Structure Driven Multiprocessor Compilation of Numeric Problems

by

G. N. Srinivasa Prasanna

B.Tech, Electrical Engineering
Indian Institute of Technology, Kanpur
(1983)

M.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1986)

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
in partial fulfillment of the requirements
for the degree of
DOCTOR OF PHILOSOPHY

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
February 1991

© 1991 Massachusetts Institute of Technology

Signature of Author: ______________________________
Department of Electrical Engineering and Computer Science
January 22, 1991

Certified by: ______________________________
Bruce. R. Musicus
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: ______________________________
A. C. Smith
Chairman, Departmental Graduate Committee

MASSACHUSETTS INSTITUTE
OF TECHNOLOGY

APR 03 1991

LIBRARIES

Structure Driven Multiprocessor Compilation of Numeric Problems

by

G. N. Srinivasa Prasanna

Submitted to the Department of Electrical Engineering and Computer Science
on January 22, 1991 in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy
in Electrical Engineering and Computer Science

ABSTRACT

The optimal automatic compilation of computation intensive numeric problems onto multiprocessors is of great current interest. While optimal compilation is NP-Hard in general, the extensive structure present in many numeric algorithms greatly facilitates their optimal compilation. This thesis explores the application of a hierarchical compilation paradigm for such algorithms. These algorithms can be specified as matrix expressions composed of matrix sums, products, and inverses, FFTs, etc. Good compilations for these algorithms can be derived by composing together good routines for these basic operators, thus yielding a hierarchical compilation strategy.

The first part of this thesis we explore the use of the extensive structure present in matrix operations to derive close to optimal routines for them, thus creating a parallel library. We show that these operator routines vary in a smooth fashion over a space of parameterised architectures.

We then present a theoretical framework for optimally composing together good library routines to generate a good compilation for the entire matrix expression dataflow graph. Classical scheduling theory is generalised for this purpose. Each operator in the expression is identified with a dynamic system (task). The state of a task represents the amount of computation completed. Computing the matrix expression is equivalent to traversing the state space from the initial uncomputed state to the final computed state, using the processor resources. This casts the problem in the framework of control theory. Fundamental new insights into multiprocessor scheduling can be obtained from this formulation.

Optimal control theory is applied to identify time-optimal control strategies with optimal schedules. A number of powerful results can be derived under very general assumptions. For certain types of (convex) task dynamics, it is shown that optimal scheduling is equivalent to shortest path and flow problems. This leads to very simple strategies for scheduling dataflow graphs composed of such tasks. These strategies have been applied to scheduling matrix expressions. A compiler utilizing these techniques has been written, generating MUL-T code for the MIT Alewife machine, and the theory validated.

Thesis Supervisor: Bruce. R. Musicus
Title: Professor of Electrical Engineering and Computer Science

To Tata,
For all that you did for me.

Acknowledgments

The work described in this dissertation has benefited from a vast variety of friends. Firstly, I must gratefully acknowledge my debt to my advisors, Prof. Bruce Musicus and Prof. Ananth Agarwal. I thank Bruce for all the guidance and support I received from him, and for tolerating my many faults. His extensive and deep knowledge and super sharp reasoning power were most valuable to my thesis. I learned a great deal about multiprocessors from Ananth. Without the Alewife Multiprocessor facilities I received from Ananth, the experimental results would have been impossible to obtain. Next I must thank my readers, Prof. Jonathan Allen and Prof. Arvind. Their feedback about various portions of the thesis was most valuable.

Any research is a group activity, and mine is no exception. I have benefited extensively from discussions with the entire MIT Alewife Group. My friends Dave Chaiken, Beng-Hong Lim, Kirk Johnson, Dann Nussbaum, John Kubiatowicz, Dave Kranz, and many others were critical to my work on the Alewife Machine. They were always ready to answer my questions, simple or complex, stupid or deep. Without them the thesis would have gone nowhere. I have also benefited extensively with discussion with members of the MIT DSPG group, in particular Dennis Fogg and Michelle Covell. May their tribe increase.

The quality of a person's work depends as much on personal support as on the professional environment. In this regard I have been extremely lucky to have a very supporting family. Tata will always be a source of inspiration to me, and will forever remind me of the value of true scholarship. Amma and Anna have displayed infinite patience in waiting for their very lazy son to finish his studies, and finally start earning! Seema, you have been so considerate in waiting for your useless husband to start work! Chacha, I do wish I emulate you someday, and give some of my life to the poor and the downtrodden. Without you all I would never have gone so far in life. I never want to lose you.

Contents

1 Structure Driven Compilation of Numeric Problems **10**
1.1 Overview . . . 10
1.2 Thesis Outline . . . 11
1.3 Multiprocessor Model . . . 14
1.4 Design of the Parallel Operator Library . . . 15
1.4.1 Characterization of basic matrix operators . . . 15
1.4.2 Partitioning . . . 15
1.4.3 Scheduling . . . 16
1.4.4 Examples . . . 16
1.5 Composing Parallel Operator Library routines . . . 17
1.5.1 Characterization of the problem . . . 17
1.5.2 Formal Specification . . . 18
1.5.3 Control Theoretic Formulation of Scheduling . . . 19
1.5.4 Results from Control Theory . . . 20
1.6 Implementation and Experimental Validation . . . 20
1.7 History of the Problem . . . 21
1.8 Summary . . . 25

2 Optimal Matrix Operator Routines **26**
2.1 Overview . . . 26
2.2 Multiprocessor model . . . 26
2.3 Optimal compilation . . . 27
2.4 Exploiting Structure in Matrix Operator Dataflow Graphs . . . 28
2.4.1 Polyhedral Representation of Matrix Operator Dataflow Graphs . 29
2.5 Matrix Sums . . . 32
2.6 Matrix Products . . . 33
2.6.1 Continuous Approximation . . . 34
2.6.2 Lower Bounds . . . 35
2.6.3 Previous Work on Partitioning Techniques . . . 42
2.6.4 Heuristics for Partitioning Matrix Products . . . 43
2.6.5 Experimental results . . . 53
2.7 Continuous processors and mode shifts . . . 53

2.7.1 Extension of Lower Bounds 56
2.7.2 Mode Shifts 57
2.8 Special cases of Matrix Sums and Products 57

3 Generalised Scheduling **62**
3.1 Introduction 62
3.2 Model of the Parallel Task System 65
3.3 Setting up the Optimal Control Solution 68
3.4 Solution Method 71
3.5 Speedup Function p^{α} 73
3.5.1 Series Tasks 74
3.5.2 Parallel Tasks 75
3.5.3 Homogeneity 75
3.6 Simplified Scheduling Algorithm 77
3.6.1 Series and Parallel Decompositions 77
3.6.2 Examples of Optimal Schedules 81
3.6.3 Trees and Forests 81
3.6.4 General Directed Acyclic Graphs (DAG's) 83
3.7 p^{α} Dynamics and Shortest Path Scheduling 85
3.8 Discussion and Summary 88

4 Experimental Results **89**
4.1 Introduction 89
4.2 Experimental Environment 90
4.2.1 Mul-T - a Parallel LISP 90
4.2.2 The MIT Alewife Machine 90
4.2.3 Alewife Machine Model 92
4.3 General Issues in Compiling Static Dataflow Graphs 94
4.4 Parallel Operator Library 96
4.4.1 Matrix Sums 96
4.4.2 Matrix Products 96
4.4.3 Other Operators 98
4.5 Composing Parallel Operator Library Routines 98
4.6 Compiler Implementation Details 102
4.7 Matrix Expression Examples 102
4.7.1 Matrix Product Parallel Library routine 102
4.7.2 g11 - Filter Bank 106
4.7.3 g12 - Larger Filter Bank 110
4.7.4 g20, g21 - Matrix Polynomials 110
4.7.5 g22 - DFP Update 113
4.8 Discussion and Further Compiler Enhancements 113

5 Conclusion and Future Work **117**
5.1 Summary 117
5.2 Contributions of the Thesis 121
5.3 Extension and Future Work 122

A Optimal Solution for Strictly Increasing Speedups **125**

B Optimal Solution for p^α Dynamics **129**

C Proof of Homogeneity Theorem **133**

List of Figures

1.1 Hierarchical Compilation Paradigm 13
1.2 Processor Model 14
1.3 Generalized Scheduling 17

2.1 Processor Model 27
2.2 Dataflow Graphs for Matrix Addition and Matrix Product 30
2.3 Dataflow Graph for $N_1 \times N_2$ Matrix Addition 33
2.4 Dataflow Graph for $N_1 \times N_2 \times N_3$ Matrix Product 34
2.5 Processor Cluster Analysis for Matrix Product 36
2.6 Optimal Processor Cluster 39
2.7 Optimal Square Partitioning 42
2.8 Optimal Tiling solution compared with Kong's Approximation 44
2.9 Partition of an $N \times N \times N$ cube into 14 chunks (a) 48
2.10 Partition of an $N \times N \times N$ cube into 14 chunks (b) 49
2.11 Communication vs Processors for $20 \times 20 \times 20$ Matrix Product 51
2.12 Communication vs $T_{compute}^{(-1/3)}$ for $20 \times 20 \times 20$ Matrix Product 52
2.13 Communication vs Processors for $10 \times 40 \times 20$ Matrix Product 54
2.14 Communication vs $T_{compute}^{(-1/3)}$ for $10 \times 40 \times 20$ Matrix Product 55
2.15 Mode-Shifts for $1 \leq P(= 1 + \epsilon) \leq 2$. 58
2.16 Optimal partition of AA^T 59
2.17 Optimally partitioned dataflow graph of AA^T, $P = 2$ blocks 60

3.1 Example of a parallel task system 66
3.2 Optimal Tree Scheduling using series-parallel reductions 82
3.3 Optimal Inverted Tree Scheduling using series-parallel reductions 83
3.4 Optimal DAG Scheduling using series-parallel reductions 84
3.5 Scheduling and Shortest Path Problems 85

4.1 The MIT Alewife Machine 91
4.2 Implementation of Matrix Product 97
4.3 Generalised Scheduling Heuristics 101
4.4 Code for Matrix Product 103
4.5 Speedup Curve for 20×20 matrix product 105
4.6 Greedy Schedule for $g11$ 106

4.7 Code for $g11$ 108
4.8 Speedup Curves - g11 109
4.9 Speedup Curves - g12 111
4.10 Speedup Curves - g20, g21 112
4.11 DFP Update 113
4.12 Speedup Curves - g22 114

Chapter 1

Structure Driven Compilation of Numeric Problems

1.1 Overview

Numeric computation is critical to many branches of science and engineering. Applications are found in electrical engineering (signal processing, VLSI design, circuit and device simulation), mechanical and civil engineering (mechanical parts design, finite element analysis), weather prediction, simulation studies in the physical sciences, etc. These problems are compute intensive, and hence their efficient multiprocessor execution is of great current interest.

Efficient multiprocessor performance on a given problem depends on many interacting factors, including architecture, operating systems, and efficient program compilation. In this thesis, we explore the issues of effectively compiling numeric problems onto shared memory multiprocessors.

1.2 Thesis Outline

Two issues have to be tackled for efficient multiprocessor compilation. First, the workload of the program must be equitably distributed across all the available processors. The distribution must be such that the communication between tasks is minimised. This is the *partitioning* problem. Next, the set of tasks for each processor must be sequenced in in a manner such that all precedence constraints are satisfied, and the processors are kept busy as far as possible. This is the *scheduling* problem. Both partitioning and scheduling are difficult, NP-Hard problems.

Clearly, general purpose graph partitioning and scheduling techniques can be used for compilation [Sar87], but they are very time consuming in general. However, for many important classes of numeric problems, the general techniques are unnecessary. Hierarchical techniques can be used to greatly speed up the compilation.

In this thesis, we explore the application of a *hierarchical compilation* strategy, to an important subclass of numeric problems, viz. those encountered in signal processing and linear algebra. The architectures targeted are shared memory architectures like the MIT Alewife machine [Aga90], the Encore Multimax, etc.

The basic paradigm is to exploit the structure in the dataflow graphs (also called DFG's) of such numeric problems. These numeric problems can be conveniently represented as compositions of basic numeric operators. If good compilations for the basic operators are known, and good techniques to compose these operator algorithms, then a good compilation for the numeric problem can be derived. The strategy will be fast if each of the two steps is fast.

The hierarchical compilation strategy is applicable if the problem domain naturally lends itself to the two level specification. Many problems in signal processing and linear algebra fall in this class. Portions of a general purpose numeric program are also well handled by this paradigm. A combination of the two-level hierarchical compilation method, and the general purpose method ([Sar87]) is necessary for a complete general purpose numeric program.

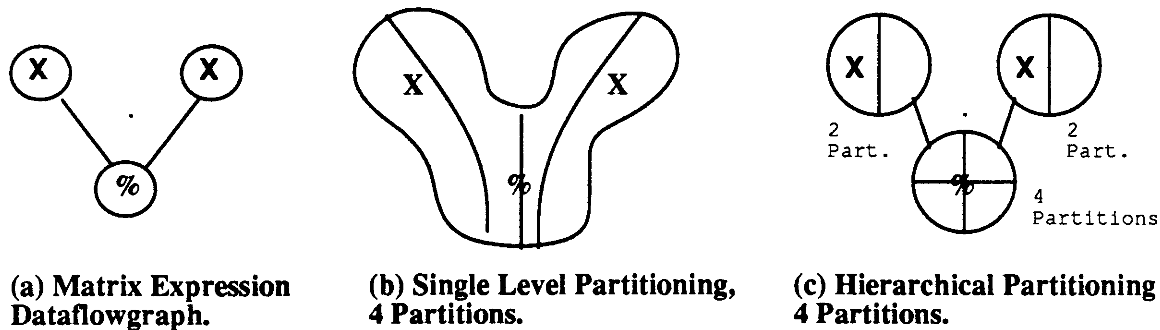
Most algorithms in signal processing and inner loops of linear algebra algorithms can be expressed in terms of expressions composed of matrix operators. These matrix operator dataflow graphs have regular data and control flow, and regular communication structure. The nodes of the matrix expression dataflow graph are matrix operators themselves (macro-nodes). The matrix expression dataflow graph (also called a macro dataflow graph) in general exhibits little structure, but generally has data-independent control. Some examples are shown below (all operators are matrix operators)

$$Y = A(B + CD) \qquad \text{Simple Matrix Expression}$$
$$Y = a_0 + a_1 A + a_2 A^2 + a_3 A^3 + \cdots + a_N A^N \qquad \text{Matrix Polynomial}$$
$$Y = WX \qquad \text{Fourier Transform, matrix W } w_{kl} = e^{-j\frac{2\pi kl}{N}}$$

This thesis has two major parts. First, it shows that it is possible to analyse and derive good algorithms for the basic matrix operators, thus deriving a parallel operator library. Second it shows that one can quickly compose library routines to get good algorithms for the complete matrix expression. This yields a speedy hierarchical compilation strategy for such structured problems. Specifically,

- We demonstrate how the structure present in matrix operator dataflow graphs can be used to derive close to optimal routines for them. Our techniques can be used to develop a parallel library of routines for these operators. It is important to note that our library routines are *parameterised* by the number of processors to run them on. In other words, each library routine is characterised by the amount of parallelism involved in its computation.

- We have developed theoretical insights into effectively composing matrix operator algorithms to yield algorithms for the matrix expression. These insights have been obtained by viewing scheduling from the perspective of optimal control theory.



(a) Matrix Expression Dataflowgraph. **(b) Single Level Partitioning, 4 Partitions.** **(c) Hierarchical Partitioning 4 Partitions.**

Figure 1.1: Hierarchical Compilation Paradigm

Minimal time scheduling strategies can be identified with time optimal control strategies. The approach has been used to derive simple heuristics for composing matrix operator algorithms to form algorithms for the expression.

- We have implemented these ideas in the form of a prototype compiler producing Multilisp code from a matrix expression in Lisp-like syntax. Timings statistics on the MIT Alewife Machine have been obtained, roughly verifying our ideas.

An example of the hierarchical compilation paradigm is shown in Figure 1.1. Figure 1.1 (a) shows the dataflow graph for a sequence of three matrix operations, two multiplies followed by an inverse. Figure 1.1 (b) shows a conventional compilation algorithm, which expands the dataflow graph (partially or completely), and then performs a partitioning and scheduling (on four processors). Figure 1.1 (c) shows the hierarchical paradigm, where parallel library routines for each of the operators (two matrix products and an inverse) are *composed* together to form a compilation for the complete expression. While tackling the composition problem, we have to determine both the sequencing of the library routines, as well as the number of processors each routine runs on (parallelism). We have developed a theoretical foundation for this problem, using the theory of optimal control. Several heuristics for determining parallelism as well as scheduling emerge from this theory. A prototype compiler incorporating these ideas has been implemented.

The rest of the overview sketches these ideas in more detail. Section 1.3 describes the multiprocessor model used in the thesis. Section 1.4 sketches the techniques used

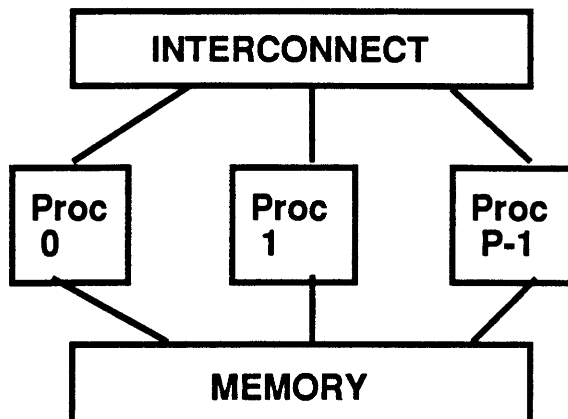


Figure 1.2: Processor Model

to design the parallel library. Chapter 2 describes these techniques in detail. Section 1.5 sketches the methods used to compose the library routines to form a routine for the complete matrix expression. Chapter 3 describes these ideas in detail. Section 1.6 specifies some of the implementation details. Chapter 4 describes these ideas in detail. Section 1.7 gives a historical perspective on the compilation problem. Section 1.8 sums up.

1.3 Multiprocessor Model

We assume a fully connected MIMD multiprocessor with P processors, with global shared memory (Figure 1.2).

The architecture is characterised by the operation times and the times to access data in another processor or in global memory. We shall show (Chapter 2) that the optimal algorithms vary smoothly as these architectural parameters vary, facilitating automatic compilation.

We assume a uniform multiprocessor structure with equal access time to any other processor. The time to access a datum in global memory is also invariant. Hence we have ignored network conflicts, memory interleaving, etc. More details on the multiprocessor model are in Chapter 2.

The MIT Alewife machine [Aga90] on which the simulation is performed is a cache coherent shared memory (2-D or 3-D) mesh connected multiprocessor. While it does not exactly match the processor abstraction, such a simple abstraction is necessary for analysis of the performance of a given algorithm on the machine. We use a simple performance metric for evaluating an algorithm, viz. the sum of communication and computation times. These will be defined precisely in Chapter 2.

1.4 Design of the Parallel Operator Library

We describe below the methodology used in designing the routines in the parallel operator library. The key idea is to exploit the structure of the operator dataflow graph to facilitate partitioning and scheduling. The simple processor abstraction (Section 1.3) facilitates analysing the performance of any given operator algorithm.

It is important to note that in general, each operator routine is parameterised by the input and output data sizes, as well as the number of processors (parallelism) to be used in its computation. These parameters are specified when these routines are composed together to yield a complete matrix expression.

1.4.1 Characterization of basic matrix operators

The basic matrix operators have dataflow graphs which can be *represented* as simple regular geometric figures (regular polyhedra) (Chapter 2). Their nodes are arranged at lattice points, and communication takes place in a regular pattern. This regular geometry makes the partitioning and scheduling task easy. Thus good algorithms for these basic numeric operators can be easily derived.

1.4.2 Partitioning

Optimal partitioning of the operator dataflow graph for P processors is equivalent to splitting up the regular geometric figure formed by the dataflow graph so as to minimise

the communication, keeping the computation balanced. In general, this necessitates packing P equal sized compact node clusters into the regular geometric figure formed by the dataflow graph. This is a difficult problem in general, for arbitrary P. But the redeeming feature is that this partitioning problem needs to be solved only once for each operator, and put in the compiler's knowledge base. It is important to note that the optimal partitioning techniques can be devised to handle an arbitrary number of partitions (parallelism). Thus one does not have to solve the partitioning problem repeatedly for every possible parallelism.

1.4.3 Scheduling

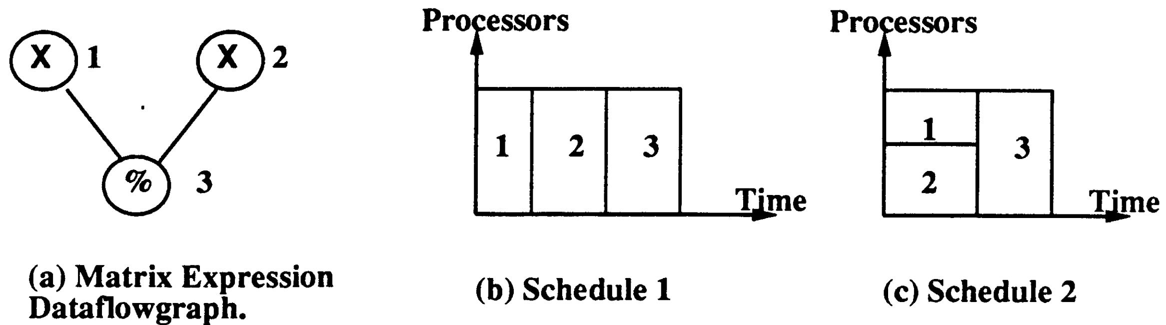
Scheduling the above formed partitions is easily done by analysing the regular precedence structure of the dataflow graph. In many cases, the computations at all nodes are independent, so scheduling is very straightforward. Of course, the extensive bookkeeping needed in all cases is best handled by an automatic compiler.

1.4.4 Examples

Chapter 2 shows how the structure present in many matrix operator dataflow graphs can be exploited to develop a parallel library. Each library routine is parameterised by the data sizes, the amount of parallelism (P), etc.

For example, in the case of a matrix product, the dataflow graph can be represented as a cube. Optimal partitioning of the dataflow graph into P partitions is equivalent to packing P compact partitions into a cube. The scheduling of the various partitions is straightforward because all nodes in the dataflow graph are independent. Similarly, the dataflow graph for a Fourier Transform forms a hypercube. Optimal partitioning is equivalent to identifying small sub-FFT's. The scheduling follows FFT precedences.

Thus, by analysing the geometric structure of the dataflow graph, we can generate a library of parallel routines, each parameterised by the amount of parallelism P.



(a) Matrix Expression Dataflowgraph.

(b) Schedule 1

(c) Schedule 2

Figure 1.3: Generalized Scheduling

1.5 Composing Parallel Operator Library routines

1.5.1 Characterization of the problem

Given the parallel operator library, we have to compose the operator routines to generate a complete algorithm (compilation) for the numeric problem. Note that the modules in the operator library are parameterised by the (apriori unknown) number of processors (parallelism) P. Thus the composition has to specify P for each operator, in addition to sequencing the operators. Different assignment of processors P will result in differing performances.

For example, Figure 1.3 shows two different schedules for a matrix expression composed of two matrix multiplies followed by an inverse. We have specified each schedule in terms of its Gantt chart. A Gantt chart shows the computations performed by each processor (y-axis) over time (x-axis).

From the figure, we see that in Schedule 1, tasks (operator routines) 1 and 2 run on more processors than in Schedule 2. Thus the overhead of parallel execution has to be higher in Schedule 1. Thus Schedule 2 is expected to be faster than Schedule 1. Hence an optimal operator parallelism has to be determined in addition to sequencing. Clearly, this problem (to be called *generalized scheduling*) is more difficult than classical scheduling, where only the sequencing is determined.

1.5.2 Formal Specification

In this section, we present a formal specification of generalised scheduling. We must note that the specification is simplified somewhat - a much more sophisticated model can be found in Chapter 3. However, most of the essential aspects are presented below.

We have a system of N tasks i, with a set of precedence constraints

$$\Phi = \{(i < j)\}$$

where $i < j$ implies that task i precedes j.

The execution time for each task varies with the number of processors allocated to it. Let the task i have an execution time $e_i(p_i)$ on p_i processors. In general, as we increase the number of processors allotted to any task, the overhead of parallel execution increases. For a well designed operator routine, with low overhead, the time can decrease almost linearly with the number of processors. For a poor routine, the time need not decrease significantly even for many processors. Hence, in general it is true that

$$e_i(p_i) > \frac{e_i(1)}{p_i}$$

The generalised scheduling problem is to allocate processors to each of the tasks, and sequence the tasks (ie. determine their starting times) in such a way that all precedence constraints are satisfied, and the complete matrix expression is computed in minimum time.

A generalised schedule S can hence be specified as a set

$$\{< i, p_i, t_i >\}, i = 1 \cdots N$$

where p_i is the number of processors allocated to the task i, and t_i is the starting time of task i. The precedence relations imply that

$$i < j \rightarrow t_i + e_i(p_i) \leq t_j$$

At any point of time t, the total number of processors allocated to all the running tasks

is less than the total number of processors available, P, ie.

$$\sum_{i \in \text{running tasks}} p_i \leq P$$

The objective is to minimise the finishing time t_F, where

$$t_F = \max_i \; (t_i + e_i(p_i))$$

It is conceivable that a schedule, in which the number of processors allocated to a task changes dynamically with time, is faster than a schedule in which it does not. Hence, in our model in Chapter 3, we have allowed the processors p_i allocated to a task i to be time varying, ie. $p_i = p_i(t)$.

It is relatively easy to think up simple heuristics to perform generalised scheduling. For example, one should as far as possible minimise the number of processors assigned to each task, since then one minimises the overhead. But one can explore this problem at greater length, and derive very interesting insights into the problem of multiprocessor scheduling itself. For this we take recourse to optimal control theory.

1.5.3 Control Theoretic Formulation of Scheduling

The fundamental paradigm is to view tasks as dynamic systems, whose state represents the amount of computation completed at any point of time. The matrix expression is then viewed as a composite task system - the operator routine tasks being its subsystems.

At each instant, state changes can be brought about by assigning (possibly varying) amounts of processing power to the tasks. Computing the composite system of tasks is equivalent to traversing a trajectory of the task system from the initial (all zero) uncomputed state to the final fully computed state, satisfying constraints on precedence, and total processing power available. The processors have to be allocated to the tasks in such a way that the computation is finished in the minimum time.

This is a classical optimal control problem. The task system has to be controlled to traverse the trajectory from start to finish. The resources available to achieve this control

are the processors. A valid control strategy never uses more processors than available, and ensures that no task is started before its predecessors are completed. A minimal time schedule is equivalent to a time-optimal control strategy (optimal processor-assignment).

1.5.4 Results from Control Theory

The results of time-optimal control theory (Chapter 3) can be invoked to yield fundamental insights into generalised scheduling. The results include

- Powerful general theorems regarding task starting and finishing times.
- Elegant rules for simplifying the scheduling problem in special cases. Equivalence of the generalised scheduling problem to constrained shortest path problems in such cases.
- General purpose heuristics for scheduling, provably optimal in special cases.

1.6 Implementation and Experimental Validation

We have written a prototype compiler which takes a matrix expression and automatically generates parallelised Multilisp code. The input language allows arbitrary matrix expressions to be specified, both trees and general DAGS. The compiler operates by first calling the generalised scheduling heuristics (Section 1.5.3, and Chapter 3) to determine the parallelism for each particular operator. The object code is formed by appropriately sequencing the calls to the library routines for each operator, with the parallelism specified above. Intermediate memory allocation is currently done within the library routines themselves. The results from the prototype compiler are in rough verification with theory, with correct and relatively efficient code being produced. A much more powerful compiler is currently in development, and should show even better results.

1.7 History of the Problem

Multiprocessor compilation has been extensively researched to date [PW86, Cof76, Kuc78, Sar87, Sch85]. The research can be grouped into two categories.

The first category attempts to compile a given, fixed dataflow graph onto a given, possibly parameterised architecture. The second category attempts to perform algorithmic changes to the dataflow graph itself, to improve performance.

Various techniques have been developed for compiling fixed dataflow graphs onto a given, possibly parameterised architecture. Firstly, techniques have been developed for vectorising and parallelising FORTRAN programs onto vector and parallel machines [Kuc78]. In conjunction with this, several code optimisation techniques for uniprocessors were also developed. Second, for the subclass of programs whose dataflow graphs are regular, retiming techniques [Lei83] exist for efficiently mapping them onto regular processor architectures. Lastly, general purpose graph partitioning and scheduling techniques have been applied [Sar87] to compile general programs onto a wide class of parameterised architectures.

These compilation techniques are not hierarchical in general. The techniques for parallelising FORTRAN [Kuc78] are primarily local, low level optimizations. The prime examples of local optimizations are the techniques to vectorise and parallelise loops. The retiming techniques [Lei83] can be viewed as methods to develop the parallel library routines, since the dataflow graphs for the library routines are regular. The general purpose partitioning and scheduling techniques [Sar87] do not have a notion of hierarchy, as the entire program dataflow graph is treated as a unit. The hierarchical compilation paradigm explored in this thesis builds on these low level optimization techniques, and incorporates composition paradigms on top. Below we describe some of the historical work in more detail.

The early studies of [Kuc78] dealt with various issues in automatic compilation, both on single processor machines, and multiprocessors. The goal was to effectively compile FORTRAN programs onto vector machines like the Cray. Both scalar and vector/parallel

optimizations were developed.

The scalar optimizations include constant and copy propagation, induction variable recognition, code reordering, etc. The parallelising optimizations dealt with various kinds of arithmetic expressions, especially recurrences. Lower bounds of various kinds on the parallel time taken to compute arithmetic expressions exploiting all allowed commutativity and associativity were derived. Many methods to vectorise and parallelise loop recurrences on vector machines were developed. Generally all these methods involved analysis of the dependency structure of the recurrence, to find an appropriate unrolling and reordering. Various tests [Ban79] to determine when loop unrolling is feasible were developed. Many of these techniques have been applied to compiling FORTRAN programs for vector and parallel machines like the Cray [PW86], Convex [Mer88], Ardent [All88], etc. Related work in the context of VLIW architectures was done by [Fis84], and applied in the Bulldog compiler.

A very general approach to the multiprocessor compilation problem was taken by ([Sar87]), to compile programs written in a single assignment language SISAL. A very large class of parameterised architectures, with varying processor and interconnect characteristics can be handled.

A dataflow graph is created for the problem, with each node representing a collection (possibly only one) of operations in the program. Execution profile information is used to derive compile time estimates of module execution times and data sizes in the program. Then, an explicit graph partitioning of the dataflow graph of the problem is performed, to determine the tasks for different processors. Finally, either a run-time scheduling system is invoked to automatically schedule the tasks, or a static scheduling of these tasks is determined at compile time. Trace driven simulation was used to obtain experimental results. Speedups of the order of 20 % to 50 % were observed for important numeric programs like SIMPLE, SLAB, FFT, etc.

These graph partitioning and scheduling techniques are time consuming in general, as they involve the NP-hard problems of partitioning and scheduling. We show in our thesis

that for a large class of important numeric programs, the general purpose partitioning and scheduling methods are unnecessary, and can be replaced by simpler hierarchical compilation techniques.

[Lei83] and others have developed *retiming* techniques for effectively scheduling problems with regular dataflow graphs - eg. matrix operators. These dataflow graphs are characterised by the existence of a large number of similar nodes, arranged in a multidimensional lattice. All nodes perform the same operation, on different data. Communication of data between nodes is regular, and generally between adjacent nodes. Flow of control at each node is relatively data independent.

The retiming technique maps the lattice of dataflow graph nodes into a lower dimensional processor lattice (a lattice whose points represent processors). A linear modulo map is generally used. In general, multiple dataflow graph nodes will map to the same processor. Hence this implicitly yields a partition of the dataflow graph nodes into sets handled by different processors. The communication structure of the dataflow graph lattice determines the communication structure of the processor lattice. Retiming techniques in general result in systolic computation methods for the dataflow graph.

In the domain of signal processing [BS83, Sch85] have extended such retiming techniques for effective compilation of the recurrences commonly found in signal processing systems. The class of cyclo-static periodic schedules was developed. Basically, these schedules *tile* the dataflow graph lattice in an optimal fashion. A basic tiling pattern specifies the distribution of computation of a single iteration across all the processors. The distribution of work changes periodically with time, as specified by the lattice vector of the schedule. These schedules exhibit various forms of optimality - with respect to throughput, delay, and number of processors.

Retiming techniques appear quite attractive for compiling regular dataflow graphs. Indeed compilers for systolic architectures have been developed. For example, [Lam87] has developed an optimising compiler for the 10 processor Warp Machine. The Warp Machine is a linear systolic array of high performance programmable processors. *Soft-*

ware pipelining and *hierarchical reduction* (of basic blocks to a macro-instruction) are extensively used for code optimization.

However, in general it is not possible to map arbitrary compositions of regular dataflow graphs using retiming techniques. Even for regular dataflow graphs, it is not easy to find linear modulo maps to map arbitrary sized problems on arbitrary number of processors. The techniques developed so far do not incorporate prespecified processor interconnectivity, except in the CRYSTAL compiler, [Che85, Che86]. Also, important code reordering techniques exploiting arithmetic properties like associativity and commutativity are not incorporated in this framework.

The work described above tackles the problem of compiling a given dataflow graph onto a given architecture, and as such falls in the first category of compilation techniques. Further major gains can be obtained by changing the dataflow graph itself (algorithm restructuring) - the second category. In the context of matrix expressions, this would mean transforming the matrix expression into equivalent forms, using the rules of matrix algebra.

[Mye86, Cov89] at the MIT DSP group have explored algorithm restructuring in the domain of signal processing. SPLICE and ESPLICE [Mye86, Cov89] apply a variety of knowledge based methods to transform signal processing systems into equivalent forms, with varying computational cost.

SPLICE and ESPLICE provide symbolic signal/system representation and manipulation methods to facilitate the design and analysis of signal processing systems. Signal and System objects can be defined in an implementation independent manner. Various properties like extent, symmetry, periodicity, real/complex, etc. can be associated with signals. Similarly, properties like additivity, homogeneity, linearity, shift-invariance, computational costs, etc. can be defined for systems.

ESPLICE provides rules for the derivation of properties of a composite signal/system. Of direct relevance here are rules for the derivation of equivalent forms (simplification, rearrangements, etc) of signal processing expressions. As of now, this portion of ESPLICE

is very crude. Essentially enumerative derivation strategies are used, without much search guidance. A very large number of equivalent forms of a given signal processing expression are generated, and those with low computational costs selected. Unfortunately, SPLICE and ESPLICE do not have realistic architectural knowledge built into them, and neither are the computational cost metrics suitable for multiprocessors. Hence an equivalent form deemed good according to ESPLICE may not necessarily be good for a multiprocessor. However, ESPLICE can in principle be modified to do this job.

MACSYMA [Mat83] is a system for symbolically differentiating and integrating algebraic expressions, factoring polynomials, etc. It has the capability of performing matrix manipulations algebraically, and thus would be of use in transforming matrix expressions into each other. Unfortunately, there is no notion of architectures, nor cost criteria.

1.8 Summary

In this thesis we will explore the hierarchical compilation paradigm in the context of matrix expressions.

This thesis shows that it is possible to derive good algorithms for the basic matrix operators (matrix sums and products), and quickly compose them to get good algorithms for the complete expression. This yields a speedy and efficient hierarchical compilation strategy for such structured problems.

We will develop theoretical insights into effectively composing matrix operator algorithms to yield algorithms for the complete matrix expression.

We will demonstrate simple experimental verification of our ideas, using a prototype compiler producing Multilisp code for a matrix expression in Lisp like syntax.

Chapter 2

Optimal Matrix Operator Routines

2.1 Overview

In this chapter we shall discuss creating a parallel library for two basic matrix operations, matrix addition and multiplication. Composing together these library routines to generate a good compilation for the complete dataflow graph will be dealt with in succeeding chapters. The basic idea is to exploit the structure of the operator dataflow graph, to devise a good partitioning and scheduling technique.

Before we discuss the partitioning and scheduling technique, we have to discuss the multiprocessor model used in some detail.

2.2 Multiprocessor model

We assume a fully connected multiprocessor with P processors, with global shared memory (Figure 2.1).

Additions take time T_a, and multiplies take time T_m. The access time for data from another processor is T_s and that from global memory is T_f. Since an interprocessor data transfer can be performed via main memory also, we must have $T_s \leq 2T_f$. We assume that upto P such accesses can occur simultaneously, so the network bandwidth is P

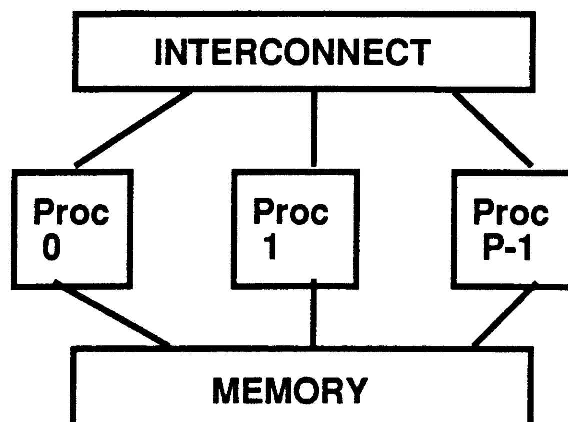


Figure 2.1: Processor Model

accesses per cycle. We shall show that the optimal algorithms vary smoothly as these architectural parameters vary, facilitating automatic compilation.

We have assumed a uniform multiprocessor structure with equal access time to any other processor. The time to access a datum in global memory is also invariant. Hence we have ignored network conflicts, memory interleaving, etc.. We have grouped the local memory access time into the compute time.

2.3 Optimal compilation

Optimal compilation of a dataflow graph means determining a sequence of operations for each processor (schedule), which minimises the execution time. Accurately determining the execution time for a schedule is a difficult problem. We approximate the execution time as follows.

Assume that N_a additions and N_m multiplies are performed by the most heavily loaded processor, with N_f memory accesses and N_s interprocessor data transfers *in all.* The total execution time T_e can be approximated by the sum of the computation time $T_{compute}$ and communication time T_{comm}. We are ignoring possibilities of overlapping communication and computation, as well as data dependencies.

$$T_e = T_{compute} + T_{comm}$$

The compute time can be expressed as

$$T_{compute} = T_a N_a + T_m N_m$$

The communication time is difficult to estimate accurately. We approximate it by the total communication load N_T divided by the total number of processors, assuming P transfers can be scheduled on each clock cycle. The total communication load is the weighted sum of the number of memory accesses and the number of interprocessor transfers,

$$N_T = N_f T_f + N_s T_s$$

Hence

$$T_{comm} = \frac{N_T}{P} = \frac{N_f T_f + N_s T_s}{P}$$

and

$$T_e = T_{compute} + T_{comm} = T_a N_a + T_m N_m + \frac{N_f T_f + N_s T_s}{P} \tag{2.1}$$

This equation will be extensively used.

We shall see below that optimal compilation on P processors means trying to partition up the dataflow graph into P roughly equal sized chunks, while choosing the shape of the chunks so as to minimise the communication. Keeping the chunks equal in size minimises $T_{compute}$, while appropriately shaped chunks minimise T_{comm}.

Clearly, the location of the input and output data elements (matrices) influences the memory accesses, and hence the optimal algorithm. In all that follows, we shall by default assume that the *inputs are located in global memory* to begin with, and the *outputs are assumed to be finally dumped in global memory* also.

2.4 Exploiting Structure in Matrix Operator Dataflow Graphs

We shall illustrate below how the structure in matrix operator dataflow graphs can be exploited to generate good parallel library modules. We stress that the specific algorithms

derived are not crucial. The method of partitioning, viz. viewing the dataflow graph as a regular geometric polyhedral figure to be partitioned up into compact chunks, is crucial (see below). In general, other operator specific information besides the geometric structure of the dataflow graph can also be used. For example, in the case of the Fourier Transform, knowledge of the row-column algorithm (derived from Thompson's analysis) is essential to obtain the optimal algorithms. An ordinary compiler, using general purpose partitioning and scheduling heuristics, will not be able to guess at these structural properties and will produce suboptimal algorithms.

2.4.1 Polyhedral Representation of Matrix Operator Dataflow Graphs

We need to represent the matrix operator dataflow graph in a manner such that the regular polyhedral nature is evident. This can be achieved by representing the dataflow graph as a lattice, with each dataflow graph node corresponding to some lattice point. The locality behaviour of the dataflow graph is reflected in the geometric locality of the lattice points. Nodes corresponding to adjacent lattice points generally have some common inputs or contribute to common outputs. We illustrate the representation using matrix multiplication as an example.

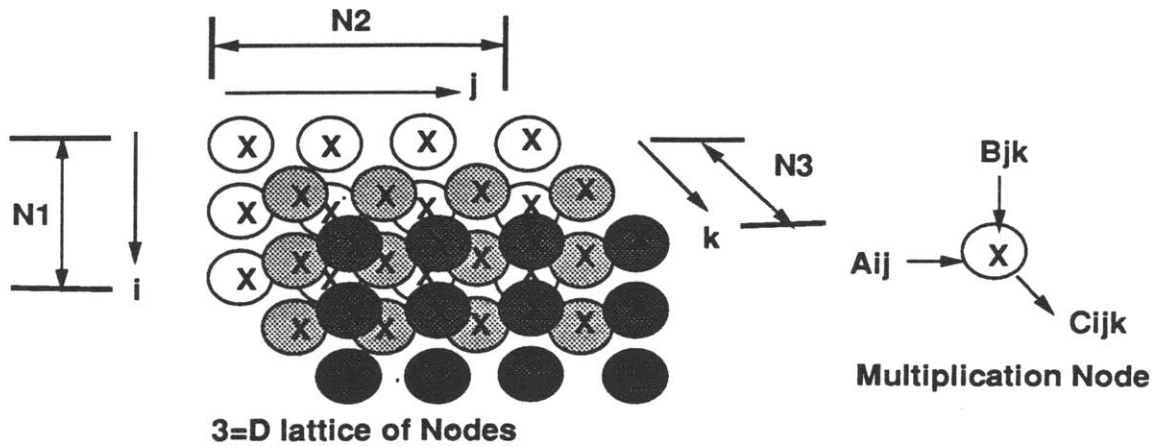
The standard algorithm for an $N_1 \times N_2 \times N_3$ matrix multiply

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

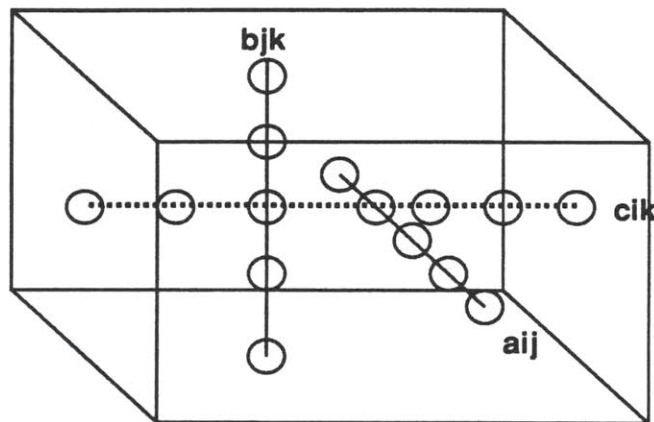
has $N_1 N_2 N_3$ multiplications, and $N_1 N_3 (N_2 - 1)$ additions. The corresponding dataflow graph can be represented by an $N_1 \times N_2 \times N_3$ lattice of multiply-add nodes (Figure 2.2 (a)). Each node (i, j, k) represents the computation

$$c_{ik} \leftarrow c_{ik} + a_{ij} b_{jk}$$

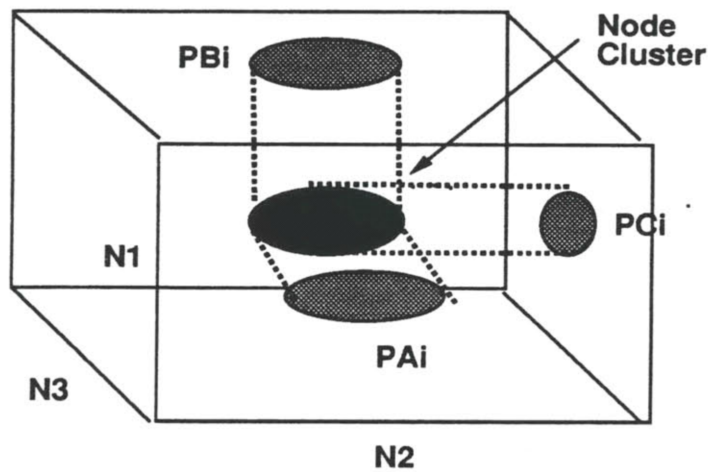
An element of A, a_{ij} is broadcast to the N multiply-add nodes having the same value of ij. These nodes are arranged in a line parallel to k axis. Thus the computation on



(a) Dataflowgraph lattice for matrix product



(b) Polyhedral representation of Dataflowgraph



(c) Dataflowgraph locality and geometric locality

Figure 2.2: Dataflow Graphs for Matrix Addition and Matrix Product

all nodes in this line exhibits *locality* with respect to the element a_{ij}. This broadcast of a_{ij} is represented by a solid line in Figure 2.2 (b). Similarly, nodes having the same value of jk share b_{jk}. This broadcast of b_{jk} is also represented by the solid vertical line in Figure 2.2 (b). Nodes having the same value of ik all sum together to yield the same output element of C, c_{ik}. This accumulation is denoted by the dotted horizontal line in Figure 2.2 (b). Note that the existence of associativity and commutativity means that the partial products can be summed in any order whatsoever. Thus the accumulates do not impose any precedence constraints. Clearly, in the diagram, nodes connected by a solid line share input matrix elements, and those connected by dotted lines contribute to output matrix elements.

It is important to note that the (solid or dotted) lines representing (broadcast or accumulation) locality do not change if two parallel planes of the lattices are interchanged. This corresponds to various permutations of the computation, possibly exploiting associativity or commutativity. For example, interchanging two planes parallel to the B face (ie. planes which differ in the index i) corresponds to permuting the rows of A and C. Similarly, interchanging planes parallel to the A face corresponds to permuting the columns of B and C. Interchanging planes parallel to the C face corresponds to permuting the columns of A, the rows of B, and adding the partial products $a_{ij}b_{jk}$ in a different manner, exploiting associativity and commutativity. Thus every possibly ordering of the computation can be represented by the geometric lattice.

Now, consider the cluster of nodes shown in (Figure 2.2 (c)). The total number of input elements of A (B) accessed from global memory by this cluster can be measured by the projected area of the cluster PA_i (PB_i) on the A (B) face of the dataflow graph. Thus PA_i and PB_i measure the total number of memory accesses due to this cluster. Similarly, the total number of output elements of C the cluster contributes to is measured by the projection on the C face of the dataflow graph (PC_i). Partial sums for each element in PC_i are formed inside this cluster, and shipped to possibly another processor for further accumulation. Thus PC_i measures the interprocessor data transfer due to this cluster.

Hence the communication of this cluster with the outside world can be minimised (locality maximised) by minimising the projected surface area on the three faces, which can be handled by geometry techniques. Hence a cluster which exhibits geometric locality (in terms of minimal projected surface area), exhibits dataflow graph locality. Thus the locality behaviour of the dataflow graph is reflected in the geometric locality of the lattice.

To summarise, the dataflow graph of a matrix multiply can be represented by a 3-dimensional polyhedral lattice, each lattice point representing a computation in the dataflow graph. Adjacent lattice points share some common broadcast inputs or are accumulated to common outputs. Dataflow Graph locality is equivalent to geometric locality.

We next demonstrate how this geometric dataflow graph representation can be used to simplify the partitioning and scheduling.

2.5 Matrix Sums

Optimal compilation of matrix sums is very easy. Assume we have to compute the sum $C = A + B$ on P processors, where all matrices are of size $N_1 \times N_2$. The dataflow graph for the matrix addition (Figure 2.3) can be represented as an $N_1 \times N_2$ lattice of addition nodes. Each node (i, j) represents the computation

$$c_{ij} = a_{ij} + b_{ij}$$

Clearly, each element of A and B is used exactly once in computing C, and there is no sharing. However, in this case, dataflow graph locality can be interpreted as locality in element indices. Nodes adjacent in the i or j dimensions access adjacent elements of A, B, and contribute to adjacent elements of C.

Each output datum c_{ij} is computed by reading the elements a_{ij} and b_{ij} from global memory, adding, and writing back the result c_{ij} to global memory. Hence two reads and a write are necessary per output datum, yielding a total communication of $3N_1N_2$

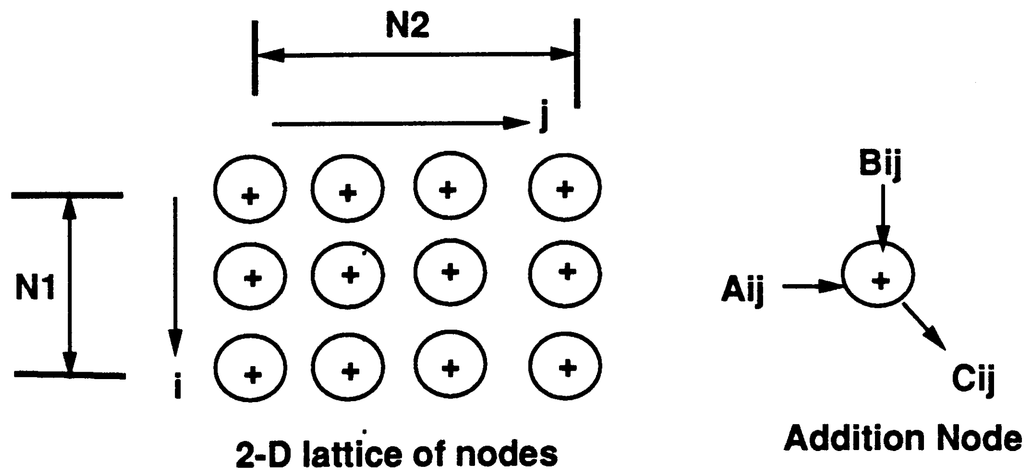


Figure 2.3: Dataflow Graph for $N_1 \times N_2$ Matrix Addition

elements, all of which are memory accesses. Here we assume that the matrices A and B do not share common elements.

This communication is readily achieved by chunking up the dataflow graph into P (equal) chunks. The shape of the chunks can be arbitrary. In practice it is preferable to use very simple chunk shapes, for reducing indexing overhead. For example, the rows can be divided into P equal sized groups, with each processor getting one group.

2.6 Matrix Products

Optimally compiling matrix-products and matrix-inverses is much more difficult, as data is extensively reused in the former case, and the dataflow graph exhibits varying parallelism in the latter. A good partition of the dataflow graph would permit extensive data reuse by each processor. We shall discuss the matrix-product in the rest of this chapter.

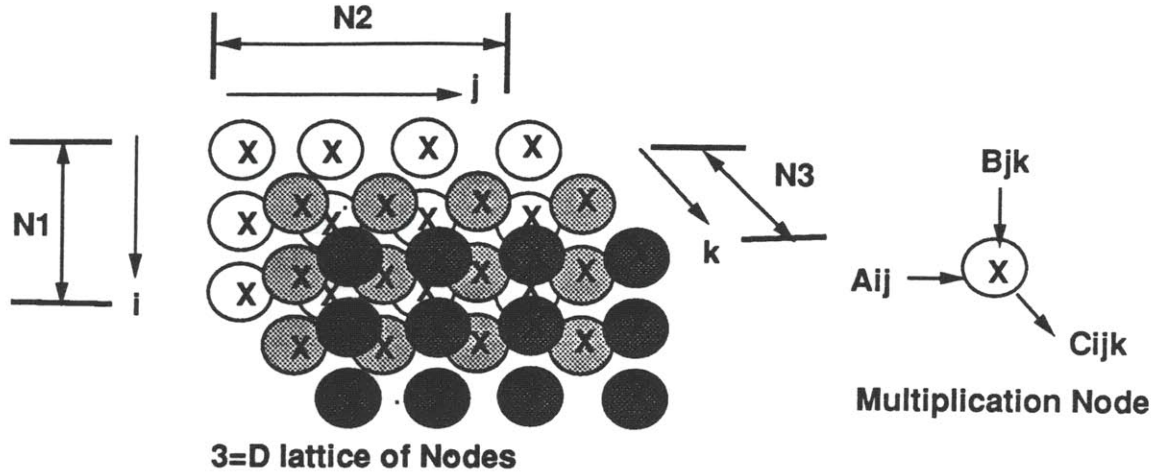


Figure 2.4: Dataflow Graph for $N_1 \times N_2 \times N_3$ Matrix Product

2.6.1 Continuous Approximation

The dataflow graph for the standard algorithm for multiplying an $N_1 \times N_2$ matrix, A, by an $N_2 \times N_3$ matrix, B,

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

is a $N_1 \times N_2 \times N_3$ lattice of multiply-accumulate nodes (Figure 2.4), as per Section 2.4. Node (i, j, k) is responsible for computing $a_{ij} b_{jk}$. Elements of A are broadcast along one axial direction (N_3 or k), B along a second (N_1 or i), and the result, $C = A \times B$, accumulated along the third (N_2 or j).

Optimal compilation of this graph on P processors involves partitioning the lattice into P roughly equal sized chunks, while minimising total communication. Once a partitioning has been performed, the scheduling is simple, because all computations except additions can be carried out independently. However, associativity and commutativity remove any precedence constraints due to ordering that the additions may impose. The scheduling is thus a trivial problem - the computations can be carried out in any order whatsoever.

We show below that determining the optimal partition is equivalent to partitioning the lattice into P roughly cubical equal sized chunks, which is a special case of bin

packing. The discreteness of the lattice makes this problem very difficult in general.

Our approach is to approximate the discrete dataflow graph lattice by a continuous $N_1 \times N_2 \times N_3$ cube, and perform partitioning in the continuous domain. The chunks so obtained are mapped to the discrete lattice using simple techniques. This continuous approximation allows us to derive lower bounds on the communication, and develop several heuristics which come close to the bound.

2.6.2 Lower Bounds

Here we derive the lower bound on communication for a variety of sizes of A and B. We shall show that the optimal algorithm and lower bound are parameterised by the architectural parameters (memory access time T_f and data transfer time T_s), and vary in a smooth fashion as these parameters vary.

The lower bounds are derived by analysis of the ideal shape of a cluster of nodes handled by a processor. The continuous approximation to the dataflow graph allows the use of continuous mathematics for the analysis, which is a great simplification.

Consider the cluster of nodes C_i handled by processor P_i, as shown in Figure 2.5. The processor which computes the nodes in this cluster will need some elements of A, some elements of B, and some interprocessor transfers or memory writes of partial sums of C (for completing the sum).

We first calculate the total number of A elements accessed from main memory. This is the projection PA_i of the cluster C_i on the $N_1 \times N_2$ face (the "A" face) of the dataflow graph. Here we are assuming that once an element of A has been read from main memory, it is stored in high speed local memory, and need not be accessed from main memory again. Similarly PB_i, the projection of the cluster C_i on the B face, is the number of elements of B needed.

Estimating the number of data transfers or memory accesses of C is more complex. Any partial sum of C in the cluster C_i either has to be shifted to another processor for further accumulation, or finally written back to main memory. PC_i, the projection on

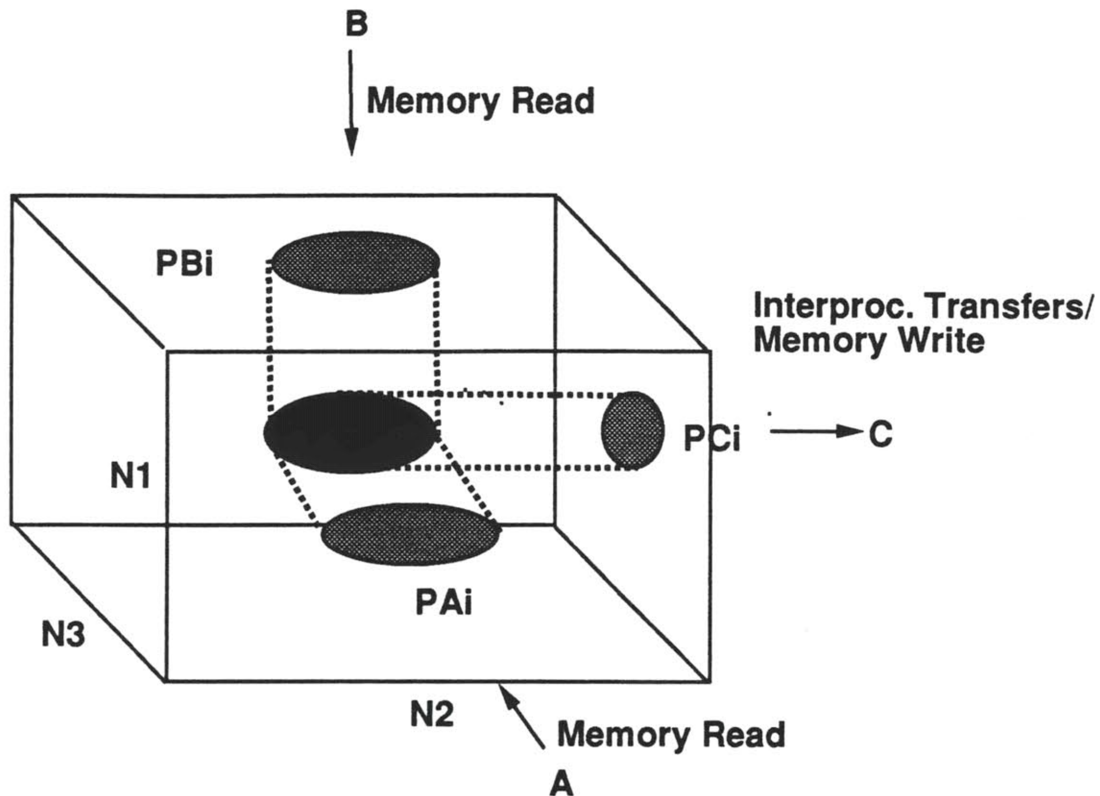


Figure 2.5: Processor Cluster Analysis for Matrix Product

the C face, is the number of partial sums of C being accumulated in this cluster. Thus PC_i measures the total number of data transfers or memory writes of C necessary.

The communication load for this cluster C_i, N_{T_i}, is the sum of loads due to accessing PA_i, PB_i, and PC_i respectively. A case analysis has to be performed depending on whether the time to fetch an item from memory, T_f, is smaller or greater than the time to shift a datum between processors, T_s. Note that by assumption $T_s \leq 2T_f$ (Section 2.2).

If $T_f \leq T_s$, then the fastest strategy is for each processor to read its sections of the A and B matrices, compute the products and partial sums, then shift them to a selected processor which will sum them and write them to global memory. The communication load, N_{T_i}, is the sum of loads PA_i and PB_i, weighted by the memory access time T_f and

the time it takes to move the partial sums for summation or final write back to main memory. This last is PC_iT_s if the partial sums are moved to another processor, and PC_iT_f if the sum is written back to main memory.

Hence, if the partial sums are moved to another processor for final summation,

$$N_{T_i} = T_f(PA_i + PB_i) + T_sPC_i$$

If the ith processor finishes the partial sums, and writes the finished sum back to memory, we have

$$N_{T_i} = T_f(PA_i + PB_i) + T_fPC_i$$

The total communication load N_T is the sum of all N_{T_i}.

$$N_T = T_f \sum_i (PA_i + PB_i) + T_s \sum_i PC_i + (T_f - T_s)N_1N_3 \tag{2.2}$$

If $T_s < T_f$, then the fastest strategy is for processors to share the burden of reading the A and B matrices into local memory once, then shift the various sections to the processors who need them. Products and partial sums are formed, shifted to the processors which will combine them, and then written back to memory. We can amortize the cost of accessing A and B from memory over all P processors.

Hence, if the partial sums are moved to another processor for final summation,

$$N_{T_i} = \frac{T_f}{P}(N_1N_2 + N_2N_3) + T_s(PA_i + PB_i) + T_sPC_i$$

If the ith processor writes the finished sum back to memory, we have

$$N_{T_i} = \frac{T_f}{P}(N_1N_2 + N_2N_3) + T_s(PA_i + PB_i) + T_fPC_i$$

Hence the total communication load is

$$N_T = T_s \sum_i (PA_i + PB_i) + (T_f - T_s)(N_1N_2 + N_2N_3) + T_s \sum_i PC_i + (T_f - T_s)N_1N_3 \tag{2.3}$$

Equations 2.2 and 2.3 can be combined to yield

$$\begin{aligned} N_T &= \min(T_f, T_s) \sum_i (PA_i + PB_i) + \\ &\quad \max(0, (T_f - T_s))(N_1N_2 + N_2N_3) + T_s \sum_i PC_i + (T_f - T_s)N_1N_3 \end{aligned} \tag{2.4}$$

The total communication load N_T can be minimised by a good choice of the projections PA_i, PB_i, and PC_i.

The part of Equation 2.4 that depends on the design of cluster i can be written in a symmetric form as

$$T_{comm_i} = \alpha_3 PA_i + \alpha_1 PB_i + \alpha_2 PC_i \tag{2.5}$$

where

$$\alpha_1 = \alpha_3 = \min(T_f, T_s) \ \alpha_2 = T_s$$

α_i is the cost of transmitting (interprocessor data transfer or memory access) a datum along the axis N_i. This form makes mathematical manipulations easier.

Thus the communication becomes the (weighted) sum of the projections of the cluster C_i on the three faces of the cubical dataflow graph. If $T_f = T_s$, the communication is exactly the projection sum.

Optimal compilation means minimising the total communication, keeping all clusters the same size ($\frac{1}{P}$ (Volume of dataflow graph)). The minimum total communication is attained (if possible), when the communication for each cluster is the minimum possible, ie. each cluster is ideal in shape.

Neglecting the constraint that the dataflow graph box has to be exactly filled by all clusters, an ideal cluster minimises the (weighted) projection sum keeping the cluster volume fixed ($\frac{1}{P}$ (Volume of dataflow graph)). The ideal cluster shape then turns out to be a rectangular block in general, with its aspect ratio depending on T_f and T_s. (Figure 2.6).

Let the sides of the rectangular processor cluster be L_1, L_2, and L_3 (L_j oriented along axis N_j) (Figure 2.6). If all clusters have the same volume V (ideal load balancing), we have

$$V = L_1 L_2 L_3 = \frac{N_1 N_2 N_3}{P}$$

$$PA_i = L_1 L_2 \ , \ PB_i = L_2 L_3 \ , \ PC_i = L_3 L_1$$

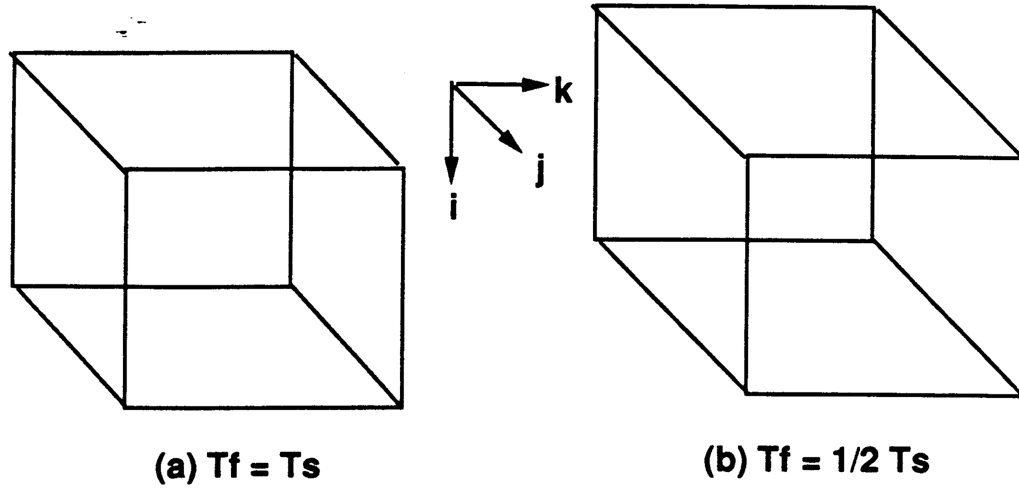


Figure 2.6: Optimal Processor Cluster

Since the ideal cluster has to fit inside the dataflow graph, we have the constraint

$$L_j \leq N_j$$

The communication load for the ith cluster can then be written as

$$N_{T_i} = \alpha_3 L_1 L_2 + \alpha_1 L_2 L_3 + \alpha_2 L_3 L_1$$

We use Lagrange multipliers to incorporate the constraints on cluster volume and maximum dimension. We form a Lagrangian,

$$\ell = N_{T_i} + \lambda(L_1 L_2 L_3 - \frac{N_1 N_2 N_3}{P}) + \sum_{j=1}^{3} \beta_j (N_j - L_j)$$

we get

$$L_j = \min(\frac{2\alpha_j}{-\lambda}, N_j),\ j = 1 \ldots 3$$

with λ chosen so that

$$L_1 L_2 L_3 = \frac{N_1 N_2 N_3}{P}$$

If $L_j < N_j \forall j$, then this simplifies to

$$L_1 = \sqrt[3]{\frac{\alpha_1^2}{\alpha_2 \alpha_3} V}$$

$$\begin{aligned} L_2 &= \sqrt[3]{\frac{\alpha_2^2}{\alpha_3\alpha_1}V} \\ L_3 &= \sqrt[3]{\frac{\alpha_3^2}{\alpha_1\alpha_2}V} \end{aligned} \tag{2.6}$$

We also have

$$\frac{L_i}{L_j} = \frac{\alpha_i}{\alpha_j}$$

This set of equations shows that the aspect ratio (L_i/L_j) is proportional to the $\alpha_i's$, or equivalently T_s and T_f. This demonstrates the smooth behaviour of the shape of the optimal cluster with respect to the architectural parameters.

If $T_f = T_s$, then $\alpha_1 = \alpha_2 = \alpha_3 = T_f$, and we have

$$L_1 = L_2 = L_3 = \sqrt[3]{V} = \sqrt[3]{\frac{N_1N_2N_3}{P}}$$

The aspect ratio is unity in this case, and the clusters are ideally cubes.

The minimum communication load for each cluster comes out to be

$$N_{T_i} \geq 3\sqrt[3]{\alpha_1\alpha_2\alpha_3}V^{2/3} \tag{2.7}$$

These equations make it clear that the lower bound varies smoothly with the α_i, and hence with respect to the architectural parameters T_s and T_f. Rewriting Equation 2.7 in terms of T_s and T_f, we get,

$$\begin{aligned} N_{T_i} &\geq 3(\min(T_f, T_s)^2 T_s)^{\frac{1}{3}} V^{\frac{2}{3}} \\ &= 3T_f V^{\frac{2}{3}} \qquad \text{if } T_f = T_s \end{aligned} \tag{2.8}$$

Assuming that all clusters are ideal in shape, we can derive a lower bound on the communication time as

$$\begin{aligned} T_{comm} &\geq 3(\min(T_f, T_s)^2 T_s)^{\frac{1}{3}} (\frac{N_1N_2N_3}{P})^{\frac{2}{3}} \\ &= \frac{3T_f N^2}{P^{\frac{2}{3}}} \; if \; N_1 = N_2 = N_3 = N, \; and \; T_f = T_s \end{aligned} \tag{2.9}$$

The key thing to note is that the minimum communication time decreases as $P^{2/3}$, less than linearly with the number of processors. This happens because as we increase the number of processors, the communication bandwidth increases linearly, but the total number of data transfers and memory accesses (number of transmissions) only increases as $P^{1/3}$.

Assuming perfect load balancing, the lower bound on the compute time is clearly

$$T_{compute} \geq (T_a + T_m)\frac{N_1 N_2 N_3}{P} \tag{2.10}$$

Assuming that load balancing is perfect, and all clusters are ideal in shape, we can derive the lower bound on the total execution time T_e as

$$T_e = T_{compute} + T_{comm} \geq (T_a + T_m)\frac{N_1 N_2 N_3}{P} + 3(\min(T_f, T_s)^2 T_s)^{\frac{1}{3}}(\frac{N_1 N_2 N_3}{P})^{\frac{2}{3}} \tag{2.11}$$

If $N_1 = N_2 = N_3 = N$ and $T_f = T_s$

$$T_e \geq (T_a + T_m)\frac{N^3}{P} + \frac{3T_f N^2}{P^{\frac{2}{3}}}$$

The speedup $S(P)$, which is the ratio of the execution time on 1 processor to the execution time on P processors, then becomes

$$S(P) = \frac{(T_a + T_m)N^3 + 3T_f N^2}{(T_a + T_m)\frac{N^3}{P} + \frac{3T_f N^2}{P^{\frac{2}{3}}}}$$

Thus the lower bound on the total time has a linearly decreasing and a less than linearly decreasing component. The linearly decreasing component is $O(N^3)$, while the less than linearly varying component is $O(N^2)$. Thus for large matrices (coarse granularity), the model predicts close to linear speedup. This is intuitive, since all overheads (even those not modeled in the multiprocessor model) decrease at coarse granularity. Similar results hold for special cases of the matrix product like the dot product (communication is $O(N)$ and computation $O(N)$) and the matrix times vector multiply (communication is $O(N^2)$, and computation $O(N^2)$).

In practice, bin packing algorithms can be used to partition the computation so that this lower bound is nearly achieved. These packing algorithms are presented below.

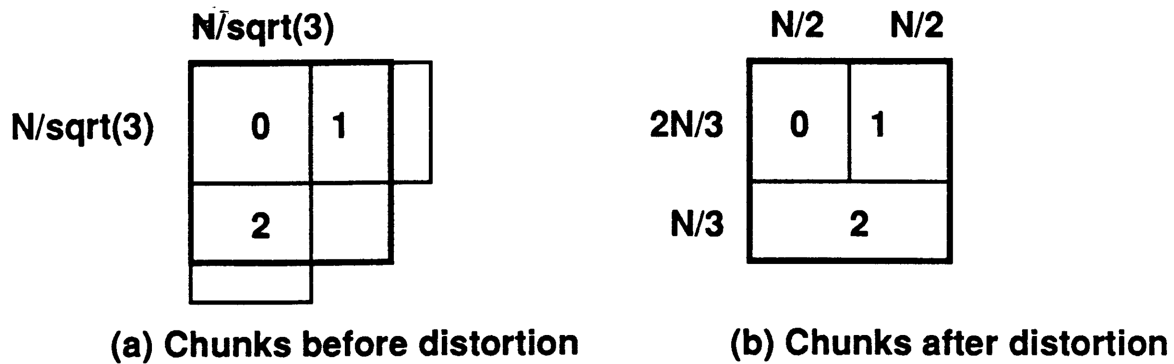


Figure 2.7: Optimal Square Partitioning

2.6.3 Previous Work on Partitioning Techniques

Previous work in the area of partitioning $N_1 \times N_2 \times N_3$ lattices into P equal sized chunks to minimise the projection sum has concentrated on the 2-D problem of chunking up an $N_1 \times N_2$ rectangle into P equal sized chunks so as to minimise the sum of projections. All projections are weighted equally ($T_s = T_f$) so the exact projection sum is being minimised. Simple algorithms are shown to be optimal.

[AK86, AK, KW87, KR89] have dealt with the 2-D problem in a continuous domain. [KR89] have applied these results as approximations to discrete 2-D lattices. We shall describe the continuous algorithms, as they are much simpler.

The 2-D $N_1 \times N_2$ rectangle partitioning algorithms [AK86, AK, KW87, KR89] basically work as follows. Since all P chunks are equal in size, the area of each chunk is fixed. To minimise the projection sum, the projection sum of each chunk should ideally be minimised, keeping its area fixed. This implies that each chunk should ideally be a square. In general, however, P equal area squares cannot be exactly fitted into an $N_1 \times N_2$ rectangle. The 2-D partitioning algorithms fix this by distorting the ideal square chunks so as to fit inside the $N_1 \times N_2$ rectangle.

For example, a partition of an $N_1 \times N_1$ square into 3 pieces is shown in Figure 2.7. Figure 2.7 (a) shows the ideal 3 square pieces. Figure 2.7 (b) shows the optimal partition obtained after distorting the pieces to fit. Since small changes in aspect ratio of any piece

does not change its projection sum very much, the projection sum in Figure 2.7 (b) is close to that in 2.7 (a).

We have applied this idea to the problem of chunking up a 3-D $N_1 \times N_2 \times N_3$ lattice.

Before we describe our algorithms, we shall make some remarks about the discrete problem.

Tiling solutions to partitioning

For the 2-D problem of partitioning a rectangular lattice, we have found an interesting *tiling* method which can achieve the optimal computationally balanced, minimal communication schedule in certain special cases. Again, for simplicity, assume that $T_s = T_f$ in what goes below, so we need to minimise the exact projection sum (all weights are unity).

Suppose the matrix is $N \times N$, the vector has N elements, and there are $P = N$ processors. Let n be an integer such that $n^2 \leq N < (n+1)^2$. Then we can carve the dataflow graph into tiles shaped as an $n \times n$ or $(n+1) \times n$ rectangle with the last row possibly incomplete. These tiles can be fitted into the square dataflow graph, with the kth tile's upper left hand corner at node $(kn \bmod N, k)$ in the lattice. Figure 2.8 shows this optimal partitioning for $N = P = 5$, and compares it with the sub-optimal partitioning suggested by [KR89]. (Our tiling method must be optimal because each tile is as "close" to square as possible.)

Unfortunately, for arbitrary rectangular or 3-D dataflow graph's, an optimal tiling solution may not necessarily exist. General purpose heuristics are needed.

2.6.4 Heuristics for Partitioning Matrix Products

Our heuristics depend on representing the dataflow graph as a continuous box, ignoring the discrete lattice structure inside. The heuristics attempt to find a partition of this box into P equal volume chunks with minimal (weighted) projected surface area. Ideally all chunks should be rectangular with the correct aspect ratio (Equation 2.6). After the

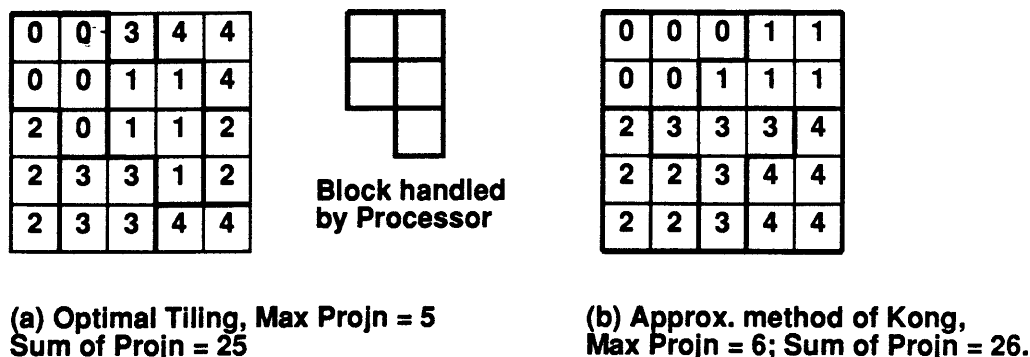


Figure 2.8: Optimal Tiling solution compared with Kong's Approximation

partition is found, we quantize the chunk boundaries so that each processor is assigned specific multiply-adds.

The theoretical lower bound on communication cost suggests that the bound is achieved when each chunk is as close as possible to being rectangular (cubical if $T_f = T_s$). It is important to note, however, that the projected surface area of a cube does not increase greatly when that cube is distorted into a rectilinear form. In fact, boxes whose aspect ratio is no worse than a factor of two (or three) have a projected surface area no more than 6% (or 15%) larger than that of a cube with the same volume. Good partitions, therefore, can be built with chunks which deviate quite strongly from a perfect cube.

This observation suggests several possible heuristics for partitioning the dataflow graph. The simplest approach partitions the matrix C into P approximately square chunks, and assigns responsibility for computing each section of the matrix product to a separate processor. This corresponds to partitioning the dataflow graph into P long columns, each oriented in the direction of the N_2 axis. Though simple, this heuristic is often far from optimal, since the chunks are so far from cubical.

Another heuristic method is a generalization of the 2-D rectangular partition algorithm in [AK86, AK, KR89], which is proven to be optimal in 2-D. Start by arranging P ideal chunks to fill a volume shaped similarly to that of the dataflow graph. The chunks will form the shape of a rectangular box, possibly with one face incomplete, and

heavily overlapping the form of the dataflow graph. Now distort all the P chunks in such a manner that they are all distorted by about the same amount, and so that the final arrangement exactly fills the dataflow graph. In practice, the initial arrangement of the cubes is critical to success, while the distortion technique used does not appear to matter as much, since the projected surface area does not increase greatly with distortion.

The boundaries may be translated into the discrete domain by simply rounding the chunk boundaries to the nearest lattice points. This makes the sizes of chunks unequal in general, but the relative error is small for large matrices (large N_1, N_2, and N_3). Using planar surfaces for the chunks also simplifies the real-time programming and reduces loop overhead. The computational balance can be improved by reallocating lattice points from one chunk to another while trying to keep the projected surface area the same. If the projections of the chunk boundaries in the summing (N_2) direction are irregular, however, this approach has the disadvantage of requiring element-level synchronization between processors at the chunk boundaries. Irregular chunk boundaries also complicate the loop control in the real-time program.

Our heuristics follow this general principle of first laying out the processor clusters approximately filling the dataflow graph, distorting them for filling the dataflow graph (while keeping loads balanced), and finally discretising. Below we present one of our heuristics.

Partitioning Heuristic

In this heuristic, the first phase consists of choosing an initial arrangement of processor clusters to approximately cover the dataflow graph. The initial arrangement is chosen to closely match the ideal arrangement of clusters predicted by Equation 2.6. The second phase consists of distorting the initial arrangement to exactly fill the dataflow graph. The third and final phase does the discretising.

The heuristic exploits the shape of the ideal processor cluster, viz. a rectangular box of the correct aspect ratio, as proved in Section 2.6.2 (Equation 2.6). In general boxes

of this aspect_ratio will not fit exactly in the dataflow graph. The three sides of the ideal rectangular cluster (L_1, L_2, L_3) (Figure 2.6), will not in general divide the sides (N_1, N_2, N_3) of the dataflow graph exactly. If we let

$$pa = \frac{N_1}{L_1} \;\; pb = \frac{N_2}{L_2} \;\; and \;\; pc = \frac{N_3}{L_3} \tag{2.12}$$

then along the N_1 axis pa ideal boxes, along N_2 axis pb ideal boxes, and along the N_3 axis pc ideal boxes will fit, where pa, pb, and pc are not in general integers. If they were integers, then the lower bound on communication could be exactly met (ignoring the discrete nature of the problem) by laying out the boxes (processor clusters) in the form of a $pa \times pb \times pc$ lattice. This arrangement would be optimal, and no distortion would be necessary. The boxes have to be distorted if pa, pb and pc are not integral, and heuristics are needed for this purpose. Below we describe each of the three phases of Heuristic 1 in turn, in the context of partitioning up an $N \times N \times N$ dataflow graph using $P = 14$ processors (Figures 2.9 and 2.10).

The first phase first computes pa, pb, and pc (Equation 2.12). Then a search is performed (integral-value-search) for integral values (p_1, p_2, p_3) close to the tuple (pa, pb, pc), satisfying the constraints that at least P processor clusters must fit inside the corresponding $p_1 \times p_2 \times p_3$ lattice (i.e. $P \leq p_1 p_2 p_3$). All possible allowed 3-tuples (p_1, p_2, p_3) in the search space are examined, and the best selected. In general the lattice will contain more than P processor clusters, and hence some lattice points will not be occupied.

The search space (for the integral-value-search) is limited as follows. p_1 is varied over all integers from the greatest factor of N_1 lower than pa to the lowest factor of N_1 greater than pa. The rationale for this choice comes from the discreteness of the dataflow graph. Error in discretizing can be eliminated if the number of boxes along each axis (p_1, p_2, p_3) exactly divides (N_1, N_2, N_3). Hence it is reasonable to look for the factors of N_1 nearest p_a. Similarly, p_2 is varied from the greatest factor of N_2 lower than pb to the lowest factor of N_2 greater than pb. p_3 is varied from the greatest factor of N_3 lower than pc to the lowest factor of N_3 greater than pc.

Selection amongst the various candidate (p_1, p_2, p_3), is done by estimating the resulting

communication relatively crudely. In the example, we assume that the search has resulted in a $3 \times 3 \times 2$ lattice, housing upto 18 processor clusters, but being filled with only 14.

Having completed the integral-value-search, we need to (partially) fill the $p_1 \times p_2 \times p_3$ lattice with P processor clusters. A raster scan scheme is used. The lattice in general is incompletely filled, with a single plane lacking some processors. For example, Figure 2.9 shows a possible layout for our example. The $3 \times 3 \times 2$ lattice ABCDEFGH has a complete 3×3 plane ABCD, and an incomplete plane EFGH with 5 processor clusters.

If the processors available cannot fill the $p_1 \times p_2 \times p_3$ lattice, $P < p_1 p_2 p_3$, then the second phase applies a distortion heuristic to the processor clusters to exactly fill the dataflow graph. The incomplete plane introduces complications. We tackle this problem by splitting the incomplete $p_1 \times p_2 \times p_3$ lattice into four complete sublattices, and also splitting the dataflow graph into four corresponding blocks. Each complete sublattice is assigned one dataflow graph block. The distortion heuristic appears in determining the boundaries of each of the four blocks.

With reference to Figure 2.10, the incomplete $3 \times 3 \times 2$ lattice is separated into 4 complete sublattices by the planes XX', YY', and ZZ'. Sublattice SL_1 contains the clusters corresponding to processors 0,1,3,4,9,10,12, and 13. Sublattice SL_2 contains clusters corresponding to processors 2 and 11. SL_3 has clusters corresponding to processors 6 and 7, while SL_4 has clusters of processors 5 and 8. The dataflow graph is also split into four corresponding blocks. The block boundaries are determined by load balancing - first we determine the position of the plane XX', then that of YY' and ZZ'.

The result is shown in Figure 2.10, where the $N \times N \times N$ cube is chopped into four cubical blocks. The 8 processors 0,1,3,4,9,10,12, and 13 in SL_1 handle the $(8N/10) \times N \times (10N/14)$ top left block, taking time $N^3/14$ in all. Similarly the two processors 5 and 8 in SL_4 handle the $(N/2) \times N \times (4N/14)$ bottom right block, and so on. The cluster sizes are all tabulated in Table 2.1.

Finally, the boundaries in the continuous domain have to be translated to the discrete domain. This translation in our case is to the best (in terms of load balancing) adjacent

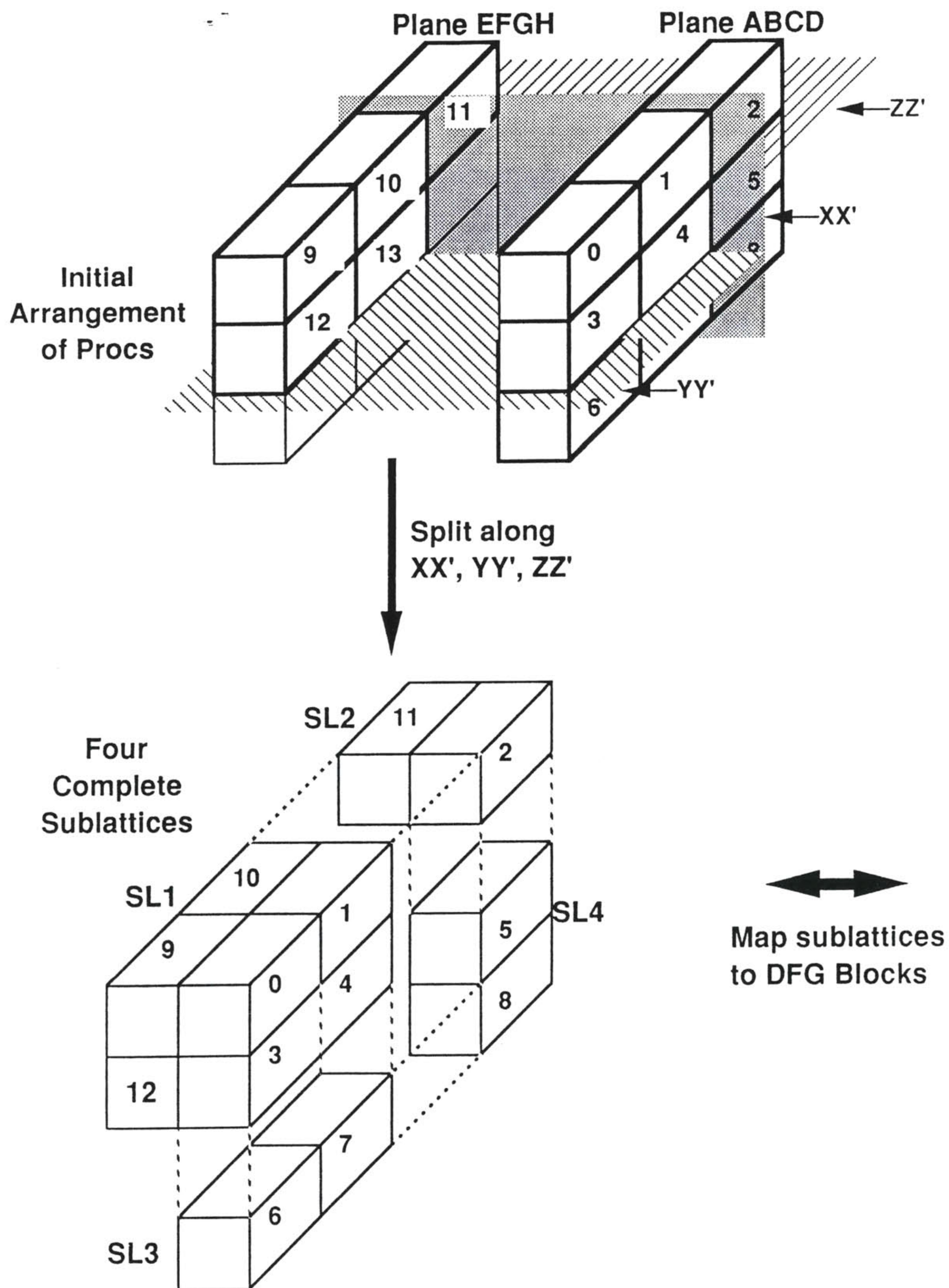


Figure 2.9: Partition of an $N \times N \times N$ cube into 14 chunks (a)

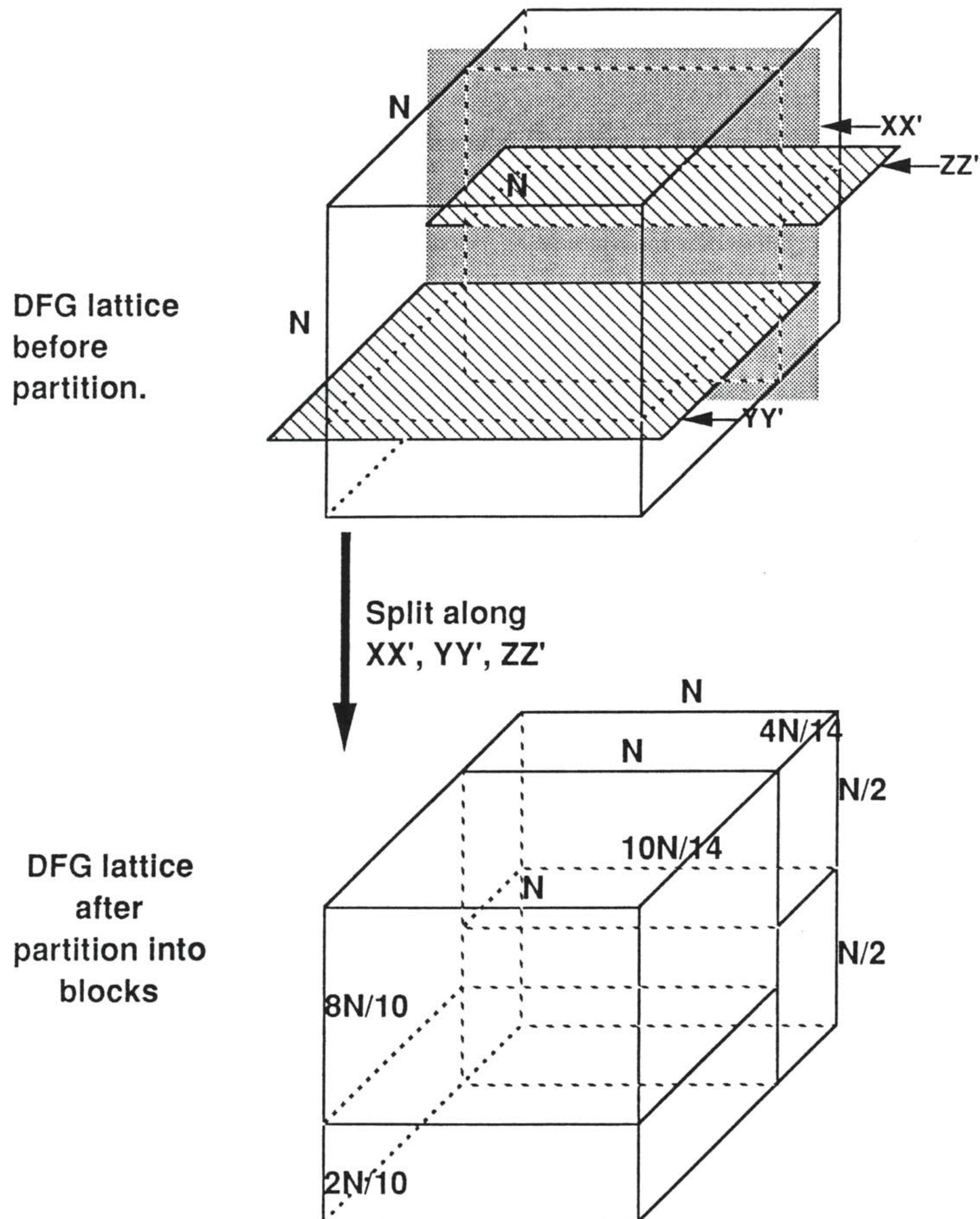


Figure 2.10: Partition of an $N \times N \times N$ cube into 14 chunks (b)

SL_1	0,1,3,4,9,10,12,13	$4N/10 \times N/2 \times 5N/14$
SL_2	2,11	$N/2 \times N/2 \times 4N/14$
SL_3	6,7	$2N/10 \times N \times 5N/14$
SL_4	5,8	$N/4 \times N \times 4N/14$

Table 2.1: Cluster sizes - Partition of $N \times N \times N$ cube into 14 chunks

integer value, and in general causes additional error. The error can be reduced by reallocating nodes from one cluster to adjacent ones till the load is sufficiently balanced (within 5 percent of optimal). This increases communication somewhat, but our simulations did not result in dramatic increase in most cases.

The node reallocation produces irregular clusters in general. This necessitates element level synchronization during accumulates, causing overhead. As the size of the matrices increases, discretization error becomes less significant, and hence this undesirable effect is reduced.

Figure 2.11 and Figure 2.12 show the performance of this algorithm, for a $20 \times 20 \times 20$ dataflow graph, as we vary the number of processors P from 1 to 100. The total communication was plotted by measuring cluster size. Processor reallocation has been used until the compute time is within 5 % of the lower bound.

The architectural parameters are $T_a + T_m = 1.0$, $T_f = 1.0$, and $T_s = 2.0$. Thus upto P multiply-adds and memory accesses can take place in a single cycle. Exchanging data between two processors takes two cycles. This very roughly models a single bus based shared memory multiprocessor like the Encore, where processors have to communicate through the global memory.

Communication load N_T and its lower bound with respect to P is plotted in Figure 2.11. Figure 2.12 shows N_T *scatter plotted* against ${T_{compute}}^{-1/3}$, which should ideally be a straight line. This result can be seen as follows.

$$N_T = PT_{comm} \propto PP^{-2/3} = P^{1/3} \propto {T_{compute}}^{-1/3} \tag{2.13}$$

The communication load in Figure 2.11 shows a somewhat random variation around

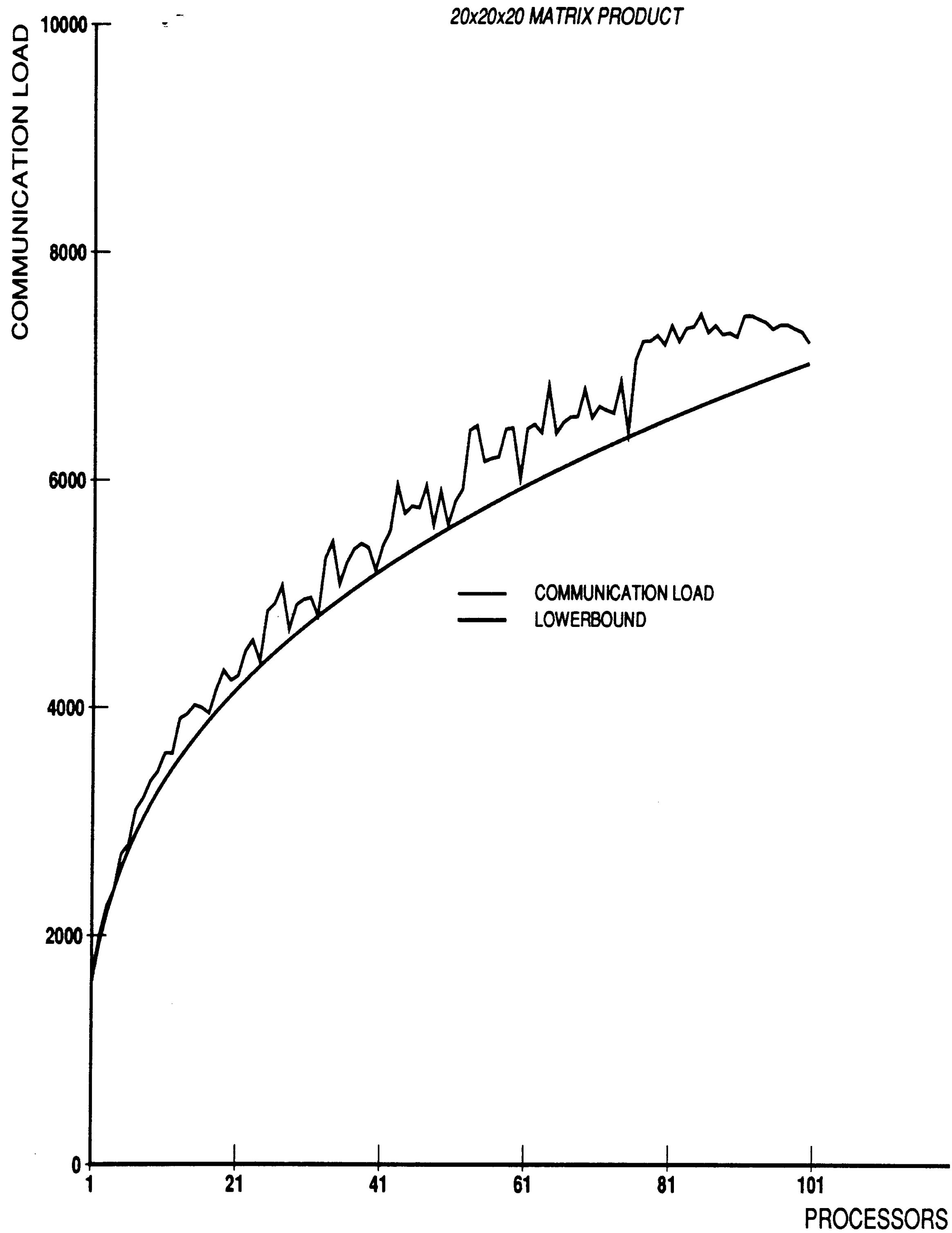


Figure 2.11: Communication vs Processors for $20 \times 20 \times 20$ Matrix Product

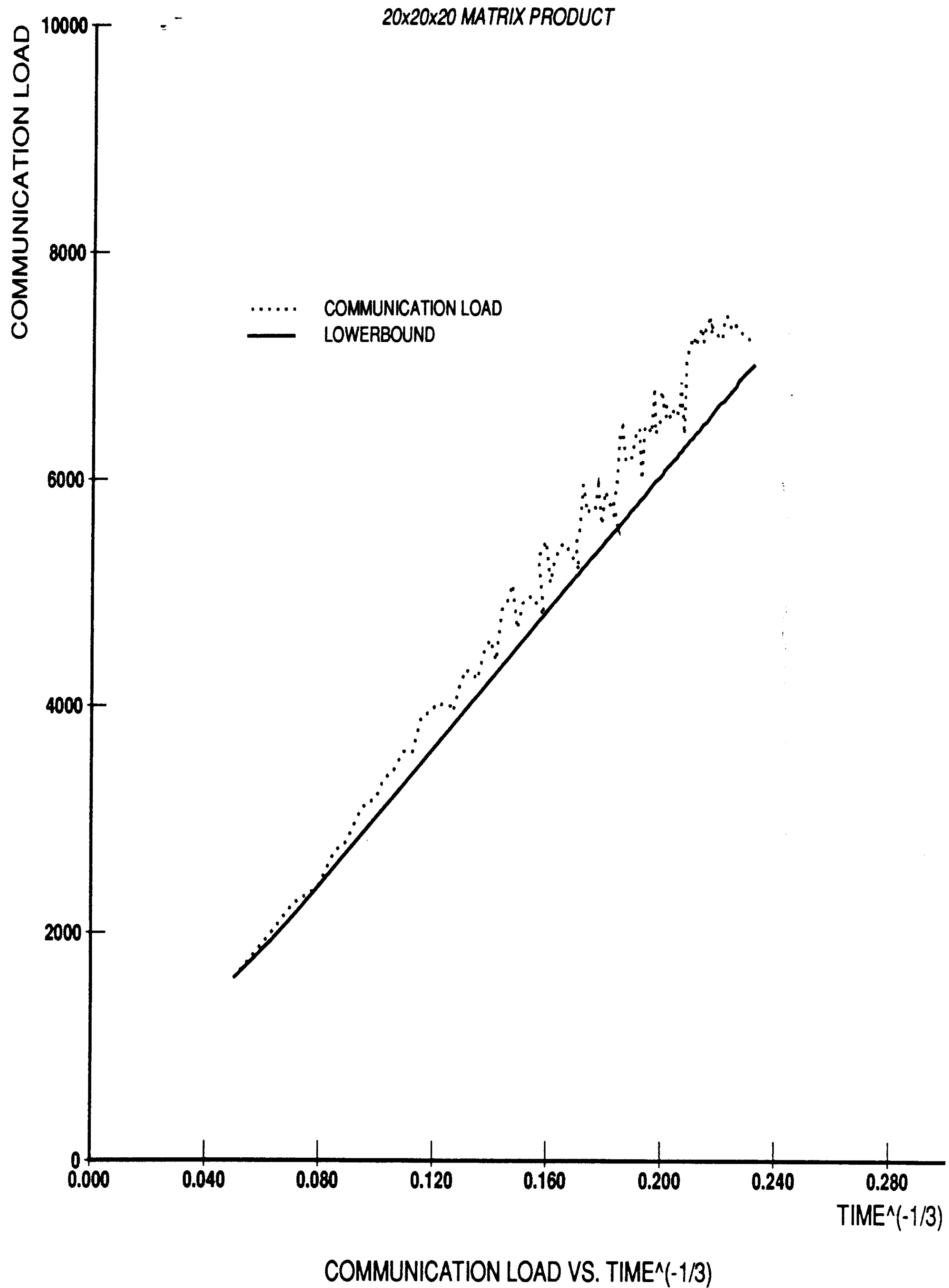


Figure 2.12: Communication vs $T_{compute}^{(-1/3)}$ for $20 \times 20 \times 20$ Matrix Product

the lower bound. It is generally within 20 % of the lower bound. For certain *matched* number of processors, the lower bound is exactly met. This is the case for $P = 4$, and $P = 32$. For these values of P, the dataflow graph dimensions fit well with the number of processors. The ideal clusters can be exactly packed into the dataflow graph without any distortion or discretization error, and the lower bound on the communication is exactly met. For other values of P, distortion and or discretization is in general needed, leading to some increase in communication load.

The results in Figure 2.11 and Figure 2.12 show that it is possible to devise reasonable partitioning heuristics, with computation time being within 5 % of the lower bound (by design), and communication load within about 20 % of the lower bound.

The relative behaviour of compute time and communication load (Equation 2.13) is also roughly verified in (b).

Figure 2.13 and Figure 2.14 show the performance of the algorithm on the product of a 10×40 matrix, with a 40×20 matrix. We have changed the aspect ratio of the dataflow graph, while keeping the same number of nodes as the previous example. Again the results are close to optimal, with the communication being within 20 % of lower bound, and the computation time being within 5 % of the lower bound.

2.6.5 Experimental results

Experimental results on the MIT Alewife machine will be described later on, in Chapter 4.

2.7 Continuous processors and mode shifts

It is convenient, at least for theoretical purposes, to allow the number of processors P to be a continuous variable. We interpret this as having $\text{int}(P)$ processors working full time on the problem, and one more processor being time shared, with a fraction $\text{frac}(P)$ of its time devoted to this problem. An application of this concept will be illustrated below in

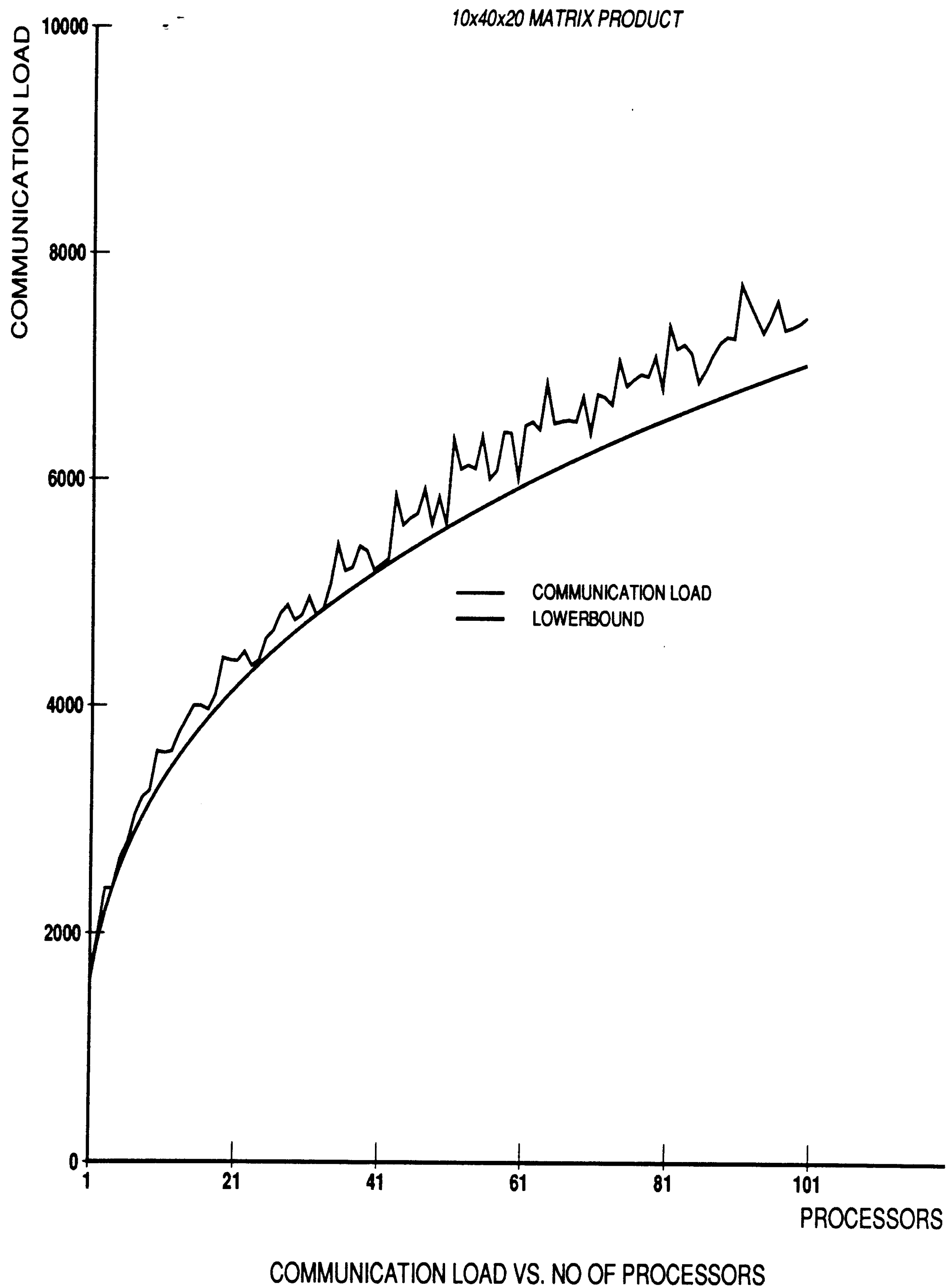


Figure 2.13: Communication vs Processors for 10 × 40 × 20 Matrix Product

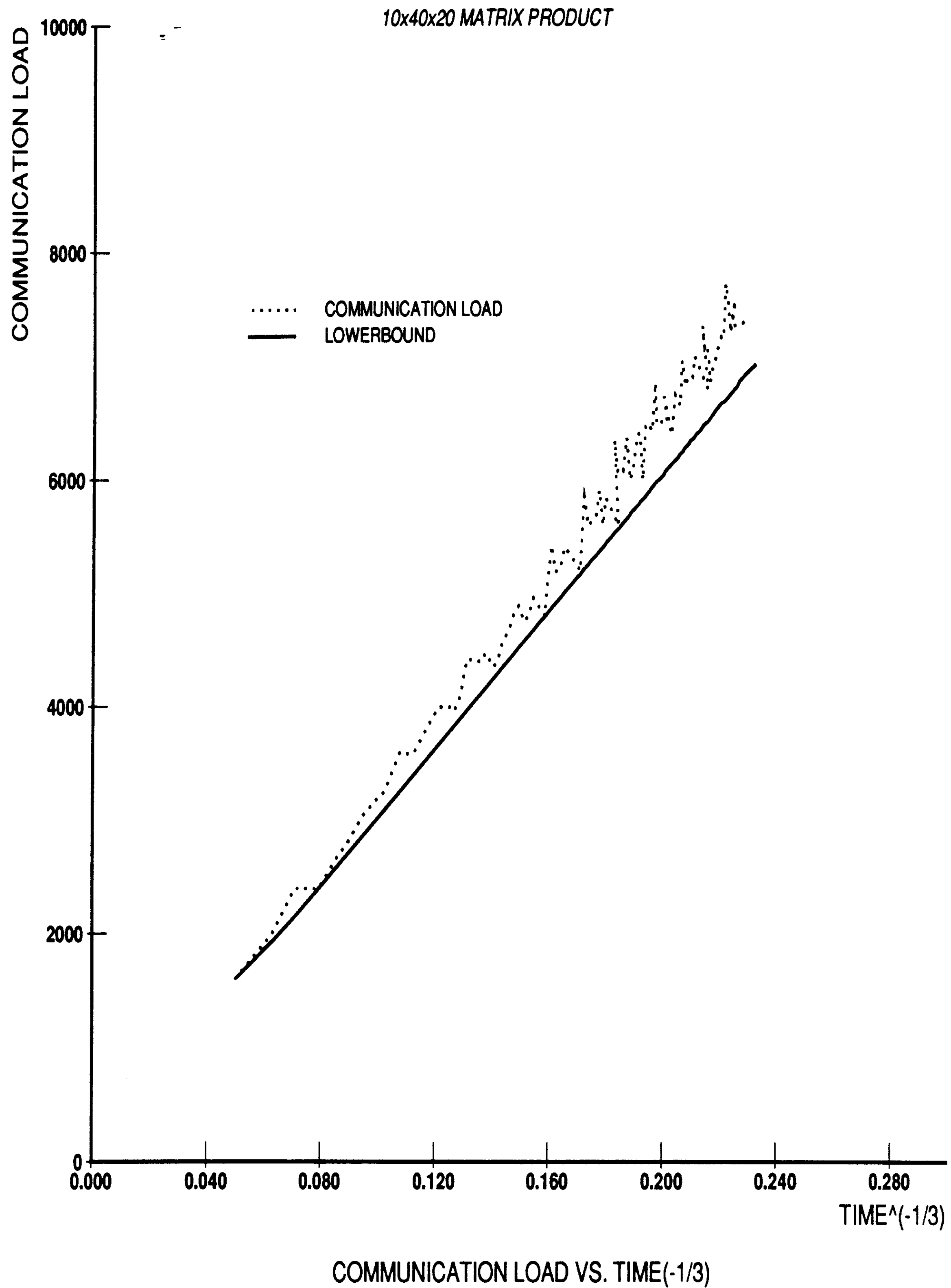


Figure 2.14: Communication vs $T_{compute}^{(-1/3)}$ for $10 \times 40 \times 20$ Matrix Product

the context of computing matrix products.

If M identical matrix products are to be performed on P processors, $P \geq M$, load balancing needs $\frac{P}{M}$ processors to be allocated to each operation. $\frac{P}{M}$ is in general not an integer. An allocation of $\frac{P}{M}$ processors means that $\text{int}(\frac{P}{M})$ processors are working full time on the operation, and one processor is working $\text{frac}(\frac{P}{M})$ of the time. Hence the last processor handles fewer nodes than the rest. Thus the sizes of the clusters handled by each processor are different, with the size of the last cluster being

$$\text{frac}(\frac{P}{M}) \tag{2.14}$$

of the others.

If $P < M$, then only one processor is working part time on the operation.

2.7.1 Extension of Lower Bounds

The extensive structure present in the matrix-product dataflow graph allows us to extend the lower bound in Equation 2.9 and devise good heuristics to perform non-integer processor allocation. Let P be between successive integers K and $K+1$.

$$K < P \leq K+1$$

$K+1$ processors cooperate to compute the matrix product. The first K processors handle clusters of size

$$V_1 = \frac{N_1 N_2 N_3}{P}$$

The $(K+1)^{th}$ processor handles a cluster of size

$$V_{K+1} = \epsilon V_1 \qquad \text{where} \epsilon = \text{frac}(P) = P - K$$

which is smaller than the others.

The communication load is minimised when all clusters are ideal in shape. If $T_s = T_f$, the clusters are ideally cubes, and

$$N_T \geq T_f \left(3K + 3\epsilon^{2/3}\right) \left(\frac{N_1 N_2 N_3}{P}\right)^{2/3} \tag{2.15}$$

ϵ increases continuously from $\epsilon = 0$ to $\epsilon = 1$ as P increases continuously from K to $K+1$. The communication load increases as $\epsilon^{2/3}$, which is very steep near $\epsilon = 0$. Hence the communication increases very steeply as P changes between successive integers.

2.7.2 Mode Shifts

In general, for non-integer P, either heuristics or a search must be used to find a good partition of the dataflow graph. The use of non-integer P introduces an interesting *mode shift* behavior in the structure of the optimal solution, with communication cost rising steeply as P varies from an integer value K to its successor $K+1$. This can be illustrated by the case of an $N \times N$ dataflow graph for a matrix-vector product, with $P = 1 + \epsilon$ processors, where $0 \leq \epsilon \leq 1$. The optimal partition of the dataflow graph is shown in figure 2.15(a) for $1 \leq P \leq 4/3$, and switches to the form shown in figure 2.15(b) for $4/3 \leq P \leq 2$. Optimal communication cost is:

$$N_T = \begin{cases} 2N + N^2 + 2N\sqrt{1 - \frac{1}{P}} & \text{for } 1 \leq P \leq 4/3 \\ 2N + N^2 + N & \text{for } 4/3 \leq P \leq 2 \end{cases} \tag{2.16}$$

Note that the communication cost for using $P = 4/3$ processors is identical to that for $P = 2$ processors; adding fractional processor power is thus quite costly. This mode shift behavior of the optimal partition, and the steeply rising cost for fractional processors, are characteristic of the general matrix multiplication scheduling problem.

2.8 Special cases of Matrix Sums and Products

The bounds on communication derived previously assumed that the matrices A and B share no common elements. Tight bounds for special cases like A^2, AA^T, A and/or B symmetric, are much more difficult to derive. In general for these problems, the optimal processor clusters are not cubical. We shall discuss optimal scheduling of AA^T in this section. For simplicity we assume that $T_s = T_f$.

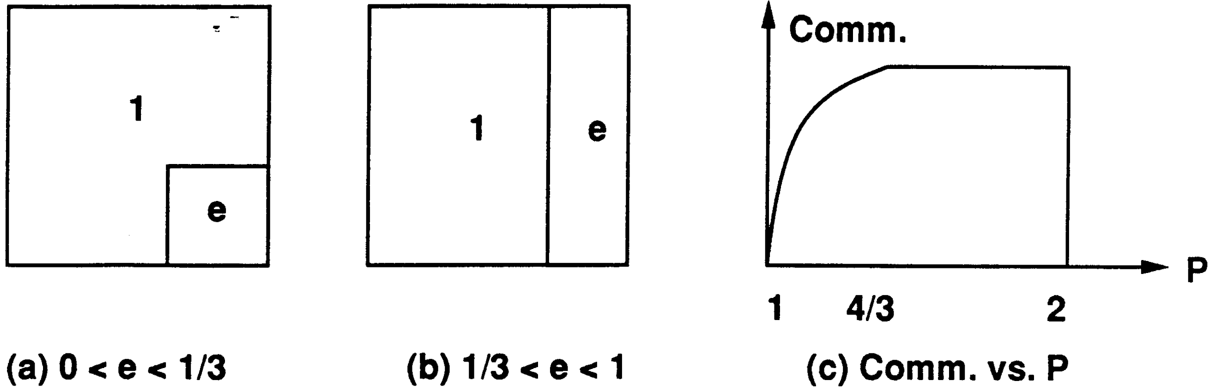


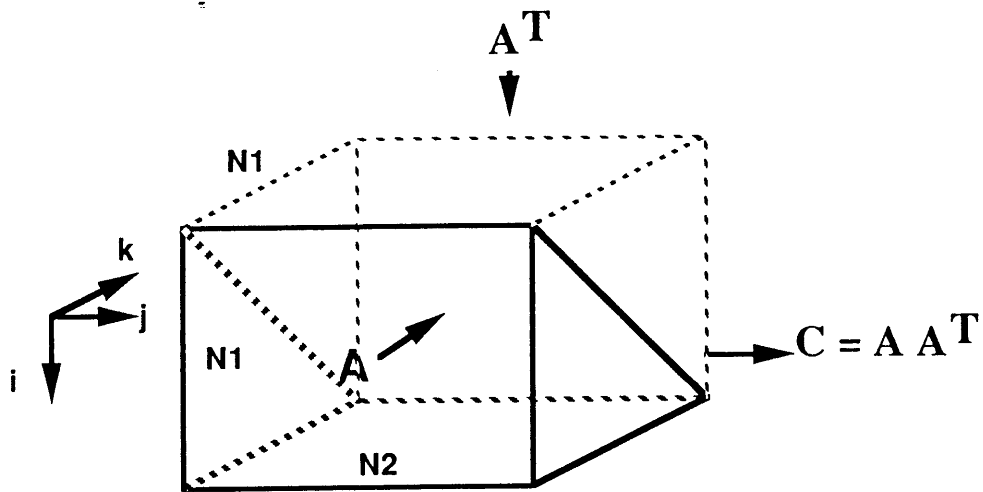
Figure 2.15: Mode-Shifts for $1 \leq P(=1+\epsilon) \leq 2$.

Since $C = AA^T$ is symmetric, only one half of the output C need be computed (above or below the main diagonal). Figure 2.16 shows the dataflow graph in (a), together with the optimal shape of a processor cluster in (b) and (c). The dataflow graph is a $N_1 \times N_2 \times N_1$ triangular prism, with its axis along the N_2 or j direction (Figure 2.16 (a)). Elements of A are broadcast from one of the two rectangular faces of the triangular prism (along the k axis), and elements of A^T from the other (along the i axis).

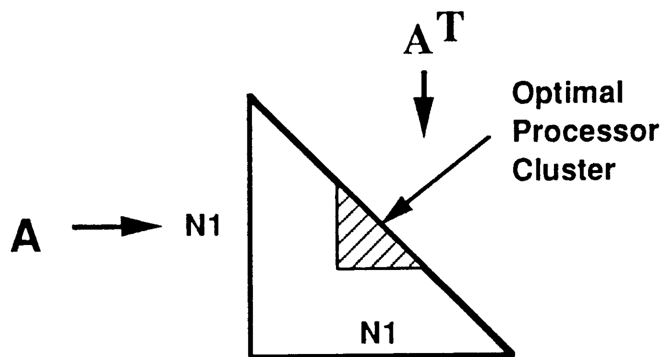
The projection of the dataflow graph prism on the N_2 or j direction in shown in Figure 2.16 (b). Because the two matrices are transposes of each other, accessing a_{ij} is equivalent to accessing a^T_{jk}, for $k = i$. Since C is symmetric, only its portion on and below the main diagonal need be computed.

It can then be seen that the processor cluster with the lowest communication for a given volume is itself a triangular prism (apart from row permutations), (Figure 2.16 (c)). The diagonal face of the prism is aligned along the long diagonal face of the dataflow graph. Then, accesses of elements of A, a_{ij}, are identical to accesses of the corresponding elements of A^T, a^T_{ji}, and the communication is halved. In this case both the shape and location of the optimal processor cluster are constrained. This is unlike the ordinary matrix product case, in which the location of the processor cluster was arbitrary.

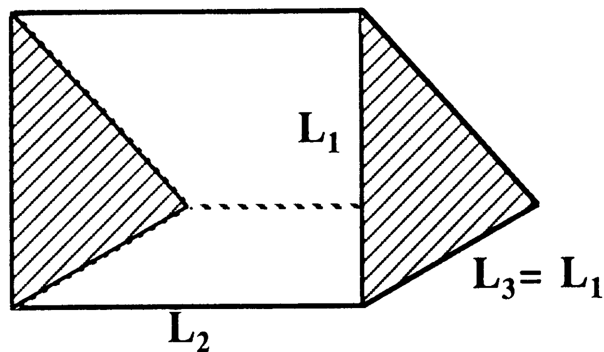
We denote the sides of the optimal triangular prism by L_1, L_2, and L_3, with $L_1 = L_3$ (Figure 2.16 (c)). Assume that the optimal solution manages somehow to make the



(a) DFG for AA^T



(b) Optimal Processor Cluster



(c) Optimal Cluster Shape

Figure 2.16: Optimal partition of AA^T

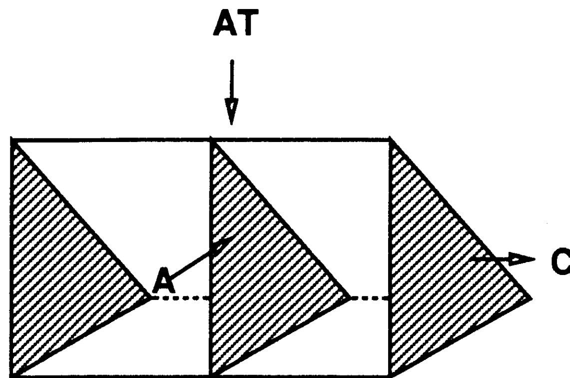


Figure 2.17: Optimally partitioned dataflow graph of AA^T, $P = 2$ blocks

clusters into triangular prisms, positioned flush against the diagonal face of the dataflow graph. Computing all the nodes in the optimal processor cluster necessitates reading elements of either A or A^T, not both. Thus only one face L_1L_2 or L_3L_2 contributes to memory accesses. The L_1L_3 face contributes to interprocessor data transfers for computing partial sums of C. Thus the communication load for the ideal (say i^{th}) processor cluster is given by

$$N_{T_i} = T_f(L_1L_2 + \frac{1}{2}L_1L_3) = T_f(\frac{V}{L_1} + \frac{1}{2}L_1^2)$$

with

$$L_1L_2L_3 = L_1^2L_2 = V = \frac{N_1^2N_2}{P}$$

being twice the volume of the triangular dataflow graph prism.

Minimising with respect to L_1 yields

$$L_1 = L_2 = L_3 = V^{(1/3)}$$

The minimum communication load for the ideal cluster (ideal in shape and position) is

$$N_{T_i} \geq 1.5T_fV^{(2/3)}$$

The total communication load is bounded by

$$N_T \geq 1.5PT_fV^{(2/3)} = 1.5T_f(N_1^2N_2)^{(2/3)}P^{(1/3)}$$

Assuming ideal-load balancing, the execution time becomes

$$T_e = T_{compute} + T_{comm} \geq (T_a + T_m)\frac{N_1^2 N_2}{2P} + 1.5T_f(\frac{N_1^2 N_2}{P})^{\frac{2}{3}} \tag{2.17}$$

If A is square ($N_1 = N_2 = N$)

$$T_e \geq (T_a + T_m)\frac{N^3}{2P} + \frac{1.5T_f N^2}{P^{\frac{2}{3}}}$$

Hitting the lower bound in this case is significantly more difficult, since the optimal processor clusters are no longer rectangular in shape, and have to be positioned along the diagonal face of the dataflow graph (apart from row permutations). However, if we have a long thin dataflow graph of aspect ratio P ($N_2 = PN_1$), then the lower bound can be met by slicing up the triangular dataflow graph prism along the N_2 or j direction into P pieces (Figure 2.17).

A similar analysis can be attempted for the case where A or B are symmetric, or have special properties like bandedness, etc. The basic paradigm is having compact processor clusters to minimise communication. Dealing with A^2 is much more complex, and we don't discuss it here.

Chapter 3

Generalised Scheduling

3.1 Introduction

Chapter 2 demonstrated how parallel library routines for each matrix operator could be devised. The basic paradigm was to exploit the polyhedral geometric structure of the dataflow graph of the operator, to devise a good parallel library routine for it. The complex discrete behaviour of the parallel library routine can be simply summarised by a speedup curve. This speedup curve can be derived by either algorithmic analysis (Section 2.6.2) or explicit measurement.

The next step in compiling a matrix expression is to compose together the library routines for each operator, to get a good compilation for the complete expression. This step involves determining the optimal parallelism for each particular library routine, as well as an optimal sequencing of these routines. Classical scheduling theory does not deal with this problem, since it does not determine the optimal parallelism of the graph. Theoretical insights for this *generalised multiprocessor scheduling* problem are hence of great value. This is presented below. The speedup curve derived for each library routine in Chapter 2 finds extensive use in the theoretical formulation.

This chapter presents a novel approach to generalised multiprocessor scheduling using the theory of optimal control. We treat each task as a dynamic system, whose state can

be changed by applying processing power to it. We allow parallelism in tasks by allowing more than one processor to be simultaneously applied to it. We characterise each task by a speedup function specifying the rate at which computation of the task proceeds as a (continuous and differentiable) function of the amount of processor power applied. Precedence constraints on the order of task execution are incorporated in the speedup function itself. Total processing power applied at any one time is bounded above by the available capacity of the multiprocessor system. With this simple model of task dynamic behavior, applicable to many problems such as matrix expression evaluation, a number of elegant theorems can be derived. In certain cases, the scheduling problem is shown to be equivalent to a shortest path problem, which permits very efficient solutions which are provably optimal. Task graphs formed by series and parallel combinations of tasks are particularly easy to schedule, and the algorithm is similar to the computation of current flow through series and parallel resistors. Previous approaches [DL89, HL89, BW86, Cof76] to the multiprocessor scheduling problem have assumed that tasks can be performed on several processors simultaneously, if needed, but have required that an integer number of processors must be applied to any task. Unfortunately, optimal scheduling then requires solving an NP-hard problem. We avoid this by allowing time-sharing of processors, and by assuming that the speedup of any task is independent of the state of that task.

The chapter first presents the scheduling model in Section 3.2. Section 3.3 formulates the optimal scheduling problem in the framework of optimal control theory. Standard methods of optimal control theory are then applied to yield the solution in Section 3.4. A theorem is proven showing that the optimal solution adjusts the processor allocations until the marginal speedups in tasks running in parallel matches a fixed ratio. When a particular speedup function is assumed (p^α), we are able to derive an extremely powerful pair of theorems in Section 3.5. The processing power assigned to each task is a fixed fraction of the total power available throughout the running of the task. As the task completes, the fraction of power assigned to that task is redistributed to those successors

of the task which are enabled to run at the moment the task completes. We also show that the structure of the optimal schedule does not depend on the total processing power available; adding more processing power only makes the optimal schedule run faster. Simplifying the results slightly yields a simple scheduling algorithm in Section 3.6.1, based on recursively clustering the tasks into series and parallel sets. Examples are presented in Section 3.6.2. Remaining issues are discussed in Section 3.8 Readers unfamiliar with optimal control theory can skip Section 3.4, and the proofs in the appendices.

3.2 Model of the Parallel Task System

We start with a formal model of a parallel task system. Let $\Omega = \{1, \ldots, N\}$ be a set of N tasks to be executed on a system with $P(t)$ processors available at each time t. Suppose task i has length L_i. Also suppose there are precedence constraints among the tasks so that task i cannot start until after all preceding tasks in the set Ω_i have finished. Let Ω^i be the set of tasks which in turn depend on task i finishing before they can start. We will assume that the tasks are partially ordered with no feedback loops.

It is convenient to define the *state* $x_i(t)$ of task i at time t to be the amount of work done so far on the task, $0 \leq x_i(t) \leq L_i$. Let t_i be the earliest time at which all predecessors of i (if any) have finished, so that i can begin running. Thus $x_i(t) = 0$ for $t < t_i$, and $x_j(t_i) = L_j$ for all of i's predecessor tasks, $j \in \Omega_i$. If task i has no predecessors, $t_i = 0$.

Let $p_i(t)$ be the processing power (number of processors) applied to task i at time t, and let $P(t)$ be the total processing power available at time t. The $p_i(t)$ are all non-negative, and must sum to less than $P(t)$. In effect, we assume that multiprocessor versions of all tasks are available so that any number of processor could be assigned to any task at any time. Furthermore, we will not constrain the $p_i(t)$ to be integers, but will assume that zero-overhead time-sharing of processors can be used to achieve fractional processor allocations.

Finally, assume that once a task's predecessors have finished, the rate at which a task proceeds, $dx_i(t)/dt$, depends in some nonlinear fashion on the amount of processor power applied, $p_i(t)$, but not on the state $x_i(t)$ of the task, nor explicitly on the time t. We call this the assumption of *space-time invariant dynamics*. Thus we can write:

$$\frac{dx_i(t)}{dt} = \begin{cases} 0 & \text{for } t < t_i \\ f_i(p_i(t)) & \text{for } t \geq t_i \end{cases} \tag{3.1}$$

where $f_i(p_i(t))$ will be called the *processing rate function*, or the *speedup function*. With no processing power applied, the task state should not change, $f_i(0) = 0$. With processing

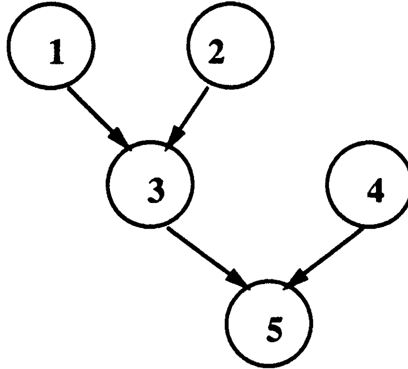


Figure 3.1: Example of a parallel task system

power applied, the task should proceed at some non-zero rate, $f_i(p) > 0$ for $p > 0$. We further assume that $f_i(p)$ is non-decreasing, so that adding more processors can only make the task run faster. Also, to ensure mathematical tractability, we will assume that $f_i(p)$ is differentiable, with non-negative derivative for $p \geq 0$. Also, note that we have allowed the processing rate function to be non-zero even after the task is finished, ie. $x_i \geq L_i$. However, this creates no problems, since the optimal solution will not waste processing resources on a finished task ($p_i = 0$ if $x_i > L_i$).

In effect, this form of the speedup function implies that tasks can be dynamically configured into arbitrary numbers of parallel modules for execution on separate processors. Processors can be added or removed at any time, and so that the processors assigned to the task can all do useful work. Although many general purpose computations can not be easily fit into this model, there are a number of problems in signal processing and numerical calculation, particularly for matrix expression evaluation, which are suitable. Furthermore, as we will see later, for certain special speedup functions, the number of processors assigned to a task in the optimal solution is actually constant.

Our goal is to finish all tasks in the minimum amount of time t^F, by properly allocating processor resources $p_i(t)$.

An example should clarify our notation. Consider the task system in Figure 3.1. Let task i have processing rate function $f_i(p_i)$, and total length L_i. Then the predecessor

task sets are:

$$\Omega_1 = \Omega_2 = \Omega_4 = \phi, \qquad \Omega_3 = (1,2) \qquad \Omega_5 = (3,4)$$

The successor task sets are:

$$\Omega^1 = \Omega^2 = (3) \qquad \Omega^3 = \Omega^4 = (5) \qquad \Omega^5 = \phi$$

The system equations are:

$$\begin{aligned}
\dot{x_1} &= f_1(p_1(t)) \\
\dot{x_2} &= f_2(p_2(t)) \\
\dot{x_3} &= \begin{cases} f_3(p_3(t)) & \text{for } t \geq t_3 \\ 0 & \text{for } t < t_3 \end{cases} \\
\dot{x_4} &= f_4(p_4(t)) \\
\dot{x_5} &= \begin{cases} f_5(p_5(t)) & \text{for } t \geq t_5 \\ 0 & \text{for } t < t_5 \end{cases}
\end{aligned}$$

3.3 Setting up the Optimal Control Solution

We can state the scheduling problem in Section 3.2 in a form appropriate for applying standard control theoretic approaches [BH75]. We must specify the state variables and their constraints, the control variables and their constraints, the system dynamics, and the objective function.

State Variables and Terminal Constraints

The state variables are the $x_i(t)$'s. Clearly they satisfy the terminal constraints

$$x_i(0) = 0, \qquad x_i(t^F) = L_i, \qquad \text{for } i = 1, \ldots, N \tag{3.2}$$

In vector notation

$$\vec{x}(0) = 0, \qquad \vec{x}(t^F) = \vec{L}$$

Control Variables and Constraints

The control variables are the processor powers allocated to each task over time, $p_i(t)$ (in vector notation $\vec{p}(t)$.) Each $p_i(t)$ is clearly non-negative. Also, the total processing power at any time t is at most $P(t)$. These two constraints can be written as

$$C(\vec{p}, t) \leq 0, \qquad \text{where } C(\vec{p}, t) = \sum_i p_i(t) - P(t) \tag{3.3}$$

$$D_i(\vec{p}, t) \leq \vec{0}, \qquad \text{where } D_i(\vec{p}, t) = -p_i(t), \qquad \text{for } i = 1, \ldots, N \tag{3.4}$$

System Dynamics

An approximation is required in order to write the state dynamics in a form suitable for control theory. First, write (3.1) in the form:

$$\frac{dx_i(t)}{dt} = f_i(\vec{x}(t), \vec{p}(t), t) \tag{3.5}$$

where we define:

$$f_i(\vec{x}(t), \vec{p}(t), t) = \begin{cases} 0 & \text{for } t < t_i \\ f_i(p_i(t)) & \text{for } t \geq t_i \end{cases}$$

We now replace this definition with an approximation which is differentiable everywhere:

$$f_i(\vec{x}(t), \vec{p}(t), t) = g_i(\vec{x}) f_i(p_i(t))$$

where

$$g_i(\vec{x}) = \prod_{j \in \Omega_i} \tilde{U}(x_j(t) - L_j)$$

where $\tilde{U}(x)$ is a differentiable approximation to a step function, $\tilde{U}(t) = 0$ for $t \leq -\epsilon$, $= 1$ for $t \geq 0$, and rises monotonically from 0 to 1 over the range $-\epsilon \leq t \leq 0$. Let $\tilde{\delta}(t)$ be the derivative of $\tilde{U}(t)$; note that $\tilde{\delta}(t)$ is a finite approximation to an impulse function with width ϵ, and which is non-zero (positive) only for $-\epsilon < t < 0$. Thus $g_i(\vec{x})$ is a function which is zero until the predecessors of task i are all within ϵ of finishing, at which point it smoothly increases up to a final value of 1 when all predecessors completely finish. For ϵ much smaller than the task lengths, this form of the state dynamics will be nearly the same as our original model (3.1).

For our previous example in figure 3.1, we would approximate the dynamics of tasks 3 and 5 as follows:

$$\dot{x}_3(t) = f_3(p_3(t)) \tilde{U}(x_1(t) - L_1) \tilde{U}(x_2(t) - L_2)$$

$$\dot{x}_5(t) = f_5(p_5(t)) \tilde{U}(x_3(t) - L_3) \tilde{U}(x_4(t) - L_4)$$

Because of this "soft" start for each task, we need to carefully define the starting and finishing times of the tasks. Let t_i be the earliest time that all predecessors of task i reach to within ϵ of completion, so that:

$$\prod_{k \in \Omega_i} \tilde{U}(x_k(t) - L_k) > 0$$

for all $t > t_i$. We call t_i the start time for task i. Similarly, let t_i^ϵ be the time at which task i first reaches to within ϵ of the end, $x_i(t_i^\epsilon) = L_i - \epsilon$, and let t_i^F be the finish time, when task i first reaches the end, $x(t_i^F) = L_i$.

Objective Function

Our goal is to minimize the final task completion time $t^F = \max_i t_i^F$, when all tasks have reached their end, $x_i(t^F) = L_i$ for all $i \in \Omega$, subject to constraints (3.2), (3.3), (3.4), and (3.5). Typical optimal control problems minimize an objective function formed of a penalty on the final state, $\phi(\vec{x}(t^F), t^F)$ plus an integrated penalty on the state trajectory and control values $L(\vec{x}(t), \vec{p}(t), t)$. We can state our objective in this form as follows:

$$\min_{\vec{p}(t)} \left[\phi(\vec{x}(t^F), t^F) + \int_0^{t^F} L(\vec{x}, \vec{p}, t) dt \right] \tag{3.6}$$

if we define:

$$\phi(\vec{x}(t^F), t^F) \equiv 0, \qquad L(\vec{x}, \vec{p}, t) \equiv 1$$

3.4 Solution Method

The problem we have defined can be solved by standard methods of optimal control. To apply these, introduce Lagrange multipliers ("influence functions") $\lambda_i(t)$, $\mu(t)$ and $\psi_i(t)$ associated with constraints (3.5), (3.3) and (3.4) respectively. Then form the Lagrangian:

$$\begin{aligned} J &= \phi(\vec{x}(t^F), t^F) + \int_0^{t^F} L(\vec{x}, \vec{p}, t)dt + \int_0^{t^F} \left(\sum_i \lambda_i(t) \left[f_i(\vec{x}, \vec{p}, t) - \dot{x}_i \right] \right) dt \quad (3.7) \\ &+ \int_0^{t^F} \left[\mu(t) C(\vec{p}, t) + \sum_i \psi_i(t) D_i(\vec{p}, t) \right] dt \end{aligned}$$

It is also convenient to define the *Hamiltonian*

$$H(\vec{x}, \vec{p}, t) = L(\vec{x}, \vec{p}, t) + \sum_i \lambda_i(t) f_i(\vec{x}, \vec{p}, t) + \mu(t) C(\vec{p}, t) + \sum_i \psi_i(t) D_i(\vec{p}, t) \quad (3.8)$$

Necessary conditions for an optimal scheduling solution for $\vec{p}(t)$ and $\vec{x}(t)$ can now be derived (c.f. [BH75, Chap 2,3]). For our problem, this optimal solution must not only satisfy all the constraints (3.2), (3.3), (3.4), and (3.5), but also the states $x_i(t)$, controls $p_i(t)$, and Lagrange multipliers $\lambda_i(t)$, $\mu(t)$, and $\psi_i(t)$ must satisfy the following constraints:

$$\frac{d\lambda_i(t)}{dt} = -\frac{\partial H(\vec{x}(t), \vec{p}(t), t)}{\partial x_i} \quad (3.9)$$

$$0 = \frac{\partial H(\vec{x}(t), \vec{p}(t), t)}{\partial p_i} \quad (3.10)$$

where

$$\begin{cases} \mu(t) > 0 & \text{if } \sum_i p_i(t) = P(t) \\ \mu(t) = 0 & \text{if } \sum_i p_i(t) < P(t) \end{cases} \quad (3.11)$$

$$\begin{cases} \psi(t) > 0 & \text{if } p_i(t) = 0 \\ \psi(t) = 0 & \text{if } p_i(t) > 0 \end{cases} \quad (3.12)$$

and also there is a terminal constraint:

$$H(\vec{x}(t), \vec{p}(t), t)|_{t^F} = 0 \quad (3.13)$$

Appendix A uses this basic result to prove the following theorem. A key quantity, we will see, is the *marginal speedup*, which we define as the partial derivative of the speedup function with respect to the processing power, $\partial f_i(p_i)/\partial p_i$.

Theorem 3.4.1 (Strictly Increasing Speedups) *Suppose that all $f_i(p_i)$ are strictly increasing, $\frac{\partial f_i(p_i)}{\partial p_i} \geq \delta > 0$, for all $p_i \geq 0$, where δ is an arbitrary small positive constant. Then the optimal solution always uses all processors, i.e.,*

$$\sum_i p_i(t) = P(t)$$

Furthermore, if two tasks i and j are scheduled to run in parallel, with non-zero processor power $p_i(t) > 0$, $p_j(t) > 0$ at time t, and neither is within ϵ of completion, and if all predecessors of both i and j have finished, so that $g_i(\vec{x}) = g_j(\vec{x}) = 1$, then the ratio of marginal speedups is fixed:

$$\lambda_i \frac{\partial f_i(p_i)}{\partial p_i} = \lambda_j \frac{\partial f_j(p_j)}{\partial p_j} = \mu(t)$$

where $\lambda_i = \lambda_i(0)$ and $\lambda_j = \lambda_j(0)$ are the initial values of the Lagrange multipliers.

In other words, if more processing power becomes available during execution, it is applied to the runnable tasks in such a way that the marginal speedups of the running tasks maintain their fixed ratios.

3.5 Speedup Function p^α

The optimal control solution takes a particularly elegant form when the speedup function is:

$$\dot{x}_i(t) = g_i(\vec{x})p_i^\alpha(t) \qquad \text{where } 0 < \alpha < 1 \tag{3.14}$$

Note that the exponent α has to be *identical* for all tasks. It is true that this function is not physically realizable for $0 < p_i(t) < 1$, since $p_i^\alpha(t)$ is supralinear in this range. Furthermore, $\partial f_i(\vec{x},\vec{p},t)/\partial p_i \to \infty$ as $p_i(t) \to 0$, so the marginal speedup with $p_i(t) \approx 0$ is asymptotically infinite. Nevertheless, this is an interesting function to study, as it is not unreasonable for $p_i(t) \geq 1$, and the solution is comparatively simple yet interesting. Appendix B proves the following.

Theorem 3.5.1 (p^α Dynamics) *With $f_i(\vec{x},\vec{p},t)$ given by (3.14), the optimal scheduling solution satisfies:*

1. $\lambda_i(t) = 0$ *for* $t \geq t_i^F$.

2. *Once a process is runnable, it is assigned non-zero processor power until it finishes,* $p_i(t) > 0$ *for all* $t_i < t < t_i^F$. *Otherwise,* $p_i(t) = 0$ *for* $t < t_i$ *and* $t > t_i^F$.

3. $\mu(t) = \dfrac{\alpha}{P(t_F)} \left[\dfrac{P(t_F)}{P(t)}\right]^{1-\alpha}$

4. $p_i(t) = \dfrac{P(t)}{P(t_F)} \left[-P(t_F)\lambda_i(t)g_i(\vec{x}(t))\right]^{1/(1-\alpha)}$

5. $\dfrac{d}{dt}\left(\dfrac{p_i(t)}{P(t)}\right) = -\sum_{j\in\Omega^i} \nu_{ij}(t) + \sum_{k\in\Omega_i} \nu_{ki}(t)$ *where* $\nu_{ij}(t)$ *can be interpreted as a flow of processing power from task* i *to a successor task* j*, with* $\nu_{ij}(t) = 0$ *except when* i *and every other predecessor of* j *are within* ϵ *of finishing.*

6. *When task* i *finishes, either* $t_i^F = t^F$ *and the entire graph is finished, or else all the processing power originally allocated to* i *is reallocated to the successors of* i *which begin at the same time that* i *ends.*

Although the formulas in this theorem look obscure, they are extremely powerful. In particular, with p^α dynamics, we can treat processing power as if it were electric charge and precedence constraints as if they were wires. Nodes with no predecessors are initially allocated a given amount of processor "charge"; nodes with predecessors are given no initial processor power. The processing power allocated to an initial node does not change until the node is within ϵ of finishing. At this time, the processor "charge" flows out of the node, along the "wires" leading out of the node, and into successors of the node. Not all successors of the node get this processor charge - only the ones which become enabled to run at this time because all of their predecessors are finishing. As these tasks finish, their processor power is released and pours into their enabled successors. This continues until finally the tasks with no successors are all running and they all complete at the same moment, t^F.

A simple consequence is that whenever any task i completes, either all tasks are finished, $t_i^F = t^F$, or else at least one successor is enabled to run. Another consequence is that the optimal schedule is non-preemptive. The fraction of the available processing power assigned to a task can only change at the beginning and the end of the task.

3.5.1 Series Tasks

This analogy of scheduling to electrical networks allows us to quickly derive optimal schedules without solving explicitly for all the Lagrange multipliers, and without having to compute the precise behavior during the transitional periods as one task finishes and another begins. To illustrate the power of this theorem, consider tasks which are in "series". We define tasks $1, \ldots, K$ to be in series if the only successor of k is $k+1$ for $k = 1, \ldots, K-1$, and the only predecessor of k is $k-1$ for $k = 2, \ldots, K$. By the theorem above, all processing power allocated to task k is reallocated to task $k+1$ when k finishes. Task $k+1$ starts execution immediately after k finishes.

Let ϕ_1 represent the total fraction of processing power poured into 1. If 1 has no predecessors, then ϕ_1 is just the initial fraction of processing power allocated to the task,

$\phi_1 = p_1(0)/P(0)$. Otherwise, we define

$$\phi_1 = \int_0^{t^F} \sum_{j \in \Omega_1} \nu_{j1}(t) dt$$

as the total fraction of processing power poured into 1 from its predecessors. The theorem implies that $\phi_k = \phi_1$ for all $k = 1, \dots, K$.

3.5.2 Parallel Tasks

Another special situation which is simplified by this theorem is when tasks are in "parallel". We define tasks $1, \dots, K$ to be in parallel if they all have the same predecessors and the same successors, $\Omega_k = \Omega_1$ and $\Omega^k = \Omega^1$ for $k = 1, \dots, K$. We can easily show that all these tasks in the parallel set must begin at exactly the same time, and must complete at about the same time. To do this, suppose task 1 is enabled to begin at time t_1. Then either 1 has no predecessors, in which case $t_1 = 0$, or else the last predecessor in the set Ω_1 is within ϵ of completion at time t_1. But in either case, all the other tasks $2, \dots, K$ in the parallel set must be enabled to run at the same time, $t_k = t_1$, since they all have the same predecessors.

Similarly, suppose task 1 finishes at time t_1^F. Either 1 has no successors, in which case $t_1^F = t^F$, or else at least one of its successors has become enabled to run. But if 1 has no successors, then neither do $2, \dots, K$, and so all must finish together at time t^F. If 1 has a successor and that successor is enabled to run when 1 finishes, then this implies that all the predecessors of that successor must be finishing at this time. In particular, tasks $2, \dots, K$ must be finishing. Thus all tasks in the parallel task set must finish at approximately the same time. (More precisely, $t_k^\epsilon < t_m^F$ for all $k, m = 1, \dots, K$.)

3.5.3 Homogeneity

An important property of systems with p^α dynamics is one which relates to the homogeneity of the speedup function. Appendix C proves the following theorem:

Theorem 3.5.2 (Homogeneity) *Assume the speedup functions are p_i^α. Suppose that $P(t)$ total processing power is available, and that the optimal scheduling solution to a graph is $p_i(t)$, with resulting states $x_i(t)$. Now suppose that $\tilde{P}(t)$ total processing power were available instead. Then the optimal scheduling solution $\tilde{p}_i(t)$ for the new situation is given by:*

$$\tilde{p}_i(\tilde{t}) = \frac{p_i(t)}{P(t)}\tilde{P}(\tilde{t})$$

for all i, with resulting states:

$$\tilde{x}_i(\tilde{t}) = x_i(t)$$

where $\tilde{t}$ is a new time variable which is a monotonically increasing function of t, $\tilde{t} = \phi(t)$, defined by:

$$dt = \frac{\tilde{P}^\alpha(\tilde{t})}{P^\alpha(t)}d\tilde{t}, \qquad \tilde{t}\Big|_{t=0} = 0$$

In other words, the available processing power $P(t)$ does not affect the structure of the optimal scheduling solution. Tasks start and complete in exactly the same order, the same fraction of processing power is allocated to the tasks, and the states evolve along the same trajectory, regardless of the power $P(t)$. The only effect of changing $P(t)$ is to effectively warp the time axis, speeding up or slowing down the optimal schedule.

3.6 Simplified Scheduling Algorithm

If we are willing to simplify the treatment of the startup and ending transients of the tasks, then very simple rules can be devised for computing the optimal schedule. In particular, let us approximate ϵ as extremely small, so that for all practical purposes tasks start up and wind down in a negligible amount of time. This implies that

$$g_i(\vec{x}(t)) \approx \begin{cases} 0 & \text{for } t < t_i \\ 1 & \text{for } t > t_i \end{cases}$$

This in turn implies that the optimal schedule for the processing assignments assigns an approximately constant fraction ϕ_i of the available processing power to task i while it runs:

$$p_i(t) = \begin{cases} \phi_i P(t) & \text{for } t_i < t < t_i^F \\ 0 & \text{else} \end{cases}$$

where

$$\phi_i = \left(-\lambda_i P^\alpha(t^F)\right)^{1/(1-\alpha)}$$

Furthermore, the optimal state trajectory is:

$$x_i(t) = \begin{cases} 0 & \text{for } t < t_i \\ \phi_i^\alpha \int_{t_i}^t P^\alpha(\tau)d\tau & \text{for } t_i < t < t_i^F \\ L_i & \text{for } t > t_i^F \end{cases}$$

Specifying an optimal schedule is thus equivalent to specifying the fractions ϕ_i of processing power to be allocated to each task while it runs. Our previous theorem implies that when task i completes at time t_i^F, its fractional power ϕ_i is redistributed to the successors of i which start at this time t_i^F. We will see that this property leads to some interesting interpretations of the optimal solution under p^α dynamics.

3.6.1 Series and Parallel Decompositions

If the task graph can be recursively decomposed into parallel and series tasks, then we will show that there is a very simple solution for the optimal schedule under p^α dynamics.

Suppose tasks $1, \ldots, K$ are a series task set. We will replace this task set with a single composite task, which we refer to as $1:K$. This task will have the same predecessors as 1, $\Omega_{1:K} = \Omega_1$, the same successors as K, $\Omega^{1:K} = \Omega^K$, and its state will summarize the state of all the tasks in the series set. To do this, define:

$$x_{1:K}(t) = \sum_{k=1}^{K} x_k(t)$$

The initial and terminal values of the state are:

$$x_{1:K}(0) = 0, \qquad x_{1:K}(t^F) = L_{1:K}$$

where we define the length of the composite series task to be:

$$L_{1:K} = \sum_{k=1}^{K} L_k$$

Let $p_{1:K}(t)$ represent the total amount of processing power allocated at time t to all subtasks of the composite task:

$$p_{1:K}(t) = \sum_{k=1}^{K} p_k(t)$$

Then because the tasks in the series run sequentially, and because all are assigned the same fractional power, we have:

$$p_{1:K}(t) = \begin{cases} \phi_{1:K} P(t) & \text{for } t_{1:K} < t < t_{1:K}^F \\ 0 & \text{else} \end{cases}$$

where we define $t_{1:K} = t_1$ as the time the first task in the series starts, and define $t_{1:K}^F = t_K^F$ as the time the last task in the series finishes, and where

$$\phi_{1:K} = \phi_1 = \cdots = \phi_K$$

It is easy to show that the composite state obeys dynamics:

$$\dot{x}_{1:K}(t) = \begin{cases} p_{1:K}^{\alpha} & \text{for } t_{1:K} < t < t_{1:K}^F \\ 0 & \text{else} \end{cases}$$

Thus the state of this composite task obeys exactly the same dynamics as the state of the original subtasks. It is clear that an optimal schedule for the original graph maps exactly into an optimal schedule for a new graph with the series set replaced by this single composite task. Thus we can solve the simpler scheduling problem with the composite task, then derive the optimal schedule for the original graph by allocating the composite power $p_{1:K}(t)$ to the individual tasks, $p_i(t)$.

Parallel task sets can be dealt with in a similar manner. Let $1, \ldots, K$ be a parallel task set; each task has the same predecessors and the same successors. As discussed before, all tasks must begin and end at the same time in the optimal schedule:

$$t_1 = \cdots = t_K, \qquad t_1^F = \cdots = t_K^F$$

But since each task is allocated a fixed fraction ϕ_i of the available power, this implies that:

$$L_i = \int_{t_i}^{t_i^F} (\phi_i P(t))^\alpha \, dt$$

or

$$\phi_i = \left(\frac{L_i}{\Delta(t_i, t_i^F)} \right)^{1/\alpha}$$

where we define

$$\Delta(t_i, t_i^F) = \int_{t_i}^{t_i^F} P^\alpha(t) dt$$

Now let us replace the parallel task set $1, \ldots, K$ with a single composite task $1 : K$. This composite task will have the same predecessors and the same successors as the subtasks, and it will start and stop at the same time:

$$t_{1:K} = t_1 = \cdots = t_K, \qquad t_{1:K}^F = t_1^F = \cdots = t_K^F$$

The processing power allocate to the composite task is just the sum of the power allocated to the individual tasks:

$$p_{1:K}(t) = \sum_{k=1}^{K} p_k(t) = \phi_{1:K} P(t)$$

where

$$\phi_{1:K} = \sum_{k=1}^{K} \phi_k$$

Now we define the state of the composite task as $x_{1:K}(t)$, and give it dynamics:

$$\dot{x}_{1:K}(t) = \begin{cases} p_{1:K}^{\alpha}(t) & \text{for } t_{1:K} < t < t_{1:K}^F \\ 0 & \text{else} \end{cases}$$

with boundary conditions

$$x_{1:K}(0) = 0, \qquad x_{1:K}(t^F) = L_{1:K}$$

where the length of the composite task, $L_{1:K}$, is the $1/\alpha$ norm of the lengths of the subtasks:

$$L_{1:K} = \ell_{1/\alpha}(L_1, \ldots, L_K)$$

where we define:

$$\ell_{1/\alpha}(L_1, \ldots, L_K) = \left(\sum_{k=1}^{K} L_k^{1/\alpha} \right)^{\alpha}$$

It is straightforward to show that with this composite processor assignment, the state of the composite task satisfies:

$$x_{1:K}(t) = \frac{L_{1:K}}{L_k} x_k(t) \qquad \text{for } k = 1, \ldots, K$$

The composite task satisfies the same dynamics as the original subtasks. If we replace the parallel task set with composite task $1 : K$, therefore, an optimal scheduling solution for the original graph maps exactly into an optimal scheduling solution for the simplified graph, and vice versa. In particular, if we solve for the optimal solution for the simplified graph, then the solution for the processing power to be assigned to the individual subtasks can be computed in terms of the fraction $\phi_{1:K}$ of power assigned to the parallel set:

$$t_k = t_{1:K}, \qquad t_k^F = t_{1:K}^F$$

$$p_k(t) = \phi_k P(t) \qquad \text{where } \phi_k = \phi_{1:K} \frac{L_k^{1/\alpha}}{L_{1:K}^{1/\alpha}}$$

The fraction of power allocated to the parallel task set is thus split up among the subtasks according to the length of each subtask, in such a way that all the subtasks will start and finish simultaneously.

Because the composite series or parallel task has the same p^α dynamics as the original tasks, we can apply this grouping technique recursively. If the graph can be expressed entirely in terms of parallel and series configurations of tasks, then we eventually reduce the entire graph to a single task with p^α dynamics. Now the run time of the graph is easily computed. Undoing the recursion, we allocate the processing power to each of the series and parallel components according to their lengths, and determine their start and stop times. Continuing recursively, we eventually derive the optimal processing schedule for each individual task.

Typical graph structures that can be recursively decomposed into parallel and series components include trees, inverted trees, and forests of trees or inverted trees. Most matrix expression computations, for example, have the form of an inverted tree, and thus if we are willing to approximate the speedup of the individual matrix operations as p^α, then the expression can be optimally scheduled by this simple parallel/series trick.

3.6.2 Examples of Optimal Schedules

In this section several examples will be presented to illustrate the power of the ideas above. All graphs will be assumed to obey p^α dynamics. Under p^α dynamics, optimal schedules are non-preemptive in nature, and both the series and the parallel reductions of Section 3.6.1 can be applied.

3.6.3 Trees and Forests

Trees and forests (also inverted trees and forests) of tasks are uniquely decomposable into series-parallel combinations. Hence optimal schedules can be written down by inspection. Two examples are given below.

Tree Schedule

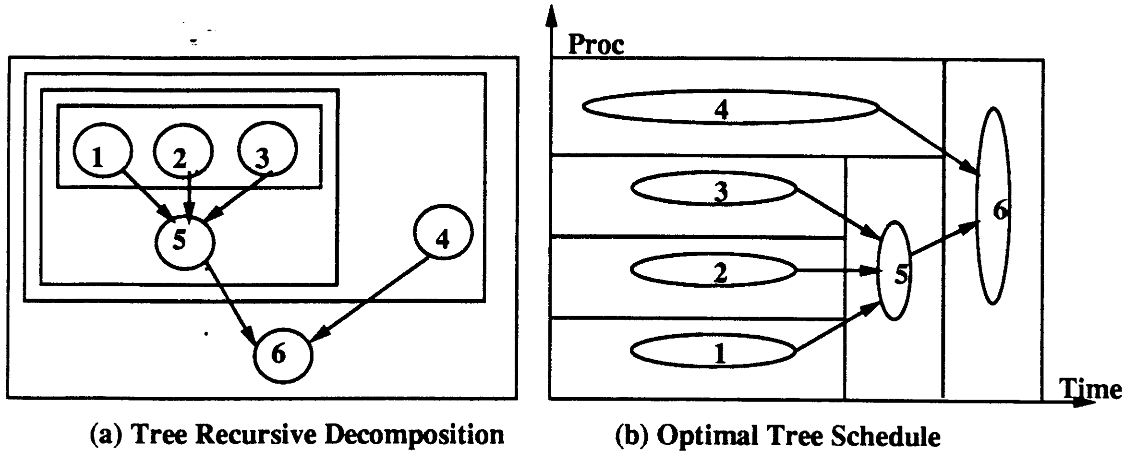


Figure 3.2: Optimal Tree Scheduling using series-parallel reductions

Figure 3.2 (a) shows a tree-structured task system, together with the unique recursive series-parallel decomposition. The equivalent task has length

$$\tilde{L} = \ell_{1/\alpha}\left(\ell_{1/\alpha}(L_1, L_2, L_3) + L_5, L_4\right) + L_6$$

and the optimal time t^F is such that

$$\int_0^{t^F} P^\alpha(t)dt = \tilde{L}$$

The optimal schedule is shown in Figure 3.2 (b), and mimics the decomposition.

Inverted Tree Schedule

Figure 3.3 shows an inverted tree-structured task system, together with the unique recursive series-parallel decomposition. The equivalent task has length

$$\tilde{L} = \ell_{1/\alpha}\left(\ell_{1/\alpha}(L_4, L_5) + L_2, L_3\right) + L_1$$

and the optimal time is such that

$$\int_0^{t^F} P^\alpha(t)dt = \tilde{L}$$

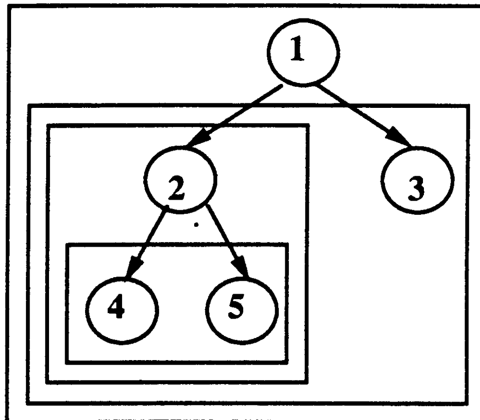


Figure 3.3: Optimal Inverted Tree Scheduling using series-parallel reductions

3.6.4 General Directed Acyclic Graphs (DAG's)

In general, DAG's cannot be uniquely decomposed into series-parallel graphs. A strategy which will work in all cases is to try all possible decompositions, and take the decomposition which yields the minimum time.

Figure 3.4 shows a task system structured as a DAG, together with three possible recursive series-parallel decompositions. The equivalent tasks have length

$$\begin{aligned}
\tilde{L_1} &= \ell_{1/\alpha}\left[(L_1 + L_3), \left(\ell_{1/\alpha}(L_4, L_5) + L_2\right)\right] \\
\tilde{L_2} &= \left[\ell_{1/\alpha}(L_1, L_2) + \ell_{1/\alpha}(L_3, L_4, L_5)\right] \\
\tilde{L_3} &= \ell_{1/\alpha}\left[(L_2 + L_5), \left(\ell_{1/\alpha}(L_3, L_4) + L_1\right)\right]
\end{aligned}$$

and the optimal time is such that

$$\int_0^{t^F} P^\alpha(t)dt = \min(\tilde{L_1}, \tilde{L_2}, \tilde{L_3})$$

Which of the three decompositions is optimal depends on relative task sizes. These determine whether task 1 finishes ahead, simultaneously with, or after, task 2. Since under p^α dynamics a successor has to start as soon as a task finishes, the relative finishing time of tasks 1 and 2 determines which of tasks 3 and 4 "pairs" with task 1 to form a

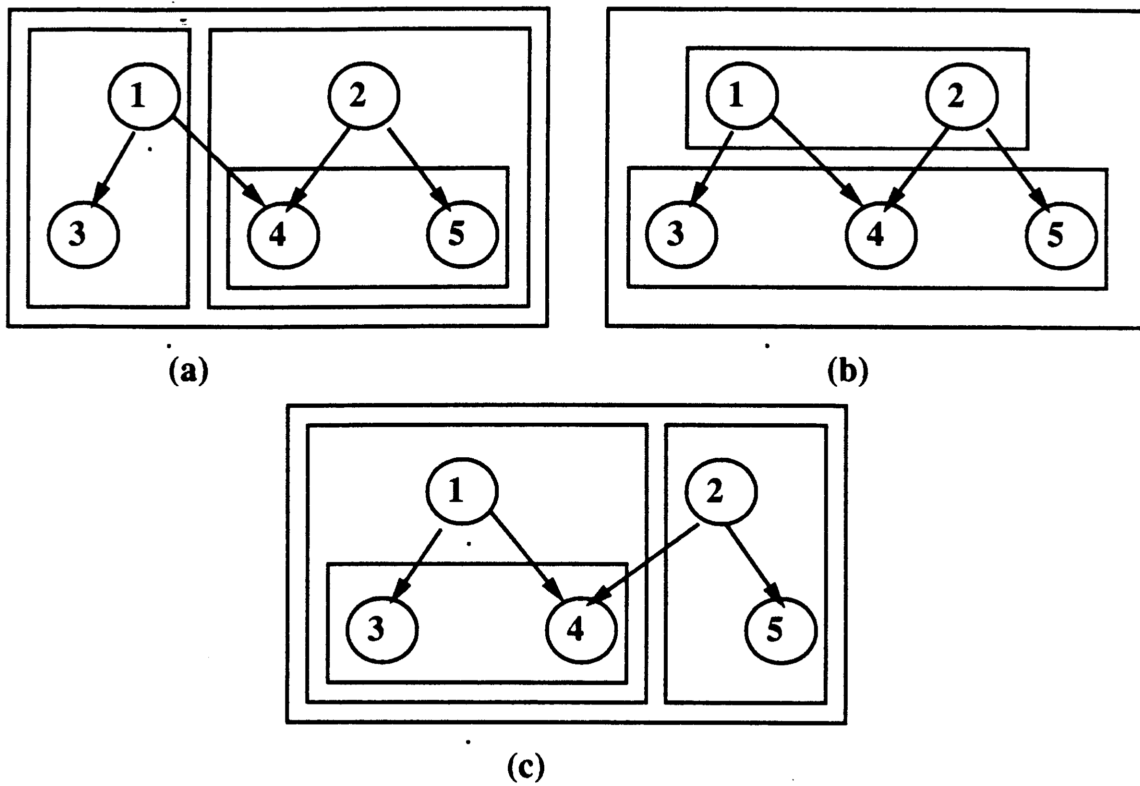


Figure 3.4: Optimal DAG Scheduling using series-parallel reductions

series combination. In general, all possible decompositions may not yield self-consistent results; the relative finishing times computed from the decomposition may be different from those assumed for the decomposition. Such decompositions have to be dropped from consideration.

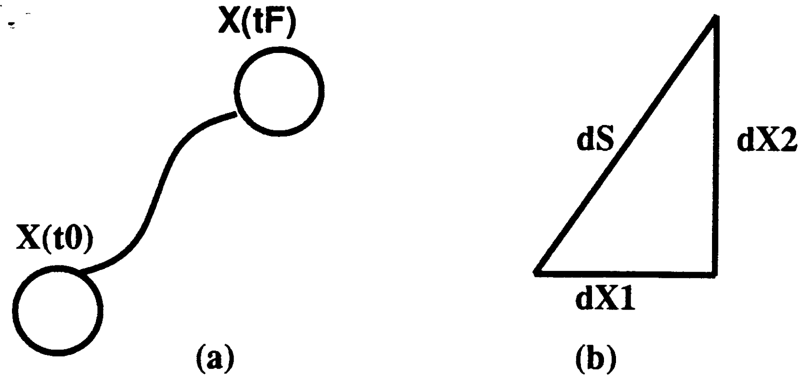


Figure 3.5: Scheduling and Shortest Path Problems

3.7 p^α Dynamics and Shortest Path Scheduling

The optimal control formulation of the scheduling problem derives the control functions (the processing powers $p_i(t)$) which drive the state trajectory along a path which goes from the initial to the terminal states as quickly as possible, subject to the precedence constraints. We will show that under p^α dynamics, the time-optimal state trajectory is the path with the shortest length, as measured by an $\ell_{1/\alpha}$ norm.

Theorem 3.7.1 (Scheduling and Shortest Path Problems) *Approximate ϵ as very small, so that transients during the start and finish of tasks can be neglected. Suppose $f_i(p_i) = p_i^\alpha$ for $0 < \alpha < 1$. Then the scheduling problem is equivalent to a shortest path problem with $\ell_{1/\alpha}$ norms.*

Proof: We demonstrate this result by explicitly computing the time taken by a schedule specified by a trajectory in task space. It is convenient to switch to a normalized time variable. Let $\tilde{t}$ be a monotonically increasing function of t, $\tilde{t} = \phi(t)$, defined by:

$$d\tilde{t} = P^\alpha(t)dt, \qquad \text{with } \tilde{t}\Big|_{t=0} = 0$$

Because $\tilde{t}$ is a monotonically increasing function of t, the schedule which completes in minimal time t must also complete in minimal time $\tilde{t}$. If the total processing power is constant, $P(t) = P$, then $\tilde{t}$ is simple a scaled version of time t.

With reference to Figure 3.5 (a), the time (measured in the $\tilde{t}$ frame) taken for a schedule specified by the state trajectory S is

$$\int_S d\tilde{t} = \int_S \frac{d\tilde{t}}{dS} dS = \int_S \frac{d\tilde{t}}{dt}\frac{dt}{dS} dS$$

where dS is the path element. The time taken for moving along dS can be computed as follows. Let

$$dS = [dx_1, dx_2, \cdots dx_N]$$

(Figure 3.5 (b)). Then a motion along dS is equivalent to processing dx_1 of the first task, dx_2 of the second, and so on, *simultaneously*. Thus the elapsed time dt is the same for all of them. If the number of processors allocated to the ith task is $p_i(t)$, then we have

$$dt = \frac{dx_1}{p_1^\alpha(t)} = \frac{dx_2}{p_2^\alpha(t)} = \cdots = \frac{dx_N}{p_N^\alpha(t)}$$

or

$$p_k(t) = \left(\frac{dx_k}{dt}\right)^{1/\alpha}$$

Since

$$\sum_{i=1}^{N} p_i(t) = P(t)$$

we can simplify this to

$$dt = \frac{\left(\sum_{i=1}^N dx_i^{1/\alpha}\right)^\alpha}{P^\alpha(t)} = \frac{\ell_{1/\alpha}(dx_1, dx_2, \cdots, dx_N)}{P^\alpha(t)}$$

or

$$d\tilde{t} = P^\alpha(t)dt = \ell_{1/\alpha}(dx_1, dx_2, \cdots, dx_N)$$

which is the $\ell_{1/\alpha}$ norm of the path increment in task space. Thus the normalized elapsed time $\tilde{t}$ is a *metric* in task space. A time-optimal state-trajectory (optimal schedule) must be time-optimal even if time is measured with our normalized time variable $\tilde{t}$. Thus the time-optimal path must also be the shortest path in task space, as measured by the $\ell_{1/\alpha}$ norm, subject to precedence constraints between tasks. If $\alpha = 1/2$, the time optimal schedule minimises the Euclidean Distance.

The precedence constraints behave like *obstacles* to the trajectory, forcing the path to reach certain hyperplanes (i.e. having a task finish, $x_i = L_i$) before it can veer off in certain directions (by having one or more successor tasks begin.)

Theorem 3.7.2 (Limiting Cases of P^α) *For any system of tasks with $f_i(p) = p^\alpha$ dynamics, the time for an optimal schedule with $P(t)$ processors total satisfies the following relations.*

$$\lim_{\alpha \to 0} t^F = CP$$

where CP is the critical path length of the precedence graph of the task system.

$$\lim_{\alpha \to 1} \int_0^{t^F} P(t)dt = \sum_{i=1}^{N} L_i$$

Proof: As α tends to 0, p_i^α tends to 1. This implies that the rate of processing for each task is always unity, *regardless* of its processor allocation (even if it is infinitesimal). Hence the time taken for the ith task is L_i. Hence the optimal time taken for the complete task system, t_F is clearly the longest chain of tasks in the precedence graph, viz. CP.

As α tends to 1, p_i^α tends to p_i. Hence each task exhibits linear speedup. Since all available processing power is used with perfect efficiency, the total work done by all the processors over the time t^F, $\int_0^{t^F} P(t)dt$, must simply equal the total work for the graph, $\sum L_i$.

3.8 Discussion and Summary

In this chapter we have discussed multiprocessor scheduling theory from the viewpoint of optimal control theory. Each task is treated as as dynamic system, whose state can be changed by applying processing power to it. We assumed that each task could be decomposed into an arbitrary number of modules to be executed in parallel on multiple processors, and that the number of processors allocated could be a continuous variable. Under these assumptions, the scheduling problem was formulated as a time-optimal control problem. Under general conditions, the formulation yielded a number of powerful and elegant theorems. In the special case of p^α dynamics, the scheduling problem was shown to be equivalent to a shortest path with obstacles problem with $\ell_{1/\alpha}$ norms. Moreover, in this special case, non-preemptive schedules were shown to be optimal over the entire class of preemptive schedules. These results greatly simplify the scheduling problem. Closed form optimal solutions for trees and forests have been derived. The results can also be extended to guide the scheduling of general Directed Acyclic Graphs (DAG's).

In conclusion, our approach to optimal scheduling provides fundamental insights into scheduling theory. Our approach has revealed deep connections between scheduling and shortest path problems. The methodology is particularly appropriate for applications such as matrix expression computation where tasks can be systematically decomposed into arbitrary numbers of parallel modules. For these applications, our theorems are quite powerful and the optimal algorithms are practical and easily applied. Furthermore, we believe that these scheduling approaches should be near optimal for systems whose dynamics are similar to, but not exactly p^α.

Chapter 4

Experimental Results

4.1 Introduction

In this chapter we present experimental results obtained from a prototype compiler, the Structure Driven Compiler (SDC), which was written to test the ideas of the thesis. The compiler uses the two level hierarchical compilation paradigm for matrix expressions. The prototype demonstrates that hierarchical compilation is a fast strategy, and produces relatively efficient code.

To recapitulate, the hierarchical compilation strategy works as follows. Efficient parallel library routines for the basic matrix operators are efficiently spliced together to form good routines for the complete matrix expression. The parallel library routines for the matrix operators exploit the geometric structure of the operator dataflow graphs, as explained in Chapter 2. The routines are spliced together using techniques explained in Chapter 3.

In this chapter, compilation of a variety of matrix expressions is illustrated, varying in complexity from simple matrix operators to complex inner loops of linear algebra routines. First, in Section 4.2, we describe the experimental facilities used to obtain our results. Section 4.3 discusses some general issues in compiling static dataflow graphs like those encountered in matrix expressions. Then the design of the parallel operator library,

(Section 4.4) and the generalised scheduling heuristics (Section 4.5) is described. The implementation details of the compiler SDC follow (Section 4.6). Finally, a variety of matrix expression examples (Section 4.7) are illustrated. Future compiler enhancements are described in Section 4.8.

4.2 Experimental Environment

The input to SDC is a matrix expression in a LISP like prefix language, with data independent control. SDC produces Mul-T output code, which is compiled and run on the MIT Alewife Machine simulator.

4.2.1 Mul-T - a Parallel LISP

Mul-T is a parallel version of LISP (SCHEME to be precise). A task T is created using a *(future T)* call. A future call specifies a task which can be run at some unspecified (later) time. A task T can be forced to finish using a *(touch T)* call. A touch call returns only when the touched task completes. The *future-touch* mechanism enables parallel tasks to be created and completed as necessary. No explicit control is provided to the programmer to schedule tasks at specific times. Tasks are scheduled at times when processor resources are available (lazy futures). However, a task T can be assigned to a specific processor P, using the *(future-on P T)* call.

The Mul-T library provides various forms of task synchronization. The *touch* call is one such. Other forms include *semaphores*, *barriers*, *readers-writers* locks, etc.

Mul-T provides a *shared memory abstraction* for handling interprocess communication.

4.2.2 The MIT Alewife Machine

The MIT Alewife Machine [Aga90] (Figure 4.1) is a 2-D / 3-D mesh connected coarse threaded multiprocessor, with strongly coherent caches. The basic system paradigm is

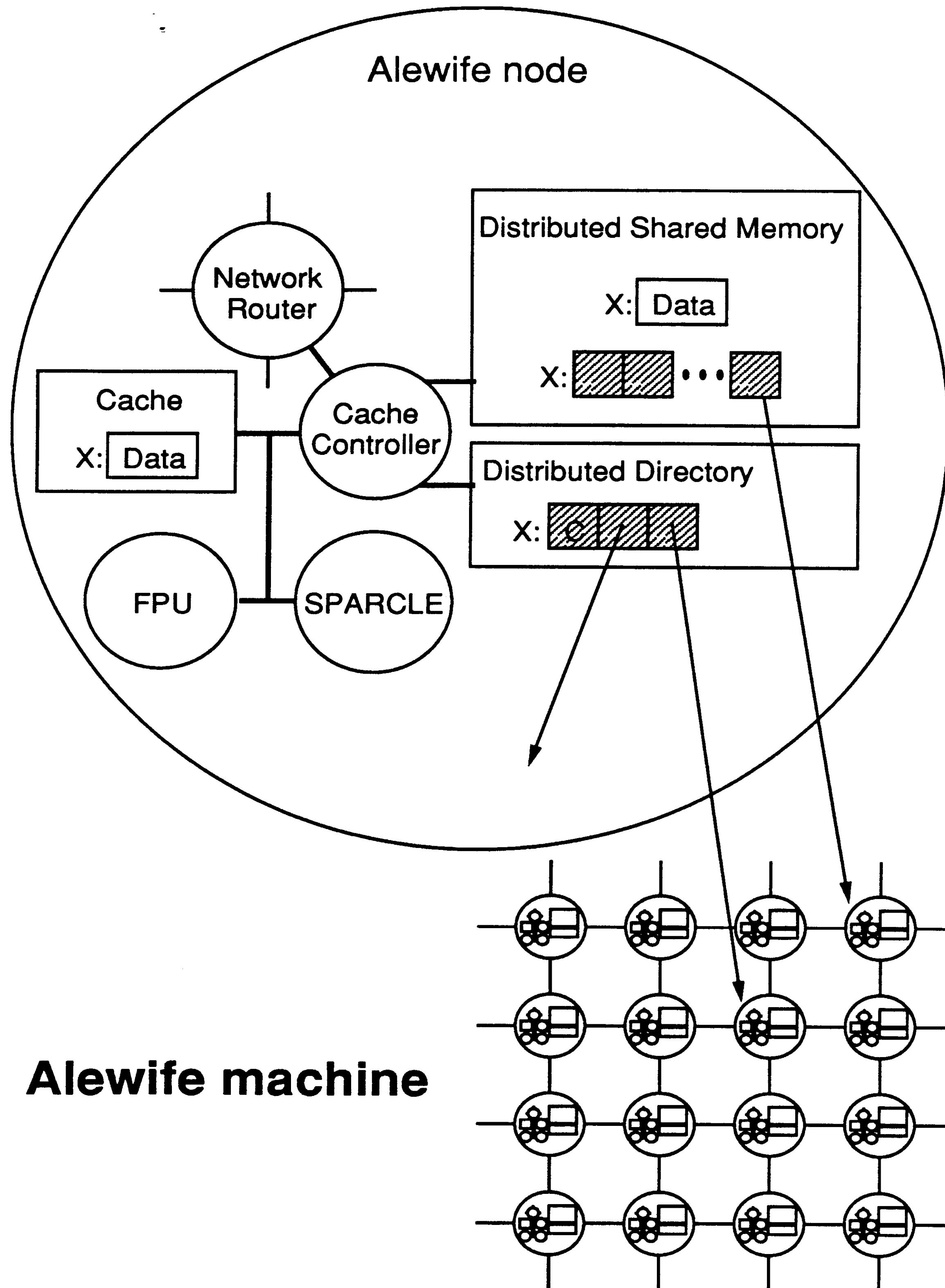


Figure 4.1: The MIT Alewife Machine

high performance through simple hardware being controlled by sophisticated software - operating systems and compilers.

The processors are connected to each other via a 2-D or 3-D mesh connected interconnection network. Each processor has associated local memory. There is no explicit global memory. As such this is a strictly distributed memory system. Processors have associated caches for fast access to frequently used data. The existence of this distributed memory hierarchy implies that locality issues are critical in determining program performance. At the time of writing, only a simulator ASIM is available for Alewife.

Extensive software instrumentation is available for ALEWIFE. The primary source language is Mul-T [KHM89]. A Mul-T program can be compiled, linked, loaded, and run on the simulator. Extensive software exists [Alent] for collecting program statistics, like total run time, the parallelism profile, memory access patterns, cache coherency transactions, etc. This software is critically important to evaluate program performance.

At an early state of the research, an Encore Multimax was used for the simulations. However, the almost complete lack of instrumentation makes the results obtained on it almost worthless.

4.2.3 Alewife Machine Model

The MIT Alewife machine (Figure 4.1) is being developed to investigate issues of program locality in multiprocessors. There are many aspects in which the model differs from the simple multiprocessor model in Chapter 2.

Firstly, the processor at each node (SPARCLE) of the machine is more complex than the simple model we have assumed, with a diverse instruction set. It is a load-store architecture, however. So, our assumption that the arithmetic operations (+,*) are characterised by execution times, ignoring pipelining, latency, etc., is reasonably accurate.

Next, the interconnection network is quite different from the simple fully connected network assumed in the model. The mesh connectivity implies processor locality. Com-

munication times from one processor to another depend on the distances between the processors. They are no longer uniform, as assumed in the model.

Task placement provides a means of overcoming the nonuniform communication time. Macro tasks which share data should be preferably placed on a set of processors near each other. Then most communication becomes local, and communication times are roughly constant.

We have assumed a 3-D mesh connected interconnection network, for all our simulations, because its higher connectivity makes it closer to the fully connected network in the model.

However, the statistical nature of the network is another source of mismatch. The communication times between the same two processors can vary widely depending on network loading conditions, hotspotting, etc. This has not been modeled in any manner in our simple model. The network will have to be operated at a relatively low loading for our model to be valid.

The ALEWIFE machine provides hardware support for cache coherency. Full-map, LimitLESS, and Chained directory schemes [Alent] are available for hardware coherency. The Full-map directory scheme maintains a complete set of hardware pointers to each processor currently caching a particular datum. These pointers are needed for coherence transactions like cache invalidates, data messages, etc. In the LimitLESS protocol, only a few (5) pointers are maintained by hardware, and software emulation accounts for the rest. LimitLESS behaves like Full-map when the sharing is limited, as in most important application programs. The Chained directory scheme (the default) maintains all pointers in hardware, in the form of a linked list distributed across processors. The Chained directory scheme behaves in some ways like LimitLESS.

Hardware caching support is not modeled in our model. In all the simulations, we have used the default Chained directory caching scheme. While the absolute timings using the other caching schemes do differ, the relative performance of our compiled programs is expected to be similar.

When a set of tasks start, considerable overhead is incurred on the Alewife machine. This includes time to spawn all the tasks in a tree like fashion, argument access, etc. This task startup overhead is completely ignored in our model. This overhead can be minimised by minimising the number of tasks which are spawned, a point to which we return later.

It is clear that the MIMD general purpose nature of the MIT Alewife Machine introduces complications which are not modeled adequately in our model. But the model can still be used, since it is simple enough for analysis, and under ideal conditions does represent the Alewife Machine reasonably well. It is not too inaccurate in modeling the processor. Communication in programs with good locality can also be reasonably modeled, especially under conditions of low network load. For programs which do not repeatedly spawn many tasks, the task startup overhead is relatively small.

4.3 General Issues in Compiling Static Dataflow Graphs

The purpose of writing the compiler SDC was to test the issues involved in parallelizing matrix expression code, and to test whether the ideas raised in the thesis were applicable to compiler writers. Before we do this, it is important to specify what the task of every compiler is.

The task of any compiler is two fold - accurate translation and efficient translation.

The first task is *accurate* translation of the source language into object code. This is a *correctness* issue. In the case of matrix expressions, this manifests itself as accurate translation from the matrix expression input language to the target language, in our case Mul-T. This involving developing correct parallel library routines to implement the various matrix operations. Generating correct code for the matrix expression is done by generating Mul-T code containing a sequence of calls to these parallel library routines, respecting all precedence constraints.

The second task is to produce *efficient* code. This is a *performance* issue. This task can be viewed as an exercise in resource optimisation. The compiler has to allocate the limited computation resources, viz. processors, memory, network etc., to optimise some criterion, eg. minimization of finishing time. In general, the more expressive the source language, the harder it is to compile efficiently. In the particular domain of compiling matrix expressions, the parameters of each parallel library routine - viz. the parallelism, various blocking parameters, etc. - have to be specified in an optimal manner.

Optimal processor allocation is typically in the domain of scheduling theory. Since the dataflow graphs in the case of matrix expressions are all static, static scheduling techniques are applicable. Chapter 3 presented a generalisation of (static) scheduling theory to determine both parallelism and sequencing of tasks. Parallelising heuristics arising out of this theory are incorporated in the compiler (Section 4.5).

Memory is needed to hold intermediate results during computation of matrix expressions. Since the dataflow graphs are all static, this memory can be preallocated at compile time. A piece of memory can oftentimes be reused for holding many intermediate results. Optimal memory reuse can be handled by graph colouring techniques. In our prototype compiler, memory allocation is performed dynamically, at run time, by the parallel library routines themselves. This induces unnecessary overhead, which should be removed in later versions.

Optimal allocation of network resources (communication) is a difficult task, as it involves scheduling of essentially random memory requests through the communications network. But the communication needs can be greatly reduced by appropriate task placement. Tasks which share data should preferably be allocated on the same processor, to minimise the communication requirements. Our prototype compiler completely ignores this issue.

Other forms of resource optimization include classical uniprocessor compiler optimizations like loop unrolling, induction variable recognition, code reordering, etc. None of this is handled by the prototype compiler.

To reiterate, the only optimization that the prototype compiler performs is to determine the parallelism of each particular parallel library routine. Memory allocation for each library routine is currently performed dynamically, by the parallel library routines itself. Task placement is completely ignored. All this should be incorporated in future versions of the compiler.

4.4 Parallel Operator Library

Here we discuss the design of the parallel operator library routines. The routines are derived from the structure driven compilation ideas in Chapter 2.

4.4.1 Matrix Sums

To compute $C = A+B$ on P processors, the matrices A and B are divided into rectangular blocks, and a block is handed to each processor. Assume that P can be factored into two roughly equal factors P_1 and P_2. Then our blocking scheme choses to split the rows into P_1 groups, and splits each group into P_2 blocks, yielding $P = P_1 P_2$ blocks in all. Since there is very little sharing in a matrix addition (Chapter 2), the performance is relatively insensitive to the exact blocking scheme chosen. Also, the matrix addition has much fewer computations compared to matrix multiplication, hence optimising it is much less important for most of the examples we tried.

4.4.2 Matrix Products

The rectangular bin-packing ideas of Chapter 2 have been incorporated in the matrix product routines.

Assume that the matrix product $C = A \times B$ is being performed. The bin-packing algorithms essentially specify the dataflow graph block handled by each processor. The task spawned for each processor computes all the partial products inside its block, and sums them. This yields a partially computed block of C, which can be accumulated with

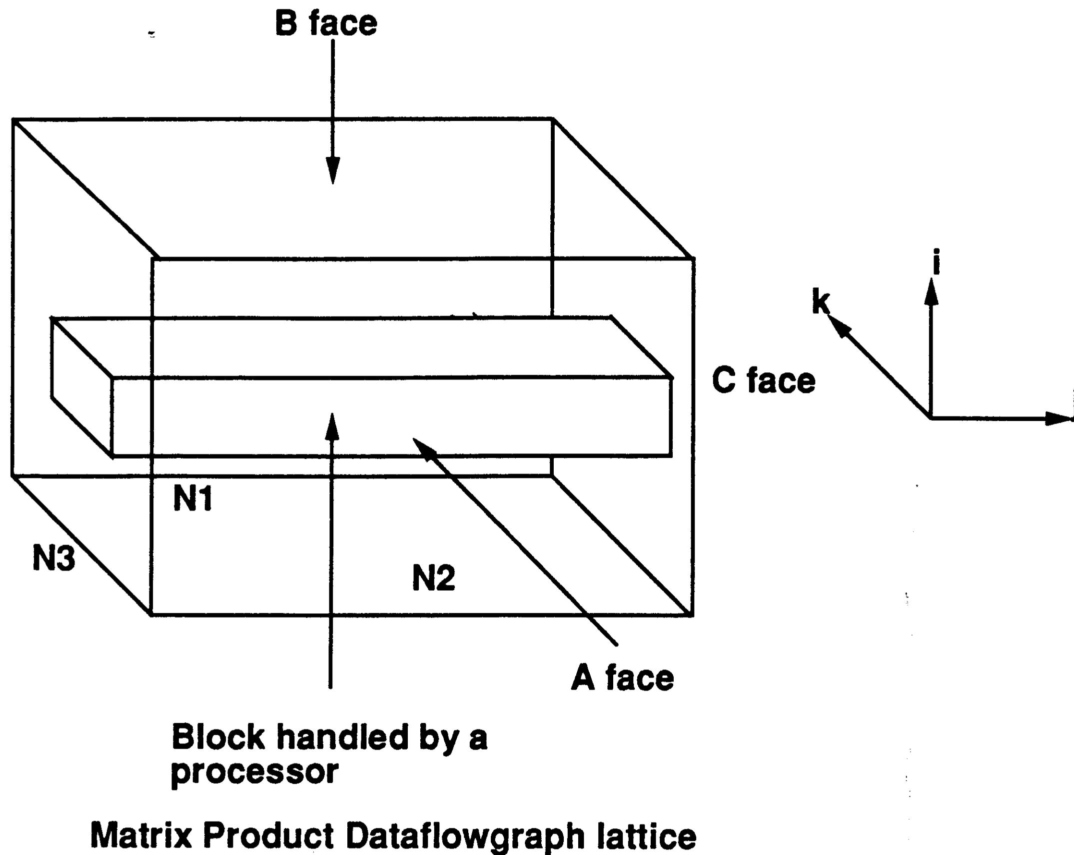


Figure 4.2: Implementation of Matrix Product

other corresponding partially computed blocks of C to form a completely computed block of C.

This accumulation necessitates synchronization between the tasks in general. On the Alewife machine this incurs excessive overhead. For the purposes of this prototype, we have eliminated this source of overhead by computing each of the N^2 dot products completely on a single processor. Thus the dataflow graph is chopped into long rectangular blocks, parallel to the summing (j) axis. (Figure 4.2). Clearly, this introduces more communication, but the increase is not much, if the aspect ratio of the blocks is not very far from ideal.

Specifying that each block has length N_2 along the j axis leaves the block sizes along each of the other dimensions i (N_1) and k (N_3) free to be specified. The sizes along each

of the other dimensions are chosen in a manner analogous to the matrix addition routine.

Specifically, the total number of processors P is factored into two roughly equal factors P_1 and P_3. P_1 is chosen to be the higher factor. If P is a perfect square, $P_1 = P_2 = \sqrt{P}$. If P is prime, $P_1 = P$, and $P_2 = 1$. Cases when the number of rows of C (columns of C) is less than P_1 (P_3) are handled specially - then a refactorization is attempted. We try to factor $P - 1$ instead of P, on the assumption that P itself is a large prime.

A even better factorization strategy would have been to use the optimal rectangle partitioning strategies of Chapter 2 (Section 2.6.2). The two strategies differ significantly only if C is very far from square, or if P does not factor well (a large prime for example).

After the factorization, rows of the output matrix C are partitioned into P_1 sets. Each set is further partitioned into P_3 blocks, yielding $P_1 P_3$ blocks in all. Each block of C is computed on one processor.

The scheme is a compromise between the optimal blocking schemes of Chapter 2, and ease of implementation. The general ideas of Chapter 2, about chunking up the dataflow graph into compact equal sized clusters is used. Only the shapes of the clusters are not the most compact possible, to avoid excessive synchronization overhead.

Speedup curves for the matrix product will be shown later, in Section 4.7.

4.4.3 Other Operators

Various other matrix expression operators have been implemented, following the same basic idea of minimising communication by choosing good block sizes. These operators include dot products, outer products, matrix scaling, transposes, etc. The details are omitted.

4.5 Composing Parallel Operator Library Routines

Once good parallel library routines are developed, algorithms have to be developed for composing them together to yield a good compilation for the matrix expression. This

Operation	Workload
Scalar Operation	1
$N_1 \times N_2$ Addition	$N_1 N_2$
N point Dot Product	N
$N_1 \times N_2$ Outer Product	$N_1 N_2$
$N_1 \times N_2$ Matrix Scale	$N_1 N_2$
$N_1 \times N_2$ Matrix Transpose	$N_1 N_2$
$N_1 \times N_2 \times N_3$ Matrix Product	$N_1 N_2 N_3$

Table 4.1: Workloads of various operators

needs methods to determine the parallelism of each particular routine, viz. the techniques of generalised scheduling outlined in Chapter 3.

The generalised scheduling theory of Chapter 3 yields a number of heuristics to determine the parallelism and scheduling order of each macro-node (task). Under P^α dynamics, several optimal scheduling techniques were obtained. If the dynamics are not P^α, these techniques can be employed as heuristics. These techniques, together with two other similar heuristic techniques, were experimented with. The heuristics are enumerated below:

- The *Naive* heuristic computes each macro-node using all the available processors, in some sequence satisfying the precedence constraints. Since each node is run at the finest granularity (maximum parallelism) possible, this method is clearly slow. But it is included because of its simplicity.

- The *Greedy* heuristic operates in cycles. In each cycle, all the nodes which can be executed (upto a maximum equal to the number of processors), are fired. The processing resource is distributed in a manner so as to equalise the finishing times of all fired nodes. In the next cycle, the set of successor nodes is fired, and so on.

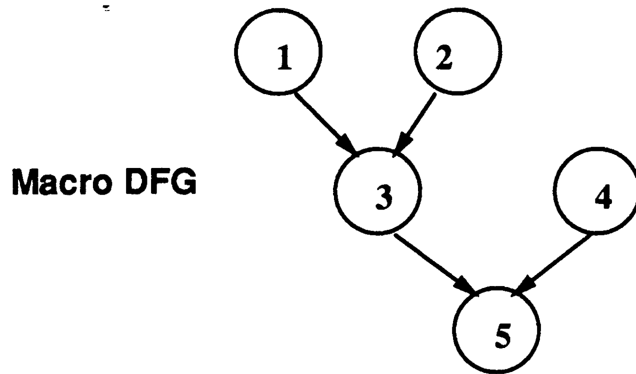
  To equalize the finishing times of all nodes fired in a cycle, the processors are distributed among them in proportion to their workloads. The workloads are estimated crudely, by a simple operation count. The workloads for each particular operation are specified in Table 4.1.

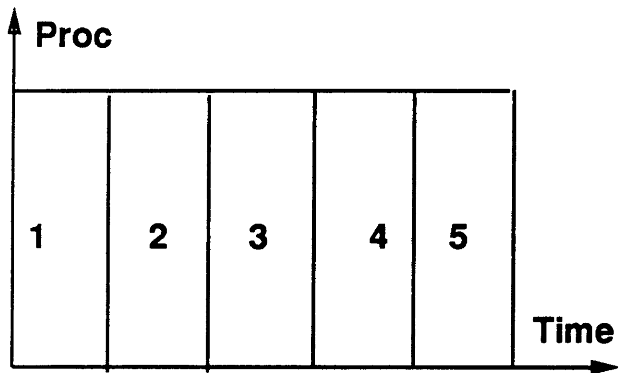
- The *Tree* heuristic, used for tree-structured matrix expressions, is derived from the optimal tree scheduling heuristic for P^α dynamics (Chapter 3). The processor resource is partitioned among sibling subtrees in proportion to their workloads. If all the nodes of each subtree have the same α, the workload of each subtree is the $\ell_{1/\alpha}$ norm of its children, plus the workload of the root (Chapter 3). But for this technique to be useful, α must be known. Also, in general fractional processor allocations result. In this version of the compiler, α is assumed to be unity. The total workload of each subtree is simply the sum of workloads of all its constituent nodes. The processor allocations have also been quantised to avoid fractional processor allocations.

  The overall dataflow graph execution time estimates are clearly extremely erroneous if we assume that $\alpha = 1$, as each task is assumed to have linear speedup. However, in practice this assumption does not cause much error in the *processor allocations.*

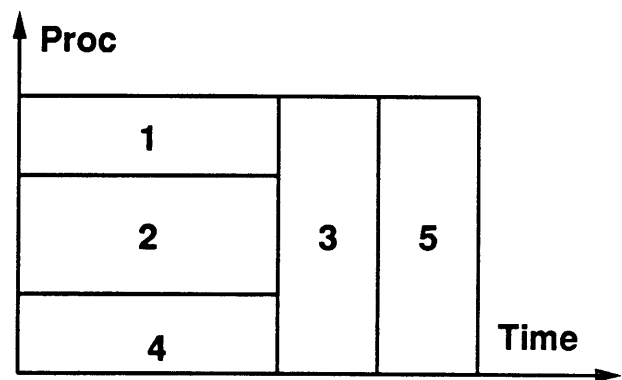
Succeeding versions of the compiler will attempt to partition the processor resource much more accurately, using actual measured operator execution times (Section 4.8).

Figure 4.3 illustrates the operation of the three heuristics on a tree structured dataflow graph. The naive heuristic (Figure 4.3 (a)) runs each macro node on all the available processors. Nodes 1, 2, 3, 4, and 5 are run in sequence. The Greedy heuristic (Figure 4.3 (b)) runs nodes 1, 2, and 4 in parallel, distributing the processor resources among them. Subsequently nodes 3 and 5 are computed, each using all available processors. The Tree heuristic (Figure 4.3 (c)) does a slightly better job of partitioning the processor resource by recognizing the fact that nodes 1, 2, and 3 form a subtree, which can be run in parallel with node 4, by splitting the available processors. Finally node 5 is run on all the available processors. The Tree heuristic outperforms Greedy, since it reduces the parallelism of node 3. The two subtrees, one comprising nodes 1, 2, and 3, and the other comprising the lone node 4 are computed in a "balanced" manner by the Tree heuristic.

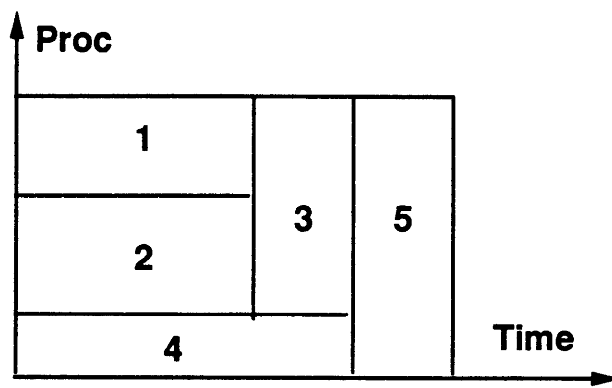




(a) Naive Heuristic



(b) Greedy Heuristic



(c) Tree Heuristic

Figure 4.3: Generalised Scheduling Heuristics

4.6 Compiler Implementation Details

The compiler has a standard structure, comprising of a front end converting the matrix expression dataflow graph into an intermediate language representation, and a back end generating code. The intermediate language representation is used for most of the analysis.

The compiler performs a two level partitioning and scheduling. In the first stage the parallelism (number of processors P) of each particular macro operator is determined by using the generalised scheduling heuristics enumerated above. The workload for each particular operator/sub-tree is estimated as specified in Table 4.1. Then, the blocking parameters for each particular operator are determined as described in Section 4.4. Finally, code consisting of a sequence of calls to the parallel library routines with all the parameters specified is generated.

The Tree heuristic can be applied only to trees. There are several algorithms for determining whether a graph is a tree or not. However, in the current version of our compiler, we explicitly specify whether an expression graph is a tree or not.

4.7 Matrix Expression Examples

The various matrix expressions are tabulated in Table 4.2. The default matrix size is 20×20. The compiler produces correct and relative high speed code automatically. Unless otherwise specified, the runs have been done using the *chained directory* caching scheme.

We present results for the the matrix product library routine first. Other library routines are written in an analogous manner, and are not described. Finally, we present results for complete expressions.

4.7.1 Matrix Product Parallel Library routine

```
                                        ;COMPUTES ONE BLOCK OF MATRIX PRODUCT
(define (fastmul_block_single id arg_list)
  (destructure*
   (((mat_a mat_b mat_c lock_c n1 n2 n3 p1 p3) arg_list)
                                        ;INDEX MANIPULATION
    (nproc (* p1 p3))
    (l_m (quotient n1 nproc))
    (r_m (mod n1 nproc))
    (start_m (+ (* l_m id) (min r_m id)))
    (end_m (+ start_m (cond ((< id r_m) (1+ l_m)) (t l_m))))
    (i (quotient id p3))
    (k (mod id p3))
    (li (quotient n1 p1))
    (ri (mod n1 p1))
    (start1 (+ (* li i) (min ri i)))
    (e1 (+ start1 (cond ((< i ri) (1+ li)) (t li))))
    (start2 0)
    (e2 n2)
    (lk (quotient n3 p3))
    (rk (mod n3 p3))
    (start3 (+ (* lk k) (min rk k)))
    (e3 (+ start3 (cond ((< k rk) (1+ lk)) (t lk)))))

                                        ;MEMORY ALLOCATION
   (do ((i start_m (1+ i)))
       ((= i end_m))
     (set (vref mat_c i) (make-vector n3))
     (semaphore-v (vref lock_c i))
     )
                                        ;INNER LOOP
   (do ((i start1 (1+ i)))
       ((= i e1))
                                        ;SYNCHONIZATION FOR MEMORY ALLOCATION
     (semaphore-p (vref lock_c i))
     (semaphore-v (vref lock_c i))
     (do ((k start3 (1+ k)))
         ((= k e3))
       (set (vref (vref mat_c i) k)
            (do ((j start2 (1+ j))
                 (sum 0 (+ sum
                           (*
                            (vref (vref mat_a i) j)
                            (vref (vref mat_b j) k)))))
                ((= j e2) sum)))))))
```

```
                                        ;TOP LEVEL ROUTINE
(define (fastmul_block
         mat_a mat_b len1 len2 len3 p1 p3 startproc)
  (let* ((ntask (* p1 p3))
         (mat_c (make-vector-on (mod startproc *_NUMBER_OF_PROC*) len1))
         (lock_c (make-vector-on (mod startproc *_NUMBER_OF_PROC*) len1))
         )
    (do ((i 0 (1+ i)))
        ((= i len1))
      (set (vref lock_c i) (make-semaphore))
      (semaphore-v (vref lock_c i))
      (semaphore-p (vref lock_c i))
      )
    (spawn startproc ntask
           fastmul_block_single
           (list mat_a mat_b mat_c lock_c len1 len2 len3 p1 p3))
   mat_c
   ))
```

Figure 4.4: Code for Matrix Product

$g0$	AB
$g11$	$AB + (EF)(GH)$
$g12$	$A_1B_1 + (E_1F_1)(G_1H_1) + A_2B_2 + (E_2F_2)(G_2H_2)$
$g20$	$I + 2A + 3A^2 + \cdots + 9A^8$
$g21$	$I + 2A + 3A^2 + \cdots + 17A^{16}$
$g22$	DFP Update (see later)

Table 4.2: Various Matrix Expressions

The parallel library routine for a matrix product $FASTMUL_BLOCK$, is shown in Figure 4.4. The parameters to the library routine are the input matrices *mat_a* and *mat_b*, the number of rows of *mat_a, len1*, the number of rows of *mat_b, len2*, and the number of columns of *mat_b, len3*. Also specified are the blocking parameters *p1* and *p3* (Section 4.4). The compiler determines *p1* and *p3* by factoring the number of processors P into roughly equal factors, as explained in Section 4.4 (P itself is determined using the generalised scheduling heuristics). *startproc* refers to the lowest numbered processor on which this routine will run. It is necessary to specify this for static scheduling purposes.

$FASTMUL_BLOCK$ works by spawning P threads ($FASTMUL_BLOCK_SINGLE$), each of which handles a different block of the matrix product dataflow graph. Extensive bookkeeping is necessary in $FASTMUL_BLOCK_SINGLE$ to determine the block boundaries, for arbitrary sized matrix products on arbitrary number of processors. Memory to hold the output matrix is allocated by the threads in parallel. A thread cannot write to an output element, before it has been allocated by a possibly different thread. This memory allocation needs synchronization in the form of semaphores in $FASTMUL_BLOCK_SINGLE$. In fact, the matrix product inner loop is a relatively small fraction of the code. A compiler and associated library which automatically manages all this bookkeeping is itself of great value. High speed is an added benefit.

The speedup curve is shown in Figure 4.5. The speedups fall dramatically after about 8 processors. Excessive task startup and communication overhead are the causes of this behaviour. There is an anomalous concave region, with the speedup first decreasing from 4 for 8 processors, to 2.9 for 16 processors, then slightly increasing to 3.2 for 35

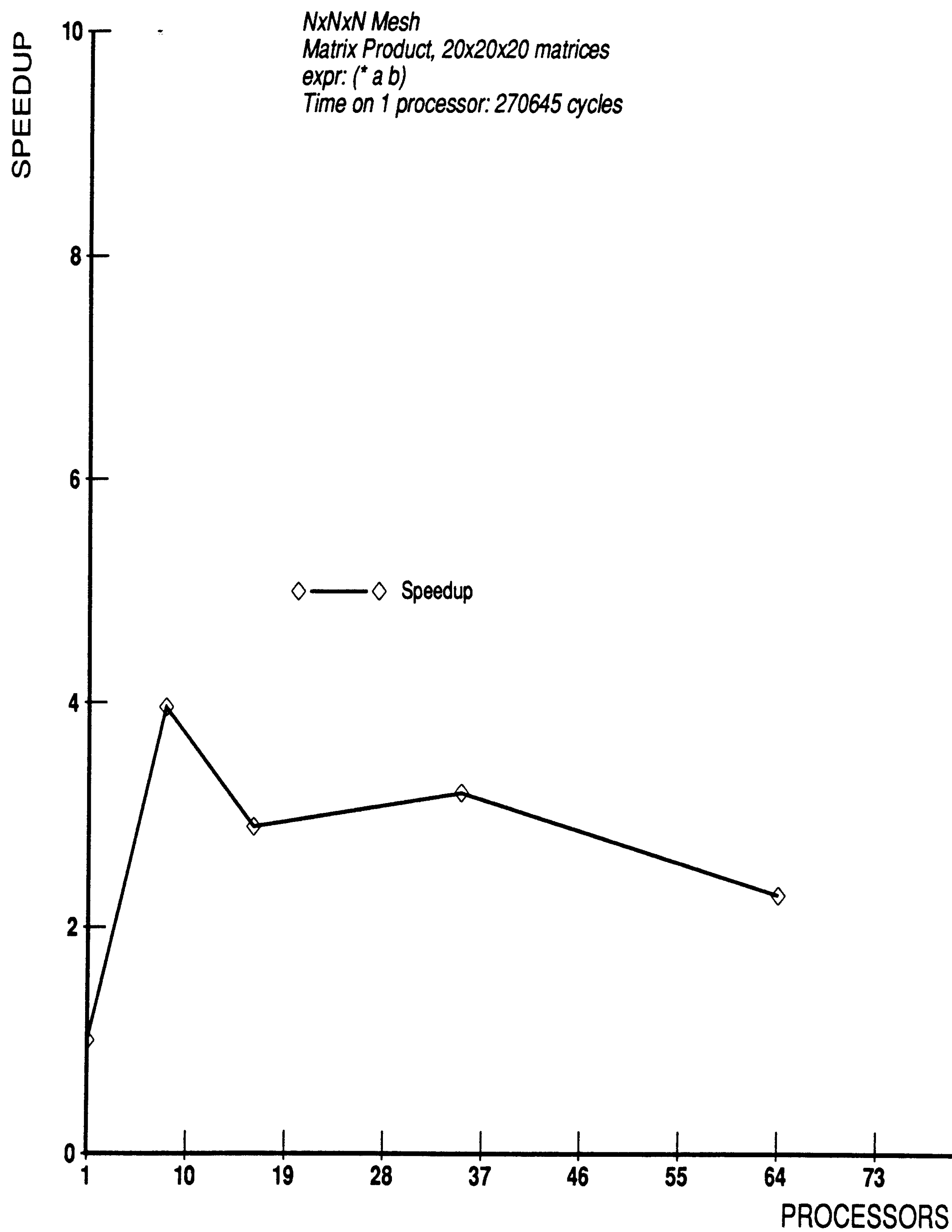


Figure 4.5: Speedup Curve for 20×20 matrix product

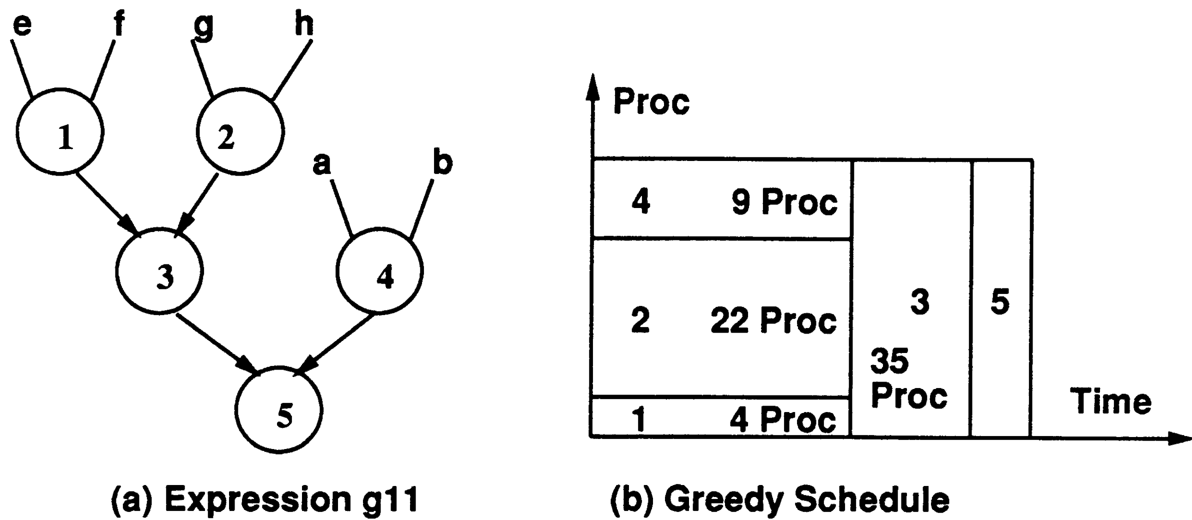


Figure 4.6: Greedy Schedule for $g11$

processors. The speedup falls dramatically to 2.3 for 64 processors.

While the speedup function is clearly not P^α, it can be approximated as a convex function, and much of the generalised scheduling theory of Chapter 3 holds. In particular, the insight that tasks have to be run in parallel if possible is critical. This will be demonstrated in the succeeding examples.

4.7.2 g11 - Filter Bank

Expression $g11$ (Figure 4.6 (a)) is prototypical of systems typically encountered in signal processing and linear algebra. It could represent the iteration of an inner loop, or a filter bank.

A and B are 20×20 matrices, E is a 20×9 matrix, F is a 9×20 matrix, G is a 20×43 matrix, while H is a 43×20 matrix. These strange data sizes have been deliberately chosen to illustrate size-matching (bin-packing) problems encountered in parallelising compilers. Each scheduling heuristic produces different parallelisms for each of the macro-operators. Writing code by hand for each problem size, and each number of processors, is a very tedious job, which should be automated. This is indeed done by

the SDC.

For example, Figure 4.6 (b) shows a Gantt chart for a schedule for 35 processors using the Greedy Heuristic. The matrix products $A \times B$, $E \times F$, and $G \times H$ are computed together. Then, after at least $E \times F$ and $G \times H$ are completed, their outputs are multiplied. Finally, the addition is performed. The matrix multiply $A \times B$ is assigned 9 processors, the matrix multiply $E \times F$ is assigned 4 processors, the matrix multiply $G \times H$ is assigned 22 processors.

The code generated by the compiler is shown in Figure 4.7. Calls to $FASTMUL_BLOCK$ are generated, with the data sizes, parallelism, blocking, and other parameters specified. Parallel tasks are spawned and completed using the future-touch mechanism in Mul-T. Static scheduling is used, since the dataflow graph is completely static. The extensive bookkeeping needed is evident.

The speedup curves show the relative performance of the scheduling heuristics. Two factors contribute to the high speed obtained. One is the speed obtained using the parallel matrix operator library routines. Well designed matrix operator routines can be 1-2 orders of magnitude faster than poorly designed routines, which needlessly spawn tasks for each particular arithmetic operation (add, multiply) in the matrix operator. The second factor accounting for speed is exploitation of parallelism by the Greedy and Tree scheduling heuristics.

The speedup curves (Figure 4.8) show two distinct regimes, one upto 8 processors, and one beyond. All three heuristics perform roughly equally well till about 8 processors. Beyond this, the Greedy and the Tree heuristic outperform the naive heuristic (by a factor of 1.5 for the Tree heuristic on 35 processors). This agrees with the considerable drop of speedup evidenced in the basic matrix product beyond 8 processors. The Tree heuristic is slightly better than the Greedy heuristic, vindicating the theory. The drop off of the curves at 64 processors probably reflects excessive task startup and communication overhead due to improper placement of tasks on processors.

The absolute speedups obtained are relatively low, around 10 for 35 processors, for

```
(herald SELECT-ALL (env tsys const_a int_op_a))
(DEFINE (SELECT_ALL A B C D E F G H)
  (LET ((T.500
         (FUTURE-ON 0
                    (LET ((_TEMP_1 A)
                          (_TEMP_2 B))
                      (FASTMUL_BLOCK _TEMP_1 _TEMP_2 20 20 20 3 3 0))))
        (T.501
         (FUTURE-ON 9
                    (LET ((_TEMP_1 E)
                          (_TEMP_2 F))
                      (FASTMUL_BLOCK _TEMP_1 _TEMP_2 20 9 20 2 2 9))))
        (T.502
         (FUTURE-ON 13
                    (LET ((_TEMP_1 G)
                          (_TEMP_2 H))
                      (FASTMUL_BLOCK _TEMP_1 _TEMP_2 20 43 20 11 2 13)))))
    (TOUCH T.502)
    (TOUCH T.501)
    (LET ((T.503
           (FUTURE-ON 0
                      (LET ((_TEMP_1 T.501)
                            (_TEMP_2 T.502))
                        (FASTMUL_BLOCK _TEMP_1 _TEMP_2 20 20 20 7 5 0)))))
      (TOUCH T.503)
      (TOUCH T.500)
      (LET ((T.504
             (FUTURE-ON
              0
              (LET ((_TEMP_1 T.500)
                    (_TEMP_2 T.503))
                (FASTADD _TEMP_1 _TEMP_2 20 20 7 5 0)))))
        (TOUCH T.504)))))
```

Figure 4.7: Code for $g11$

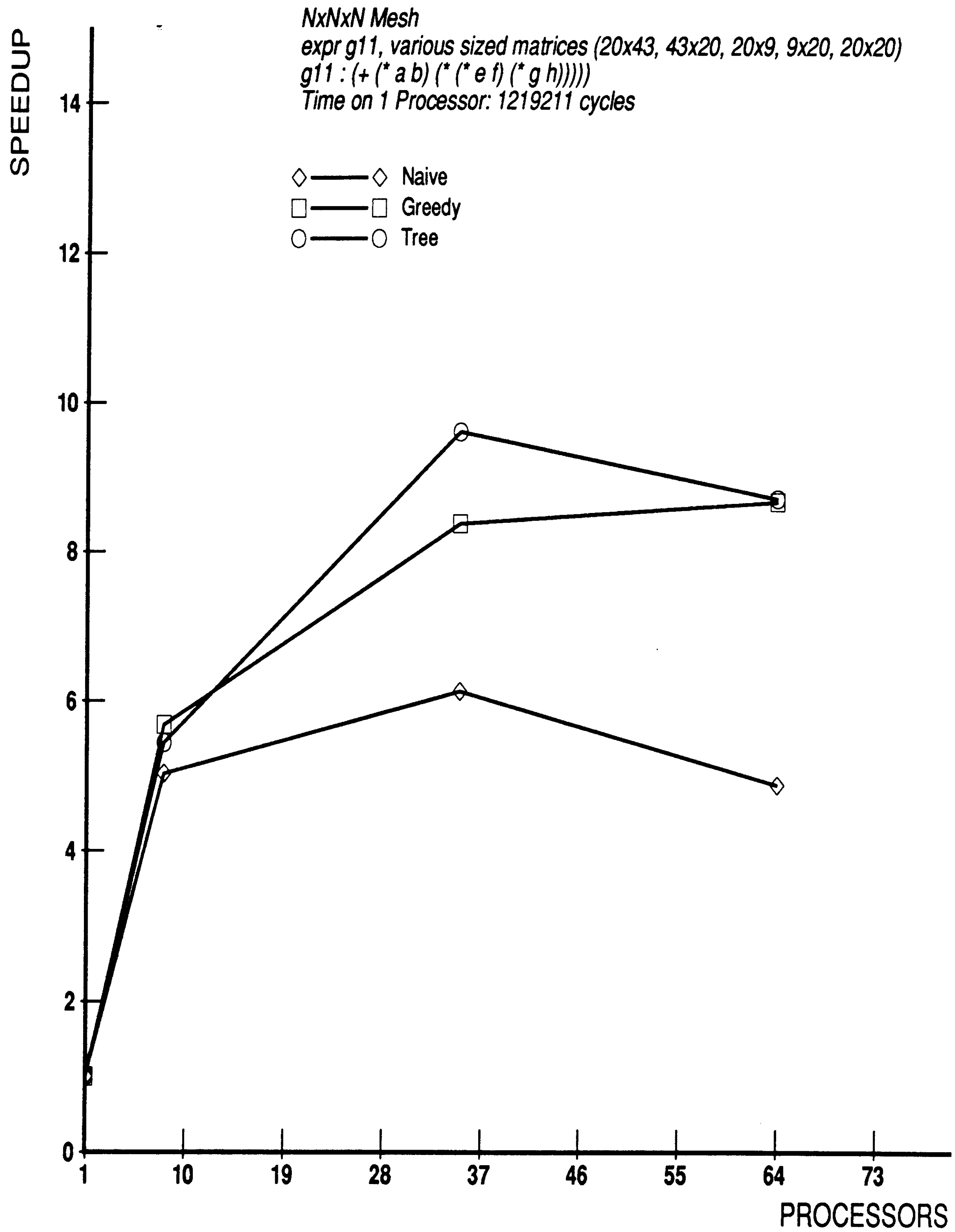


Figure 4.8: Speedup Curves - g11

the Tree heuristic. While various forms of overhead like task startup, communication, and memory allocation undoubtedly play a role, it is important to note that the speedups shown are close to the "true" speedups for the expression. The programs for each number of processors are *different*. The uniprocessor program is optimised for a single processor, that for 8 processors is optimised for 8 processors, and so on. Thus it is expected that the speedups shown will be less than linear. The speedups shown will improve as both the problem size and parallelism increase. This will be illustrated in the next example.

4.7.3 g12 - Larger Filter Bank

$g12$ is essentially two copies of $g11$ performed together in parallel. This could be obtained from unrolling two iterations of an inner loop involving $g11$. Or, if $g11$ represented a filter bank, this could represent a larger filter bank formed by running two banks in parallel.

The increased parallelism in $g12$ enables tasks to be run at coarser granularity. Again two regimes are evident in the speedup curves (Figure 4.9), with a break around 8 processors. The substantial improvement in absolute speedups is evident, with the speedup at 64 processors being close to 15 (25 %) for the Tree heuristic. The Greedy and Tree heuristics outperform the naive heuristic by factors of 2 to 3. The gain is substantially more than in the case of $g11$. The Tree heuristic is also substantially better than the Greedy heuristic in this case. This example well illustrate the gains in performance by exploiting parallelism in the macro dataflow graph.

4.7.4 g20, g21 - Matrix Polynomials

These matrix polynomials are prototypical non-trees, since the powers of the matrices are generated in a recursive doubling fashion. Hence the Tree heuristic is not relevant in this case.

$g20$ is an 8th degree polynomial, while $g21$ is a 16th degree polynomial. The speedup curves (Figure 4.10), again exhibit a break around 8 processors. The Greedy heuristic outperforms the naive heuristic by a factor of 2 for g20, and a factor of 3 to 4 for g21.

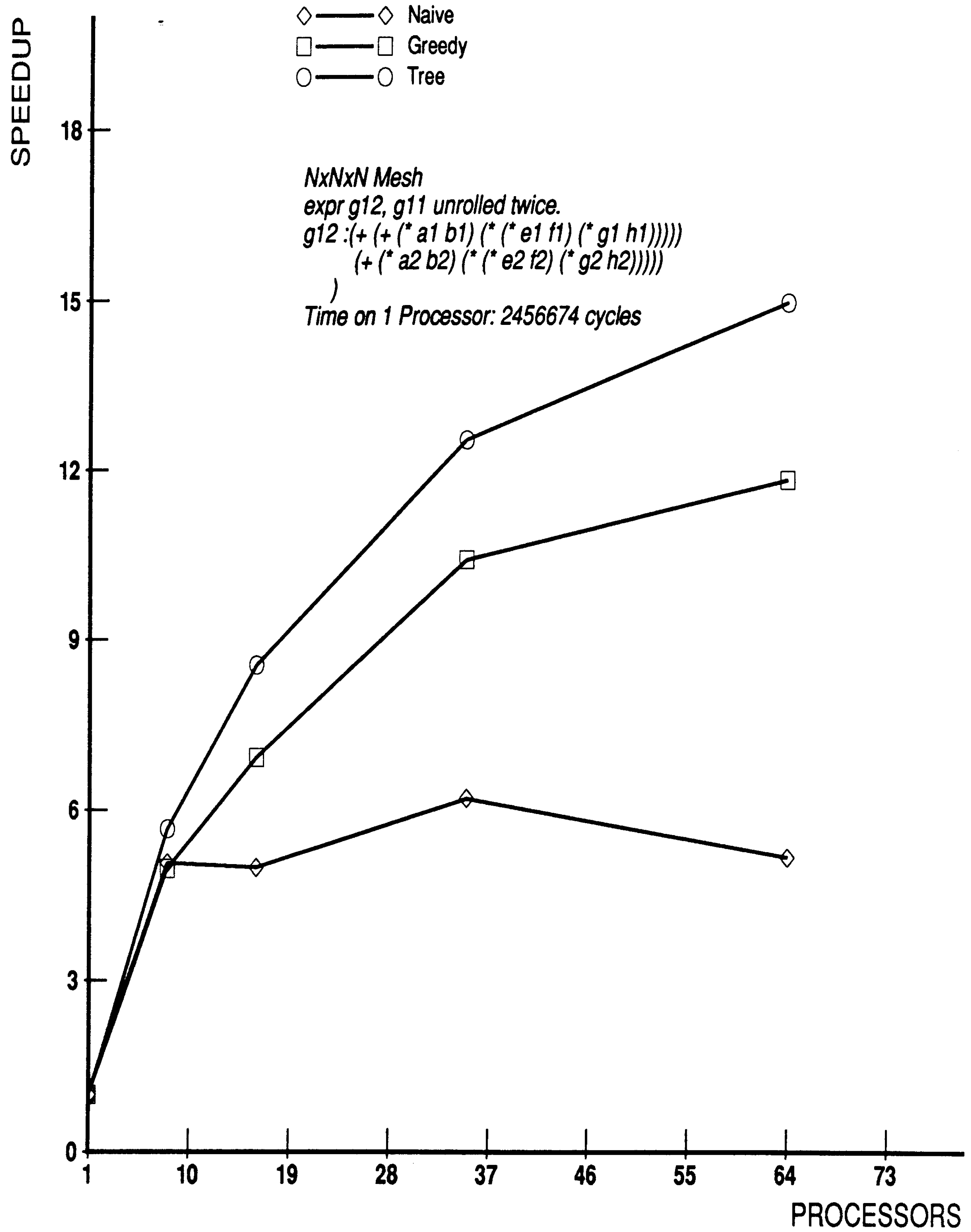


Figure 4.9: Speedup Curves - g12

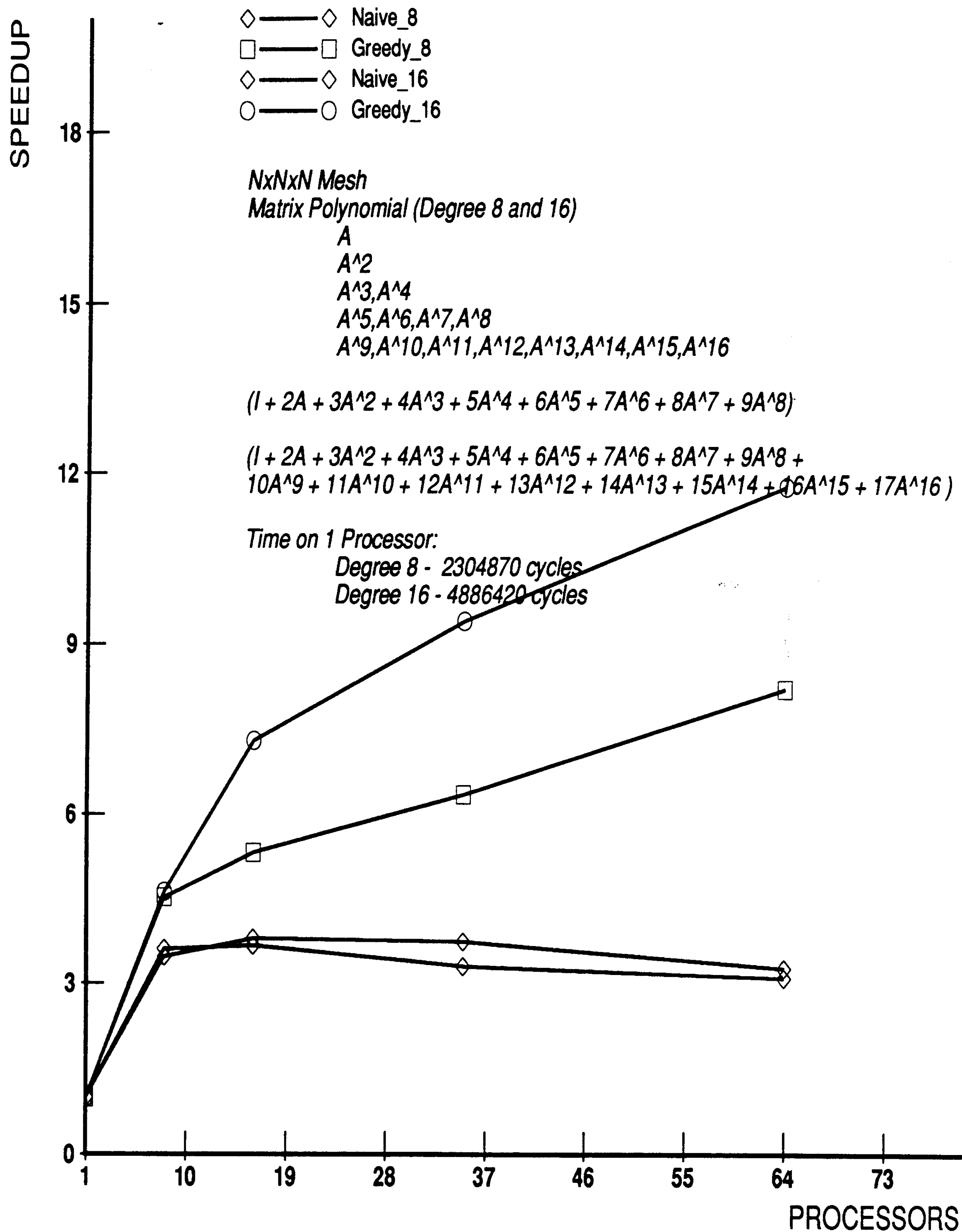


Figure 4.10: Speedup Curves - g20, g21

$$
\begin{aligned}
\Delta X &= x_{i+1} - x_i \\
\Delta G &= \nabla f_{i+1} - \nabla f_i \\
H\Delta G &= H_i \times \Delta G \\
\Delta X\Delta G &= \Delta X.\Delta G \\
\Delta GH\Delta G &= \Delta G.H\Delta G \\
H_{i+1} &= H_i + \frac{\Delta X \otimes \Delta X}{\Delta X\Delta G} - \frac{H\Delta G \otimes H\Delta G}{\Delta GH\Delta G}
\end{aligned}
$$

Figure 4.11: DFP Update

The gains encountered in exploiting increased parallelism (in $g21$) are evident.

Independent of the high speed obtained from the Greedy heuristics, it must be pointed out that the complexity of the expressions is such that writing code by hand, would be an impossibly tedious task. Thus just the fact that the compiler automatically produced correct code, itself makes it useful.

4.7.5 g22 - DFP Update

$g22$ (Figure 4.11) is one iteration of the inner loop of a DFP (Davidon-Fletcher-Powell) update. This example includes many basic matrix operators like inner products, outer products, transposes, matrix sums and products, etc.. 100 element vectors have been used to get the workload in the DFP update comparable to those in the previous examples.

Again the compiler produces correct and relatively fast code. The mere correctness of the code is enough to make the compiler of value. The speedup curves (Figure 4.12) again shows a break beyond 8 processors. The drop-off of the speedup beyond 35 processors is probably due to communication overhead caused by incorrect task placement.

4.8 Discussion and Further Compiler Enhancements

The compiler SDC has provided preliminary experimental validation of the ideas of Chapter 2, and Chapter 3. Developing the parallel library routines has followed the general

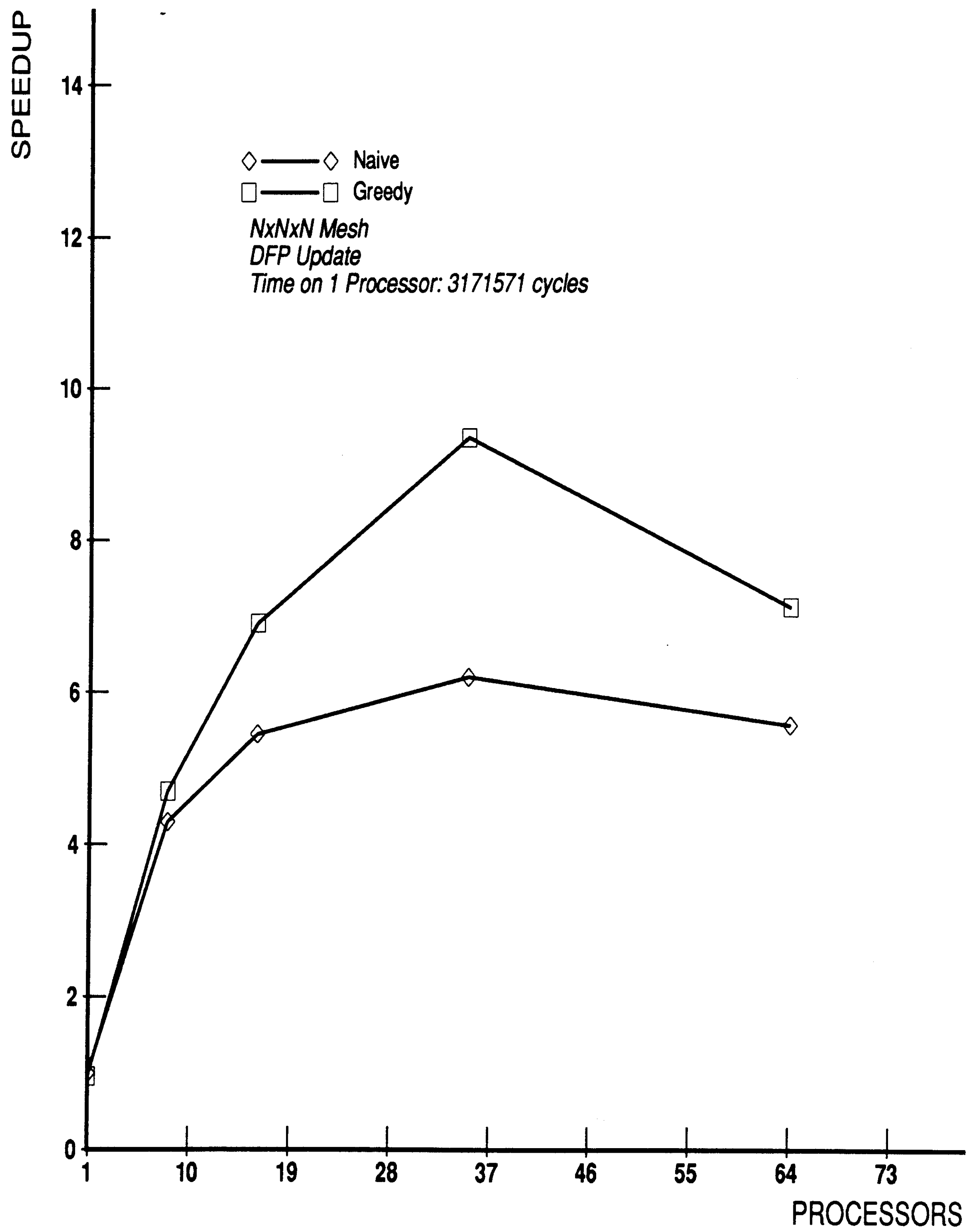


Figure 4.12: Speedup Curves - g22

style of forming compact node clusters, as per Chapter 2. The generalised scheduling heuristics have been implemented in a preliminary form. However, more work has to be done to obtain more thorough experimental results.

Firstly, the multiprocessor model needs to be refined. The processor model needs to be enhanced to reflect the complexities of instruction sets of real processors. This is a relatively simple issue. Much more critical is improving the network model. The fully interconnected, constant access time network assumed is totally unrealistic for modern multiprocessors. The parallel library algorithms should be rederived using the interconnectivity patterns of real networks (eg meshes).

The compiler algorithms need to improved greatly in all respects - determining parallelism, memory allocation, and optimizing communication.

Firstly, the implementation of the generalised scheduling heuristics used by the compiler to determine the parallelism of each particular library routine can be greatly improved. The best strategy would be to use a *measured* speedup function for each particular operator. Processors would be distributed among a set of operator tasks not in proportion to their workloads (Table 4.1), but on the basis of their measured speedup. This enables task finishing times to match more accurately, thus improving both the Greedy and Tree heuristics.

Memory allocation is currently handled dynamically, and introduces some additional overhead, most likely in the form of increased unnecessary network traffic, which leads to hotspotting. Since the matrix expression dataflow graph is static, it is pertinent to allocate memory at compile time itself. Future compiler versions should do this.

The compiler currently ignores task placement issues almost completely. Thus a processor may be asked to compute a block of one matrix product, a totally unrelated block of a successor, and so on. While communication within each operator is minimised by the blocking operator algorithms, communication between two operators is ignored. This communication needs to be minimised in future compiler versions, by proper task placement.

Several other compile time optimizations are also necessary. We have organised the software as a sequence of parallel library calls, with varying parallelism. Each library call spawns, executes, and shuts down a number of threads equal to its parallelism. Further the parameters of the tasks spawned by each library operator (the block boundaries, specifically) are determined at run time (eg. the bookkeeping overhead in $FASTMUL_BLOCK_SINGLE$ in Section 4.7). The repeated thread spawning, parameter computation and thread completion introduces unnecessary overhead. The overhead can be minimised by creating a thread for each processor, and specifying the entire sequence of tasks handled by it over time. Explicit synchronization code would be embedded in the code for each thread. All this can be done at compile time. In essence, we create a process shell for each processor at compile time.

Many of the optimizations alluded to above, especially memory allocation and process shell creation are relatively insignificant for very large size problems. But they are significant for the matrix sizes (20×20) used in the thesis, which are not atypical of signal processing systems, and linear algebra calculations.

The prototype compiler, inspite of all its deficiencies, produces correct and relatively efficient code. Future versions of the compiler, incorporating the changes mentioned above, should yield even better performance.

Chapter 5

Conclusion and Future Work

5.1 Summary

Effective multiprocessor compilation is a critical issue in parallel processing. It is one of the three main issues influencing the performance of a multiprocessor on any algorithm of interest, architecture and operating systems support accounting for the other two. Efficient multiprocessor compilation is a difficult discrete NP-hard optimization problem, as it involves the twin tasks of partitioning and scheduling.

However, for algorithms which have a natural hierarchical structure, the compilation can be greatly simplified, by exploiting the hierarchy. The hierarchical compilation paradigm proceeds by effectively compiling the basic building blocks of the hierarchy first, then composing together routines for the basic building blocks to form a compilation for the complete algorithm.

In this thesis we explored the hierarchical compilation paradigm in the context of matrix expressions. Matrix expressions are an important class of numeric algorithms, as they can represent most signal processing systems, basic linear algebra algorithms, etc. The basic building blocks of matrix expressions are clearly matrix operators. We have developed techniques for compiling operators effectively, thus developing a parallel operator library. Techniques have also been devised to compose together library routines

for compiling the complete matrix expression.

The techniques for compiling parallel library routines for matrix operators (Chapter 2) all rely on using as much apriori knowledge of the structure of the operators, as possible. They exploit the regular geometric lattice structure of the operator dataflow graphs, to greatly simplify the partitioning and scheduling. Continuous polyhedral approximations to the dataflow graph lattice permit continuous polyhedral partitioning techniques to be used in place of expensive discrete graph partitioning techniques.

Optimal compilation needs the dataflow graph to be partitioned into equal sized clusters, and choosing the shape of the clusters to minimise communication. The polyhedral representation shows that communication is proportional to the surface area of the clusters. Hence an optimal partition tries to choose clusters with minimum surface area, for a fixed volume. This is a classic bin packing problem, which can be handled by geometry techniques. The surface area does not increase greatly if clusters are distorted from their optimal shapes. Hence optimal run-times are possible even with large deviations in cluster shapes. This implies that relatively simple, easy to implement heuristics work well in practice. For example, long skinny clusters (assigning each processor to a fixed set of C elements in an $C = A \times B$ matrix multiply) have slightly more communication, but have vastly simpler loop structure, and reduced synchronization overhead.

The parallel library composition techniques (Chapter 3) determine both the parallelism and sequencing of each of the matrix operators (macro nodes) of the matrix expression dataflow graph. A framework using the theory of optimal control has been developed for this purpose.

The fundamental paradigm is to view tasks as dynamic systems, whose state represents the amount of computation completed at any point of time. The matrix expression is then viewed as a composite task system - the operator routine tasks being its subsystems.

At each instant, state changes can be brought about by assigning (possibly varying) amounts of processing power to the tasks. Computing the composite system of tasks

is equivalent to traversing a trajectory of the task system from the initial (all zero) uncomputed state to the final fully computed state, satisfying constraints on precedence, and total processing power available. The processors have to be allocated to the tasks in such a way that the computation is finished in the minimum time.

This is a classical optimal control problem. The task system has to be controlled to traverse the trajectory from start to finish. The resources available to achieve this control are the processors. A valid control strategy never uses more processors than available, and ensures that no task is started before its predecessors are completed. A minimal time schedule is equivalent to a time-optimal control strategy (optimal processor-assignment).

The optimal control theoretic formulation allows some extremely general and powerful theorems to be derived. For example, if all tasks exhibit monotonically increasing speedup as more processors are added, the optimal schedule always uses all processors. If the rate of processing of every task i, on P processors varies as P^{α} (P^{α} dynamics), the theorems can be greatly strengthened. The scheduling problem can be shown to be equivalent to shortest path and network flow problems. Using this, very simple optimal schedules can be derived for tree structured matrix expressions. We split the available processor resources among the subtrees in proportion to the effective workloads. This technique (Tree), together with two other heuristics (Naive and Greedy) has been incorporated in the compiler.

It is important to note that both Chapter 2 and Chapter 3 succeed by approximating the discrete, complicated structure of realistic library routines, with simple speedup functions characterized by a single continuous parameter (processing power applied). The analysis of Chapter 2 yields some insights into the form of the speedup functions, while Chapter 3 makes use of the speedup functions to determine both the parallelism and sequencing of macro tasks. The simple speedup model enables continuous mathematics to be used for making strong general statements about optimality of schedules.

The results are exact only when all the assumptions about the multiprocessor (Chapter 2), and task dynamic behaviour (P^{α} dynamics in Chapter 3) are valid. However, even

if all the assumptions are not exactly satisfied, the results are not too bad. The reasons are enumerated below.

Firstly, in the case of the parallel library operator routines, the basic paradigm is to minimise communication while keeping computation balanced, by choosing compact low surface area clusters for each task. The exact cluster shape depends on the specific architecture. However, since the surface area does not vary very strongly as the cluster shape changes, clusters which are close to optimal do not have much greater communication. This implies that routines which are optimal for one specific architectural model, are close to optimal for many other models. Thus routines based on the simple multiprocessor model are not too bad for realistic machines.

In the case of the generalised scheduling heuristics, the optimal techniques (eg. Tree) need accurate knowledge of the speedup functions. However, as Chapter 4 shows, even if the speedup functions are quite different from the assumptions, the optimal scheduling techniques yield good performance. This is because the optimal scheduling techniques try to maximize the granularity of tasks being computed. The coarse task granularity minimises overhead for a very wide range of multiprocessors. Thus knowledge of the exact speedup function and the exact machine model, is not critical to good performance.

A prototype matrix expression compiler incorporating these ideas has been developed (Chapter 4). A parallel library has been developed, composed of routines for each particular matrix operator. The compiler uses the parallel library routines, and the generalised scheduling heuristics, to generate correct and fast code for the matrix expression. A major gain is the compiler's ability to automatically parallelize code in an environment where writing correct tightly coupled parallel code is quite difficult. The high speed provided is an additional win. A wide variety of examples, have been tested on the MIT Alewife Machine. The results from the prototype provide preliminary verification of the theory.

5.2 Contributions of the Thesis

The major contributions of the thesis have been in the theoretical framework for generalised multiprocessor scheduling. Tasks have been treated as dynamic systems, whose state can be changed by applying processing power. The problem of finding a minimum time schedule can then be equated to the problem of finding a time optimal control strategy. The powerful framework of optimal control can then be applied. The approach adopted is very novel, as it tackles a discrete optimization problem, using approximations based on continuous mathematics.

Previous work in the area [DL89, HL89, BW86, Cof76] generally tackles the discrete problem directly, and confines itself to proving NP-completeness results. Several approximation algorithms, albeit in the discrete domain, have also previously been investigated [HL89].

The continuous approximation has enabled us to derive powerful scheduling theorems, making very few assumptions about task dynamic behaviour. Such theorems cannot be derived in the discrete domain. In the special case of P^α dynamics, the theorems have been greatly strengthened. Connections between scheduling, shortest path and flow problems have been demonstrated.

The continuous approximations used in our work have the flavour of interior point methods, since the constraint that the number of processors is discrete is relaxed. With some more work, it is conceivable that our solution could be used as a first approximation for deriving a schedule wherein all processors are restricted to be integers, thus yielding an interior point scheduling algorithm.

The generalised scheduling theory is of great use in writing good parallelising compilers for macro dataflow graphs. Once the dynamic behaviour of tasks (macro nodes) is characterised by speedup functions, the processor resource can be effectively partitioned to balance the workload. If the tasks have exactly the same P^α dynamics, the optimal partitioning techniques are extremely trivial. Even if the dynamics differ much from P^α, with possible state varying speedups, task interference, etc., the optimal control

problem can be solved numerically to yield an optimal processor assignment. Even in the absence of accurate knowledge of task dynamics, the Greedy and Tree generalised scheduling heuristics do not perform too badly, since they maximise task granularity, thus decreasing various forms of overhead.

A secondary contribution of this thesis is the approach to developing the parallel matrix operator library routines. The regular polyhedral nature of their dataflow graphs has been exploited to derive good partitions and schedules for them, on an arbitrary number of processors. Again, the continuous approximation allows us to bypass the complexities of partitioning the discrete dataflow graph lattice.

Finally, preliminary experimental evidence vindicating the ideas of the thesis has been obtained on the MIT Alewife machine. It indicates that the combination of the parallel operator library routines, and the generalised scheduling heuristics works well in most cases.

5.3 Extension and Future Work

The main contribution of the thesis has been in the theoretical framework for generalised scheduling. There is an enormous amount of work which still remains to be done in this area - the thesis having barely scratched the surface. Some of the more interesting and important areas for future research are mentioned below.

- The generalised scheduling theory (Chapter 3) should be extended with more results for tasks satisfying general realistic speedup functions. A characterisation of the dynamics of real tasks should be carried out, to determine properties of real speedup functions. In general one may have to model interactions between tasks.

- The theorems under the special assumption of tasks satisfying P^α dynamics can be strengthened greatly. In particular, much more work is necessary to elucidate the relations between scheduling, shortest path, and network flow problems. Very deep insights between these problems could emerge from this effort. Good algorithms

for one problem, say shortest path, could be used to yield good algorithms for the other two.

- The results on generalised scheduling ignore the discreteness of the processor variables. It is of great interest to explore what happens in the case where the processors assigned to a task are assumed to be discrete. This could conceivably lead to an interior point scheduling algorithm, as mentioned in Section 5.2 (Contributions). More directly, once the discreteness of the processor variables is incorporated, the results can be directly compared and contrasted with those of classical scheduling theory. This would potentially yield new insights into the classical scheduling problem itself.

- Another very interesting and useful area of research would be to include computation resources other than processors in the task dynamic model. Thus the dynamic behaviour of a task would be a function not only of the processing power applied to it, but also a function of memory and network resources assigned to it. A multidimensional optimization would have to be performed.

The most important piece of empirical work which needs to be done is to develop the next version of the compiler, incorporating better parallelization algorithms including accurate (measured) operation execution times, more compile time optimisations including static memory allocation, good processor placement, etc. This would enable much more complete experimental evidence to be obtained.

The research as it stands enables a parallelising compiler for signal processing to be developed with some more effort. More work is necessary in developing good parallel library routines for the many special operators encountered in signal processing, like convolutions, Fourier transforms and its variants, subsamplers, etc. Once the speedup functions for these operators are determined, the hierarchical compilation technique can be applied.

The hierarchical compilation paradigm assumes that the dataflow graph of the problem is already known. Great gains can be made by applying *algorithm transformations* to generate different variants of the dataflow graph. SPLICE and ESPLICE [Cov89, Mye86] are signal processing algorithm transformation packages developed at the MIT DSPG group. These packages could serve as front ends to our compiler, yielding a very sophisticated multiprocessor compilation system for signal processing.

From a global point of view, the whole hierarchical compilation paradigm explored in the thesis has been in the realm of compiling a very restricted subset of static dataflow graphs. The static hierarchical nature of these dataflow graphs enabled a hierarchical partitioning and scheduling strategy to be applied, at compile time. The static nature of the dataflow graphs also enables compile time optimization of other computation resources, eg. memory, network, etc. It is very interesting to note how far these compile time optimization techniques can be exploited for general dataflow graphs, which have a mixture of static and dynamic elements.

The generalized scheduling theory can certainly be extended to handle dynamic dataflow graphs. Many on-line classical scheduling algorithms can be extended to determine both parallelism and sequencing on-line. Dynamic versions of the Greedy and Tree heuristics can certainly be envisaged. Insights from this theory can be applied to write a compiler handling general dataflow graphs. Of course, the same ideas could also be applied to optimally allocating memory and network resources at run-time.

To sum up, this thesis has shown that hierarchical compilation paradigm is an effective and fast compilation technique for matrix expressions. The technique is capable of being extended for much more general dataflowgraphs, static as well as dynamic. The work has provided very promising insights into generalised scheduling. All in all, the thesis has opened new vistas in the domain of parallel compiling and multiprocessor scheduling.

Appendix A

Optimal Solution for Strictly Increasing Speedups

In this appendix we use the basic formulas for the necessary solution to the minimum-time problem to derive specific formulas for the states, controls, and Lagrange multipliers. First, let t_i^ϵ be the earliest moment that $x_i(t_i^\epsilon) = L_i - \epsilon$, and let t_i^F be the earliest moment that the task fully completes, $x_i(t_i^F) = L_i$. Also, we will say that a task is *enabled to run* at the earliest time t_i that all of its predecessors $j \in \Omega_i$ are within ϵ of completing, $t_i = \max_{j \in \Omega_i} t_j^\epsilon$. We now prove the following lemma:

Lemma A.0.1 ($\lambda_i(t)$, $\mu(t)$ characteristics) *The Lagrange multipliers $\lambda_i(t)$, $\mu(t)$ have the following properties. For $t < t_i^\epsilon$, the multiplier $\lambda_i(t)$ is constant:*

$$\lambda_i(t) = \lambda_i < 0$$

For $t_i^\epsilon \le t \le t_i^F$, $\lambda_i(t)$ is monotonically increasing and strictly negative:

$$\frac{\lambda_i(t)}{dt} \ge 0, \qquad \lambda_i(t) < 0$$

The derivative is non-zero only if a successor task starts during this period. Finally, for $t > t_i^F$, $\lambda_i(t)$ is constant again:

$$\lambda_i(t) = \lambda_i^+ \le 0$$

The Lagrange multiplier $\mu(t) > 0$ is strictly positive.

To prove this, substitute the definition of the Hamiltonian (3.8) into (3.9), giving

$$\begin{aligned} \dot{\lambda}_i(t) &= -\sum_j \lambda_j(t) \frac{\partial f_j(\vec{x}(t), \vec{p}(t), t)}{\partial x_i} \\ &= -\tilde{\delta}\left(x_i(t) - L_i\right) \sum_{j \in \Omega^i} \lambda_j(t) f_j(p_j(t)) \prod_{\substack{k \in \Omega_j \\ k \neq i}} \tilde{U}\left(x_k(t) - L_k\right) \end{aligned} \tag{A.1}$$

Since $\tilde{\delta}(x_i(t) - L_i) = 0$ for all $x_i(t) \le L_i - \epsilon$ and $x_i(t) \ge L_i$, then $\lambda_i(t)$ must have zero derivative at all times except between t_i^ϵ and t_i^F. Furthermore, between t_i^ϵ and t_i^F, the right hand side of (A.1) is positive, so $\lambda_i(t)$ is monotonically increasing. Note that $\lambda_i(t)$ will have zero derivative even during this interval between t_i^ϵ and t_i^F unless there is at least one successor task j of i such that j becomes enabled to run during the last ϵ of task i, $t_i^\epsilon \le t_j < t_i^F$, and non-zero processing power $p_j(t)$ is allocated to task j before t_i^F. Thus if i has no successors, or if no successor of i can begin when i finishes, then $\lambda_i(t)$ is constant for all time, $\lambda_i(t) = \lambda_i$.

Now substituting (3.8) into equation (3.10) gives:

$$\begin{aligned} \mu(t) - \psi_i(t) &= -\lambda_i(t) \frac{\partial f_i(\vec{x}(t), \vec{p}(t), t)}{\partial p_i} \\ &= -\lambda_i(t) \frac{\partial f_i(p_i(t))}{\partial p_i(t)} \prod_{j \in \Omega_i} \tilde{U}\left(x_j(t) - L_j\right) \end{aligned} \tag{A.2}$$

If one or more predecessors of i have not finished so that i can not yet run, $t < t_i$, then the right hand side is zero, and so $\psi_i(t) = \mu(t) \ge 0$. After time t_i, non-zero processing power will have to be applied to task i for some period of time to complete the task. Thus for some t in this interval, $p_i(t) > 0$ and also by (3.12), $\psi_i(t) = 0$. Since $\mu(t) \ge 0$ always by (3.11), (A.2) implies that $\lambda_i(t) \le 0$ during any period during which task i runs. Equation (A.2) thus implies that $\lambda_i \le 0$ and also $\lambda_i(t) \le 0$ even during the time that task i is finishing, $t_i^\epsilon \le t < t_i^F$. Since $\lambda_i(t)$ is differentiable, we also have that $\lambda_i(t) = \lambda_i^+ \le 0$ for $t \ge t_i^F$.

Now we prove that $\mu(t) > 0$. Note that the terminal constraint (3.13) implies that

$$0 = 1 + \sum_i \lambda_i(t^F) f_i(p_i(t^F))$$

This can only be true if there is at least one task, say i_0, with no successors, which is allocated non-zero processing power at the very end of the schedule, and which has $\lambda_{i_0}(t^F) < 0$. But a task with no successors has $\lambda_{i_0}(t) = \lambda_{i_0}$ for all t. Therefore $\lambda_{i_0} < 0$. But during the entire time that task i_0 is runnable, equation (A.2) implies that

$$-\lambda_{i_0} \frac{\partial f_{i_0}(\vec{x}(t), \vec{p}(t), t)}{\partial p_{i_0}} = \mu(t) - \psi_{i_0}(t)$$

For all $t_i < t < t^F$, the left hand side is strictly positive. This can only be true, however, if $\mu(t)$ is strictly positive throughout the interval t_i through t^F.

Now to apply the argument recursively. If i_0 has no predecessors, then $t_i = 0$ and we have shown that $\mu(t) > 0$ for all t. Otherwise, let i_1 be the predecessor of i_0 which is the last to get within ϵ of finishing, $x_{i_1}(t_{i_0}) = L_{i_1} - \epsilon$. Thus there must be some period of time between t_{i_0} and t^F when non-zero processing power must be applied to i_1 in order to finish the last ϵ of work. During this time, $p_{i_1}(t) > 0$ and $\psi_{i_1}(t) = 0$. But (A.2) implies that during this time:

$$-\lambda_{i_1}(t) \frac{\partial f_{i_1}(\vec{x}(t), \vec{p}(t), t)}{\partial p_{i_1}} = \mu(t)$$

Since $\mu(t) > 0$ for all time $t > t_{i_0}$, the left hand side must be strictly positive during the time that task i_1 completes, which is only possible if $\lambda_{i_1}(t) < 0$ throughout this interval. Since $\lambda_{i_1}(t)$ is monotonically increasing, $\lambda_{i_1}(t) < 0$ for all $t < t^F_{i_1}$. But by equation (A.2),

$$-\lambda_{i_1}(t) \frac{\partial f_{i_1}(\vec{x}(t), \vec{p}(t), t)}{\partial p_{i_1}} = \mu(t) - \psi_{i_1}(t)$$

For $t_{i_1} < t < t^F_{i_1}$, the left hand side is strictly positive. This can only be possible, however, if $\mu(t) > 0$ for all the time that i_1 is enabled to run, $t_{i_1} < t < t^F_{i_1}$. Since this time interval overlaps $t_{i_0} < t < t^F$, we conclude $\mu(t) > 0$ for all $t > t_{i_1}$.

If i_1 has any predecessors, then apply this argument recursively to the last predecessor of i_1 to get within ϵ of completion. Ultimately, we reach a task with no predecessors, we prove that its Lagrange multiplier $\lambda_i(t)$ is strictly negative until the task finishes, and that $\mu(t) > 0$ for all time $t > 0$.

Now consider any task i at a moment in time when it has not yet completed, and when non-zero processing power is being applied, $p_i(t) > 0$. During this time $\psi_i(t) = 0$,

and thus by equation (A.2)

$$-\lambda_i(t)\frac{\partial f_i(\vec{x}(t),\vec{p}(t),t)}{\partial p_i} = \mu(t)$$

Since $\mu(t) > 0$, we must have $\lambda_i(t) < 0$ for all time $t < t_i^F$.

This completes the proof of all the points in the lemma. To prove the theorem, note that the strict positivity of $\mu(t)$ implies, by (3.11), that all available processing power must be used at all times, $\sum_i p_i(t) = P(t)$. Also, once all the predecessors of task i have completed, $g_i(\vec{x}(t)) = 1$, and if the task is not within ϵ of finishing, $t < t_i^\epsilon$ then (A.2) implies that:

$$-\lambda_i\frac{df_i(p_i)}{dp_i} = \mu(t) - \psi_i(t)$$

If non-zero processing power is allocated to task i at time t, $p_i(t) > 0$, then $\psi_i(t) = 0$, and

$$-\lambda_i\frac{df_i(p_i)}{dp_i} = \mu(t)$$

This must hold for all tasks which are running at this moment in time, which are not within ϵ of finishing, and whose predecessors are completely finished. Thus the marginal speedups of all such tasks are fixed to a constant ratio, regardless of the amount of processing power available.

Appendix B

Optimal Solution for p^{α} Dynamics

We start by deriving a formula for $\mu(t)$. At the optimal solution, the Hamiltonian H in (3.8) will have the value:

$$\begin{aligned}
H(\vec{x}(t), \vec{p}(t), t) &= 1 + \sum_i \lambda_i(t) f_i(\vec{x}(t), \vec{p}(t), t) \\
&= 1 + \sum_i \lambda_i(t) g_i(\vec{x}(t)) p_i^{\alpha}(t) \\
&= 1 + \sum_i \left[\lambda_i(t) g_i(\vec{x}(t)) \alpha p_i^{(\alpha-1)}(t)\right] p_i(t)/\alpha \\
&= 1 - \sum_i \left[\mu(t) - \psi_i(t)\right] p_i(t)/\alpha \\
&= 1 - \frac{\mu(t)P(t)}{\alpha} + \frac{1}{\alpha}\sum_i \psi_i(t) p_i(t) \\
&= 1 - \frac{\mu(t)P(t)}{\alpha}
\end{aligned}$$

where we used (A.2), the fact that all processors must be used at all times since p_i^{α} is a strictly increasing speedup, and where the last line follows because $\psi_i(t) = 0$ whenever $p_i(t) \neq 0$. From the terminal constraint (3.13), we get:

$$0 = H|_{t^F} = 1 - \frac{\mu(t^F)P(t^F)}{\alpha}$$

so that:

$$\mu(t^F) = \frac{\alpha}{P(t^F)} \tag{B.1}$$

Now take the derivative of H:

$$\frac{dH}{dt} = -\frac{\dot{\mu}(t)P(t) + \mu(t)\dot{P}(t)}{\alpha} \tag{B.2}$$

But from Bryson and Ho we know that:

$$\begin{aligned}\frac{dH}{dt} &= \sum_i \frac{\partial H}{\partial x_i}\frac{\partial x_i}{\partial t} + \sum_i \frac{\partial H}{\partial p_i}\frac{\partial p_i}{\partial t} + \frac{\partial H}{\partial t} \\ &\qquad + \sum_i \frac{\partial H}{\partial \lambda_i}\frac{\partial \lambda_i}{\partial t} + \sum_i \frac{\partial H}{\partial \psi_i}\frac{\partial \psi_i}{\partial t} + \frac{\partial H}{\partial \mu} \\ &= \frac{\partial H}{\partial t} \\ &= -\mu(t)\dot{P}(t)\end{aligned}$$

Combining this with (B.2) gives:

$$\frac{\dot{\mu}(t)}{\mu(t)} = (\alpha - 1)\frac{\dot{P}(t)}{P(t)}$$

or

$$\frac{d \log \mu(t)}{dt} = (\alpha - 1)\frac{d \log P(t)}{dt}$$

Integrating from t to t^F:

$$\log \mu(t^F) - \log \mu(t) = (\alpha - 1)\left[\log P(t^F) - \log P(t)\right]$$

Solving for $\mu(t)$, and combining with (B.1) gives:

$$\mu(t) = \frac{\alpha}{P(t^F)}\left[\frac{P(t^F)}{P(t)}\right]^{1-\alpha} \tag{B.3}$$

Now let us look at equation (A.2) for the $f_i(p_i) = p_i^{\alpha}$ speedup function:

$$-\lambda_i(t)g_i(\vec{x})\alpha p_i^{(\alpha-1)}(t) = \mu(t) - \psi_i(t) \tag{B.4}$$

Consider times $t < t_i$ before task i is enabled to run. $g_i(\vec{x}(t)) = 0$ during this period, so the left hand side is zero. Thus $\psi_i(t) = \mu(t) > 0$, and we must have $p_i(t) = 0$. No processors are allocated to the task until it is enabled to run.

For times $t_i < t < t_i^F$, we have $g_i(\vec{x}(t)) > 0$. Also $\lambda_i(t) < 0$. If $p_i(t)$ were zero at any time during this interval, then the left hand side would be infinite. However, the right hand side cannot be infinite since $\mu(t)$ is clearly finite for finite $P(t)$. Thus $p_i(t) > 0$ during this period. Thus, throughout the time that task i is enabled to run, it must be assigned non-zero processor power. Note that after task i gets to within ϵ of finishing, $t_i^\epsilon < t < t_i^F$, then $\lambda_i(t)$ will start increasing (towards zero) and $p_i(t)$ will decrease as processing resources are withdrawn from the finishing task.

After time t_i^F, either the entire graph is finished executing, $t_i^F = t^F$, or else all processing power is withdrawn from task i, $p_i(t) = 0$, so that this task state does not advance beyond L_i. To prevent the left hand side of (B.4) from going to infinity, we will need $\lambda_i(t) = 0$ for $t > t_i^F$.

Now let's solve (B.4) for $p_i(t)$. During the time that task i runs, $t_i < t < t_i^F$, we have $\psi_i(t) = 0$ and

$$p_i(t) = \left[-\frac{\lambda_i(t) g_i(\vec{x}(t))\alpha}{\mu(t)}\right]^{1/(1-\alpha)}$$

Substituting the formula (B.3) for $\mu(t)$ gives:

$$p_i(t) = \frac{P(t)}{P(t^F)}\left[-P(t^F)\lambda_i(t) g_i(\vec{x}(t))\right]^{1/(1-\alpha)} \tag{B.5}$$

Note that since $\lambda_i(t) = 0$ for $t > t_i^F$ and $g_i(\vec{x}(t)) = 0$ for $t < t_i$, this formula is actually valid for all time t. Also note that $\lambda_i(t) = \lambda_i$ is constant until task i is within ϵ of finishing, and that $g_i(\vec{x}(t)) = 1$ once all the predecessors of i have finished. Thus $p_i(t)$ is a fixed fraction of $P(t)$ throughout the running of task i, except for transients at the beginning and end of the task.

Next to derive the dynamics of the processor assignments. Take the derivative of (B.5):

$$\begin{aligned}
\frac{d}{dt}\left(\frac{p_i(t)}{P(t)}\right) &= \frac{1}{P(t^F)}\left(\frac{1}{1-\alpha}\right)\left[-P(t^F)\lambda_i(t) g_i(\vec{x}(t))\right]^{\frac{1}{(1-\alpha)}-1} \\
&\qquad \cdot \left[-P(t^F)\dot{\lambda}_i(t) g_i(\vec{x}(t)) - P(t^F)\lambda_i(t)\dot{g}_i(\vec{x}(t))\right] \\
&= \left(\frac{1}{1-\alpha}\right)\frac{p_i(t)}{P(t)}\left[\frac{\dot{\lambda}_i(t)}{\lambda_i(t)} + \frac{\dot{g}_i(\vec{x}(t))}{g_i(\vec{x}(t))}\right]
\end{aligned}$$

(Care must be used in interpreting this for $t < t_i$ and $t > t_i^F$.) Substituting the derivative $\dot{\lambda}_i(t)$ from (A.1), and using chain rule on $\dot{g}_i(\vec{x}(t)) = \sum_j \frac{dg_i(\vec{x}(t))}{dx_j}\frac{dx_j}{dt}$, we get:

$$\frac{d}{dt}\left(\frac{p_i(t)}{P(t)}\right) = \left(\frac{1}{1-\alpha}\right)\frac{p_i(t)}{P(t)}\left[\frac{-\sum_{j\in\Omega^i}\lambda_j(t)\frac{\partial g_j(\vec{x}(t))}{\partial x_i}p_j^\alpha(t)}{\lambda_i(t)} + \frac{\sum_{k\in\Omega_i}\frac{\partial g_i(\vec{x}(t))}{\partial x_k}g_k(\vec{x}(t))p_k^\alpha(t)}{g_i(\vec{x}(t))}\right] \quad \text{(B.6)}$$

This can be simplified. Multiply equation (B.4) by $p_i(t)$, and note that $\psi_i(t)p_i(t) = 0$ for all t, since $\psi_i(t) = 0$ whenever $p_i(t) \neq 0$.

$$-\lambda_i(t)g_i(\vec{x}(t))\alpha p_i^\alpha(t) = \mu(t)p_i(t)$$

for all t. Replacing i with k and dividing by $\lambda_k(t)\alpha$ gives:

$$g_k(\vec{x}(t))p_k^\alpha(t) = -\frac{\mu(t)p_k(t)}{\lambda_k(t)\alpha}$$

Similarly, replacing i with j and dividing by $g_j(\vec{x}(t))\alpha$,

$$\lambda_j(t)p_j^\alpha(t) = -\frac{\mu(t)p_j(t)}{g_j(\vec{x}(t))\alpha}$$

Substituting these two equations into (B.6) and simplifying gives:

$$\frac{d}{dt}\left(\frac{p_i(t)}{P(t)}\right) = -\sum_{j\in\Omega^i}\nu_{ij}(t) + \sum_{k\in\Omega_i}\nu_{ki}(t)$$

where

$$\nu_{ij}(t) = \begin{cases} -\left(\frac{1}{1-\alpha}\right)\frac{1}{P(t)}\frac{\frac{\partial g_j(\vec{x}(t))}{\partial x_i}p_i(t)p_j(t)\mu(t)}{\lambda_i(t)g_j(\vec{x}(t))\alpha} & \text{if } \lambda_i(t) \neq 0,\ g_j(\vec{x}(t)) \neq 0 \\ 0 & \text{else} \end{cases}$$

Note that $\nu_{ij}(t)$ is proportional to $\partial g_j(\vec{x})/\partial x_i$. Therefore, $\nu_{ij}(t) = 0$ for $t < t_i^\epsilon$ and $t > t_i^F$. We can interpret $\nu_{ij}(t)$ as a "processor flow" from task i to its successor j. As task i completes, the fraction of processing power $\frac{d}{dt}(p_i(t)/P(t))$ dedicated to task i decreases, with amounts $\nu_{ij}(t)$ pouring out of task i and into each successor j for which $\partial g_j(\vec{x}(t))/\partial x_i > 0$ at this moment. Thus the processing power allocated to i is released to those successors of i which are enabled to run at the moment that i completes.

Appendix C

Proof of Homogeneity Theorem

First compute the dynamics of the warped state $\tilde{x}_i(\tilde{t})$:

$$\begin{aligned}
\frac{d\tilde{x}_i(\tilde{t})}{d\tilde{t}} &= \frac{dx_i(t)}{dt}\frac{dt}{d\tilde{t}} \\
&= (g_i(\vec{x}(t))p_i^\alpha(t))\frac{\tilde{P}^\alpha(\tilde{t})}{P^\alpha(t)} \\
&= g_i(\tilde{\vec{x}}(\tilde{t}))\left(\frac{p_i(t)}{P(t)}\tilde{P}(\tilde{t})\right)^\alpha \\
&= g_i(\tilde{\vec{x}}(\tilde{t}))\tilde{p}_i^\alpha(\tilde{t})
\end{aligned}$$

Thus $\tilde{x}_i(\tilde{t})$ satisfies the correct dynamics as a function of $\tilde{t}$. The only remaining issue is to show that the corresponding schedule finishes all the tasks in the minimum amount of time, $\tilde{t}^F = f(t^F)$. Suppose there were another schedule with $\tilde{P}(t)$ total processing power which could finish the tasks in time less than $\tilde{t}^F$. But then we could apply the inverse time warp, $t = f^{-1}(\tilde{t})$ to produce a processor schedule for the system with power $P(t)$ which would finish earlier than t^F. This, however, contradicts the assumption that our original schedule were optimal. We conclude that an optimal schedule for one processor function $P(t)$ maps directly into an optimal schedule for any other processor function $\tilde{P}(t)$.

Bibliography

[Aga90] Anant Agarwal. Overview of the Alewife Project. July 1990. Alewife Systems Memo #10.

[AK] N. Alon and D.J. Kleitman. Partitioning a Rectangle into Small Perimeter Rectangles.

[AK86] N. Alon and D.J. Kleitman. Covering a Square by Small Perimeter Rectangles. *Discrete and Computational Geometry*, 1:1–7, 1986.

[Alent] Alewife Systems Group. Alewife Systems Memos. 1989-present.

[All88] Allen, R. Unifying Vectorization, Parallelization, and Optimization: The Ardent Compiler. In *Third International Supercomputing Conference, Vol II*, pages 176–185, 1988.

[Ban79] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, 1979.

[BH75] Bryson and Ho. *Applied Optimal Control.* Halstead Press, 1975.

[BS83] T.P. Barnwell and D.A. Schwartz. Optimal Implementation of Flow Graphs on Synchronous Multiprocessors. In *1983 Asilomar Conf. on Circuits and Systems, Pacific Grove, CA*, Nov 1983.

[BW86] M. Błazewicz, J. Drabowski and J. Welgarz. Scheduling multiprocessor tasks to minimise schedule length. *IEEE Transactions on Computers*, C-35(5):389–393, 1986.

[Che85] M.C. Chen. *A Parallel Language and its Compilation to Multiprocessor Machines or VLSI.* Technical Report YALEU/DCS/RR-412, Yale Univ., August 1985.

[Che86] M.C. Chen. Transformations of Parallel Programs in Crystal. In *Proc. of the IFIP's 86*, Sept 1986.

[Cof76] Coffman E.F., Jr., editor. *Computer and Job Shop Scheduling Theory.* John Wiley and Sons, N.Y., 1976.

[Cov89] M.M Covell. *An Algorithm Design Environment for Signal Processing.* PhD thesis, Dept of Electrical Engineering, MIT, Sept 1989.

[DL89] J. Du and J.Y.T Leung. Complexity of Scheduling Parallel Task Systems. *SIAM J. Discrete Math.*, 2(4):473–487, Nov 1989.

[Fis84] Fisher, J.A., et al. Parallel Processing: A Smart Compiler and a Dumb Machine. *SIGPLAN Notices*, 19-6, June 1984.

[HL89] C.C Han and K.J. Lin. Scheduling Parallelizable Jobs on Multiprocessors. In *IEEE Conf. on Real-Time Systems*, pages 59–67, 1989.

[KHM89] D. Kranz, R. Halstead, and E. Mohr. Mul-T: A High-Performance Parallel Lisp. In *Proceedings of SIGPLAN '89, Symposium on Programming Languages Design and Implementation*, June 1989.

[KR89] T.Y. Mount D.M. Kong and A.W. Roscoe. The decomposition of a rectangle into rectangles of minimal perimeter. *SIAM Journal of Computing*, 1215–1231, 1989.

[Kuc78] D.J. Kuck. *The Structure of Computers and Computations.* John Wiley and Sons, Inc, 1978.

[KW87] T.Y. Mount D.M. Kong and M. Werman. The decomposition of a square into rectangles of minimal perimeter. *Discrete Applied Mathematics*, 16:239–243, 1987.

[Lam87] M.S.L. Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Dept. of Computer Science, Carnegie Mellon University, 1987.

[Lei83] C.E. Leiserson. Optimising Sychronous Circuitry by Retiming. In *Third Caltech Conf. on VLSI*, pages 87–116, 1983.

[Mat83] Mathlab Group. *MACSYMA Reference Manual.* The Mathlab Group, LCS, MIT, 1983.

[Mer88] R. Mercer. The CONVEX FORTRAN 5.0 compiler. In *Third International Supercomputing Conference*, pages 164–175, 1988.

[Mye86] C.M. Myers. *Signal Representation for Symbolic and Numerical Processing.* PhD thesis, Dept. of Electrical Engineering, MIT, August 1986.

[PW86] D.A. Padua and M.J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, Dec 1986.

[Sar87] V. Sarkar. *Partitioning and Scheduling Programs for Multiprocessors.* Technical Report, Computer Systems Laboratory, Stanford University, 1987.

[Sch85] D.A. Schwartz. *Synchronous Multiprocessor Realizations of Shift-Invariant Flow Graphs.* PhD thesis, Georgia Institute of Tech., Electrical Engineering, 1985.