MIT/LCS/TR-565

AN EVALUATION OF MULTIPROCESSOR SUPPORT FOR FINE-GRAIN SYNCHRONIZATION IN PRECONDITIONED CONJUGATE GRADIENT

Donald Yeung

February 1993

This blank page was inserted to preserve pagination.

An Evaluation of Multiprocessor Support for Fine-Grain Synchronization in Preconditioned Conjugate Gradient

by

Donald Yeung

B.S., Computer Systems Engineering
Stanford University
(1990)

Submitted to the
DEPARTMENT OF ELECTRICAL ENGINEERING AND
COMPUTER SCIENCE
in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

February 1993

Signature of Author: _____
Department of Electrical Engineering and Computer Science
January 15, 1993

Certified by: _____
A. Agarwal
Associate Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by: _____
C. L. Searle
Chairman, Departmental Graduate Committee

This empty page was substituted for a blank page in the original document.

An Evaluation of Multiprocessor Support for Fine-Grain Synchronization in Preconditioned Conjugate Gradient

by

Donald Yeung

Submitted to the Department of Electrical Engineering and Computer Science
on January 15, 1993 in partial fulfillment of the
requirements for the Degree of

Master of Science

in Electrical Engineering and Computer Science

ABSTRACT

This thesis explores the use of fine-grain synchronization in the preconditioned conjugate gradient (PCG) method using the modified incomplete Cholesky factorization of the coefficient matrix as a preconditioner. The PCG algorithm being studied represents a large class of algorithms that have been widely used but traditionally difficult to implement efficiently on vector and parallel machines. Through a series of experiments conducted using a simulator of a distributed shared-memory multiprocessor, the thesis addresses two major questions related to fine-grain synchronization in the context of this application. First, what is the overall impact of fine-grain synchronization on performance? Second, what are the individual contributions of the following three mechanisms typically provided to support fine-grain synchronization: language-level support, full-empty bits for compact storage and communication of synchronization state, and efficient processor operations on the state bits?

The experiments indicate that fine-grain synchronization improves overall performance by a factor of 3.7 on 16 processors using the largest problem size that was feasible to simulate; the thesis also projects that a significant performance advantage will be sustained for larger problem sizes. Preliminary experience shows that the bulk of the performance advantage for this application can be attributed to exposing increased parallelism through language-level expression of fine-grain synchronization. A smaller fraction relies on a compact implementation of synchronization state, while an even smaller fraction results from efficient full-empty bit operations. The thesis also shows that the last two components are likely to have a greater impact on performance as mechanisms for latency tolerance are employed.

Thesis Advisor: A. Agarwal

Title: Associate Professor of Computer Science and Engineering

This empty page was substituted for a blank page in the original document.

Acknowledgments

The Master's thesis has certainly been an interesting **experience** for me. I emphasize the word "experience" because I feel that even more important than the actual results that I obtained, or the tangible byproducts such as the paper I wrote, is the process that I had to undergo from finding a research topic, to doing the research, to having the argumentative conversations with colleagues, to stressing out amongst family and friends, to finally writing the thesis.

The list of people that made this experience a reality for me is long, so I hope I don't leave anyone out. Foremost, I would like to thank my research advisor Anant for steering me in the direction of my topic in the first place, and for providing the endless guidance to a completely inexperienced researcher throughout the entire research process. Anant is additionally acknowledged for writing significant portions of the abstract and introduction of the paper we coauthored (parts of these sections of the paper appear in this thesis as well). Also, many thanks goes to Norm for introducing me to the preconditioned conjugate gradient algorithm when I was desparately seeking an application for my study.

I would also like to thank the members of the Alewife team who have served as both my colleagues in research and as my friends. Without them, I would never have had ASIM, that awesome simulator of the Alewife machine, nor would I have had valuable sparring mates to tell me how wrong all my conclusions were and to keep me in check. Particularly worthy of mention are the following people. Kirk Johnson, who would usually check up on me past midnight when I was deep into my research, provided many valuable conversations that gave me a better understanding of the architectural tradeoffs which are central to my thesis. Beng-Hong Lim assisted me on all the problems that I had with the Alewife synchronization library. Dave Chaiken had to put up with my numerous requests for yet another simulator statistic which I just HAD to have for my results. And to my endearing officemates Ken and John, I owe many hours of soothing and not so soothing music. I would also like to acknowledge Jonathan for the relevant discussions on application classes and

the not so relevant discussions on wafer-scale integration. And of course, I can't leave Kubi out, who has provided me with many useful comments on my work and who has been the role model to end all role models. And to the whole bunch of Alewives, thanks for listening to my talks over, and over, and over... and for putting up with my simulations that took multiple days, multiple machines, and megabytes and megabytes of memory.

Thanks also goes to Debby, Brad, Ken, Kubi, Beng, Steve, Jonathan, Dave, Dave, John, and Arthur for reviewing my paper which forms the core of this thesis. Plus, I am indebted to all those who participated in my talks, particularly professors Bill Weihl, Bill Dally, and Rishiyur Nikhil for showing up at the Multigroup Parallel Software Meeting and grilling me in tandem.

I would also like to acknowledge my cheerful, pleasant roommates Steve, Mike, and Koleman for providing a happy domestic environment.

And finally, by far my greatest asset through all of this, has been Mom, Dad, my sister Marina, and Melany for the emotional support that kept me sane these past two and a half years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes an in-depth investigation of the impact of fine-grain synchronization in MIMD machines on the performance of the preconditioned conjugate gradient problem using the modified incomplete Cholesky factorization of the coefficient matrix as a preconditioner (henceforth referred to as MICCG3D). An application study of this sort is important because it tells architects not only how programmers will use the mechanisms provided in parallel machines, but also the relative usefulness of various mechanisms provided in the system as evidenced by their impact on end application performance.

One of the challenges in such a design methodology lies in finding appropriate applications which will provide meaningful information concerning a specific set of mechanisms. The problem of finding an application that is both important and suitable for investigating fine-grain synchronization is particularly difficult because benchmarks that pose challenges for synchronization are virtually nonexistent in previous studies of parallel applications. Applications used to study the performance of MIMD multiprocessors have traditionally employed coarse-grain synchronization where synchronization operations are infrequent and are separated by large amounts of useful computation (see [23]). For these problem domains, special mechanisms for synchronization are not necessary. The MICCG3D application meets the criteria for

this thesis because it is an important application with challenging synchronization requirements.

1.1 Contributions of this Thesis

The investigation of fine-grain synchronization in this thesis makes two major contributions. First, the thesis shows how fine-grain synchronization can be employed in MICCG3D to provide significant performance gains over coarse-grain synchronization, and it determines quantitatively this resulting performance gain. The study uses a simulator of Alewife, a distributed memory multiprocessor that provides hardware support for the shared-memory abstraction [1]. The result of this first contribution is that applications for which synchronization is challenging do exist. Furthermore, implementations on MIMD machines can achieve good performance by employing fine-grain synchronization.

Second, through a sequence of experiments, the thesis provides insight into where the "muscle" of fine-grain synchronization lies. A common conception of fine-grain synchronization – one which has contributed to the preference for coarse-grain approaches – has been that its success relies on efficient, but expensive, hardware-supported synchronization primitives. This thesis demonstrates that the most significant contributions of fine-grain synchronization for MICCG3D do not rely on hardware acceleration; rather, they arise from the expressivity and flexibility of language-level support.

1.2 Overview

The rest of this thesis proceeds as follows. Chapter 2 addresses the differences between coarse-grain and fine-grain synchronization and reveals the benefits that fine-grain synchronization affords. In addition, the chapter identifies different levels of support for fine-grain synchronization that machines can provide. Chapter 3 describes

the MICCG3D application, and motivates the need for fine-grain synchronization by discussing why MICCG3D is hard to parallelize. Chapter 4 discusses the details concerning the parallel implementation of MICCG3D using coarse-grain synchronization and fine-grain synchronization. Chapter 5 describes the experimental environment, and in particular, outlines the level of support assumed for fine-grain synchronization. Chapter 6 presents experimental results and discusses their significance. Finally, Chapter 7 includes a summary of the results from this thesis as well as concluding remarks.

Chapter 2

Synchronization

Synchronization in shared-memory MIMD multiprocessors ensures correctness by enforcing two conditions: read-after-write data dependency and mutual exclusion. Read-after-write data dependency is a contract between a producer and a consumer of shared data. It ensures that a thread reading a value produced by another thread performs the read only *after* the write has completed. Mutual exclusion, on the other hand, enforces atomicity. When a data object is accessed by multiple threads, mutual exclusion allows the accesses of a specific thread to proceed without intervening accesses by the other threads. Since MICCG3D only involves synchronization arising from read-after-write data dependency, the rest of this chapter will only address read-after-write data dependency.

2.1 Coarse-Grain Versus Fine-Grain for Producer-Consumer Computations

A coarse-grain solution to enforcing read-after-write data dependency is barrier synchronization. Barriers are typically used in programs involving several phases of computation where the values produced by one phase are required in the computation of the next phase. Parallelism is realized within a single phase of computation,

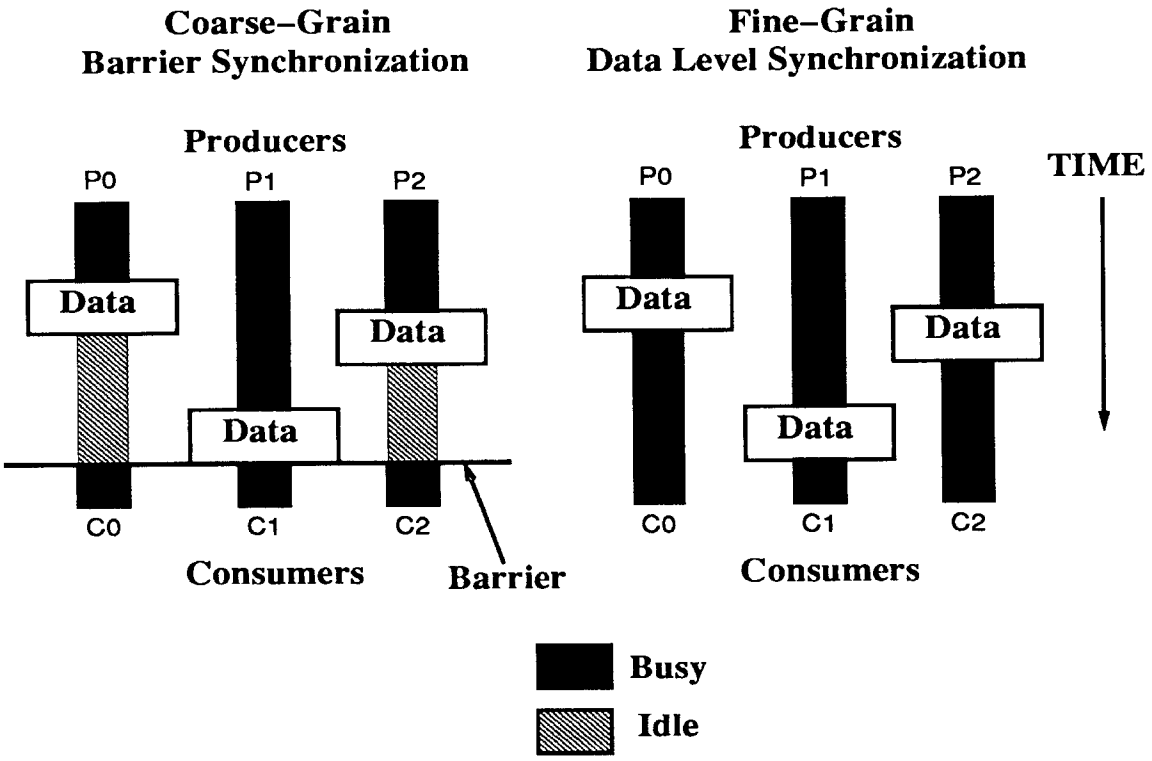

Figure 2.1: Enforcing read-after-write data dependencies in producer-consumer computations using coarse-grain barriers and fine-grain data-level synchronization.

but between phases, a barrier is imposed which requires that all work from one phase be completed before the next phase is begun. Therefore, all the consumers in the system must wait for all the producers. In contrast, a fine-grain solution provides synchronization at the data level. Instead of waiting on processors, fine-grain synchronization allows a consumer to wait on the data that it is trying to consume. When the data has been filled by the producer, the consumer is allowed to continue processing.

Fine-grain synchronization provides two primary benefits over coarse-grain synchronization.

- Unnecessary waiting is avoided because a consumer waits only for the data it needs.

- Global communication is eliminated because consumers communicate only with

those producers upon which they depend.

The significance of the first benefit is that parallelism is not artificially limited. Barriers impose false dependencies and thus lose parallelism to unnecessary waiting. Figure 2.1 illustrates this by showing a trace of three threads of computation in time. In the figure, consumers $C0$ - $C2$ depend on the values computed by producers $P0$ - $P2$. The example shows the data dependencies enforced using both coarse-grain barriers and fine-grain data-level synchronization. Notice in the barrier example that consumers $C0$ and $C2$ are artificially held in time until the last producer, in this case producer $P1$, has arrived at the barrier. This waiting is unnecessary because the data that these two consumers need are actually ready prior to when the last producer arrives at the barrier, but they are nonetheless forced to wait because of skew in the runtimes of the producer threads. Because data-level synchronization does not introduce the false dependency of consumers $C0$ and $C2$ on producer $P1$, such runtime skews do not adversely affect data-level synchronization.

The significance of the second benefit listed above is that each synchronization operation is much more efficient and much less costly than a barrier. This means that synchronization operations can be performed more frequently without incurring significant overhead.

It is important to note that these benefits are a manifestation of the expressivity provided by fine-grain synchronization; they do not depend on assumptions of the underlying hardware implementation. This is an important observation because it underscores the fact that fine-grain expression of synchronization and the implementation of synchronization primitives are orthogonal issues.

2.2 Mechanisms for Fine-Grain Synchronization

In this thesis, three mechanisms to support fine-grain synchronization are identified. They are:

- Language constructs for the expression of fine-grain synchronization.
- Special hardware to compactly store synchronization state.
- Efficient operations on synchronization state.

The first component of support provides the programmer with a means to express synchronization at a fine granularity, resulting in increased parallelism. Another attractive consequence is simpler, more elegant code [14].

The second component of support is based on the intuition that an application using fine-grain synchronization will need a large number of synchronization objects– typically one for every data item. Providing specially allocated state for these objects can lead to an efficient implementation from the standpoint of the memory system. This benefit is referred to as *memory efficiency.* As Chapters 5 and 6 will discuss in detail later in the thesis, memory efficiency has two consequences. The first is that synchronization objects are implemented with lower overhead than if no special state is provided. The second consequence is that memory efficiency results in less communication. With special state, there is a unique association between synchronization objects and data, so whenever a piece of data is fetched, the synchronization object associated with the data will be fetched as well. This gives both the data and the synchronization object in one memory transaction.

Finally, the last component of support is motivated by the expectation that synchronizations will occur frequently. Therefore, support for the manipulation of synchronization objects can reduce the number of processor cycles incurred while accessing synchronization objects. This benefit is referred to as *cycle efficiency.*

In Chapter 6 where the results are reported, each of these components of support for fine-grain synchronization is investigated in isolation. This will expose the impact that each component has on the performance of the MICCG3D application, and thus attribute a notion of relative importance for the three components of support.

Chapter 3

Preconditioned Conjugate Gradient

The Conjugate Gradient (CG) algorithm is a semi-iterative method for solving a system of linear algebraic equations expressed in matrix notation as $Ax = b$. The rate of convergence of the CG method can be improved substantially by preconditioning the system of equations with a matrix K^{-1} and then applying the CG method to the preconditioned system. The idea is to choose a preconditioner such that $K^{-1}A$ is close to the identity matrix I. When this condition is met, the CG algorithm convergences much more rapidly. This is illustrated in Figure 3.1 where the number of iterations to convergence for the CG algorithm with and without preconditioning is plotted for several problem sizes. The numbers were obtained by running a sequential version of both CG algorithms on a SUN workstation.

Since the operations in the basic CG method consist of vector updates, inner products, and sparse matrix-vector multiplies, efficient parallel versions of the algorithm have been demonstrated on many vector machines and MIMD multiprocessors [26] [27]. Preconditioned CG methods, however, have not enjoyed the same success. In many of the most popular preconditioning techniques, the preconditioner steps typically involve recurrence relations which do not vectorize or parallelize easily. Many attempts

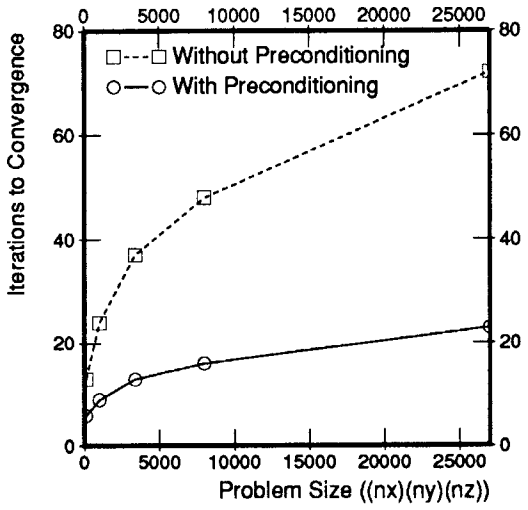

Figure 3.1: Conjugate gradient convergence rates with and without preconditioning. nx, ny, nz denote the discretization degree in the x, y, and z dimensions. The problem size is the product of these three values.

have been made to choose preconditioners or to develop new techniques to perform the preconditioner steps such that the algorithm becomes easier to vectorize or to parallelize [30] [31] [32] [34]. While many of these techniques are successful, they commonly suffer from one of two drawbacks. Some of the approaches do not converge quickly because they employ a suboptimal preconditioner with desirable parallel properties. Other approaches are complex and are tailored to a specific machine configuration. (See Section 3.3).

3.1 MICCG3D

This thesis studies a particular implementation of the preconditioned CG method known as the Modified Incomplete Cholesky Factorization Conjugate Gradient in 3-Dimensions (MICCG3D). MICCG3D is a preconditioned conjugate gradient method that assumes the coefficient matrix A in the system $Ax = b$ is sparse, and symmetric positive definite (SPD). Although this thesis centers around a particular CG

implementation, the general problem being addressed involves increasing performance through the recurrence relations in the preconditioner steps. It is important to emphasize that this problem is common to almost all preconditioned iterative methods, a very large class of applications. Therefore, the solution using fine-grain synchronization proposed by this thesis has general consequences for the large number of algorithms that MICCG3D represents.

3.1.1 Application of MICCG3D

As mentioned above, MICCG3D assumes the coefficient matrix A is SPD. Such a matrix arises from the discretization of many different partial differential equations. Some examples of engineering problems for which this would have application are

- Laplace's Equation: $\nabla^2 u(x,y,z) = 0$.
  Applicable in potential theory for gravitational, electrostatic, and magnetostatic potentials in empty space. Also has application in problems from classical hydrodynamics and heat condution.

- Poisson's Equation: $-\nabla^2 u(x,y,z) = f(x,y,z)$.
  Governs problems of gravitational potentials with continuous distributions of matter density and electrostatic potentials with continuous distribution of charge density.

- Helmoltz's Equation: $-\nabla^2 u(x,y,z) + k^2 u(x,y,z) = 0$.
  Governs problems involving wave propagation such as those arising from Maxwell's equations.

- General 4th-Order PDEs: $\nabla^4 u(x,y,z) = f(x,y,z)$.
  Governs many problems from solid mechanics. An example would be the deflection of a stiff beam by an applied force distribution $f(x,y,z)$.

3.1.2 How the Algorithm Works

This thesis considers the standard 7-point discretization of second-order elliptic partial differential equations in three dimensions. In such a discretization, there are 7 non-zero diagonals in matrix A; all other elements are zero. Given that A is SPD, it is possible to write $A = L + diag(A) + L^T$ where L is a lower triangular matrix. From this expression, the incomplete Cholesky factorization of matrix A, which is denoted as K, can be computed as follows and is given in [30].

$$K = (L + D)D^{-1}(D + L^T) \tag{3.1}$$

In this expression, D is a diagonal matrix. Using the notation where d_i denotes the element in row i, column i of matrix D, and $a_{i,j}$ denotes the ith element in the jth non-zero diagonal away from the center diagonal in matrix A, the elements of D can be expressed as

$$\begin{aligned} d_i \;=\; & a_{i,1} - a_{i-1,2} * (a_{i-1,2} + \alpha a_{i-1,3} + \alpha a_{i-1,4})/d_{i-1} \\ & - a_{i-nx,3} * (\alpha a_{i-nx,2} + a_{i-nx,3} + \alpha a_{i-nx,4})/d_{i-nx} \\ & - a_{i-nxny,4} * (\alpha a_{i-nxny,2} + a_{i-nxny,3} + \alpha a_{i-nxny,4})/d_{i-nxny} \end{aligned} \tag{3.2}$$

α is a parameter ranging from 0.0 to 1.0 (the subscript notation involving nx and ny will be explained shortly). An optimal rate of convergence can be achieved by choosing a value for α which suits the problem domain. It turns out that for most academic (modeling) problems, a value of α closer to 1.0 yields faster convergence rates, while for most industrial (more realistic) problems, a value of α closer to 0.0 is better (see [30]). Throughout this study, for simplicity, it is assumed that $\alpha = 0.0$. Since K is an approximate factorization of A, K^{-1} is used as the preconditioning matrix and the conjugate gradient method is applied to the preconditioned system. The steps for the preconditioned CG method are given below; derivation of the algorithm can be found in [29].

Initialization

$\beta_{-1} = \rho_{-1} = \rho_0 = 0$

$x_0 =$ initial guess

$r_0 = b - Ax_0$

solve $Kw_0 = r_0$

Iterate on i

(1) $p_i = w_i + \beta_{i-1} p_{i-1}$

(2) $q_i = Ap_i$

(3) $\alpha_i = \rho_i / (p_i, q_i)$

(4) $x_{i+1} = x_i + \alpha_i p_i$

(5) $r_{i+1} = r_i - \alpha_i q_i$

(6) if $\rho_i <$ TOLERANCE then quit

(7) solve $Kw_{i+1} = r_{i+1}$

(8) $\rho_{i+1} = (r_{i+1}, w_{i+1})$

(9) $\beta_i = \rho_{i+1} / \rho_i$

3.1.3 Where the Bottleneck Lies

As mentioned earlier, the challenge of MICCG3D lies in parallelizing the vector solution step involving the preconditioner (which will be referred to as the "solver operation"). The solver operation is a significant fraction of the total work in the main iteration loop. Table 3.1 gives a cycle breakdown for one iteration of MICCG3D on a problem size of 8x8x8, where the problem size (nx)x(ny)x(nz) signifies the degree of discretization in the x, y, and z dimensions respectively (the numbers were acquired from a single processor simulation of the Alewife machine which will be described in Chapter 5). The last column shows the relative cost of each vector operation as a percentage of the total number of cycles in the iteration. Notice the solver is the costliest vector operation. If poor parallel performance is suffered in this part of the application, the potential parallel performance of the entire application will be

Vector Operation	Cycles	%
Vector Update	10589	6.17
Sparse Matrix-Vector Multiply	56099	32.7
Inner Product	6212	3.62
Vector Update	9295	5.41
Vector Update	9288	5.41
Solver	74058	43.1
Inner Product	6212	3.62

Table 3.1: Cycle breakdown of an iteration of MICCG3D.

severely limited.

3.2 Why MICCG3D is Difficult to Parallelize

MICCG3D is difficult to parallelize because the recurrence relations in the solver operation impose data dependencies which are numerous and complex. The solver tries to compute w_i, the residual vector in the preconditioned system. Given the solution vector of the current iteration step x_i, w_i can be expressed as

$$\begin{aligned} w_i &= K^{-1}b - K^{-1}Ax_i \\ &= K^{-1}(b - Ax_i) \\ &= K^{-1}(r_i) \end{aligned} \tag{3.3}$$

$r_i = b - Ax_i$ is the residual vector in the original system without preconditioning. Although K^{-1} is the preconditioner, actually calculating it is infeasible. Since A is sparse, it follows that K will be sparse as well because it is an approximate factorization of matrix A. K^{-1}, being the inverse of a sparse matrix, will be dense. Not only will the calculation of K^{-1} be intractable, but the memory space required to store its result will be unmanageable for real problem sizes. Therefore, instead of solving 3.3, it is desirable to solve

$$Kw_i = r_i \tag{3.4}$$

Since the factorization of matrix K is known as the product of a lower triangular matrix and an upper triangular matrix, w_i can be found by first employing back substitution followed by forward substitution. As an example, the backward substitution step can be expressed as follows.

$$w_i = (r_i - l_{i-1,2}w_{i-1} - l_{i-nx,3}w_{i-nx} - l_{i-nxny,4}w_{i-nxny})/l_{i,1} \tag{3.5}$$

where L is the lower triangular factor in K and $l_{i,j}$ is the ith element in the jth non-zero diagonal away from the center diagonal in matrix L. Because of the recurrence in w, it is not possible to perform the entire backward substitution step in parallel. A similar problem exists for the forward substitution step. The dependencies imposed by the recurrence relation in 3.5 result in what is known as "wavefront computation." This form of computation derives its name from the fact that the solutions which can be computed in parallel at any instant in time form a wavefront in the solution space which propagates forward as time elapses. Figure 3.2 shows a snapshot of the wavefront in the three-dimensional solution space of MICCG3D. In MICCG3D, the wavefront forms a plane perpendicular to the diagonal of the three-dimensional solution space and propagates from coordinate (0,0,0) towards coordinate(nx,ny,nz).

3.2.1 The Challenge of Wavefront Computation

There are two reasons why the wavefront computation in MICCG3D is difficult to parallelize. First, the parallelism is not uniform. At the beginning and the end of the computation, there is very little parallelism. In the middle of the computation, parallelism is abundant. During the portions of computation where little parallelism exists, there may not be enough work for all the processors. Second, the dependencies exist across all three spatial dimensions. That is, an element can be computed only if all the elements to the left of it, behind it, and below it in space have been computed. Consequently, it is impossible to choose any cartesian axis in the solution space along which to partition work for the different processors and simultaneously avoid heavy dependencies.

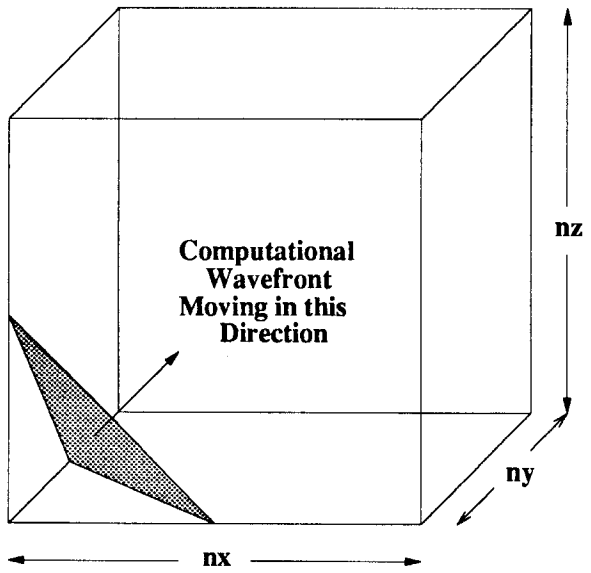

Figure 3.2: Computational wavefront at an instant in time. nx, ny, nz denote the discretization degree in the x, y, and z dimensions.

Machine	Peak MFlops	Vector Update	Inner Product	Sparse MxV	2-Term Recurrence
CRAY-1 (1 Processor)	160	46	75	54	10
CRAY-2 (1 Processor)	1951	71	88	77	8.5
CRAY X-MP (1 Processor)	941	166	166	178	5.7
CYBER 205	400	200	100	100	2.9
ETA-10P	333	167	83.3	83.3	1.6
IBM 3090 (1 Processor)	800	21	29	27	6.5
NEC SX/2	1300	548	905	843	21
Alliant FX/80 (6 Processors)	188	17.1	30.5	16.8	3

Table 3.2: Performance of vector operations in MICCG3D on vector machines.

Machine	PEs	Speedup	MFlops	Peak	%
FX 8	8	5.6	14.5	188	10
Monsoon	8	6.0	2.4	80	3
HP	1	1.0	7	50	14
Cray X-MP	4	2.4	220	940	23

Table 3.3: Performance survey from Rubin's study. The PE column is the number of processors, MFlops is the number of Mega-Flops, the Peak column is the peak Mega-Flops rating of the machine, and the % column is the percentage of peak Mega-Flops that was sustained.

The difficulty in performing computations involving recurrence expressions is well known. Table 3.2 shows performance numbers on some vector computers as presented in [26]. The first column of numbers shows the absolute peak floating point performance of the machine. The remaining columns give the maximum performance in MFlops on each of the four vector operations that appear in MICCG3D. Notice how performance degrades for recurrence expressions as compared to the other vector operations. Although only results for 2-term recurrence expressions are given, the general trends apply to the solver operation in MICCG3D (which involves a 3-term recurrence expression).

3.3 Related Work in Preconditioned Conjugate Gradient

Many studies of the preconditioned conjugate gradient algorithm appear in the literature, which attests to the challenge in making it run efficiently and the importance of the algorithm to the scientific community. This section is only meant to scratch the surface–a comprehensive survey could fill an entire thesis itself.

Since the work in this thesis was inspired by Rubin's study [24], much of the discussion in his study is relevant to the context of this thesis. The greatest similarity is that the exact same algorithm is used. The experimental context for Rubin's study, though, is on Monsoon [10], and the language used to implement the application is

the dataflow language, Id. Rubin's study includes an execution of the application on MINT, a simulator of an ideal dataflow machine. The simulator assumes that an infinite number of instructions can be executed on any given cycle; the only constraint is that the operands for each instruction must be available before it can execute (or "fire"). Also, the simulator assumes that memory access has zero latency. Thus, this simulator yields the maximum parallelism inherent in the application given an ideal machine. For a fairly small problem size ($8x8x8$), Rubin found an average parallelism of about 40. On real Monsoon hardware, Rubin was able to attain a speedup of 6 on 8 processors. Since the Monsoon machine at his disposal was configured with only 8 PEs, he was unable to explore performance on larger machine configurations. Another interesting result provided by Rubin's study is a survey of performance results on various machines for 3-dimensional preconditioned conjugate gradient algorithms where Cholesky factorization is used in the preconditioner. The results of that survey are reprinted in Table 3.3.

Particularly abundant in the literature are studies that try to come up with variants on the basic PCG algorithm which vectorize. Van der Vorst has been particularly prolific in this area. In one of his studies [33], Van der Vorst takes a factorized preconditioner similar to the one used in this study and expresses it as a sum of smaller upper/lower traingular matrices. Each new matrix contains only one diagonal from the original LU factorization. Then, van der Vorst forms the power series expansions of one of these new matrices and truncates the series after m terms. The resulting truncated preconditioner vectorizes, but does not have as good convergence behavior as the orginal preconditioner. A tradeoff in the algorithm is in choosing a value for m. The smaller m is, the less computation is required at each iteration, but the poorer the convergence rate. On the other hand, the larger m is, the more computation at each iteration, but convergence behavior is improved. Van der Vorst demonstrates for two different data sets on the Cray-1 that the truncated preconditioner always performs better than the original preconditioner (he measures about 40% improvement

on both data sets).

Another study which approaches some of the parallel issues for the PCG algorithm is Meurant's study [34]. In this study, Meurant tries to eliminate some of the data dependencies in the back substitution step (referred to as the solver operation in this thesis) so that the preconditioning becomes more parallel. The target for his parallel preconditioner is a 2 and 4 processor CRAY X-MP/48. The approach is to block decompose the preconditioner and use what is known as "twisted factorization" on the blocks. This results in a preconditioner that is only an approximation to the original preconditioner, but some of the recurrence dependencies are eliminated. The experimental results of the study are that very good performance is attained on 2 processors, but diminishing returns are seen on 4 processors due to memory contention and because the parallel preconditioner is not as effective as the original preconditioner in increasing the convergence rate. Furthermore, because his factorization approach does not generalize easily, obtaining parallel preconditioners for machines larger than 4 processors is not an easy task.

Chapter 4

Parallelizing MICCG3D

This chapter discusses two ways of parallelizing MICCG3D. One uses coarse-grain barrier synchronization and the other uses fine-grain data-level synchronization.

4.1 Coarse-Grain MICCG3D

In the coarse-grain approach, the solution space is partitioned along the z dimension and nz/P contiguous planes in the solution space are assigned to each processor, where P is the number of processors. To maximize physical locality of data reference, the partitioning scheme ensures that the nz/P planes assigned to each processor are allocated in that processor's local memory. For this partitioning of the data, communication occurs only in the sparse matrix-vector multiply and solver operations when elements residing on the outermost planes assigned to a processor are being computed. Moreover, this communication is to near-neighbor processors.

4.1.1 Enforcing the Data-Dependencies Using Barriers

Barriers are placed in between vector operations to ensure that results are fully completed before being used in subsequent computation. For all but the solver operation, this is sufficient to guarantee correctness. Dependencies arising from the recurrence

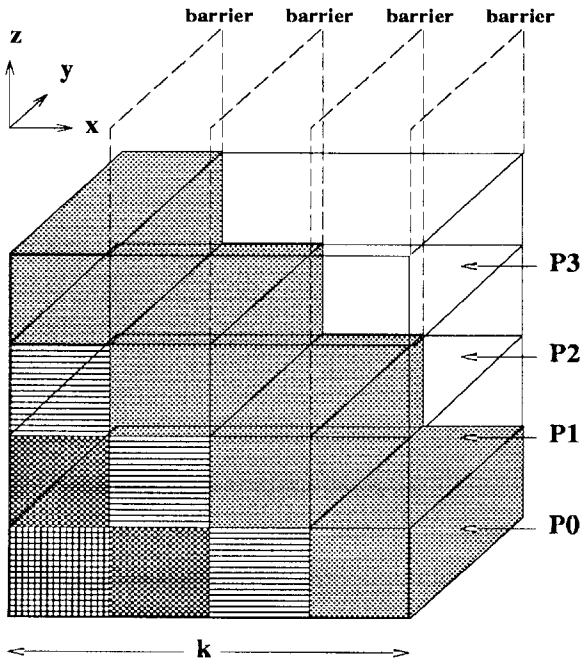

Figure 4.1: Coarse-grain parallel implementation of the solver operation. Similarly shaded blocks are computed in parallel. Pi denotes processor i, and k is the number of computational blocks in the x-dimension.

relation in the solver require further use of barriers. Computation in the solver is further sectioned into k parts along the x dimension. This partitions the nz/P planes assigned to each processor into k rectangular blocks of work each containing $(nz/P)(nx/k)ny$ elements. Each processor computes all the results in one block and enters a barrier before it is allowed to move on to the next block.

Within each block as well as between blocks assigned to the same processor, the dependencies of the recurrence relation are enforced by making sure the order in which the elements are computed follows the dependencies. Enforcing this invariant is possible because computation is sequential on a given processor. Dependencies between blocks across processors must be enforced by staggering the computation. This process is illustrated in Figure 4.1 on an example that has been partitioned for four processors with k equal to four. The blocks that are computed in parallel between barriers are filled with the same hash pattern. Staggering the blocks results in a staircase-like propagation of computation.

4.1.2 Problems with the Coarse-Grain MICCG3D Implementation

The main problem with the coarse-grain MICCG3D implementation is the lack of parallelism in parts of the solver operation. At the beginning of the solver operation, when processor P0 computes its first block, processors P1, P2, and P3 must remain idle. When P0 moves on to its second block after the first barrier, only P1 is allowed to start computing; P2 and P3 are still idle, and so on. Not until P0 is on its 4th block are all processors busy (note the same problem occurs at the end of the solver operation). The degree to which parallelism is limited depends on the value of k. The larger k is, the sooner all processors are computing in parallel.

To find an upper bound on speedup in the solver computation, notice that sequential execution time is proportional to kP, the total number of blocks. Parallel execution time is proportional to the number of block intervals per processor, where

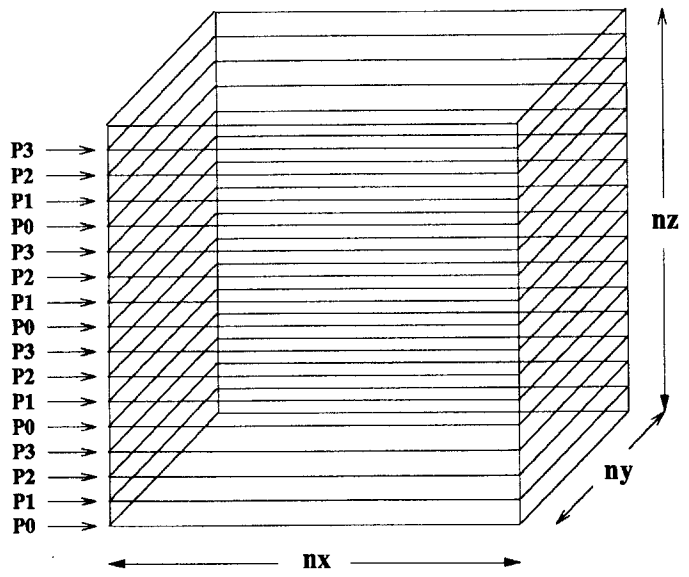

Figure 4.2: Fine-grain parallel implementation of the solver operation. nx, ny, nz denote the discretization degree in the x, y, and z dimensions.

a block interval is the amount of time between barriers. There are k block intervals spent on useful work, and $P-1$ block intervals spent idling. Therefore, the number of block intervals per processor is $k+P-1$. Taking the ratio of sequential to parallel execution time gives the upper bound on speedup.

$$S_{upper} = \frac{kP}{k+P-1} \tag{4.1}$$

Notice this is only an upper bound because it ignores the overhead of barrier operations which becomes more significant as k increases.

4.2 Fine-Grain MICCG3D

Like the coarse-grain implementation, each processor is assigned nz/P planes partitioned along the z dimension; however, in the fine-grain implementation, these planes are not contiguous. Instead, processors are allocated planes modulo P where P is the number of processors. This scheme, illustrated in Figure 4.2, results in substantially more communication during the sparse matrix-vector multiply and solver operations.

Because the data is partitioned into blocks in the coarse-grain implementation, only elements on the outermost planes need to communicate values. In the fine-grain implementation, all elements being computed require communication.

4.2.1 Enforcing the Data-Dependencies Using Data-Level Synchronization

Synchronization is done at the word-level using fine-grain synchronization. This programming model completely eliminates the need for barriers except in the inner product where an implicit barrier occurs in the accumulate of all the individual scalar multiplies. Not only does word-level synchronization enforce the dependencies between vector operations, it also automatically enforces the recurrence dependencies in the solver operation. In the fine-grain version of the solver, each processor can compute results for its elements as fast as possible. If a thread tries to read a value that has not yet been computed, the semantics of the data-level synchronization force that thread of execution to stop and wait until that location has been filled. This provides several benefits. First, processors never wait unnecessarily. Computation proceeds as long as the values that are needed are available. Second, the details of where synchronizations occur is abstracted from the programmer. All the programmer needs to do is specify the algorithm. The dependencies are handled by the system at runtime, making the code less complex. In addition, the code is more efficient because the order in which the elements are computed is more natural compared to the awkward staggering of blocks in the coarse-grain implementation, thus simplifying the computation of array indices.

4.2.2 Theoretical Fine-Grain Performance

The theoretical performance of the fine-grain implementation is similar to the theoretical coarse-grain performance in that there is a startup time during which processors

are idle waiting for the first set of values they depend upon to be produced. However, in the fine-grain implementation, this startup time is significantly smaller because of the grain size of the data protected by each synchronization operation. In the coarse-grain implementation, that grain size is a whole chunk of work containing $(nz/P)(nx/k)ny$ elements. In the fine-grain implementation, this grain size is only one element. So for a reasonable problem size (reasonable relative to the machine size), the startup effects (and the similar effects at the end of the computation) are miniscule. In this limit, the theoretical speedup is linear (*i.e.*, $S_{upper} = P$).

It is possible that processors may have to idle in the middle of the computation in the fine-grain implementation because data that they require has not yet been produced. These effects can reduce the theoretical linear speedup predicted above. However, as it will be shown in Chapter 6, although such idling does occur, it is rare and certainly not the common case.

Chapter 5

Implementation Environment

The results reported in Chapter 6 are in the context of the Alewife Machine. This chapter describes the general architecture of Alewife, how synchronization is done in Alewife, and the special support which exists for fine-grain synchronization.

5.1 The Alewife Hardware

Alewife consists of a scalable number of homogeneous processing nodes connected in a 2-dimensional mesh network. Each Alewife node consists of a 32-bit RISC processor, a floating point unit, 64 KBytes of cache memory, 8 MBytes of dynamic RAM, a controller memory management unit (CMMU), and a network routing chip. The channels in the 2-D mesh network are bidirectional; end-around connections do not exist at the edges of the network. Figure 5.1 shows the topology of the machine and a detail of one of the nodes.

The RISC processing element, named Sparcle, is the SPARC processor with some modifications both to the basic hardware and to the way existing hardware is used. These modifications provide functionality that is useful in a multiprocessor environment. Some of the more important modifications are listed below.

- The register windows in the SPARC architecture are used for block multithread-

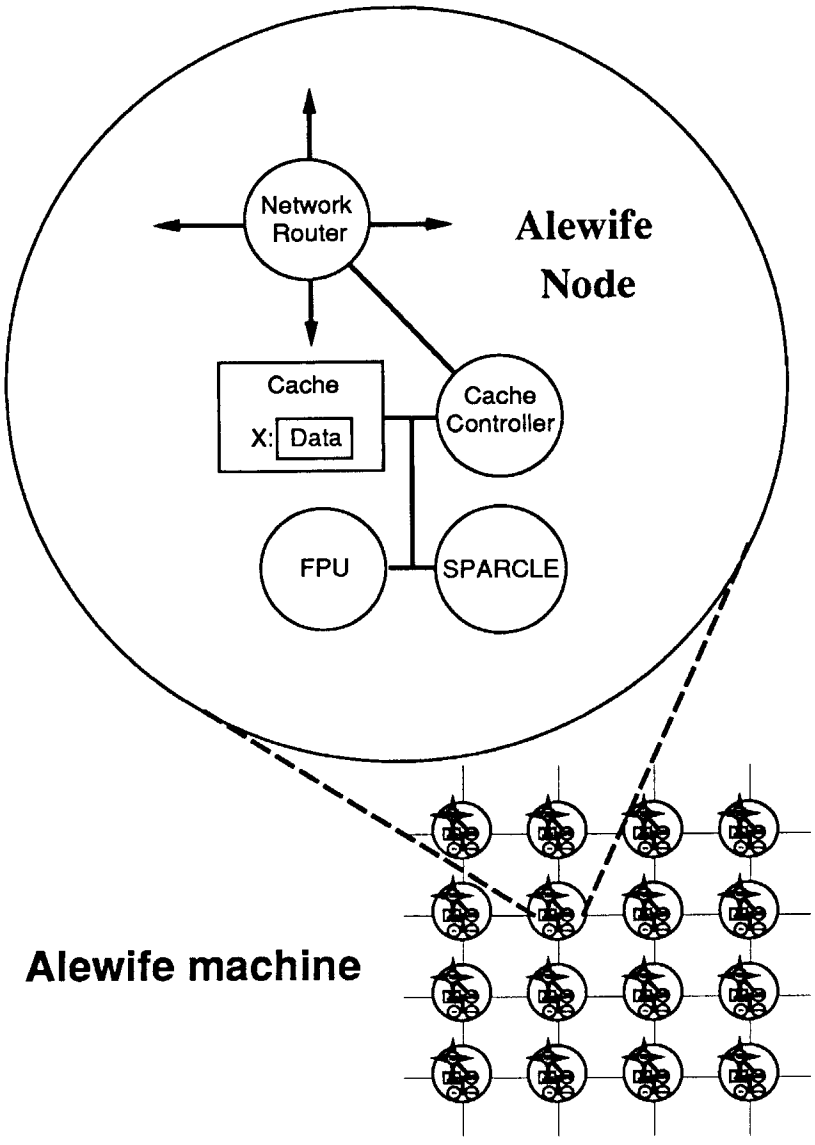

Figure 5.1: Topology of the Alewife Machine and detail of one node.

ing [1]. The original eight windows are organized as four register sets. Each register set, known as a context, caches the state associated with one thread of computation. The registers of each context are partitioned into 2 frames: a user frame of 32 registers (8 of which are global to all the contexts), and a trap frame of 16 registers. Eight of the registers from the user and trap frames belonging to the same context overlap. A context switch on Sparcle simply amounts to changing the current window pointer to point to a new context, saving the Program Counter (PC) and Processor Status Register (PSR), and flushing the processor pipeline. In the current implementation of Sparcle, this can be done in 12 cycles. The ability to context switch rapidly between multiple contexts facilitates block multithreading, which is a useful technique for hiding the latency of remote memory requests and failed synchronization operations.

- The Alternate Space Indicator (ASI) field in the SPARC architecture has been used to tag certain load/store instructions. The tags identify these load/store instructions as special synchronized accesses that are recognized by the CMMU. The CMMU is responsible for invoking the appropriate synchronization action for these instructions. Through this mechanism, synchronization at an instruction-by-instruction granularity is feasible. Also, the SPARC instruction set has been augmented to provide instructions which compile directly down to these special ASI values.

- Alewife relies heavily on traps to the processor to handle many frequently occurring events such as context switching, synchronization failures, and interprocessor interrupts. Thus, it is important to detect and dispatch traps efficiently. Sparcle supports this by expanding the number of pins upon which synchronous traps can be detected, and doing more decoding of trap conditions at the time of the trap. Both of these features reduce the amount of computation necessary to figure out the exact trapping condition once the processor enters the trap handler so that traps can be processed more rapidly. Also, the dedicated trap

frames in each context remove the need to save and restore user registers.

The floating point unit and the cache memory chips are standard off-the-shelf parts from the SPARC chip set, and the network router is the CalTech Mesh Routing Chip (MRC). The MRC provides two dimensions of routing with two 8-bit channels in each dimension. Each channel is capable of a peak bandwidth of 100 Mbytes/sec thus giving 200 Mbytes/sec along each dimension.

The CMMU part is a VLSI gate array being designed and implemented by the Alewife group and fabricated by LSI logic. It is the heart of each Alewife node. Among its many functions, it provides to the user at the programming level a sequentially consistent view of a monolithic shared memory address space. It also maintains coherence between all the caches in the system via a directory scheme known as LimitLESS [2]. In addition, it handles all the forming, dispatching, and processing of messages to and from the network. (For more information about the CMMU, see [3]).

5.2 Synchronization on Alewife

Synchronization can be expressed in the Alewife system in many different styles. Not only does a substantial synchronization library exist [4] providing the programmer easy access to many of the more commonly used synchronization primitives, but mechanisms at a very low level are also made available to the user so that he or she can synthesize new primitives to meet his or her needs. Here, we present a brief discussion on a subset of existing library primitives that are relevant to the coarse-grain and fine-grain implementations of MICCG3D as described in Chapter 4. In particular, the discussion will be limited to ways of doing synchronization for producer-consumer computations using coarse-grain barrier synchronization and fine-grain data-level synchronization.

5.2.1 Coarse-Grain Barrier Synchronization

The barrier provided by the Alewife synchronization library is a software combining tree barrier. Processors arriving at the barrier increment a barrier count in shared-memory. The last processor to arrive also sets a release flag which signals to all the waiting processors that they can leave the barrier. In this barrier, the counter and release flag are distributed across all the processors over a combining tree structure so that contention for these resources is minimized. The software combining tree barrier in the Alewife synchronization library is able to perform a barrier operation on 16 processors in approximately 2500 cycles.

Although it was not used for the implementation in this thesis, Alewife also provides a tree barrier which uses direct message sends via the interprocessor interrupt mechanism (see [3]). This barrier implementation bypasses the overhead of the cache coherence protocol associated with communicating via shared-memory. Because the barrier counters and release flags in the software combining tree barrier are read and written by multiple processors in the system during the barrier operation, lots of communication between processors occurs solely for the purpose of maintaining consistency on those values. This extra communication can be avoided by directly sending messages through the tree structure everytime a processor arrives at the barrier. This results in a much more efficient barrier operation.

While the message-passing barrier is quite a bit more efficient than the shared-memory barrier, the thesis shows in Chapter 6 that skew in thread runlengths and not barrier operation latency is the main reason for performance degradation in the coarse-grain barrier implementation of MICCG3D. Using a more efficient barrier reduces the latency of each barrier operation but does not address skew. Therefore, it is not expected that the results of this thesis will change noticeably if the message-passing barrier is used in place of the software combining tree barrier.

5.2.2 Fine-Grain Data-Level Synchronization

Alewife supports the fine-grain synchronization capabilities described in Chapter 2. Data-level synchronization for producer-consumer types of computations is supported by the J-structure language construct. A J-structure is essentially an array of data. Each element, in addition to storing the data value, can be in one of two states: full or empty. Initially, all the elements of the J-structure start in the empty state. As producers and consumers manipulate the elements in the J-structure, this state changes. The semantics for producers and consumers are as follows.

- **Consumer Semantics:** A read to a location that is in the full state proceeds as a normal read. A read to a location in the empty state causes the thread of computation issuing the read to block, and prior to blocking, the thread enqueues a continuation at the location it needs.

- **Producer Semantics:** A write to a location in the empty state proceeds as a normal write, except for when there are queued continuations at the location. In this case, the data is forwarded to all the continuations and those threads are allowed to proceed. A write to a location in the full state generates an error requiring the program to abort.

The reader might notice that J-structures are very similar to I-structures [14]. Indeed they are with the exception that J-structures can be reset back to the empty state once in the full state and thus can be reused. I-structures do not provide this resetting capability and thus are never reused.

Alewife also supports data-level synchronization primitives for enforcing mutual exclusion on data. The primitive for this kind of synchronization is known as L-structures. Since L-structures are not used in the implementation in this thesis, a discussion of them is omitted here. For a discussion on L-structures and a more detailed discussion on J-structures, see [25].

5.3 Support for Fine-Grain Synchronization

The implementation of J-structures (and L-structures) in Alewife is facilitated by full-empty bits in the memory hardware and fast operations on full-empty bits in the processor hardware.

A full-empty bit is associated with every word in shared-memory. Each full-empty bit acts as a dedicated hardware lock for that memory location. During special load-store instructions supported by the processor, the state of this bit can be altered or tested for some condition. The full-empty state associated with J-structure elements can be implemented directly via these full-empty bits.

5.3.1 Memory Efficiency

Full-empty bits provide the memory efficiency benefit discussed earlier in Chapter 2. As was indicated in that chapter, this benefit has two consequences. First, the memory overhead for synchronization objects is low. Without full-empty bits, a programmer would have to explicitly allocate extra memory for every synchronization variable. For data-level synchronization and word long synchronization variables, this can potentially double the memory requirements of an application. It is possible to allocate multiple synchronization variables in a single word of memory, but this would introduce the problem of false sharing. In a system without full-empty bits, it would probably be best to allocate some small number of synchronization variables per memory word to control the amount of overhead and to minimize the severity of false sharing effects.

The second consequence of memory efficiency is that there is a reduction in communication. A synchronized data-level access on Alewife brings both the datum and the synchronization variable to the processor in one memory system transaction. This is possible because full-empty bits are closely associated with the data to which they belong. If the datum and synchronization variable are stored in separate memory

locations, two memory transactions would be needed.

5.3.2 Cycle Efficiency

All operations on the full-empty bits occur simultaneously with the load or store of the associated data facilitated by special support in the processor hardware. These operations are thus atomic and do not cost any additional cycles. This constitutes the cycle efficiency benefit described in Chapter 2. If hardware support did not exist in the processor to test the full-empty bit simultaneously with loading the datum, a synchronized access would take at least three instructions: one to access the full-empty bit, another to test the bit, and a third to access the datum.

The result of conditional tests on the bit can be used to affect processor activity. In particular, the processor can be trapped depending on the type of load-store being executed and the state of the full-empty bit. Exceptional cases such as failed synchronizations can be identified using this trapping mechanism and then processed in a software trap handler. This approach provides extremely efficient support for successful synchronization operations since successful synchronizations proceed as normal load/store instructions. Failed synchronizations, however, are relegated to slower software trap handlers and thus incur a high latency. The philosophy driving this approach is that successful synchronization operations will be the common case when using fine-grain synchronization.

Towards the end of the next chapter, this thesis will compare the importance of cycle efficiency and memory efficiency against the benefit of increased parallelism offered by language-level support.

Chapter 6

Results

This chapter presents the experimental results obtained on the MICCG3D application. The results fall under two categories. The first set of results compares the performance between the coarse-grain and fine-grain MICCG3D implementations. Not only will the simulation data quantifying the performance difference be presented, but there will also be discussion on how these results will change as problem size and machine size are scaled beyond what can be simulated. The second set of results investigates the performance gains observed for the fine-grain implementation and tries to explain what is responsible for these performance gains. In particular, what is the impact of memory and cycle efficiency on the performance of the application?

6.1 Comparing the Coarse-Grain and Fine-Grain Implementations

Simulation results were obtained for both the coarse-grain and fine-grain implementations on 1, 4, and 16 processor Alewife configurations. A problem size of 16x16x16 was used.

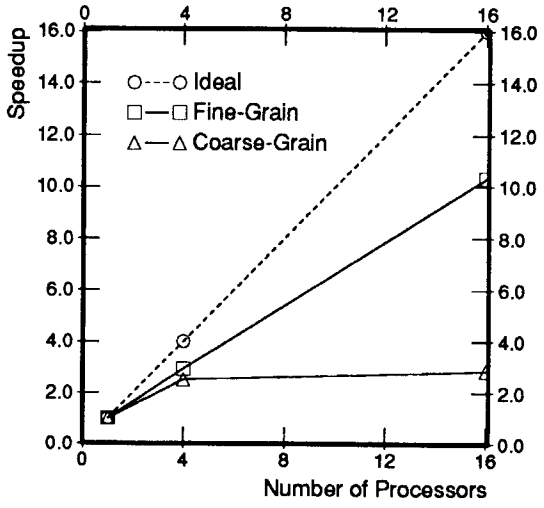

Figure 6.1: Coarse-grain and fine-grain speedups on MICCG3D. Problem Size = 16x16x16.

6.1.1 Speedup Results

Figure 6.1 shows the speedups observed on the Alewife simulator for the coarse-grain and fine-grain MICCG3D implementations. Notice that the fine-grain implementation does consistently better than the coarse-grain implementation. The difference in performance can be predominantly attributed to the solver operation.

To graphically show the performance difference between the two implementations in the solver operation, a synchronization trace of the solver operation in both the coarse-grain and fine-grain implementations was recorded. These traces appear in Figures 6.2 and 6.3 respectively. 16 processors are shown executing the solver on a 16x16x16 problem size.

In both traces, the signature of the wavefront computation in first the backward substitution step and then the forward substitution step is visible. Black bars represent useful work and the interspersed white space signifies waiting for synchronization. In the fine-grain trace, failed J-structure references (which are referred to as "JREF misses") are traced by a cross appearing above the barline of the processor that ex-

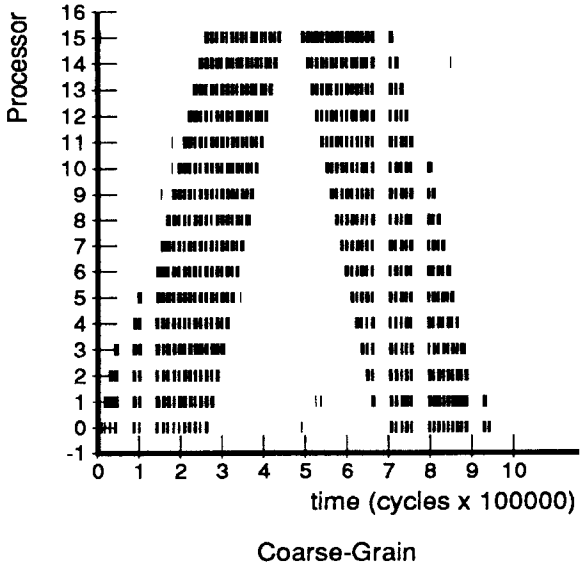

Figure 6.2: Coarse-grain barrier trace in the solver operation of MICCG3D. Problem size = 16x16x16. Barlines show useful work. White space shows waiting for barrier synchronization.

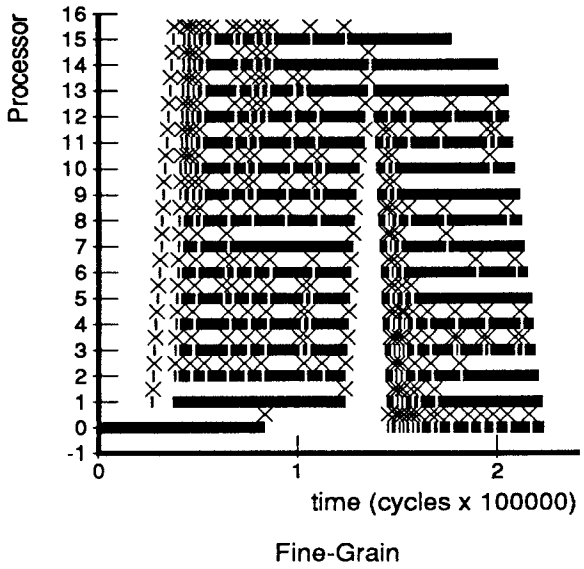

Figure 6.3: Fine-grain JREF trace in the solver operation of MICCG3D. Problem size = 16x16x16. Barlines show useful work. White space shows waiting for synchronization. Crosses show JREF misses.

perienced the event. After every JREF miss, processors idle until the desired value is filled by its producer. Comparing the amount of idle time (white space) in each trace, one can see that the coarse-grain implementation is waiting much more than the fine-grain implementation. Although both traces have been sized for clarity, notice that the coarse-grain trace is approximately four times longer than the fine-grain trace.

As was described in Section 3.2 of Chapter 3, there is no axis along which a partitioning of the wavefront computation can avoid heavy dependencies across the partitions. Since the high cost of barrier synchronization forbids synchronizing on every one of these dependencies, the coarse-grain implementation must group several of them together and synchronize once for the entire group. This requires that the computations associated with several dependencies be sequenced in such a way that dependencies are enforced. Thus, the processors must wait unnecessarily; this waiting is most visible at the beginning and end of the two wavefront steps in the coarse-grain implementation.

In contrast, one can see in the fine-grain implementation that almost immediately, all 16 processors start doing useful work; idling occurs only when the parallelism is limited at the beginning and the end of the two wavefront steps. No waiting for artificial dependencies occurs. In addition, the amount of waiting at each barrier is greater than the amount of waiting at each JREF miss. This is a reflection of the fact that the barrier is doing global communication whereas each fine-grain synchronization operation is not.

Table 6.1 indicates where performance is being lost by showing a cycle breakdown of the 16x16x16 problem size simulation for both implementations. There are three sources of overhead: waiting for the memory system labeled "cache", waiting for JREFs labeled "JREF", and waiting at barriers. Barrier overhead is further split into two components. The first, "Bar Time," is the number of cycles from when the last thread enters the barrier until the last thread leaves the barrier. The second

Coarse-Grain MICCG3D.					
Processors	Total	Cache	JREF	Bar Time	Bar Skew
1	6943004	568540	N/A	17921	0
		8.19%		0.258%	0.0%
4	2769725	85157	N/A	73799	946855
		3.07%		2.66%	34.19%
16	2428515	179888	N/A	945195	735474
		7.41%		38.92%	30.28%
Fine-Grain MICCG3D.					
Processors	Total	Cache	JREF	Bar Time	Bar Skew
1	6831696	472287	0	1270	0
		6.91%	0.0%	0.019%	0.0%
4	2328728	305456	111627	11636	57288
		13.12%	4.79%	0.50%	2.46%
16	662230	65929	36301	12496	68996
		9.96%	11.55%	1.89%	10.42%

Table 6.1: Cycle breakdown for simulations. Raw numbers are in the units of cycles. Percentages of total execution for each overhead appears below overhead value. Problem size = 16x16x16.

is "Bar Skew" which is the number of cycles from when the first thread arrives at the barrier until the last thread arrives at the barrier. "Bar Time" is a measure of the cost of the barrier operation after all threads have arrived, and "Bar Skew" is a measure of skew in the runtimes of the threads. Both are totals across all barriers.

In Sections 6.1.2, 6.1.3, and 6.1.4, each of these sources of overhead is discussed and their impact is described. In particular, attention is given to how they behave as problem size and machine size are increased beyond what can be simulated. Then, in Section 6.2, the different components of support for fine-grain synchronization discussed in Chapters 2 and 5 are considered, and their impact on the performance of the fine-grain version of MICCG3D is explored in order to better understand the source of the fine-grain implementation's performance advantage.

6.1.2 Memory System Overhead

The overhead of the memory system is consistently one of the smallest overheads in Table 6.1, and the effect of the memory system will be even less at larger problem sizes. In the coarse-grain implementation, the number of remote accesses will stay constant and the number of local accesses will increase as the problem size is increased. Thus, for realistic problem sizes, the fraction of remote memory accesses will tend to zero. In the fine-grain implementation, a new J-structure is allocated on each iteration thereby bypassing the need to reset the J-structure in between iterations. This results in no data reuse. In a real implementation, J-structures will be reset between iterations and reused thus giving rise to better cache performance. One important observation is the cache wait time for the fine-grain implementation will typically be higher than the coarse-grain implementation simply because there are so many more remote data accesses that must be made in the sparse matrix-vector multiply and the solver operations of the fine-grain implementation.

Communication of data in the coarse-grain and fine-grain implementations of MICCG3D is an easy problem for the following two reasons. First, the communication patterns are static and are thus known completely at compile-time. This means optimal partitioning can be found with relative ease. Although the idea has not been investigated, it is expected that very naive prefetching will be successful at eliminating almost all memory system latency. This should make the fact that the fine-grain implementation has more remote-accesses insignificant. Second, there is good physical locality. Most accesses are local, and the communication that needs to occur is always confined to near-neighbor processors. As problem size and machine size grow, this means that the cost of communication will not grow.

6.1.3 Controlling J-Structure Synchronization Overhead

Accessing a J-structure location incurs an overhead in the fine-grain implementation whenever a consumer tries to read a J-structure location not yet filled by its pro-

ducer. For the vector update and inner product operations, processors consume only those values which they produce in previous vector operations, so JREF misses do not occur in these two operations. In the sparse matrix-vector multiply operation, processors occasionally consume values produced by near-neighbor processors in a previous vector operation. Theoretically, JREF misses can occur here; however, these misses are infrequent because a consumer thread would have to be at least an entire vector operation ahead of the producer thread upon which it depends. This case is unlikely. The only JREF misses that have been observed occur in the solver operation.

In the fine-grain implementation of the solver, processor P_i consumes values produced by processor P_{i-1}. It is expected that the number of JREF misses is related to how far producers and consumers are apart in their computations. If producers are well ahead of their consumers, then very few JREF misses will occur. If, however, values are consumed immediately after they are produced, then there is a much greater chance of JREF misses. A graphical analogy can be visualized by recalling Figure 3.2. A low JREF miss rate would correspond to a wavefront that is almost parallel to the x-y plane, whereas a high JREF miss rate would result from a wavefront that has a steeper inclination. The significance of this observation is that JREF miss rate is not dependent on the problem size; rather, it depends only on the relative computational progress that each processor has made with respect to one another.

Using Backoff

The intuition that JREF miss rate is not dependent on problem size is supported by the simulation results reported in Figures 6.4 and 6.5. Simulations were run on 4 and 16 processors while varying the problem size, and the JREF miss rates were recorded under two different failed synchronization policies which are called *spinning* and *backoff*.

When *spinning* on a JREF miss, the processor waiting for the JREF continually spins on the missing value. Once the value gets filled by the producer, the consumer

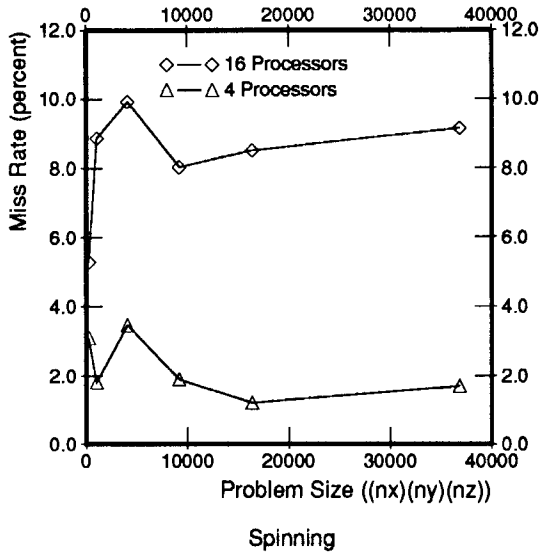

Figure 6.4: JREF miss rate as a function of problem size on the Spinning synchronization failure policy.

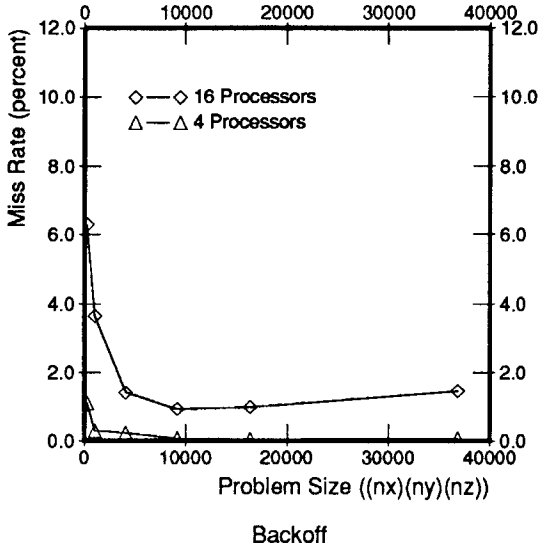

Figure 6.5: JREF miss rate as a function of problem size on the Backoff synchronization failure policy.

immediately reads it and continues computing. This policy will result in a steeper wavefront since it allows consumers to consume values very close to when they are produced. In the *backoff* policy, whenever a processor encounters a failed JREF, it idles for some number of cycles before retrying the read. Backoff allows the producer to make computational progress ahead of the consumer and thus results in a wavefront with a smaller inclination to the x-y plane.

Figures 6.4 and 6.5 verify that backoff achieves a dramatically lower JREF miss rate and also confirms that for non-trivial problem sizes, the miss rate is constant with respect to problem size.

False Sharing

Although the results reported above seem to make intuitive sense, the real benefit of backoff is not obvious. In actuality, the total time spent waiting for JREF misses is higher in the backoff case than it is in the spinning case. This is because with backoff, each JREF miss takes much more time to service than with spinning and more than makes up for the reduction in JREF miss rate. To show this, the number of cycles spent waiting for JREF misses for each datapoint in Figures 6.4 and 6.5 is plotted in Figures 6.6 and 6.7. Notice that for both the 4 and 16 processor simulations, the actual waiting time is always higher when backoff is used as compared to when spinning is used.

However, backoff does better overall because it reduces false sharing. In the fine-grain implementation of MICCG3D, four J-structure locations fit into each cache line. Even if a value that a consumer reads has been filled, the other values belonging to the same cache line may not. If a consumer has a read copy of a cache line containing J-structure elements yet to be filled, the producer will have to send an invalidate to the consumer to retain write permission on the cache line. Then, the consumer will take a cache miss when it tries to read subsequent values on that cache line. This ping-pong effect due to false sharing can happen up to four times per cache line

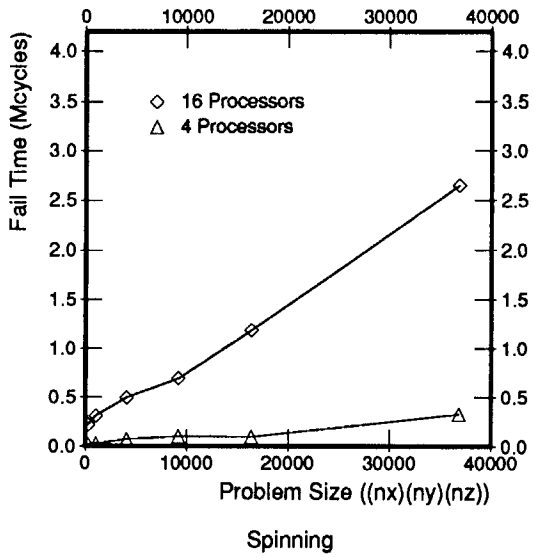

Figure 6.6: Waiting time on JREF misses as a function of problem size on the Spinning synchronization failure policy.

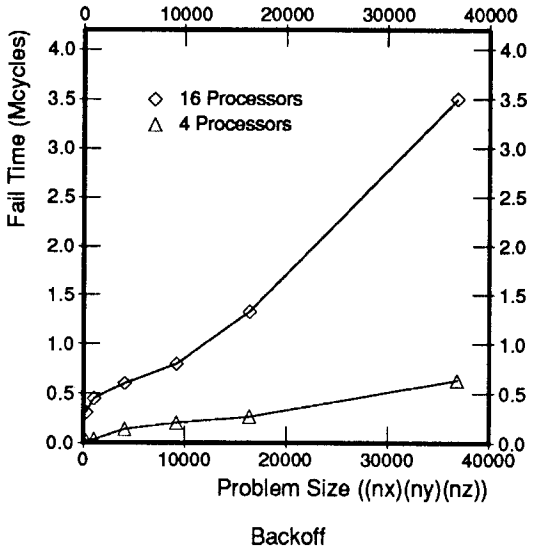

Figure 6.7: Waiting time on JREF misses as a function of problem size on the Backoff synchronization failure policy.

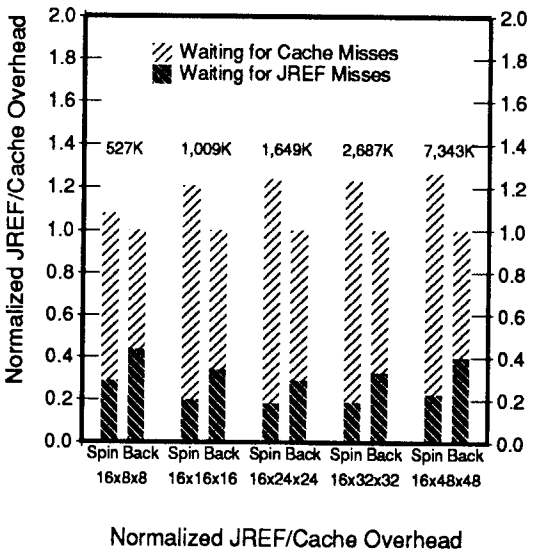

Figure 6.8: The Effect of Backoff on False Sharing.

and can significantly degrade performance. The real benefit of backoff is to hold the waiting processor long enough for the producer to fill at least one cache line. When the consumer comes out of the backoff and retries the JREF, not only will it find that the value it was waiting for has been filled, but it will also have prefetched an entire cache line of filled values for subsequent computation.

This effect of backoff is quantized in Figure 6.8. Here, the five largest problem sizes in Figures 6.4 and 6.5 simulated using 16 processors have been chosen and the amount of waiting due to the memory system and synchronization under both the spinning and backoff policies are shown in a bar graph. Each pair of bars has been separately normalized against the backoff overhead in the pair. The normalizing constant that is used for each pair appears in a label directly above the pair. The label is in the units of thousands of cycles. Notice that for all the simulations, the spinning policy actually has a lower JREF wait component but has a much higher cache wait component due to false sharing. As a result, the backoff policy consistently has the lower overall overhead.

6.1.4 Barrier Overhead

Some of the barrier overhead in Table 6.1 is attributed to synchronization between vector operations. Large amounts of computation separate these barriers, and as problem size is increased, their overhead becomes insignificant. Since this is the only way barriers are used in the fine-grain implementation (for the accumulate in the inner product operations), it is not expected that barrier overhead will be a problem there.

The more serious barrier overhead appears in the coarse-grain implementation of the solver where barriers are used to enforce the dependencies caused by the recurrence relations. To better understand how this overhead effects performance as a function of problem size and machine size, the theoretical speedup in the coarse-grain solver operation, equation 4.1, is rederived below to include barrier overhead. Ignoring communication costs, the time to execute the solver in parallel is

$$T_{par} = \frac{T_{seq}}{S_{upper}} + Bn_B \tag{6.1}$$

T_{par} is the parallel execution time, T_{seq} is the sequential execution time, S_{upper} is the theoretical solver speedup, B is the average cost of a barrier synchronization (includes skew in the runtimes of threads), and n_B is the number of barriers encountered. Using equation 4.1, and recalling from Section 4.1 that $n_B = k+P-1$, equation 6.1 becomes

$$T_{par} = \frac{T_{seq}(k+P-1)}{kP} + B(k+P-1) \tag{6.2}$$

The solver speedup including barrier overhead $S(k,P) = T_{seq}/T_{par}$ is

$$\begin{aligned} S(k,P) &= \frac{T_{seq}}{\frac{T_{seq}(k+P-1)}{kP} + B(k+P-1)} \\ &= \left(\frac{kP}{k+P-1}\right)\left(\frac{T_{seq}}{T_{seq}+BkP}\right) \end{aligned} \tag{6.3}$$

From Figure 4.1, it can be seen that the run-length between barriers is proportional to a single block of computation. Defining this run-length as r_l, it is possible to express

T_{seq} as $T_{seq} = r_l kP$ since there are kP blocks in total. Therefore, equation 6.3 can be rewritten as

$$S(k,P) = \left(\frac{kP}{k+P-1}\right)\left(\frac{r_l}{r_l+B}\right) \tag{6.4}$$

Equation 6.4 expresses the speedup with barrier overhead as the product of the ideal theoretical solver speedup and the overhead of a barrier operation in comparison to the average run-length between barriers. Notice that both these terms cannot be optimized simultaneously. Making k large increases the theoretical speedup term; however, it reduces the run-length between barriers, r_l, which makes the barrier overhead B more significant in the denominator of equation 6.4. Conversely, decreasing k helps the overhead term but lowers the theoretical speedup term.

Projecting Barrier Overhead for Large Problem Sizes

To understand how the overhead term behaves for large problem sizes, one block of computation in the solver (one barrier step) on a 16 processor Alewife machine was simulated and the barrier overhead was observed. Observing the behavior for a single block of computation is equivalent to looking at the behavior for a problem size that is a factor kP larger since there are kP blocks in the solver operation (see Section 4.1). For a decent theoretical speedup, k needs to be $O(P)$; therefore, these simulations predict the barrier overhead for problems approximately P^2 times larger. Even for modest machine sizes, simulating the largest feasible block size is in fact equivalent to looking at fairly large problem sizes.

The result of this experiment appears in Figure 6.9. Various block sizes were run, and the barrier operation cost (appearing as "BAR TIME") and the average skew in the runtimes of threads (appearing as "BAR SKEW") was recorded. Since the cost of a barrier operation depends only on the number of processors, it is constant with respect to problem size. Furthermore, its overhead is significant only at the smallest problem sizes. Skew in the runtimes of threads, however, is a significant source of overhead even at the largest block sizes simulated. Therefore, even for large problem

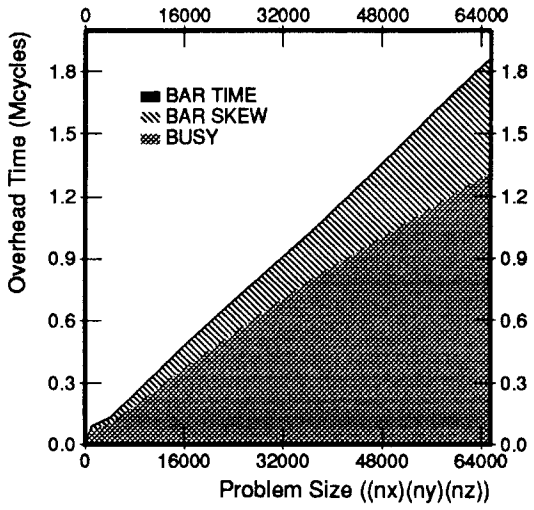

Figure 6.9: Barrier overhead as a function of problem size.

sizes, synchronization overhead due to barriers will remain significant.

6.2 Evaluating the Support for Fine-Grain Synchronization

The results thus far indicate that an implementation of MICCG3D that uses fine-grain synchronization performs better than an implementation that uses coarse-grain synchronization. This thesis will now investigate what contributes to this difference in performance. In particular, what is the impact of cycle efficiency and memory efficiency provided by the Alewife implementation of fine-grain synchronization (discussed in Chapters 2 and 5)?

6.2.1 The Effect of Cycle Efficiency

To understand the effect of cycle efficiency, a 16x16x16 problem size on 4 and 16 processors was simulated, varying the cost of a successful JREF between 1 cycle and

21 cycles. This was accomplished by introducing a variable number of stall cycles in the simulator immediately before each JREF. The results of these simulations appear in Figures 6.10 and 6.11 which show the effect on the solver operation in isolation and on one entire MICCG3D iteration respectively. As one can see, increasing the cost of a successful JREF does not dramatically impact overall runtime. The cost of a successful JREF is simulated out to 21 cycles only to show extreme effects. Any realistic JREF implementation should cost less than 10 cycles.

The degree to which performance will be affected depends on the frequency of JREFs. The more frequent JREFs are, the greater the effect increasing successful JREF cost will have. For MICCG3D on 16 processors, an average JREF frequency of approximately one J-structure reference every 80 cycles has been observed.

The frequency of JREFs is determined by three factors: the amount of computation between JREFs, the number of memory transactions between JREFs for both local and remote data, and the amount of waiting associated with failed synchronization attempts between JREFs. For this particular application, computation between JREFs is minimal, but for each JREF, at least one remote data value needs to be fetched; this cost is significant. Waiting on failed synchronization attempts can also lower the JREF rate significantly. This is why the 4 processor simulation is affected by a higher successful JREF cost more than the 16 processor simulation. The 4 processor case, with fewer synchronization failures, exhibits a much higher JREF rate than the 16 processor case. It is expected that greater synchronization failure rates will be a trend as machine size grows; therefore, on larger machines, it will be more difficult to sustain high JREF rates.

6.2.2 The Effect of Memory Efficiency

The effect of memory efficiency can be measured by implementing J-structures with explicit synchronization variables for each J-structure element instead of using the full-empty bits. This doubles the memory requirement and forces two memory trans-

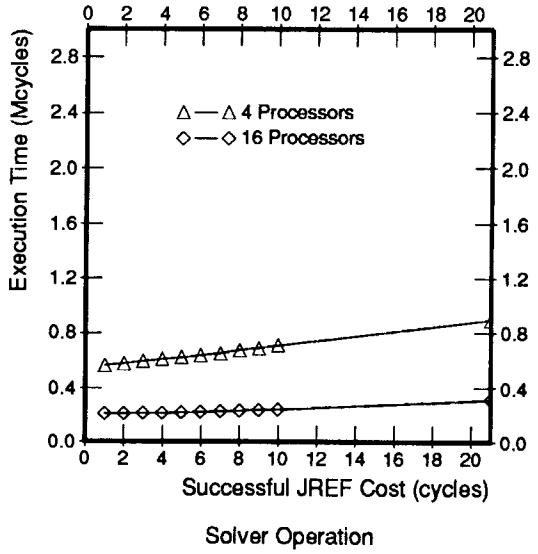

Figure 6.10: Effect of increasing successful JREF cost in the solver operation. Problem size = 16x16x16.

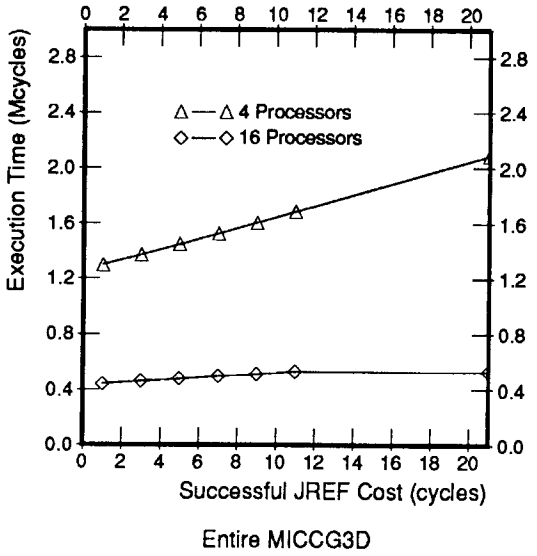

Figure 6.11: Effect of increasing successful JREF cost in an entire MICCG3D iteration. Problem size = 16x16x16.

actions to occur for each JREF. This modification was performed on the solver operation and the resulting performance was compared with the original hardware implementation of J-structures in simulations of 4 and 16 processors on 16x16x16 and 32x32x32 problem sizes each. The results of this experiment appear in Figures 6.12 and 6.13.

In Figure 6.12, execution times are compared. For each machine size and problem size, there is a group of three bars. The "H" bar (for Hardware) uses the Alewife support for J-structures. The "S1" bar (for Software1) uses explicit synchronization variables for each J-structure element, and "S2" (for Software2) uses the same J-structure implementation as "S1" except the cache size has been doubled from 64 KBytes to 128 KBytes. All three bars in the group have been normalized against the "H" execution time; the normalizing factor in raw cycles appears directly above each group (the units are in thousands of cycles).

The software version consistently runs about 35% slower. The 16x16x16 problem size with 16 processors does a bit worse than 35% which will be explained shortly. Notice that doubling the cache size does not help, indicating that the difference in performance is not due to cache pollution from the extra synchronization variables in the software version. This is expected because in MICCG3D, the producer-consumer computation exhibits very little data reuse. In fact, all cache misses to J-structures are cold start misses.

In Figure 6.13, the synchronization and cache overhead components for each of the simulations appearing in Figure 6.12 are shown. Again, times have been normalized against the hardware time in each group of three, and the normalizing factor appears over each group. Notice that cache overhead is significantly higher in the software versions. In general, the JREF overhead for the software versions goes up as well. This is because in order for the backoff mechanism to be effective for the software implementations, the backoff time needs to be increased. This change in JREF overhead is most pronounced in the 16 processor, 16x16x16 problem size simulation. In

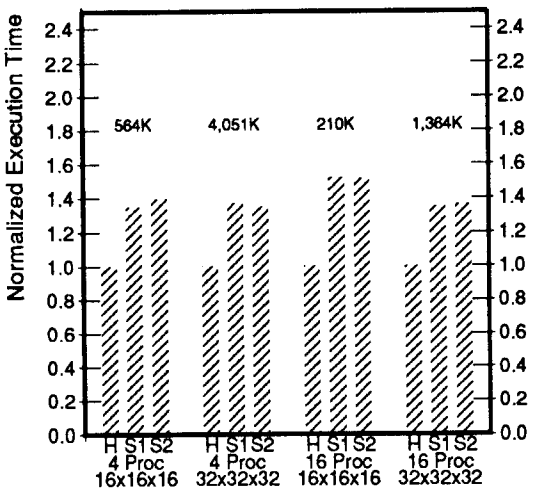

Figure 6.12: Execution times in hardware versus software J-structures in the solver operation. "H" = Hardware, "S1" = Software1, and "S2" = Software2.

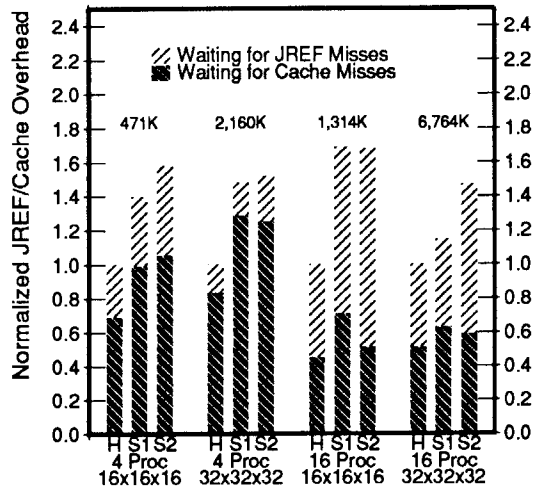

Figure 6.13: Overheads in hardware versus software J-structures in the solver operation. "H" = Hardware, "S1" = Software1, and "S2" = Software2.

this simulation, because of the relatively small problem size for the relatively large machine size, total execution time is very short. This means the JREF overhead is dominated by the JREF misses which occur at the beginning and end of the forward and back substitution steps. In these phases of the solver step, JREF misses are frequent, so the increase in backoff time for the software implementation has a significant impact on total JREF overhead. For more realistic problem sizes, this "JREF cold start" effect will not be significant and instead, a "steady state" JREF miss frequency will be observed. This trend is already visible in the 4 processor 32x32x32 simulation. Since the JREF miss rate is independent of problem size (shown in Section 6.1.3), the asymptotic overhead for real problem sizes will be dominated by the cache overhead.

All of the above data is from the solver operation only. To see how much these results vary for the entire application, the exact experiments were run on one full iteration of the MICCG3D application. The data from these simulations appear in Figures 6.14 and 6.15. Unfortunately, the 32x32x32 problem size proved to be too large to simulate.

Generally, the same trends that were observed in Figures 6.12 and 6.13 are reflected in the data for the entire MICGG3D iteration. The main discrepency is that a greater difference in performance (about 45%-50%) between the software and hardware implementations of J-structures is observed. The explanation for this occurrence is as follows. In the vector operations other than the solver, processors need not fetch data belonging to other processors. In this case, a much higher JREF rate can be achieved because there are no non-local cache misses and no synchronization failures. Thus, the memory efficiency benefit becomes increasingly important.

6.2.3 Interpreting the Fine-Grain Performance Gains

The previous discussion examined in detail the importance of the different components of support for fine-grain synchronization. In this section, a coherent argument is made using data from the previous discussion to convey the fundamental source of the fine-

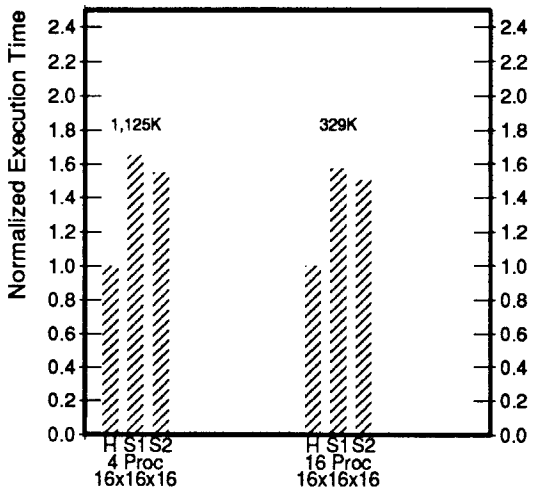

Figure 6.14: Execution times in hardware versus software J-structures in an entire MICCG3D iteration. "H" = Hardware, "S1" = Software1, and "S2" = Software2.

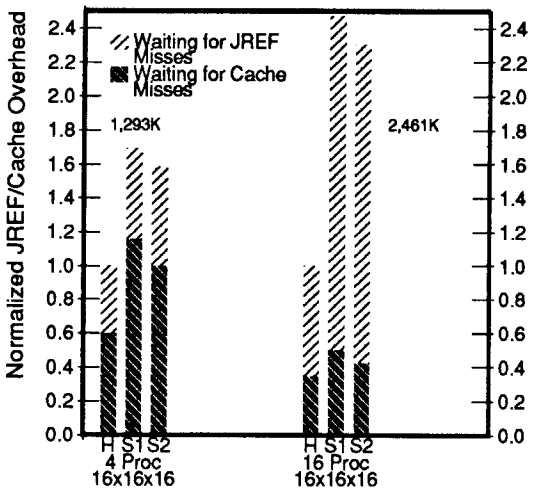

Figure 6.15: Overheads in hardware versus software J-structures in an entire MICCG3D iteration. "H" = Hardware, "S1" = Software1, and "S2" = Software2.

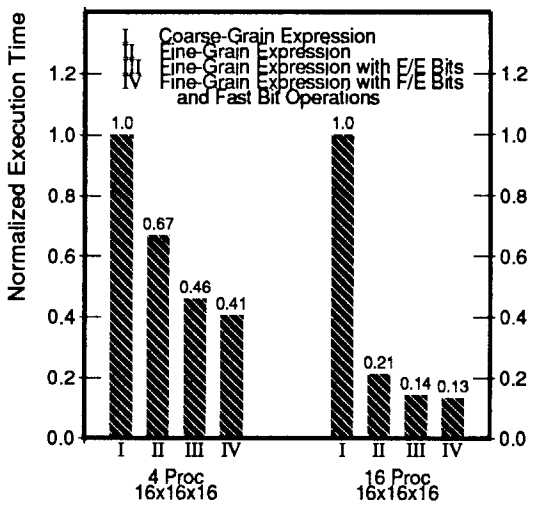

Figure 6.16: Benefits of the fine-grain implementation added incrementally.

grain implementation's performance advantage.

To do this, data is compiled from earlier parts of the thesis to show the increase in performance as the components of support for fine-grain synchronization are added incrementally. Figure 6.16 shows compiled data for 4 and 16 processor machine configurations of normalized execution times on the 16x16x16 problem size of MICCG3D. In each set, Bar I shows the performance of MICCG3D implemented with coarse-grain barriers. Bar II shows the performance when J-structures are implemented without support for cycle efficiency and memory efficiency. Bar III shows the performance when memory efficiency is provided through the use of full-empty bits on J-structures that can be successfully accessed in 5 cycles. Bar IV shows the final addition of cycle efficiency by allowing single cycle successful access to J-structures with full-empty bits (*i.e.*, full Alewife support for fine-grain synchronization).

The cycle efficiency benefit provided by fast operations on full-empty bits gives little performance gain in both cases. The memory efficiency benefit of using full-empty bits provides a more significant gain of about 35% in the solver operation and 45% in an entire iteration of MICCG3D. But the largest gain in performance comes

from the increase in parallelism due to the expressivity of fine-grain synchronization. This is especially true on the larger machine configuration because with more processors, extracting the most parallelism out of the application is more crucial for good performance.

It is important to reflect here that 16 processors is a very modest machine size. With a more ambitious number of processors, it is expected that the sharp difference in performance gains observed for the 16 processor machine will be even more pronounced. Although the difference in performance gains is not as dramatic in the 4 processor machine, notice that even for a trivially small machine size, expressivity is *still* more important than cycle and memory efficiency. Also, the 4 processor data along with the 16 processor data show that expressivity becomes increasingly important in comparison with memory and cycle efficiency as machine size increases. This is in agreement with what is expected.

As was discussed in Section 6.2, the importance of memory and cycle efficiency ultimately depends on the frequency of JREFs. This is determined by the amount of computation, memory system latency, and synchronization failure latency between every JREF. In the context of the experimental environment used in this thesis, MICCG3D running on 16 processors sustained an average JREF rate of about 1 JREF every 80 cycles. This sustained rate can be increased by optimizing for the overheads mentioned. More aggressive compiler optimizations can reduce the cost of the computation, prefetching can reduce the overhead of the memory system, and context switching can hide the latency of synchronization failures. With these optimizations, the cost of successful JREFs should become increasingly important and thus so will the benefits of memory and cycle efficiency.

Chapter 7

Conclusion

Previous application studies have dealt with problems which do not present a challenge for synchronization. To obtain a better understanding of what the synchronization needs of programmers will be, a more comprehensive look into how applications synchronize is needed. MICCG3D, a preconditioned conjugate gradient algorithm using the incomplete Cholesky factorization of the coefficient matrix as a preconditioner, is an application for which synchronization is a challenging problem. It is an important application having received much attention in previous work, and it represents a larger class of preconditioned iterative methods which have traditionally been hard to parallelize. By implementing MICCG3D using both a coarse-grain and fine-grain approach, this thesis shows that the application benefits greatly from fine-grain synchronization.

Due to the recurrence relations in the solver operation, the dependencies that arise in MICCG3D are complex. Consequently, there is no partition of the problem that avoids frequent synchronization. By allowing synchronization at the data-level, fine-grain synchronization provides the expressive power to enforce all the dependencies without introducing significant overhead or sacrificing parallelism. Because coarse-grain synchronization lacks the ability to synchronize at all the dependencies, it must group synchronization points together. Synchronizing at a coarse granularity

sacrifices much of the available parallelism by introducing false dependencies. These false dependencies degrade performance since skew in the runtimes of threads results in unnecessary waiting at the barriers.

7.1 MICCG3D Attains Better Performance with Fine-Grain Synchronization

In problem sizes that were simulated, the implementation using fine-grain synchronization executed 3.7 times faster than the coarse-grain implementation on a 16 processor Alewife machine. Since JREF overhead does not depend on problem size, the fine-grain version should maintain its performance as problem size is scaled. If the coarse-grain implementation is to have any hope of achieving comparable performance at large problem sizes, the run-lengths between barrier points must increase enough to make the overhead of the barriers less significant. However, due to the nature of the dependencies and the fashion in which barriers can be used to enforce them, an increase in the problem size only affects an increase in run-length that is P^2 times smaller. Even for modest machine sizes and especially for large machines, run-lengths should be small enough on realistic problem sizes that barrier overhead will remain significant. Therefore, this thesis projects that the fine-grain implementation will sustain a significant performance advantage over the coarse-grain implementation at large problem sizes.

7.2 Relative Importance of Components of Support for Fine-Grain Synchronization

After evaluating the performance of MICCG3D implemented with both coarse-grain and fine-grain synchronization, the study in this thesis was extended to understand how fine-grain synchronization achieved its performance advantage. First, three com-

ponents of support for fine-grain synchronization were identified and the benefits they provided for applications were enumerated: increased parallelism through expressivity, cycle efficiency, and memory efficiency. Next, the degree to which cycle efficiency and memory efficiency were responsible for the performance gains observed in the fine-grain implementation was ascertained. The conclusion of the thesis is that for the MICCG3D application, cycle efficiency has the least impact while memory efficiency provides a consistent 35% to 45% increase in performance. But the single most important benefit for MICCG3D comes from the ability to express synchronization at a fine granularity. By employing optimizations to reduce the cost of the computation, memory system, and synchronization failures, the contribution of memory and cycle efficiency will become more significant, but in the absence of further evidence, it is the conclusion of this thesis that fine-grain expression of synchronization will remain the most important benefit.

7.3 Directions for Future Work

There are two major points that can potentially influence the conclusions of this thesis. The first is the issue of resetting of J-structures. It is a well-known fact [14] that part of the cost of J-Structures is in resetting them after they have been filled. This is a necessary operation if the J-structures are to be reused in future computations. In the fine-grain implementation used in this study, for simplicity, the J-structure resetting issue was bypassed by allocating enough J-structures during initialization such that each iteration has its own J-structure and thus there is no need to reset any of them. For real problem sizes and a realistic number of iterations, this will be infeasible from the standpoint of memory space. If J-structures have to be reset at each iteration point, it is possible that much of the performance gains in the fine-grain implementation will be loss to this operation.

Although this effect isn't studied in this thesis, there are some comments to be

made in defense of the thesis' conclusions. First, the J-structure resetting operation is one that can be done easily in parallel. Second, each processor should only reset those elements that are local to that processor such that the high latency of interprocessor communication will not be incurred. Both of these considerations tend to lessen the cost of J-structure resetting.

The other important point absent from this study is the investigation of the effect of latency tolerance techniques on the impact of memory and cycle efficiency. Recall from the discussion in Section 6.2.3 of Chapter 6 that the impact of memory and cycle efficiency depends on the frequency of JREFs. For the fine-grain implementation of the MICCG3D application, an average of one JREF every 80 cycles was observed, a relatively low JREF rate. This rate was difficult to increase because of the high latencies of remote memory requests and synchronization failures. If prefetching (which should be pretty easy to perform given the regularity of the access patterns in MICCG3D) is employed to lower the effective latencies of the memory system, and context switching is employed to address synchronization latencies, the JREF frequency will increase and the impact of memory and cycle efficiency will become greater. By how much this impact will increase needs to be quantified through further simulation.

Aside from these two major points, one minor detail left out of this thesis is the issue of effective backoff techniques. Recall in Section 6.1.3 of Chapter 6 that backoff was employed on failed JREF synchronizations in order to alleviate the problem of false sharing in the fine-grain implementation. The amount of backoff is a parameter which can significantly influence performance. Too little backoff will fail to remove the false sharing problem, and too much backoff will result in unnecessary processor idling. For the purposes of this thesis, through trial and error, this backoff parameter was chosen such that it would be optimal for this particular application. In reality, what would be needed is some sort of adaptive backoff approach [16] which would converge on the optimal (or near-optimal) value after some startup time. It would be

interesting to investigate how effective such an adaptive technique would be.

Bibliography

[1] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife Machine: A Large-Scale Distributed-Memory Multiprocessor. *MIT/LCS/TM 454*, MIT Laboratory for Computer Science, June 1991.

[2] David Chaiken, John Kubiatowicz, and Anant Agarwal. LimitLESS Directories: A Scalable Cache Coherence Scheme. *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 224-234, April 1991.

[3] John Kubiatowicz. Users Manual for the Alewife 1000 Controller. *Alewife Systems Memo #19*, MIT Laboratory for Computer Science, December 1991.

[4] Beng-Hong Lim. A Synchronization Library for ASIM. *Alewife Systems Memo #12*, MIT Laboratory for Computer Science, December 1991.

[5] Gail Alverson, Robert Alverson, and David Callahan. Exploiting Heterogeneous Parallelism on a Multithreaded Multiprocessor. *Workshop on Multithreaded Computers, Proceedings of Supercomputing '91*, ACM Sigraph & IEEE, November 1991.

[6] Thomas H. Dunigan. Kendall Square Multiprocessor: Early Experiences and Performance. *Technical Report ORNL/TM-12065,* Oak Ridge National Laboratory, March 1992.

[7] W. J. Dally et al. Architecture of a Message-Driven Processor. *Proceedings of the 14th Annual Symposium on Computer Architecture*, pp 189-196, New York, June 1987.

[8] James R. Goodman and Philip J. Woest. The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor. *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp 422-431, Hawaii, June 1988.

[9] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. *Proceedings*

of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 64-75, Boston, April 1989.

[10] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token-Store Architecture. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, New York, June 1990.

[11] B. J. Smith. Architecture and Applications of the HEP Multiprocessor Computer System. *SPIE*, Vol 298, pp. 241-248, 1981.

[12] Arvind, D. E. Culler, and G. K. Maa. Assessing the Benefits of Fine-Grained Parallelism in Datalflow Programs. *Proceedings of Supercomputing '88*, IEEE, November 1988.

[13] Arvind, Rishiyur S. Nikhil. A Dataflow Approach to General-Purpose Parallel Computing. *MIT CSG Memo 302*, MIT Laboratory for Computer Science, July 7, 1989.

[14] Arvind, and Rishiyur S. Nikhil. I-Structures: Data Structures for Parallel Computing. *ACM Transactions on Programming Languages and Systems*, Vol. 11, No. 4, pp. 598-632, October 1989.

[15] Helen Davis and John Hennessy. Characterizing the Synchronization Behavior of Parallel Programs. *ACM/SIGPLAN Notices*, Vol. 23, No. 9, pp. 198-211, September 1988.

[16] Anant Agarwal and Mathews Cherian. Adaptive Backoff Synchronization Techniques. *Proceedings 16th Annual International Symposium on Computer Architecture*, June 1989.

[17] James R. Goodman. Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors. *Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pp 64-75, April 1989.

[18] Geoffrey C. Fox. What Have We Learnt from Using Real Parallel Machines to Solve Real Problems? *The Third Conference on Hypercube Concurrent Computers and Applications*, Vol. 2, pp. 897-955, January 1988.

[19] Geoffrey C. Fox. Hardware and Software Architectures for Irregular Problem Architectures. *ICASE Workshop on Unstructured Scientific Computation on Scalable Microprocessors*, October 1990.

[20] Geoffrey C. Fox. Lessons from Massively Parallel Applications on Message Passing Computers. *COMPCON 92*, pp. 103-114, 1992.

[21] James Hicks. Report on Running Id Applications on Monsoon. MIT Laboratory for Computer Science, August 20, 1991.

[22] James Hicks, Derek Chiou, Boon Seong Ang, and Arvind. Performance Studies of the Monsoon Dataflow Processor. *MIT CSG Memo 345-1*, MIT Laboratory for Computer Science, September 16, 1992.

[23] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared-Memory, *SIGARCH Computer Architecture News*, Spring 1992.

[24] Norman Rubin. Data Flow Computing and the Conjugate Gradient Method. *MCRC-TR-25*, Motorola Cambridge Research Center, 1992.

[25] David Kranz, Beng-Hong Lim, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. *MIT/LCS/TM 470*, MIT Laboratory for Computer Science, June 1992.

[26] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. Solving Linear Systems on Vector and Shared Memory Computers. *SIAM*, Philadelphia. 1991.

[27] Louis A. Hageman and David M. Young. Applied Iterative Methods. *Academic Press*, New York. 1981.

[28] Gene H. Golub and Charles F. van Loan. Matrix Computations. *Johns Hopkins University Press*, Baltimore, Maryland. 1983.

[29] John C. Strikwerda. Finite Difference Schemes and Partial Differential Equations. *Wadsworth and Brooks/Cole Advanced Books and Software*, Pacific Grove, California.

[30] Henk A. van der Vorst. High Performance Preconditioning. *SIAM Journal of Scientific Statistical Computing*, Vol. 10, No. 6, pp. 1174-1185, November 1989.

[31] Youcef Saad and Martin H. Schultz. Parallel Implementations of Preconditioned Conjugate Gradient Methods. *Research Report YALEU/DCS/RR-425*, Yale University, Department of Computer Science, October 1985.

[32] Henk A. van der Vorst. The Performance of FORTRAN Implementations for Preconditioned Conjugate Gradients on Vector Computers. *Parallel Computing*, Vol 3, pp 49-58, 1986.

[33] Henk A. van der Vorst. A Vectorizable Variant of Some ICCG Methods. *Journal of Scientific and Statistical Computing*, Vol 3, No. 3, pp. 350-356.

[34] Gerard Meurant. Multitasking the Conjugate Gradient Method on the CRAY X-MP/48. *Parallel Computing*, Vol 5, pp 267-280, 1987.

[35] Garrett Birkhoff. The Numerical Solution of Elliptic Equations. *SIAM*, Philadelphia. 1971.

[36] M. K. Jain. Numerical Solution of Differential Equations. *John Wiley & Sons*, New York. 1979.

[37] Kenneth S. Miller. Partial Differential Equations in Engineering Problems. *Prentice-Hall, INC*, New York. 1953.