COMPUTATIONAL MODELS AND PROGRAM SYNTHESIS FOR PARALLEL OUT-OF-CORE COMPUTATION

by

Zhiyong Li

Department of Computer Science
Duke University

Date: 6/28/96
Approved:

John H. Reif, Supervisor

Robert A. Wagner

Jan F. Prins

Carla S. Ellis

John A. Board

Dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

1996

UMI Number: 9704729

Copyright 1996 by
Li, Zhiyong

All rights reserved.

UMI Microform 9704729
Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Copyright © 1996 by Zhiyong Li
All rights reserved

ABSTRACT

(Computer Science)

COMPUTATIONAL MODELS AND PROGRAM SYNTHESIS FOR PARALLEL OUT-OF-CORE COMPUTATION

by

Zhiyong Li

Department of Computer Science
Duke University

Date: 6/28/96

Approved:

John H. Reif, Supervisor

Robert A. Wagner

Jan F. Prins

Carla S. Ellis

John A. Board

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University

1996

Abstract

As the performance gap between processors and memory systems continues to increase, memory and I/O subsystems are increasingly becoming the major bottleneck for many I/O-intensive out-of-core applications. To address this problem, new models of parallel computation and new methods of program synthesis for out-of-core computation are needed. This thesis presents our contributions in these two areas.

We first introduce the concept of resource metrics to characterize various models of parallel computation. Based on this concept, we introduce a LogP-HMM model for modeling parallel machines with multi-level memories and a communication network. More specifically, LogP-HMM is a representative of a class of models that are formed by combining a network model with a hierarchical memory model. We introduce a variant of the LogP-HMM model, the LogP-UMH model, which combines the LogP model with the UMH model. We present several near-optimal fast Fourier transform (FFT) and sorting algorithms for both models.

We then introduce an algebraic method for synthesizing efficient parallel out-of-core programs. This method uses tensor products to represent a class of algorithms with recursive computational structures, known as block recursive algorithms. To obtain the improved performance, we synthesize programs of the block recursive algorithms for two variants of the LogP-HMM model, a single-processor multi-disk

model, and a multi-processor multi-disk model. These two models share a common property, namely that out-of-core data is distributed in a block-cyclic manner. We introduce a methodology, based on tensor bases, to describe this data distribution on multiple disks. We then derive efficient programs by the following steps. First, we use a greedy or a dynamic programming algorithm to transform the tensor product formulas of block recursive algorithms into an efficient form. Second, we synthesize efficient programs by analyzing the structures of the mathematical representations for both the input computation and the data distribution.

We demonstrate the effectiveness of our approach by synthesizing parallel out-of-core FFT programs for the CM-5 with the parallel file access. The experimental results show that, the synthesized FFT programs with the parallel file access run up to 10 times faster than the FFT programs with the serial file access.

Acknowledgements

I thank first my advisor, John Reif, for his valuable advice, support and guidance throughout my graduate career. John has supported my work on several different research projects and has helped me focus on models and tensor products for I/O intensive applications. Without his broad knowledge and interests in both theoretical and experimental research, this thesis would never have been completed.

I thank Robert Wagner for serving on my Ph.D. committee. He has spent many hours helping me clarify my earlier ideas. He has also provided detailed technical and editorial comments on drafts of this thesis. I am grateful to have worked with Jan Prins. His insight has helped me to always look for better answers and reliable results. I would also like to thank him for taking time from his busy schedule to attend my prelim exam and my Ph.D. Defense. Finally, I would like to thank other members of my committee, Carla Ellis and John Board, for their time and guidance on my thesis.

Parts of this thesis represent collaborative efforts with other researchers. Peter Mills has worked very closely with me on models of parallel computation. Peter also provided editorial comments on earlier drafts of this thesis. Sandeep Gupta brought his expertise and valuable experience on tensor products and parallel compilers from Ohio State University.

The joint project, Proteus, directed by John Reif, Jan Prins, and Alan Goldberg at Kestrel Institute, provided not only part of my support but also an ideal research environment. The regular Parallel Lunch meeting of Proteus has largely broaden my knowledge on parallel programming languages and compilers. My working experience at IBM has helped me to understand tensor products and tensor product methodology. Thanks to Jerry Pechanek and John Glossner at IBM.

Several other faculty at Duke and other schools showed their interests in my work. Thanks to Jeff Vitter. His two-level memory model provides a starting point for my modeling parallel machines with secondary storage. Thanks to Don Rose and Xiaobai Sun for inviting me to present parts of my work in their Numerical Seminar. Thanks to Ming Kao for discussions on models of parallel computation. Thanks to Tom Cormen at Dartmouth and Charles Koelbel at Rice for their comments. Thanks to David Kotz for maintaining the Parallel I/O archives.

Over the course of four years at graduate school, I have been fortunate to enjoy the company of many excellent students at Duke and elsewhere. I am grateful for their help and friendship. I hope I have listed all of them: Eric Anderson, Subhrajit Bhattacharya, Shenfeng Chen, Yong Gao, Patrick Harubin, Wei Jin, Stephen Majercik, Daniel Palmer, Apratim Purakayastha, Vijay Srinivasan, Hongyan Wang, Bill Yakowenko, Aki Yoshida, and Yun Zhang.

I would also like to thank the people who make the Department of Computer Science a great place to work and to learn. I thank Tina Gather for solving many financial problems; Ken Robinson for assistance at various aspects; and Robert Ingram for bringing the CM-5 back on-line when I continuously broke the Scalable Disk Array.

Finally, I wish to thank my parents, my sisters and other relatives. Their encouragement and support have made possible my accomplishments. I thank my wife Jun.

Her love, understanding and interests in my research have kept me going through those long and stressful four years. My son's happy smile has brought a lot of joyful moments to my graduate life.

Contents

Abstract iv

Acknowledgements vi

List of Figures xiv

List of Tables xvii

1 Introduction 1

1.1 Motivation 1

1.2 Background 3

1.2.1 Models and Algorithm Design 4

1.2.2 Methods of Program Synthesis 5

1.2.3 Other Work in Parallel I/O 7

1.3 Summary of Our Approach 8

1.3.1 Models for Parallel Machines with Multi-Level Memories . 8

1.3.2 Synthesizing Efficient Out-of-Core Programs 10

1.4 Contributions of This Thesis 15

2 Models and Resource Metrics for Parallel Computation 17

2.1 Resource Metrics 20

2.2 Basic Synchronous Models 23

2.2.1 PRAM 23

2.2.2 VRAM 23

2.3 Extensions of the Basic Models 24

2.3.1 Asynchronous Models 25

2.3.2 Models Incorporating Communication Latency and Bandwidth 27

2.3.3 Hierarchical Models 33

2.4 Conclusions 36

3 Models for Distributed Hierarchical Memory Machines 39

3.1 The LogP-HMM Model 40

3.1.1 Definition of the Model 41

3.1.2 Algorithm Design and Analysis 41

3.1.3 An Alternative Optimal FFT Algorithm for P-HMM 46

3.2 The LogP-UMH Model 46

3.2.1 Algorithm Design and Analysis 47

3.2.2 Matching LogP-UMH to Practical Parallel Machines 54

3.3 Conclusions 57

4 Synthesizing Programs from Tensor Products 58

4.1 Tensor Product Algebra 59

4.1.1 Definition of Tensor Products 59

4.1.2 Stride Permutations 61

4.1.3 Tensor Bases 61

4.2 Tensor Product Formulation of Block Recursive Algorithms 64

4.3 Code Generation from Tensor Products 68

4.3.1 Writing Programs from Tensor Products 68

4.3.2 Efficient Tensor Product Transformations 71

4.4 Conclusions 77

5 Semantics of Out-of-Core Data Distributions and Access Patterns 79

5.1 Data Organization on Multi-Disk Systems 80

5.2 Tensor Bases for Data Distributions and Access Patterns 82

5.2.1 Data Distribution Bases 82

5.2.2 Tensor Bases for Data Access 84

5.3 Parallel Data Access to Multi-Disk Systems 87

5.3.1 Parallel I/O Operations on Multi-Disk Systems 87

5.3.2 Generating Parallel I/O Operations Using Tensor Bases . . . 88

6 Synthesizing Out-of-Core Programs for Single-Processor Multi-Disk Systems 92

6.1 Machine Model and Performance Metrics 94

6.2 Overview of Program Synthesis 96

6.3 Synthesizing I/O-Efficient Programs 97

6.3.1 Parallel I/O Code Generation 97

6.3.2 Synthesizing Programs for Stride Permutations 102

6.3.3 Synthesizing Programs for Tensor Products 110

6.3.4 Determining Efficient Data Distributions 124

6.3.5 Transforming Tensor Product Formulas 124

6.4 Performance Results of Synthesized Programs 127

6.4.1 Matrix Transposition 127

6.4.2 Tensor Products 128

6.4.3 Tensor Product Formulas 129

6.5 Conclusions 130

7 Synthesizing Out-of-Core Programs for Multi-Processor Multi-Disk Systems 132

7.1 Machine Model and Performance Metrics 134

7.2 Semantics of Out-of-Core Data Distributions 135

7.3 Disk-Optimal Programs 137

7.3.1 Parallel Out-of-Core SPMD Code Generation 137

7.3.2 Disk-Optimal Programs for Tensor Products 138

7.4 Transforming Tensor Product Formulas 142

7.4.1 Determining Efficient Data Distributions 143

7.5 An Example: The Cooley-Tukey FFT 144

7.6 Conclusions 146

8 Synthesizing Out-of-Core FFT Programs for the CM-5 Machine 148

8.1 Overview of the CM-5 Machine 148

8.2 Synthesizing FFT Programs for the CM-5 151

8.3 Performance Issues 152

8.4 Conclusions 156

9 Conclusions **158**

9.0.1 Contributions 158

9.0.2 Future Research 159

A Computing Sub-Arrays Using Tensor Bases **166**

Bibliography **170**

Biography **181**

List of Figures

1.1 Machine model for distributed-memory machines. 9

1.2 Machine models for distributed-memory machines with multi-disks as secondary storage. 11

1.3 Procedure for synthesizing I/O programs for block recursive algorithms. 14

2.1 A 16-input FFT graph. 19

2.2 The PRAM model. 24

3.1 Structure of the LogP-HMM model. 40

3.2 The initial input is regarded as a two-dimensional matrix. 51

3.3 The initial and final data distributions on two processors. Note that we only show the fourth level of the memory hierarchy. 53

4.1 Vector factorization and array linearization. 63

5.1 Data organization on multi-disks for $N = 64$, $B_d = 4$, and $D = 4$. . 81

5.2 Data organization on multi-disks for $N = 64$, $B_d = 4$, $D = 4$, and $B = 8$. 82

5.3 An example data organization and access pattern. 86

6.1 Procedure of code generation for a tensor product. 100

6.2 Code for $I_4 \otimes F_2 \otimes I_4$, where X is an array of size M and $A = I_2 \otimes F_2 \otimes I_4$. 102

6.3 Algorithm for determining input and output loop bases for stride permutations. 104

6.4 Relations among θ_m, θ_μ, λ_{μ_2}, λ_{μ_1}, and λ_μ. 107

6.5 Parallel I/O Program for L_4^{36} 110

6.6 Example matrix transposition. (a) Inputs when viewed as an 8×4 two-dimensional array. (b) Input data distribution on two disks. (c) Load physical tracks T_0, T_4, in-core permutation, and write to physical tracks T_0, T_2. (d) Load physical tracks T_1, T_5, in-core permutation, and write to physical tracks T_4, T_6. 111

6.7 Example matrix transposition. (a) Load physical tracks T_2, T_6, in-core permutation, and write to physical tracks T_1, T_3. (b) Load physical tracks T_3, T_7, in-core permutation, and write to physical tracks T_5, T_7. (c) Outputs. 112

6.8 Algorithm for determining input loop bases and the value of Z for a tensor product. 116

6.9 Algorithm for computing R_d. 117

6.10 Algorithm for computing S. 118

6.11 Constructing portions of memory-loads from a physical block. . . . 119

6.12 Algorithm for computing the efficient size of data distributions. . . . 124

7.1 Procedure of SPMD code generation for a tensor product. 139

7.2 Node program for the tensor product with $VC \leq M$. 140

7.3 Constructing a memory-load from out-of-core data on disks. 141

7.4 Node program for the tensor product with $VC \geq M \geq VDB$. . . . 142

7.5 Synthesized out-of-core Cooley-Tukey FFT program. 146

8.1 An example architecture of the CM-5 machine. 149

8.2 Out-of-core FFT computation (time in seconds) under various block sizes (in bytes). 152

8.3 Execution times (in seconds) of using serial file access vs parallel file access for out-of-core FFT on the CM-5 with 16 nodes. 154

8.4 MFLOPS of out-of-core FFT program on the CM-5 with 16 nodes. . 157

9.1 In-core data distribution and redistribution. 162

9.2 Diagonal tracks and data redistribution. 164

A.1 Constructing sub-arrays from the input vector or array. 167

List of Tables

2.1 Resource metrics chosen by different models of parallel computation. 37

3.1 Comparison of the CM-5 node memory hierarchy with the LogP-UMH. $\alpha = 256$, $\rho = 32$ and $1/f(\ell) = 4(byte/cycle)$. 55

3.2 Comparison of the SP-2 node memory hierarchy with the LogP-UMH. $\alpha = 32$, $\rho = 64$ and $1/f(\ell) = 4(\ell + 1)$. 56

4.1 Summary of the symbols used in Chapter 5 and Chapter 6. 78

6.1 Number of I/O passes for stride permutation L_Q^{PQ}. $D = 4$, $B_d = 4$, $M = 64$, and $N = PQ = 2048$. 127

6.2 Number of I/O passes for stride permutation L_Q^{PQ}. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = PQ = 2^{50}$. 128

6.3 Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$. 128

6.4 Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$. 129

6.5 Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$. 129

6.6 Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$ with various data distributions. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = RVC$. 130

6.7 Number of I/O passes for the synthesized programs using Greedy, Dynamic programming (D.P), and Multiple-step dynamic programming(M.D.P) methods ($D = 16$, $B_d = 512$, and $M = 2^{22}$). 130

8.1 B values chosen from the prediction function. 155

Chapter 1

Introduction

1.1 Motivation

Parallel computers, including distributed-memory machines, promise to provide efficient solutions for many important computationally demanding applications in areas such as seismic signal processing, computational fluid dynamics, and molecular dynamics [64]. Many of these applications, besides needing enormous computing power, also use enormous amounts of data. For example, applications in environmental and earth science such as four-dimensional data assimilation require 100 Mbytes to 1 Gbyte of external data per run, whereas some applications in computational fluid dynamics require as much as 1 Tbyte of disk space [71]. This large amount of data, henceforth called out-of-core data, may not fit into the main memory and thus needs to be stored on secondary storage. We call an application whose memory requirement during the course of computation exceeds the aggregate memory space of processing nodes, an *out-of-core* problem. The programs for solving out-of-core problems are normally I/O intensive since their execution may involve multiple passes over out-of-core data.

However, over the last two decades, the performance of I/O subsystems has no-

ticeablely lagged behind the advances of processors and communication networks. For example, the performance of processors and communication networks has increased about 30% to 50% each year. On the other hand, the performance of memory and disk access has only improved about 3% to 5% each year. Moreover, the performance gap between CPU and memory systems may become even larger in the near future. Therefore, to effectively solve out-of-core problems, I/O efficient solutions must be developed, a task requiring efforts from both software and hardware [71].

In the area of software there are various efforts in directions such as developing parallel I/O models, parallel file systems, parallel workload characterizations, and, in particular, out-of-core data parallel languages and parallel compilers. It is well-known that developing efficient *in-core* programs with the goal of optimizing communications on parallel machines is in itself a difficult and error-prone procedure. The task of developing efficient out-of-core programs with the goal of minimizing the overhead including both in-core communications and I/Os is even more challenging. Techniques based on an efficient data distribution that minimizes the I/O overhead, may not minimize the overhead of in-core communications simultaneously. Therefore, there is a need to develop systematic methods, such as methods of program synthesis, for assisting the development of efficient out-of-core programs.

To synthesize efficient out-of-core programs, it is necessary first to model I/O subsystems as well as the entire memory hierarchy appropriately. Moreover, in the larger vein, the accurate and flexible modeling of memory hierarchy plays a key role in parallel algorithm design. A computational model hides the architectural details from software designers, and guides the high-level design of parallel algorithms as well as providing accurate estimations of performance. However, unlike sequential machines, different parallel machines typically have very different architectures such as different inter-connection networks and different configurations of I/O subsys-

tems. Each architecture of the parallel machines has distinct properties on which the performance of algorithms may depend. To design algorithms and portable software to accommodate the specifics of various machines, one promising approach is to build more detailed parameterized models of parallel computation [55, 59, 34]. However, current refined models inadequately address the costs of both network communication and memory hierarchy.

In this thesis, we present our efforts on two fronts: improved models of parallel computation and automatically generating I/O efficient programs for an important class of applications. We first develop a parallel model for distributed-memory machines, in which each processor has its own memory hierarchy. Then, we develop methods for automatically generating efficient out-of-core programs of *block recursive algorithms* for distributed-memory machines modeled by two variants of the proposed parallel model. Lastly we demonstrate the effectiveness of our approach by synthesizing efficient out-of-core FFT programs for the parallel machine CM-5.

In the next section, we present background relating to software support for out-of-core computation.

1.2 Background

The awareness of the importance of I/O subsystems on the overall performance of parallel machines has spurred a large research interest in various aspects of out-of-core applications [24, 16, 78, 29, 69]. In this section, we review these research efforts in the areas of models, program synthesis, file systems, languages, compilers, etc.

1.2.1 Models and Algorithm Design

A model of parallel computation is an abstraction of a parallel machine. An ideal parallel model should be simple enough to support algorithm design as well as contain enough parameters such that the model can be used to accurately predicate the performance of an algorithm on a given machine. For a distributed-memory machine with multi-level memory, an ideal model needs to take both the cost of memory access and the cost of communication into account. However, most of the current models either address only the cost of memory access or the cost of network communications.

Many parallel hierarchical memory models are extensions of sequential hierarchical memory models such as HMM [1] and UMH [6]. These parallel hierarchical memory models, such as P-HMM and P-UMH [84, 63], normally employ simple assumptions about inter-connection networks, such as the processors are connected by a PRAM or by a network which can sort P records in $O(\log P)$ time. Many algorithms for these models have been developed by extending the corresponding algorithms developed for sequential hierarchical memory models. Vitter and Shriver introduced a simpler variant of parallel hierarchical memory models, the two-level memory model [83], which mainly considers the cost of I/O operations. They developed many algorithms such as matrix transposition, sorting, and FFT graph computations for the two-level memory model. They have also proven lower and upper bounds for those algorithms. Cormen developed a class of efficient permutation algorithms such as BPC and BMMC permutations for this model [22].

Historically, the Parallel Random Access Machine (PRAM) is the most widely used parallel model [30]. The PRAM model assumes that all processors work synchronously and that interprocessor communication is essentially free. The PRAM model can be regarded as one extreme which makes a large number of assumptions

in order to simplify algorithm design [43, 70]. However, for many current parallel machines, the PRAM is often inaccurate in predicting the actual running time and resource utilization of algorithms. This problem has spurred the development of several extensions of the PRAM which attempt to make the model more practical while still preserving much of its simplicity. The variations extend the PRAM to incorporate realistic aspects such as *asynchrony* of processes (e.g., the Phase PRAM [33] and APRAM [18]), *communication costs* such as network latency and bandwidth (e.g., the LPRAM [4], the Postal Model [9], BSP [82], LogP [25]), and *memory hierarchy* such as the models discussed in the previous paragraph.

In practice, both the performance of inter-connection networks which connect processors together and memory hierarchies which are attached to each individual processor (or I/O processor) are important for the performance of algorithms developed for a given machine. For example, the inter-connection network of the CM-5 machine is a fat-tree and each processor has a register file, a cache and an internal memory. The experimental results presented in [26] show that to estimate the performance of sorting algorithms for the CM-5 machine, only using the LogP model, which mainly measures the communication latency, can not provide an accurate performance predication. The influence of cache needs to be taken into account for accurately predicating performance. Therefore a parallel hierarchical memory model, which takes both the cost of memory access and the cost of communications into account, is needed to reflect the performance properties of a practical machine.

1.2.2 Methods of Program Synthesis

Various research efforts have studied the problem of synthesizing distributed-memory parallel programs from an algebraic specification of an algorithm. For example, ten-

sor (Kronecker) [35] products have been successfully used for synthesizing programs for block recursive algorithms for various architectures such as vector, shared-memory, and distributed-memory machines [44, 40, 37, 27]. The significance of the tensor product lies in its ability to model the computational structures occurring in a wide spectrum of important applications, as well as the underlying hardware structures such as distributed memories and inter-connection networks.

Other notable examples of program generation include Crystal [15], Algorithmic Skeletons [17], and Powerlists [60]. Crystal is a high-order strongly-typed functional language with an equational form. The program represented in Crystal can be transformed into an efficient distributed-memory program by the recurrence equations. A *skeleton* is a high-level abstraction for a paradigm of algorithm design. For example, the divide-and-conquer method can be represented as a single primitive in a skeleton language. The transformation rules are used to derive efficient implementations on distributed-memory machines. Powerlists, introduced by Misra, can express parallel algorithms with 2-way divide-and-conquer properties by using the succinct constructs $\bowtie$ ($\bowtie$ forms a Powerlist by taking elements from two argument Powerlists alternatively) and $|$ ($|$ forms a Powerlist by simply concatenating two argument Powerlists). The algorithms represented by Powerlists are then mapped to architectures such as Hypercube by program derivations [50].

However, to the best of our knowledge, only tensor products have been used for synthesizing programs cognizant of memory hierarchies. For example, a method of program synthesis for a single disk system is discussed in [49]. However they have not addressed the issues of data distributions on multi-disk systems. In [51], Kumar, Huang, Sadayappan and Johnson discussed the method of program synthesis for cache memory, where they addressed the issue of data layouts on a set-associated cache. Our work presented in this thesis enhances the tensor product framework for

synthesizing programs for multi-disk systems.

1.2.3 Other Work in Parallel I/O

Parallel File Systems

The widely used UNIX file is a sequential stream which is normally accessed by one process at a time. However, in a parallel computing environment, parallel processes tend to access the same file simultaneously and with different (often identical) access patterns. Many current file systems use *file mode* to support these different access patterns. For example, a CMMD_sync_bc call in the CM-5 machine specifies that the same portions of a file can be accessed simultaneously by all nodes, and each node gets the same copy of the data. However, the file is still one dimensional from a user's point of view.

While file modes can be used to support different access patterns, programmers may have to form a different view from a one-dimensional structure. This may result in inefficiencies in disk access. To address this problem, several current file systems, such as the Vesta [21] and the PIOUS [62], provide a high-dimensional view, which can be specified by *block* or *cyclic* layout directives, besides the file modes. A high-dimensional file structure also provides a convenient interface for the development of high performance out-of-core compilers, such as those for the High Performance Fortran (HPF) [31] and the HPC++ [10].

Languages, Compilers and Runtime Support

Most of the research on languages, compilers and runtime support for out-of-core computation is focused on data parallel languages, such as the HPF. While in-core compilers for compiling HPF to distributed-memory machines have been available,

building an out-of-core compiler is an active research topic. Several compilation models for out-of-core computation have been proposed by Kennedy's group at Rice university [65] and Choudhary's group at Syracuse university [12, 13]. There is also research on developing out-of-core C compilers. For example, out-of-core Data-Parallel C developed at Oregon State by Quinn's group [42] and out-of-core ViC* developed at Dartmouth by Cormen's group [23]. The runtime library is important because of the diversity of the interface for parallel file systems. PASSION [79] is a run-time library under developed at Syracuse University. MPI-IO [68] is an extension of MPI for supporting parallel I/O operations.

Several other parallel I/O related researches include disk-directed I/O, two-phase I/O, and I/O characterization of out-of-core computation through tracing and instrumentation [8].

1.3 Summary of Our Approach

In this thesis, we first introduce a model of parallel computation for distributed-memory machines with multi-level memories. Then we present an algebraic method for automatically synthesizing efficient out-of-core parallel programs for block recursive algorithms for both single-processor multi-disk systems and multi-processor multi-disk systems (defined below).

1.3.1 Models for Parallel Machines with Multi-Level Memories

A machine organization for a distributed-memory machine, in which each processor has its own memory-hierarchy, is shown in Fig. 1.1. To model each multi-level memory hierarchy, we need to define the data unit to be accessed by every memory operation, the cost to access every data unit, and the transfer rate of each data unit.

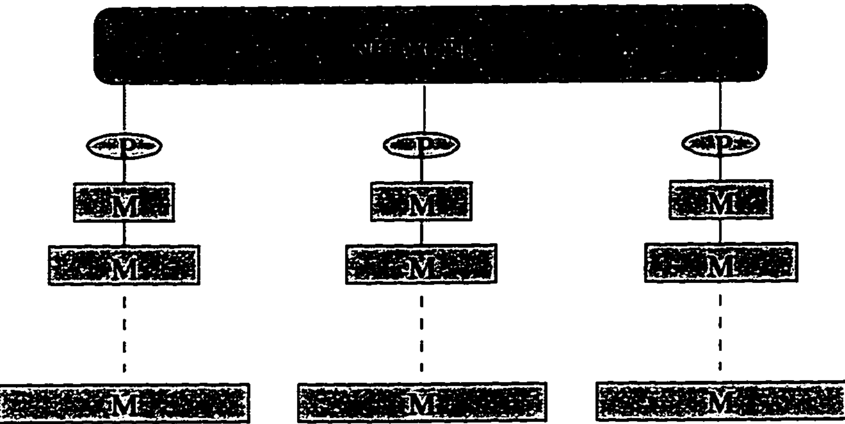


Figure 1.1: Machine model for distributed-memory machines.

To model the inter-connect network, we need to define the cost of inter-processor communications in terms of bandwidth, latency, etc. A model called LogP-HMM, which defines these parameters and takes both the overhead of inter-processor communications and the overhead of accessing multi-level memories, is introduced in this thesis.

We begin with the discussion of various model of parallel computation under the framework of *resource metrics*. We then propose a new hybrid model of parallel computation, the LogP-HMM model. The LogP-HMM model reflects the approach of recent models to abstract architectural details into several generic parameters (or resource metrics). More specifically, the LogP-HMM model is a representative of a class of models formed by combining a network model with any of several existing hierarchical memory models. We also introduce a variant of the LogP-HMM model, the LogP-UMH model, which combines the LogP model with the Uniform Memory Hierarchy (UMH) model. Several near-optimal FFT and sorting algorithms are introduced for both models.

In other part of this thesis, we use two variants of the LogP-HMM model as the

target models for out-of-core program synthesis. These two models mainly capture the performance properties of disk access. They are simpler than the LogP-HMM model. However, they may be closer to the practical distributed-memory machines with secondary storage than the LogP-HMM model.

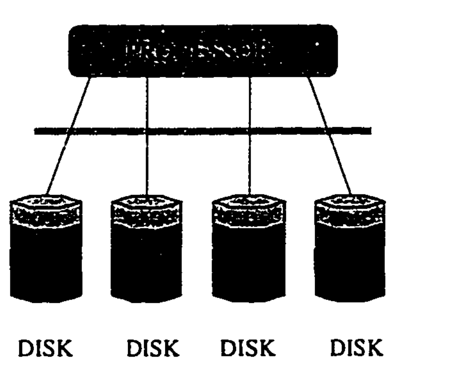
1.3.2 Synthesizing Efficient Out-of-Core Programs

Our method of program synthesis is based on tensor (Kronecker) products. The tensor product is used to represent a class of algorithms with recursive computational structure, known as *block recursive algorithms*. The tensor product representation of a block recursive algorithm, called a *tensor product formula*, is normally organized in phases, each phase is represented by *a (simple) tensor product*. Our method of program synthesis first uses the algebraic properties of tensor products to transform the input tensor product formula into an equivalent formula, such that the computation for the resulting tensor product formula can be implemented efficiently on a given machine. It then synthesizes efficient out-of-core programs for each computational phase in the transformed tensor product formula.

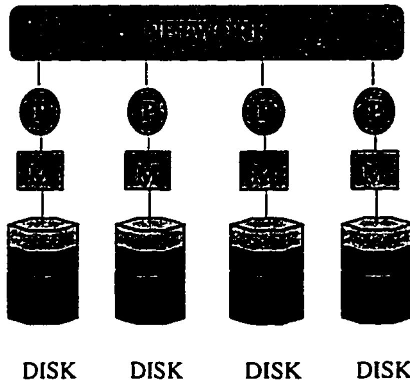
Machine Models for Program Synthesis

In this thesis, we develop methodology for synthesizing programs of the block recursive algorithms for two types of machines which are modeled by two variants of the LogP-HMM model, a single-processor multi-disk model and a multi-processor multi-disk model, respectively. These two models share the common property that out-of-core data are distributed by a block-cyclic data distribution, which is supported by the HPF and modern parallel file systems.

Assume that the last-level memory in Fig. 1.1 consists of disks. If we only con-



(A) Single-processor multi-disk system



(B) Multi-processor multi-disk system

Figure 1.2: Machine models for distributed-memory machines with multi-disks as secondary storage.

sider the last-level memory and we regard the multiple processors as a single processor (In other words, we ignore the cost of inter-processor communications.), then we have a simplification of the LogP-HMM model as shown in Fig. 1.2 (A). By defining data organization on multiple disks appropriately, this model can be viewed as a variant of the two-level memory model defined by Vitter and Shriver [83]. We call this model a single-processor multi-disk model. A refinement of this model is shown in Fig. 1.2 (B), where by exposing both internal memories and disks, we assume that inter-processor communications are not free.

Besides the organization of processors and memory hierarchy, another important issue is how data is distributed (or stored) on multi-level memory hierarchies. For practical distributed-memory machines with secondary storage, the traditional approach provides a one-dimensional view or a sequential file system for data access. All of the architectural details are hidden from users. However, the studies on file-access patterns show that high performance for many applications can not be obtained from this sequential file system. Moreover, the experiments suggest that

a two-dimensional view of file data can largely increase the throughput of I/O subsystems. In our models, data file is striped among multiple processors (or disks) in a block-cyclic manner (defined later). Each processor can also regard the data in its own memory hierarchy as a sub-file. Therefore, our models support two-dimensional file structures.

The single-processor multi-disk model allows us to focus on optimizing the overhead of disk access (or I/O overhead). The multi-processor multi-disk model allows us to further develop multiprocessor programs and to optimize both the overhead of communications and I/Os. The multi-processor multi-disk model is an appropriate approximation for practical distributed-memory machines. For example, in Chapter 8, the methodology of program synthesis for the multi-processor multi-disk model is used to synthesize efficient I/O programs for the parallel machine CM-5.

Methodology of Program Synthesis

As we have mentioned, the block recursive algorithms considered in this thesis are represented by tensor product formulas. We will also introduce a methodology, which is based on a special tensor product called a tensor basis, to capture the semantics of data distributions and access patterns on multi-disk systems.

Fig. 1.3 shows our procedure of synthesizing efficient programs for a block recursive algorithm. It consists of three major steps as denoted by three boxes. The first step takes a tensor product formula and the parameters in the target model as inputs. It then uses either a greedy algorithm or a dynamic programming algorithm and the properties of tensor products to transform the input tensor product formula into an efficient form. It also outputs the data distribution information. The second and the third steps will be applied to each computational step. Since each computational step is represented by a tensor product, the second and the third steps will synthesize code

for each tensor product. An outermost loop structure will then be used to construct the final program for the overall tensor product formula. More specifically, the second step decomposes the computation of each tensor product into sub-computations by analyzing data access patterns and exploiting locality and concurrency. The results of these analyses are represented as an *augmented tensor basis*. The third step of code generation will use the augmented tensor basis to generate the final parallel I/O programs.

However, we use the reverse order to present the steps in the derivation of efficient implementation for a tensor product formula. We first present a procedure for code generation by using the information contained in the augmented tensor basis. Then, we determine an efficient implementation for a simple tensor product with a given data distribution on a given model. Lastly, we present a greedy or a dynamic programming algorithm to determine an efficient implementation for the tensor product formula. The reason we use this order of presentation is that the program transformation needs the performance of each tensor product, which we will discuss with the second step of deriving efficient programs for a tensor product.

For the single-processor multi-disk system, the second and the third steps of synthesizing efficient programs for each individual computational step include determining efficient data access patterns, partitioning computations for each memory-load (defined in Chapter 5), and generating parallel I/O operations for accessing out-of-core data. Our method allows a general *logical* block-cyclic distribution, instead of the normally used *physical track* distributions. We further use a dynamic programming algorithm to determine the efficient tensor product transformations and the efficient data distribution for the tensor product formulas.

A similar procedure is used for synthesizing efficient programs for the multi-

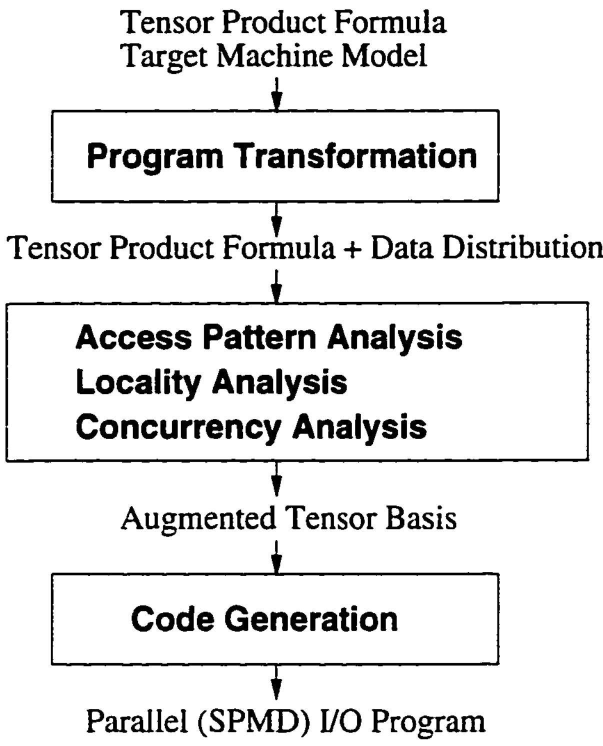


Figure 1.3: Procedure for synthesizing I/O programs for block recursive algorithms.

processor multi-disk model. The major differences are as follows. We now want to optimize the overhead including both I/Os and in-core computation, which includes in-core communications. We take the approach of first optimizing the I/O overhead and then using known approaches to optimize in-core computation. In order to achieve this goal, in the second and the third step, we need further to determine data distributions inside the internal memories for each sub-computation, to partition sub-computations further for each processor, and to synthesize node programs. For the tensor product formula, the method of minimizing I/O overhead uses a greedy algorithm with a prediction function, which is dependent on the number of passes over out-of-core data and the I/O costs in each pass, to determine the efficient tensor

product transformations and the efficient data distribution.

The effectiveness of our approach is demonstrated by synthesizing out-of-core parallel fast Fourier transform (FFT) programs on the CM-5. We use the CM-5's parallel file system, the Scalable File System which is based on the Scalable Disk Array, to simulate the two-dimensional file structure specified by our approach of program synthesis. Experimental results show that, by using the parallel file access, a speedup of up to ten can be obtained for the synthesized FFT programs over the serial file access.

1.4 Contributions of This Thesis

The principal contributions of this thesis are summarized as follows.

- A framework based on *resource metrics* for characterizing various models of parallel computation. Chapter 2 defines resource metrics and examines various models based on the concept of the resource metrics.

- A generic model, the LogP-HMM model, for modeling parallel machines with multi-level memory and a communication network. Chapter 3 discusses this model and introduces several near-optimal FFT and sorting algorithms for both this model and a variant of this model.

- A methodology based on tensor bases for capturing the semantics of data distributions and data access patterns on multi-disk systems. Chapter 4 introduces tensor products. Chapter 5 presents how to describe data organization and data access patterns on multi-disk systems by tensor bases.

- A method for synthesizing efficient parallel I/O programs for striped two-level memory model with various block-cyclic data distributions. Chapter 6 presents

the method of program synthesis by tensor products. We also investigate the implications of using various block-cyclic distributions, instead of the normally used *physical track* distributions, on the performance of the synthesized programs.

- A method for synthesizing efficient out-of-core parallel programs for multiprocessor multi-disk systems. Chapter 7 generalizes the results in Chapter 6 to handle multiple processors. However, here we do not use the concept of physical tracks. Instead, we use a logical track as a unit of a parallel I/O operation.

- Implementations. In Chapter 8, we apply the methodology developed in Chapter 7 to the CM-5 machine. A two-dimensional view of data file is constructed from the file stored on the CM-5's Scalable Disk Array. A FFT algorithm synthesized based on the method developed in Chapter 7 is implemented. The performance of the synthesized program is then discussed.

We conclude in Chapter 9 with discussion of future research directions.

Chapter 2

Models and Resource Metrics for Parallel Computation

While the emergence of a large number of highly parallel computers have vastly increased the potential for solving large-scale scientific and engineering applications, the effective design and implementation of algorithms for them remains problematic. One challenge is the lack of a general model of parallel computation, which balances being sufficiently detailed to reflect realistic aspects impacting performance while still remaining abstract enough to be machine-independent and amenable to analysis [74, 34].

Historically, the Parallel Random Access Machine (PRAM) is the most widely used parallel model [30]. The PRAM model assumes that all processors work synchronously and that interprocessor communication is essentially free. However, for many current parallel machines, the PRAM is often inaccurate in predicting the actual running time and resource utilization of algorithms since it hides details which impact performance such as the time required for network communication as well as issues of asynchrony and memory hierarchy. This problem has spurred the development of several extensions of the PRAM which attempt to make the model

more practical while still preserving much of its simplicity. The variations extend the PRAM to incorporate realistic aspects such as *asynchrony* of processes (e.g., the Phase PRAM [33] and APRAM [18]), *communication costs* such as network latency and bandwidth (e.g., the LPRAM [4], Postal Model [9], BSP [82], and LogP [25]), and *memory hierarchy*, reflecting the effects of multileveled memory such as differing access times for registers, local cache, main memory, and disk I/O (e.g., the P-HMM [84], PMH [7], and P-UMH [63]).

The approach followed by these extended models is that of a *parameterized* (or generic) model, which abstracts the architectural details into several generic parameters which we call *resource metrics*. Typical resource metrics include the number of processors, communication latency, bandwidth, block transfer capability, network topology, memory hierarchy, memory organization, and degree of asynchrony. Using such a parameterized model, one can design broadly applicable parameterized algorithms that can be tailored to specific machines by instantiating the parameters, such as latency and bandwidth, to match machine characteristics. The more recent models typically try to use more parameters to more finely capture the resource characteristics of parallel machines. However, a careful balance must be struck between incorporating detail and being too finely parameterized (in too many dimensions) so as to render optimal algorithm design impossible. Therefore, identifying resource metrics and appropriately choosing them is critical in the design of models of parallel computation.

In this chapter, we identify resources and resource metrics which are important for the performance of parallel machines and use them as a framework to characterize the variety of parallel models. Within this framework we will categorize models into four classes: basic synchronous models, asynchronous models, models which incorporate notions of latency and bandwidth, and models which address hierarchi-

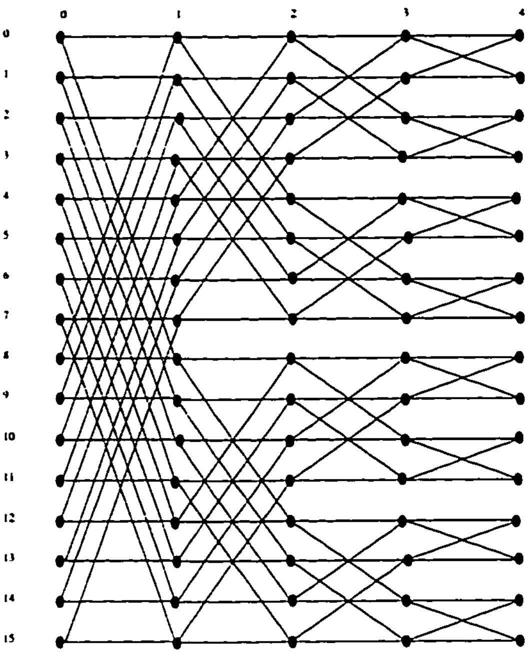


Figure 2.1: A 16-input FFT graph.

cal memory. The models discussed here are generally in an increasing order of the number of resource metrics considered.

Throughout the discussion, we use the core fast Fourier transform (FFT) computation to illustrate the principles of algorithm design and complexity analysis for many of the different models. The data movement of the core FFT computation forms a butterfly graph, also called an FFT graph. The N-point FFT graph with $N = 2^m$ can be defined as follows: there are N input and output points denoted as $x_{0,0}, x_{0,1}, \ldots, x_{0,N-1}$ and $x_{m,0}, x_{m,1}, \ldots, x_{m,N-1}$ respectively. The computation is often called pebbling and is denoted as $x_{j,q} = f(x_{j-1,q}, x_{j-1,r})$, where f is a constant cost function and the only difference between q and r is in the $(j-1)$th position when both are represented as binary numbers. A 16-input FFT graph is shown in Figure 2.1, where the left-most column contains the input data and the right most column denotes the

output.

2.1 Resource Metrics

We first present definitions of resources, resource metrics, and models of parallel computation.

Definition 1.1 *A* resource *refers to an architectural feature that significantly affects the performance of a parallel machine. A* resource metric *is a measure of the corresponding resource, which could be quantitative or qualitative. The value of a quantitative resource metric is normally a multiple of the unit processor execution time.*

Definition 1.2 *A* computational model *is an abstraction of a computing machine which is characterized by the choice of several resources and the corresponding resource metrics. A computational model may thus be identified with a set of resource metrics. Moreover, algorithms will be designed and analyzed based on these resource metrics.*

For example, a sequential computer is suitably characterized by the resources of sequential computation time and space usage. It is commonly accepted that the sequential computational models, such as RAM and its hierarchical memory extensions HMM, BT_f and UMH [1, 2, 6], reflect these resources quite well and therefore provide a common base for sequential computation. Resource metrics for a practical parallel machine are far more complicated than those of sequential machines. We identify the following list of significant resources and resource metrics for parallel computation:

Processors. The resource of processors can be captured by the following two resource metrics.

- Number of processors P. A theoretical model normally assumes that there is an unlimited number of processors available, while a more practical model assumes a bounded number of processors, such as hundreds or thousands.
- Degree of asynchrony. Processors may run synchronously, semi-synchronously (loosely synchronously) or asynchronously. Semi-asynchrony refers to a computation that is divided into a sequence of independent execution phases; within each phase, the program runs asynchronously, but all of the processors are synchronized at the end of each phase (i.e., barrier synchronization occurs).

Memory organization. Machines may be characterized as having physically shared memory or distributed memory. The former often has a local cache. The latter typically uses explicit message passing primitives.

Communication network. The performance properties of communication network can be captured by the following four parameters.

- Communication latency. The costs of accessing local memory and global memory in a shared memory machine are quite different; similarly, in a distributed memory machine, the costs of accessing local memory and communication with other processors are quite different. Latency is one measure of the cost of global memory access.
- Bandwidth. Communication bandwidth and memory bandwidth are both limited in practice. Currently, communication bandwidth lags far behind internal

processor memory bandwidth. The bisection bandwidth, defined as the bandwidth across a line that separates the network into two approximately equal parts, is normally used for the bandwidth resource metric.

- Overhead of a processor for message handling. The communication overhead is the time that the processor engages in sending and receiving a message. In most cases, the value is dependent on the communication protocol implemented in a practical machine. For example, in the CM-5 it is a linear function of the message size [25].
- Block transfer capability. In most architectures, a significant cost (latency) is incurred to access the first of a contiguous block of words, but after that, successive words can be accessed in unit time.

Memory hierarchy. Many sophisticated machines have several layers of memory with differing access times, such as register, cache, main memory and secondary memory. A model capturing this memory hierarchy can organize the memory as a tree or a sequence of layers with increasing sizes, where each node or level is parameterized by memory size, block size, and inter-module bandwidth.

Memory contention. Memory can be accessed as a block or a set of banks. Protocols for resolving conflict in concurrent memory access include EREW, CREW, CRCW and QRQW.

Network topology. The processors may be interconnected using a mesh, cube, fat tree, ring, or other topology. The most common metric for this resource is the diameter of the network.

In the subsequent sections we present a detailed discussion of each model and its choice of resource metrics.

2.2 Basic Synchronous Models

2.2.1 PRAM

The PRAM model extends the sequential RAM model by replicating the processor part. A PRAM machine is a set of sequential processors sharing a global memory and each having its own private unbounded local memory, as shown in Figure 2.2. A PRAM computation is a sequence of read, write and computation steps. All processors execute in lock-step, that is, they are synchronized before they execute the next instruction. The costs of memory access, either to local or global memory, and computation steps are uniform. The cost of synchronization is free. Several variations of the PRAM model use different protocols to handle simultaneous access of several processors to the same location of global memory. Protocols include EREW (exclusive read - exclusive write), CREW (concurrent read - exclusive write), and CRCW (concurrent read - concurrent write). The latter protocol can be further divided into several classes by the semantics of the concurrent write. The most recent variation is QRQW [32], which assumes that simultaneous access to the same memory block will be inserted into a request queue and served in a FIFO manner.

2.2.2 VRAM

Another extension of the serial RAM model is the Vector Random Access Machine (VRAM) [11]. The VRAM is a serial random access machine with the addition of a vector memory, a vector processor, and vector input and output ports. Typical vector

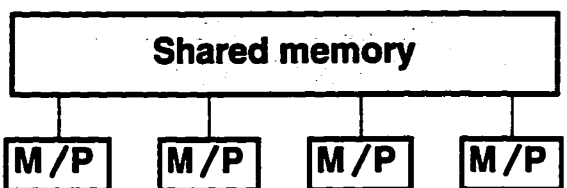


Figure 2.2: The PRAM model.

instructions include elementwise operations, data movement operations, scans, and packs.

Two measures, step and element complexity, can be derived for a problem of size N in the VRAM model. Step complexity (s) is the total number of instructions executed and element complexity (e) is the vector length per primitive instruction call, summed over the number of the calls. These values give a measure of the parallel time and the total work respectively. The PRAM complexity of an algorithm designed in the VRAM model can be derived as $O(e/P + s)$, where P is the number of processors in the PRAM model.

The PRAM has proven to be useful by permitting algorithm designers to focus on the structure of computational tasks rather than the architectural details of a currently available machine. A large number of efficient algorithms have been developed by exploiting its simplistic assumptions, but in practice, some of the architectural issues that the PRAM ignores are important.

2.3 Extensions of the Basic Models

The PRAM model provides an abstraction that ignores concerns such as asynchrony, communication delay, and memory hierarchy. In this section we discuss several ex-

tensions of the PRAM model which incorporate some of these measures. The extensions may be viewed as the addition of more resource metrics to the PRAM model in order to gain improved performance measures.

2.3.1 Asynchronous Models

Among the first extensions to the PRAM were the Phase PRAM and APRAM models, which incorporate some notion of asynchronous execution.

Phase PRAM

The Phase PRAM [33] extends the PRAM model with semi-asynchrony. A Phase PRAM machine consists of a shared global memory, a set of P sequential processors, and a private local memory for each processor. The computational task is separated into a set of phases of asynchronous execution, each ended by an explicit barrier synchronization. The cost of global read, global write and local operations are the same constant. The cost of a synchronization step, $B(P)$, is dependent on the number of processors P. As an example of Phase PRAM algorithm design, below we present an algorithm for FFT computation from [33]. It is worth noting that a variant of the Phase PRAM, the Phase LPRAM model, accounts as well for the cost of communication latency.

FFT for Phase PRAM. The algorithm presented in [33] for the EREW Phase PRAM divides the columns of an N-input FFT graph into $\log N/\log B$ stages, each consisting of $\log B$ columns. It also divides the rows into N/B sets, each consisting of B rows. The computation is performed from the first stage to the last stage. In order to get a cost-efficient algorithm, $(N \log B)/B$ processors are used. In other words, we choose $P = (N \log B)/B$. In each stage, each set of the rows will be

assigned $\log B$ processors. Therefore the time to compute one set in each stage is $O(B \log B)/ \log B$, which is equal to $O(B)$. The total time, including the synchronization cost after each stage, is $O(B \log N/ \log B) = O((N/P) \log N)$, which is asymptotically optimal.

APRAM

The Asynchronous PRAM (APRAM) is a "fully" asynchronous model [18, 19]. The APRAM model consists of a global shared memory and a set of processes with their own local memories. The basic operations executed by the APRAM process are called events. An APRAM computation is denoted as the set of possible serializations of events executed by the processes. A virtual clock is associated with each serialization. This virtual clock assigns a time $t(e)$ to each event e. The clock "ticks" when each process has executed at least one event. Events may be read and write events, which operate on the shared and local memory, or local events. All events are charged unit cost.

The pair [round complexity, number of processes] is used to measure the complexity of an APRAM algorithm, where a round is defined as the sequence of events between two clock ticks in a computation. The round complexity for a computation is defined to be the maximum number of possible ticks for that computation. For an algorithm the round complexity is defined as the maximum round complexity over all of the possible computations.

Summation for APRAM. As an example of APRAM algorithm design we present an example of summation from [18]. As in the PRAM model, a binary tree is used to sum the N input values, which initially reside at the N leaves. Each node in the tree contains an extra valid bit, which initially holds the true value for the input nodes

and false values for all other nodes. N processes are used and with each process p are also associated three variables $V(p)$, $L(p)$ and $R(p)$, which denote the value of p, the left child of p, and the right child of p respectively. The algorithm for process i, as described in [18], consists of two steps: (1) wait until $L(i)$ and $R(i)$ are valids, which means that their associated valid bits become true; (2) set $V(i) := L(i) + R(i)$ and $valid(i) := true$. The algorithm terminates when the valid bit at the root of the tree is true. After the valid bit associated with the two children of a process is true, a process needs at most five rounds to validate the variable associated with it. Therefore, the round complexity of this algorithm is $O(\log N)$. Using the same strategy as for the PRAM, an algorithm with the same round complexity but using $N/\log N$ processes can be easily obtained.

2.3.2 Models Incorporating Communication Latency and Bandwidth

The models discussed here consist of two subclasses: the synchronous latency models such as the LPRAM and BPRAM which add the notion of latency into the PRAM model, and models such as the BSP and LogP which not only incorporate asynchrony and latency but also address the issue of bandwidth limitation.

LPRAM

The Local-Memory PRAM (LPRAM) model [4] consists of a shared global memory and a set of processors with unbounded local memory executing in lock-step. The access protocol to global memory is CREW. At every time step, each processor can perform either a communication step, in which it can write and then read a word from the global memory, or a computation step, which is an operation that accesses at most two words from its local memory.

BPRAM

An extension of the LPRAM, the Block PRAM (BPRAM), is described in [3]. The BPRAM takes into account the reduced cost for transferring a contiguous block of data. The BPRAM model is defined with two parameters l (latency or startup time) and P (number of processors). The cost of accessing local memory is unit time. However the cost of transmitting a block of size b of contiguous locations from global memory is $l + b$.

We present the following FFT algorithm for the BPRAM model from [3]. We follow the EREW assumption and the input and the output are both stored in global memory. The pair [$time$, $work$] is used to measure the complexity of the LPRAM algorithms. Let T_M denote the minimal running time and w_M denote the work. Then the relation $w_M = PT_M$ holds for a P-processor machine.

FFT for BPRAM. The FFT algorithm consists of two cases.

1. $lP \leq N$: The FFT graph is simply computed from the first level to the last level and the data is transmitted in blocks of size l. Because $lP \leq N$, each level needs at most $O(N)$ work. Therefore, the total work is $O(N \log N)$, which is optimal.

2. For $lP > N$, the block layout method is used. The size of the block is 2^k, where $N/P \leq 2^k \leq l$. Therefore, the N-input FFT graph is separated into $(\log N)/k$ stages and each stage has $N/2^k$ 2^k-input FFT subgraphs. The computation is performed from stage to stage. At each stage, the processors compute the 2^k-input FFT. Because $N/2^k \leq P$, each stage can be completed in $O(l + k2^k)$ time. Therefore all stages can be completed in $O(((\log N)/k)(l +$

$k2^k$)) time. The permutation is needed to rearrange the data after each stage and this permutation can be done in time

$$O(\frac{l \log(min(l,P))}{\log(lP/N)})$$

by [3]. In total, $(\log N)/k$ permutations are needed. If we take

$$k = \log(max(\frac{l}{\log l}, \frac{N}{P}))$$

then it yields work

$$O(N \log N + lP + \frac{lP \log N \log(min(l,P))}{\log l \log \frac{lP}{N}})$$

Postal Model

The Postal model [9] is a distributed memory model with the constraint that the point-to-point communication has latency λ. It can be regarded as a model described by two parameters: $< P, \lambda >$, where P stands for the number of processors. Several elegant optimal broadcast and summation algorithms have been designed based on this model, which were then extended for the LogP model [46]. Algorithms other than broadcast and summation have largely not been presented for this model.

Bulk-Synchronous Parallel Model

The BSP is a distributed memory model [82]. Like the Phase PRAM, the BSP is also a semi-asynchronous model because it requires synchronization after each "superstep", within which the processes can run asynchronously. The BSP model is described in terms of three elements: processor/memory modules, a router which delivers the messages between pairs of components, and a synchronizer which synchronizes all or a subset of the components. Within the framework of resource metrics,

a router is just an abstraction of network bandwidth and latency. A computational task in the BSP model consists of a sequence of supersteps. In each superstep, every component is allocated a task which contains some combination of local computation steps and message transmissions. The local computations, including reads and writes to local memory, are charged unit cost. The message transmission is accomplished by the router, which can send and receive a certain number of messages in each superstep (or in BSP terminology, the router can realize an h-relation). The cost of realizing such an h-relation is assumed to be $gh + s$ time units, where g can be thought of as the reciprocal of the communication bandwidth and s denotes the startup cost or the latency. If the length of a superstep is L, then L local operations and a $\lfloor L/g \rfloor$ - relation message pattern can be realized. The parameters of the machine are therefore L, g and P (the number of processors). Below we present a FFT algorithm for the BSP as described in [82].

FFT for BSP. The algorithm divides the FFT graph into successive layers where each layer is a superstep. In each layer, the FFT graph is separated into a set of small input size FFTs. Each of them is computed by a different processor. Let us assume that the number of layers is $\log N/\log d$ and each layer consists of $(N \log d)/d$ independent butterfly graphs of $d/\log d$ inputs each (assume that $d \geq 2$ and the expressions are bounded to integers appropriately). Therefore, the graph can be computed on $P = (N \log d)/d$ processors in $\log N/\log d$ supersteps, in each of which each processor computes d local operations and sends and receives $h = d/\log d$ messages. When we take $g = O(\log d) = O(\log(N/P))$ and $L \leq d = O((N/P)\log(N/P))$, the cost of each superstep is $O(d)$, since $h = d/\log d$. The running time of the algorithm is $O(d \log N/\log d) = O((N/P)\log N)$, since $d/\log d = N/P$, which is optimal.

LogP Model

The LogP model is motivated by current technological trends in high performance computing towards networks of large-grained sophisticated processors. The LogP model uses the parameters L (an upper bound of latency for transmitting a single message), o (the computation overhead of handling a message), g (a lower bound of time interval between consecutive message transmissions at a processor) and P (the number of processors) [25]. In contrast to the BSP model, it removes the barrier synchronization requirement (h-relation in BSP) and allows the processors to run asynchronously. The network of a LogP machine has a *finite capacity* such that at any time at most $\lfloor L/g \rfloor$ messages can be in transit from or to any processor. It can support shared or distributed memory.

The LogP model encourages well-known general techniques of designing algorithms for distributed memory machines including exploiting locality, reducing communication complexity, and overlapping communication and computation. The LogP model also promotes balanced communication patterns by introducing the limitation on network capacity so that no processor is overloaded with incoming messages. Moreover, it is often reasonable to ignore the parameter of o in a practical machine, such as in a machine with low bandwidth (high g value). Examples using this strategy can be found in the FFT algorithm discussed below and also in [46].

FFT for LogP. The data layout and communication scheduling are two key aspects to achieving an efficient algorithm for the FFT problem under the LogP model. Three methods of data layout are discussed in [25]. The cyclic layout assigns the ith row of the butterfly to the ith processor. The block layout places the first N/P rows on the first processor, the next N/P rows on the second processor, and so on. Un-

der either layout, each processor spends $(N/P)\log N$ time computing and sends and receives N/P messages, which needs $(gN/P + L)\log P$ time. The third method is called hybrid layout, which switches from cyclic to blocked layout at any column between the $\log P$-th and the $\log(N/P)$-th (assuming $N > P^2$). With this layout, each processor sends N/P^2 messages to every other processor, requiring only $g(N/P - N/P^2) + L$ time. Therefore, this method leads to an algorithm with running time $O((N/P)\log N + (N/P - N/P^2)g + L)$, which is within a factor $(1 + g/\log N)$ of optimal.

The naive communication schedule stalls on the first send. The technique of overlapping computation and communication can be used to eliminate this stall for large problem instances. The hybrid method introduced in [73] staggers the different starting rows for different processors: processor i starts with its iN/P^2-th row, proceeds to the last row, and wraps around. This leads to an optimal algorithm for large problem instances and reasonable g value.

Candidate Type Architecture

The Candidate Type Architecture (CTA) [75] is an earlier precursor to parameterized latency models which presents a more direct abstraction of a practical parallel machine. The CTA consists of a front-end and a back-end. The front-end works as a control processor; the back-end is a MIMD machine. The MIMD machine consists of a set of independent sequential machines, which are connected by an abstract network. The network has a limited bandwidth and introduces a latency L for message transmission which is dependent on the network topology. Therefore, the CTA model encompasses a large set of resource metrics. Moreover, the resource metrics in CTA are qualitative and therefore the performance analysis of an algorithm is dependent on the interconnection network. Rigorous analysis of algorithms is generally

difficult to accomplish within this model due to the richness of the resource metrics and their qualitative properties.

2.3.3 Hierarchical Models

The Parallel Memory Hierarchy model (PMH) [7] and Parallel Hierarchical Memory model (P-HMM) [84] discussed in this section address the concerns of memory hierarchy in a parallel setting. The P-HMM primarily originates from considering in a parallel network the existence of secondary or disk memory. PMH on the other hand uses "memory hierarchy" as a more general technique to model not only the hierarchy within a processor but also the communication characteristics of a parallel machine.

Parallel Memory Hierarchy Model

The PMH is a so-called generic model which defines a class of specific models [7]. In the PMH model, a parallel computer is modeled as a tree and each node of the tree is called a *module*. All of the leaf modules are used to denote the processors and the internal modules hold the data. A child module is connected to its parent by a unique channel with a certain amount of bandwidth. Data in a module is partitioned into blocks which are the basic unit of data transfer between the child and parent. Thus communication between two processors proceeds up the tree by some path and then down the tree to the target processor.

The model is characterized by four parameters specified for each module m: *blocksize* s_m, *blockcount* N_m which denotes the number of blocks in each module, *childcount* c_m which denotes the number of children for each module, and *transfer time* t_m which denotes the number of cycles used to transfer a block between the current module and its parent. To model a particular computer, one chooses a tree structure

and values for the parameters appropriate for the machine's communication capabilities and memory hierarchy. Even though this model is termed a parallel memory hierarchy, the internal modules need not necessarily correspond to actual memory modules of the real machine; when modeling the CM-5 for example, many of the modules are used to capture the interprocessor communication capabilities [7].

Many sequential algorithms have been developed for the antecedent sequential UMH model. However, perhaps because of the complexity and generality of the PMH model, not too many algorithms have been developed for the model.

Parallel Hierarchical Memory Model

The P-HMM model is also called the parallel I/O model [84]. It originates from the consideration that data must often reside on secondary storage rather than main memory; in a parallel setting this may involve the parallel access of multiple disks. Therefore it is necessary to design parallel algorithms which consider the possible data movement between main and secondary memory [5], and more generally which consider multiple levels of memory including register and cache.

In the P-HMM model, each processor has a memory hierarchy organized into discrete levels, much like the memory organization in the HMM, and all of P separate memories are connected together at the base level of each hierarchy. A further assumption is that the P hierarchies can each function independently. Communication between hierarchies takes place at the base memory level (level 0) which consists of location 1 from each of the P hierarchies. The interconnection network for the P base memory level locations is normally assumed to be a hypercube (or cube-connected cycles) so that the P records in the base memory level can be sorted in $O(\log P)$ time. The model can be extended to allow block transfer; the resulting model is called the P-BT model [84]. Several other extensions which use the UMH

memory model and PRAM interconnection respectively are discussed in [63].

Two factors are critical for developing efficient P-HMM algorithms: data placement and movement between the levels of memory hierarchy, and data movement among the processors.

FFT for P-HMM. We present the following P-HMM algorithm from [84], computing the N-input FFT when $N \geq P^2$.

1. Compute $\sqrt{N}$ $\sqrt{N}$-input FFTs. Assume that the ith FFT is on the ith contiguous group of $\sqrt{N}/P$ tracks.
2. Shift the records in the kth hierarchy to hierarchy $1 + (k+\mathit{offset}-1) \bmod P$, where $\mathit{offset} = (i-1) \bmod P$ for the ith group.
3. Shuffle the records to form $\sqrt{N}$ new contiguous groups of $\sqrt{N}/P$ tracks. For every $1 \leq i \leq \sqrt{N}$, the ith new group consists of the ith record from each of the original $\sqrt{N}$ groups.
4. Do $\sqrt{N}$ $\sqrt{N}$-input FFTs for the new groups.

When $P \leq N \leq P^2$, we first do N/P P-input FFTs, followed by a shuffle-merge, and then do N/P-input FFTs.

The time required by this algorithm, $T(N, P)$, is explained as follows. Steps 1 and 4 take $\sqrt{N}T(\sqrt{N}, P) + (N/P)\log(N/P)$ time. The first term in this formula is easy to understand. The second term arises because the different $\sqrt{N}$ $\sqrt{N}$-input FFTs sit in different memory levels. We need to move them into the lowest memory levels in order to recursively compute the $\sqrt{N}$-input FFT. Similarly, the shifted elements need to be brought to the base memory and to be transmitted by the network

which connects the base memory. Therefore, the shift time is $O((N/P)(\log P + log(N/P)))$. The shuffling can be done in the order of input size times the data movement in the hierarchy, which is $O((N/P)\log(N/P))$. Therefore, we have following recurrence

$$T(N,P) = 2\sqrt{N}T(\sqrt{N},P) + O(\tfrac{N}{P}\log N),$$

which gives the result $O((N/P)\log N \log(\log N/\log P))$. This matches the FFT lower bound in [84] and so is optimal.

2.4 Conclusions

This chapter presented a framework of using resource metrics to characterize various models of parallel computation. Using the properties of resource metrics, we can classify models into the basic synchronous models and extensions which incorporate notions of asynchrony, communication costs, and memory hierarchy. The merits and disadvantages of these models are brought out by examining their characteristics in terms of resource metrics and their utility in designing such algorithms as FFT.

An examination of the resource metrics chosen by various models, as summarized in Table 2.1, reveals a lack of appropriate models that accurately treat both network communication and multi-level memories. For example, the LogP model does not address the problem of several layers of memory [1]. Yet it is important to model the several layers of memories which exist in many machines, since differing access times to local cache and disk may strongly affect performance. On the other hand, models such as the P-HMM model or the P-UMH model address in detail the

[1] Even though the authors of the LogP model mentioned that cache affects the performance of the FFT algorithm, they concluded that the effect is small compared with the carefully scheduled communication. But that may not always be true especially if one considers disk memories, which have greater speed difference from main memory than that of cache.

	Proc	Synchrony/ asynchrony	Memory Organization	Latency	Bandwidth	Block Transfer	Overhead	Memory Hierarchy
PRAM	P	Synchronous	Shared					
VRAM	1	Synchronous	Vector					
LPRAM	P	Synchronous	Shared	√				
BPRAM	P	Synchronous	Shared	√		√		
Postal	P	Asynchronous	Distributed	√				
PHASE PRAM	P	Semi-asynch	Shared					
PHASE LPRAM	P	Semi-asynch	Shared	√				
APRAM	P	Asynchronous	Shared					
CTA	P	Asynchronous	Distributed	√	√	√		
BSP	P	Semi-asynch	Distributed	√	√			
LogP	P	Asynchronous	Both	√	√		√	
PMH	P	Asynchronous	Distributed	√	√	√	√	√
P-HMM	P	Asynchronous	Distributed					√
H-PRAM	P	Semi-asynch	Both	√				
LogP-HMM LogP-UMH	P	Asynchronous	Distributed	√	√		√	√

Table 2.1: Resource metrics chosen by different models of parallel computation.

resource of memory hierarchy, but not so much the accurate characterization of the communication network. These models normally use simple assumptions about the network, such as a PRAM connection or an abstracted hypercube connection. The recently proposed PMH model contains the resource metrics for both network and memory hierarchy. However, because its generality and complexity, it is not appropriate for algorithm development and analysis.

As an example of the process of developing improved computational models in the framework of resource metrics, in the next chapter, we propose a new hybrid model of parallel computation, the LogP-HMM model. We demonstrate the potential of the LogP-HMM model on serving the design and analysis of practical algorithms for a class of machines by developing several near optimal sorting and FFT algorithms, and by matching the model to practical parallel machines.

Chapter 3

Models for Distributed Hierarchical Memory Machines

As we discussed in Chapter 2, each of the existing models of parallel computation addresses differing aspects of resource usage, which suggests that resource metrics are an appropriate tool to categorize and help to understand different models of parallel computation. We also mentioned that there is a lack of appropriate computational models that accurately treat both network communication and multi-level memories. In this chapter, we introduce the LogP-HMM model, which can treat both network communication and multi-level memories as well as can be used for algorithm design and analysis.

The LogP-HMM model, pictured in Figure 3.1, extends an existing parameterized network model (the LogP, with resource metrics of latency, bandwidth, and overhead on handling messages) with memory hierarchy at each processor (there following the sequential hierarchical memory (HMM) model). The LogP-HMM represents a pragmatic refinement of parallel computational models within the framework of resource metrics to accommodate more detailed performance measures.

More generally, the LogP-HMM model is representative of a class of models formed by combining a network model with any of several existing hierarchical mem-

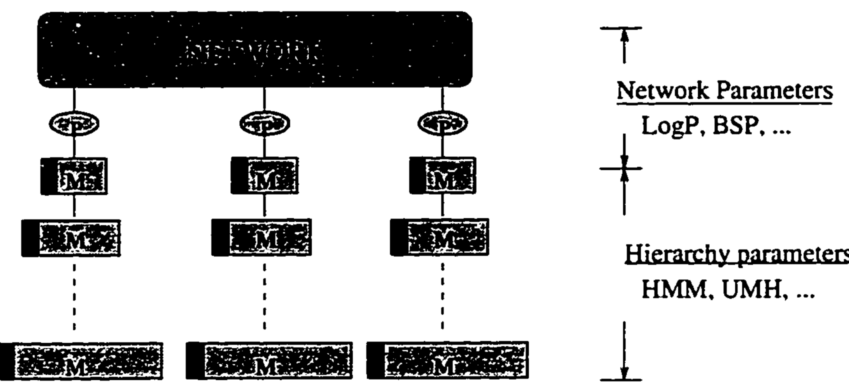


Figure 3.1: Structure of the LogP-HMM model.

ory models. Based on this idea, we introduce a variant of the LogP-HMM model, the LogP-UMH model, which combines the LogP model with the Uniform Memory Hierarchy (UMH) model. We discuss the potential of the LogP-UMH model to more accurately reflect parallel machines by matching the model to the CM-5 and the IBM SP-2. We examine the potential utility of both the LogP-HMM and the LogP-UMH models in the design of several near optimal FFT and sorting algorithms. It turns out that one of the near optimal FFT algorithms, which is based on the hybrid data layout, can run optimally in the P-HMM model. Our approach of using the LogP-HMM model (and its variant the LogP-UMH) in the design of FFT algorithms has broader applicability to the more general class of *ascend*-and-*descend* algorithms [67] (whose data movement follows an FFT graph), used for solving such problems as sorting, polynomial multiplication and convolution.

3.1 The LogP-HMM Model

The LogP-HMM model, discussed in this section, consists of two parts: the network part and the memory part. The network part is captured by the LogP model, while

the memory part of each node is described by the HMM model.

3.1.1 Definition of the Model

The LogP-HMM model is defined much like the parallel hierarchy memory model [84]. A LogP-HMM machine consists of a set of asynchronously executing processors, each with an unlimited local memory. The local memory is organized as a sequence of layers with increasing size, where the size of a memory block is 1 and the size of layer i is 2^i. Each memory location can be accessed randomly. The cost of accessing a memory location at address x is $\log x$ (using access cost function $f(x) = \log x$). The processors are connected by a LogP network at level 0. In other words, the four LogP parameters, L, o, g, and P, are used to describe the interconnection network. A further assumption is that the network has a finite capacity such that at any time at most $\lfloor L/g \rfloor$ messages can be in transit from or to any processor.

3.1.2 Algorithm Design and Analysis

Exploiting locality is the key for designing efficient algorithms for the LogP-HMM model. In LogP-HMM, there are two potential sources of data locality: the network part and the memory part. In the network part, we want to distribute data in such a manner that each processor will use the data extensively before it needs the data in other processors. Similarly, in the memory part, we want to maximize the probability that before we move data to higher memory levels, the data have been used and may not be needed after moving. Several algorithms presented below exploit these ideas. These algorithms assume one of two different approaches to memory layout. The first, which we call the *global* (or *track-oriented*) view, assumes that each block of b records from each processor with the same relative position in a level of memory

hierarchy form a *track*, and organizes the algorithm around this unit of management. This addressing scheme might be more appropriate to a (distributed) shared memory system. Note that for the LogP-HMM model, we assume b=1 if we take this global view of the address space. The second approach, which we call the *local* view, treats each memory hierarchy as its own address space, and may be more appropriate to distributed memory systems. Before we present the algorithms, we first prove the following theorem.

Theorem 1.1 *The lower bound of sorting $N \geq P$ elements (or computing an FFT graph) in the LogP-HMM model with memory access cost $f(x)=\log x$ is*

$$\Omega(\tfrac{N}{P} \log \tfrac{N}{P} \log \log \tfrac{N}{P}).$$

Proof: In a LogP-HMM machine with P processors, if we divide N elements into P sets of equal size and place the elements among the processors already in interprocessor-sorted order, then each processor can simultaneously sort N/P elements in its local memory and no communication between processors is required. In this case, the sorting lower bound for N/P elements in each processor is also the lower bound for the whole sorting procedure. By the result in [1], the sorting lower bound for N/P elements in the HMM model with memory access cost $f(x)=\log x$ is exactly $\Omega((N/P)\log(N/P)\log\log(N/P))$. A similar argument can be used for FFT computation. Therefore we prove the theorem. □

FFT – Algorithm 1. This algorithm is designed for the global (or track-oriented) view of the memory system. We use the techniques in [84] and the algorithm presented in Section 2.3.3 for the P-HMM model, to compute the FFT on a LogP-HMM machine. The time needed by this algorithm is

$$O(\frac{L+o}{L} g \frac{N}{P} \log \frac{N}{P} \log\log \frac{N}{P}).$$

Steps 1 and 4 take $\sqrt{N}T(\sqrt{N}, P) + (N/P)\log(N/P)$ time. The shuffling step needs time $O((N/P)\log(N/P))$ after each group is "shifted" by an appropriate offset, because the shuffling can be done in the local memory hierarchy and does not cause any communication. The shifting can be done using the following method. Each processor sends the "shift" messages to the other processors simultaneously. When $\lceil L/g \rceil$ messages have been sent, every processor begins to receive the $\lceil L/g \rceil$ messages. This procedure is continued until each processor has sent and received all of the N/P messages. The time required for this communication is

$$O((\frac{N}{P}/\lceil \frac{L}{g} \rceil)(4L + 4o)) = O(\frac{L+o}{L} g \frac{N}{P}).$$

But each record sent may reside at a different level of the memory hierarchy. In order to move them into the lower memory levels, we need another $\log(N/P)$ factor, which gives

$$O(\frac{L+o}{L} g \frac{N}{P} \log \frac{N}{P}).$$

Therefore we have the following recurrence

$$T(N, P) = 2\sqrt{N}T(\sqrt{N}, P) + O(\frac{L+o}{L} g \frac{N}{P} \log \frac{N}{P}),$$

which yields the bound given above. The algorithm is thus within a factor $g(L+o)/L$ of optimal.

FFT – Algorithm 2. Algorithm 1 basically uses block layout for the input data. This algorithm will use the hybrid data layout and a tighter upper bound can be derived. Since the algorithm will operate on the local data set and use message passing to redistribute the data, we will refer to this algorithm as a local view algorithm. The

idea of the hybrid method has been discussed in Section 2.3.2. A more detailed discussion can be found in [73].

Two steps are used to compute the N-input FFT. In Step I, each processor computes an N/P-input butterfly; after each processor finishes its computation, it sends N/P^2 messages to each of the other processors. After each processor accepts the $N/P - N/P^2$ messages, it begins Step II which comprises the computation of the non-input nodes of N/P^2 disjoint P-input butterflies. By the result of [1], Step I computation needs $O((N/P)\log(N/P)\log\log(N/P))$ time. Step II computation needs

$$O((N/P^2)P\log P\log\log P) = O((N/P)\log P\log\log P)$$

time plus the time to move the N/P^2 P-input FFTs to the lower memory levels, which is bounded by $O(P(N/P^2)\log(N/P))$ or $O((N/P)\log(N/P))$. When there is no memory hierarchy, the communication time is

$$((\tfrac{N}{P} - \tfrac{N}{P^2})/\lceil\tfrac{L}{g}\rceil)(4L + 4o) = \tfrac{L+o}{L}g(\tfrac{N}{P} - \tfrac{N}{P^2}).$$

However, the messages sent may reside at different memory levels. This gives the communication time to be

$$\tfrac{L+o}{L}g(\tfrac{N}{P} - \tfrac{N}{P^2})\log\tfrac{N}{P}.$$

Thus the total running time is:

$$O(\tfrac{N}{P}\log\tfrac{N}{P}\log\log\tfrac{N}{P} + \tfrac{L+o}{L}g(\tfrac{N}{P} - \tfrac{N}{P^2})\log\tfrac{N}{P}).$$

Because $N/P \geq P$, the running time is within a factor $1+g(L+o)/(L\log\log(N/P))$ of optimal.

Sorting. A near optimal sorting algorithm can be obtained by using columnsort [54] and the modified median-sort algorithm proposed in [1] for a local view of memory layout. We will denote this modified median-sort algorithm as HMM-sorting in the rest of the chapter and we will use the result that HMM-sorting can sort N elements in $O(N \log N \log\log N)$ time. Columnsort performs sorting on a column-major $r \times c$ matrix with the conditions of $c \bmod r = 0$ and $r > 2(c-1)^2$. It executes eight consecutive steps, of which the odd-numbered steps sort the r elements in each column and the even-numbered steps permute the data among the processors.

In order to derive an efficient algorithm in the LogP-HMM model, we take $c = P-2$ and $r = N/P$. The columnsort condition is satisfied if $N > 2P(P-3)^2$. Each processor will be responsible for sorting a column. Therefore, by using HMM-sorting, each of the odd-numbered steps will take $O((N/P)\log(N/P)\log\log(N/P))$ time. In steps 2 and 4, each processor sends and receives $N/P - N/P^2$ elements. These elements may reside at or send to different memory levels, therefore the communication overhead would be

$$\tfrac{L+o}{L} g(\tfrac{N}{P} - \tfrac{N}{P^2}) \log \tfrac{N}{P}.$$

In step 6 and 8, each processor sends and receives $N/(2P)$ elements. The communication overhead would be

$$\tfrac{L+o}{L} g \tfrac{N}{2P} \log \tfrac{N}{P}.$$

Therefore the sorting can be done in time

$$O(\tfrac{N}{P} \log \tfrac{N}{P} \log\log \tfrac{N}{P} + \tfrac{L+o}{L} g \tfrac{N}{P} \log \tfrac{N}{P}),$$

when $N > 2P(P-3)^2$, which is within a factor $1 + g(L+o)/(L \log\log(N/P))$ of optimal.

3.1.3 An Alternative Optimal FFT Algorithm for P-HMM

The hybrid algorithm discussed above can be adapted for the P-HMM model and an optimal running time can be achieved. The analysis is summarized below:

1. The time for the N/P-input FFT is $O((N/P)\log(N/P)\log\log(N/P))$ on a single processor.

2. In the shuffle stage, each processor sends $N/P - N/P^2$ elements. In the P-HMM, sending a message over the network takes $\log P$ time. Therefore, the time for this stage is $(N/P - N/P^2)(\log P + \log(N/P))$, where the second term of $\log(N/P)$ is for transferring the data in the memory hierarchy.

3. The time to compute N/P^2 P-input FFTs is $O((N/P^2)(P\log P\log\log P) + (N/P)\log(N/P))$. Again the second term is for transferring the data in the memory hierarchy.

Summing all of them together, we get the following result:

$$O(\tfrac{N}{P}\log\tfrac{N}{P}\log\log\tfrac{N}{P}),$$

which matches the lower bound $O((N/P)\log N\log(\log N/\log P))$ proven in [84]. And therefore it is optimal.

3.2 The LogP-UMH Model

The principal difference between the LogP-UMH model and the LogP-HMM model is that in the former the memory of each processor is organized in a manner following the Uniform Memory Hierarchy (UMH) [6]. The UMH is an alternative model

for multilevel memories, and is an instance of the more general MH model also described in [6]. The memory hierarchy in the MH model consists of several levels of memory modules, where each level is characterized by three parameters. The parameters associated with the ℓth level are defined as follows.

- s_ℓ: the number of elements in a block
- n_ℓ: the number of blocks
- b_ℓ: the time to move a block of size s_ℓ from level ℓ to level $\ell + 1$

These parameters are sufficient to describe many practical memory hierarchies; however, the large number of parameters also makes algorithm design and analysis difficult. Moreover, some type of consistency among the parameters of different memory levels can facilitate the portability of designed algorithms. To address these concerns, a simplification of the MH model, the $\text{UMH}_{\alpha,\rho,f(\ell)}$ model, defines the ℓth memory level $M(\ell)$ as $M(\ell) = \langle s_\ell, n_\ell, b_\ell, \rangle = \langle \rho^\ell, \alpha\rho^\ell, \rho^\ell f(\ell) \rangle$, where α and ρ are integer constants. In other words, the ℓth memory level consists of $\alpha\rho^\ell$ blocks, each of size $s(\ell) = \rho^\ell$, and is connected to levels $\ell - 1$ and $\ell + 1$. Each block on level ℓ can be randomly accessed as a unit and transferred to or from level $\ell + 1$ with a cost of $\rho^\ell f(\ell)$, where $f(\ell)$ is a well-behaved function for ℓ and is called the transfer cost function ($1/f(\ell)$ is the bandwidth). The blocks at each level can be regarded as fully associated.

3.2.1 Algorithm Design and Analysis

We will present two near optimal FFT algorithms for the LogP-UMH model, using a global view and a local view of the address space respectively. (As discussed in Section 3.1.2, we can take two types of views of the address space.) For the global (or

track-oriented) view, we will assume that $b = \rho$. The performance of the algorithm will depend on the network properties, the data movement through the memory hierarchy, and also the memory bandwidth function $f(\ell)$. In order to simplify the algorithm and its analysis, however, we will assume that $f(\ell) = 1$. Under this condition, we also assume that a sub-block of size ρ in memory level i (where the full size of the block is ρ^i) can move down or up a level in ρ time units.

Lemma 2.1 *If a vector of length N is stored at level $\lceil \log_{\rho^2}(N/\alpha) \rceil$ originally, then moving the vector to fill the memory hierarchy beginning from the lowest level will take O(N) time.*

Proof: Assume that each time we move ρ elements, then we can pipeline the moving of the elements down the memory hierarchy. The first ρ elements take $\rho\lceil \log_{\rho^2}(N/\alpha) \rceil$ time to reach its destination, but each of the successive blocks of size ρ will take 0 or ρ time units depending on whether its predecessor is the last block for that memory level. When the last block of ρ elements reaches its destination memory level, the work will be done. Since there are N/ρ blocks each of size ρ, the total cost would be bounded by $\rho\lceil \log_{\rho^2}(N/\alpha) \rceil + ((N/\rho) - 1)\rho$. Therefore the total cost is O(N). □

Lemma 2.2 *If a vector of length N is stored at level $\lceil \log_{\rho^2}(N/\alpha) \rceil$ originally, then moving a subset of the vector with the length $\sqrt{N}$ to the memory level $\lceil \log_{\rho^2}(\sqrt{N}/\alpha) \rceil$ will take O($\sqrt{N}$) time.*

Proof: We use the same pipeline schedule as in the proof for Lemma 1. Now consider the traffic on the bus which connects the $\lceil \log_{\rho^2}(\sqrt{N}/\alpha) \rceil$th and $(\lceil \log_{\rho^2}(\sqrt{N}/\alpha) \rceil + 1)$th memory levels; it will keep busy after the first block of ρ elements arrives until the last block of elements passes through that bus. Therefore the total time would be $\rho\lceil \log_{\rho^2}(\sqrt{N}/\alpha) \rceil + ((\sqrt{N}/\rho) - 1)\rho$, which is bounded by $\sqrt{N}$. □

FFT – Algorithm 1 This algorithm works for the global view of the memory address space. We assume that $b = \rho$ as mentioned above. The algorithm consists of three steps similar to those in Algorithm 1 for the LogP-HMM model. We assume that the inputs are stored in the $(\lceil \log_{\rho^2}(N/(\alpha P)) \rceil + 1)$th memory level in each of the P processors initially. More specifically, the inputs are distributed among processors in a round-rabin manner with a data unit of ρ. The inputs can be viewed as a $\sqrt{N} \times \sqrt{N}$ matrix with row-major ordering, so that the $\sqrt{N}$-input FFTs discussed below can be regarded as computing $\sqrt{N}$-input FFTs for each row.

1. Compute $\sqrt{N}$ $\sqrt{N}$-input FFTs for each row.
2. Perform data rearrangement.
3. Compute $\sqrt{N}$ $\sqrt{N}$-input FFTs for each new row.

The data rearrangement is a matrix transposition. It consists of two steps: local sub-matrix transposition and communication. We highlight the analysis of the algorithm under the assumptions that $\sqrt{N}/(\rho P)$ is an integer. Therefore the elements in the same column will be stored in the same processor initially.

1. We claim that the number of messages that needs to be sent by each processor is equal to N/P. The main observation is that the elements at each column can be divided into $\sqrt{N}/(\rho P)$ subsections. The first ρ elements in each section can stay at the same processor; all of the other elements, which is $\rho(P-1)$ elements for each section, need to be transferred to other processors. Therefore each column will have $\sqrt{N}(P-1)/P$ elements to be sent to other processors, and all of the messages that need to be sent by all of the processors would be equal to $T = N(P-1)/P$. By symmetry, each processor will send $T/P = N(P-1)/P^2 \approx N/P$ messages.

2. The local data rearrangement is a $\sqrt{N} \times \sqrt{N}/P$ sub-matrix transposition. The local sub-matrix for the k-th processor where $0 \leq k < P$, consists of columns $\{\{j + k\rho + i\rho P | 0 \leq j < \rho\} | 0 \leq i \leq \sqrt{N}/(\rho P)\}$ in the original $\sqrt{N} \times \sqrt{N}$ matrix. After transposing this sub-matrix, the elements that need to stay and those that need to be communicated will be put into different memory sub-blocks each of size ρ. The matrix transposition can be done in O(N/P) time by using the algorithm in [6].

3. The transposed sub-matrix is stored at memory level $\lceil \log_{\rho^2}(N/(\alpha P)) \rceil$. We can now pipeline the blocks that need to be sent down the memory hierarchy and then sent through the network (we can schedule it so that the message passing through the network forms a ring). We also pipeline the blocks that need to be relocated down the memory hierarchy and then store them back through the memory hierarchy to the right location. If we assume that the values of ρ and g are comparable or that the bandwidth in the memory part is larger than $1/g$, then the network becomes the bottleneck due to its finite capacity. Therefore the communication cost would be

$$\tfrac{N}{P} + g\tfrac{L+o}{L}\tfrac{N}{P},$$

where the first term is due to moving through the memory hierarchy using a schedule similar to that in Lemma 2.1, and the second term is the cost for the communication.

Summarizing the above analysis and assuming that $g > 1$, we have the following recurrence:

$$T(N,P) = 2\sqrt{N}T(\sqrt{N},P) + O(\tfrac{L+o}{L}g\tfrac{N}{P}),$$

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 16 & 17 & 18 & 19 & 20 & 21 & 22 & 23 \\ 24 & 25 & 26 & 27 & 28 & 29 & 30 & 31 \\ 32 & 33 & 34 & 35 & 36 & 37 & 38 & 39 \\ 40 & 41 & 42 & 43 & 44 & 45 & 46 & 47 \\ 48 & 49 & 50 & 51 & 52 & 53 & 54 & 55 \\ 56 & 57 & 58 & 59 & 60 & 61 & 62 & 63 \end{pmatrix}$$

Figure 3.2: The initial input is regarded as a two-dimensional matrix.

which yields the bound

$$O(\frac{L+o}{L} g \frac{N}{P} \log \frac{N}{P}).$$

We now use an example to illustrate the data organization and rearrangement. This example demonstrates that data rearrangement can be accomplished by a local data transposition and then a global data communication. Let $N = 64$, $\rho = 2$, $P = 2$ and $b = 2$.

1. **Initial data distribution.** Figure 3.2 shows the initial inputs when they are viewed as an 8×8 square matrix. According to our convention, the input data is distributed across two processors in a round-robin manner with a block of size two. In terms of the original 8×8 data matrix, we can see that the highlighted columns consist of an 8×4 sub-matrix. Initially, this sub-matrix is stored in processor 0 at memory level 4. The rest of the columns constitute another sub-matrix, which is initially stored in processor 1 at memory level 4. This initial data distribution is shown in Figure 3.3 (A), where we use the shadowing to highlight the elements in the first row of the original 8×8 square matrix.

2. **Local sub-matrix transposition.** Data in each local memory hierarchy can be explained as an 8×4 sub-matrix. Figure 3.3 (B) shows the data organization after applying matrix transposition to this sub-matrix. We explain the highlights in Figure 3.3 (B) in the next step.

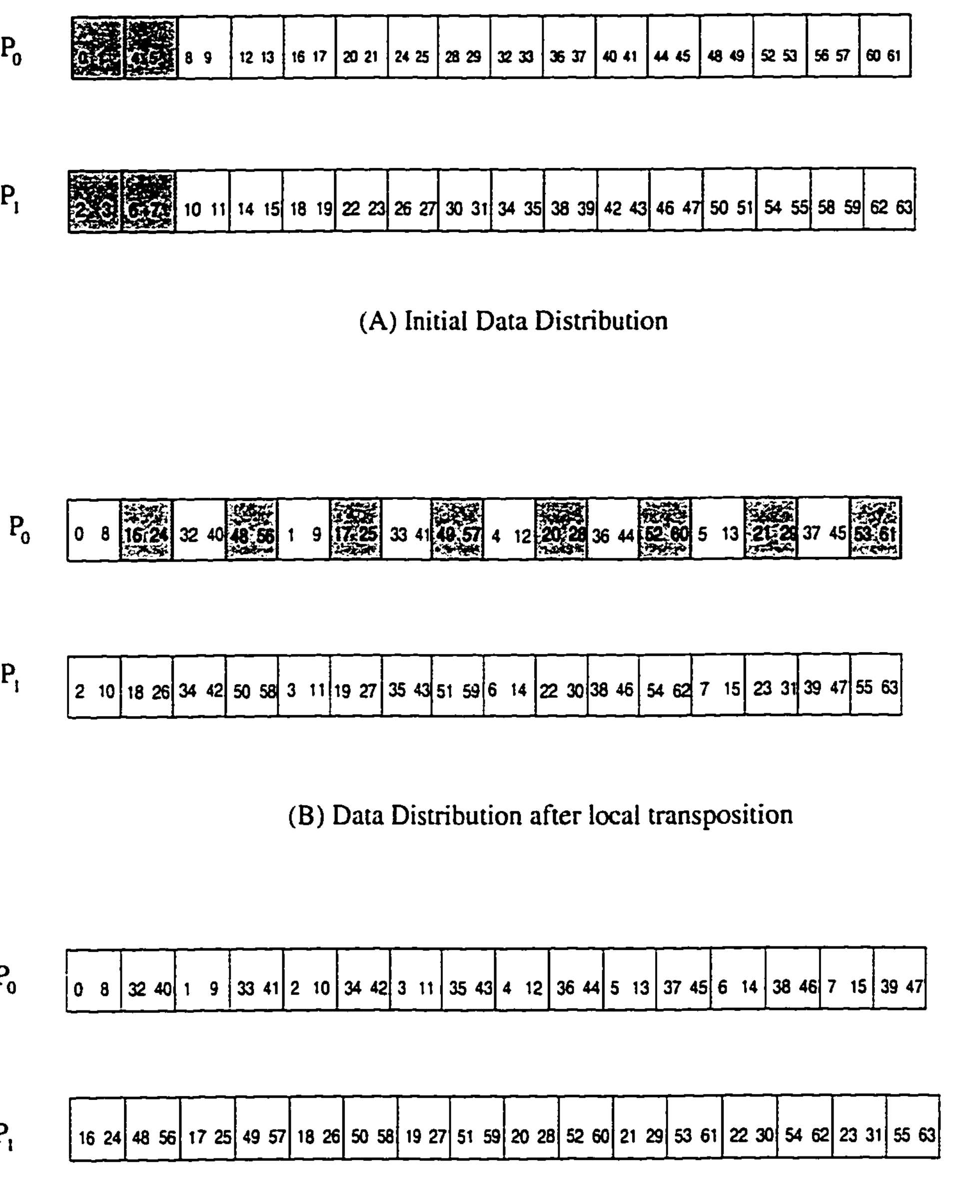
3. **Global data communication.** After the local sub-matrix transposition, data which should stay and be sent are stored in different blocks as shown in Figure 3.3 (B), where we highlight blocks which should be sent out from processor 0. For processor 0, records in rows 2,3,6, and 7 in the original input matrix should be sent to other processors. The data distribution after communications is shown in Figure 3.3 (C). It can be verified that now data is distributed among processors in a round-robin manner with a block of size two, however, data is linearized in the column-major order.

FFT – Algorithm 2 This algorithm embodies a local view of the address space for the LogP-UMH model. The algorithm consists of two steps similar to Algorithm 2 for the LogP-HMM model. However, here the local computation will be conducted in a memory hierarchy characterized by UMH parameters. Since the communication pattern is the same as the algorithm presented for the LogP-HMM model, we need only analyze the local FFT computation. We use a three step algorithm [6, 57] for the local m-input FFT computation, where $m = N/P$, as follows:

1. Regard the m inputs as a $\sqrt{m} \times \sqrt{m}$ matrix stored in column major order at level $\lceil \log_{\rho^2}(m/\alpha) \rceil$. Compute $\sqrt{m}$ $\sqrt{m}$-input FFTs along the rows.

2. Transpose the matrix.

3. Do $\sqrt{m}$ $\sqrt{m}$-input FFTs along the new rows.



(A) Initial Data Distribution

(B) Data Distribution after local transposition

(C) Data distribution after Step 2 of data arrangement

Figure 3.3: The initial and final data distributions on two processors. Note that we only show the fourth level of the memory hierarchy.

Assuming that the total running time is $T(m)$, then Step 1 will take time $\sqrt{m}T(\sqrt{m})$ plus the time for moving data from memory level $\ell_2 = \lceil \log_{\rho^2}(m/\alpha) \rceil$ to memory level $\ell_1 = \lceil \log_{\rho^2}(\sqrt{m}/\alpha) \rceil$ and storing the data back to level ℓ_2. The first term in this formula is obvious. The second term arises because the different $\sqrt{m}$ $\sqrt{m}$-input FFTs sit at memory level $\lceil \log_{\rho^2}(m/\alpha) \rceil$ originally [1]. We need to move them into the lower memory level in order to recursively compute the $\sqrt{m}$-input FFT. By Lemma 2.2, this will cost O(m) time (since $\sqrt{m}O(\sqrt{m}) = O(m)$). Step 2 of matrix transposition takes O(m) time using the algorithms described in [6]. Step 3 takes the same amount of time as Step 1. Therefore the total time can be described by the following recurrence,

$$T(m) = 2\sqrt{m}T(\sqrt{m}) + O(m),$$

which gives the bound $O(m \log m)$. This matches the FFT lower bound and therefore is optimal.

By using this result and following the same analysis presented in Section 3.1.2 for Algorithm 2 for the LogP-HMM model, we can conclude that the total running time is:

$$O(\frac{N}{P} \log \frac{N}{P} + \frac{L+o}{L} g(\frac{N}{P} - \frac{N}{P^2})).$$

3.2.2 Matching LogP-UMH to Practical Parallel Machines

The LogP-UMH model, in comparison to the LogP-HMM model, can be used to more accurately model distributed memory machines. The interconnection network of the machine will be captured by the four parameters L, o, g, and P. The multi-level memory for each node – which typically consists of a register file, a cache, and

[1] There is an omission in [6] for the similar analysis.

level	CM-5 node processor n_ℓ	s_ℓ	$1/f(\ell)$	LogP-UMH memory n_ℓ	s_ℓ	$1/f(\ell)$
0	264	4	4	256	1	4
1	2048	32	3.87	8196	32	4
2	8K	4K		256K	1K	

Table 3.1: Comparison of the CM-5 node memory hierarchy with the LogP-UMH. $\alpha = 256$, $\rho = 32$ and $1/f(\ell) = 4(byte/cycle)$.

an internal memory (plus a disk storage for each node for some machines such as the SP-1 and the SP-2) – can be captured by the parameters α, ρ, and bandwidth $1/f(\ell)$, which in turn determine the blocksize, number of blocks, and block transfer time for each memory level.

For example, the CM-5 can be modeled as follows. The use of L, o, g and P to model the CM-5 has been discussed in [25], where it is determined that the values of $L = 6\ \mu s$, $o = 2\ \mu s$ and $g = 4\ \mu s$ are fairly appropriate. The obvious omission of memory hierarchy in this characterization can be alleviated by adding the parameters of α, ρ and bandwidth $1/f(\ell)$ for the memory levels. To match the values suggested in the CM-5 technical specification documents [80], we take α=256, ρ=32 and $1/f(\ell)$=4 to approximate the memory hierarchy. A comparison of the memory hierarchy (i.e., blocksize and block transfer times) derived using these values and the real values of the CM-5 is shown in Table 3.1, where levels 0, 1, and 2 represent the register, the cache, and the internal memory respectively [2]. It can be seen that the LogP-UMH model captures the flavor of the CM-5 machine.

The IBM SP-2 provides a more interesting example since each node of the SP-

[2]The node processor of the CM-5 is a SPARC microprocessor. In that processor, the register file is organized as register windows and, according to the architecture configuration, there may be between 6 to 32 register windows. Each window has 24 working registers (but 8 overlap and we therefore count only 16 in each window), plus 8 global registers. The value given in the table is for an average number of register windows.

level	SP-2 node processor n_ℓ	s_ℓ	$1/f(\ell)$	LogP-UMH memory n_ℓ	s_ℓ	$1/f(\ell)$
0	64	4	4	64	1	4
1	512	128	12.8	2048	64	8
2	32K	4K		128K	4K	
3	1G(total)			8M	256K	

Table 3.2: Comparison of the SP-2 node memory hierarchy with the LogP-UMH. $\alpha = 32$, $\rho = 64$ and $1/f(\ell) = 4(\ell + 1)$.

2 not only has a cache and an internal memory, but also has a large disk storage. The interconnection network of the SP-2 is a high-performance switch [14, 77]. Using the LogP-UMH model, we can abstract the communication characteristics of this switch by four parameters L, o, g, and P. Because the bi-directional bandwidth is 40 megabytes per second with a latency of 500 nanoseconds, and sending an empty message takes 30 μs(when using the EUIH port), we have L = 0.4 μs, o = 30 μs, and g = 0.025 μs. The memory can be characterized by the parameters of α, ρ and bandwidth $1/f(\ell)$. We take α=32, ρ=64 and $1/f(\ell) = 4(\ell + 1)$ to approximate the memory hierarchy. A comparison of the memory hierarchy defined by these values and the real values of the SP-2 is shown in Table 3.2, where the values are for the RS/6000 560 microprocessor (one of the POWER series chips). Levels 0, 1, 2, and 3 represent the register, the cache, the internal memory, and the disk memory respectively. All of the values in this table as well as in Table 3.1 are approximate because they may be dependent on the machine configuration.

3.3 Conclusions

This chapter showed that resource metrics can be used not only to understand existing models, but also to guide the design of new models. The LogP-HMM model proposed in this chapter (and its variant the LogP-UMH model) serves as an illustration of a model designed to fill the gap observed between models which address communication and those which address memory hierarchy. The design of near optimal sorting and FFT algorithms for the LogP-HMM model and the LogP-UMH model give promise that the models have the potential to serve as a viable tool for the design and analysis of practical algorithms for a large class of machines. In matching the LogP-UMH model to the CM-5 and the IBM SP-2, we demonstrated that the model has the potential to reflect practical parallel machines.

In the rest of the thesis, we will use two simpler variants of the LogP-HMM model, which consider a fewer number of memory-levels and mainly capture the performance properties of disk access. We use these two variants as the target models for which we will develop the algebraic methods for synthesizing efficient parallel out-of-core programs.

Chapter 4

Synthesizing Programs from Tensor Products

Beginning from this chapter, we turn our focus from models of parallel computation to program synthesis for out-of-core block recursive algorithms. Our methodology of program synthesis is based on the tensor product algebra.

Many algorithms in the area of numeric computation can be represented as a matrix-vector multiplication. Further, the (computational) matrix has the following properties. The computational matrix can be decomposed as a sequence of matrix-matrix multiplication. Each of the decomposed matrices is either a permutation matrix or a block matrix, and both of the permutation matrix and the block matrix can be represented by tensor products. In this chapter, we introduce tensor products and other matrix operations, discuss computations which can be represented by tensor products, and illustrate how to generate efficient programs from the tensor product representation of an algorithm on a given architecture.

4.1 Tensor Product Algebra

4.1.1 Definition of Tensor Products

The tensor (Kronecker) product [35] of an $m \times n$ matrix $A^{m,n}$ and a $p \times q$ matrix $B^{p,q}$ is a block matrix $A^{m,n} \otimes B^{p,q}$ obtained by replacing each element $a_{i,j}$ of $A^{m,n}$ by the matrix $[a_{i,j}B^{p,q}]$.

$$A^{m,n} \otimes B^{p,q} = \begin{bmatrix} a_{0,0}B^{p,q} & \cdots & a_{0,n-1}B^{p,q} \\ \vdots & \ddots & \vdots \\ a_{m-1,0}B^{p,q} & \cdots & a_{m-1,n-1}B^{p,q} \end{bmatrix}^{mp,nq} . \tag{4.1}$$

A tensor product involving an identity matrix can be implemented as parallel operations. An identity matrix of order n is denoted as I_n. Consider the application of $(I_n \otimes B^{p,q})$ to a vector X of length nq:

$$(I_n \otimes B^{p,q})(X) = \begin{bmatrix} B^{p,q} & & \\ & \ddots & \\ & & B^{p,q} \end{bmatrix} \begin{bmatrix} X_0 \\ \vdots \\ X_{n-1} \end{bmatrix}, \tag{4.2}$$

where $X_i = X[iq : (i+1)q - 1], 0 \leq i < n$. This can be interpreted as n copies of $B^{p,q}$ acting in parallel on n disjoint segments of X. We therefore call $I_n \otimes B^{p,q}$ a parallel form. Similarly, the application $(A^{m,n} \otimes I_p)$ to a vector X can be written as,

$$(A^{m,n} \otimes I_p)X = \begin{bmatrix} a_{0,0}I_p & \cdots & a_{0,n-1}I_p \\ \vdots & \ddots & \vdots \\ a_{m-1,0}I_p & \cdots & a_{m-1,n-1}I_p \end{bmatrix} \begin{bmatrix} X_0 \\ \vdots \\ X_{n-1} \end{bmatrix}, \tag{4.3}$$

where $X_i = X[ip : (i+1)p - 1], 0 \leq i < n$. This can be interpreted as p parallel applications of $A^{m,n}$ on p disjoint segments of X. However, the inputs for each application of $A^{m,n}$ are accessed at a stride of p and the outputs are also stored at a stride

of p [1]. In general, $(I_m \otimes A^{n,p} \otimes I_q)$ can be interpreted as mq parallel applications of $A^{n,p}$.

The properties of tensor products can be used to transform the tensor product representation of one algorithm into an equivalent form, which can take advantage of the parallel operations discussed above. For example, by using the following tensor product factorizations,

$$A^{m,n} \otimes B^{p,q} = (A^{m,n} \otimes I_p)(I_n \otimes B^{p,q}) = (I_m \otimes B^{p,q})(A^{m,n} \otimes I_q); \tag{4.5}$$

$A \otimes B$ can be implemented by first applying q parallel applications of A and then m parallel applications of B [2]. Several other key properties of tensor products are listed below [44]:

1. $A \otimes B \otimes C = A \otimes (B \otimes C) = (A \otimes B) \otimes C$;
2. $(A \otimes B)(C \otimes D) = AC \otimes BD$; assuming that the ordinary multiplications AC and BD are defined.
3. $\prod_{i=0}^{n-1}(I_m \otimes A_i) = I_m \otimes (\prod_{i=0}^{n-1} A_i)$;
4. $\prod_{i=0}^{n-1}(A_i \otimes I_m) = (\prod_{i=0}^{n-1} A_i) \otimes I_m$.

Property 2 is also called *factor grouping*. It transforms the multiplication of two tensor products into one tensor product by first multiplying the matrices A with C and B with D, respectively.

---

[1] $A^{m,n} \otimes I_p$ can also be interpreted as a vector operation as follows,

$$(A^{m,n} \otimes I_p)X = \begin{bmatrix} a_{0,0}X_0 + \cdots + a_{0,n-1}X_{n-1} \\ \vdots \\ a_{m-1,0}X_0 + \cdots + a_{m-1,n-1}X_{n-1} \end{bmatrix} X, \tag{4.4}$$

where $a_{i,j}X_j$ denotes a scalar-vector multiply and + denotes a vector addition. We therefore call $A^{m,n} \otimes I_p$ a vector form.

[2] We ignore the dimensions of matrices whenever they are clear from the context.

4.1.2 Stride Permutations

A *stride permutation* L_n^{mn} is an $mn \times mn$ permutation matrix. The application of L_n^{mn} to X of length mn results in a vector Y of length mn such that:

$$Y = \left[\begin{array}{c} X(0:mn-1:n), X(1:mn-1:n), \ldots, X(m-1:mn-1:n) \end{array} \right]^T \quad (4.6)$$

where $X(i : mn - 1 : n)$, $0 \leq i < n$, denotes a vector consisting of the elements in the set $\{X[i + j * n] | 0 \leq j \leq (m - 1)\}$. In other words, Y can be constructed from X by the following methods: starting from the first element and taking every nth element from X and then starting from the second element and taking every nth element from X, and so on. This operation is also called an n-way *perfect shuffle*. Several properties of stride permutations are listed below.

5. $L_{st}^{rst} = L_s^{rst} L_t^{rst}$;

6. $L_n^{mn}(A \otimes B) = (B \otimes A) L_n^{mn}$, where A and B are square matrices

4.1.3 Tensor Bases

In contrast to tensor products which can be used to describe various computations, the tensor product of *vector bases*, called a *tensor basis*, can be used to describe data access and storage patterns of a multi-dimensional array. A *vector basis* e_i^m, $0 \leq i < m$, is a column vector of length m with a one at position i and zeros elsewhere. We can use e_i^m to select the ith element of a one-dimensional array of size m by a dot product. Similarly, we can use $e_i^m \otimes e_j^n$ to select $[i, j]$-th element of a two-dimensional $m \times n$ array. Since $e_i^m \otimes e_j^n = e_{in+j}^{mn}$ [44], we associate $e_i^m \otimes e_j^n$ with the *indexing function* $in + j$, which can be used to access or *linearize* a two-dimensional array in row-major order by using the following program,

```
DO i = 1, m
  DO j = 1, n
    Y(in + j) = X[i, j]
ENDDO ENDDO
```

where X is an $m \times n$ array and Y is a vector of length mn. Because of this property, we will use the same notation $e_i^m \otimes e_j^n$ to denote both a single $[i, j]$-th element of X and an mn-length vector Y linearized in row-major order from the $m \times n$ matrix X.

In general, the tensor basis $e_{i_t}^{m_t} \otimes \cdots \otimes e_{i_1}^{m_1}$ corresponds to index $[i_t, \ldots, i_1]$ of a t-dimensional $m_t \times m_{t-1} \times \cdots \times m_1$ array. The indexing function needed to access elements of a multi-dimensional array can be obtained by *linearizing* the tensor basis. For example, linearizing the tensor basis $e_{i_t}^{m_t} \otimes \cdots \otimes e_{i_1}^{m_1}$ results in a vector basis $e_{i_t m_{t-1} \cdots m_1 + \cdots + i_2 m_1 + i_1}^{m_t \cdots m_1}$. The index in the linearized tensor basis is exactly the indexing function needed for accessing a t-dimensional array in row-major order. Equivalently, a vector basis e_i^M can be factored into a tensor product of vector bases $e_{i_t}^{m_t} \otimes \cdots \otimes e_{i_1}^{m_1}$, where $M = m_t \cdots m_1$ and $i_k = (i \; div \; M_{k-1}) \; mod \; m_k$, $M_k = \prod_{i=1}^{k} m_i$, $M_0 = 1$. For example, $e_i^{12} = e_{i_3}^2 \otimes e_{i_2}^3 \otimes e_{i_1}^2$, where $i_3 = (i \; div \; 6)$, $i_2 = (i \; div \; 2) \; mod \; 3$, and $i_1 = (i \; mod \; 3)$. Factorization of a vector basis corresponds to viewing a one-dimensional array as a multi-dimensional array. Fig. 4.1 shows an array of size twelve when it is viewed as a vector and its index is represented by the vector basis e_i^{12}, and when it is viewed as a three-dimensional array and its index is represented by the tensor basis $e_{i_3}^2 \otimes e_{i_2}^3 \otimes e_{i_1}^2$.

Using tensor bases, the semantics of the stride permutation L_n^{mn} can be formally expressed as:

$$L_n^{mn} \left(e_i^m \otimes e_j^n \right) = e_j^n \otimes e_i^m, \tag{4.7}$$

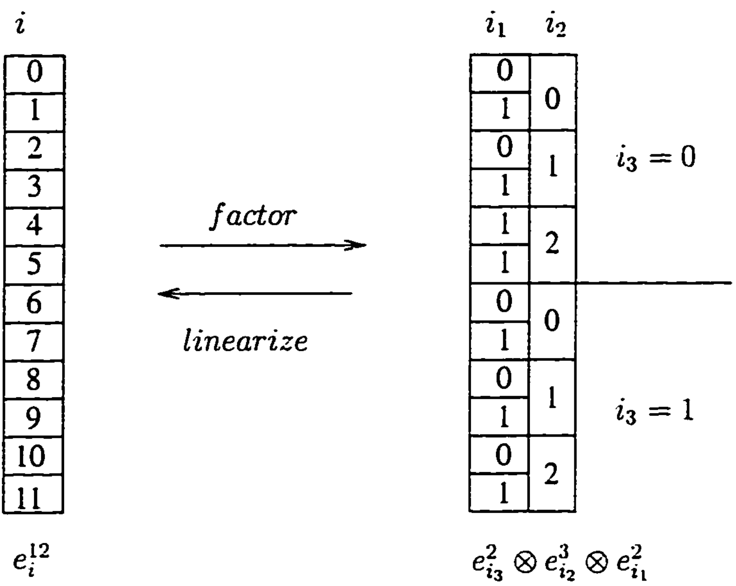


Figure 4.1: Vector factorization and array linearization.

which can be explained as follows. We regard the input vector as a two-dimensional array, which is denoted by the *input basis*, $e_i^m \otimes e_j^n$. The stride permutation is equivalent to linearizing this array in column-major order. In other words, the output vector can be viewed as an array denoted by the *output basis*, $e_j^n \otimes e_i^m$. Thus, the implementation of a stride permutation is equivalent to exchanging the two vector bases in the input basis.

By appropriately factoring the vector basis for an input vector, we can use the resulting tensor basis to describe the data access pattern of a tensor product. For example, to implement $I_n \otimes B^{p,q}$ in parallel, we can apply $B^{p,q}$ to each segment of the input vector. This can be interpreted as follows. We first factor the input vector basis as the following input basis, $e_i^n \otimes e_j^q$. For each segment denoted by index i, we can then apply $B^{p,q}$ to it simultaneously. The index of the output vector can be denoted by the output basis, which can be obtained by applying $I_n \otimes B^{p,q}$ to the input basis.

The details on how to obtain an output basis and how to write the corresponding programs are discussed in Section 4.3.

Operations on Tensor Bases. We further introduce several operations applied to tensor bases. These operations will be used extensively in the rest of the thesis.

Definition 1.3 *Let $\mathcal{S}$ and $\mathcal{G}$ be two tensor bases. Their difference is denoted as $\mathcal{S}$-$\mathcal{G}$ and is a tensor basis which is constructed by deleting all of the vector bases in $\mathcal{G}$ from $\mathcal{S}$.*

Definition 1.4 *Let $\mathcal{S}$ be a tensor basis $\otimes_{s=1}^{q} e_{i_s}^{x_s}$, where $\otimes_{s=1}^{q} e_{i_s}^{x_s} = e_{i_q}^{x_q} \otimes \cdots \otimes e_{i_1}^{x_1}$. Let α be a permutation on $[1 \ldots q]$, then the permutation of $\mathcal{S}$ is a tensor basis defined as follows, $\alpha(\mathcal{S}) = \otimes_{s=1}^{q} e_{i_{\alpha(s)}}^{x_{\alpha(s)}}$.*

Definition 1.5 *The notation $\mid \mathcal{S} \mid$ denotes the size of the tensor basis $\mathcal{S}$, which is equal to the product of the dimensions of each vector basis in $\mathcal{S}$.*

For example, assume that $\mathcal{S} = e_{i_3}^{m_3} \otimes e_{i_2}^{m_2} \otimes e_{i_1}^{m_1}$ and $\mathcal{G} = e_{i_4}^{m_4} \otimes e_{i_2}^{m_2} \otimes e_{i_1}^{m_1}$. Then $\mathcal{S} - \mathcal{G} = e_{i_3}^{m_3}$. $\mid \mathcal{S} \mid = m_3 m_2 m_1$. The permutation which puts $e_{i_1}^{m_1}$ as the first factor of $\mathcal{S}$ produces the following tensor basis $e_{i_1}^{m_1} \otimes e_{i_3}^{m_3} \otimes e_{i_2}^{m_2}$.

4.2 Tensor Product Formulation of Block Recursive Algorithms

We define a *block recursive algorithm* as the following tensor product formula multiplied by the input vector X of length $r_j v_j c_j$,

$$\prod_{j=1}^{k} (I_{r_j} \otimes A_{v_j} \otimes I_{c_j}), \tag{4.8}$$

where A_{v_j} is a $v_j \times v_j$ square linear transformation, $\prod_{i=1}^{k} F_i$ denotes $F_k \cdots F_1$, and $r_j v_j c_j = r_i v_i c_i$, for $1 \leq i, j \leq k$. In order that A_{v_j} can be computed in main memory, we require that $v_j \leq M$, the size of the main memory. If this is not true, then we may need to factor A_{v_j} further.

The tensor product representation of Formula (4.8) can be derived from a recursive tensor factorization of a computational matrix. A typical example of this derivation is the FFT algorithm, which can be derived from tensor factorizations of the discrete Fourier transform (DFT) matrix [72, 57]. Other examples of block recursive algorithms include block matrix transposition, bitonic sort, ascend/descend algorithms [67], Strassen's matrix multiplication [41, 45], convolution [36], and fast sine/cosine transforms [57].

There are several advantages for the algorithms derived from tensor factorizations. First, the algorithms can be computationally more efficient than those that directly use the unfactored matrix. For example, computing the DFT of a vector of size N by directly multiplying it by an $N \times N$ DFT matrix requires $O(N^2)$ operations. However, by using the FFT algorithm, the same computation only requires $O(N \log N)$ operations. Second, by using the algebraic properties of tensor products, the derived tensor product representation can have a high-performance implementation on a given architecture.

For example, the identity term I_{r_j} allows a decomposition of the computation into a set of smaller size computations (called sub-computations henceforth), where each sub-computation accesses the inputs contiguously and may be computed in main memory. Similarly, the identity term I_{c_j} allows a decomposition of the computation into a set of sub-computations, where each sub-computation accesses the inputs in a stride. Although, these parallel and stride computational structures help in decomposing the

computation into smaller in-core computations, the task of combining these decompositions with the goal of minimizing I/O overhead for the entire computation is a challenging problem. We next use the Cooley-Tukey FFT and block matrix transposition algorithms as examples to illustrate the tensor product formulations of block recursive algorithms.

Fast Fourier Transform The Fourier transform can be denoted by the following matrix-vector multiplication,

$$Y = F_N X, \tag{4.9}$$

where F_N is an $N \times N$ discrete Fourier matrix. $F_N(i,j) = \omega_N^{i \times j}$, where $0 \leq i, j < N$ and ω_N is the N-th primitive root of unity ($\omega_N = e^{2\pi\sqrt{-1}/N}$). If $N = rs$, then the discrete Fourier matrix F_N can be factored as follows,

$$F_N = (F_r \otimes I_s) T_s^{rs} (I_r \otimes F_s) L_r^{sr}, \tag{4.10}$$

where, the diagonal matrix T_s^{rs} holds the twiddle factors and is defined by $T_s^{rs} e_i^r \otimes e_j^s = \omega^{ij} e_i^r \otimes e_j^s$. This factorization forms the basis for deriving the Cooley-Tukey FFT algorithm. Different tensor product factorizations of the discrete Fourier matrix can result in different tensor product formulations of various FFT algorithms [44, 57]. Although all of these algorithms are computationally equivalent, they have different computational structures and different data access patterns. For example, consider the following tensor product formulation of the radix-2 Cooley-Tukey FFT:

$$F_{2^n} = \left(\prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(I_{2^{n-i}} \otimes T_{2^{i-1}}^{2^i}) \right) R_{2^n}, \tag{4.11}$$

$$R_{2^n} = \prod_{i=1}^{n} (I_{2^{i-1}} \otimes L_2^{2^{n-i+1}}), \text{ and } F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}.$$

where $T^{2^i}_{2^{i-1}}$ is a diagonal matrix of constants and R_{2^n} permutes the input sequence to a bit-reversed order. From Eq. (4.11), we can see that a 2^n-point FFT contains n computational steps after performing the initial bit-reversal permutation. At each step, the data from the previous step is scaled by multiplying by the twiddle factors $X'_{i-1} = (I_{2^{n-i}} \otimes T^{2^i}_{2^{i-1}})(X_{i-1})$, followed by the butterfly computation $X_i = (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}})(X'_{i-1})$. In other words, if we ignore the initial bit-reversal operation, then the Cooley-Tukey FFT algorithm can be represented by the tensor product formula defined by Formula (4.8). The initial bit-reversal operation consists of a set of *tensor permutations* (a tensor permutation has the form $I_r \otimes L^{mn}_n \otimes I_c$), which can be computed once we know how to compute stride permutations.

Matrix Transposition The transposition of a $p \times q$ matrix $M^{p,q}$ can be expressed using a stride permutation L^{pq}_q as $(M^{pq})^T = L^{pq}_q(M^{pq})$, where M^{pq} is the row-major linear representation of $M^{p,q}$. Various matrix transposition algorithms can be expressed using tensor product formulas involving stride permutations [39]. For example, the block matrix transposition algorithm for transposing a $p \times q$ matrix can be described by the following formula:

$$L^{pq}_q = (I_{q_2} \otimes L^{p_2q_1}_{q_1} \otimes I_{p_1})(L^{p_2q_2}_{q_2} \otimes I_{p_1q_1})(I_{p_2q_2} \otimes L^{p_1q_1}_{q_1})(I_{p_2} \otimes L^{p_1q_2}_{q_2} \otimes I_{q_1}), \quad (4.12)$$

where $p = p_2p_1$ and $q = q_2q_1$. The first (rightmost) factor converts the row-major representation of the input matrix to a row-major representation of the input matrix viewed as a $p_2 \times q_2$ block matrix consisting of $p_1 \times q_1$ size blocks. The second and third factor express transposition of each block and transposition of the block matrix, respectively. The fourth factor is the inverse of the first and it reverts the block row-major representation to row-major representation of the output. The correctness of this representation can be seen by applying the factors to the input basis

$\beta_s \equiv e_{i_2}^{p_2} \otimes e_{i_1}^{p_1} \otimes e_{j_2}^{q_2} \otimes e_{j_1}^{q_1}$ to get the following sequence of bases,

$$\beta_s \rightarrow e_{i_2}^{p_2} \otimes e_{j_2}^{q_2} \otimes e_{i_1}^{p_1} \otimes e_{j_1}^{q_1} \rightarrow e_{i_2}^{p_2} \otimes e_{j_2}^{q_2} \otimes e_{j_1}^{q_1} \otimes e_{i_1}^{p_1} \rightarrow e_{j_2}^{q_2} \otimes e_{i_2}^{p_2} \otimes e_{j_1}^{q_1} \otimes e_{i_1}^{p_1}$$

$$\rightarrow e_{j_2}^{q_2} \otimes e_{j_1}^{q_1} \otimes e_{i_2}^{p_2} \otimes e_{i_1}^{p_1} = \beta_t,$$

and noting that $\beta_t = L_q^{pq}(\beta_s)$. Note that we have used the identity

$$(A^{m,n} \otimes B^{p,q})(e_i^n \otimes e_j^q) = A^{m,n}(e_i^n) \otimes B^{p,q}(e_j^q).$$

4.3 Code Generation from Tensor Products

In this section, we discuss how to relate a tensor product representation of a block recursive algorithm with a program. We also review how to use the properties of tensor products to transform the tensor product representation of a block recursive algorithm into an equivalent formula which can be implemented efficiently on a given architecture.

4.3.1 Writing Programs from Tensor Products

Consider the tensor product $I_r \otimes A^{m,n} \otimes I_c$, a basic building block for block recursive algorithms. As we have pointed out, this formula can be implemented as $r \times c$ parallel applications of the computational matrix $A^{m,n}$. Let us first discuss how to relate a program with the computational matrix $A^{m,n}$. Note that $A^{m,n}$ is applied to a vector of length n. The code for this application is nothing more than the code for a matrix-vector multiplication. For example, let $A^{m,n}$ be F_2. Let the input and the output vectors be denoted as X and Y, respectively. Then F_2 can be computed by the following code,

$$Y(0) = X(0) + X(1)$$
$$Y(1) = X(0) - X(1)$$

We next discuss how to compute $I_r \otimes A^{m,n}$. As we have mentioned, $I_r \otimes A^{m,n}$ can be implemented as r parallel applications of $A^{m,n}$ to r segments of the input vector. To obtain the index for each segment, we can factor the input vector basis e_i^{rn} as the input basis $e_i^r \otimes e_j^n$, where e_j^n is known as an *operator basis*. This input basis is *compatible* with the input tensor product, since it has the same number of factors as the tensor product and the size of each factor is equal to the number of elements required by the corresponding matrix in the tensor product. For example, for the tensor product $I_2 \otimes A^{2,2} \otimes I_4$, $e_i^2 \otimes e_j^2 \otimes e_k^4$ is a compatible tensor basis. However, $e_i^2 \otimes e_j^4 \otimes e_k^2$ is not a compatible tensor basis. Further, we can apply the tensor product to the compatible input basis to obtain a (compatible) output basis as follows,

$$(I_r \otimes A^{m,n})(e_i^r \otimes e_j^n) = I_r e_i^r \otimes A^{m,n} e_j^n = e_i^r \otimes A^{m,n} e_j^n, \tag{4.13}$$

and by replacing $A^{m,n} e_j^n$ by $e_{j'}^m$. Using these input and output bases, we can determine the input and output data elements required by the application of $A^{m,n}$ and derive a program as follows,

```
DOALL i = 0, r − 1
  Code for i-th application of A^{m,n}
ENDDOALL
```

The indices of the input data elements to the i-th application can be obtained from the linearized input basis e_{in+j}^{rn} as $\{in + j | 0 \leq j < n\}$. Similarly, the output indices can be determined from the linearized output basis as $\{im + j' | 0 \leq j' < m\}$. Note that, there are no loops corresponding to indices j and j' in the above program.

For $I_r \otimes A^{m,n} \otimes I_c$, the input vector basis e_s^{rnc} can be factored to obtain the input basis $e_i^r \otimes e_j^n \otimes e_k^c$, where e_j^n is the operator basis. The output vector basis e_s^{rmc} can be factored to obtain the output basis $e_i^r \otimes e_{j'}^m \otimes e_k^c$, which can also be determined by the

following identity,

$$(I_r \otimes A^{m,n} \otimes I_c)(e_i^r \otimes e_j^n \otimes e_k^c) = I_r e_i^r \otimes A^{m,n} e_j^n \otimes I_c e_k^c = e_i^r \otimes A^{m,n} e_j^n \otimes e_k^c, \quad (4.14)$$

and by replacing $A^{m,n}e_j^n$ by $e_{j'}^m$. Using these input and output bases, we now can determine the input and the output data elements required by the application of $A^{m,n}$ and derive a program with a two-dimensional iteration space:

```
DOALL i = 0, r - 1
  DOALL k = 0, c - 1
    Code for (i, k)-th application of A^{m,n}
ENDDOALL ENDDOALL
```

Similarly, the indices of the input data elements to the (i, k)-th application of $A^{m,n}$ can be obtained from the linearized input basis $e_{inc+cj+k}^{rnc}$ as $\{inc + cj + k | 0 \leq j < n\}$. The output indices can be determined from the linearized output tensor basis as $\{imc + cj' + k | 0 \leq j' < m\}$. Note that, there are no loops corresponding to indices j and j' in the above program.

We now use the Cooley-Tukey FFT computation defined by Formula (4.11) as an example to illustrate how to relate a tensor product formula with a program. For simplicity, we ignore the initial bit-reversal operation and the twiddle factor computation. In other words, we consider the following *core* Cooley-Tukey FFT algorithm,

$$F_{2^n} = \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}). \quad (4.15)$$

This algorithm constitutes multi-step computations to the input vector X. At step i, the application is denoted by the tensor product $I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}$. Using the method discussed above, the indexing function for the computation at the ith step can be obtained by the following input and output bases,

$$B_i = e_{i_3}^{2^{n-i}} \otimes e_{i_2}^2 \otimes e_{i_1}^{2^{i-1}}, \quad (4.16)$$

$$B_o = e_{i_3}^{2^{n-i}} \otimes e_{i_2}^{2} \otimes e_{i_1}^{2^{i-1}}. \tag{4.17}$$

Therefore, the program for the computation at the ith step can be written as,

```
DOALL i3 = 0, 2^(n-i) - 1
  DOALL i1 = 0, 2^(i-1) - 1
    T(2^i i3 + i1) = S(2^i i3 + i1) + S(2^i i3 + 2^(i-1) + i1)
    T(2^i i3 + 2^(i-1) + i1) = S(2^i i3 + i1) - S(2^i i3 + 2^(i-1) + i1)
ENDDOALL ENDDOALL
```

There are n computational steps. Therefore the complete program can be written as,

```
S <- X
DO i = 1, n
  DOALL i3 = 0, 2^(n-i) - 1
    DOALL i1 = 0, 2^(i-1) - 1
      T(2^i i3 + i1) = S(2^i i3 + i1) + S(2^i i3 + 2^(i-1) + i1)
      T(2^i i3 + 2^(i-1) + i1) = S(2^i i3 + i1) - S(2^i i3 + 2^(i-1) + i1)
  ENDDOALL ENDDOALL
  S <- T
ENDDO
Y <- S
```

where, X and Y are the input and output vectors respectively, and S and T are temporary arrays.

4.3.2 Efficient Tensor Product Transformations

There are two fundamental approaches for transforming the tensor product representation of a block recursive algorithm into an efficient formula on a given machine.

We call them *top-down* and *bottom-up* approaches, respectively. The top-down approach uses the following strategies. It finds an optimized tensor product representation by trying different possible tensor factorizations for the computational matrix. For example, to implement a block recursive algorithm on a vector machine, we keep each factored tensor product in the vector form by using stride permutations when necessary. To use the bottom-up approach, we first fully factor the computational matrix for a block recursive algorithm. Then the algebraic properties of tensor products and/or tensor bases are used to transform the tensor product formula to an equivalent formula which can be implemented efficiently on a given machine. In both cases, we need to estimate the performance of a simple tensor product on a given machine. This performance estimation and program synthesis for a simple tensor product are obtained by using the algebraic properties of tensor bases.

We next use the following example to further illustrate both approaches. Assume that we want to implement the Cooley-Tukey FFT algorithm in a distributed-memory machine with P processors.

Top-Down Approach

The top-down approach for program transformations uses the following three steps. The principal idea of this approach is similar to the methodology used by Johnson et al. in [44] for deriving efficient Fourier Transform (FT) implementations.

1. Determine which tensor product construction and the associated permutations have efficient implementations. The efficient implementations may require a specific data distribution.

2. Formulate an appropriate algorithm using tensor product notation. Use the properties of tensor products to transform the representation of the algorithm

to a formula which has an efficient implementation for each individual tensor product and also can minimize the cost of the overall computation.

3. Implement the expression obtained in (2) using the techniques developed in (1).

We now consider the problem we are trying to solve. To execute an application efficiently on a distributed-memory machine, we need to maximize parallelism and minimize communication. Therefore, the tensor products, which can run in parallel and each of their sub-computations needs only local data, have efficient implementations. To maximize parallelism, we transform the first factor in Formula (4.10) into the parallel form. The resulting formula is shown below,

$$F_{rs} = L_r^{rs}(I_s \otimes F_r)L_s^{rs}T_s^{rs}(I_r \otimes F_s)L_r^{rs}. \quad (4.18)$$

To minimize communication, we exploit the locality of the computation by appropriately choosing the values of r and s as well as the input data distribution. For example, by choosing $r = P$, $s = \frac{N}{P}$ and assuming that data is block distributed among processors, communication will only be required in three stages corresponding to the three stride permutations. The reasons are further explained as follows. When $r = P$, $s = \frac{N}{P}$, Formula (4.18) can be written as follows,

$$F_N = L_P^N(I_{\frac{N}{P}} \otimes F_P)L_{\frac{N}{P}}^N T_{\frac{N}{P}}^N(I_P \otimes F_{\frac{N}{P}})L_P^N. \quad (4.19)$$

Since data is block distributed and each $F_{\frac{N}{P}}$ in the second factor (beginning from right) is applied to the contiguous records of the inputs, each $F_{\frac{N}{P}}$ in the second factor can be computed locally. Similarly, each F_P in the fourth factor can also be computed locally. Twiddle factor scaling T_s^{rs} can be implemented by simply multiplying

each element, which does not introduce any communication if the twiddle factors are computed on-line.

Using this approach, further optimizations for local memory hierarchies such as cache can be obtained by appropriately factoring F_s and F_r. An example of using this approach to develop efficient FFT programs on a parallel DSP machine can be found in [56, 66].

Bottom-Up Approach

We begin with the fully factored Cooley-Tukey FFT algorithm represented by Formula (4.15), where, for simplicity, we have ignored the initial bit-reversal operation and the twiddle factor computation.

The ith step computation in Formula (4.15) is $I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}$. The input and the output bases for the ith step computation are the same and can be described as $e_{i_3}^{2^{n-i}} \otimes e_{i_2}^2 \otimes e_{i_1}^{2^{i-1}}$, where $e_{i_2}^2$ is the operator basis. It is obvious that each $F_2 \otimes I_{2^{i-1}}$ can be computed in parallel. In terms of tensor bases, the computations in the tensor product which are corresponding to $e_{i_2}^2 \otimes e_{i_1}^{2^{i-1}}$ can be computed in parallel. Moreover, in a distributed-memory machine, if data for each parallel computation are locally available, then each parallel computation can be computed by different processors without communication. The question is how to determine whether data are locally available. Moreover, can we distribute data to guarantee that data are locally available?

A methodology of using tensor bases to capture the semantics of the regular data distributions, such as block, cyclic, and block-cyclic distributions, on distributed-memory machines was proposed in [38]. A block distribution of 2^n elements on a 2^p processors' distributed-memory machine can be described by the following ten-

sor basis, $\rho_i^{2^p} \otimes e_j^{2^{n-p}}$, where we have used ρ to denote a processor basis. A "basis marker" algorithm was introduced in [37] to determine whether the ith step computation can be computed locally under certain data distributions. The principal idea is to check whether the operator basis in the input basis is overlapped with the processor basis in the data distribution basis. If they are overlapped, then communications are required. Otherwise, the computation can be computed locally without communication. By using the "basis marker" algorithm and assuming an initial block distribution, we can rewrite Formula (4.15) as follows,

$$F_{2^n} = \left(\prod_{i=p+1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}) \right) \left(\prod_{i=1}^{p} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}) \right) \tag{4.20}$$

All of the tensor products in the first group can now be computed without communication. If we keep the data distribution as the block distribution, then every tensor product in the second group needs communication. However, if we change the data distribution (called *data redistribution*) from the block to the cyclic data distribution after finishing the computations required by the tensor products in the first group, then all of the tensor products in the second group can now be computed without communication. The reason can be explained as follows. A cyclic distribution can be described by the tensor basis, $e_j^{2^{n-p}} \otimes \rho_i^{2^p}$. It is easy to see that every input (or output) basis for the tensor products in the second group is not overlapped with the operator basis in the data distribution basis of cyclic data distribution. The above analysis suggests that instead of keeping the block distribution and doing communications for each tensor product in the second group, we can change data distribution between the first and the second group computations. By doing so, only one communication step is required for the overall computations.

By examining the tensor bases for the block and the cyclic data distributions, we

can easily verify that changing data distribution from block to cyclic can be described by a stride permutation. Further, communication statements for this data redistribution can be generated from analyzing the indices in the input and the output data distribution bases. The details on how to generate communication statements and optimize this permutation on distributed-memory machine using the properties of tensor products can be found in [47].

In summary, for the core Cooley-Tukey FFT algorithm, by using an analysis based on the algebraic properties of tensor bases, we can produce an equivalent tensor product formula which can be implemented in two computational steps and one communication step.

In the methods of synthesizing efficient out-of-core programs discussed in the next chapters, we will use an instance of bottom-up approach for transforming tensor product formulas. However, instead of using "basis marker" techniques which is based on the properties of tensor bases, we use a greedy algorithm (or more generally, a dynamic programming algorithm) to transform the input tensor product formulas. The following example illustrates the basic ideas of our approach in the context of in-core computation. By using the properties of tensor products, Formula (4.20) can be further transformed as follows,

$$F_{2^n} = ((\otimes_{i=1}^{n-p} F_2) \otimes I_{2^p})(I_{2^{n-p}} \otimes (\otimes_{i=1}^{p} F_2)). \qquad (4.21)$$

There are two factors in the above formula. The first one can be implemented communication-free if we use a block distribution. The second factor can be implemented communication-free if we use a block-cyclic distribution $cyclic(2^i)$, where $1 \leq i \leq p$. We therefore result in the same conclusions that the core Cooley-Tukey FFT computation can be implemented in two computational steps and one communication step.

4.4 Conclusions

In this section, we have introduced tensor products, tensor bases and block recursive algorithms. We then discussed how to relate a tensor product with a program. We further classified the methods of program transformations for deriving efficient tensor product implementations into two approachs: top-down and bottom-up approaches.

M	size of the main memory
N	size of the inputs
D	number of the disks
B_d	size of a physical block
B	size of a logical block
B_b	number of the physical blocks in a logical block
$\mathcal{D}$	data distribution basis
$\mathcal{L}$	loop basis
β	input data distribution basis
λ	input loop basis
λ_n	portions of the input loop basis used to specify memory-loads
λ_m	portions of the input loop basis
λ_μ	portions of the input loop basis, whose vector bases do not generate loops
λ_{μ_1}	portion of $\mathcal{I}_\mu$, whose vector bases do not generate loops
λ_{μ_2}	portions of $\mathcal{I}_\mu$, whose vector bases are moved and used to generate loop nests
δ	output data distribution basis
θ	output loop basis
θ_n	portions of the output loop basis used to specify memory-loads
θ_m	portions of the output loop basis
θ_μ	portions of the output loop basis, whose vector bases do not generate loop nests
G	number of the logical tracks needed for storing the inputs
M_t	maximum number of physical tracks in a memory-load
N_t	number of physical tracks where the records for an A_V computation are stored
R_b	maximum number of desired records for an A_V computation in a physical block
R_t	number of desired records in a physical track
R_d	number of disks where desired records for an A_V are stored.
S	stride of physical tracks which contain desired records
b_b	index for the physical blocks
b_d	index for the records inside a physical block
d	index for disks
g	index for G
n	logarithm of N

Table 4.1: Summary of the symbols used in Chapter 5 and Chapter 6.

Chapter 5

Semantics of Out-of-Core Data Distributions and Access Patterns

In this chapter, we discuss how data are stored on multiple disks and accessed by processors on both the single-processor multi-disk and the multi-processor multi-disk systems. For simplicity, we will focus on how to load/store data from/to multiple disks and ignore how data are distributed among internal memories. This assumption is equivalent to considering how to access out-of-core data for a single-processor multi-disk system. However, the method developed in this chapter is also applicable to the multi-processor multi-disk systems. The reason is that for the multi-processor multi-disk systems, our approach of program synthesis first concentrates on how to load/store data from/to multiple disks and then determines how data are distributed among processors.

We first introduce a tensor basis, called a *data distribution basis*, to describe data organization on the multi-disk systems. Then, we use the algebraic properties of tensor bases to describe the semantics of different data access patterns. Further, we show how to use the semantic information of different data access patterns captured by tensor bases to implement parallel data access supported by our multi-disk models.

Selected symbols used in this chapter and the next chapter are summarized in Table 4.1.

5.1 Data Organization on Multi-Disk Systems

We first introduce several parameters to describe the multi-disk systems. For the multi-disk system, pictured in either Fig. 1.2 (A) or Fig. 1.2 (B), we use two parameters, the number of disks D and the size of each *physical block* B_d, to describe physical properties of disks. The physical blocks which have the same relative positions on each disk constitute a *physical track*. The physical tracks are numbered contiguously with the outermost track having the lowest address and the innermost track having the highest address. The ith physical track is denoted by T_i.

A typical way to store inputs on this multi-disk system can be described as follows. The inputs are first organized as blocks. Each block has the same size as the size of the physical block. Then, the blocks are assigned to disks in a round-robin manner beginning from the first physical track. In the terminology of the High Performance Fortran, this data organization is called a block-cyclic data distribution and is denoted as $cyclic(B_d)$. However, this organization differs from the HPF in that now data is distributed among disks rather than processors. Fig. 5.1 shows an example $cyclic(4)$ data layout with $B_d = 4$, $D = 4$, and $N = 64$. In that figure, each column is a disk; each box is a physical block; each row consists of a physical track; and the numbers in each box denote the record indices.

Moreover, we can assume that data can be distributed with an arbitrary block size. (Tom Cormen has called this data organization on disks as a banded data layout [22] and studied the performance for a class of permutations and several other basic primitives of the NESL language [11].) Fig. 5.2 shows the data organization for the same

	D_0	D_1	D_2	D_3
T_0	0 1 2 3	4 5 6 7	8 9 10 11	12 13 14 15
T_1	16 17 18 19	20 21 22 23	24 25 26 27	28 29 30 31
T_2	32 33 34 35	36 37 38 39	40 41 42 43	44 45 46 47
T_3	48 49 50 51	52 53 54 55	56 57 58 59	60 61 62 63

Figure 5.1: Data organization on multi-disks for $N = 64$, $B_d = 4$, and $D = 4$.

values of the parameters D, B_d, and N as in Fig. 5.1, but with a $cyclic(8)$ distribution. Note that the size of the physical track and the size of the physical block are not changed. However, they now contain different records. We will call B records in a block formed by a $cyclic(B)$ distribution as a *logical block*. In that figure, the first left shadowed box denotes a typical logical block. Similarly, the logical blocks which have the same relative positions on each disk constitute a *logical track*. The ith logical track is denoted as LT_i. In that figure, there are two logical tracks LT_0 and LT_1. Each of them consists of two physical tracks.

In the rest of the discussion, for simplicity, we make the following assumptions. The input and the output data are stored in the separate sets of disks. All parameters are powers of two. (The results can be easily generalized to allow all parameters to be powers of any integer.)

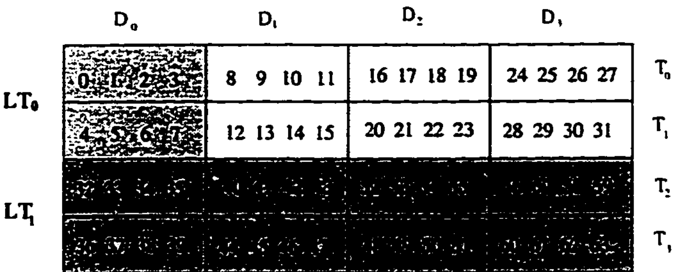


Figure 5.2: Data organization on multi-disks for $N = 64$, $B_d = 4$, $D = 4$, and $B = 8$.

5.2 Tensor Bases for Data Distributions and Access Patterns

5.2.1 Data Distribution Bases

As discussed in [37], a block-cyclic distribution can be algebraically represented by a tensor basis. That approach can be adopted to the disk model by substituting disks for processors. However, because of the existence of physical blocks and physical tracks, the method of using tensor bases to define a block-cyclic distribution for multiprocessors needs to be generalized. We achieve this generalization by further factoring the tensor basis. We call this factored tensor basis an (out-of-core) *data distribution basis*, which is defined as follows:

Definition 2.6 *Let $B = B_b B_d$. If a vector of length N, where $N = GBD$ and G is an integer, is distributed according to the cyclic(B) distribution on D disks, then its data distribution basis is defined as:*

$$\mathcal{D} = e_g^G \otimes e_d^D \otimes e_{b_b}^{B_b} \otimes e_{b_d}^{B_d}. \tag{5.1}$$

We use $\mathcal{D}(s)$ to refer to the sth factor (from the left), e.g., $\mathcal{D}(2) = e_d^D$.

We can view the inputs stored on disks using the *cyclic*(B) distribution as a four-dimensional array. Moreover, each dimension is denoted by a vector basis of the data distribution basis of Formula (5.1). For example, the first dimension is along logical tracks and e_g^G denotes the index for accessing logical tracks. The second dimension is along disks and e_d^D denotes the index for accessing disks. The third dimension denotes the number of physical tracks in a logical block and is represented by the third vector basis $e_{b_b}^{B_b}$. The fourth dimension denotes the number of records in a physical block and is represented by the fourth vector basis $e_{b_d}^{B_d}$.

The collections of the instantiation of the indices g, d, b_b, and b_d in Formula(5.1) give the indices of all the inputs. In order words, we use $\mathcal{D}$ to denote all of the records stored on secondary storage rather than a single $[g, d, b_b, b_d]$-th element.

For example, the data distribution basis for Figure 5.2 is $e_g^2 \otimes e_d^4 \otimes e_{b_b}^2 \otimes e_{b_d}^4$, where the size of each physical block is four; each logical block contains two physical blocks; there are four disks; and the inputs are stored on two logical tracks. The data distribution basis for Figure 5.1 can be written as $e_g^4 \otimes e_d^4 \otimes e_{b_d}^4$, where $B_b = 1$. A selected portion of the distribution basis in Formula (5.1) can be used to obtain the indexing function needed to denote a particular data unit such as a logical track or a physical track. Let,

$$\texttt{logical-track-basis}(\mathcal{D}) = e_g^G \tag{5.2}$$

$$\texttt{physical-track-basis}(\mathcal{D}) = e_g^G \otimes e_{b_b}^{B_b} \tag{5.3}$$

Then the indexing function for accessing the physical tracks can be obtained by linearizing `physical-track-basis`$(\mathcal{D})$. By taking the *difference* of the data distribution basis with each of them, we can have tensor bases which denote the records inside a logical track and a physical track, respectively. These tensor bases are called the *logical track-element basis* $(e_d^D \otimes e_{b_b}^{B_b} \otimes e_{b_d}^{B_d})$ and the *physical track-element basis*

$(e_d^D \otimes e_{b_d}^{B_d})$, respectively. This concept can be further extended to denote other subsets of data which we will discuss later.

5.2.2 Tensor Bases for Data Access

When the data distribution is determined, different orders of instantiating the indices in the indexing function of the data distribution basis defined in Formula (5.1) can result in different access patterns for out-of-core data. For example, if we instantiate the indices in the order from right to left (which is what we have used to interpret a tensor basis so far), i.e. g is the slowest and b_d is the fastest changing indices, then we actually access data first in the first logical block in the first disk and then access the first logical block in the second disk. After finishing the access to the first logical track sequentially, the second logical track is accessed, and so on. This data access pattern can be better understood by examining the following code, which uses the indices in each vector basis as an iterative variable.

```
DO g = 0, G - 1
  DO d = 0, D
    DO b_b = 0, B_b - 1
      DO b_d = 0, B_d - 1
        read(gB_bDB_d + dB_bB_d + b_bB_d + b_d)
ENDDO ENDDO ENDDO ENDDO
```

If we instantiate the index b_b in $e_{b_b}^{B_b}$ after the index d in e_d^D in Formula (5.1), then it results in an access pattern where first the data along a physical track is accessed and then the successive physical tracks are accessed. This change in the instantiation order of the indices can be regarded as a permutation of the data distribution basis. We will call a permutation of a data distribution basis as a *loop basis*. For the above

example, the loop basis can be denoted as,

$$\mathcal{L} = e_g^G \otimes e_{b_b}^{B_b} \otimes e_d^D \otimes e_{b_d}^{B_d} \tag{5.4}$$

Data distribution bases and loop bases together specify a specific data access pattern. To synthesize a program with this data access pattern, every index in a loop basis may be used to generate a loop nest. Moreover, the order of the loop nests is determined by the order of the vector bases in the loop basis. A program which can access out-of-core data specified by the loop basis denoted by Formula (5.4) is shown in below.

```
DO g = 0, G − 1
  DO b_b = 0, B_b − 1
    DO d = 0, D
      DO b_d = 0, B_d − 1
        read(gB_bDB_d + dB_bB_d + b_bB_d + b_d)
ENDDO ENDDO ENDDO ENDDO
```

Note that in the above program, the indexing function for accessing each record is obtained by linearizing the data distribution basis. The order of loops is specified by the loop basis. In terms of programs, a loop basis can be understood as a notation specifying how to re-order the loop nests and further how to split a loop nest [85].

As we have mentioned, out-of-core data can be viewed as organized as a four-dimensional structure. For example, the data layouts in Fig. 5.2 can be viewed as a four-dimensional array, where each logical block is viewed as a $B_b \times B_d$ matrix. Further we can convert B_d to a two-dimensional structure. We can combine contiguous records in some of the disks in the same physical track together as a sub-matrix. These other views can be denoted by factoring and regrouping data distribution bases and can be used to form different data access patterns. Fig. 5.3 shows an example

Figure 5.3: An example data organization and access pattern.

of data organization and access pattern. We assume that $D = 4$, $G = 4$, $B = 2B_d$, where each logical track consists of two physical tracks. We further decompose each physical block B_d into two sub-blocks, each of them with size B_{d_1}. In order to reflect this data organization, we can factor the data distribution basis as follows,

$$\mathcal{D} = e_g^G \otimes e_d^D \otimes e_{b_b}^{B_b} \otimes e_{b_{d_2}}^{\frac{B_d}{B_{d_1}}} \otimes e_{b_{d_1}}^{B_{d_1}} \tag{5.5}$$

After permuting this factored data distribution basis, we can have loop bases which access subsets of data in different patterns. Assume that we want to access data in Fig. 5.3 in the following order: first access the darker shadowed sub-blocks in row-major order and then access the lighter shadowed sub-blocks in row-major order. Then we can move $\mathcal{D}(3)$ and $\mathcal{D}(4)$ before $\mathcal{D}(2)$ and $\mathcal{D}(1)$, respectively. This results in the following loop basis,

$$\mathcal{L} = e_{b_{d_2}}^{\frac{B_d}{B_{d_1}}} \otimes e_g^G \otimes e_{b_b}^{B_b} \otimes e_d^D \otimes e_{b_{d_1}}^{B_{d_1}} \tag{5.6}$$

We can verify the correctness of this loop basis as follows. There are five indices corresponding to the five vector bases in this loop basis. The index b_{d_2} chooses the same shadowed sub-blocks. The index g chooses the logical tracks. The index b_b

chooses the logical blocks. The index d chooses disks. The index b_{d_1} chooses the records inside a sub-block. Since the increasing of the indices is in the reversed order of the above five steps, the loop basis specifies an order which accesses records inside the first darker shadowed sub-block in the first disk and then accesses the records in the same shadowed sub-block in the second disk, and so on. After accessing all the records in the same darker shadowed sub-blocks in row-major order, we repeat the procedure for the lighter shadowed sub-blocks.

5.3 Parallel Data Access to Multi-Disk Systems

We now discuss how those data access patterns described by tensor bases can be used by the multi-disk system to access out-of-core data. We first discuss how data is accessed in our parallel multi-disk models.

5.3.1 Parallel I/O Operations on Multi-Disk Systems

The critical issue is the definition of the data unit, on which a parallel I/O operation is applied. The assumption in our multi-disk systems is that each parallel I/O operation can read or write a physical track. This assumption follows the assumption of the striped two-level memory model of Vitter and Shriver.

Using the physical track as the unit of a parallel I/O operation, parallelism in data access is at two levels: elements in one physical block are transferred concurrently and D physical blocks can be transferred in one I/O operation. However, because we use the *striped disk* access model, physical blocks in one I/O operation come from the same track, as opposed to the *independent I/O* model in which blocks can come from different tracks. Moreover, we will use the parallel primitives, *parallel_read*(i) and *parallel_write*(i), to denote a read from and a write to the physical track T_i, re-

spectively.

Note that for various block-cyclic data distributions, in which the size of a data block is not equal to the size of a physical block, each parallel I/O operation still accesses a physical track not a logical track. Hence, several parallel I/O operations are needed to access a logical track. For example, to load the logical track LT_1 in Fig. 5.2, two *parallel_read* operations $parallel_read(2)$ and $parallel_read(3)$, which respectively load the physical tracks T_2 and T_3, are needed.

5.3.2 Generating Parallel I/O Operations Using Tensor Bases

In this subsection, we discuss how to use the algebraic properties of tensor bases, such as data distribution bases and loop bases, to generate parallel I/O operations for accessing out-of-core data on the multi-disk model.

By the definition of loop bases, each index in a loop basis may correspond to a loop in the synthesized program. However, under our striped I/O model, each I/O operation will read or store all the records in a physical track. Therefore only part of the loop basis will explicitly appear in the synthesized program. For example, if we use parallel I/O operations to access out-of-core data in the order determined by the loop basis in Formula (5.4), then we do not need to generate loop nests for the indices d and b_d. The vector bases e_d^D and $e_{b_d}^{B_d}$ together denote the records inside a physical track, which is loaded as a unit in our models. A program using parallel I/O operations to implement the same data access pattern is shown below,

DO $g = 0, G$
  DO $b_b = 0, B_b - 1$
    $parallel_read(gB_b + b_b)$
ENDDO ENDDO

Moreover, in addition to determine an efficient data access pattern for an out-of-core computation, we also have to consider the limited size of main memory. We need to determine which portions of data being accessed by a specific data access pattern should be kept in main memory, such that this subset of data is needed by the current computation. We call each such subset of data a *memory-load*. Therefore, in general, an out-of-core computation is decomposed as a set of sub-computations and then each sub-computation will operate on a memory-load each time. Thus, the question is how to construct a memory-load using tensor bases under the constraints of the limited size of main memory. We next use several examples to illustrate our answers to this question.

In the following discussion, we assume that the size of main memory is one half of the size of the inputs. Consider the data access pattern described by the loop basis in Formula (5.4). If we assume that each sub-computation is applied to one half of the inputs loaded in the order specified by the loop basis in Formula (5.4), then to construct a memory-load, we can simply split the outermost loop nest into two loop nests as shown in the following program,

```
DO g_1 = 0, G_1 - 1
  DO g_2 = 0, G_2 - 1
    DO b_b = 0, B_b - 1
      parallel_read(g_1 B_b G_2 + g_2 B_b + b_b)
ENDDO ENDDO ENDDO
```

which is equivalent to further factoring the vector basis e_g^G as $e_{g_1}^{G_1} \otimes e_{g_2}^{G_2}$, where $G_1 = \sqrt{G}$ and $G_2 = \sqrt{G}$. The inside two loops are used to construct a memory-load. However, the procedure of constructing a memory-load can be much more sophisticated in the following situation.

The number of the physical tracks, where the records required by a sub-computation are stored, exceeds the number of the physical tracks which the internal memory can hold.

The implications of this situation can be explained as follows. First, we can not keep all of the records in a loaded physical track and we need to decide which portions of the loaded physical tracks should be used to construct the current memory-load. Second, the same physical tracks need to be loaded several times, and each time different portions of the loaded tracks are used to construct a memory-load. Consider Fig. 5.3 and assume that there are two sub-computations, each of them needing the data in the same shadowed region. Because the unit of a parallel I/O operation is a physical track and the size of main memory is limited, only half of the data in a loaded physical track can be kept and is actually needed. Therefore, for each loaded physical track, half of the data needs to be discarded. Further, to finish two sub-computations, we need to load each physical track two times.

In terms of tensor bases, loading the same track several times can be achieved by looping over some of the vector bases in the physical track-element basis. For this example, by making $\lambda(4)$ as the first factor in the data distribution basis to form a loop basis, the same track is loaded $\frac{B_d}{B_{d_1}}$ times, since the size of $\lambda(4)$ is $\frac{B_d}{B_{d_1}}$. Moreover, the records which should be kept for each loaded physical track can also be determined by the vector basis. In this case, it is $\lambda(4)$ or $e_{b_{d_2}}^{\frac{B_d}{B_{d_1}}}$. Each instantiation of the index b_{d_2} determines the sub-blocks in a physical track which should be kept for the current memory-load. The detailed program for loading out-of-core data and constructing memory-loads is shown below. In general, determining each of sub-blocks which should be kept is equivalent to computing the sub-array from the input array (factored from the input vector). In Appendix A, we will present a procedure of keeping sub-

arrays in a physical track for the current memory-load using tensor bases.

DO $b_{d_2} = 0, \frac{B_d}{B_{d_1}} - 1$
  DO $g = 0, G - 1$
    // construct a memory-load
    DO $b_b = 0, B_b - 1$
      $A(0 : B_d D - 1) \leftarrow parallel_read(gB_b + b_b)$
      // keep portions of records for current memory-load
      DO $d = 0, D - 1$
        $X(dB_{d_1} : (d+1)B_{d_1} - 1)$
          $\leftarrow A(dB_d + b_{d_2} B_{d_1} : dB_d + b_{d_2} B_{d_1} + B_{d_1} - 1)$
      ENDDO
ENDDO ENDDO ENDDO

where A is a temporary array for holding a physical track and X holds a memory-load.

Chapter 6

Synthesizing Out-of-Core Programs for Single-Processor Multi-Disk Systems

In this chapter, we present a framework for using tensor products to synthesize programs for block recursive algorithms for the single-processor multi-disk model. This model is equivalent to Vitter and Shriver's striped two-level memory model. However, it permits various block-cyclic distributions of out-of-core data on disks. We investigate the implications of various block-cyclic distributions $\mathit{cyclic}(B)$ on the performance of out-of-core block recursive algorithms, such as the fast Fourier transform (FFT) and matrix transposition algorithms.

Synthesizing I/O efficient programs for a tensor product formula may introduce stride permutations. We therefore first present procedures for synthesizing efficient out-of-core programs for both tensor products and stride permutations using a $\mathit{cyclic}(B)$ distribution of the data. We then discuss several strategies, such as factor grouping and data rearrangement, to improve the performance for tensor product formulas. We formalize the procedure of synthesizing efficient out-of-core programs for tensor product formulas with various data distributions as a dynamic programming problem.

However, since data rearrangement, which can be described by a stride permutation, is too expensive to implement on our target model as discussed in Section 7, we have not incorporated it into our dynamic programming approach. In this sense, the procedure of synthesizing I/O efficient programs for stride permutations should be mainly understood as a method of program synthesis for matrix transpositions – a stride permutation can be interpreted as a matrix transposition. Because a stride permutation can be interpreted as a matrix transposition, synthesizing efficient out-of-core programs for stride permutations is important in itself [76, 28, 48].

We illustrate the effectiveness of this dynamic programming approach through an example out-of-core FFT program. We further examine the performance issues of synthesized programs. We show that:

1. The choice of data distribution has a large influence on the performance of the synthesized programs,
2. Our simple algorithm for selecting an appropriate size of data distribution is very effective, and
3. The dynamic programming approach can always reduce the number of passes to access out-of-core data compared with the greedy algorithm discussed in [39].

As we discussed in Chapter 4, a stride permutation can be factored and then represented as a tensor product formula. Therefore, it can be computed using the same method used for tensor product formulas. However, in this chapter, we will present another method of directly computing a stride permutation. The main advantage of this method is that we can easily predict the performance of the stride permutation on our parallel I/O model.

6.1 Machine Model and Performance Metrics

The single-processor multi-disk model used in this chapter is equivalent to the striped two-level memory model of Vitter and Shriver [83]. Moreover, as we discussed in Chapter 5, in our model, data on disks can be distributed in different (logical) block sizes.

We now give a formal definition of the model. The model consists of a processor with an internal random access memory and a set of disks. The storage capacity of each disk is infinite. Each disk is organized as blocks. Each block has a fixed size and is called a *physical* block. Five parameters: N (the size of the input), M (the size of main memory), B_d (the size of each *physical* block), D (the number of disks), and B (the size of each *logical* block) are used in this model. We assume that $M < N$, $1 \leq B_d \leq \frac{M}{2}$, $1 \leq D \leq \frac{M}{B_d}$ and $B_d \leq B$.

In this model, parallel I/Os occur in a physical track of size DB_d. Each parallel operation, such as *parallel_read*(i) and *parallel_write*(i), will read or write physical track i. We define the performance of programs as the number of parallel I/Os required.

As we have discussed in the previous chapter, in this model we also allow logical track distribution. We next present a simple example to show the advantages of using logical distributions on developing I/O-efficient programs for block recursive algorithms.

Why Logical Data Distributions? Assume that we want to implement $F_8 \otimes I_8$ on our target model under the parameters given in Fig. 5.1. Further, we assume that the size of main memory is one half of the size of the inputs. Because we are mainly interested in data access patterns, we ignore the real computations specified by F_8.

The only thing we need to remember is that F_8 needs eight elements with a stride of eight because of the existence of the identity matrix I_8.

We first consider implementing $F_8 \otimes I_8$ on the physical block distribution. From the above discussion, we know that the first F_8 needs to be applied to eight elements: $0, 8, 16, 24, 32, 40, 48,$ and 56. From Fig. 5.1, we can see that these elements required by the first F_8 computation are stored on four physical tracks. However, our main memory can hold only two physical tracks, so that we can not simply load all of the four physical tracks into the main memory and accomplish the computation in one pass of I/O. To get around this memory limitation, we can use two different approaches.

First, we load the first physical track and keep the first half of the records in each physical block in that loaded physical track and discard the other half of the records. We do this for every physical track. Then we do the computation for the remaining records in the main memory. After finishing the computation for these records, we write the results out. Then we repeat the above procedure. However, we now keep the other half of the records in the main memory for each loaded track. By doing the computation in this way, it is obviously that we need two passes to load out-of-core data.

Another method is to use logical block distribution. Suppose that the size of a logical block is eight as shown in Fig. 5.2. Now, the eight records required by one F_8 are stored on two physical tracks, physical track one and three, or physical track two and four. Therefore, if we can load physical tracks one and three first and do the computation, then load physical track two and four and do the computation, we can finish the computation in one pass. This example clearly shows the advantages of using logical distributions comparing with using only physical track distributions.

However, there are several problems which we have not addressed here, such as how to determine the block size of logical distributions and how to determine data access patterns. We will discuss these issues in the rest of the chapter.

Some other details including the semantics of data access patterns, the definition of data distribution bases, and the definition of loop bases, can be found in Chapter 5. In addition, for simplicity, we assume that the block size B of the logical distribution is a multiple of B_d.

6.2 Overview of Program Synthesis

As we have discussed in the introduction, our method of program synthesis consists of three steps. Efficient implementations of block recursive algorithms are obtained by first using the properties of tensor products to transform tensor product representations for the block recursive algorithms. In addition to transforming tensor product formulas, the first step will also use an algorithm to determine the efficient data distribution. The second step takes the transformed tensor product formula and the information on data distributions to generate augmented tensor bases for each computational step. For multi-processor multi-disk systems, the augmented tensor basis consists of the following four components: data distribution bases, loop bases, subcomputations and memory-loads. These four components are then used by the third step of the code generation algorithm to generate parallel I/O programs.

If the input tensor product formula consists of only a stride permutation, then the first step of the program transformation will only determine efficient data distributions. Then it will use the method presented in Section 6.3.2 to generate an augmented tensor basis. However, the code generation step for both tensor products and stride permutations will use the same procedure presented in Section 6.3.1.

As we have mentioned, our presentation of the derivation of efficient implementations for the block recursive algorithms is in the reverse order of Fig. 1.3. We first present a procedure for code generation by using the information contained in the augmented tensor basis. Then we determine efficient implementations for a stride permutation and a simple tensor product with a given data distribution on a given model by determining the corresponding augmented tensor bases. Further, we develop a simple algorithm to determine the data distribution which can result in an efficient implementation. Furthermore, we use a dynamic (or a multi-step dynamic) programming algorithm to determine an efficient implementation for the block recursive algorithms. The dynamic programming algorithm will use the properties of tensor products and the performance of each tensor product. The method of estimating the performance for each tensor product will be presented in Section 6.3.2 and Section 6.3.3 with the analysis of the second step (determining augmented tensor bases).

6.3 Synthesizing I/O-Efficient Programs

6.3.1 Parallel I/O Code Generation

In this subsection, we consider the task of generating parallel I/O code for a tensor product assuming that the augmented tensor basis is given. An augmented tensor basis for a single-processor multi-disk system includes data distribution bases, loop bases, memory-loads and operations on each memory-load. Moreover, for a tensor product computation, the input and output data may be organized and accessed differently. We therefore need to use *input data distribution basis* β, *output data distribution basis* δ, *input loop basis* λ, and *output loop basis* θ to denote them respectively. Before we present the detailed definition of the augmented tensor basis, we

first discuss the structure of the program synthesized by the second step and how to construct loop bases to generate I/O programs with that structure.

To minimize the number of I/O operations for a synthesized program for a tensor product, we need to exploit locality by reusing the loaded data. This requires decomposing the computation and reorganizing data and data access patterns to maximize data reuse. In the synthesized program, the same sub-computation is performed several times over different data sets. Hence, the loop structure of the synthesized program is constructed as follows. An outer loop nest enclosing three inner loop nests: *read loop nest* for reading in the data, *computation loop nest* for performing sub-computation on the loaded data, and *write loop nest* for writing the output data back to the disk. The inner read loop nest should load out-of-core data without overflowing main memory. The inner computation loop nest should perform sub-computation on a memory-load. And the data sets should be accessed using parallel primitives, *parallel_read* and *parallel_write*, to load or store a physical track each time.

To reflect the structure of the outer and inner loops described above, we need to separate loop bases into several different parts. More specifically, we first separate the input loop basis into two parts such that the first part specifies memory-loads and the second part specifies the records inside a memory-load. As we discussed in Chapter 5, under our striped I/O model, each I/O operation will read or store all the records in a physical track each time. Hence, only part of the indices in the loop basis will appear explicitly in the synthesized program. We therefore further separate the second part of the loop basis into two parts: one part, denoted as λ_m, is used to construct a memory-load and another part, denoted as λ_μ, will not generate loop nests in the synthesized program. In other words, we can write the input loop basis as follows:

$$\lambda = \lambda_n \otimes \lambda_m \otimes \lambda_\mu, \tag{6.1}$$

where, we call λ_n a *memory basis*, since each instantiation of the indices in λ_n corresponds to a memory-load. Similarly, we can separate the output loop basis as follows,

$$\theta = \theta_n \otimes \theta_m \otimes \theta_\mu . \tag{6.2}$$

Moreover, our method of determining loop bases will guarantee that θ_n is a permutation of λ_n. Thus, we will let $\theta_n = \lambda_n$ and we can use the indices in λ_n to generate the outermost loops.

Let us further examine Formulas (6.1) and (6.2). Obviously, if λ_μ and θ_μ consist of the physical-track-element bases for input and output data respectively, then the out-of-core data needs to be accessed only once. In terms of memory-loads, each memory-load has the following properties. The input for each memory-load occupies all of the locations in a set of physical tracks specified by the input data distribution basis. And after computing these records in the main memory, they are organized to occupy all of the locations in a set of physical tracks specified by the output data distribution basis. We call this type of memory-load a *perfect memory-load.* If we can construct memory-loads in this manner, then we can synthesize a program which accesses out-of-core data only once (called a *one-pass program*).

However, it may not be possible to construct perfect memory-loads for some computations. In that case, we may need to keep only part of the records from a loaded physical track in main memory and discard other records. Therefore, a *multiple-pass program* needs to be synthesized in which the same physical track is loaded several times, as we discussed in the last section of Chapter 5.

More specifically, if we assume that the initial loop bases λ and θ have the properties that λ_μ and θ_μ consist of the physical-track-element bases from the input and the output data distribution bases, respectively, then we may need to further separate

1. Generate loops for indices in λ_n
2. Generate loops for indices in λ_m
3. *Parallel_read* using input distribution basis
4. Keep records for current memory-load
5. End the loops corresponding to λ_m
6. Perform operations to a memory-load
7. Generate loops for indices in θ_m
8. *Parallel_write* using output distribution basis
9. End the loops corresponding to θ_m
10. End loops corresponding to λ_n

Figure 6.1: Procedure of code generation for a tensor product.

λ_μ (or θ_μ) into two parts, λ_{μ_1} and λ_{μ_2}. We then take λ_{μ_2} out of λ_μ and put it into λ_n. This moved tensor basis λ_{μ_2} is used to determine which portions of a physical block should be kept for the current memory-load. The size of this moved vector basis is equal to the number of times the same physical tracks are loaded.

We now give the following definition of an augmented tensor basis.

Definition 3.7 *An augmented tensor basis constitutes the following four components,*

1. *Data distribution basis. Let data be distributed by cyclic(B) on D disks. Let $B = B_b B_d$ and the number of data elements be N, where $N = GBD$. Then the (input or output) data distribution basis can be written as:*

$$\mathcal{D} = e_g^G \otimes e_d^D \otimes e_{b_b}^{B_b} \otimes e_{b_d}^{B_d}. \tag{6.3}$$

2. *Loop Basis. An (input or output) loop basis has the following generic form,*

$$\mathcal{L} = \mathcal{L}_n \otimes \mathcal{L}_m \otimes \mathcal{L}_{\mu_1} \tag{6.4}$$

where,

- *$\mathcal{L}_{\mu_1}$ is a subset of $\mathcal{L}_\mu$, where $\mathcal{L}_\mu = \mathcal{D}_2 \otimes \mathcal{D}_4$ and $\mathcal{L}_{\mu_1} = \mathcal{L}_\mu - \mathcal{L}_{\mu_2}$;*
- *$\mathcal{L}_m$ consists of the last portions of $\mathcal{D} - \mathcal{L}_{\mu_1}$ such that $\mid \mathcal{L}_m \mid = \frac{M}{|\mathcal{L}_{\mu_1}|}$;*
- *$\mathcal{L}_n = \mathcal{D} - \mathcal{L}_m - \mathcal{L}_{\mu_1}$.*

3. *Memory-load. The records in each memory-load are denoted by $\mathcal{L}_m \otimes \mathcal{L}_{\mu_1}$. More specifically, each memory-load is obtained by an instantiation of indices in $\mathcal{L}_n$, looping over indices in $\mathcal{L}_m$, and using $\mathcal{L}_{\mu_2}$ to identify which portions in each loaded physical track should be kept for the current memory-load.*
4. *Sub-computation. The decomposed computation which will be applied to each memory-load.*

Note that the input and the output data distribution bases can be different. Moreover, the input data distribution basis can be obtained by factoring the input basis. The output data distribution basis can be obtained by applying the corresponding tensor product or stride permutation to the input data distribution basis.

Using this augmented tensor basis and assuming that $\theta_n = \lambda_n$, a generic program can then be obtained as described in Fig. 6.1. Note that when we *parallel_read* a track, the track number is obtained from the indexing function of physical-track-basis(β) (part of the input data distribution basis) as defined in Formula (5.3). The *parallel_write* is similar in this respect. Also, several details may need to be incorporated into the procedure of Fig. 6.1. For example, we need a sub-routine to keep the records of the loaded tracks in the current memory-load by using the indices in λ_{μ_2}, which is now instantiated in λ_n. We will discuss this sub-routine in Appendix A. In addition, as we will discussed in the next subsection, the write-out portions of the procedure also need to be modified slightly.

Fig. 6.2 shows an example synthesized program for $I_4 \otimes F_2 \otimes I_4$. We assume that $M = 16$, $D = 2$, $B_d = 2$, $B = 2$, F_2 is a 2×2 matrix, and data are distributed in

```
DO g2 = 0, 1
   DO g1 = 0, 3
     // Parallel read from a track
     X(4g1 : 4g1 + 3) <- parallel_read(4g2 + g1)
   ENDDO
   // Perform operations for a memory-load
   Code(X(1 : 16) <- A x X(1 : 16))
   // Write the result back
   DO g1 = 0, 3
     // Parallel write to a track
     parallel_write(4g2 + g1) <- X(4g1 : (4g1 + 3))
ENDDO ENDDO
```

Figure 6.2: Code for $I_4 \otimes F_2 \otimes I_4$, where X is an array of size M and $A = I_2 \otimes F_2 \otimes I_4$.

a *cyclic*(2) manner. It uses $e_g^8 \otimes e_d^2 \otimes e_{b_d}^2$ as both the input and the output distribution bases. The input and the output loop bases are also the same as $e_{g_2}^2 \otimes e_{g_1}^4 \otimes e_d^2 \otimes e_{b_d}^2$, where $e_{g_2}^2 \otimes e_{g_1}^4$ is a factorization of e_g^8. The sub-computation is denoted by $I_2 \otimes F_2 \otimes I_4$. The memory basis is $e_{g_2}^2$. The details of how to determine this information are discussed in Section 6.3.3.

6.3.2 Synthesizing Programs for Stride Permutations

In this subsection, we discuss how to determine an efficient augmented tensor basis for stride permutations using a *cyclic*(B) distribution. The performance of synthesized programs will be represented as a function of the size of a sub-tensor basis, whose value can be obtained when the size of the data distribution is given.

As we have mentioned, our goal is to decompose computations into a sequence of sub-computations performed on perfect memory-loads. However, this may not always be possible because of the limited memory size. In that case, we minimize the

number of times the data is loaded for each memory-load as well as ensuring that each physical track of the output is written only once in parallel. We will develop an approach to determining the input and output loop bases for the given distribution *cyclic*(B). Based on these loop bases and data distribution bases, we determine memory-loads and operations on the memory-loads. Following this a program can be synthesized by using the procedure presented in Section 6.3.1. The cost of the program can also be determined from the loop bases. We summarize our results in the following theorem and then present a constructive proof, which constructs the augmented tensor basis.

Theorem 3.2 *Let* $Y = L_Q^{PQ} X$, *where* $PQ = N$ *and* X *and* Y *are input and output vectors with length* N, *respectively. Let* X *and* Y *be distributed according to* *cyclic*(B) *and the data distribution bases be denoted as* β *and* δ, *respectively. Further let* $\lambda_\mu = \beta(2)\otimes\beta(4)$ *and* $\theta_\mu = \delta(2)\otimes\delta(4)$. *Then a program can be synthesized with* $\frac{N}{B_d D}(1 + max\{1, \lceil\frac{|\lambda_\mu - \theta_\mu| B_d D}{M}\rceil\})$ *parallel I/O operations for the stride permutation* $Y = L_Q^{PQ} X$.

Proof: We present an algorithm as shown in Fig. 6.3 for determining the input and the output loop bases. The algorithm is further explained in Step 1 as shown below. In Step 2 and Step 3, we show how to construct memory-loads and operations for a memory-load. In Step 4, we show that I/O costs can be obtained from this information.

1. **Determine input and output loop bases.** We begin with the following construction for the input and the output loop bases,

$$\lambda = (\lambda - (\theta_\mu - \lambda_\mu) - \lambda_\mu)\otimes(\theta_\mu - \lambda_\mu)\otimes\lambda_\mu, \tag{6.5}$$

$$\theta = (\theta - (\lambda_\mu - \theta_\mu) - \theta_\mu)\otimes(\lambda_\mu - \theta_\mu)\otimes\theta_\mu \tag{6.6}$$

// Initialization
$\lambda = \beta(1)\otimes\beta(3)\otimes\beta(2)\otimes\beta(4)$
$\theta = \delta(1)\otimes\delta(3)\otimes\delta(2)\otimes\delta(4)$
$\lambda_\mu = \beta(2)\otimes\beta(4), \theta_\mu = \delta(2)\otimes\delta(4)$
$\lambda_m = \theta_\mu - \lambda_\mu, \theta_m = \lambda_\mu - \theta_\mu$
$\lambda_{\mu_1} = \lambda_\mu, \lambda_{\mu_2} = \phi$
// One-pass or multiple-pass implementation
if (| $(\theta_\mu - \lambda_\mu)\otimes\lambda_\mu$ |$\leq M$) then
  $\lambda_n = \lambda - \lambda_m - \lambda_\mu$
else
  Factor $(\lambda_\mu - \theta_\mu)$ such that θ_m consists of the last
  factors of the factored tensor basis and $| \theta_m | = \frac{M}{B_d D}$.
  $\lambda_{\mu_2}=(\lambda_\mu - \theta_\mu) - \theta_m, \lambda_{\mu_1}=\lambda_\mu - \lambda_{\mu_2}$
  $\lambda_n = \lambda - \lambda_m - \lambda_{\mu_1}$
// The final input and output loop bases
$\theta_n = \lambda_n$
$\lambda = \lambda_n\otimes\lambda_m\otimes\lambda_{\mu_1}$
$\theta = \theta_n\otimes\theta_m\otimes\theta_\mu$.

Figure 6.3: Algorithm for determining input and output loop bases for stride permutations.

where we use the convention that λ appearing on the right hand side refers to the original representation, which is equal to $\beta(1)\otimes\beta(3)\otimes\beta(2)\otimes\beta(4)$, and λ appearing on the left hand side refers to an update. So does θ. Further, we assume that $\lambda_\mu = \beta(2)\otimes\beta(4), \theta_\mu = \delta(2)\otimes\delta(4)$. It is easy to verify that $(\theta_\mu - \lambda_\mu)\otimes\lambda_\mu$ is a permutation of $(\lambda_\mu - \theta_\mu)\otimes\theta_\mu$. Therefore, they denote the same records. Thus, if the number of records denoted by $| (\theta_\mu - \lambda_\mu)\otimes\lambda_\mu |$ is less than the size of the main memory, then we can simply take $\lambda_m = \theta_\mu - \lambda_\mu$ and $\theta_m = \lambda_\mu - \theta_\mu$. However, the number of the records denoted by $| (\theta_\mu - \lambda_\mu)\otimes\lambda_\mu |$ may exceed the size of the main memory. In that case, we want

to construct memory-loads which can be obtained by reading the input data several times while writing the output data only once. In terms of tensor bases, as we discussed in Section 6.3.1, this reloading can be achieved by looping over part of the indices in λ_μ. In other words, we need to factor λ_μ as λ_{μ_2} and λ_{μ_1} such that the instantiation of the indices in λ_{μ_2} selects which sub-blocks should be kept for a loaded physical track and the instantiation of the indices in λ_{μ_1} denotes records inside each sub-block. Further, $|\ \lambda_{\mu_2}\ |$ is equal to the number of times we will reload each physical track. This reloading is achieved by taking $\lambda_m = \theta_\mu - \lambda_\mu$ and moving λ_{μ_2} before λ_m. In summary, the input and output loop bases in Formulas (6.5) and (6.6) are modified as follows:

- Factor $(\lambda_\mu - \theta_\mu)$ such that θ_m consists of the last factors of the factored tensor basis and the size of θ_m is equal to $\frac{M}{B_d D}$.
- For input loop basis, let λ_{μ_2}=$(\lambda_\mu - \theta_\mu) - \theta_m$, λ_{μ_1}=$\lambda_\mu - \lambda_{\mu_2}$.

Thus, the input and output loop bases can be written as,

$$\lambda = \lambda_n \otimes \lambda_m \otimes \lambda_{\mu_1}, \tag{6.7}$$

$$\theta = \theta_n \otimes \theta_m \otimes \theta_\mu. \tag{6.8}$$

where $\lambda_n = \lambda - \lambda_m - \lambda_{\mu_1}$ and $\theta_n = \theta - \theta_m - \theta_\mu$. We further verify the following facts.

First, $\lambda_m \otimes \lambda_{\mu_1}$ and $\theta_m \otimes \theta_\mu$ contain the same vector bases, although in a different order. The proof is presented as follows.

If we are not concerned the order of the vector bases occurring in a tensor basis, then we can regard a tensor basis as a set of vector bases. Hence, we use set notation in the following presentations. The proof consists of the following two steps.

(a) We first prove that for any vector basis v, if $v \in \theta_m \otimes \theta_\mu$, then $v \in \lambda_m \otimes \lambda_{\mu_1}$.

i. Assume that $v \in \theta_\mu$. If $v \notin \lambda_\mu$, then $v \in \lambda_m$, since $\lambda_m = \theta_\mu - \lambda_\mu$. If $v \in \lambda_\mu$, then $v \notin \lambda_\mu - \theta_\mu$. Since λ_{μ_2}=$(\lambda_\mu - \theta_\mu) - \theta_m$, $v \notin \lambda_{\mu_2}$. Therefore, $v \in \lambda_{\mu_1}$.

ii. Assume that $v \in \theta_m$. Then, $v \notin \lambda_{\mu_2}$ and $v \in \lambda_\mu - \theta_\mu$. From $v \in \lambda_\mu - \theta_\mu$, we have that $v \in \lambda_\mu$. Moreover, since $v \notin \lambda_{\mu_2}$, we have that $v \in \lambda_{\mu_1}$.

In either case (i) or (ii), we have that $v \in \lambda_m \otimes \lambda_{\mu_1}$.

(b) We now prove the other direction. For any vector basis v, if $v \in \lambda_m \otimes \lambda_{\mu_1}$, then $v \in \theta_m \otimes \theta_\mu$.

i. Assume that $v \in \lambda_m$. Since $\lambda_m = \theta_\mu - \lambda_\mu$, we have that $v \in \theta_\mu$.

ii. Assume that $v \in \lambda_{\mu_1}$. If $\theta_\mu \cap \lambda_\mu = \phi$, then $\lambda_\mu - \theta_\mu = \lambda_\mu$. Thus, $\lambda_{\mu_2} = \lambda_\mu - \theta_m$. Therefore, $\lambda_{\mu_1} \in \theta_m$. Thus, $v \in \theta_m$. Now we consider that $\theta_\mu \cap \lambda_\mu \neq \phi$. Notice that θ_m =last portions of $\lambda_\mu - \theta_\mu$, λ_{μ_2} = first portions of $\lambda_\mu - \theta_\mu$, and $\lambda_{\mu_1} = \lambda_\mu - \lambda_{\mu_2}$. We can visualize their relationships as in Fig. 6.4. From the figure, we can readily see that $v \in \theta_m$ or $v \in \theta_\mu$.

Therefore, in either case (i) or (ii), we have that $v \in \theta_m \otimes \theta_\mu$.

Combining Case (a) and Case (b) proves the claim.

Therefore, $\lambda_m \otimes \lambda_{\mu_1}$ and $\theta_m \otimes \theta_\mu$ denote the same records, although, in different order.

Second, from the previous results, we have that $\mid \lambda_m \otimes \lambda_{\mu_1} \mid = \mid \theta_m \otimes \theta_\mu \mid = M$. Therefore the records denoted by them can fit into a memory-load. Third, $\mid \lambda_m \mid > \mid \theta_m \mid (= \frac{M}{DB_d})$, which means that we have loaded more records

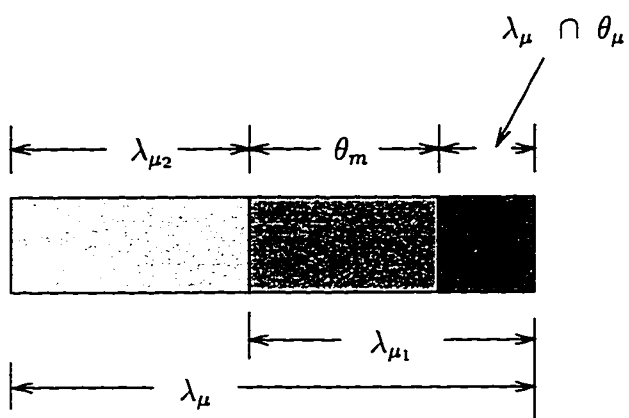


Figure 6.4: Relations among θ_m, θ_μ, λ_{μ_2}, λ_{μ_1}, and λ_μ.

than can fit into main memory and we need to discard some of the records. The details for determining which records to be discarded will be discussed in the next step. Fourth, λ_n and θ_n contain the same vector bases. We therefore can set θ_n=λ_n, which will only change the order of writing results onto physical tracks.

2. **Determine memory-load.** When $|\ (\theta_\mu - \lambda_\mu)\otimes\lambda_\mu\ | \leq M$, $\lambda_m = \theta_\mu - \lambda_\mu$ and $\theta_m = \lambda_\mu - \theta_\mu$. Therefore, the records denoted by $\lambda_m \otimes \lambda_\mu$ or $\theta_m \otimes \theta_\mu$ can be used to form a perfect memory-load. However when this condition is not satisfied, we need to use Formulas (6.7) and (6.8) as the input and output loop bases, respectively. Because $|\ \lambda_m \otimes \lambda_{\mu_1}\ | = |\ \theta_m \otimes \theta_\mu\ | = M$, the size of each memory-load can be set to be equal to the size of the main memory. However, as we mentioned before, we need to discard some records from each loaded track to form the memory-load. This can be done by linearizing λ_{μ_2}. Each instantiation of the indices in λ_{μ_2} will give a set of sub-blocks in a physical track which should be kept.

3. **Determine operations for a memory-load.** As we mentioned above, for each

memory load, the tensor vectors in the input and output loop bases which denote the records inside a memory-load are the same, but in a different order. In other words, one is a permutation of the other. Because the input and output loop bases are permutations of the input and output data distribution bases, we actually permute a memory-load of data each time. Therefore, each in-memory operation is nothing more than a permutation for a subset of data distribution bases denoted by $\lambda_m \otimes \lambda_{\mu_1}$ and $\theta_m \otimes \theta_\mu$. Note that when $\lambda_{\mu_2} = \phi$, $\lambda_{\mu_1} = \lambda_\mu$.

4. **I/O cost of synthesized programs.** It is easy to see that if $|(\theta_\mu - \lambda_\mu) \otimes \lambda_\mu| \leq M$, a one-pass program can be synthesized, i.e., the number of parallel I/Os is $\frac{2N}{B_d D}$. When the above condition does not hold, we keep $|\lambda_{\mu_1}|$ records for each loaded physical track and load the same physical track $|\lambda_{\mu_2}|$ times. Moreover, since $|\theta_m| = \frac{M}{DB_d}$, it can be easily determined that $|\lambda_{\mu_2}| = \frac{|\lambda_\mu - \theta_\mu| B_d D}{M}$. Because we write out each record only once, the number of parallel I/O operations is $(1 + \frac{|\lambda_\mu - \theta_\mu| B_d D}{M}) \frac{N}{B_d D}$. Combining these two cases yields the performance results presented in the theorem. Further, a program with this performance can be synthesized by using the procedure listed in Fig. 6.1.

□

We now use an example to illustrate the methods of determining augmented tensor bases and synthesizing parallel I/O programs for stride permutations. Assume that we have a stride permutation L_4^{36}, which can be interpreted as an 8×4 matrix transposition . The parameters of the model are defined as follows: $D = 2$, $B_d = 2$, $B_b = 2$, and $M = 8$. Then the input and output data distribution bases can be written as follows,

$$\beta = e_g^4 \otimes e_d^2 \otimes e_{b_b}^2 \otimes e_{b_d}^2, \tag{6.9}$$

$$\delta = e^4_{g'} \otimes e^2_{d'} \otimes e^2_{b'_b} \otimes e^2_{b'_d}. \tag{6.10}$$

Moreover, the output data distribution basis can also be obtained by applying the stride permutation L^{36}_4 to the input data distribution basis. In other words, it can be written as,

$$\delta = e^2_{b_b} \otimes e^2_{b_d} \otimes e^4_g \otimes e^2_d. \tag{6.11}$$

Then, following the procedure of the proof of Theorem 3.2, we can first determine the input and output loop bases as follows. We first factor e^4_g as $e^2_{g_2} \otimes e^2_{g_1}$. Then, by the algorithm presented in Fig. 6.3, we have,

$$\lambda_\mu = e^2_d \otimes e^2_{b_d}, \theta_\mu = e^2_{g_2} \otimes e^2_d, \tag{6.12}$$

$$\lambda_m = e^2_{g_2}, \theta_m = e^2_{b_d}, \tag{6.13}$$

$$\lambda_n = e^2_{g_1} \otimes e^2_{b_b}, \theta_n = e^2_{g_1} \otimes e^2_{b_b}. \tag{6.14}$$

Further, the records denoted by $\lambda_m \otimes \lambda_\mu$ or $\theta_m \otimes \theta_\mu$ will be used to form perfect memory-loads. The in-core computation can be determined by finding out the permutation which permutes $\lambda_m \otimes \lambda_\mu$ to $\theta_m \otimes \theta_\mu$. This can be easily determined as L^8_2. Since $| \ (\theta_\mu - \lambda_\mu) \otimes \lambda_\mu \ | \leq M$, a one-pass program, as shown in Fig. 6.5, can be synthesized by using the information determined above and the code generation algorithm presented in the previous subsection.

The procedure of computing L^{36}_4 using the synthesized program is illustrated in Fig. 6.6, and Fig. 6.7. Fig. 6.6 shows the input vector when explained as a matrix and its initial data distribution on two disks. It also shows the first two intermediate sub-transposition steps. Fig. 6.7 illustrates the successive two intermediate steps and the final outputs. Each of the intermediate sub-transposition steps reads a block of matrix, transposes the block in the internal memory and then writes the block onto disks. For clarity, we assume that the outputs are written on a different set of disks.

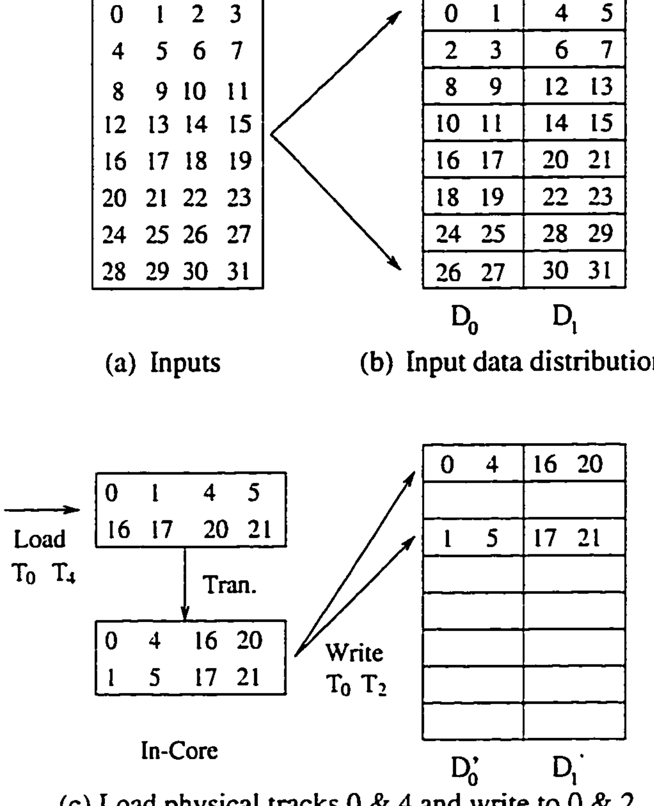
```
DO g1 = 0,1
  DO bb = 0,1
    DO g2 = 0,1
      // Parallel read from a track
      X(4g2 : 4g2 + 3) ← parallel_read(4g2 + 2g1 + bb)
    ENDDO
    // Perform operations for a memory load
    Code(X(1 : 16) ← L^8_2(X(1 : 16)))
    // Write the result back
    DO bd = 0,1
      // Parallel write to a track
      parallel_write(4bb + 2bd + g1) ← X(4bd : (4bd + 3))
    ENDDO
ENDDO  ENDDO
```

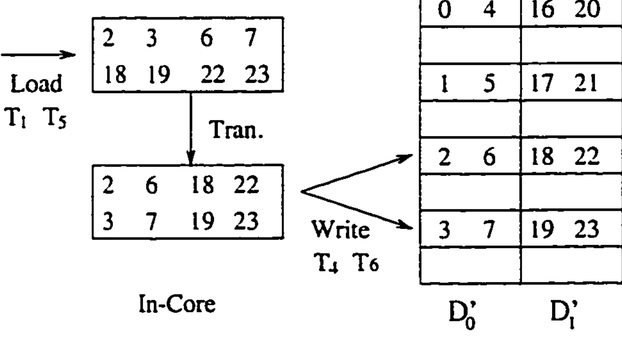
Figure 6.5: Parallel I/O Program for L^{36}_4

6.3.3 Synthesizing Programs for Tensor Products

In this subsection, we first present an algorithm to determine efficient loop bases for a tensor product under a given data distribution *cyclic*(B). Based on these loop bases and data distribution bases, we can determine memory-loads and operations to each memory-load. In other words, the augmented tensor basis can be obtained. Therefore, a program can be generated by using the procedure discussed in Section 6.3.1. We also show that the cost of the program synthesized can be obtained from the algorithm.

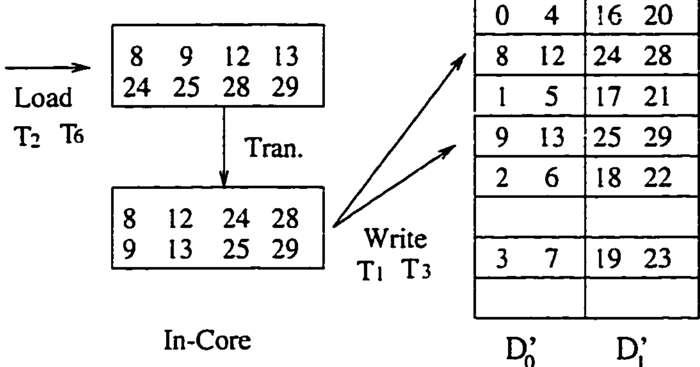
Since the computation of the tensor product $I_R \otimes A_V \otimes I_C$ does not change the order of the inputs (or it can be computed in-place), we will use the same input and output data distribution bases for the input and output data and also the same input and output loop bases for programs synthesized in this subsection. Therefore, we



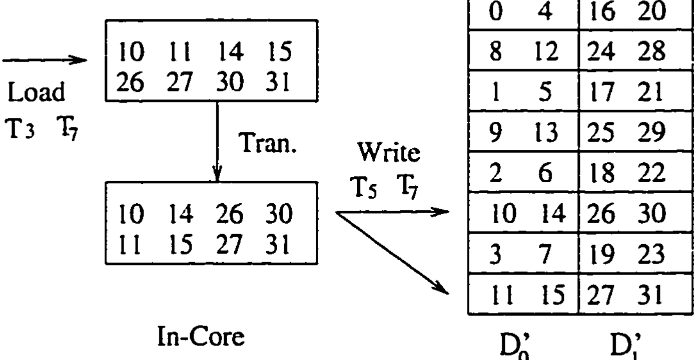


(d) Load physical tracks 1 & 5 and write to 4 & 6.

Figure 6.6: Example matrix transposition. (a) Inputs when viewed as an 8×4 two-dimensional array. (b) Input data distribution on two disks. (c) Load physical tracks T_0, T_4, in-core permutation, and write to physical tracks T_0, T_2. (d) Load physical tracks T_1, T_5, in-core permutation, and write to physical tracks T_4, T_6.



(a) Load physcial tracks 2 & 4 and write 1 & 3.



(b) Last in-core computation and output data distribution

0	4	8	12	16	20	24	28
1	5	9	13	17	21	25	29
2	6	10	14	18	22	26	30
1	7	11	15	19	23	27	31

(c) Output Data

Figure 6.7: Example matrix transposition. (a) Load physical tracks T_2, T_6, in-core permutation, and write to physical tracks T_1, T_3. (b) Load physical tracks T_3, T_7, in-core permutation, and write to physical tracks T_5, T_7. (c) Outputs.

will only consider input, input distribution and input loop bases. We summarize our results as a theorem and then present a constructive proof which constructs the augmented tensor basis. Before we present the theorem, we first introduce the concept of *desired records* and discuss several properties of the possible locations in which the desired records may reside on disks.

For the tensor product $I_R \otimes A_V \otimes I_C$, the major computational matrix A_V is applied to V input records and these V records have a stride C in the input vector. We call each of these V records for the first A_V computation a *desired record.* More specifically, V desired records can be denoted as $\{X[iC] | 0 \leq i \leq V-1\}$. Note that all of the other A_V computations will have a similar data access pattern. For example, the second A_V computation is applied to the V inputs beginning from the second record with the same stride C.

We now discuss several properties of the possible locations in which the desired records may reside on disks.

- The consecutive desired records will be first stored in a logical block, and then the successive desired records will be stored to other logical blocks on other disks. Thus, for example, when $C > B_d$ and $VC < B$, the number of physical tracks which holds the desired records is $\frac{V}{C/B_d}$ rather than $\frac{V}{C/(B_d D)}$.

- If the desired records are stored on several disks, then each of these disks will contain the same number of desired records and the desired records in each of these disks are stored in the same relative locations.

- If the desired records are stored on several logical tracks, then all of the logical tracks which contain the desired records will have the same number of desired records and the desired records in each logical track are stored in the same relative locations.

The correctness of these properties follows the definition of data distribution, the regular data access pattern of each computational matrix in the input tensor product, and the assumptions that all of the parameters in the machine model and the input tensor product are powers of two. For example, the correctness of the first property can be explained as follows. Since $VC < B$, all of the desired records are stored in the first logical block. The distance of the physical blocks which contain the desired records is $\frac{C}{B_d}$. Therefore, the number of physical tracks which hold the desired records is $\frac{V}{C/B_d}$. These properties will be used in the proof of the following theorem.

Theorem 3.3 *Let the input data be distributed according to cyclic*(B)*. Let* N_t *denote the number of physical tracks where the records for an* A_V *computation are stored. Then for the tensor product* $I_R \otimes A_V \otimes I_C$*, where* $RVC = N$ *and* $V \leq M$*, if* $N_t \leq \frac{M}{B_d D}$*, a program can be synthesized with* $\frac{2N}{B_d D}$ *parallel I/O operations; otherwise a program can be synthesized with* $\frac{3N}{M} N_t$ *parallel I/O operations.*

The above theorem can also be stated in terms of tensor bases as follows. Let λ be the input data distribution basis. Let $\lambda_\mu = \beta(2) \otimes \beta(4)$. Further assume that λ_{μ_1} denotes a subset of λ_μ and $\lambda_\mu - \lambda_{\mu_1} (= \lambda_{\mu_2})$ is moved into the memory basis. Then for the tensor product $I_R \otimes A_V \otimes I_C$, where $RVC = N$ and $V \leq M$, if $\lambda_{\mu_1} = \lambda_\mu$, a program can be synthesized with $\frac{2N}{B_d D}$ parallel I/O operations; otherwise a program can be synthesized with $| \lambda_{\mu_2} | \frac{3N}{B_d D}$. In the following proof of the theorem, we will show how to construct λ_{μ_1} and λ_{μ_2}. We will also prove that $| \lambda_{\mu_2} | = \frac{B_d D}{M} N_t$.

Proof:

1. **Determine input loop basis.** If the desired records for an A_V computation are stored in N_t physical tracks and $N_t \leq \frac{M}{B_d D}$, then we can simply load the N_t physical tracks each time and therefore a one-pass program can be generated.

However, when $N_t > \frac{M}{B_d D}$, we can not keep all of the records in N_t physical tracks in the main memory. We take the following simple approach: we keep as many records as possible which follow the desired records in the same physical track in the main memory. Then we reload these tracks to finish the computations for the other records. In terms of tensor bases, we need do nothing more than factor and permute the input data distribution basis to reflect this data access pattern.

More specifically, we begin with $\lambda_\mu = \lambda(2)\otimes\lambda(4)$, and $\lambda_n\otimes\lambda_m = \lambda - \lambda_\mu$, where λ has the same initial value as defined in Section 6.3.2. For a one-pass program, we factor and permute $\lambda_n\otimes\lambda_m$ to change the order of accessing physical tracks. However, for a multi-pass program, we need to factor and permute all of the λs, since we need to keep part of the records loaded in the main memory and discard other records. As we discussed before, the part of the records to be kept or discarded can be denoted by a subset of the vector bases in the physical-track-element basis. Chapter 5 discussed the examples of factoring and permuting data distribution bases to construct memory-loads. In general, in order to factor and permute a tensor basis to a desired form, we need to examine the relative values of the parameters in the targeted I/O model, the tensor product and the size B of the data distribution. We summarize the above ideas as an algorithm in Fig. 6.8, which is further explained as follows.

- **Initialization.** This step initializes the values of $\lambda_n\otimes\lambda_m$, λ_μ and several temporary variables. For example, R_b denotes the maximum number of the desired records for an A_V computation in a physical block. R_t is the number of the desired records in a physical track. R_d is the number of disks where the desired records for an A_V are stored. S is the distance of

// Initialization

$\lambda_n \otimes \lambda_m = e_g^G \otimes e_{b_b}^{B_b}$, $\lambda_\mu = e_d^D \otimes e_{b_d}^{B_d}$, $R_b = \lceil \frac{B_d}{C} \rceil$

$Compute(R_d)$

$R_t = R_b R_d$, $N_t = \lceil \frac{V}{R_t} \rceil$

$Compute(S)$

// One-pass program

// Determine the order to access physical tracks

If $S \leq B_b$ then

  $B_{b_1} = S$, Factor $e_{b_b}^{B_b}$ as $e_{b_{b_2}}^{B_{b_2}} \otimes e_{b_{b_1}}^{B_{b_1}}$

  $\lambda_n \otimes \lambda_m = e_g^G \otimes e_{b_{b_1}}^{B_{b_1}} \otimes e_{b_{b_2}}^{B_{b_2}}$

else

  $G_1 = \frac{S}{B_b}$, $G_2 = \frac{G}{G_1}$, Factor e_g^G as $e_{g_2}^{G_2} \otimes e_{g_1}^{G_1}$

  $\lambda_n \otimes \lambda_m = e_{g_1}^{G_1} \otimes e_{b_b}^{B_b} \otimes e_{g_2}^{G_2}$

If $C \leq B_d$ then $Z = C$ else $Z = \frac{D}{R_t} B_d$

// Multi-pass program

If $N_t > \frac{M}{B_d D}$ then

  // Further determine λ_{μ_2} and λ_{μ_1}

  $X = \min\{B_d, C, \frac{M}{R_t N_t}\}$

  If $X = \min\{B_d, C\}$ then $Y = \frac{M}{R_d B_d N_t}$ else $Y = 1$

  // Factor e_d^D and $e_{b_d}^{B_d}$

  $e_d^D = e_{d_3}^{R_d} \otimes e_{d_2}^{\frac{D}{R_d Y}} \otimes e_{d_1}^{Y}$, $e_{b_d}^{B_d} = e_{b_{d_3}}^{R_b} \otimes e_{b_{d_2}}^{\frac{B_d}{R_b X}} \otimes e_{b_{d_1}}^{X}$

  $\lambda_{\mu_2} = e_{d_2}^{\frac{D}{R_d Y}} \otimes e_{b_{d_2}}^{\frac{B_d}{R_b X}}$, $\lambda_{\mu_1} = \lambda_\mu - \lambda_{\mu_2}$

  // Compute the value of Z

  If $X = C$ or $X = \frac{M}{R_t N_t}$ then $Z = X$

  If $X = B_d$ then $Z = Y B_d$

Figure 6.8: Algorithm for determining input loop bases and the value of Z for a tensor product.

$Compute(R_d)$
If $C \leq B_d$ then $R_d = \min\{\frac{VC}{B}, D\}$
If $B_d < C \leq B$ then $R_d = \min\{\frac{VC}{B}, D\}$
If $B < C \leq BD$ then $R_d = \frac{BD}{C}$
If $C > BD$ then $R_d = 1$

Figure 6.9: Algorithm for computing R_d.

two consecutive physical tracks which contain the desired records. Since the stride of two desired records is C, R_b can be determined as $\lceil \frac{B_d}{C} \rceil$. The correctness of R_t and N_t can be similarly verified. *Compute* will invoke a procedure to compute the values such as R_d and S. Fig. 6.9 and Fig. 6.10 shows the details on how to determine those two values.

The correctness of the algorithm in Fig. 6.9 for computing R_d can be proved as follows. When $C \leq B_d$, the successive disks may contain the same number of the desired records if the desired records can not be stored in one logical block. The number of these successive disks is dependent on the value of V. Further, since there are $R_b B_b$ desired records per logical block, and $R_b B_b = \frac{B}{C}$ (since $R_b = \frac{B_d}{C}$ in this case), the number of disks which contain the desired records is equal to the smaller of $\frac{V}{B/C}$ and D. This results in the first case of the algorithm. Similarly, when $B_d < C \leq B$, the successive disks may contain the desired records. Since in this case, each logical block contains $\frac{B}{C}$ desired records, the number of disks which contain the desired records is again equal to the smaller of $\frac{V}{B/C}$ and D. For the third case, any two disks which contain two consecutive desired records have a stride $\frac{C}{B}$. Therefore, $R_d = \frac{D}{C/B}$. The last case is trivial. Similarly, we can prove the correctness of the algorithm

$Compute(S)$
If $C \leq B_d$ then $S = 1$
If $B_d < C \leq B$ then $S = \frac{B}{B_d C}$
If $B < C \leq BD$ then $S = 1$
If $C > BD$ then $S = \frac{C}{B_d D}$

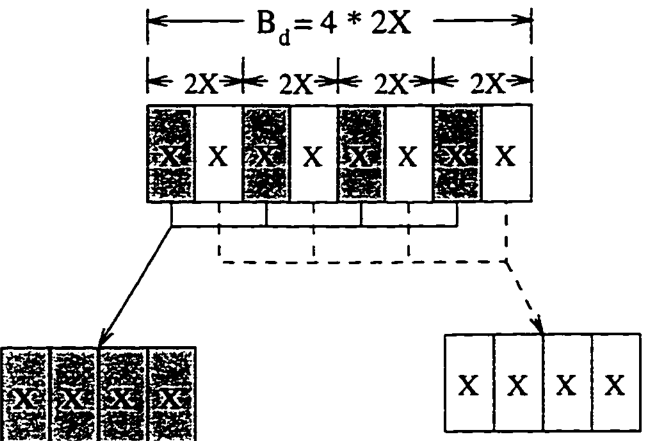
Figure 6.10: Algorithm for computing S.

in Fig. 6.10.

- **One-pass program.** This step determines how to access physical tracks. The idea is straightforward. It determines the decompositions and permutations for $\lambda_n \otimes \lambda_m$ based on the stride between two consecutive physical tracks which contain the desired records. The result from this step may also be needed for the next step to determine the final loop basis for synthesizing multi-pass programs.

- **Multi-pass program.** If the number of physical tracks which hold the records for an A_V computation is larger than the number of physical tracks which the main memory can hold, then a multi-pass program needs to be synthesized. More specifically, we need to determine which portions of the records in a physical track should be kept for each pass of computation. The basic idea of keeping records for the current memory-load can be described as follows.

  First, for each desired record, we want to take $X - 1$ successive records and keep these $X - 1$ records with the corresponding desired record as the current memory-load. One approach of determining X is to take X as large as possible. However, X needs to satisfy the following three conditions. First, X must be less than the gap between any two consecutive



Odd-numbered sub-blocks Even-numbered sub-blocks

Figure 6.11: Constructing portions of memory-loads from a physical block.

desired records in a physical block. Second, X must be less than the size of a physical block. Third, all of the desired records with their $X - 1$ successive records should be able to fit into the main memory, which means that $(XR_t)N_t \leq M$, or $XV \leq M$. These three conditions can be expressed as $X = \min\{C, B_d, \frac{M}{R_t N_t}\}$.

Fig. 6.11 shows an example of how to construct memory-loads by taking portions of the records from a physical block, where we assume that there are four desired records in a physical block, and $C = 2X$. The example can be interpreted as follows. The physical block is first broken into eight sub-blocks. Then we take the records in the odd-numbered sub-blocks to construct one memory-load and take the records in the even-numbered sub-blocks to construct another memory-load. In terms of tensor bases, we first decompose $e_{b_b}^{B_b}$ as $e_{b_{d_3}}^4 \otimes e_{b_{d_2}}^2 \otimes e_{b_{d_1}}^X$. Then, we permute the resulting tensor basis as $e_{b_{d_2}}^2 \otimes e_{b_{d_3}}^4 \otimes e_{b_{d_1}}^X$.

Second, we apply a similar idea for disks. For each disk which contains the desired record, we take $Y-1$ successive disks and we keep the records at the same relative locations with the original disk in each successive disk for the current memory-load. We want to take the largest possible value of Y given the condition that the number of the records kept must fit into the main memory. We consider the following two cases. First, $X = \min\{B_d, C\}$. In this case, either all of the records between any two desired records or all of the records in a physical block are chosen to be kept for the current memory-load. However, if all of the records between any two desired records are chosen, all of the records in a physical block will be covered. Thus, it is identical to the case that all of the records in a physical block are chosen to be kept. Further, R_d disks contain desired records. Therefore, $R_d B_d$ records are chosen from each physical track. In order to not overflow the main memory, we need that $R_d Y B_d N_t \leq M$. Second, $X = \frac{M}{R_t N_t}$. In this case, we do not choose all of the records between two desired records. However, since we have already chosen the largest possible value for X, the main memory has been filled up in this case. Therefore, we can not add any more records from successive disks from this approach. In other words, $Y = 1$.

An example, which is similar to the example shown in Fig. 6.11, can be constructed for disks. More specifically, if we view the records in a physical block as disks, X as Y, R_b as R_d, then we have an example for disks. Further, in terms of tensor bases, we can interpret this idea as follows. We first decompose e_d^D as $e_{d_3}^{R_d} \otimes e_{d_2}^{\frac{D}{R_d Y}} \otimes e_{d_1}^{Y}$. Then, we permute it as $e_{d_2}^{\frac{D}{R_d Y}} \otimes e_{d_3}^{R_d} \otimes e_{d_1}^{Y}$. The resulting tensor basis allows us to access odd-

numbered subset of disks first and then even-numbered subset of disks.

We now consider an example which contains both disks and records in physical blocks. More specifically, we consider the example in which data can be represented by combining factored e_d^D and $e_{b_b}^{B_b}$. Assume that we want to access the records first in the odd-numbered disk sub-blocks and then in the even-numbered disk sub-blocks. Further, for each physical block we want to access the records first in the odd-numbered sub-blocks and then in the even-numbered sub-blocks. To achieve this data access pattern, we move $e_{d_2}^{\frac{D}{R_d Y}}$ and $e_{b_{d_2}}^{\frac{B_d}{R_b X}}$ from their current locations in $e_d^D \otimes e_{b_b}^{B_b}$ to the beginning of $e_d^D \otimes e_{b_b}^{B_b}$. In the algorithm presented in Fig. 6.8, we have denoted $e_{d_2}^{\frac{D}{R_d Y}} \otimes e_{b_{d_2}}^{\frac{B_d}{R_b X}}$ as λ_{μ_2}. Therefore, to construct each memory-load, we can simply move λ_{μ_2} into $\lambda_n \otimes \lambda_m$ and put them anywhere in λ_n [1].

For the following analysis, we assume that we have found the subsets of λ_μ, namely λ_{μ_1} and λ_{μ_2}, by the above algorithm. λ_{μ_2} is moved into the memory basis and will generate loop nests for data access. The other portions of the algorithm, which are used for computing the value of Z, will be discussed in Step 3.

2. **Determine memory-load.** For a one-pass program, we can simply factor $\lambda - \lambda_\mu$ as $\lambda_n \otimes \lambda_m$ and take $\mid \lambda_m \mid = \frac{M}{B_d D}$. For a multiple-pass program, we factor $\lambda - \lambda_{\mu_1}$ to be $\lambda_n \otimes \lambda_m$ such that $\mid \lambda_m \mid = \frac{M}{|\lambda_{\mu_1}|}$ and all of the vector bases in λ_{μ_2} appear in λ_n. Moreover, for the multiple-pass program, as discussed in

[1] More specifically, the initial $\lambda_n \otimes \lambda_m$ should be modified to $\lambda'_n \otimes \lambda'_m$, where λ'_m contains the last factors of $\lambda_n \otimes \lambda_m$ and $\mid \lambda'_m \mid = \frac{M}{|\lambda_{\mu_1}|}$, and λ'_n contains $\lambda_n \otimes \lambda_m$-$\lambda'_m$ and λ_{μ_2}.

Section 6.3.2, we use λ_{μ_2} to determine which records should be kept for the current memory-load.

3. **Determine operations for a memory-load.** The original tensor product can be regarded as R parallel applications of A_V to the inputs with a stride C. When data are distributed among disks and loaded in units of physical tracks, the net effect is to possibly reduce the stride of the records which each A_V will access in main memory. The operations on a memory-load have the general form of $I_{\frac{M}{VZ}} \otimes A_V \otimes I_Z$. However, the value of Z will depend on the relative values of the parameters in the target machine model and the input tensor product. The algorithm presented in Fig. 6.8 can be used to determine this value. The correctness of the value of Z obtained from the algorithm can be proven as follows. For one-pass programs, when $C \leq B_d$, we do not change the stride for sub-computations. Therefore, $Z = C$. Otherwise, the stride will be reduced to be equal to the distance of two consecutive desired records in a physical track, which is equal to $\frac{D}{R_t} B_d$. For multi-pass programs, when $X = C$, we choose all of the records between any two desired records for the current memory-load, so the stride of in-core computation does not change. When $X = \frac{M}{R_t N_t}$, we reduce the stride of in-core computation from C to X. When $X = B_d$, the next desired record is not in the same physical block. Since we keep Y disks as a subset of disks, we reduce the stride from C to $Y B_d$.

4. **I/O cost of synthesized programs.** For a one-pass program which does not move any vector bases in λ_μ, the number of parallel I/Os is simply equal to $\frac{2N}{B_d D}$. In other words, the synthesized program is optimal in terms of the number of I/Os. For a multi-pass program, we need to read the inputs $\mid \lambda_{\mu_2} \mid$ times. Therefore the number of parallel I/O operations is $\mid \lambda_{\mu_2} \mid \frac{3N}{B_d D}$. From the al-

gorithm presented in Fig. 6.8, we can determine that $|\lambda_{\mu_2}| = \frac{DB_d}{M} N_t$. We therefore can attain the performance presented in the Theorem.

The constant 3 can be explained as follows. When we store a physical track, we need to read that physical track into main memory again, since portions of the records in that physical track have been discarded. By reloading this physical track, we can reassemble the physical track with the part of updated records and then write it out in parallel. Otherwise, part of the records to be written out in that physical track may not be correct. Further, "reassembling" the physical track needs to use the tensor basis θ_{μ_2} (notice that θ_{μ_2} is equal to λ_{μ_2}) to put the updated records into the correct locations on the physical track. This is similar to using λ_{μ_2} to take sub-blocks out from a loaded physical track for the current memory-load.

Now, a program with the performance discussed above can be synthesized by using the procedure listed in Fig. 6.1. However, to be accurate, when synthesizing a multi-pass program, we need to incorporate the idea of "reassembling" a physical track into the write-out part of the procedure listed in Fig. 6.1, which, as we discussed above, is nothing more than using the linearization of θ_{μ_2} to put sub-blocks in the current memory-load into the correct locations of the reloaded physical track.

□

Note that the value of N_t can be determined at the initialization step. Therefore, the performance of the synthesized program for a tensor product can be determined without generating the whole augmented tensor basis. This result is used in the first phase of transforming tensor product formulas, where we need the performance value for each tensor product to determine efficient transformations.

$B = B_d$
$Cost$ = number of I/Os when using $cyclic(B)$
while ($Cost \neq \frac{2N}{DB_d}$ and $B \leq \frac{N}{D}$) do
  $B = 2 \times B$
  C_{new} = number of I/Os when using $cyclic(B)$
  If $C_{new} \leq Cost$ then $Cost = C_{new}$ else break
output $distribution_size = B/2, number_of_I/Os = Cost$

Figure 6.12: Algorithm for computing the efficient size of data distributions.

6.3.4 Determining Efficient Data Distributions

In the previous subsections, we presented approaches for synthesizing efficient I/O programs for a given data distribution. We now present an algorithm to determine a data distribution which optimizes the performance of the synthesized program. The idea of the algorithm is as follows. We begin with the physical track distribution $cyclic(B_d)$, i.e., initially $B = B_d$. If a one-pass program can be synthesized under this data distribution, then B_d is the desired block size for the data distribution. Otherwise, we double the value of B. If the performance of the synthesized program under this distribution increases, we continue this procedure. Otherwise, the algorithm stops and the current block size is the desired size of data distributions. We formalize this idea in Fig. 6.12.

6.3.5 Transforming Tensor Product Formulas

In this subsection, we discuss techniques of program synthesis for tensor product formulas. There are several strategies for developing I/O-efficient programs, such as exploiting locality and exploiting parallelism in accessing the data. Similar ideas have been discussed in [49], where they use *factor grouping* to exploit locality and

data rearrangement to reduce the cost of I/O operations. We have also presented a greedy method which uses factor grouping to improve the performance of block recursive algorithms for Vitter and Shriver's striped two-level memory model with a fixed block size of data distribution [39].

Factor grouping combines contiguous tensor products in a tensor product formula and therefore reduces the number of passes to access secondary storage. Consider the core Cooley-Tukey FFT computation represented by Formula (4.15). For i=2 and i=3, we have the tensor products $I_{2^{n-2}} \otimes F_2 \otimes I_2$ and and $I_{2^{n-3}} \otimes F_2 \otimes I_{2^2}$, respectively. Assuming that each of these tensor products can be implemented optimally, the number of parallel I/O operations required to implement these two steps individually is $\frac{4N}{DB}$. However, they are contiguous tensor products in Formula (4.8). Hence, by using the properties of tensor products, such as Properties 1 and 2 listed in Chapter 4, they can be combined into one tensor product,

$$
\begin{aligned}
&(I_{2^{n-2}} \otimes F_2 \otimes I_2)(I_{2^{n-3}} \otimes F_2 \otimes I_{2^2}) \\
&\quad = (I_{2^{n-3}} \otimes I_2 \otimes F_2 \otimes I_2)(I_{2^{n-3}} \otimes F_2 \otimes I_2 \otimes I_2) \\
&\quad = (I_{2^{n-3}} \otimes (I_2 \otimes F_2) \otimes I_2)(I_{2^{n-3}} \otimes (F_2 \otimes I_2) \otimes I_2) \\
&\quad = I_{2^{n-3}} \otimes F_2 \otimes F_2 \otimes I_2,
\end{aligned}
$$

which may also be implementable optimally by using only $\frac{2N}{DB_d}$ parallel I/O operations.

Data rearrangement uses the properties of tensor products to change data access patterns. For example, the tensor product $I_R \otimes A_V \otimes I_C$ can be transformed into the equivalent form $(I_R \otimes L_V^{VC})$ $(I_{RC} \otimes A_V)$ $(I_R \otimes L_C^{VC})$. In the best case, the number of parallel I/Os required is $\frac{6N}{DB_d}$ after using this transformation, since at least three passes are needed for the transformed form. Because of the extra passes introduced by this transformation, it is not profitable to use it for our targeted machine model. Further,

the first and the last terms in the transformed formula may not be implementable optimally. Therefore, we have not incorporated this transformation into our current optimization procedures.

Minimizing I/O Cost by Dynamic Programming Since factor grouping (as shown above) and the size of the data distribution (as will be shown in the next section) have a large influence on the performance of synthesized programs, we take the following approach for determining an optimal manner in which a tensor product formula can be implemented. We use the algorithm for determining the optimal data distribution presented in Fig. 6.12 as a main routine. However, for each *cyclic*(B) data distribution, we use a dynamic programming algorithm to determine the optimal factor grouping. Hence, we also call this method a multi-step dynamic programming method.

Let $C[i,j]$ be the optimal cost (the minimum number of I/O passes required to access the out-of-core data) for computing $(j-i)$ tensor factors from the ith factor to the jth factor in a tensor product formula. Then $C[i,j]$ can be computed as follows:

$$C[i,j] = \begin{cases} C_0 & \text{if } i = j \\ \min_{i \leq k \leq j}\{C[i,k], C[k+1,j]\} & \text{if } i \leq j \end{cases}$$

In the above formula, C_0 denotes the cost for computing a tensor product. The method of determining the cost of a tensor product has been discussed in Section 6. The values of C_0 can be computed using the results in Theorem 3.3 and the algorithm presented in Fig. 6.9 to compute N_t. A special case of $k = j$ needs to be further explained. When $k = j$, we assume that $C[j+1,j] = 0$ and we use $C[i,k]$ to represent the cost of grouping all the tensor product factors from i to j together. Because the grouped tensor product is a simple tensor product, the value of $C[i,k]$ in this case can

also be determined by using the results in Theorem 3.3 and the algorithm presented in Fig. 6.9 to compute N_t. However, in this case, if $k-i > m$, or the size of grouped operations is larger than the size of the main memory, we do not want to group all of the $k-i$ factors together. We assign a large value such as ∞ to $C[k, j]$ to prevent it from being selected.

6.4 Performance Results of Synthesized Programs

6.4.1 Matrix Transposition

Given the flexibility of choosing different data distributions, we can synthesize programs with better performance than those obtained using fixed size data distributions for stride permutations. We present a set of experimental results for the number of I/O operations required by the *cyclic*(B_d) distribution and *cyclic*(B) distribution, where the size B of the distribution varies. These results are summarized in Table 6.1 and Table 6.2. From the tables, we can see that the number of passes is not a monotonically increasing or decreasing function. However, it normally decreases and then increases as B is increased. Therefore it is likely that the algorithm in Fig. 6.12 will find an efficient size of data distributions.

B	B_d	$2B_d$	$4B_d$	$8B_d$	$16B_d$	$32B_d$
$P=2^3, Q=2^8$	2	1	2	4	4	2
$P=2^4, Q=2^7$	4	2	1	2	2	1
$P=2^5, Q=2^6$	4	4	2	1	1	2
$P=2^6, Q=2^5$	4	4	2	1	1	2
$P=2^7, Q=2^4$	4	2	1	2	2	1

Table 6.1: Number of I/O passes for stride permutation L_Q^{PQ}. $D=4$, $B_d=4$, $M=64$, and $N=PQ=2048$.

B	B_d	2^3B_d	2^6B_d	2^9B_d	$2^{12}B_d$	$2^{15}B_d$
$P=2^{10}, Q=2^{40}$	2	1	1	8	16	16
$P=2^{15}, Q=2^{35}$	16	8	1	1	2	16
$P=2^{20}, Q=2^{30}$	16	16	16	4	1	1
$P=2^{25}, Q=2^{25}$	16	16	16	16	16	1
$P=2^{30}, Q=2^{20}$	16	16	16	4	1	1

Table 6.2: Number of I/O passes for stride permutation L_Q^{PQ}. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = PQ = 2^{50}$.

6.4.2 Tensor Products

The number of I/O passes required by the synthesized programs are summarized in Table 6.3, Table 6.4, and Table 6.5 by going through various cases of N_t. In those tables, M_t $(= \frac{M}{B_dD})$ is the maximum number of physical tracks in a memory-load. We can verify that the results presented here are more comprehensive than the results presented in [39]. In most cases, using the approach presented in Section 6.3.3, we can actually synthesize programs with better performance. For example, when $VC > M$, $M < VDB_d$ and $M > VB_d$, from [39], a program with $\frac{VB_dD}{M}$ passes will be synthesized. However, for these conditions, we have that $C > B_d$, and $VC > M$. If we further assume that $C < B_dD$, then from the results in Table 6.3 and Table 6.4, we can synthesize a program with $\frac{VC}{M}$ passes, which is less than $\frac{VB_dD}{M}$.

$C \geq B$			
$C \geq BD$		$C < BD$	
$V \leq M_t$	$V > M_t$	$\frac{VC}{BD} \leq M_t$	$\frac{VC}{BD} > M_t$
1	$\frac{VB_dD}{M}$	1	$\frac{VCB_d}{BM}$

Table 6.3: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

We now show that by using an appropriate $cyclic(B)$ data distribution, a better

$B_d \leq C < B$					
$VC \leq B$		$B < VC \leq BD$		$VC > BD$	
$V \leq M_t$	$V > M_t$	$\frac{B}{C} \leq M_t$	$\frac{B}{C} > M_t$	$\frac{V}{D} \leq M_t$	$\frac{V}{D} > M_t$
1	$\frac{VB_dD}{M}$	1	$\frac{BB_dD}{MC}$	1	$\frac{VB_d}{M}$

Table 6.4: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

$C < B_d$					
$VC \leq B$		$B < VC \leq BD$		$VC > BD$	
$\frac{B}{C} \leq M_t$	$\frac{B}{C} > M_t$	$\frac{B}{B_d} \leq M_t$	$\frac{B}{B_d} > M_t$	$\frac{VC}{B_dD} \leq M_t$	$\frac{VC}{B_dD} > M_t$
1	$\frac{VCD}{M}$	1	$\frac{BD}{BM}$	1	$\frac{VC}{M}$

Table 6.5: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$.

performance program can be synthesized for most of the cases. Several typical examples are shown in Table 6.6. We notice that when we increase B, we can reduce the number of passes of data access for most of the cases and the decrease in the number of passes can be as large as eight times. The values in the table also suggest that we can use the algorithm presented in Fig 6.12 to find an efficient size of data distributions for a given tensor product. We also notice that for some cases, such as $C \leq B_d$, we can not improve the performance. The reason is that the stride required by A_V is less than the size of the physical block, and we can not reduce it further by redistribution.

6.4.3 Tensor Product Formulas

We show the effectiveness of the multi-step dynamic programming method by comparing the programs synthesized by it with the programs synthesized by the greedy method and the dynamic programming method (applied to a data distribution of fixed size), respectively. The example we use is the core Cooley-Tukey FFT computation.

B	B_d	2^3B_d	2^6B_d	2^9B_d	$2^{12}B_d$	$2^{15}B_d$
$V=2^8, C=2^{15}$	1	1	1	1	1	1
$V=2^{10}, C=2^{15}$	2	2	2	1	1	1
$V=2^{14}, C=2^{15}$	32	32	32	16	8	4
$V=2^{16}, C=2^{15}$	128	128	128	64	32	16
$V=2^{16}, C=2^8$	4	4	4	4	4	4

Table 6.6: Number of I/O passes for the tensor product $I_R \otimes A_V \otimes I_C$ with various data distributions. $D = 16$, $B_d = 512$, $M = 2^{22}$, and $N = RVC$.

The results for several typical sizes of inputs are shown in Table 6.7. We find that using dynamic programming for a fixed size *cyclic*(B_d) distribution normally can not improve performance over the greedy method. However, by using the multi-step dynamic programming method, we can reduce the number of passes for the synthesized programs by at least 1 if N is very large. Because the input size is large, the performance gain by eliminating even one pass to access out-of-core data is significant.

N	2^{50}	2^{70}	2^{90}	2^{100}	2^{150}
Greedy ($B = B_d$)	5	7	9	10	16
D.P. ($B = B_d$)	5	7	9	10	16
M.D.P.	4 ($B = 2^{10}$)	6 ($B = 2^{12}$)	8 ($B = 2^{13}$)	9 ($B = 2^{15}$)	15 ($B = 2^{11}$)

Table 6.7: Number of I/O passes for the synthesized programs using Greedy, Dynamic programming (D.P), and Multiple-step dynamic programming(M.D.P) methods ($D = 16$, $B_d = 512$, and $M = 2^{22}$).

6.5 Conclusions

This chapter presented a framework for synthesizing out-of-core programs for block recursive algorithms using the algebraic properties of tensor products. We used Vitter and Shriver's striped two-level memory model as our target machine model. How-

ever, instead of using the simpler *physical track* distribution normally used by this model, we used various block-cyclic distributions supported by the HPF to organize data on disks. We showed that by using the algebraic properties of tensor products, we can decompose computations and arrange data access patterns to generate out-of-core programs automatically.

We demonstrated the importance of choosing the appropriate data distribution for efficient out-of-core implementations through a set of experiments. The experimental results also show that our simple algorithm for choosing an efficient data distribution is very effective. From the observations about the importance of data distributions and factor grouping for tensor products, we proposed a dynamic programming approach to determine an efficient data distribution and the factor grouping. This dynamic programming approach can reduce the number of I/O passes by at least one for an example FFT computation.

In the next chapter, we generalize the method presented in this chapter to the multi-processor multi-disk system, which is closer to practical distributed-memory machines with secondary storage. The major extensions include further identifying sub-computations for each processor for each memory-load and determining data distributions for each memory-load in the internal memories.

Chapter 7

Synthesizing Out-of-Core Programs for Multi-Processor Multi-Disk Systems

In this chapter, we enhance the framework presented in the previous chapter to synthesize efficient out-of-core programs for block recursive algorithms for distributed-memory machines modeled by the multi-processor multi-disk model.

In order to synthesize efficient parallel out-of-core programs for a distributed-memory machine with secondary storage, it is important to optimize all the factors which contribute to the overall execution time. These include I/O time, inter-processor communication time, and local computation time. Since out-of-core block recursive algorithms involve multiple passes over out-of-core data, the I/O cost can be considerable. Hence, the following approach is taken for optimizing performance. First, the I/O overhead is optimized. Second, the communication cost is optimized by using the program synthesis framework based on redistribution of in-core data for in-core distributed-memory computation. Finally, the computation cost is minimized by using optimized computation kernels for in-core computation. The optimization methods for the second and the third steps are discussed elsewhere [38].

In this chapter, we present the approach for synthesizing programs with opti-

mized I/O costs. Moreover, we will derive an approach which can also determine in-core data distributions so that we can use in-core methods to generate in-core efficient programs.

The major differences between the method presented in this chapter and the method discussed in the previous chapter are follows.

1. In addition to determining sub-computations for each memory-load, we need to further determine *local sub-computations* (defined later) for each processor for each memory-load.

2. We need to determine in-core data distribution in order to use in-core methods to synthesize communication-efficient programs.

3. Instead of using the dynamic programming approach, we use a greedy method to transform tensor product formulas.

4. Instead of using the physical track as a unit of data access, we use the logical track as a unit of data access. The efficient size of a logical block (or track) is obtained by using a performance prediction function.

5. Instead of using the number of passes as the performance measure, we use the number of passes and the I/O cost at each pass to measure the performance of synthesized programs.

The rest of the chapter is organized as follows. We first extend the results of Chapter 5 to describe not only out-of-core data distributions but also internal data distributions on multi-processor multi-disk systems. Then we describe how to synthesize SPMD programs for a tensor product. Further, we use a greedy algorithm to generate I/O efficient programs for tensor product formulas. Since the overall I/O

cost required to compute a block recursive algorithm is dependent upon the number of passes over out-of-core data and the I/O cost in each pass, which in turn is dependent on distribution of out-of-core data, we use a function of the size of the data distribution to predicate the I/O performance. The minimum of this function is used to determine efficient data distributions.

7.1 Machine Model and Performance Metrics

The multi-processor multi-disk model is pictured in Fig. 1.1(B), where we assume that a parallel machine consists of a set of processor nodes, each node consisting of a processor, a local memory and a local disk used as secondary storage. The distinguished features of the model are assumptions on data organization.

1. Data files are declustered among disks in blocks of size B. In other words, data are first organized as blocks. These blocks are then stored on disks in a round-robin fashion. The set of blocks on a disk constitutes a *sub-file*.
2. Each processor node has its own sub-file, which is stored on its local disk. Therefore, the number of sub-files, or the number of disks, is equal to the number of processor nodes.

Fig. 1.1(B) shows a system with four processor nodes and data is striped among four disks in size B. In general, we use the following five parameters to quantify the model: N (the size of the input), M (the total size of the internal memory in all nodes), B (the size of each block), D (the number of disks), and P (the number of processor nodes). We assume that $M < N$, $1 \leq B \leq \frac{M}{2}$, $1 \leq D \leq \frac{M}{B}$, and $P = D$.

The cost of an application on this model consists of three parts: computational cost, communication cost and I/O cost. Since the I/O cost is the dominating factor

for large out-of-core applications, we will mainly optimize the I/O overhead. The I/O cost of transferring a block of B records is normally estimated as $\tau_s + B\tau_e$, where τ_s is the setup time of an I/O operation and τ_e is the data transfer rate between a disk and a processor. For an out-of-core computation, if it needs t I/O operations, then the I/O cost for the computation would be equal to $t(\tau_s + B\tau_e)$, whose value depends on the size of B.

We introduce the following definition for the number of I/O operations. If a set of blocks on different disks is accessed simultaneously, then it counts as one (parallel) I/O operation. If the number of I/Os for a program is equal to $\frac{2N}{DB}$, then the program is called *disk-optimal*. In our approach, we always synthesize programs with balanced I/O requests, so that each I/O operation can load D blocks. Moreover, the synthesized program will load D blocks in a track. Therefore, the I/O overhead of the synthesized program can be estimated as $\frac{2kN}{DB}(\tau_s + B\tau_e)$, where k is the number of passes to access out-of-core data. It is obvious that if B is fixed, then the fewer the number of passes, the smaller the I/O overhead.

7.2 Semantics of Out-of-Core Data Distributions

Since we do not use the physical track as a unit of data access, we do not need to include the factor of $e_{b_d}^{B_d}$ in our definition of data distribution bases. However, many of the similar concepts such as using subsets of a data distribution basis to denote different data units can be similarly defined.

Definition 2.8 *If a vector of length N, where $N = GBD$ and G is an integer, is distributed according to the cyclic(B) distribution, then its data distribution basis is defined as:*

$$\mathcal{I}_D = e_g^G \otimes e_d^D \otimes e_b^B \tag{7.1}$$

A selected subset of the tensor basis in Formula (7.1) can be used to obtain the indexing function for a data unit, such as a track, a block, or a sub-file. For example, $e_g^G \otimes e_d^D$ is called a *block basis* since its linearization results in a function which can be used to denote the (global) index of a block. Similarly, e_d^D denotes an indexing function for sub-files.

Because of the presence of multiple processors, the numbering for blocks can either refer to the whole system (global block number) or to a processor (local block number); and the numbering for blocks can either refer to the block's index within a processor's memory (in-core block number) or to its index within a disk (out-of-core block number). The index number of a block in these various contexts can all be determined by taking a selected subset of a factored data distribution basis.

However, since we assume a distributed-memory model, we mainly use the local view. For example, we use $OCLB$ (*Out-of-Core Local Block Basis*) and $ICLB$ (*In-Core Local Block Basis*) to refer to out-of-core and in-core blocks of data corresponding to a local processor, respectively. If data is distributed by a *cyclic*(B) distribution among disks, then it is easy to determine that $OCLB = e_g^G$. Moreover, we assume that the $ICLB$ is a subset of the factored $OCLB$. In other words, we assume that blocks on disks are separated into disjoint sets of blocks which can fit into each processor's main memory. We call each of these sets a *local memory-load*. Further, P local memory-loads from P different disks constitute a *global memory-load*, or a *memory-load*. Note that we have assumed that all of the records in each loaded block will be used by the current computation. This is ensured in the approach taken in this chapter. The general case, where only the part of the records in a loaded block is required by the current computation, has been discussed in the previous chapter.

The following properties for a memory-load determine in-core data distribution for each memory-load.

Lemma 2.3 *Each memory-load of in-core data is distributed by a cyclic(B) distribution.*

Proof: The records of in-core data in a memory-load can be described by the tensor basis: $ICLB \otimes e_d^D \otimes e_b^B$, where e_d^D is the disk basis. Since in our model, each processor has a local disk, e_d^D can be regarded as a processor basis. Therefore, a memory-load of in-core data is distributed in cyclic(B). □

The final form of the $ICLB$ may be dependent on a specific computation. In the next section, we will describe how to factor data distribution bases and therefore determine the $OCLB$ and the $ICLB$. Moreover, we show how to write a disk-optimal program by using the $OCLB$, the $ICLB$ and other information.

7.3 Disk-Optimal Programs

As described in Fig. 1.3 in Chapter 1, our method of program synthesis consists of three steps. In this section, we discuss the second and the third steps: determining augmented tensor bases and code generation. Specifically, we will discuss the conditions under which a disk-optimal program can be synthesized for a tensor product. We will also present these disk-optimal programs. In the next section, we will use these conditions to develop a greedy algorithm for determining an efficient implementation for a tensor product formula.

7.3.1 Parallel Out-of-Core SPMD Code Generation

We begin with the discussion on how to write an I/O-efficient data-parallel (SPMD) program from $OCLB$, $ICLB$, in-core data distribution and sub-computations, which

constitute an *augmented tensor basis* for multi-processor multi-disk systems. Because we assume that the input and the output data distributions are the same, we do not distinguish $OCLB$, $ICLB$ as input or output.

By the definitions of the $OCLB$ and the $ICLB$, we can use them to access out-of-core data and construct a memory-load. In addition, in order to write a data-parallel program, we need to know the sub-computations on each memory-load. For example, Fig. 7.1 shows a procedure which can be used to write a parallel I/O code from $OCLB$, $ICLB$, in-core data distribution among processors for each memory-load, and operations applied to in-core data. The correctness of Fig. 7.1 closely follows the definition of the augmented tensor basis. For example, the first statement specifies different memory-loads, and the second statement uses the indices in the OCLB to load a memory-load.

The sub-computation for each memory-load will be computed by all of the processors. To perform this sub-computation, we can use in-core methods developed elsewhere or we can determine *local sub-computations* directly for some cases. A local sub-computation is a decomposed sub-computation which is computed by each processor node for a memory-load. From the viewpoint of each node processor, a local sub-computation is applied to each loaded local memory-load.

7.3.2 Disk-Optimal Programs for Tensor Products

Note that the program generated from Fig. 7.1 only accesses out-of-core data once. Therefore, it is disk-optimal. In general, we have the following results which specify the requirements for generating disk-optimal programs.

Theorem 3.4 *For the tensor product* $I_R \otimes A_V \otimes I_C$ *with* A_V *being a* $V \times V$ *square matrix and out-of-core data being distributed according to cyclic*(B) *among disks,*

1. Generate loops for indices in $OCLB$ but not in $ICLB$
2. Generate loops for indices in $ICLB$
3. Read a block using $OCLB$ as block index
4. End the loops from line 2
5. Synchronization
6. Synthesize in-core communication-efficient program
7. Write out a memory-load
8. End loops from line 1

Figure 7.1: Procedure of SPMD code generation for a tensor product.

if either $VC \leq M$ or $VDB \leq M < VC$, then a disk-optimal program can be generated.

Proof: We present synthesized programs by first determining $OCLB$, $ICLB$, in-core data distribution, and (local) sub-computation for each of these two cases. Note that for the second case, the in-core computation synthesized below is computed locally by each processor, no communication is required.

1. $VC \leq M$. Since data required by each $A_V \otimes I_C$ are stored on contiguous tracks and can fit into main memory, we can construct a memory-load by simply reading contiguous tracks. In terms of tensor bases, we can factor the data distribution basis as $e_{g_2}^{\frac{N}{M}} \otimes e_{g_1}^{\frac{M}{DB}} \otimes e_d^D \otimes e_b^B$. Therefore, $OCLB = e_{g_2}^{\frac{N}{M}} \otimes e_{g_1}^{\frac{M}{DB}}$. Further, we can use the records denoted by $e_{g_1}^{\frac{M}{DB}} \otimes e_d^D \otimes e_b^B$ as a memory-load. Thus, $ICLB = e_{g_1}^{\frac{M}{DB}}$. By Lemma 2.3, each memory-load is distributed in the *cyclic*(B) manner among processors. The operations for each memory-load are represented by $I_{\frac{M}{VC}} \otimes A_V \otimes I_C$. The pseudocode node program with $\frac{2N}{DB}$ I/O operations, as shown in Fig. 7.2, can now be synthesized by using the code generation algorithm presented in Fig.7.1. In Fig. 7.2, M_1 consists of blocks in a

```
Program-for-node-i
DO g2 = 0, N/M - 1
   DO g1 = 0, M/DB - 1
     // each processor reads a block
     M1(g1 B : (g1 + 1)B - 1) <- read(B_{g2 M/B + g1})
   ENDDO
   Barrier-synchronization
   // Perform the operations for a memory-load
   In-core (global) program for I_{M/UC} ⊗ A_V ⊗ I_C
   // Write the result back
   DO g1 = 0, M/DB - 1
     // write to the disk
     write(B_{g2 M/B + g1}) <- M1(g1 B : (g1 + 1)B - 1)
ENDDO ENDDO
```

Figure 7.2: Node program for the tensor product with $VC \leq M$.

processor's local memory and $\mathcal{B}_i$ denotes the i-th block on a processor's local disk. Since we know the in-core data distribution and the operations for each memory-load (represented by a tensor product), we therefore can use the method presented in [37] to determine local sub-computations and synthesize communication-efficient programs for each memory-load.

2. $VC > M$ and $M \geq VDB$. In this case, the records in the tracks which hold the out-of-core data corresponding to each $A_V \otimes I_C$ can not fit into main memory. Therefore, we can not read contiguous tracks to form a memory-load. Instead, we access contiguous $\frac{M}{VDB}$ tracks in a stride of $\frac{N}{RM}$. Fig. 7.3 shows an example access pattern. In terms of tensor bases, we factor the memory basis as $e_{g_4}^R \otimes e_{g_3}^V \otimes e_{g_2}^{\frac{N}{RM}} \otimes e_{g_1}^{\frac{M}{VDB}} \otimes e_d^D \otimes e_b^B$. Thus $OCLB = e_{g_4}^R \otimes e_{g_3}^V \otimes e_{g_2}^{\frac{N}{RM}} \otimes e_{g_1}^{\frac{M}{VDB}}$.

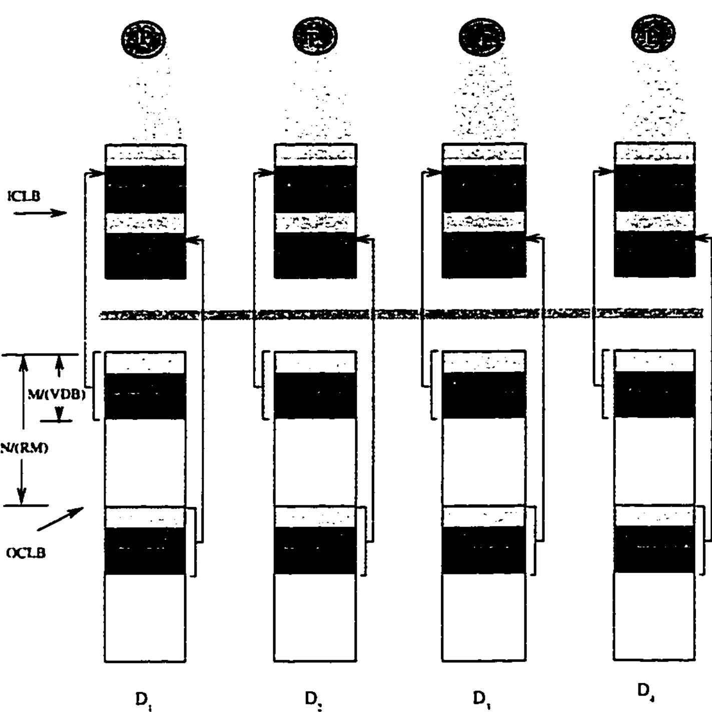


Figure 7.3: Constructing a memory-load from out-of-core data on disks.

We use the records denoted by $e_{g_3}^{V} \otimes e_{g_1}^{\frac{M}{VDB}} \otimes e_{d}^{D} \otimes e_{b}^{B}$ as a memory-load. Thus, $ICLB = e_{g_3}^{V} \otimes e_{g_1}^{\frac{M}{VDB}}$. By Lemma 2.3, each memory-load is distributed in the $cyclic(B)$ manner among processors. The operations for each memory-load can be obtained by the following observations. In the original formula, each A_V will be applied to the records of the input vector which are C records apart. Since we read in the vector in a striped fashion, we shorten the distance of the records which A_V will be applied. The new distance of the records applied by each A_V is $\frac{M}{V}$. Further, since $\frac{M}{V} \geq P$, the adjacent records required by an A_V are stored in the same node. However, the distance is reduced to $\frac{M}{VD}$ if we take a local view. We therefore can determine that the local sub-computation is $A_V \otimes I_{\frac{M}{VD}}$. The pseudocode node program with $\frac{2N}{DB}$ I/O operations can now

Program-for-node-i

DO $g_4 = 0, R-1$
  DO $g_2 = 0, \frac{N}{RM} - 1$
    DO $g_3 = 0, V-1$
      DO $g_1 = 0, \frac{M}{VDB} - 1$
        // Each processor reads a block
        $M_1((g_3\frac{M}{VB} + g_1)B : (g_3\frac{M}{VB} + g_1 + 1)B - 1)$
          $\leftarrow read(\mathcal{B}_{g_4\frac{N}{RDB}+g_3\frac{N}{RVDB}+g_2\frac{M}{VDB}+g_1})$
    ENDDO ENDDO
    Barrier-synchronization
    // Perform the operation for a memory-load
    In-core (local) program for $A_V \otimes I_{\frac{M}{P}}$
    // Write the result back
    DO $g_3 = 0, U-1$
      DO $g_1 = 0, \frac{M}{VDB} - 1$
        $write(\mathcal{B}_{g_4\frac{N}{RDB}+g_3\frac{N}{RVDB}+g_2\frac{M}{VDB}+g_1})$
          $\leftarrow M_1((g_3\frac{M}{VB} + g_1)B : (g_3\frac{M}{VB} + g_1 + 1)B - 1)$
ENDDO ENDDO ENDDO ENDDO

Figure 7.4: Node program for the tensor product with $VC \geq M \geq VDB$.

be synthesized as shown in Fig. 7.4.

□

7.4 Transforming Tensor Product Formulas

Since our first goal is to minimize I/O overhead, we can regard the internal processors and the corresponding network as a single processor with a main memory which is equal to the sum of every processor's local memory. Therefore, we can use a similar analysis to that discussed in the previous chapter to minimize I/O overhead. The main idea of the method is to reduce the number of passes over the out-of-core data by

grouping contiguous tensor products in a tensor product formula. However, instead of using a dynamic programming approach, we introduce a simpler greedy algorithm as follows.

Since grouping contiguous tensor products will decrease the number of passes to access secondary storage, we would like to group as many contiguous tensor products as possible. However, we also want to be able to generate I/O-efficient programs for each grouped tensor product. The detailed algorithm is described as follows. Beginning from the last factor, we group tensor products together using the properties of tensor products described in Chapter 4 only if the conditions in Theorem 3.4 are satisfied. We continue this procedure for the rest of the tensor product formula until all of the tensor products have been considered.

For example, consider Formula (4.8) and assume that $I_{r_j} = I_{V^{n-j}}$, and $I_{c_j} = I_{V^{j-1}}$. By using the greedy algorithm described above, we begin the procedure of factor grouping from the last factor and group from there greedily. If the number of factors to be grouped is k, then the constraint is $V^kC \leq M$ or $V^kC > M \geq V^kDB$. Since each factor in the resulting formula has the form $I_R \otimes A_{V^k} \otimes I_C$, the conditions of $V^kC \leq M$ or $V^kC > M \geq V^kDB$ can also be expressed as: the size of the inputs required by the computation corresponding to A_{V^k} in each stage is not larger than V^k and k satisfies either $V^kC \leq M$ or $V^kC > M \geq V^kDB$.

7.4.1 Determining Efficient Data Distributions

The running time of a synthesized program will be dependent on data distributions, which are determined by choosing the size B of data distributions. We now discuss how to choose B for Formula (4.8). If we assume that the resulting tensor product formula after using the greedy algorithm presented in the previous section contains

k factors, then the I/O overhead of the synthesized program can be expressed as,

$$\frac{2N}{DB}(\tau_s + B\tau_e)k, \tag{7.2}$$

where the first factor is the number of I/O operations in each pass, the second factor is the overhead of each I/O operation, and the third factor is the number of passes. Note that the number of passes k is also a function of block size B. Thus, we can further regard Formula (7.2) as a function of b as follows,

$$f(b) = \frac{2N}{D2^b}(\tau_s + 2^b\tau_e)k. \tag{7.3}$$

Therefore, the question of finding an efficient distribution has been converted to the problem of finding the minimum of the function $f(b)$. It is obvious that the solution will be dependent on the values of τ_s and τ_e, which are themselves dependent on the architectural properties of the target machine. We will discuss how to determine $\tau_s + 2^b\tau_e$ and therefore $f(b)$ for the CM-5 machine in the next chapter.

7.5 An Example: The Cooley-Tukey FFT

We now illustrate the procedure of program synthesis discussed so far by synthesizing I/O efficient programs for a 2^n-point Cooley-Tukey FFT. For simplicity, we assume that twiddle factors are evaluated on-line so that scaling by twiddle factors requires no I/O operations. We therefore do not include them in the following analysis. We also ignore the initial bit-reversal operation.

Let $N = 2^n$, $D = 2^d$, $B = 2^b$, and $M = 2^m$. We apply the greedy algorithm to the Cooley-Tukey FFT algorithm represented by Formula (4.8). The first m factors can be grouped together since the grouped tensor product satisfies the condition of the first case of Theorem 3.4. Successive $m-d-b$ factors are grouped and each grouped

tensor product satisfies the conditions of the second case (The last group may have fewer than $m - d - b$ factors). We now present a general form of the grouped FFT computation for an input size of 2^n.

Assume that k_1 and k_2 are integers and satisfy the following conditions,

$$k_1 = \lfloor \frac{n-m}{m-d-b} \rfloor, \quad \text{and } k_2 = (n-m) \bmod (m-d-b). \tag{7.4}$$

Then we have the following transformations for F_{2^n}.

$$\begin{aligned} F_{2^n} &= \prod_{i=1}^{n} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}) \\ &= \prod_{l=(k_1+1)m-k_1(b+d)+1}^{k_2+(k_1+)m-k_1(b+d)} (I_{2^{n-l}} \otimes F_2 \otimes I_{2^{l-1}}) \\ &\quad \prod_{j=1}^{k_1} (\prod_{i=jm-(j-1)(d+b)+1}^{(j+1)m-j(b+d)} (I_{2^{n-j}} \otimes F_2 \otimes I_{2^{j-1}})) \prod_{i=1}^{m} (I_{2^{n-i}} \otimes F_2 \otimes I_{2^{i-1}}) \\ &= ((\otimes_{i=1}^{k_2} F_2) \otimes I_{2^{(k_1+1)m-k_1(b+d)}}) \\ &\quad \prod_{j=1}^{k_1} (I_{2^{n-(j+1)m+j(b+d)}} \otimes (\otimes_{i=1}^{m-(b+d)} F_2) \otimes I_{2^{jm-(j-1)(b+d)}}) \\ &\quad (I_{2^{n-m}} \otimes (\otimes_{i=1}^{m} F_2)), \end{aligned}$$

where, remember $(\otimes_{i=1}^{3} F_2) = F_2 \otimes F_2 \otimes F_2$. For each factor in the above formula, we can use the routines presented in Section 7.3 to generate I/O-efficient programs. The first grouped tensor product will produce the program as presented in Fig. 7.2. The remaining $k_2 + 1$ grouped tensor products will produce the program as presented in Fig. 7.4. A higher level description of the generated program is shown in Fig. 7.5. Since each of these tensor products can be implemented in $\frac{2N}{DB}$ I/O operations, the total number of I/O operations is $\frac{2(k_1+2)N}{DB}$.

By substituting for k in Formula (7.3) by the number of passes of this algorithm, we have the following formula,

$$f(b) = \frac{2N}{D2^b}(\tau_s + 2^b \tau_e)(\lfloor \frac{n-m}{m-d-b} \rfloor + 2), \tag{7.5}$$

Out-of-core Cooley-Tukey FFT

1. First stage (corresponding to the last grouped factor)
   i. Read a memory-load data
   ii. Change in-core distribution as block distribution
   iii Perform in-core computation for the memory-load
   iv. Change in-core distribution as *cyclic*(B)
   v. Write the result back
   vi. Goto (i) unless all the inputs have been accessed
2. Successive stages (for successively grouped factors)
   DO $j = 1, k_1 + 2$
      i. Read a memory-load
      ii. Perform local computation on each node
      iii Write the result back
      iv. Goto (i) unless all the inputs have been accessed
   ENDDO

Figure 7.5: Synthesized out-of-core Cooley-Tukey FFT program.

which can be used to determine an efficient data distribution by finding b which minimizes the function $f(b)$.

7.6 Conclusions

In this chapter, we have introduced an algebraic approach for synthesizing efficient out-of-core block recursive algorithms for multi-processor multi-disk systems. Similar with the method of program synthesis discussed in Chapter 6, the algorithms are described by tensor product formulas. The out-of-core data is striped among disks and organized as a two-dimensional structure. This data organization is described by tensor bases. Efficient programs are then synthesized by using the algebraic properties of tensor products. We further investigated the influence of using different data

distributions. The optimal data distribution is obtained from finding the minimum of a performance prediction function.

Data distribution on multi-disk systems matches the two-dimensional file structure supported by current parallel file systems, such as the PIOUS system [61] on network connected work-stations and the PIOFS file system on the IBM SP-2 [20]. Hence, the framework of program synthesis presented in this paper can be used to build a program development tool for implementing out-of-core applications on modern distributed hierarchical memory machines. In the next chapter, we will use the methodology developed in this chapter to synthesize parallel I/O programs for the CM-5 machine.

Chapter 8

Synthesizing Out-of-Core FFT Programs for the CM-5 Machine

In this chapter, we discuss how to use the methodology developed in Chapter 7 to synthesize efficient parallel I/O programs for the CM-5 machine. We use the Cooley-Tukey FFT algorithm as a runing example. We discuss the performance issues of the synthesized FFT programs. Our implementation has a comparable performance with the well-known parallel out-of-core FFT implementation presented in [52]. We begin with the discussion of the architecture and parallel file systems of the CM-5 machine.

8.1 Overview of the CM-5 Machine

The Thinking Machines CM-5 is a distributed-memory machine [81]. The CM-5 machine contains two types of nodes: I/O nodes and computational nodes. Both of them are interconnected by a fat-tree data network and a binary tree control network. Secondary storage is attached to the I/O nodes but not directly to the computational nodes. The architecture of the CM-5 machine is shown in Fig. 8.1, where we assume that there are four computational nodes and one I/O node. Moreover, the I/O node

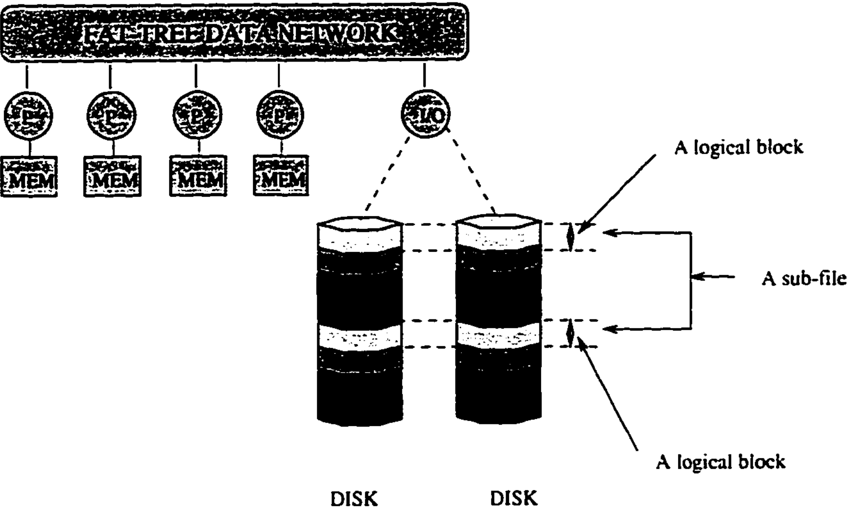


Figure 8.1: An example architecture of the CM-5 machine.

has two disks attached to it.

The CM-5 machine supports several different file systems. We are interested in the Scalable File System (SFS) [58] for the following reasons. First, the SFS supports several parallel I/O interfaces and can provide high bandwidth up to 95 MB/s [53]. Second, the SFS is based on the scalable disk array (SDA), which is a collection of high speed commodity disks connected to the CM-5 I/O node. A SFS file is always available in a serial order, however, it is physically striped across disks in unit of sixteen bytes.

Our program accesses the SFS file through the CMMD message passing library. We choose hardware-independent file formats rather than hardware-dependent file formats for the reason of portability. The parallel interface to the hardware-independent files provided by the CMMD library is implemented by new *I/O modes*. There are four I/O modes defined by the CMMD.

- Local independent. Each node maintains its own local file descriptor and file

pointer. Each processor can read (or write) different files independently.

- Global independent mode. All nodes can read (or write) a single file independently. Each node has its local file pointer, can move its pointer about at will, and does not need to coordinate with other nodes.

- Global synchronous broadcast. All processors read the same data from a file simultaneously. Logically, it is equivalent to one processor reading the data and then broadcasting to all of the other processors. For the write operation, only one processor will write successfully.

- Global synchronous sequential. All nodes read (or write) the data from (or to) a file simultaneously, however each node read (or write) sequential portions of the file. The data read from the file are assigned to each node sequentially, with node zero receiving the first sequential portion, where the size of the portion can be specified by each node. For file write, data from each node is written in sequential order, beginning with node zero.

We use the following two modes: Global independent mode and Global synchronous sequential. The reason we use the Global synchronous sequential is that in our implementation, every node normally needs to access different portions of a file in one I/O operation. *Since the global synchronous operation invokes underlying parallel I/O routines, we therefore refer to it as a parallel I/O operation.* We also implemented our code using global independent file mode. *Since the global independent file access is equivalent to the serial file access. We therefore refer to it as a serial I/O operation.*

8.2 Synthesizing FFT Programs for the CM-5

As we mentioned in the previous section, the file on the SDA is striped across disks in unit of sixteen bytes. It seems that this striped file structure matches the data organization used in the machine model in Fig. 1.2. However, the fundamental difference is that the CM-5 does not allow users to access data on each disk independently. In other words, the file on the CM-5 is still one-dimensional even though it is striped across a set of disks. In order to support our machine model, a logical two-dimensional file is employed. The idea is that we first organize the one-dimensional file into blocks. Then we assign each ith block to a logical disk ($i \bmod D$), where D is the number of logical disks, which is also equal to the number of node processors. Data on each logical disk can be accessed as a sub-file by a specific node. Therefore, we have converted a one-dimensional file into a two-dimensional structure and each node has a logical file (or disk) associated with it.

Fig. 8.1 shows an example organization of logical disks. The inputs are first striped on two physical disks by the SFS. A logical block constitutes the contiguous records striped on two disks with the same shadow. Two of the same shadowed logical blocks consist of a sub-file. The first four logical blocks consist of a logical track. There are two logical tracks in the figure.

The correct access to this logical file can be implemented by appropriately moving file pointers by each node processor. When using the Global synchronous sequential file mode, the correct file pointer for each node can be simply determined by the indices in the linearized $OCLB$.

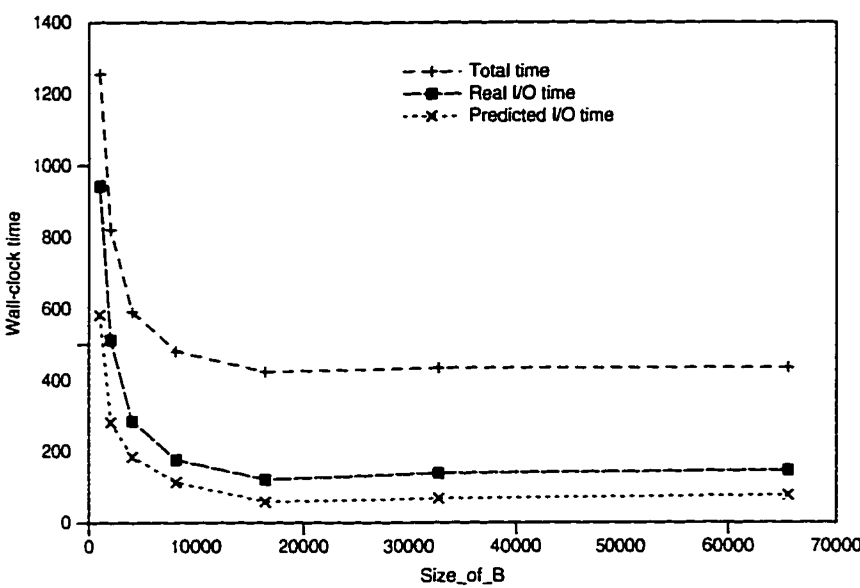


Figure 8.2: Out-of-core FFT computation (time in seconds) under various block sizes (in bytes).

8.3 Performance Issues

The method of program synthesis discussed in the previous section has been used to derive the implementation of the single precision floating-point Cooley-Tukey FFT algorithm on the CM-5 machine, where we ignore the bit-reversal operation and compute the twiddle factors on-line. The experiments were conducted on the CM-5 at Duke University. This CM-5 has two partitions, each of them has sixteen nodes. The SDA at this CM-5 has sixteen disks: fourteen data disks, one parity disk and one spare disk. The measurements used the CMMD communication library(Version 3.1 Final), the C compiler (gcc Version 2.6.0), and the CMOST operating system (Version 7.3 Final.1-rev.3). We discuss two performance issues: the influence of choosing different data distributions on performance; and the performance gain by using the parallel file access.

As discussed in the previous chapter, we can use function $f(b)$ to predict the performance of the synthesized programs. The optimal distribution can be obtained by finding the minimum of the function $f(b)$. In order to do this, we need to determine the values of τ_s and τ_e. However, our experiments on the CM-5 show that τ_s and τ_e are not constants for various messages stored to the SDA through the global synchronous sequential file mode. We therefore take the following approach. We determine the values of $\tau_s + 2^b\tau_e$ by directly measuring the execution time when every node processor accesses a block of 2^b records through the global synchronous sequential mode. Since $BD < M$, we have a small number of possible choices for b (remember that $B = 2^b$). We therefore compute the minimum of the function $f(b)$ by directly comparing all of the possible values of $f(b)$ for different b. Fig. 8.2 shows a comparison of the predicted I/O time with the real I/O and the total execution times given that $N = 2^{24}$, $M = 2^{21}$, $D = P = 2^4$, and $2^{10} \leq B \leq 2^{16}$. From this figure, we can make the following observations.

- The real I/O time and the total execution time decrease and then increase as the size of data distribution increases. Therefore a careful choice of data distribution is critical for optimizing the performance.

- Three curves for the predicted I/O time, the real I/O time, and the total execution time have the same shape. Further, the value b (in this case $b = 14$) which results in the predicted minimal I/O time also results in the real minimal I/O time and the total execution time. Therefore our prediction function is accurate. It also suggests that for this out-of-core application, by optimizing I/O performance, we can improve the performance for the overall computation.

Fig. 8.3 shows a comparison of the execution times of using parallel I/O operations (the global synchronous sequential file mode) with the execution times of using

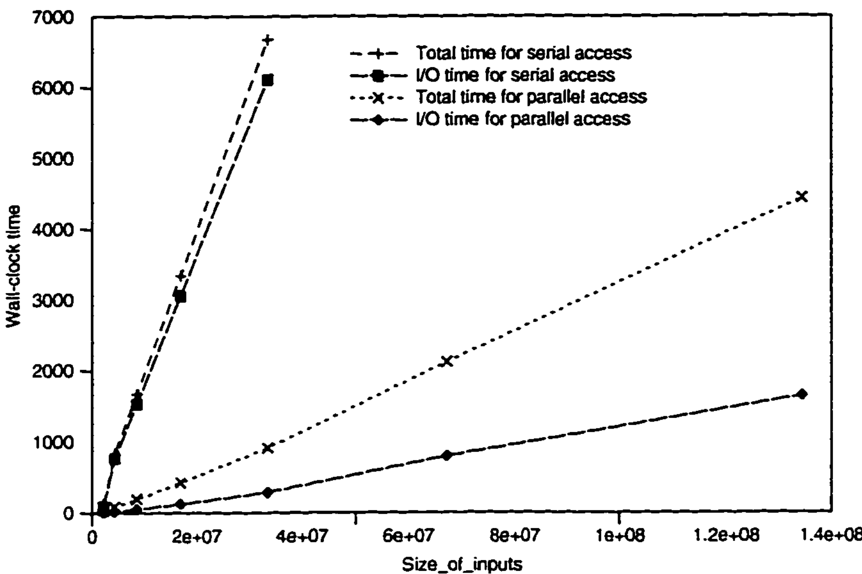


Figure 8.3: Execution times (in seconds) of using serial file access vs parallel file access for out-of-core FFT on the CM-5 with 16 nodes.

serial I/O operations (the global independent file mode). We use the following values for the parameters in the machine model. $M = 2^{21}$, and $D = P = 2^4$. B is obtained from evaluating the minimum of the function $f(b)$. The efficient values of B at the typical inputs for both the serial and the parallel file accesses are summarized in Table 8.1. The sizes of the inputs range from $1M$ to $128M$ (However, for the serial file access, the largest N is $32M$. The reason is that the execution time increases very fast and we can not get results for inputs whose sizes are larger that $32M$.). Note that the efficient values of B for the serial file access are also obtained from evaluating the minimum of the function $f(b)$. Moreover, similar with the parallel file access, our experiments on the CM-5 show that τ_s and τ_e are not constants for various messages stored to the SDA through the serial file access. Therefore, we determine the values of $\tau_s + 2^b\tau_e$ by directly measuring the execution time when all of the node processors access a block of 2^b records through the global independent file mode.

	Size-of-block	
Size-of-inputs	Serial	Parallel
2097152	4096	65536
4194304	4096	65536
8388608	4096	32768
16777216	4096	16384
33554432	4096	32768
67108864		16384
134217728		16384

Table 8.1: B values chosen from the prediction function.

The following results can be drawn from Fig. 8.3.

- By using the parallel file access, the ratio of the I/O overhead over the total execution time can be significantly reduced compared with that of using the serial file access.
  - For the serial file access, the I/O operations take roughly 90% of the total execution time, even though we use the prediction function to choose an "efficient" block size for data distribution.
  - For the parallel file access, when we choose the block size for data distribution from the prediction function, the I/O overhead takes a relatively smaller portion of the total execution time ranging from 17% to 37%.
- The speedup of using the parallel file access over the serial file access is significant, and the average is about a factor of 10.

Fig. 8.4 shows the performance of the out-of-core FFT implementation in terms of MFLOPS. The floating-point operation count for the FFT computation is derived from the total running time divided by the total number of floating point operations

required for a given problem size, where the total number of floating point operations for inputs of size N is estimated as $5N \log N$.

From the figure, we can see that the performance ranges from 4 MFLOPS to 6 MFLOPS. They scale very well for the reason that we choose an efficient data distribution to optimize the performance. These results are very encouraging for out-of-core FFT computations. To the best of our knowledge, the only other out-of-core implementation of the FFT computation is presented in [52] by Kuszmaul. Kustzmaul discussed out-of-core FFT performance implemented on the CM-5 and the iPSC/860. The CM-5 implementation outperforms the iPSC/860 implementation. Further, the CM-5 implementation has a performance between 5 MFLOPS and 10 MFLOPS, on a 32-node CM-5 machine. Considering that we are using only half of their number of processors, our results are very encouraging. However, we ignored the initial bit-reversal operation, which may take additional 30% running time. We conclude that our results are at least comparable with the results presented in that paper.

8.4 Conclusions

Using the framework developed in the previous chapter, we have synthesized a parallel out-of-core FFT program for the CM-5. Timing results of this implementation show that the distribution of out-of-core data has a large influence on the performance of the synthesized programs. Further, the I/O performance cost predictor is very effective. The performance predictor for determining the optimal data distribution can result in programs with both minimized I/O time and overall execution time. Furthermore, the synthesized program using the predicted data distribution and the parallel file operations run up to 10 times faster than those using the serial file access. Finally,

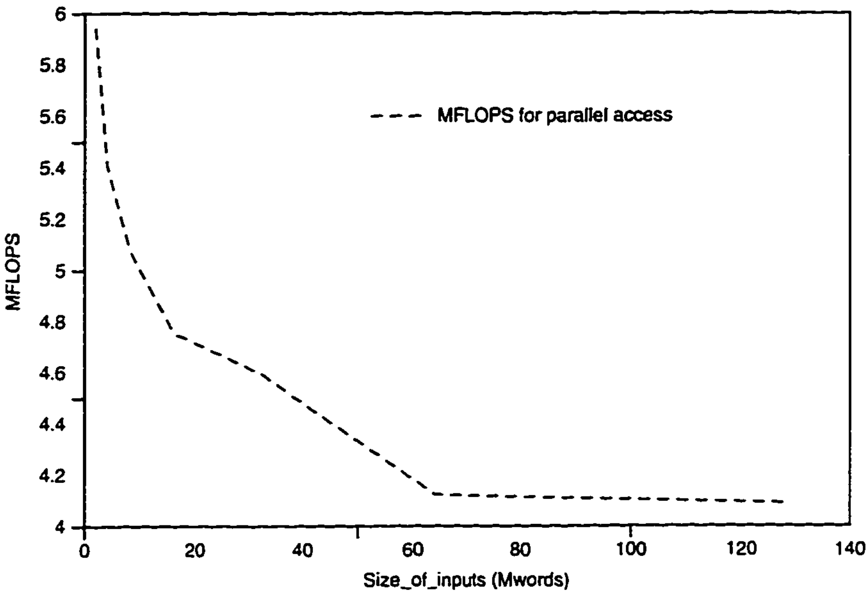


Figure 8.4: MFLOPS of out-of-core FFT program on the CM-5 with 16 nodes.

the synthesized program has a comparable performance with the well-known results presented in [52]. Our experimental results are very encouraging, considering that our programs can be generated automatically.

Chapter 9

Conclusions

In this thesis, we have presented significant results in two areas: models of parallel computation and tools for automatically synthesizing efficient programs for out-of-core block recursive algorithms.

9.0.1 Contributions

In the area of models of parallel computation, we have introduced a concept of resource metrics for capturing various models for parallel computation. The concept of resource metrics can be used not only to understand existing models, but also to guide the design of new models. As an example of using the framework of resource metrics to design new models, we have introduced the LogP-HMM model and the LogP-UMH model to capture the distributed-memory machine with multi-level memories. We have developed several near-optimal FFT and sorting algorithms for both models.

In the area of methods of program synthesis, we have presented a methodology based on tensor bases to describe the semantics of out-of-core data organization and access patterns on multi-disk systems. This methodology of data organization gen-

eralizes the idea of using tensor bases to represent in-core regular data distributions presented in [37]. Moreover, we have first presented the methodology of using tensor products to capture the semantics of data access patterns. Further, we have introduced the tensor product framework for synthesizing programs for a deeper level of memory hierarchy. Specifically, we have presented methods for synthesizing efficient out-of-core programs for two variants of parallel I/O models using tensor products. We have demonstrated the effectiveness of our approach by synthesizing efficient out-of-core FFT programs for the Thinking Machine CM-5.

9.0.2 Future Research

We discuss some of the future research directions on both models and program synthesis.

Models of parallel computation.

New algorithms and models can be developed based on our framework. We have developed several algorithms such as the FFT and sorting algorithms for the LogP-HMM model. Many other algorithms and other sorting algorithms need to be designed for this model. It would be also useful to examine the performance of these algorithms on practical parallel machines with hierarchical memories.

There are many research efforts on building programming environment for Valiant's BSP parallel model. However, the BSP model does not include a component for handling memory hierarchies. Therefore, our framework based on the LogP-HMM model may be used to build a BSP-based hierarchical memory model.

Program synthesis.

Our framework of program synthesis can be extended in the following aspects.

- Describe the semantics of file structures for the IBM's Parallel I/O File System (PIOFS) and the MPI-IO. The major feature of the PIOFS is that it can support high-dimensional structures of file data. More specifically, a file in the PIOFS can span multiple disks and servers. Moreover, the file can be logically divided into subfiles. Each subfile contains a portion of the file data. This data distribution of file data on multiple servers can be described by such as block-cyclic data distribution primitives. As we have demonstrated in this thesis, tensor bases can be used to describe the semantics of data distributions on a multi-dimensional storage structure. Therefore, it is possible to describe the data organization supported by the PIOFS. Further, using the methodology developed in this thesis, we can synthesize out-of-core block recursive algorithms on top of the PIOFS. Since the MPI-IO provides a similar abstraction as the PIOFS for supporting high-dimensional file organization, we can also develop methods of program synthesis for the MPI-IO.

- Synthesize out-of-core HPF programs. The approaches used by Rice and Syracuse on developing parallel out-of-core compilers are very similar to our approach. The most interesting part is that we are using very similar out-of-core models. Thus, for example, we may be able to use our algebraic methodology to describe out-of-core data distributions on their model and therefore generate out-of-core HPF programs for a class of computations. Also, our method can help to generate I/O operations and communications for their compilation systems.

- Overlap I/O with communication. In Chapter 7, we present the method of synthesizing I/O-efficient programs and then synthesizing communication-efficient programs by in-core methods. However, there are two drawbacks in that ap-

proach. First, we have assumed that before we perform in-core computation, we have loaded all of the required records. For example, we put a barrier synchronization between constructing memory-loads and in-core computation. This is not efficient because we have to wait for the completion of I/O operations before we can perform any in-core computation. Second, when we load tracks, we have not taken the requirements of in-core data distribution into account.

One solution for the first problem is to overlap I/O operations with in-core computation, which may include both in-core communication and local computation. For example, after loading the second track, we can issue I/O operations to load the successive track. At the same time, we can send the loaded blocks of the first track in each processor to their destination processors according to the requirements of in-core data distribution, and we can also perform some local computation to the first loaded track by each processor (this may depend on whether the blocks loaded in the first track have been sent to their destination processors.) However, the following example shows that this naive approach may result in inefficient communication. Assume that we want to compute $I_2 \otimes F_{32}$. And let $D = P = 4, M = 32, B = 2$, and $N = 64$. Thus, the out-of-core data is distributed according to the $cyclic(2)$ distribution. We can first load the first four tracks and perform the in-core F_{32} computation, and then load the successive four physical tracks and again perform the other in-core F_{32} computation. Since the two sub-computations are the same, we now consider the first in-core F_{32} computation.

According to the method of program synthesis discussed in Chapter 7, before we perform the first in-core F_{32} computation, the in-core data is distributed according to the $cyclic(2)$ distribution as shown in Fig. 9.1 (a). In that figure,

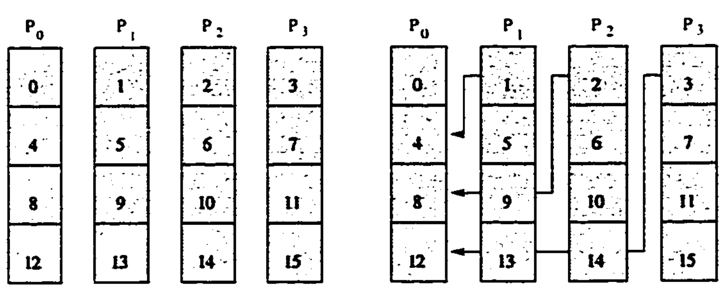


Figure 9.1: In-core data distribution and redistribution.

each box is a block, and the number in each box is the global index number of the corresponding block. According to the in-core method of program synthesis discussed in [37], F_{32} can be computed efficiently using one-communication step and two-computation steps if the initial data is distributed according to the block distribution. In order to distribute the in-core data in block distribution, all of the three blocks in processors $1, 2$ and 3 loaded from the first track will need to be sent to processor 0 as shown in Fig. 9.1 (b). It is obvious that if we overlap the communication required for moving the blocks loaded from the first track with I/O operations required for loading the second track, then the processor 0 will become the bottleneck of communication. In other words, the unefficiency is coming from the unbalanced communication patterns. We next discuss an approach to load blocks which can achieve the desired in-core data distribution with balanced communication patterns, and therefore can solve the above two problems simultaneously.

In other words, our goal is that when we overlap communication with I/O operations, we want the loaded blocks in a track can form a balanced communication pattern. More specifically, each processor will send and receive a block.

We achieve this by allowing each processor to load blocks independently. However, for the above example, we show that instead of using "fully" independent disk access, we can still load blocks of a track in a regular pattern. And the loaded blocks can be used to generate balanced communication patterns. The main idea is that we will introduce another definition of tracks as unit of data access. If we call the originally defined tracks *horizontal tracks*, then we will use *diagonal tracks* to load blocks. For example, in the top part of Fig. 9.2, we show four different diagonal tracks, which are distinguished by different shadows. Fig. 9.2 also shows that by simply shifting the loaded blocks in each diagonal track among processors, we can achieve the block distribution for each memory-load of in-core data. In that figure, the middle part shows the shift operations for each diagonal track. The bottom part shows the final block distribution of the in-core data.

Thus, an interesting research topic is to generalize this idea and develop a method for overlapping communication and I/O operations.

- Incorporate our methodology into the APRA supported project, EXTENT (An *E*xpert system for *TEN*sor product *T*ransform), developed at Ohio State University. Currently, EXTENT supports program synthesis for both shared and distributed-memory machines. The incorporation of our methodology will enhance the functionality of EXTENT. Hence, a general tool of program synthesis for both out-of-core and in-core computation can be constructed in a single system.

- Another research direction is to extend the tensor product framework to express a wider class of algorithms. In this thesis, we have limited our discussion to a class of block recursive algorithms, where we mainly consider that

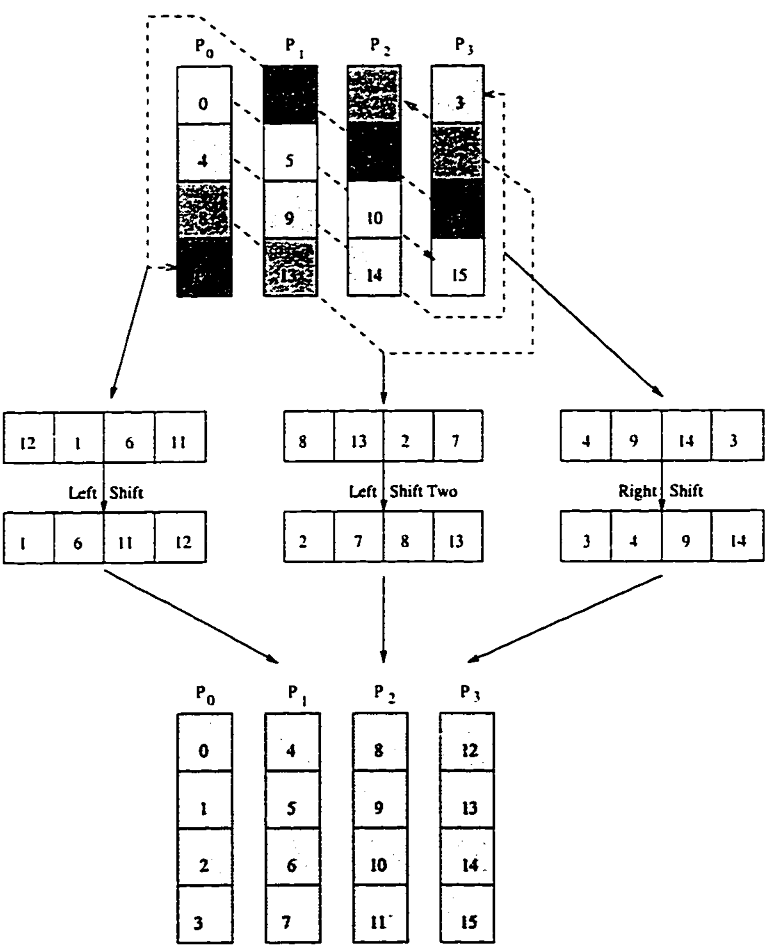


Figure 9.2: Diagonal tracks and data redistribution.

the computational matrix is a square linear transformation. An extension is to allow the computational matrix to be a rectangular linear transformation. Further, these block recursive algorithms are a sub-class of regular problems. An interesting question is whether we can extend the framework of tensor products to express other types of algorithms including irregular algorithms. Another limitation of our approach is that algorithms must be expressed as a tensor product formula. This may not be a problem for people with sound mathematical background. However, it may take time for normal programmers to understand this notation. One solution is to develop algorithms to learn the computational structures from source programs and identify the structures which can be represented by tensor products. Then, we can apply our methodology to those portions of the programs.

Appendix A

Computing Sub-Arrays Using Tensor Bases

As we have mentioned in Chapter 5 and Chapter 6, one of the important operations in constructing a memory-load is to keep portions of the records from the loaded physical track. Since the records in the physical track can be regarded as a multi-dimensional array, keeping portions of the records in this physical track can be implemented as taking sub-arrays from the multi-dimensional array of the physical track. In this appendix, we show how to implement this operation using the properties of tensor bases.

We begin with the following simple example. Assume that we have a vector X of length 64. We view the vector as a three-dimensional array denoted by the tensor basis $\mathcal{D} = e_i^4 \otimes e_j^4 \otimes e_k^4$, called *data basis*. Further, assume that we want to access the records in X in the following order: for each value of j, where j takes the values from 0 to 3, access all of the $[i, j, k]$-th records, where $0 \leq i, k \leq$. Fig. A.1 shows this access pattern. In terms of tensor bases, to form this data access pattern, we can construct a loop basis by making e_j^4 as the first factor of $\mathcal{D}$. In other words, we have the loop basis $\mathcal{L} = e_j^4 \otimes e_i^4 \otimes e_k^4$. Using the data basis $\mathcal{D}$ and the loop basis $\mathcal{L}$, we can write the following program to implement the access pattern specified in Fig. A.1.

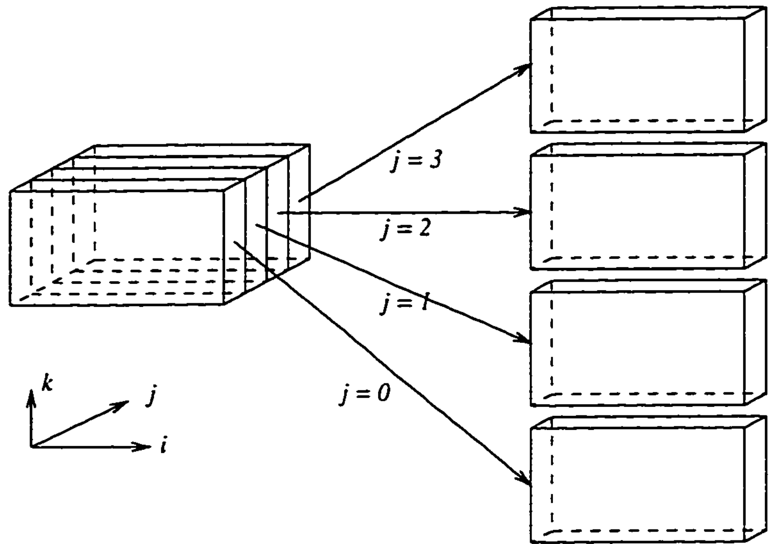


Figure A.1: Constructing sub-arrays from the input vector or array.

DO $j = 0, 3$
DO $i = 0, 3$
$Y(4i : 4i + 3) \leftarrow X(16i + 4j : 16i + 4j + 3)$
ENDDO ENDDO

In the above program, we have not generated the explicit loop nest for index k. Instead, we use $Y(i : j)$ to denote the contiguous values from i to j of Y, where Y is a vector which holds the records of each sub-array. The index ranges of the interval in Y are obtained by linearizing $e_i^4 \otimes e_k^4$ to form the indexing function $4i + k$, and then adding $-k$ and $-k + 3$ to it, respectively. The index ranges of the interval in X are obtained by linearizing the original tensor basis $\mathcal{D} = e_i^4 \otimes e_j^4 \otimes e_k^4$ to form the indexing function $16i + 4j + k$ and then adding $-k$ and $-k + 3$ to it, respectively.

In general, we consider the following tensor basis,

$$\mathcal{D} = e_{i_k}^{M_k} \otimes e_{j_k}^{N_k} \otimes e_{i_{k-1}}^{M_{k-1}} \otimes e_{j_{k-1}}^{N_{k-1}} \otimes \cdots \otimes e_{i_1}^{M_1} \otimes e_{j_1}^{N_1},$$

which is used to denote a vector X of length $M_k \cdots M_1 N_k \cdots N_1$. Thus, $\mathcal{D}$ is the data basis for the vector X. Moreover, after linearizing this data basis, we can use the indexing function to access the vector in the natural order. We now assume that each time we want to access the records denoted by the following sub-tensor basis:

$$e_{j_k}^{N_k} \otimes e_{j_{k-1}}^{N_{k-1}} \otimes \cdots \otimes e_{j_1}^{N_1}. \tag{A.1}$$

We can achieve this data access pattern by constructing the following loop basis,

$$\mathcal{L} = e_{i_k}^{M_k} \otimes e_{i_{k-1}}^{M_{k-1}} \otimes \cdots \otimes e_{i_1}^{M_1} \otimes e_{j_k}^{N_k} \otimes e_{j_{k-1}}^{N_{k-1}} \otimes \cdots \otimes e_{j_1}^{N_1}.$$

In other words, we use the instantiation of indices i_s, $1 \leq s \leq k$, to choose different sub-arrays, each of these sub-arrays is denoted by the sub-tensor basis in Formula A.1. The following program implements the access pattern described above,

```
DO i_k = 0, M_k - 1
  DO i_{k-1} = 0, M_{k-1} - 1
    ⋱
      DO i_1 = 0, M_1 - 1
        // Keep portions of records in Y
        DO j_k = 0, N_k - 1
          DO j_{k-1} = 0, N_{k-1} - 1
            ⋱
              DO j_2 = 0, N_2 - 1
                Y(ylow : yhigh) ← X(xlow : xhigh)
        ENDDO ENDDO ENDDO
ENDDO ENDDO ENDDO
```

In the above program, Y is a vector which holds the records of each sub-array. The index ranges $[ylow, yhigh]$ of the interval in Y are obtained by linearizing the sub-tensor basis in Formula A.1 to form the indexing function

$$N_{k-1} \cdots N_1 j_k + \cdots + N_1 j_2 + j_1,$$

and then adding $-j_1$ and $-j_1 + N_1 - 1$ to it, respectively. Thus,

$$ylow = N_{k-1} \cdots N_1 j_k + \cdots + N_1 j_2,$$

$$yhigh = N_{k-1} \cdots N_1 j_k + \cdots + N_1 j_2 + N_1 - 1.$$

The index ranges $[xlow, xhigh]$ of the interval in X are obtained by linearizing the original tensor basis $\mathcal{D}$ to form the indexing function

$$N_k \cdots N_1 M_{k-1} \cdots M_1 i_k + N_{k-1} \cdots N_1 M_{k-1} \cdots M_1 j_k + \cdots N_1 i_1 + j_1,$$

and then adding $-j_1$ and $-j_1 + N_1 - 1$ to it, respectively. Thus,

$$xlow = N_k \cdots N_1 M_{k-1} \cdots M_1 i_k + N_{k-1} \cdots N_1 M_{k-1} \cdots M_1 j_k + \cdots + N_1 i_1,$$

$$xhigh = N_k \cdots N_1 M_{k-1} \cdots M_1 i_k + N_{k-1} \cdots N_1 M_{k-1} \cdots M_1 j_k + \cdots + N_1 i_1 + N_1 - 1.$$

Bibliography

[1] A. Aggarwal, B. Alpern, A.K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. 19th Symp. on Theory of Comp*, May 1987.

[2] A. Aggarwal, A.K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. 28th Symp. on Foundations of Comp. Sci.*, October 1987.

[3] A. Aggarwal, A.K. Chandra, and M. Snir. On communication latency in PRAM computation. In *Proc. of the ACM Symposium on Parallel Algorithms and Architecture*, pages 11–21, Santa Fe, New Mexico, June 1989.

[4] A. Aggarwal, A.K. Chandra, and M. Snir. Communication complexity of PRAM. *Communications of the ACM*, March 1990.

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Comm. of the ACM*, 31(9):1116–1127, Sept. 1988.

[6] B. Alpern, L. Carter, and E. Feig. The uniform memory hierarchy model of computation. *Algorithmica*, 12(2-3):72–109, Dec. 1994.

[7] B. Alpern, L. Carter, and J. Ferrante. Modeling parallel computers as memory hierarchies. In *First Int. Conference on Massively Parallel Programming Models*, IEEE, 1993.

[8] R. Bagrodia, A. Chien, Y. Hsu, and D. Reed. Input/output: Instrumentation, characterization, modeling and management policy. Technical Report CCSF-41, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[9] A. Bar-Noy and S. Kipnis. Designing broadcasting algorithms in the postal model for message-passing systems. In *Proc. of the ACM Symposium on Parallel Algorithms and Architectures*, June 1992.

[10] P. Beckman, D. Gannon, and E. Johnson. Portable parallel programming in HPC++. Technical Report, Indiana University, 1996.

[11] G. E. Blelloch. *Vector models for data-parallel computing*. The MIT Press, 1990.

[12] R. Bordawekar, A. Choudhary, K. Kennedy, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, July 1995. Also available as the following technical reports: NPAC Technical Report SCCS-0696, CRPC Technical Report CRPC-TR94507-S, SIO Technical Report CACR SIO-104.

[13] R. R. Bordawekar. *Compilation Techniques for I/O intensive parallel programs*. Ph.D. thesis, Syracuse University, Department of Electrical and Computer Engineering, 1996.

[14] J. Celuch. The 9076 SP1 high-peoformance communication network. Technical Report, KGNVMC, Kingston, 2 1994.

[15] M. Chen, Y. Choo, and J. Li. Compiling parallel programs by optimizing performance. *The Journal of Supercomputing*, 2:171–207, 1988.

[16] A. Choudhary, I. Foster, G. Fox, K. Kennedy, C. Kesselman, C. Koelbel, J. Saltz, and M. Snir. Languages, compilers, and runtime systems support for parallel input-output. Technical Report CCSF-39, Scalable I/O Initiative, Caltech Concurrent Supercomputing Facilities, Caltech, 1994.

[17] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, Mass., 1989.

[18] R. Cole and O. Zajicek. The APRAM: Incorporating asynchrony into the PRAM model. In *Proc. of the Symposium on Parallel Arch. and Algorithms*, pages 169–178, Santa Fe, New Mexico, June 1989.

[19] R. Cole and O. Zajicek. The expected advantage of asynchrony. In *Proc. 2nd ACM Symposium on Parallel Arch. and Algorithms*, pages 85–94, Crete, Greece, July 1990.

[20] P. F. Corbett, D. G. Feitelson, J.-P. Prost, G. S. Almasi, S. J. Baylor, A. S. Bolmarcich, Y. Hsu, J. Satran, M. Snir, R. Colao, B. Herr, J. Kavaky, T. R. Morgan, and A. Zlotek. Parallel file systems for the IBM SP computers. *IBM Systems Journal*, pages 222–248, 1995.

[21] Peter F. Corbett, Sandra Johnson Baylor, and Dror G. Feitelson. Overview of the Vesta parallel file system. In *IPPS '93 Workshop on Input/Output in Parallel Computer Systems*, pages 1–16, 1993. Also published in Computer Architecture News 21(5), December 1993, pages 7–14.

[22] T. H. Cormen. *Virtual Memory for Data-Parallel Computing*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[23] T. H. Cormen and A. Colvin. ViC*: A preprocessor for virtual-memory C*. Technical Report PCS-TR94-243, Dept. of Computer Science, Dartmouth College, November 1994.

[24] T. H. Cormen and D. Kotz. Integrating theory and practice in parallel file systems. Technical Report PCS-TR93-188, Dept. of Math and Computer Science, Dartmouth College, March 1993. Revised 9/20/94.

[25] D. Culler, R.M. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Asntos, R. Subramonian, and T.V. Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[26] D. E. Culler, A. C. Dusseau, R. P. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Portability and Performance for Parallel Processing*, pages 71–98. John Wiley & Sons, 1993.

[27] D. L. Dai, S. K. S. Gupta, S. D. Kaushik, J. H. Lu, R. V. Singh, C.-H. Huang, P. Sadayappan, and R. W. Johnson. EXTENT: A portable programming environment for designing and implementing high performance block recursive algorithms. In *Supercomputing '94*, pages 49–58, 1994.

[28] J. O. Eklundh. A fast computer method for matrix transposing. *IEEE Trans. on Computers*, 20(7):801–803, 1972.

[29] D. G. Feitelson, P. F. Corbett, Y. Hsu, and J.-P. Prost. Parallel I/O systems and interfaces for parallel computers. In C.-L. Wu, editor, *Multiprocessor Systems — Design and Integration*. World Scientific, 1995. To appear.

[30] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proc. 10th ACM Symposium on Theory of Computing*, pages 114 – 118, 1978.

[31] High Performance Fortran Forum. High performance fortran language specification. version 1.0. Technical Report CRPC-TR 92225, Rice University, 1 1993.

[32] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM : Accounting for contention in parallel algorithms. In *Proc. of Discrete*, pages 638–647, 1994.

[33] P.B. Gibbons. A more practical PRAM model. In *Proc. of the Symposium on Parallel Arch. and Algorithms*, pages 158–168, Santa Fe, New Mexico, June 1989.

[34] M. T. Goodrich. Parallel algorithms column I: Models of computation. *SIGART News*, 24(4):16–21, Dec. 1993.

[35] A. Graham. *Kronecker Products and Matrix Calculus: With Applications*. Ellis Horwood Limited, 1981.

[36] J. Granta, M. Conner, and R. Tolimieri. Recursive fast algorithms and the role of the tensor product. *IEEE Transaction on Signal Processing*, 40(12):2921–2930, Dec. 1992.

[37] S. K. S. Gupta. *Synthesizing Communication-efficient Distributed-Memory*

Parallel Programs for Block Recursive Algorithms. PhD thesis, The Ohio State Univ., March 1995.

[38] S. K. S. Gupta, C.-H. Huang, P. Sadayappan, and R. W. Johnson. A framework for synthesizing distributed-memory programs for block recursive algorithms. *J. Parallel and Distributed Computing*, 1996. To appear.

[39] S. K. S. Gupta, Z. Li, and J. H. Reif. Generating efficient programs for two-level memories from tensor products. In *Prof. of the Seventh IASTED/ISMM Int. Conf. on Parallel and Distributed Computing and Systems*, pages 510–513, Washington,D.C.,USA, Oct. 1995.

[40] C.-H. Huang, J. R. Johnson, and R. W. Johnson. A tensor product formulation of Strassen's matrix multiplication algorithm. *Appl. Math Letters*, 3(3):67–71, 1990.

[41] C.-H. Huang, J. R. Johnson, and R. W. Johnson. Generating parallel programs from tensor product formulas: A case study of Strassen's matrix multiplication algorithm. In *Proc. Int'l Conf. Parallel Processing 1992*, pages 104–108, Aug. 1992.

[42] P. J. Hatcher J. A. Moore and M. J. Quinn. Stream*: Fast, flexible data-parallel I/O. In *Parallel Computing'95*, Ghent,Belgium, September 1995.

[43] J. JáJá. *An introduction to parallel algorithms*. Addison-Wesley Publishing Company, 1989.

[44] J. R. Johnson, R. W. Johnson, D. Rodriguez, and R. Tolimieri. A methodology for designing, modifying and implementing Fourier transform algorithms on

various architectures. *Circuits Systems and Signal Processing*, 9(4):450–500, 1990.

[45] R. W. Johnson, C.-H. Huang, and J. R. Johnson. Multilinear algebra and parallel programming. *J. Supercomputing*, 5:189–218, 1991.

[46] R.M. Karp, A. Sahay, and E. Santos. Optimal broadcast and summation in the LogP. Technical Report, UCB, 1993.

[47] S. D. Kaushik, , C.-H. Huang, R. W. Johnson, and P. Sadayappan. An approach to communication-efficient data redistribution. In *the 8th ACM International Conference on Supercomputing*, 1994.

[48] S. D. Kaushik, C.-H. Huang, J. R. Johnson, R. W. Johnson, and P. Sadayappan. Efficient transposition algorithms for large matrices. In *Supercomputing '93*, Nov. 1993.

[49] S. D. Kaushik, C.-H. Huang, R. W. Johnson, and P. Sadayappan. A methodology for generating efficient disk-based algorithms from tensor product formulas. In *Proc. Sixth Annual Workshop on Languages and Compilers for Parallel Computing*, pages 358–338, Aug. 1993.

[50] J. Kornerup. *Mapping powerlists onto Hypercubes*. Ph.D. thesis, The University of Texas at Austin, Department of Computer Science, 1995.

[51] B. Kumar, C.-H. Huang, P. Sadayappan, and R. W. Johnson. An algebraic approach to cache memory characterization for block recursive algorithms. In *1994 International Computer Symposium*, pages 336–342, 1994.

[52] C. L. Kuszmaul. Out of core ffts in parallel application environment. Technical Report RND-93-013, NASA AMES Research Center, 1993.

[53] T. T. Kwan and D. A. Reed. Performance of the CM-5 scalable file system. In *Proceedings of the 8th ACM International Conference on Supercomputing*, pages 156–165, July 1994.

[54] T. Leighton. Tight bounds on the complexity of parallel sorting. *IEEE Transactions on Computers*, 34(3):344–354, 1985.

[55] Z. Li, P. Mills, and J. Reif. Models and resource metrics for parallel and distributed computation. *Parallel Algorithms and Applications*, 8(35-59), 1996. To appear. A short version of this paper is appeared in the Proc. of HICSS'28.

[56] Z. Li, G. G. Pechanek, C. J. Glossner, and C. H. L. Moller. Design and implementation of FFT algorithms for M.F.A.S.T. using tensor products. IBM Technical Report 29.2126, IBM Corporation, April 1996.

[57] C. V. Loan. *Computational frameworks for the fast Fourier transform*. SIAM, 1992.

[58] S. J. LoVerso, M. Isman, A. Nanopoulos, W. Nesheim, E. D. Milne, and R. Wheeler. *sfs*: A parallel file system for the CM-5. In *Proceedings of the 1993 Summer USENIX Conference*, pages 291–305, 1993.

[59] B.M. Maggs, L.R. Matheson, and R.E. Tarjan. Models of parallel computation: A survey and synthesis. In *Proc. 28th Annual Hawaii International Conference on System Sciences*, pages 61–70, Hawaii, January 1996.

[60] J. Misra. Powerlist: A structure for parallel recursion. *ACM Trans. on Parallel Language and Systems*, 11, 1994.

[61] S. A. Moyer and V. S. Sunderam. PIOUS: a scalable parallel I/O system for

distributed computing environments. In *Proceedings of the Scalable High-Performance Computing Conference*, pages 71–78, 1994.

[62] S. A. Moyer and V. S. Sunderam. Characterizing concurrency control performance for the PIOUS parallel file system. Technical Report CSTR-950601, Emory University, June 1995.

[63] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. of ACM SPAA*, July 1991.

[64] *Grand Challenges: High Performance Computing and Communications*. A Report by the Committee on Physical, Mathematical and Engineering Sciences, NSF/CISE, 1800 G Street NW, Washington, D.C. 20550, 1991.

[65] M. Paleczny, K. Kennedy, and C. Koelbel. Compiler support for out-of-core arrays on data parallel machines. In *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Computation*, pages 110–118, McLean, VA, February 1995.

[66] G. G. Pechanek, C. J. Glossner, Z. Li, C. H. L. Moller, and S. Vassiliadis. Tensor product FFT's on M.f.a.s.t.: A highly parallel single chip DSP. In *DSP'95, Digital Signal Processing and its Applications*, Paris, France, Oct. 1995.

[67] F. P. Preparata and J. Vuillemin. The cube-connection cycles: A versatile network for parallel computation. *Comm. of the ACM*, 24(5):300–309, May 1981.

[68] J.-P. Prost, M. Snir, P. Corbett, and D. Feitelson. MPI-IO, a message-passing interface for concurrent I/O. Technical Report RC 19712 (87394), IBM T.J. Watson Research Center, August 1994.

[69] A. Purakayastha, C. S. Ellis, D. Kotz, N. Nieuwejaar, and M. Best. Characterizing parallel file-access patterns on a large-scale multiprocessor. In *Proceedings of the Ninth International Parallel Processing Symposium*, pages 165–172, April 1995.

[70] J. H. Reif. *Synthesis of parallel algorithms*. the Benjamin/Cummings Publishing Company, Inc, 1993.

[71] J. M. Rosario and A. Choudhary. High performance I/O for parallel computers: Problems and prospects. *IEEE Computer*, 27(3):59–68, March 1994.

[72] D. J. Rose. Matrix identities of the fast Fourier transform. *Linear Algebra and its Applications*, 29(2):423–443, 1980.

[73] A. Sahay. Hiding communication costs in bandwidth-limited parallel FFT computation. Technical Report, UCB, 1993.

[74] D. B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133–158, 1991.

[75] L. Snyder. Type architecture, shared memory, and the corollary of modest potential. *Annual Review of Computer Science*, pages 289–317, 1986.

[76] H. S. Stone. Parallel processing with the perfect shuffle. *IEEE Trans. on Computers*, 20(2):153–161, 1971.

[77] C. B. Stunkel, D. G. Dhea, D. G. Grice, P. H. Hochschild, and M. Tsao. The SP1 high-performance switch. In *Proc. of the Scalable High Performance Computing Conference*, pages 150–157, Knoxville, TN, May 1994.

[78] R. Thakur, R. Bordawekar, and A. Choudhary. Compilation of out-of-core data parallel programs for distributed memory machines. In *IPPS '94 Workshop on Input/Output in Parallel Computer Systems*, pages 54–72. Syracuse University, April 1994. Also appeared in Computer Architecture News 22(4).

[79] R. Thakur, R. Bordawekar, A. Choudhary, R. Ponnusamy, and T. Singh. PASSION runtime library for parallel I/O. In *Proceedings of the Scalable Parallel Libraries Conference*, pages 119–128, October 1994.

[80] Thinking Machine Corporation. DPEAC reference manual, 1992.

[81] *The Connection Machine CM-5 Technical Summary*. Thinking Machines Corporation, October 1991.

[82] L.G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[83] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[84] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory II: Hierarchical multilevel memories. *Algorithmica*, 12(2-3):148–169, 1994.

[85] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1996.

Biography

Zhiyong Li was born in Wuhan, P.R. China, in 1963. He received his B.S. and M.S. degrees in Computer Science and Engineering from Huazhong University of Science and Technology, P.R. China, in 1984 and 1987, respectively.

Zhiyong has a strong interest in science and education. He has devoted himself in research for over ten years. He has also spent five years as a lecturer teaching computer science at Huazhong University of Science and Technology. He is a student member of ACM.

Zhiyong has worked in several different areas including parallel computing and Artificial Intelligence. He has published papers in both areas. Several his publications are listed below,

1. "Models and Resource Metrics for Parallel and Distributed Computation", Z. Li, P. H. Mills and J. H. Reif, Journal of Parallel Algorithms and Applications, Vol.8, pp.35-59, 1996. A short version appeared in Proc. 28th Annual Hawaii International Conference on System Sciences (HICSS-28 Parallel Algorithms Software Technology Track), IEEE Press, 1995.

2. "The Proteus System for the Development of Parallel Applications", A. Goldberg, J. Prins, J. Reif, R. Faith, Z. Li, P. Mills, L. Nyland, D. Palmer, J. Riely

and S. Westfold, April 1994, in Prototyping Languages and Prototyping Technology, M. Harrison, ed., (to appear) Springer-Verlag, 1996. (40 pp)

3. "Synthesizing Efficient Out-of-Core Programs for Block Recursive Algorithms using Block-Cyclic Data Distributions", Z. Li, J. H. Reif and S. K. S. Gupta, Technical Report, CS-1996-04, Duke University, 1996. A short version is to appear in International Conference on Parallel Processing, 1996.

4. "Generating Efficient Programs for Two-level Memories from Tensor Products", S. K. S. Gupta, Z. Li and J. H. Reif, Proceedings of the Seventh International Conference on Parallel and Distributed Computing and Systems, Washington, D.C., Oct., 1995.

5. "Tensor Product FFT's on M.f.a.s.t.: A Highly Parallel Single Chip DSP", G. G. Pechanek, C. J. Glossner, Z. Li, C. H. L. Moller, and S. Vassiliadis, DSP'95, Digital Signal Processing and its Applications, Paris, Oct., 1995.

6. "RFUNLOG: A Reduction Based Functional/Logical Language", , Z. Li, Y. Fuming, H. Yin, and Y. Zhang, Computer Journal, Vol. 19, 1-3(19 91). (in Chinese)

7. "ECES: An Electroplating Consulation Expert System", Z. Li and Y. Zhang, J. of HUST, Vol. 17, 4(1989). (in Chinese)

8. "LISP-ELP: Design and Applications", Y. Zhang and Z. Li, Chinese Journal of Computers, Vol. 12, 2(1989). (in Chinese)

He has published a software.

1. "LISP/PROLOG language(a software product)", Z. Li and Y. Zhang, HUST Press, 1992, P.R.China.