of application domain parallelism* (SADP).

Definition 7.2 *The scalability of application domain parallelism is determined by gradient of the ratio of the workload that can be computed in parallel to the total workload.*

In general, we have

$$\omega = \frac{Q_P}{Q_T} \leq 1$$

here Q_T is the total workload, and Q_P is the workload that can be computed in parallel. Since ω is a function of Q_T and Q_P which are functions of problem size (the bound of loop iteration variable i), the gradient of ω against i can be obtained by using the derivative of ω , denoted as

$$\tau = \omega'.$$

To study SADP, we check three typical cases of τ . In the first case, if $\tau = 0$ since $\omega = \omega_0$ (ω_0 is a constant), then SADP is *linear*. In the second case, assume $\tau > 0$ when Q_T increases, then ω is monotonically increasing ($\omega \rightarrow 1$) as $Q_T \rightarrow \infty$. It means the application domain parallelism has a *superlinear* scalability when the problem size increases. In the third case, if $\tau < 0$ when Q_T increases, then ω is monotonically decreasing ($\omega \rightarrow 0$) as $Q_T \rightarrow \infty$. It shows that there is a *sublinear* scalability of ADP (SADP is poor). In other words, it is impossible to get good scalable performance for a problem if the workload increases.

Using the concept of SADP, we revise the scalability metrics described in [Hwa93] to suit the open workload. As shown in Fig. 7.5, the scalability analysis of architectures and algorithms should be based on SADP of a problem rather than the problem size. Precisely, SADP of a problem should be examined when we study the scalability of a particular implementation if the problem size is left open. This examination benefits both parallel algorithm design and architecture selection. If function ω for a given problem could be exactly defined, then we could check whether the problem is suitable for scalable parallel computation. That is, we say a given problem is suitable for scalable parallel computing if $\tau = 0$; it is well suited if $\tau > 0$; or it is not suitable if $\tau < 0$. In practice, however, it is difficult to obtain the function ω for a given problem.

According to the definition of ω , there is a particular ω_p when a problem is expressed in a parallel program. The difference between ω and ω_p is determined by the subjective factors of the program. Without removing these factors, ω_p can only be used to study the scalability of the program. Back to our approach, we introduce certain methods to study SADP of a problem by using an optimised solution with objective parallelism.

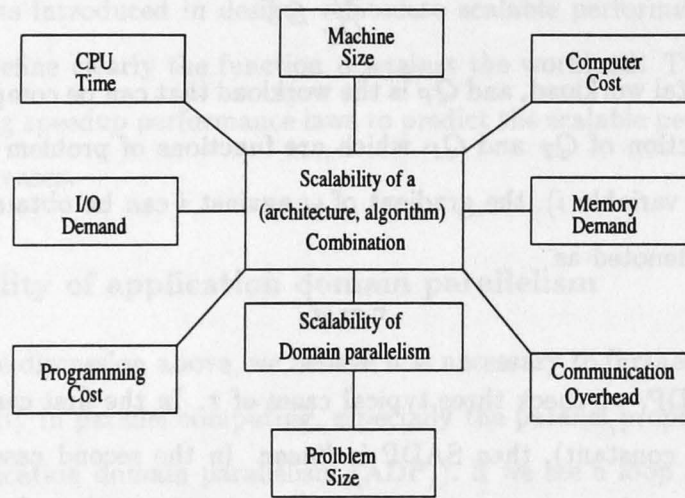


Figure 7.5: The revised scalability metrics.

In such a solution where subjective factors are free, therefore, we are able to study ω of the problem. Our discussion is still based on a superblock (loop). We use the loop bound as the parameter of the workload, which covers both situations of workload increasing. Using the discussion of parallelism speculation, first, we introduce a method to analyse SADP when the size of a problem is changed.

In terms of the definition of ω , we need to distinguish the sequential part and the parallel part of computation for a given computation workload. In ABCOM, an optimised solution with a total cost Q_T takes HP_0 logical steps. The computations involved at these steps of the critical path are inherently sequential. All other computations can be executed in parallel with no additional time required. It means the parallel computing workload is

$$Q_P = Q_T - HP_0. \quad (7.2)$$

Therefore, function ω can be expressed as

$$\omega = 1 - \frac{HP_0}{Q_T}. \quad (7.3)$$

To check SADP when the problem size (loop control variable i) changes, we see ω as a continuous function so that the derivate of the function to i can be expressed as

$$\omega' = \frac{-Q_T \frac{d(H_{P_0})}{di} + H_{P_0} \frac{d(Q_T)}{di}}{Q_T^2}. \quad (7.4)$$

Since $\tau = \omega'$, we can rewrite the expression 7.4 as

$$\tau = \frac{1 - \frac{Q_T}{H_{P_0}} \times \frac{d(H_{P_0})}{d(Q_T)}}{\frac{Q_T^2}{H_{P_0} \times \frac{d(H_{P_0})}{di}}}. \quad (7.5)$$

In a discrete form, function ω' (or τ) becomes computable if we replace $\frac{d(Q_T)}{di}$ and $\frac{d(H_{P_0})}{di}$ by $\Delta Q(i)$ and $\Delta y(i)$ respectively.

Theorem 7.1 For a given solution with i as the bound of loop, if $\frac{\Delta y(i)}{\Delta Q(i)} \leq \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, then the solution has a superlinear scalability of application domain parallelism.

Proof: We approximate $\frac{d(Q_T)}{di}$ and $\frac{d(H_{P_0})}{di}$ respectively by $\Delta Q(i)$ and $\Delta y(i)$ in a discrete form for $i = 1, 2, \dots, n$, then the formula 7.5 becomes

$$\tau = \frac{1 - \frac{\Delta y(i)}{\Delta Q(i)} \times \frac{Q_T(i)}{H_{P_0}(i)}}{Q_T^2(i)/H_{P_0}(i)\Delta Q(i)}. \quad (7.6)$$

When the problem size is i , the speedup of the optimised solution is stated as

$$S_{ABCOM}(i) = \frac{Q_T(i)}{H_{P_0}(i)}. \quad (7.7)$$

Consequently, if $\frac{\Delta y(i)}{\Delta Q(i)} < \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, then

$$1 - \frac{\Delta y(i)}{\Delta Q(i)} \times \frac{Q_T(i)}{H_{P_0}(i)} > 0,$$

namely, $\tau > 0$ holds. That is, the solution has a superlinear scalability of application domain parallelism. \square

Theorem 7.2 For a given solution with i as the bound of loop, if $\frac{\Delta y(i)}{\Delta Q(i)} = \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, then the solution has a linear scalability of application domain parallelism.

Theorem 7.3 For a given solution with i as the bound of loop, if $\frac{\Delta y(i)}{\Delta Q(i)} > \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, then the solution has a sublinear scalability of application domain parallelism.

Theorem 7.2 and Theorem 7.3 can be proved similarly.

In fact, there exists another approach to analyse scalability of application domain parallelism by directly using the concept of speedup. As discussed before, we can see speedup as a function of problem size, as expressed in the expression 7.7. Thus, the derivate of the function is

$$S'_{ABCOM} = \frac{H_{P^0} \frac{dQ_T}{di} - Q_T \frac{dH_{P^0}}{di}}{H_{P^0}^2}. \quad (7.8)$$

Here we can substitute $\frac{dH_{P^0}}{di}$ and $\frac{dQ_T}{di}$ by using $\Delta y(i)$ and $\Delta Q(i)$ in a discrete form such that S'_{ABCOM} become completely computable at points ($i = 1, 2, 3, \dots, n$). This concept can be explained as the *gradient of speedup*. If assume the gradient of speedup equal to zero, then SADP is linear when the size of problem changes. This is a special case. In fact, an increased speedup is possible for many problems when their sizes increase. Thus, better speedup should be considered for both program design and architecture selection.

Theorem 7.4 For a given solution with i as the bound of loop, if $\frac{\Delta y(i)}{\Delta Q(i)} < \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, the solution has an increasable speedup when the value of i increases.

Proof: We substitute $\frac{dH_{P^0}}{di}$ and $\frac{dQ_T}{di}$ by using $\Delta y(i)$ and $\Delta Q(i)$ in the formula 6.1. Then, we get

$$S'_{ABCOM} = \frac{H_{P^0} \Delta Q(i) - Q_T \Delta y(i)}{H_{P^0}^2}, \quad (7.9)$$

Example	$H_{P^0}(i)$	$Q_T(i)$	$\Delta y(i)$	$\Delta Q(i)$	SADP
5	$4i - 2$	$2i^2$	2	$4i - 2$	$\tau(i) > 0$
6	$3(i - 1)$	$(3(i - 1)i)/2$	3	$3(i - 1)$	$\tau(i) > 0$
7	$3(i - 1)$	$i(i - 1)(4i + 1)/6$	3	$(i - 1)(2i - 1)$	$\tau(i) > 0$

Table 7.2: SADP analysis of Example 5, 6 and 7.

or

$$S'_{ABCOM} = \frac{1 - S_{ABCOM}(i) \times \frac{\Delta y(i)}{\Delta Q(i)}}{H_{P^0}(i) \Delta Q(i)} \quad (7.10)$$

where

$$S_{ABCOM}(i) = \frac{Q_T(i)}{H_{P^0}(i)}.$$

Consequently, if $\frac{\Delta y(i)}{\Delta Q(i)} \leq \frac{1}{S_{ABCOM}(i)}$ holds for $i = 1, 2, \dots, n$, then $S'_{ABCOM} > 0$ holds at all these points such that the solution has an increasable speedup. \square

To check SADP of Example 5, 6 and 7 for $i = 3, 4, \dots, n$, we use their optimised solution illustrated in Fig. 6.5, Fig. 6.6 and Appendix A.2, and can get the general expressions of $H_{P^0}(i)$, $Q_T(i)$, $\Delta y(i)$ and $\Delta Q(i)$, and the nature of τ for these examples as shown in Table 7.2.

We have described certain methods to analyse scalability of application domain parallelism of solving a given problem when represented in the form of ABCOM. It is difficult to achieve good performance in parallel programming; but it is more difficult to develop a scalable program when the workload increases. Analysing SADP for a real world problem brings a useful knowledge of parallel properties that can be used for a scalable computer system.

The above discussion provides methods to study how the performance or speedup of solving a problem will change as the size of the problem changes. Although SADP based on an optimised solution of ABCOM may not be physically realisable since various constraints are introduced by selected architecture and implementation, it can be useful.

7.6 Other Applications

The ABCOM-based parallel computing platform is aimed to provide a foundation for integrating techniques and tools in parallel computing. In this section we briefly describe certain issues regarding to integrating ABCOM platform with a cost system of parallel programming and solution derivation.

7.6.1 Integrating with a cost system

In order to make a correct decision in program design or apply proper transformation rules in a solution derivation against a particular architecture, the calculational approach is highly desirable and becoming deservedly popular for parallel software development and program transformation [Bir89]. This approach requires more concrete methods to estimate the cost of computation. One of the important features of the approach is the provision of cost information at intermediate stages in a derivation. Skillicorn *et al* provide a comparison on existing parallel cost systems and developed a cost calculus for parallel functional programming [SC94].

It is hard to build a useful cost system for parallel computation because there are many more degrees of freedom. In general, a cost system must be provided with sufficient information regarding to the following important factors:

- Details of the structure of the program;
- The size of the problem;
- The extent to which the work to be done depends on values of the input, rather than their number and sizes;
- The way in which the program is decomposed into threads that can execute on different (virtual) processors;
- The way in which communication between threads and the synchronisation rules associated with it are arranged;
- The way in which the threads are mapped to physical processors;
- The mapping of communication actions to the target processor's interconnection;

- The extent to which the computation exhibits dynamic behaviour.

Since it is difficult to deal with all these factors together, at present, the only known way to build cost systems is to dynamically compromise certain factors [SC94].

The central problem in building a cost system is to provide the right level of abstraction. This abstraction should hide much of the underlying complexity, but be able to reveal enough for decisions about one choice of an algorithm over another. In Section 7.2, we demonstrated that the profiling information abstracted from a solution expressed in ABCOM can be represented by using the Bird-Meertens formalism. This enables us to integrate ABCOM platform with the cost system developed by Skillicorn so that those methods and results provided in [SC94] can be properly used for solution derivation or to support mapping an optimised solution into a specific architecture.

The reasons that we can integrate ABCOM with such a cost system is

- A machine-independent representation, in particular a solution with objective parallelism, can be obtained using ABCOM. Moreover, profiling information of an optimised solution can be abstracted by the Bird-Meertens formalism which is a bridge between the ABCOM platform and the cost system. Thus, all information about data structure and data manipulation are available for the cost system.
- ABCOM-based computation inference can be used by the cost system.
- The compositional property of the representation is required by the cost system. This requirement can be well satisfied by using the superblock-based strategy used in our approach.
- The Skillicorn's cost system is developed to assist program transformation or derivation. Thus, an optimised solution expressed in ABCOM can naturally be used as a source code so that the transformation or derivation can be carried out based on a background with sufficient information on application domain parallelism.

The task of integration of ABCOM with a cost system can be mainly divided into two parts. The first is to develop computation pattern testing rules or interaction mechanism for a programmer to help in identifying some special patterns such that the

profiling information can be properly abstracted as much as possible into the expression of Bird-Meertens Formalism. The second is to create a cost reference table for those recognised patterns based on different architectures that are considered to be supported by the programming platform.

7.6.2 Solution Derivation Support

Transformational programming and parallel computation are two emerging fields that may ultimately depend on each other for success. Because *ad hoc* programming for parallel machines is so hard, and because progress in software construction has lagged behind architectural advances for such machines, there is much greater need to develop parallel programming and transformational methodologies. The challenge of parallel solution derivation and program transformation is that it represents perspectives from two different communities — transformational programming and parallel computing — to discuss programming, transformational, and compiler methodologies for parallel architectures, and paradigms, techniques, and tools for parallel machine models.

A number of interesting studies on parallel program transformation are reported in [Pep93], [PPP93], [GY93], [Smi93], [Par93], [RR93], [Lan93]. We discuss here the feasibility of integrating derivation techniques of parallel programs with ABCOM.

According to the discussion in [Smi93], programs can be treated as a highly optimised composition of information about the problem being solved, algorithm paradigms, data structures, target architectures and so on. An attempt to provide automated support for program design must be based on:

- a formal model of the composition process;
- representation of problem domain knowledge;
- representation of programming knowledge.

The research on parallel algorithm derivation and program transformation is based on the idea to produce formally verified software. Therefore, derivation is usually based on a selected formal specification of a problem. The main difficulty in derivation lies in building up the problem domain theory within which the algorithm is inferred.

And it is also known that methods and tools for achieving this goal still are research topics. The techniques suggested in the literature are split into: either i) verification-oriented techniques to provide a proof that a program conforms to a specification; or ii) transformation-oriented techniques to generate an executable program from a specification by applying a series of transformations. Comparing with a specification language, there are certain advantages if an ABCOM-based solution is used as the source code for transformation.

- Unlike a solution expressed in a specification language, ABCOM-based solutions are executable in a general sense of computing. It will reduce the difficulty of transformation in dealing with execution semantics for construction of an executable program associated with a particular architecture.
- ABCOM-based solutions can be presented with objective parallelism. This provides certain assurances for achieving a good performance for a derived solution. This is important for success of transformational programming.
- ABCOM is grain-dependent. Thus, transformation strategies can be developed at different levels of representation from program structure and optimisation points of view.
- Computation inference based on ABCOM can be used to develop transformation rules. Building up connections between ABCOM and other techniques will benefit development of transformation rules and construction of a transformational programming framework. This framework can provide some interactive programming features to deal with decision making in transformation since a cost system can be combined through ABCOM.

Transformational programming can make use of ABCOM's power in revealing parallelism such that decisions in selecting parallel properties of solving a problem are based on well exploited information. From a theoretical point of view finding the optimum solution is NP-hard. In practice, however, parallel programs are written in certain styles for different types of architectures. The style of programming is mainly determined by a number of decisions made in the following respects:

- *Data or control parallelism;*
- *Partitioning;*
- *Scheduling and co-ordination;*
- *Communications;*
- *Skeletons.*

Lemma 5.1 is a general rule to identify which elements in ABCOM are currently ready for execution at each timestep. This rule can be implemented in an efficient way to support static scheduling. That is, to identify the ready elements, we need only tracing of the movement of the 'bottoms' of all independent *CDOAGs* in an optimised solution. At each logical step during the execution of the solution, only those elements that are in the 'bottoms' of the *CDOAGs* are ready for execution.

The solution derivation from ABCOM codes should be studied from all these respects. As a long term project, we need support from people working in different areas to bring those related research and development results together to form a multi-functions and integrated programming environment.

7.6.3 ABCOM-based programming paradigm

The features of ABCOM-based parallel processing and the discussion in the connections between ABCOM and other techniques demonstrate certain new approaches to improving parallel programming methodologies. The main improvement is to provide adequate support in revealing parallelism when a program is developed. Also existing techniques can use or be further developed in association with such support within a programming framework. Hence, programming can be carried out with less chance of performance failures. This improvement, as illustrated in Fig. 7.6, can be explained as follows:

- A sequential program or other executable and machine-independent specifications (for example GAMMA-based program) can be transformed into ABCOM representation as an initial version of a solution. This solution can be optimised until objective parallelism is revealed.

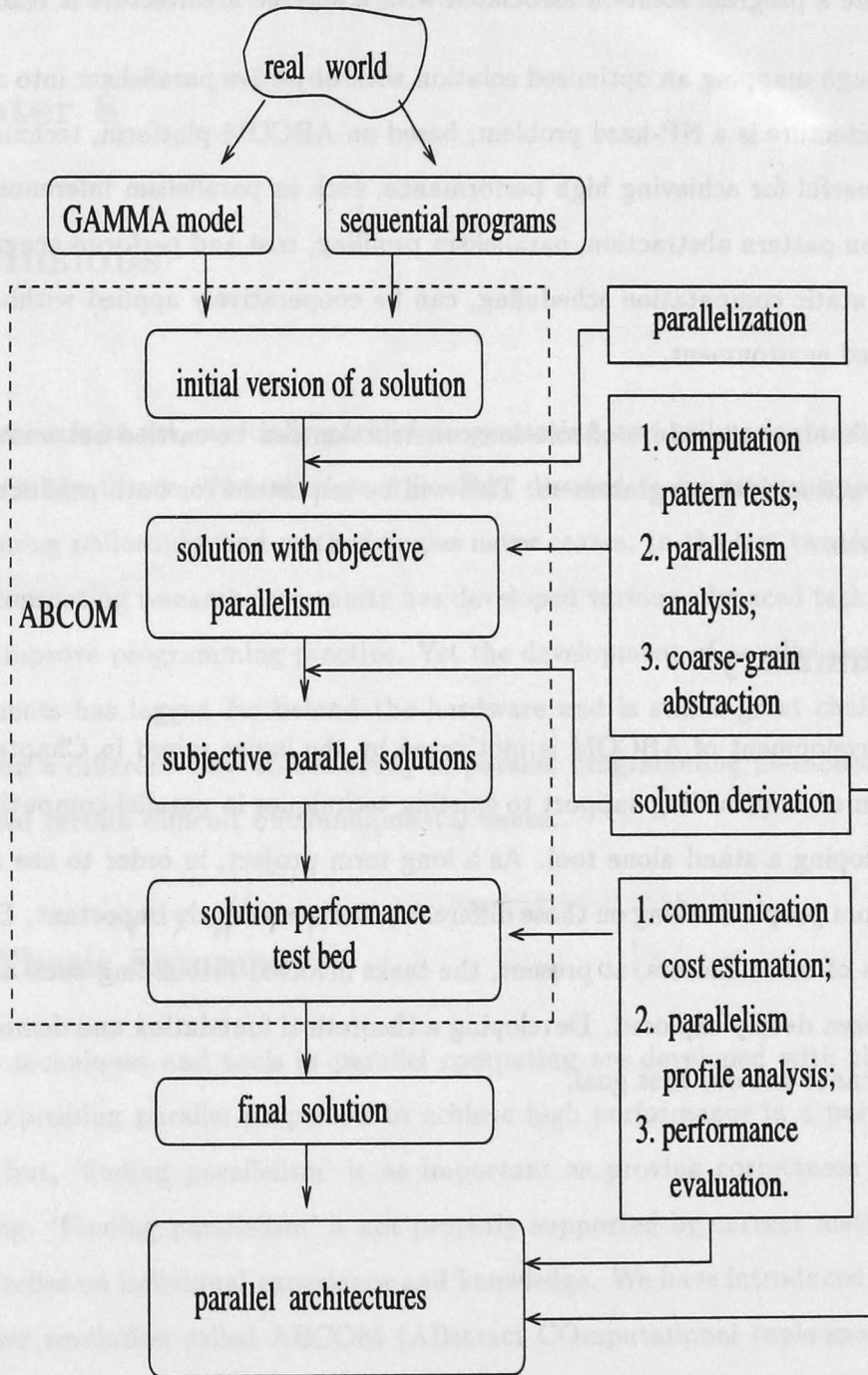


Figure 7.6: ABCOM-based parallel programming paradigm

- The objective parallelism exploitation ensures a sound information background of parallel properties of solving a problem to be available and well represented before a program solution associated with a specific architecture is reached.
- Though mapping an optimised solution with objective parallelism into a selected architecture is a NP-hard problem, based on ABCOM platform, techniques that are useful for achieving high performance, such as parallelism inference, computation pattern abstraction, parallelism profiling, cost and performance prediction and static computation scheduling, can be cooperatively applied within an integrated environment.
- A trial-error procedure of solution construction can be carried out under certain interaction with programmers. This will be important for both productivity and success.

7.7 Summary

The development of ABCOM is motivated by the issues raised in Chapter 2. Our effort is aimed at providing support to existing techniques in parallel computing rather than developing a stand alone tool. As a long term project, in order to use ABCOM, support from people working on those different areas is extremely important. Due to the limitations of many factors, at present, the tasks involved in building such a platform have not been deeply explored. Developing a theoretical foundation and demonstrating the significance are our first goal.

Chapter 8

Conclusions

Programming is an art, especially parallel programming, and will undoubtedly continue to be so in the future. Nevertheless, the effort devoted to innovative approaches to programming philosophy and methodologies never ceases. In the last two decades, the parallel computing research community has developed various advanced techniques and tools to improve programming practice. Yet the development of parallel programming environments has lagged far behind the hardware and is still a great challenge. We have taken a different view to the study of parallel programming methodologies, and have raised certain difficult but fundamental issues.

8.1 Thesis Summary

Most techniques and tools in parallel computing are developed with the primary goal of expressing parallel properties to achieve high performance in a parallel architecture; but, 'finding parallelism' is as important as proving correctness in parallel computing. 'Finding parallelism' is not properly supported by current methodologies, and still relies on individual experience and knowledge. We have introduced a model of parallelism revelation called ABCOM (ABstract COmputational tuple-space Model), and examine its properties and its power to support parallelism analysis, inference, profiling, speculation and abstraction, solution reconstruction and performance prediction.

The main contributions claimed for this thesis are summarised as below:

- This work broadens the research area of parallel computing from both the theoretical and practical points of view, by introducing concepts such as parallelism revelation models, subjective parallelism, objective parallelism, and the scalability of application domain parallelism. We advocate that the objective aspects of parallelism of a real world problem should be studied before an implementation is attempted. The main benefit of this study is to obtain a sound knowledge of the parallel properties of the problem. Such a knowledge can help making parallel programming decisions and reduce the possibility of performance failures in parallel implementation.
- A parallelism revelation model, called 'ABCOM' is introduced in this thesis. The notation and properties of ABCOM exhibit its capability as a foundation to reveal parallelism features and to support parallelism inference. The parallelism inference feature can be implemented using relational algebraic techniques on a programming database.
- ABCOM is intended to be at a level below the language level and is compatible with a variety of language styles. Thus, it will have applications in various research areas in parallel computing.
- Based on ABCOM, we have presented new approaches to detect exact data dependence and parallelise program solutions. Trace-generation based transformation strategies of ABCOM contribute to generation of an abstract computational tuple-space for a given source code. From such a tuple-space the topological (spatial), structural and temporal properties required in solving a problem can be fully recovered; an optimisation towards achieving the objective parallelism can be carried out. This optimisation is not only machine independent but also programmer-view independent. All the computations represented in an optimised solution with the objective parallelism are driven by data flows. Consequently, the difference in performance between any two optimised solutions to the same problem is given by the difference in the respective depths of the deepest *CDOAGs* of the solutions. In other words, the difference in performance between any two optimised solutions equals the logical time difference to compute the longest data

flows for individual data in these two solutions.

- The ABCOM-based approach can provide special tools for reviewing existing parallel programs to detect deficiencies with the aim of improving a given solution. Also it is possible to compare inherent parallelism in two different algorithms for the same problem before their physical and subjective parallel implementations.
- The tools and techniques developed in association with ABCOM, including parallelism inference, abstraction, speculation and so on, will be of significant assistance in making programming decisions and selecting a suitable architecture for a particular application.
- Being machine-independent, ABCOM can serve as an standard parallel abstract model that can separate hardware features of architecture from software concerns, hence it can promote software portability and scalability.
- The outcome of this research is the design of a parallel computing platform that can be eventually developed as a unified framework for parallel programming methodologies and for integrated development environments.

8.2 Limitations

Fully implementing the ABCOM-based parallel computing platform is a very large and complex task. Therefore we introduced several restrictions to comply with the resources at our disposal. In particular, we have given priority to: i) introducing ABCOM as a parallelism revelation model with the emphasis on its significant features for improving parallel programming; ii) illustrating its applications in association with different research interests and techniques in parallel computing. In this context the ABCOM-based parallel computing platform is primarily a research prototype of a programming supporting environment that contains transformation, inference, optimisation, profiling and performance prediction techniques. The results obtained using these techniques support our claim that ABCOM is useful as a parallelism revelation model and is very helpful in enhancing existing research and techniques.

At present, there are a number of deficiencies in ABCOM and one can ask many questions regarding its applications to various real world problems. This section points out some of the limitations of this work.

- Though the transformation methods of conditional statements have been described in Chapter 4, we have not shown the application of ABCOM for a problem containing uncertain control structures arising from conditional statements in its program solution. It is true that the involvement of these structures complicates computation analysis in ABCOM. Nondeterministic execution features of computation usually results, if there are conditional statements contained in a loop. As pointed out in Chapter 5, relation-based computation inference techniques are not applicable to nondeterministic computation. The ABCOM-based parallelism revelation approach may not successfully exploit objective parallelism for such problems. From the discussion concerning Example 8 in this thesis, however, it can be observed that parallelism visibility would be achievable in a certain sense, if some supporting techniques are developed. We suggest the use of heuristics and interaction with the programmer to guide nondeterministic computational analysis, inference and abstraction. However, we have not yet identified suitable heuristics.
- In this thesis, we discussed the properties and applications of ABCOM mainly based on a fine-grained representation. This has more or less limited our investigation. It is still too early to claim that a sophisticated and complete parallel programming environment has been developed; in particular, we have not yet provided adequate methods of using ABCOM to the more challenging task of mapping real world problems onto specific architectures. To broaden the applications of ABCOM, we need first to extend the investigation from the fine-grained representation to a higher-level representations — e.g., a medium-grained or coarse-grained level. We should pursue further:
 - Element-grouping strategies;
 - Higher-level inference and analysis techniques;

- Higher-level optimisation methods.

Research into these respects must be carried out in association with many traditional programming issues, such as programming style, program structure and partitioning strategies, and also certain architecture-related programming issues, including data-access mode, interconnection among processors and communication methods.

- Through the use of a fine-grained representation, the methods of abstracting parallelism described in the previous chapters are mainly based on a step-wise strategy for data parallelism. As mentioned before, the main requirements for relation-based computation inference is the existence of certain kinds of relationships among elements with respect to the concepts of data, operation or time. Consequently, these techniques should be developed in a more flexible way so that static scheduling, computation pattern and type (or shape discussed in [Jay95]) abstraction and solution derivation can use ABCOM.
- To illustrate principles of our approaches, superblock-based strategies are used in the discussion of this thesis. The synthesis of the results obtained from a number of superblocks in inference, analysis, abstraction, profiling and speculation is necessary and important for successful applications of ABCOM. Although it is not difficult to develop suitable techniques to synthesise information for problems with simple algorithmic structure, we have not explained how to build up a framework to conduct synthesis of various information, especially for large and comprehensive applications.

8.3 Future Work

The work reported in this thesis is basically an introduction to a long term research project for parallelism revelation. We believe the most fruitful approach to exploiting parallelism is to start with a few principles and to make use of them as far as possible. There are many appealing avenues to pursue for future work. We have already pointed out a number of limitations and their possible improvements in the previous section.

The work in progress includes developing computational inference and abstraction techniques based on ABCOM, especially with respect to nondeterministic computation and higher-level representation, and implementing a research prototype of a parallel computing platform in a UNIX-based environment with support of ORACLE as a programming database.

Also we need to relate ABCOM with other relevant techniques or tools. We realise that the development of ABCOM-based techniques for a higher-level representation is vital for the successful applications of our approach.

Beside the work mentioned above, we would also like to highlight two open problems:

- **Mapping — an NP-complete problem**

Though a real world problem can be optimised to reach an objective parallelism, finding an optimal solution to execute in a particular architecture is still an NP-complete problem. The ABCOM-based approach can provide assistance in building up application domain knowledge of parallel properties, but cannot optimally map this knowledge into a selected architecture. Instead of pursuing an optimal solution, in practice, people usually think a solution with a satisfactory performance as a goal of implementation. As stated in Chapter 1, ABCOM is intended to enhance the existing techniques and tools in current practice rather than replacing them. Therefore, whether ABCOM can really help to improve parallel programming methodologies is mainly determined by whether it can be successfully applied in the mapping procedure performed by either a programmer or a compiler.

- **Challenge to the future generation of compilers**

As one of the important tasks of parallelising compilers, parallelism exploitation can be carried out now in a non-traditional way. Can this approach help to design a parallelising compiler? The answer to this question is not known at this stage. But at least we have shown that it is possible to reveal objective parallelism for a given deterministic computation problem, as long as it is specified in a conventional way. Comparing the information of parallelism obtained and tasks performed by a parallelising compiler, we see that ABCOM-based ap-

proaches provide a better knowledge of parallelism inherent in the problem, but require different mapping strategies to derive or reconstruct a solution based on a particular architecture.

By developing ABCOM, we believe the application of our parallelism revelation model will ease the tasks of domain experts and programmers in parallel computing, but apply more pressure to architecture suppliers and compiler designers since it will be a significant advantage if they can demonstrate that their products are more efficient and better in performance when a domain knowledge of parallelism is available. As this domain knowledge of parallelism is machine-independent, the portability issue of parallel computing can be resolved if ABCOM representation can be processed by different compilers.

At the NATO sponsored Advanced Research Workshop on 'Software for Parallel Computation' in 1992, Kowalik and Neves pointed out that if parallel computing is to be successful, it will require an unprecedented cooperation among application developers, compiler writers, systems software professionals, and hardware architects [KN93]. However, it is observed that such a cooperation has embarrassed the designers of programming languages due to various different and even conflicting requirements from these people. Also it is realised that the field of parallel computation is going through a period of unrest: a growing rift between theory and practice suggests that more realistic models of computation are needed [FS92]. We are hopeful that the introduction of a parallelism revelation model will provide a possible avenue for further developments. The extent of success of the ABCOM model can be judged only when this model is applied in different areas of parallel computing.

... the extent of success of the ABCOM model can be judged only when the model is applied in different areas of parallel computing.

• **Challenges to the future**

As one of the important tasks of parallel computing, parallel algorithms can be carried out in a new paradigm. Can this approach help to design a parallelizing compiler? The answer to this question is not known at this stage. But at least we have shown that it is possible to model a parallel problem as a graph or a set of nodes and edges, and to transform it into a sequential problem. This is a conventional way. Comparing the information of parallel and sequential algorithms, we see that ABCOM-based approach is a better way to design a parallelizing compiler.

Appendix A

A.1 ABCOM code of Example 7.

Consider a sequential code for Gaussian Elimination (without pivoting).

For $k = 1$ to n

For $i = k + 1$ to n

$$a(i, k) = a(i, k) / a(k, k)$$

For $j = k + 1$ to n

$$a(i, j) = a(i, j) - a(k, j) \times a(i, k)$$

Let $n = 6$, a transformed code of this solution is shown as follows:

$$u_1 : (/ , \{a_{21}, a_{11}\}, \{a_{21}\}, 1)$$

$$u_2 : (\times , \{a_{12}, a_{21}\}, \{v_1\}, 2)$$

$$u_4 : (\times , \{a_{13}, a_{21}\}, \{v_2\}, 4)$$

$$u_6 : (\times , \{a_{14}, a_{21}\}, \{v_3\}, 6)$$

$$u_8 : (\times , \{a_{15}, a_{21}\}, \{v_4\}, 8)$$

$$u_{10} : (\times , \{a_{16}, a_{21}\}, \{v_5\}, 10)$$

$$u_{12} : (/ , \{a_{31}, a_{11}\}, \{a_{31}\}, 12)$$

$$u_{13} : (\times , \{a_{12}, a_{31}\}, \{v_6\}, 13)$$

$$u_{15} : (\times , \{a_{13}, a_{31}\}, \{v_7\}, 15)$$

$$u_{17} : (\times , \{a_{14}, a_{31}\}, \{v_8\}, 17)$$

$$u_{19} : (\times , \{a_{15}, a_{31}\}, \{v_9\}, 19)$$

$$u_{21} : (\times , \{a_{16}, a_{31}\}, \{v_{10}\}, 21)$$

$$u_{23} : (/ , \{a_{41}, a_{11}\}, \{a_{41}\}, 23)$$

$$u_{24} : (\times , \{a_{12}, a_{41}\}, \{v_{11}\}, 24)$$

$$u_3 : (- , \{a_{22}, v_1\}, \{a_{22}\}, 3)$$

$$u_5 : (- , \{a_{23}, v_2\}, \{a_{23}\}, 5)$$

$$u_7 : (- , \{a_{24}, v_3\}, \{a_{24}\}, 7)$$

$$u_9 : (- , \{a_{25}, v_4\}, \{a_{25}\}, 9)$$

$$u_{11} : (- , \{a_{26}, v_5\}, \{a_{26}\}, 11)$$

$$u_{14} : (- , \{a_{32}, v_1\}, \{a_{32}\}, 14)$$

$$u_{16} : (- , \{a_{33}, v_7\}, \{a_{33}\}, 16)$$

$$u_{18} : (- , \{a_{34}, v_8\}, \{a_{34}\}, 18)$$

$$u_{20} : (- , \{a_{35}, v_9\}, \{a_{35}\}, 20)$$

$$u_{22} : (- , \{a_{36}, v_{10}\}, \{a_{36}\}, 22)$$

$$u_{25} : (- , \{a_{42}, v_{11}\}, \{a_{42}\}, 25)$$

$u_{26} : (\times, \{a_{13}, a_{41}\}, \{v_{12}\}, 26)$	$u_{27} : (-, \{a_{43}, v_{12}\}, \{a_{43}\}, 27)$
$u_{28} : (\times, \{a_{14}, a_{41}\}, \{v_{13}\}, 28)$	$u_{29} : (-, \{a_{44}, v_{13}\}, \{a_{44}\}, 29)$
$u_{30} : (\times, \{a_{15}, a_{41}\}, \{v_{14}\}, 30)$	$u_{31} : (-, \{a_{45}, v_{14}\}, \{a_{45}\}, 31)$
$u_{32} : (\times, \{a_{16}, a_{41}\}, \{v_{15}\}, 32)$	$u_{33} : (-, \{a_{46}, v_{15}\}, \{a_{46}\}, 33)$
$u_{34} : (/ , \{a_{51}, a_{11}\}, \{a_{51}\}, 34)$	
$u_{35} : (\times, \{a_{12}, a_{51}\}, \{v_{16}\}, 35)$	$u_{36} : (-, \{a_{52}, v_{16}\}, \{a_{52}\}, 36)$
$u_{37} : (\times, \{a_{13}, a_{51}\}, \{v_{17}\}, 37)$	$u_{38} : (-, \{a_{53}, v_{17}\}, \{a_{53}\}, 38)$
$u_{39} : (\times, \{a_{14}, a_{51}\}, \{v_{18}\}, 39)$	$u_{40} : (-, \{a_{54}, v_{18}\}, \{a_{54}\}, 40)$
$u_{41} : (\times, \{a_{15}, a_{51}\}, \{v_{19}\}, 41)$	$u_{42} : (-, \{a_{55}, v_{19}\}, \{a_{55}\}, 42)$
$u_{43} : (\times, \{a_{16}, a_{51}\}, \{v_{20}\}, 43)$	$u_{44} : (-, \{a_{56}, v_{20}\}, \{a_{56}\}, 44)$
$u_{45} : (/ , \{a_{61}, a_{11}\}, \{a_{61}\}, 45)$	
$u_{46} : (\times, \{a_{12}, a_{61}\}, \{v_{21}\}, 46)$	$u_{47} : (-, \{a_{62}, v_{21}\}, \{a_{62}\}, 47)$
$u_{48} : (\times, \{a_{13}, a_{61}\}, \{v_{22}\}, 48)$	$u_{49} : (-, \{a_{63}, v_{22}\}, \{a_{63}\}, 49)$
$u_{50} : (\times, \{a_{14}, a_{61}\}, \{v_{23}\}, 50)$	$u_{51} : (-, \{a_{64}, v_{23}\}, \{a_{64}\}, 51)$
$u_{52} : (\times, \{a_{15}, a_{61}\}, \{v_{24}\}, 52)$	$u_{53} : (-, \{a_{65}, v_{24}\}, \{a_{65}\}, 53)$
$u_{54} : (\times, \{a_{16}, a_{61}\}, \{v_{25}\}, 54)$	$u_{55} : (-, \{a_{66}, v_{25}\}, \{a_{66}\}, 55)$
$u_{56} : (/ , \{a_{32}, a_{22}\}, \{a_{32}\}, 56)$	
$u_{57} : (\times, \{a_{23}, a_{32}\}, \{v_{26}\}, 57)$	$u_{58} : (-, \{a_{33}, v_{26}\}, \{a_{33}\}, 58)$
$u_{59} : (\times, \{a_{24}, a_{32}\}, \{v_{27}\}, 59)$	$u_{60} : (-, \{a_{34}, v_{27}\}, \{a_{34}\}, 60)$
$u_{61} : (\times, \{a_{25}, a_{32}\}, \{v_{28}\}, 61)$	$u_{62} : (-, \{a_{35}, v_{28}\}, \{a_{35}\}, 62)$
$u_{63} : (\times, \{a_{26}, a_{32}\}, \{v_{29}\}, 63)$	$u_{64} : (-, \{a_{36}, v_{29}\}, \{a_{36}\}, 64)$
$u_{65} : (/ , \{a_{42}, a_{22}\}, \{a_{42}\}, 65)$	
$u_{66} : (\times, \{a_{23}, a_{42}\}, \{v_{30}\}, 66)$	$u_{67} : (-, \{a_{43}, v_{30}\}, \{a_{43}\}, 67)$
$u_{68} : (\times, \{a_{24}, a_{42}\}, \{v_{31}\}, 68)$	$u_{69} : (-, \{a_{44}, v_{31}\}, \{a_{44}\}, 69)$
$u_{70} : (\times, \{a_{25}, a_{42}\}, \{v_{32}\}, 70)$	$u_{71} : (-, \{a_{45}, v_{32}\}, \{a_{45}\}, 71)$
$u_{72} : (\times, \{a_{26}, a_{42}\}, \{v_{33}\}, 72)$	$u_{73} : (-, \{a_{46}, v_{33}\}, \{a_{46}\}, 73)$
$u_{74} : (/ , \{a_{52}, a_{22}\}, \{a_{52}\}, 74)$	
$u_{75} : (\times, \{a_{23}, a_{52}\}, \{v_{34}\}, 75)$	$u_{76} : (-, \{a_{53}, v_{34}\}, \{a_{53}\}, 76)$
$u_{77} : (\times, \{a_{24}, a_{52}\}, \{v_{35}\}, 77)$	$u_{78} : (-, \{a_{54}, v_{35}\}, \{a_{54}\}, 78)$
$u_{79} : (\times, \{a_{25}, a_{52}\}, \{v_{36}\}, 79)$	$u_{80} : (-, \{a_{55}, v_{36}\}, \{a_{55}\}, 80)$
$u_{81} : (\times, \{a_{26}, a_{52}\}, \{v_{37}\}, 81)$	$u_{82} : (-, \{a_{56}, v_{37}\}, \{a_{56}\}, 82)$
$u_{83} : (/ , \{a_{62}, a_{22}\}, \{a_{62}\}, 83)$	
$u_{84} : (\times, \{a_{23}, a_{62}\}, \{v_{38}\}, 84)$	$u_{85} : (-, \{a_{63}, v_{38}\}, \{a_{63}\}, 85)$
$u_{86} : (\times, \{a_{24}, a_{62}\}, \{v_{39}\}, 86)$	$u_{87} : (-, \{a_{64}, v_{39}\}, \{a_{64}\}, 87)$
$u_{88} : (\times, \{a_{25}, a_{62}\}, \{v_{40}\}, 88)$	$u_{89} : (-, \{a_{65}, v_{40}\}, \{a_{65}\}, 89)$

$u_{90} : (\times, \{a_{26}, a_{62}\}, \{v_{41}\}, 90)$	$u_{91} : (-, \{a_{66}, v_{41}\}, \{a_{66}\}, 91)$
$u_{92} : (/ , \{a_{43}, a_{33}\}, \{a_{43}\}, 92)$	
$u_{93} : (\times, \{a_{34}, a_{43}\}, \{v_{42}\}, 93)$	$u_{94} : (-, \{a_{44}, v_{42}\}, \{a_{44}\}, 94)$
$u_{95} : (\times, \{a_{35}, a_{43}\}, \{v_{43}\}, 95)$	$u_{96} : (-, \{a_{45}, v_{43}\}, \{a_{45}\}, 96)$
$u_{97} : (\times, \{a_{36}, a_{43}\}, \{v_{44}\}, 97)$	$u_{98} : (-, \{a_{46}, v_{44}\}, \{a_{46}\}, 98)$
$u_{99} : (/ , \{a_{53}, a_{33}\}, \{a_{53}\}, 99)$	
$u_{100} : (\times, \{a_{34}, a_{53}\}, \{v_{47}\}, 100)$	$u_{101} : (-, \{a_{54}, v_{47}\}, \{a_{54}\}, 101)$
$u_{102} : (\times, \{a_{35}, a_{53}\}, \{v_{48}\}, 102)$	$u_{103} : (-, \{a_{55}, v_{48}\}, \{a_{55}\}, 103)$
$u_{104} : (\times, \{a_{36}, a_{53}\}, \{v_{49}\}, 104)$	$u_{105} : (-, \{a_{56}, v_{49}\}, \{a_{56}\}, 105)$
$u_{106} : (/ , \{a_{63}, a_{33}\}, \{a_{63}\}, 106)$	
$u_{107} : (\times, \{a_{34}, a_{63}\}, \{v_{50}\}, 107)$	$u_{108} : (-, \{a_{64}, v_{50}\}, \{a_{64}\}, 108)$
$u_{109} : (\times, \{a_{35}, a_{63}\}, \{v_{51}\}, 109)$	$u_{110} : (-, \{a_{65}, v_{51}\}, \{a_{65}\}, 110)$
$u_{111} : (\times, \{a_{36}, a_{63}\}, \{v_{52}\}, 111)$	$u_{112} : (-, \{a_{66}, v_{52}\}, \{a_{66}\}, 112)$
$u_{113} : (/ , \{a_{54}, a_{44}\}, \{a_{54}\}, 113)$	
$u_{114} : (\times, \{a_{45}, a_{54}\}, \{v_{53}\}, 114)$	$u_{115} : (-, \{a_{55}, v_{53}\}, \{a_{55}\}, 115)$
$u_{116} : (\times, \{a_{46}, a_{54}\}, \{v_{54}\}, 116)$	$u_{117} : (-, \{a_{56}, v_{54}\}, \{a_{56}\}, 117)$
$u_{118} : (/ , \{a_{64}, a_{44}\}, \{a_{64}\}, 118)$	
$u_{119} : (\times, \{a_{45}, a_{64}\}, \{v_{55}\}, 119)$	$u_{120} : (-, \{a_{65}, v_{55}\}, \{a_{65}\}, 120)$
$u_{121} : (\times, \{a_{46}, a_{64}\}, \{v_{56}\}, 121)$	$u_{122} : (-, \{a_{66}, v_{56}\}, \{a_{66}\}, 122)$
$u_{123} : (/ , \{a_{65}, a_{55}\}, \{a_{65}\}, 123)$	
$u_{124} : (\times, \{a_{56}, a_{65}\}, \{v_{57}\}, 124)$	$u_{125} : (-, \{a_{66}, v_{57}\}, \{a_{66}\}, 125)$

A.2 Optimised solution of Example 7.

$u_1 : (/ , \{a_{21}, a_{11}\}, \{a_{21}\}, 1)$	
$u_2 : (\times , \{a_{12}, a_{21}\}, \{v_1\}, 2)$	$u_3 : (- , \{a_{22}, v_1\}, \{a_{22}\}, 3)$
$u_4 : (\times , \{a_{13}, a_{21}\}, \{v_2\}, 2)$	$u_5 : (- , \{a_{23}, v_2\}, \{a_{23}\}, 3)$
$u_6 : (\times , \{a_{14}, a_{21}\}, \{v_3\}, 2)$	$u_7 : (- , \{a_{24}, v_3\}, \{a_{24}\}, 3)$
$u_8 : (\times , \{a_{15}, a_{21}\}, \{v_4\}, 2)$	$u_9 : (- , \{a_{25}, v_4\}, \{a_{25}\}, 3)$
$u_{10} : (\times , \{a_{16}, a_{21}\}, \{v_5\}, 2)$	$u_{11} : (- , \{a_{26}, v_5\}, \{a_{26}\}, 3)$
$u_{12} : (/ , \{a_{31}, a_{11}\}, \{a_{31}\}, 1)$	
$u_{13} : (\times , \{a_{12}, a_{31}\}, \{v_6\}, 2)$	$u_{14} : (- , \{a_{32}, v_1\}, \{a_{32}\}, 3)$
$u_{15} : (\times , \{a_{13}, a_{31}\}, \{v_7\}, 2)$	$u_{16} : (- , \{a_{33}, v_7\}, \{a_{33}\}, 3)$
$u_{17} : (\times , \{a_{14}, a_{31}\}, \{v_8\}, 2)$	$u_{18} : (- , \{a_{34}, v_8\}, \{a_{34}\}, 3)$
$u_{19} : (\times , \{a_{15}, a_{31}\}, \{v_9\}, 2)$	$u_{20} : (- , \{a_{35}, v_9\}, \{a_{35}\}, 3)$
$u_{21} : (\times , \{a_{16}, a_{31}\}, \{v_{10}\}, 2)$	$u_{22} : (- , \{a_{36}, v_{10}\}, \{a_{36}\}, 3)$
$u_{23} : (/ , \{a_{41}, a_{11}\}, \{a_{41}\}, 1)$	
$u_{24} : (\times , \{a_{12}, a_{41}\}, \{v_{11}\}, 2)$	$u_{25} : (- , \{a_{42}, v_{11}\}, \{a_{42}\}, 3)$
$u_{26} : (\times , \{a_{13}, a_{41}\}, \{v_{12}\}, 2)$	$u_{27} : (- , \{a_{43}, v_{12}\}, \{a_{43}\}, 3)$
$u_{28} : (\times , \{a_{14}, a_{41}\}, \{v_{13}\}, 2)$	$u_{29} : (- , \{a_{44}, v_{13}\}, \{a_{44}\}, 3)$
$u_{30} : (\times , \{a_{15}, a_{41}\}, \{v_{14}\}, 2)$	$u_{31} : (- , \{a_{45}, v_{14}\}, \{a_{45}\}, 3)$
$u_{32} : (\times , \{a_{16}, a_{41}\}, \{v_{15}\}, 2)$	$u_{33} : (- , \{a_{46}, v_{15}\}, \{a_{46}\}, 3)$
$u_{34} : (/ , \{a_{51}, a_{11}\}, \{a_{51}\}, 1)$	
$u_{35} : (\times , \{a_{12}, a_{51}\}, \{v_{16}\}, 2)$	$u_{36} : (- , \{a_{52}, v_{16}\}, \{a_{52}\}, 3)$
$u_{37} : (\times , \{a_{13}, a_{51}\}, \{v_{17}\}, 2)$	$u_{38} : (- , \{a_{53}, v_{17}\}, \{a_{53}\}, 3)$
$u_{39} : (\times , \{a_{14}, a_{51}\}, \{v_{18}\}, 2)$	$u_{40} : (- , \{a_{54}, v_{18}\}, \{a_{54}\}, 3)$
$u_{41} : (\times , \{a_{15}, a_{51}\}, \{v_{19}\}, 2)$	$u_{42} : (- , \{a_{55}, v_{19}\}, \{a_{55}\}, 3)$
$u_{43} : (\times , \{a_{16}, a_{51}\}, \{v_{20}\}, 2)$	$u_{44} : (- , \{a_{56}, v_{20}\}, \{a_{56}\}, 3)$
$u_{45} : (/ , \{a_{61}, a_{11}\}, \{a_{61}\}, 1)$	
$u_{46} : (\times , \{a_{12}, a_{61}\}, \{v_{21}\}, 2)$	$u_{47} : (- , \{a_{62}, v_{21}\}, \{a_{62}\}, 3)$
$u_{48} : (\times , \{a_{13}, a_{61}\}, \{v_{22}\}, 2)$	$u_{49} : (- , \{a_{63}, v_{22}\}, \{a_{63}\}, 3)$
$u_{50} : (\times , \{a_{14}, a_{61}\}, \{v_{23}\}, 2)$	$u_{51} : (- , \{a_{64}, v_{23}\}, \{a_{64}\}, 3)$
$u_{52} : (\times , \{a_{15}, a_{61}\}, \{v_{24}\}, 2)$	$u_{53} : (- , \{a_{65}, v_{24}\}, \{a_{65}\}, 3)$
$u_{54} : (\times , \{a_{16}, a_{61}\}, \{v_{25}\}, 2)$	$u_{55} : (- , \{a_{66}, v_{25}\}, \{a_{66}\}, 3)$
$u_{56} : (/ , \{a_{32}, a_{22}\}, \{a_{32}\}, 4)$	
$u_{57} : (\times , \{a_{23}, a_{32}\}, \{v_{26}\}, 5)$	$u_{58} : (- , \{a_{33}, v_{26}\}, \{a_{33}\}, 6)$
$u_{59} : (\times , \{a_{24}, a_{32}\}, \{v_{27}\}, 5)$	$u_{60} : (- , \{a_{34}, v_{27}\}, \{a_{34}\}, 6)$

$u_{61} : (\times, \{a_{25}, a_{32}\}, \{v_{28}\}, 5)$	$u_{62} : (-, \{a_{35}, v_{28}\}, \{a_{35}\}, 6)$
$u_{63} : (\times, \{a_{26}, a_{32}\}, \{v_{29}\}, 5)$	$u_{64} : (-, \{a_{36}, v_{29}\}, \{a_{36}\}, 6)$
$u_{65} : (/ , \{a_{42}, a_{22}\}, \{a_{42}\}, 4)$	
$u_{66} : (\times, \{a_{23}, a_{42}\}, \{v_{30}\}, 5)$	$u_{67} : (-, \{a_{43}, v_{30}\}, \{a_{43}\}, 6)$
$u_{68} : (\times, \{a_{24}, a_{42}\}, \{v_{31}\}, 5)$	$u_{69} : (-, \{a_{44}, v_{31}\}, \{a_{44}\}, 6)$
$u_{70} : (\times, \{a_{25}, a_{42}\}, \{v_{32}\}, 5)$	$u_{71} : (-, \{a_{45}, v_{32}\}, \{a_{45}\}, 6)$
$u_{72} : (\times, \{a_{26}, a_{42}\}, \{v_{33}\}, 5)$	$u_{73} : (-, \{a_{46}, v_{33}\}, \{a_{46}\}, 6)$
$u_{74} : (/ , \{a_{52}, a_{22}\}, \{a_{52}\}, 4)$	
$u_{75} : (\times, \{a_{23}, a_{52}\}, \{v_{34}\}, 5)$	$u_{76} : (-, \{a_{53}, v_{34}\}, \{a_{53}\}, 6)$
$u_{77} : (\times, \{a_{24}, a_{52}\}, \{v_{35}\}, 5)$	$u_{78} : (-, \{a_{54}, v_{35}\}, \{a_{54}\}, 6)$
$u_{79} : (\times, \{a_{25}, a_{52}\}, \{v_{36}\}, 5)$	$u_{80} : (-, \{a_{55}, v_{36}\}, \{a_{55}\}, 6)$
$u_{81} : (\times, \{a_{26}, a_{52}\}, \{v_{37}\}, 5)$	$u_{82} : (-, \{a_{56}, v_{37}\}, \{a_{56}\}, 6)$
$u_{83} : (/ , \{a_{62}, a_{22}\}, \{a_{62}\}, 4)$	
$u_{84} : (\times, \{a_{23}, a_{62}\}, \{v_{38}\}, 5)$	$u_{85} : (-, \{a_{63}, v_{38}\}, \{a_{63}\}, 6)$
$u_{86} : (\times, \{a_{24}, a_{62}\}, \{v_{39}\}, 5)$	$u_{87} : (-, \{a_{64}, v_{39}\}, \{a_{64}\}, 6)$
$u_{88} : (\times, \{a_{25}, a_{62}\}, \{v_{40}\}, 5)$	$u_{89} : (-, \{a_{65}, v_{40}\}, \{a_{65}\}, 6)$
$u_{90} : (\times, \{a_{26}, a_{62}\}, \{v_{41}\}, 5)$	$u_{91} : (-, \{a_{66}, v_{41}\}, \{a_{66}\}, 6)$
$u_{92} : (/ , \{a_{43}, a_{33}\}, \{a_{43}\}, 7)$	
$u_{93} : (\times, \{a_{34}, a_{43}\}, \{v_{42}\}, 8)$	$u_{94} : (-, \{a_{44}, v_{42}\}, \{a_{44}\}, 9)$
$u_{95} : (\times, \{a_{35}, a_{43}\}, \{v_{43}\}, 8)$	$u_{96} : (-, \{a_{45}, v_{43}\}, \{a_{45}\}, 9)$
$u_{97} : (\times, \{a_{36}, a_{43}\}, \{v_{44}\}, 8)$	$u_{98} : (-, \{a_{46}, v_{44}\}, \{a_{46}\}, 9)$
$u_{99} : (/ , \{a_{53}, a_{33}\}, \{a_{53}\}, 7)$	
$u_{100} : (\times, \{a_{34}, a_{53}\}, \{v_{47}\}, 8)$	$u_{101} : (-, \{a_{54}, v_{47}\}, \{a_{54}\}, 9)$
$u_{102} : (\times, \{a_{35}, a_{53}\}, \{v_{48}\}, 8)$	$u_{103} : (-, \{a_{55}, v_{48}\}, \{a_{55}\}, 9)$
$u_{104} : (\times, \{a_{36}, a_{53}\}, \{v_{49}\}, 8)$	$u_{105} : (-, \{a_{56}, v_{49}\}, \{a_{56}\}, 9)$
$u_{106} : (/ , \{a_{63}, a_{33}\}, \{a_{63}\}, 7)$	
$u_{107} : (\times, \{a_{34}, a_{63}\}, \{v_{50}\}, 8)$	$u_{108} : (-, \{a_{64}, v_{50}\}, \{a_{64}\}, 9)$
$u_{109} : (\times, \{a_{35}, a_{63}\}, \{v_{51}\}, 8)$	$u_{110} : (-, \{a_{65}, v_{51}\}, \{a_{65}\}, 9)$
$u_{111} : (\times, \{a_{36}, a_{63}\}, \{v_{52}\}, 8)$	$u_{112} : (-, \{a_{66}, v_{52}\}, \{a_{66}\}, 9)$
$u_{113} : (/ , \{a_{54}, a_{44}\}, \{a_{54}\}, 10)$	
$u_{114} : (\times, \{a_{45}, a_{54}\}, \{v_{53}\}, 11)$	$u_{115} : (-, \{a_{55}, v_{53}\}, \{a_{55}\}, 12)$
$u_{116} : (\times, \{a_{46}, a_{54}\}, \{v_{54}\}, 11)$	$u_{117} : (-, \{a_{56}, v_{54}\}, \{a_{56}\}, 12)$
$u_{118} : (/ , \{a_{64}, a_{44}\}, \{a_{64}\}, 10)$	
$u_{119} : (\times, \{a_{45}, a_{64}\}, \{v_{55}\}, 11)$	$u_{120} : (-, \{a_{65}, v_{55}\}, \{a_{65}\}, 12)$
$u_{121} : (\times, \{a_{46}, a_{64}\}, \{v_{56}\}, 11)$	$u_{122} : (-, \{a_{66}, v_{56}\}, \{a_{66}\}, 12)$

A.2 $u_{123} : (/ , \{a_{65}, a_{55}\}, \{a_{65}\}, 13)$

$u_{124} : (\times , \{a_{56}, a_{65}\}, \{v_{57}\}, 14)$ $u_{125} : (- , \{a_{66}, v_{57}\}, \{a_{66}\}, 15)$

Bibliography

- [AE88] D. A. Abramson and G. K. Egan. An overview of RIMT/CSIRO parallel systems architecture project. *The Australian Computer Journal*, 20(3), August 1988.
- [AS93] T. M. Austin and G. S. Sohi. Tetra: Evaluation of serial program performance on fine-grain parallel processors. Technical Report TR1162, University of Wisconsin-Madison, July 1993.
- [ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
- [Ban88] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publisher, Boston, 1988.
- [Ban90] U. Banerjee. Unimodular transformations of double loops. In *Proc. of 3rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 192-219, Irvine, CA, August 1990.
- [Bar81] H. P. Barendregt. *The Lambda Calculus-its syntax and semantics*. North-Holland, 1981.
- [Bet al94a] B. Blume and *et al.* Polaris: The next generation in parallelizing compilers. Technical Report 1375, University of Illinois at Urbana-Champaign, 1994.
- [Bet al94b] W. Blume and *et al.* Automatic detection of parallelism: A grand challenge for high-performance computing. Technical Report 1348, University of Illinois at Urbana-Champaign, 1994.

- [BGet a/94] D. F. Bacon, S. L. Graham, and *et al.* Compiler transformations for high performance computing. *ACM Computing Surveys*, 26(4):295-420, December 1994.
- [Bir87] R. S. Bird. An introduction of the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3-42. Springer-Verlag Berlin Heidelberg, 1987.
- [Bir89] R. S. Bird. Algebraic identities for program calculation. *The Computer Journal*, 32(2):122-126, Feb 1989.
- [BM90] J. P. Banstre and D. L. Metayer. The GAMMA model and its discipline of programming. *Science of Computer Programming*, 15:55-77, 1990.
- [BM93] J. P. Banstre and D. L. Metayer. Programming by multiset transformation. *Communications of the ACM*, 36(1):98-111, 1993.
- [Can69] L. E. Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. Montana State University, 1969. Ph.D thesis.
- [CBF91] S. Chatterjee, G. E. Blelloch, and A. L. Fisher. Size and access inference for data-parallel programs. In *Proceedings of ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.
- [CG89] N. J. Carriero and D. Gelernter. Linda in context. *Communications of ACM*, 32(4):444-458, 1989.
- [CG90] N. Carriero and D. Gelernter. *How to Write Parallel Programs*. MIT Press, 1990.
- [CG95] I. Chabini and B. Gendron. Parallel performance measures revisited. In *High Performance Computing Symposium'95 (HPCS95)*, Montreal, Canada, July 1995.

- [Che85] D. K. Chen. Maxpar: An execution driven simulator for studying parallel systems. Technical Report Master Thesis, University of Illinois at Urbana-Champaign, 1985.
- [Chu46] A. Church. *The Calculi of λ -conversion*. Princeton University Press, 1946.
- [CK95a] P. Chen and E. V. Krishnamurthy. ABCOM: A parallelism revelation model. In *Proceedings of the Seventh IASTED International Conference on Parallel and Distributed Computing and Systems*, Washington D. C. USA, Oct. 1995.
- [CK95b] P. Chen and E. V. Krishnamurthy. Relation-based inference of parallelism in programming. In *Proceedings of the Australasian Conference on Parallel and Real-Time Systems (PART'95)*, Fremantle, WA, Australia, 28-29 Sept. 1995.
- [CK95c] P. Chen and E. V. Krishnamurthy. A tuple-space methodology to reveal parallelism and achieve high performance in computation. In *Proceedings of High Performance Computing Symposium 95 (HPCS'95)*, Montreal, Canada, 10-12 July 1995.
- [CKY95] P. Chen, E. V. Krishnamurthy, and J. Yang. Program parallelization based on an abstract computational tuple-space model. *Australian Computer Science Communications*, 17(1):66-75, 1995.
- [CM88] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Company, 1988.
- [Col89] M. Cole. *Algorithm Skeletons: Structured Management of Parallel Computation*. Pitman/MIT, 1989.
- [Col92] M. Cole. Parallel software paradigms. In Lydia Kronsjo and Dean Shumsheruddin, editors, *Advances in Parallel Algorithms*. Blackwell Scientific Publications, 1992.
- [Cor90] Thinking Machines Corporation. *CM Fortran User's Guide*. Version 0.7, 1990.

- [CSE93] D. E. Culler, K. E. Schauser, and T. V. Eicken. Two functional limits on dataflow multiprocessing. In *Proceedings of the IFIP WG 10.3 Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. Elsevier Science Publisher, January 1993.
- [Det al93] J. Darlington and *et al.* Parallel programming using skeleton functions. In *PARLE'93*, pages 146–160. Springer-Verlag, LNCS 694, 1993.
- [Dij72] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, N.J., 1972.
- [DK82] A. L. Davis and R. M. Keller. Data flow program graphs. *IEEE Computer*, pages 26–41, February 1982.
- [Eka91] K. Ekanadham. A perspective on Id. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 197–247. ACM Press, 1991.
- [Ell86] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures*. MIT Press, 1986.
- [Fea94] P. Feautrier. Fine-grain scheduling under resource constraints. In *Languages and Compilers for Parallel Computing, Proceeding of the 7th International Workshop, LNCS-892*, Ithaca, NY, USA, Aug. 1994.
- [FJea88] G. Fox, M. Johnson, and *et al.* Matrix algorithms I: Matrix multiplication (chapter 10). In G. Fox, M. Johnson, and *et al.*, editors, *Solving Problems on Concurrent Processors*, pages 167–185. Prentice Hall, 1988.
- [Fox90] G. C. Fox. Hardware and software architectures for irregular problem architectures. In J. Saltz P. Mehrotra and R. Voigt, editors, *Unstructured Scientific Computation on Scalable Microprocessors*, pages 125–160. The MIT Press, 1990.
- [Fox92] G. C. Fox. Parallel computers and complex systems. In David. Green and Teery Bossomaier, editors, *Complex Systems: from biology to computation*, pages 272–287. IOS Press, 1992.

- [FS92] Y. Feldman and E. Shapiro. Spatial machines: A more realistic approach to parallel computation. *Communications of The ACM*, 35(10), Oct. 1992.
- [Gea91] G. Goff and et al. Practical dependence testing. *SIGPLAN Note*, 26(6):15–29, 1991.
- [GGB93] G. Gao, J. L. Gaudiot, and L. Bic. Special issue on dataflow and multithreaded architectures. *Journal of Parallel and Distributed Computing*, 18:271–272, 1993.
- [Gis84] J. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle*, Ph.D. Thesis. Dept. of Comp. Sci., Stanford University, 1984.
- [GR88] A. Gibbons and W. Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.
- [Gre75] I. Greif. Semantics of communicating parallel processes, PhD Thesis. Technical Report Project MAC Report TR-154, MIT, 1975.
- [GY93] A. Gerasoulis and T. Yang. Scheduling program task graphs on mimd architectures. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 153–186. Kluwer Academic Publishers, 1993.
- [Hea91] M. Heath. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, Sep. 1991.
- [Hea93] M. W. Hall and et al. Fiat: A framework for interprocedural analysis and transformation. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Aug. 1993.
- [HMA95] M. W. Hall, B. R. Murphy., and S. P. Amarasinghe. Interprocedural analysis for parallelization: Design and experience. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, Feb. 1995.

- [Hwa93] K. Hwang. *Advanced Computer Architecture, Parallelism Scalability, Programmability*. McGraw-Hill Inc., 1993.
- [Jay95] C. B. Jay. Polynomial polymorphism. *Australian Computer Science Communications*, 17(1):237-2243, 1995.
- [JP93] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, Jun. 1993.
- [Kea94] C. H. Koebel and et al. *The High Performance Fortran Handbook*. The MIT Press, 1994.
- [KGS94] R. Kramer, R. Gupta, and M. L. Soffa. The combining DAG: A technique for parallel data flow analysis. *IEEE Trans. on Parallel and Distributed Systems*, 5(8), August 1994.
- [KMC72] D. J. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, 21(12), Dec. 1972.
- [KN91] D. Kimelman and T. Ngo. The RP3 program visualization environment. *IBM J. of R. and D.*, 35(5/6):635-651, 1991.
- [KN93] J.S. Kowalik and K.W. Neves. Software for parallel computing: Key issues and research directions. In J. S. Kowalik and L. Grandinette, editors, *Software for Parallel Computation*, pages 3-33. Springer-Verlag, 1993.
- [KR90] M. R. Karp and V. Ramachandran. Parallel algorithms for shared memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 870-941. Elsevier Science Publishers, 1990.
- [Kum88] M. Kumar. Measuring parallelism in computation-intensive scientific engineering applications. *IEEE Transactions on Computers*, 37(9):1088-1098, 1988.

- [Lam74] L. Lamport. The parallel execution of do loops. *Communications of The ACM*, 17(2):83-93, Feb 1974.
- [Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.
- [Lan93] M. A. Langston. Time-space optimal parallel computation. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Programming Transformation*, pages 207-223. Kluwer Academic Publishers, 1993.
- [Lar90] J. R. Larus. Abstract execution: A technique for efficiently tracing programs. *Software Practice and Experience*, 20(12), December 1990.
- [Lea90] Z. Li and et al. Data dependence analysis on multi-dimensional array references. *IEEE, Tran. Paralle. Distrib. Syst*, 1(1):26-34, 1990.
- [Lew94] T. G. Lewis. *Foundations of Parallel Programming— A Machine Independent Approach*. IEEE Computer Society Press, 1994.
- [Lil93] D. J. Lilja. The impact of parallel loop scheduling strategies on prefetching in a shared-memory multiprocessor. Technical report, University of Minnesota, 1993.
- [Lil94] D. J. Lilja. Exploiting the parallelism available in loops. *IEEE Computer*, pages 13-26, Feb 1994.
- [LW92] M.S. Lam and R.P. Wilson. Limits of control flow on parallelism. In *Proceedings of ISCA-19*, pages 46-57, Gold Coast, Australia, May 1992.
- [Mal90] G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255-279, 1990.
- [MC82] T. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys*, 14(4), Dec. 1982.

- [MC91] B. Miller and J Choi. Breakpoints and halting in distributed programs. In *Proceedings of 8th Conference on Distributed Computing Systems*, pages 316-323, 1991.
- [McG82] J. R. McGraw. The VAL language: Description and analysis. *ACM Trans. on Programming Languages and Systems*, 4(1):44-82, January 1982.
- [Mea91] D. E. Maydan and et al. Efficient and exact data dependence analysis. *SIGPLAN Note*, 26(6):1-14, 1991.
- [Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theor. Comput. Sci.*, 25, 1983.
- [MK95] V. K. Murthy and E. V. Krishnamurthy. Probabilistic parallel programming based on multiset transformation. *Future Generation Computer Systems*, 11, 1995.
- [MPC90] S. P. Midkiff, D. Padua, and R. Cytron. Compiling programs with user parallelism. In *Languages and Compilers for Parallel Computing*, pages 402-422. The MIT Press, 1990.
- [MPK91] W. Hseush M. Pongami and G. Kaiser. Debugging multi-threaded programs with MPD. *IEEE Software*, 8(3):37-43, 1991.
- [NA91] D. Nussbaum and A. Agarwal. Scalability of parallel machine. *Commun. ACM*, 34(3), 1991.
- [NF84] A. Nicolau and J. A. Fisher. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers*, 33:968-976, Nov 1984.
- [NN94] S. Novack and A. Nicolau. Mutation scheduling: A unified approach to compiling for fine-grain parallelism. In *Languages and Compilers for Parallel Computing, Proceeding of the 7th International Workshop, LNCS-892*, Ithaca, NY, USA, Aug. 1994.

- [Pan91] C. M. Pancake. Software support for parallel computing: Where are we headed? *Communications of ACM*, 34(11):53-64, Nov 1991.
- [Par93] H. Partsch. Some experiments in transforming towards parallel executability. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Programming Transformation*, pages 71-110. Kluwer Academic Publishers, 1993.
- [Pep93] P. Pepper. Deductive derivation of parallel programs. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 1-54. Kluwer Academic Publishers, 1993.
- [PKL80] D. A. Padau, D. J. Kuck, and D. H. Lawrie. High speed multiprocessors and compilation techniques. *IEEE Trans. on Computers*, 29(9), September 1980.
- [PP90] P. Peterson and D. Padua. Machine-independent evaluation of parallelizing compilers. Technical Report 1173-CSR, University of Illinois at Urbana-Champaign, 1990.
- [PP92] P. Peterson and D. Padua. Dynamic dependence analysis: A novel method for data dependence evaluation. In *Proceedings of 5th International Workshop on Languages and Compilers for Parallel Computing*, pages 64-81, 1992.
- [PPP93] A. Pettorossi, E. Pietropoli, and M. Proietti. The use of tupling strategy in the development of parallel programs. In J. Reif R. Paige and R. Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 111-152. Kluwer Academic Publishers, 1993.
- [Pra86] V. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33-71, 1986.
- [Pra94] V. Pratt. Time and information in sequential and concurrent computation. *LNCS*, 907:1-24, 1994.

- [Pug92] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, pages 102–114, August 1992.
- [PW84] S. S. Pinter and P. Wolper. A temporal logic to reason about partially ordered computations. In *Proceedings of 3rd ACM Symp. on Principles of Distributed Computing*, Vancouver, Aug. 1984.
- [PW94] W. Pugh and D. Wonnacott. Static analysis of upper and lower bounds on dependences and parallelism. Technical Report Technical Report CS-TR-3250, Department of Computer Science, University of Maryland, March 1994.
- [RR93] S. Rajasekaran and J. H. Reif. Derivation of randomized sorting and selection algorithms. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Programming Transformation*, pages 187–205. Kluwer Academic Publishers, 1993.
- [Sab88] G. Sabot. *The Paralation Model: Architecture-Independent Parallel Programming*. MIT Press, 1988.
- [SC94] D. B. Skillicorn and W. Cai. A cost calculus for parallel functional programming. Technical report, Queen's University, 1994.
- [Sha85] J. A. Sharp. *Data Flow Computing*. Ellis Horwood Limited, 1985.
- [Ske91] S. K. Skedzielewski. Sisal. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 105–152. ACM Press, 1991.
- [Ski90] D. B. Skillicorn. Architecture-independent parallel computation. *The Computer Journal*, 23(12), 1990.
- [Ski91] D. B. Skillicorn. Models for practical parallel computation. *Parallel Programming*, 20(3):133–158, 1991.
- [Ski93] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J.S. Kowalik and L. Grandinetti, editors, *Software for Parallel Computation (NATO ASI Series F)*. Springer-Verlag, 1993.

- [Smi93] D. R. Smith. Derivation of parallel sorting algorithms. In John Reif Robert Paige and Ralph Wachter, editors, *Parallel Algorithm Derivation and Program Transformation*, pages 55–70. Kluwer Academic Publishers, 1993.
- [Spi89] J. M. Spivey. A categorical approach to the theory of lists. *Lecture Notes in Computer Science*, 375:399–408, 1989.
- [SW85] S. K. Skedzielewski and M. L. Welcome. Data flow graph optimisation in IF1. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architectures*. Springer-Verlag, New York, 1985.
- [Szy91] B. K. Szymanski. EPL — parallel programming with recurrent equations. In Boleslaw K. Szymanski, editor, *Parallel Functional Languages and Compilers*, pages 51–102. ACM Press, 1991.
- [TGH92a] K.B. Theobald, G.R. Gao, and L. J. Hendren. The effects of resource limitations on program parallelism. In *Proceedings of Workshop on Data-Flow Computing*, Hamilton Island, Australia, May 1992.
- [TGH92b] K.B. Theobald, G.R. Gao, and L. J. Hendren. On the limits of program parallelism and its smoothability. In *MICRO-25*, pages 10–19, Portland, Oregon, Dec 1992.
- [Val90a] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
- [Val90b] L. G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 944–971. Elsevier Science Publishers, 1990.
- [Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of ASPLOS-IV*, pages 176–188, Santa Clara, California, Apr 1991.
- [WB87] M. Wolfe and U. Banerjee. Data dependence and its application to parallel processing. *Parallel Programming*, 16(2):137–178, 1987.

- [Wet al94] C. Eric Wu and *et al.* Trace-based analysis and tuning fro distributed parallel applications. In *Proceedings of the 1994 international Conference on Parallel and Distributed Systems*, pages 716-723, Hsinchu, Taiwan, Dec 1994.
- [Win80] G. Winskel. Events in computation. Technical Report PhD Thesis, CST-10-80, University of Edinburgh, 1980.
- [Win84] G. Winskel. A new definition of morphisms on Petri Nets. In *CMU/SERC Workshop on Analysis of Concurrency, Springer-Verlag LNCS 196*, Pittsburgh, 1984.
- [Wol89] M. Wolfe. *Optimizing Supercompilers for Supercomputers*. MIT Press, 1989.
- [WT92] M. Wolfe and C. Tseng. The power test for data dependence. *IEEE Trans. Parall. Distrib. Syst.*, 2(4):591-601, 1992.