Experience with Fine-Grain Synchronization in MIMD Machines for Preconditioned Conjugate Gradient

Donald Yeung and Anant Agarwal
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

This paper discusses our experience with fine-grain synchronization for a variant of the preconditioned conjugate gradient method. This algorithm 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, this paper 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?

Our experiments indicate that fine-grain synchronization improves overall performance by a factor of 3.7 on 16 processors using the largest problem size we could simulate; we project that a significant performance advantage will be sustained for larger problem sizes. We also show 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.

1 Introduction

This paper describes an in-depth investigation of the impact of fine-grain synchronization in MIMD machines on the performance of the preconditioned conjugate gradient method. The method uses the modified incomplete Cholesky factorization of the coefficient matrix to form the preconditioner (we will henceforth refer to this application 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 most parallel application studies for MIMD multiprocessors have focused on problems that are uninteresting from the standpoint of synchronization. One of the reasons for this lack of benchmarks is the lack of machines that support fine-grain synchronization. The MICCG3D application meets our criteria because it is an important application with challenging synchronization requirements.

Our investigation of fine-grain synchronization has two major aspects. First, we investigate how fine-grain synchronization can be gainfully employed in MICCG3D and determine quantitatively the resulting performance benefits. We analyze the benefits both with a fixed problem size and when problem size is scaled with machine size. Our study uses a simulator of Alewife, a distributed memory multiprocessor that provides hardware support for the shared-memory abstraction [1]. Our first result 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, we try to understand exactly 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. We demonstrate that the most significant contributions of fine-grain synchronization for MICCG3D do not rely on hardware acceleration; rather, they arise from the expressiveness and flexibility of language-level support.

The rest of this paper proceeds as follows. Section 2 discusses the different styles of synchronization, and identifies the levels of support for fine-grain synchronization that machines can provide. Section 3 describes the MICCG3D application and motivates the need for fine-grain synchronization by discussing why MICCG3D is difficult to parallelize. Section 4 discusses how we actually parallelize MICCG3D using coarse- and fine-grain synchronization. Section 5 describes our experimental environment. Section 6 presents our results and discusses their significance. Finally, Section 7 summarizes our work and makes some concluding remarks.

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 consumer reads a value only *after* it has been written by a producer. Mutual exclusion 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. Because the MICCG3D application uses the producer-consumer style of synchronization, the rest of this paper will only address read-after-write data dependency.

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 subsequent phases. Parallelism is realized within a single phase, but between phases, a barrier is imposed which requires that all work from one phase be completed before the next phase is begun. Under the producer-consumer model, this means that all the consumers in the system must wait for all the producers at a common synchronization point. In contrast, a fine-grain solution provides synchronization at the data level. Instead of waiting on all the producers, fine-grain synchronization allows a consumer to wait only for the data that it is trying to consume. Once the needed data is made available by the producer(s), 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 inhibit parallelism because of unnecessary waiting. The significance of the second benefit is that each fine-grain synchronization operation is much less costly than a barrier. This means that synchronizations can occur more frequently without incurring significant overhead.

It is important to emphasize that these benefits are manifested by the expressiveness of fine-grain synchronization; they will be felt regardless of the underlying hardware implementation. This observation is important because it underscores the fact that fine-grain expression of synchronization and the implementation of synchronization primitives are orthogonal issues.

In this paper, we identify three mechanisms to support fine-grain synchronization. They are:

- Language-level support for the expression of fine-grain synchronization.
- Memory hardware support to compactly store synchronization state.
- Processor hardware support to operate efficiently 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 [3]. The second component of support addresses the fact that an application using fine-grain synchronization will need a large synchronization name space. Providing special synchronization state can lead to an efficient implementation from the standpoint of the memory system. We refer to this benefit as *memory efficiency*. Finally, the last component of support addresses the fact that synchronizations will occur frequently. Therefore, support for the manipulation of synchronization objects can reduce the number of processor cycles incurred. We refer to this benefit as *cycle efficiency*.

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 [5].

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 [4, 6]. Preconditioned CG methods, however, have not enjoyed the same success. In many of the most popular preconditioning techniques, the preconditioner steps involve recurrence relations which do not vectorize or parallelize easily. Algorithmic solutions have been proposed which use different preconditioning techniques to obtain better parallel performance [8, 11, 13, 14]; however, these approaches commonly suffer from reduced convergence rates. Also, some of these algorithms target a specific number of processors and are too complex to generalize to arbitrary machine configurations.

In this paper, we study the preconditioned CG method known as the Modified Incomplete Cholesky Factorization Conjugate Gradient in 3-Dimensions (MICCG3D). Although our study centers around a particular implementation, the general problem being addressed involves increasing parallel performance through the recurrence relations in the preconditioner steps, a problem that is common to almost all preconditioned iterative methods. Therefore, our solution using fine-grain synchronization has general consequences for the large number of algorithms that MICCG3D represents.

3.1 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). Such a matrix commonly arises from the discretization of elliptic partial differential equations. In this paper, we use MICCG3D to solve Laplace's equation in three dimensions.

Discretizing Laplace's equation in three dimensions using a standard 7-point discretization results in a SPD coefficient matrix A that has 7 non-zero diagonals; all other elements are zero. Since matrix A is symmetric, we can write $A = L + diag(A) + L^T$ where L is a lower triangular matrix. Using the incomplete Cholesky factorization method, we obtain an approximate L-U factorization of matrix A which we denote as K. K can be computed as follows and is given in [13].

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

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 1: Cycle breakdown of one iteration of MICCG3D.

In this expression, L is the same lower triangular matrix mentioned above, and D is a diagonal matrix which can be easily computed from matrix A (see [13]). Since K is an approximate factorization of A, we use K^{-1} as the preconditioning matrix and apply the conjugate gradient method to the preconditioned system.

As mentioned earlier, the challenge of MICCG3D lies in parallelizing the vector solution step involving the preconditioner (which we shall refer to as the "solver operation"). Table 1 shows a cycle breakdown for one iteration of MICCG3D on a problem size of $8 \times 8 \times 8$, where the problem size $n_x \times n_y \times n_z$ 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 Section 5. Notice the solver is the most costly vector operation. If poor parallel performance is suffered in this part of the application, the potential parallel performance of the entire application will be severely limited.

3.2 Parallelization Issues

MICCG3D is difficult to parallelize because the recurrence relations in the solver operation impose data dependencies which are numerous and complex. The solver computes w_i, the residual vector in the preconditioned system. w_i is given by

$$\begin{aligned} w_i &= K^{-1}b - K^{-1}Ax_i \\ &= K^{-1}r_i \end{aligned} \tag{2}$$

where x_i is the solution vector of the current iteration step and $r_i = b - Ax_i$ is the residual vector in the original system without preconditioning. Although K^{-1} is the preconditioner, actually calculating it is infeasible because it is the inverse of a sparse matrix (and thus will be dense). Therefore, instead of solving equation 2, we solve

$$Kw_i = r_i \tag{3}$$

Since we have the factorization of K as the product of a lower triangular matrix and an upper triangular matrix (equation 1), we can solve for w_i by first employing back substitution followed by forward substitution. As an example, the backward substitution step can be expressed as follows.

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

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 equation 4 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 1 shows a snapshot of the wavefront in the three-dimensional solution space of MICCG3D.

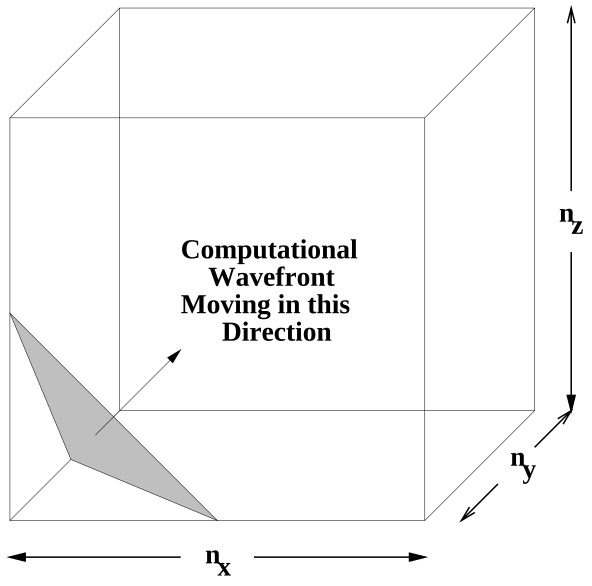


Figure 1: Computational wavefront at an instant in time. n_x, n_y, and n_z denote the discretization degree in the x, y, and z dimensions, respectively.

Machine	Peak MFlops	Vector Update	Inner Product	Sparse MxV	2-Term Recur
CRAY-1	160	46	75	54	10
CRAY-2	1951	71	88	77	8.5
CRAY X-MP	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	800	21	29	27	6.5
NEC SX/2	1300	548	905	843	21
Alliant FX/80	188	17.1	30.5	16.8	3

Table 2: Performance of vector operations in MICCG3D on vector machines. The last column is for the solution of a 2-term recurrence relation.

Wavefront computation in MICCG3D is difficult to parallelize for two reasons. First, the parallelism is not uniform. While there is sufficient parallelism in the middle of the computation, there is very little parallelism at the beginning and the end of the computation. 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 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.

The difficulty in performing computations involving recurrence relations is well known. Table 2 shows performance numbers on some vector computers as presented in [4]. 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

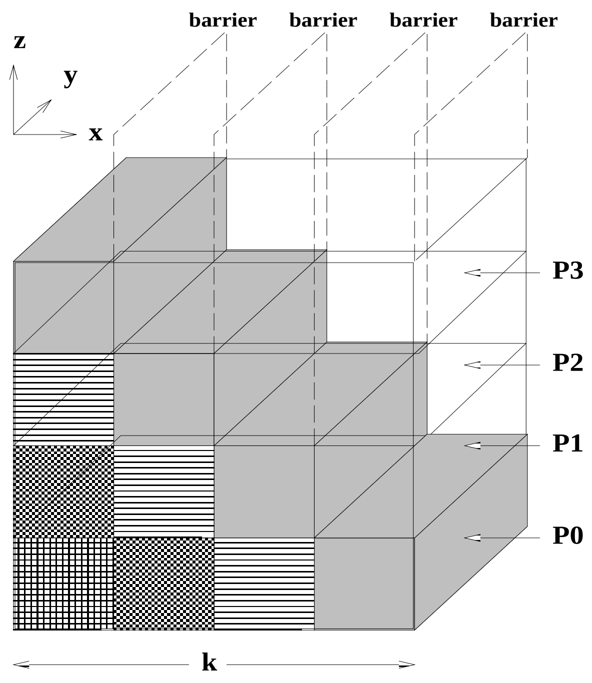


Figure 2: Coarse-grain parallel implementation of the solver operation for 4 processors. Similarly shaded blocks are computed in parallel. Pi denotes processor i, and k is the number of computational blocks in the x-dimension.

MICCG3D. Notice how performance degrades for recurrence relations as compared to the other vector operations. Although only results for 2-term recurrence relations are given, the general trends apply to the solver operation in MICCG3D (which involves a 3-term recurrence relation).

4 Parallel Implementation

In this section, we discuss two ways of parallelizing MICCG3D. One uses coarse-grain barrier synchronization and the other uses fine-grain data-level synchronization. Since we will study these implementations side-by-side in Section 6, it is important that they are in some sense a fair comparison. We believe our two implementations present a fair comparison based on the fact that both require equal programming effort. We do not claim that either implementation is *the best* that can be done; however, we are confident that they are both reasonable implementations.

4.1 Coarse-Grain MICCG3D

In the coarse-grain approach, we partition the solution space along the z dimension and assign n_z/P contiguous planes in the solution space to each processor, where P is the number of processors. To maximize physical locality of data reference, we ensure that all planes assigned to a 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. Furthermore, communication occurs only between adjacent processors and only when computing elements in the solution space which reside in planes at partition boundaries.

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 relation in the solver require further use of barriers. Computation in the solver is further sectioned into k parts along the x dimension. Each processor computes all the results in one block and enters a barrier before it is allowed to move on to the next block. Dependencies between blocks across processors must be enforced by staggering the computation. This process is illustrated in Figure 2 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.

Notice this scheme suffers from limited parallelism. At the beginning of the solver operation, when processor P0 computes its first block, processors P1, P2, and P3 must remain idle. After the first barrier, P0 moves on to its second block, but 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 finer the grain of data associated with each synchronization and thus the smaller the amount of serialization. For all the simulations reported in Section 6, we used $k = P$.

To find an upper bound on speedup in the solver computation, we observe that sequential execution time is proportional to kP, the total number of blocks. Parallel execution time is proportional to the number of blocks per processor, k, added to the number of intervals between barriers each processor spends idling, $P - 1$. Taking the ratio of sequential to parallel execution time gives the upper bound on speedup.

$$S_{upper} = \frac{kP}{k + P - 1} \qquad (5)$$

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 n_z/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. This scheme is illustrated in Figure 3. Notice that compared to the coarse-grain implementation, this partitioning scheme results in substantially more communication in the sparse matrix-vector multiply and solver operations because the computation of every element in the solution space depends on a value belonging to a remote processor.

Synchronization is done at the word-level using fine-grain synchronization. The need for barriers is completely eliminated except in the inner product where an implicit barrier occurs in the accumulate of all the individual scalar multiplies. Word-level synchronization automatically enforces the recurrence dependencies in the solver operation. In the fine-grain version of the solver, each processor can compute results 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 the value becomes available. Therefore, processors never wait unnecessarily.

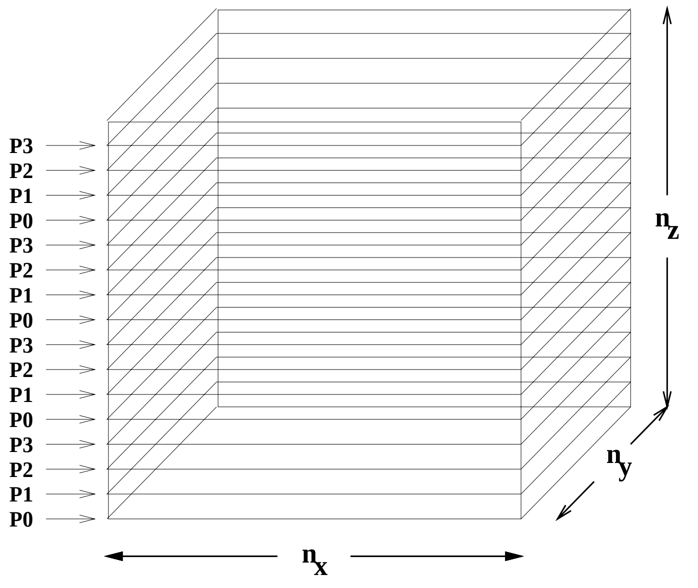


Figure 3: Fine-grain parallel implementation of the solver operation for 4 processors. n_x, n_y, and n_z denote the discretization degree in the x, y, and z dimensions, respectively.

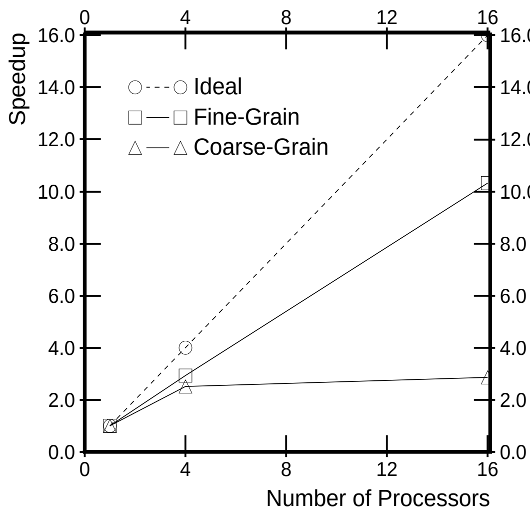


Figure 4: Coarse-grain and fine-grain speedups on MICCG3D. Problem Size = 16 $\times$ 16 $\times$ 16.

For this implementation, the theoretical speedup is linear (*i.e.*, $S_{upper} = P$).

5 Implementation Environment

The results we report in Section 6 are in the context of the Alewife Machine. Alewife consists of a scalable number of homogeneous processing nodes connected in a 2-D mesh topology. The network channels are bidirectional; end-around connections do not exist at the edges of the network. Each Alewife node consists of a 32-bit RISC processor, a floating point unit, 64K bytes of cache memory, 8M bytes of dynamic RAM, and a network routing chip. Although the memory is distributed across processors, a shared-memory abstraction is implemented in hardware which includes support for maintaining cache coherence.

Alewife supports the fine-grain synchronization capabilities described in Section 2. At the language level, L-structure and J-structure constructs allow the expression of synchronization at data-level granularity. L-structures enforce mutual exclusion and J-structures provide producer-consumer synchronization (a detailed discussion on language-level support for fine-grain synchronization in Alewife appears in [7]). Memory efficiency and cycle efficiency as discussed in Section 2 are facilitated by full-empty bits in the memory hardware and fast operations on full-empty bits in the processor hardware, respectively.

Alewife facilitates memory efficiency by allocating a full-empty bit for every word in memory. Each full-empty bit acts as a dedicated hardware synchronization variable. The memory efficiency 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. Second, memory efficiency reduces communication. A synchronized data access on Alewife brings both the datum and the synchronization variable to the processor in one memory system transaction. If the datum and synchronization variable were stored in separate memory locations, two memory transactions would be needed.

Alewife facilitates cycle efficiency by providing special processor hardware that enables the manipulation of a full-empty bit simultaneously with the load or store of its associated data. Thus, a synchronized data access costs no more than a normal load or store. Without hardware support, 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. Alewife uses the result of full-empty tests to conditionally trap the processor. This trapping mechanism is used to identify exceptional cases such as failed synchronizations so they can be handled by software trap handlers. This approach provides efficient hardware support for successful synchronization operations, but relegates the processing of failed synchronizations to less efficient software handlers. The philosophy driving this approach is that successful synchronization operations will be the common case when using fine-grain synchronization.

In Section 6, we will compare the importance of cycle efficiency, memory efficiency, and the benefit of increased parallelism offered by language-level support.

6 Results

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 16 $\times$ 16 $\times$ 16 was used.

Figure 4 shows the speedups calculated by normalizing execution times against the uniprocessor execution time of the coarse-grain implementation. We see 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 show this graphically, we recorded a synchronization trace of both solver implementations in Figure 5. 16

processors are shown executing the solver on a $16 \times 16 \times 16$ problem size. Black bars represent useful work and the interspersed white space signifies waiting for synchronization. In the fine-grain trace, failed J-structure references (which we will refer to as "JREF misses") are traced by a cross appearing above the bar line of the processor that experienced the event. After every JREF miss, processors idle until the desired value is filled by a producer. Comparing the amount of idle time (white space) between the two traces, we see that the coarse-grain implementation waits much more than the fine-grain implementation. (Notice that the two time axes are on different scales; the coarse-grain trace is approximately four times longer than the fine-grain trace).

Table 3 shows a cycle breakdown of the $16 \times 16 \times 16$ 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 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 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 between the runtimes of the threads. Both are totals across all barriers averaged for each processor.

In the next three sections, we discuss each of these sources of overhead and their significance. In particular, we try to extrapolate the behavior as problem size and machine size are increased beyond what we are able to simulate in a reasonable amount of time.

6.1 Memory System Overhead

The overhead of the memory system is consistently one of the smallest overheads in Table 3, and we expect the effect of the memory system to be even less at larger problem sizes. In the coarse-grain implementation, the number of remote accesses will grow with the surface area of each processor partition while the number of local accesses will grow with the volume. Thus, as problem size is increased, the overall cost of memory system overhead will decrease. 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. Moreover, because of the static nature of the computation, we expect very naive prefetching to be effective in hiding most of the memory latency in both implementations.

6.2 Controlling J-Structure Synchronization Overhead

In the fine-grain implementation of the solver, processor P_i consumes values produced by processor P_{i-1}. We expect the number of JREF misses to be related to how far producers and consumers on neighboring processors are apart in their computations. If producers are well ahead of their consumers, then we expect very few JREF misses. If, however, values are consumed immediately after they are produced, then there is a much greater chance for JREF misses. 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 neighboring processors make with respect to one another.

This intuition is supported by the simulation results reported in Figure 6. We ran simulations on 4 and 16 processors while varying the problem size and recorded the JREF miss rates under two different failed synchronization policies which we call *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 immediately reads it and continues computing. This policy 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 a fixed number of cycles before retrying the read. Backoff allows the producer to make computational progress ahead of the consumer. Figure 6 verifies 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. Along with a lower JREF miss rate, backoff has the added benefit of reducing false sharing effects in the cache.

6.3 Barrier Overhead

The most serious barrier overhead reported in Table 3 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, we rederive the theoretical speedup in the coarse-grain solver operation, equation 5, to include barrier overhead. With barrier overhead (but still ignoring communication costs), the time to execute the solver in parallel, T_{par}, is

$$T_{par} = \frac{T_{seq}}{S_{upper}} + Bn_B \tag{6}$$

where 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), and n_B is the number of barriers encountered per processor. Using equation 5, we get

$$T_{par} = \frac{T_{seq}(k+P-1)}{kP} + B(k+P-1) \tag{7}$$

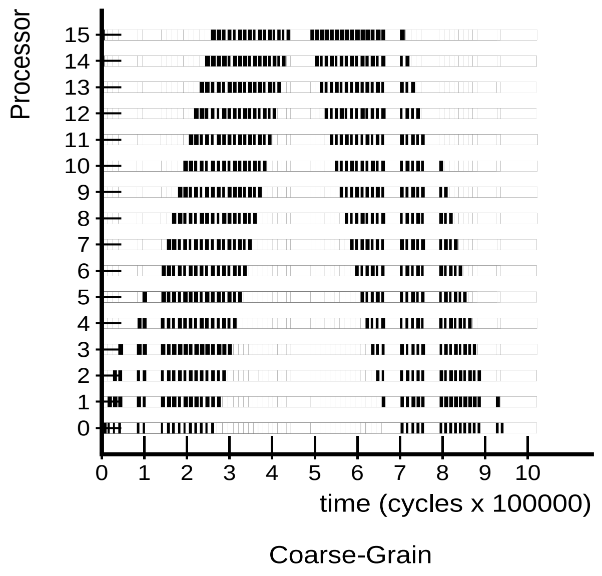
where we have used $n_B = k + P - 1$ by recognizing that n_B is equivalent to the number of blocks encountered by each processor as discussed in Section 4.1. The solver speedup including barrier overhead $S(k,P) = T_{seq}/T_{par}$ is

$$\begin{aligned} S(k,P) &= \frac{T_{seq}}{\frac{T_{seq}(k+P-1)}{kP} + B(k+P-1)} \\ &= \left(\frac{kP}{k+P-1}\right)\left(\frac{T_{seq}}{T_{seq}+BkP}\right) \end{aligned} \tag{8}$$

From Figure 2, we see that the run-length between barriers is proportional to a single block of computation. If we define this run-length as r_l, then we can express T_{seq} as $T_{seq} = r_l kP$ since there are kP blocks in total. Therefore, we can rewrite equation 8 as

$$S(k,P) = \left(\frac{kP}{k+P-1}\right)\left(\frac{r_l}{r_l+B}\right) \tag{9}$$

Equation 9 is 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



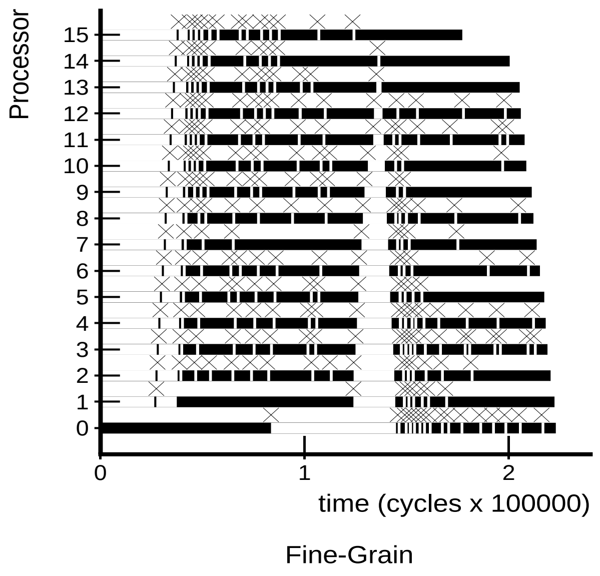
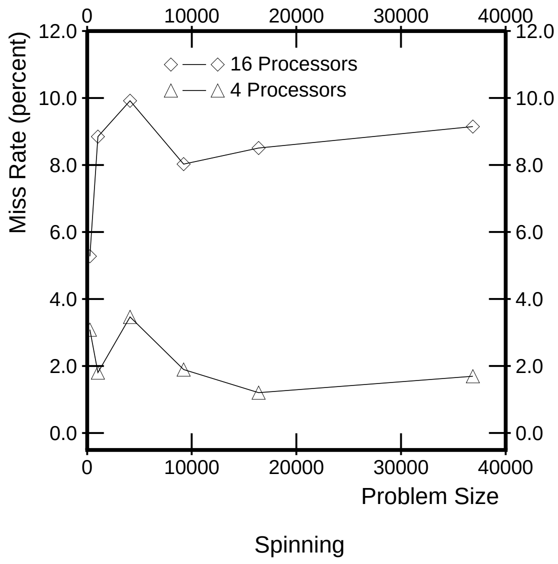


Figure 5: Traces in the solver operation of MICCG3D. Problem size = $16 \times 16 \times 16$. Bar lines show useful work, and white space shows waiting for synchronization. Crosses indicate JREF misses. The staircase shape is the signature of the back and forward substitution steps.

Coarse-Grain MICCG3D.									
Processors	Total	Cache	%	JREF	%	Bar Time	%	Bar Skew	%
1	6943004	568540	8.19	N/A	N/A	17921	0.258	0	0.0
4	2769725	85157	3.07	N/A	N/A	73799	2.66	946855	34.19
16	2428515	179888	7.41	N/A	N/A	945195	38.92	735474	30.28

Fine-Grain MICCG3D.									
Processors	Total	Cache	%	JREF	%	Bar Time	%	Bar Skew	%
1	6831696	472287	6.91	0	0.0	1270	0.019	0	0.0
4	2328728	305456	13.12	111627	4.79	11636	0.50	57288	2.46
16	662230	65929	9.96	36301	11.55	12496	1.89	68996	10.42

Table 3: Cycle breakdown for simulations. Problem size = $16 \times 16 \times 16$. "Cache" is waiting on the memory system, "JREF" is waiting on failed J-structure references, "Bar Time" is the cost of all the barriers (without skew), and "Bar Skew" is the total skew between the runtimes of the threads at all the barriers.



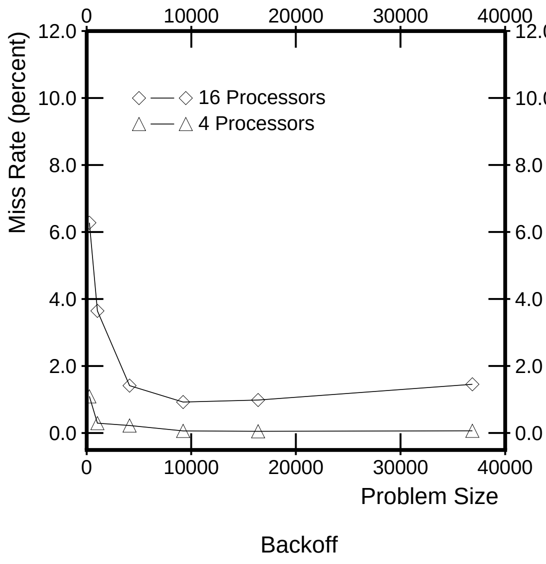


Figure 6: JREF miss rate as a function of problem size in the Spinning and Backoff synchronization failure policies. Problem size is given by $n_x \times n_y \times n_z$ where n_x, n_y, and n_z are the degree of discretization in the x, y, and z dimensions, respectively.

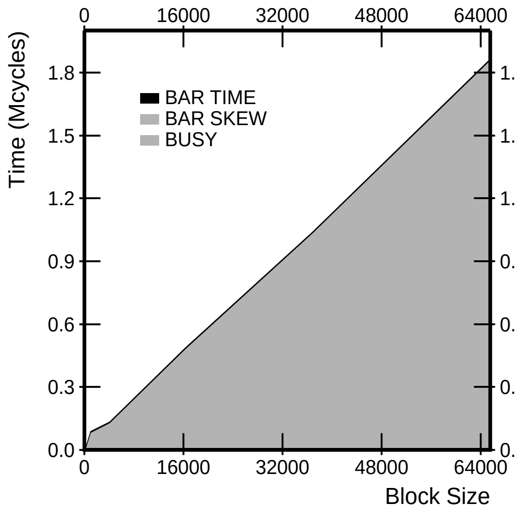


Figure 7: Barrier overhead as a function of block size for 16 processors. Block size is given by $n_x \times n_y \times n_z$ where n_x, n_y, and n_z are the number of elements in one block in the x, y, and z dimensions, respectively.

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. Similarly, decreasing k helps the overhead term but lowers the theoretical speedup term.

To understand how barrier overhead behaves for large problem sizes, we simulated one block of computation in the solver on a 16 processor Alewife machine. Observing the barrier overhead for a single block is equivalent to looking at the total barrier overhead for a problem size that is a factor kP larger since there are kP blocks in the entire coarse-grain solver operation (see Section 4.1). Even for modest values of P and k, 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 7 which shows the average barrier operation cost (appearing as “BAR TIME”) and the average skew in thread runtimes (appearing as “BAR SKEW”) for various block sizes. Since the cost of a barrier operation depends only on the number of processors, it is constant with respect to block size and is trivial for most block sizes. Skew, however, is a significant source of overhead even at the largest block sizes simulated (roughly 30%). We have observed that skew in MICCG3D is due to nonuniform cache hit rates and network latency across the machine and not due to load imbalance.

We draw two conclusions from Figure 7. First, the barrier overhead of 68% reported for our 16 processor simulation in Table 3 is pessimistic and is due to the fact that we can only simulate small problem sizes. Second, although we expect better performance as problem size is scaled, synchronization overhead will still be significant at large problem sizes.

6.4 Evaluating Support for Fine-Grain Synchronization

Our results thus far indicate that an implementation of MICCG3D that uses fine-grain synchronization performs better than an implementation that uses coarse-grain synchronization. We will now investigate the causes of this difference. In particular, we investigate the impact of cycle efficiency and memory efficiency provided by the Alewife implementation of fine-grain synchronization (discussed in Sections 2 and 5).

To understand the effect of cycle efficiency, we simulated a $16 \times 16 \times 16$ problem size on 4 and 16 processors, varying the cost of a successful JREF between 1 cycle and 21 cycles. This was accomplished by artificially introducing stall cycles immediately before each JREF. The results of these simulations appear in Figure 8 and show the effect on both the solver operation in isolation and on one entire MICCG3D iteration. As we can see, increasing the cost of a successful JREF does not dramatically impact overall runtime. We simulated the cost out to 21 cycles only to show extreme effects. We expect any realistic JREF implementation to cost less than 10 cycles. Notice the 4 processor simulation is more sensitive to the cost of a successful JREF than the 16 processor simulation (exhibited by a steeper slope). This is because for a fixed problem size, there are more JREFs in the critical path of execution for smaller machine sizes; thus, a greater increase in execution time will result from a given increase in the cost of each JREF.

The degree to which performance is affected by the cost of a successful JREF 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, we have observed frequencies of approximately 1 JREF every 80 cycles. We were surprised to find such a low JREF frequency.

The frequency of JREFs is determined by three factors: the amount of computation between JREFs, the amount of waiting on the memory system between JREFs, and the amount of waiting on failed synchronizations 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 also lowers the JREF rate significantly. This effect tends to be greater on large machines since the number of failed JREF attempts increases with more processors. We expect greater synchronization failure rates to be a trend as machine size grows; therefore, on larger machines, it will be more difficult to sustain high JREF rates.

The effect of memory efficiency can be measured by implementing J-structures with explicit synchronization variables for each J-structure element in place of the full-empty bits. This doubles the memory requirement and forces two memory transactions to occur for each JREF. We performed this modification to the solver operation and compared its performance with the original full-empty bit implementation of J-structures in simulations of 4 and 16 processors on $16 \times 16 \times 16$ and $32 \times 32 \times 32$ problem sizes each. The results of this experiment appear in Figure 9.

In the left graph of Figure 9, we compare execution times. 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 64K bytes to 128K bytes. All three

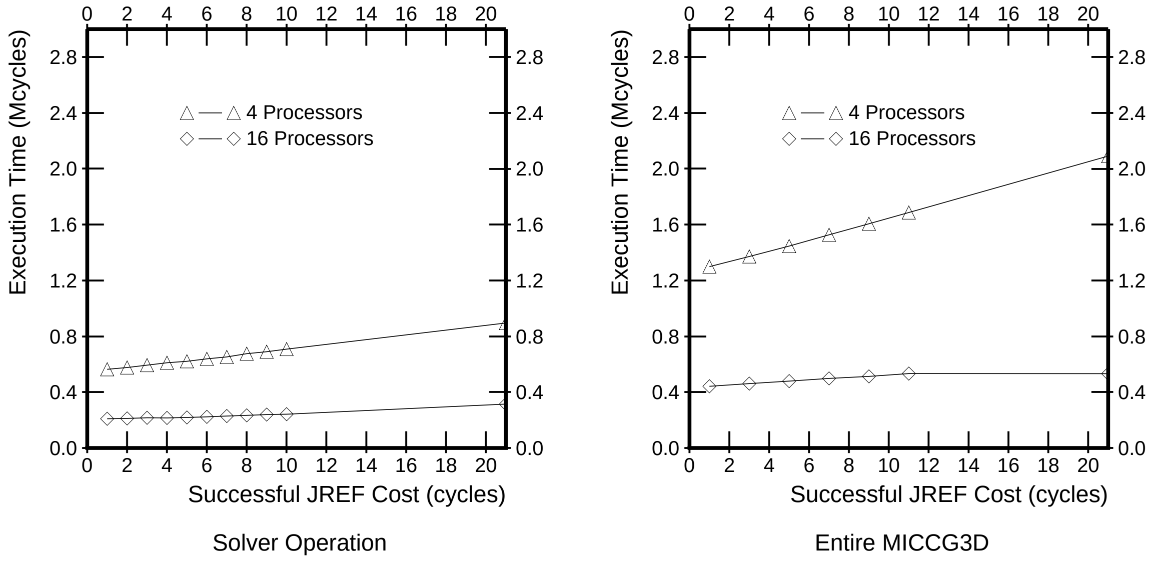


Figure 8: Effect of increasing successful JREF cost in both the solver operation and an entire MICCG3D iteration. Problem size = $16 \times 16 \times 16$.

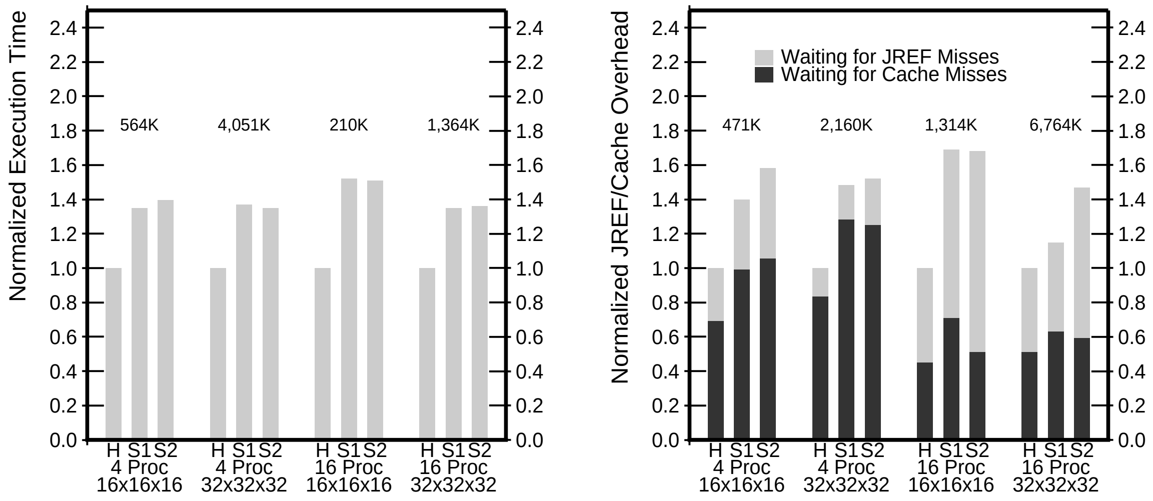


Figure 9: Execution times and overheads in hardware versus software implementations of J-structures in the solver operation. The "H" bars (for "Hardware") use full-empty bit support, the "S1" bars (for "Software1") use explicit software variables, and the "S2" bars (for "Software2") use the same implementation as the "S1" bars except cache size has been doubled to study the effects of cache pollution.

bars in the group have been normalized against the "H" execution time; the normalizing constant in raw cycles appears directly above each group (the units are in thousands of cycles). We see that the software version consistently runs about 35% slower. 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 the right graph of Figure 9, the synchronization and cache overhead components for each of the simulations appearing on the left graph are shown. Again, times have been normalized against the hardware time in each group of three, and the normalizing constant appears over each group. Notice that indeed, 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 (discussed in Section 6.2) 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, $16 \times 16 \times 16$ problem size simulation. In 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 back and forward 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, we should see a "steady state" JREF overhead. This trend is already visible in the 4 processor $32 \times 32 \times 32$ simulation. Since the JREF miss rate is independent of problem size (shown in Section 6.2), the asymptotic overhead for real problem sizes will be dominated by the cache overhead.

6.5 Interpreting the Fine-Grain Performance Gains

The discussion in the previous section examined in detail the impact of two components of support for fine-grain synchronization on application performance, those two which involve hardware-level support. In this section, we consider the essential results of this study and relate it to the importance of language-level support.

We collect data from earlier parts of the paper to show the increase in performance of the MICCG3D application as the components of support for fine-grain synchronization are added incrementally. Figure 10 shows normalized execution times for the $16 \times 16 \times 16$ problem size of MICCG3D. Two sets of data are shown, one for 4 processors and one for 16 processors. In each set, bar I shows the execution time of MICCG3D implemented with coarse-grain barriers. Bar II shows the execution time when J-structure style synchronization is used; however, the J-structures are implemented purely in software without the benefit of cycle efficiency and memory efficiency. Bar III shows the execution time when the J-structures are supported by full-empty bits that can be accessed in 5 cycles thus providing memory efficiency (but not cycle efficiency). Finally, bar IV shows the addition of cycle efficiency by allowing single cycle access to full-empty bits (*i.e.,* full Alewife support for fine-grain synchronization).

The impact of cycle efficiency is quantified by the difference in

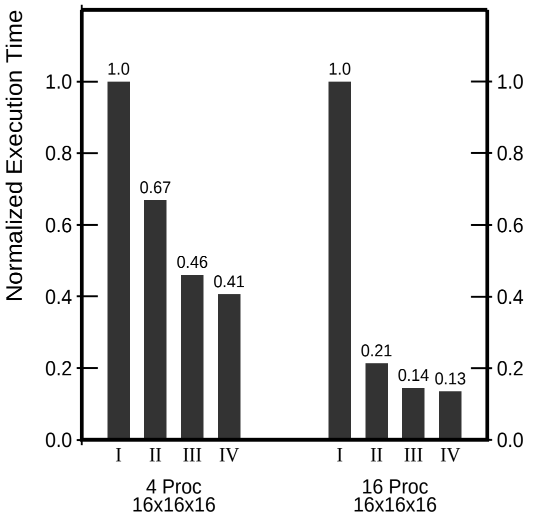


Figure 10: Benefits of the fine-grain implementation added incrementally. Bar I uses only coarse-grain expression, bar II allows fine-grain expression, bar III allows fine-grain expression with full-empty bit support, and bar IV allows fine-grain expression with full-empty bits and fast bit operations.

execution times between bars III and IV. In both sets of data, this performance gain is small, on the order of 10%. The impact of memory efficiency is quantified by the difference between bars II and III, and indicates a more significant performance improvement of about 40%. But by far the largest gain in performance comes from increased parallelism due to the expressiveness of fine-grain synchronization as quantified by the difference between bars I and II. The two sets of data together also show that as machine size is scaled, fine-grain expression of synchronization becomes even more critical in improving application performance in comparison to memory and cycle efficiency. Notice that 16 processors is still a very modest machine size. With further machine scaling, we expect that the sharp difference in performance gains observed for the 16 processor data set will be even more pronounced.

There are two issues which help define the bounds within which this conclusion is valid. First, we recognize that this discussion has only considered machine scaling on a fixed problem size. This is important if we are interested in running a *given problem* as fast as possible. What if we are interested in scaling problem size while keeping machine size fixed? We expect that a larger problem size will improve the execution time of the coarse-grain implementation more than the fine-grain implementation, so the dramatic difference between bars I and II should decrease with problem size scaling. The question is, how much will it decrease? Recall from Section 6.3 that even for fairly large problem sizes, barrier overhead will remain significant due to skew in the runtimes of threads. Therefore, we predict that limited parallelism will remain a problem in the MICCG3D application and fine-grain expression of synchronization will continue to be important.

Second, we recognize that the importance of memory and cycle efficiency ultimately depends on the frequency of JREFs (recall the discussion in Section 6.4). JREF frequency is determined by the amount of computation, memory system latency, and synchronization failure latency between every JREF. Therefore, the JREF frequency can be increased by employing the following optimizations. More aggressive compiler optimizations can reduce the cost of the computation, prefetching can reduce the overhead of the memory system, and multithreading combined with fast context switching can hide the latency of synchronization failures. The extent to which these optimizations will influence the importance of memory and cycle efficiency relative to fine-grain expression requires further study.

7 Summary

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 [8, 11, 13, 14], and it represents a larger class of preconditioned iterative methods which have traditionally been hard to parallelize. By implementing MICCG3D using both a coarse- and fine-grain approach, we have discovered that the application benefits greatly from fine-grain synchronization.

In problem sizes that we simulated, we observe that 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, we expect the fine-grain version to maintain its performance as problem size is scaled. Although the barrier overhead in the coarse-grain implementation will improve as problem size is scaled, simulations show that skew in the runtimes of threads will remain significant for realistic problem sizes. Fundamentally, this is due to the fact that an increase in problem size only affects an increase in run-length between barriers that is kP times smaller. Even for modest machine sizes, and especially for large machines, we expect run-lengths to be small enough on realistic problem sizes that skew at the barriers will remain significant. Therefore, we anticipate that the fine-grain implementation will sustain a significant performance advantage over the coarse-grain implementation at large problem sizes.

After evaluating the performance of MICCG3D implemented with both coarse- and fine-grain synchronization, we extended our study to understand how fine-grain synchronization achieves its performance advantage. First, we identified three components of support for fine-grain synchronization and enumerated the benefits they provide for applications: increased parallelism through expressiveness, cycle efficiency, and memory efficiency. Next, we ascertained the degree to which each of these benefits were responsible for the performance gains we observed in the fine-grain implementation of MICCG3D. Our conclusion is that for the MICCG3D application, cycle efficiency has the least impact while memory efficiency provides a more significant 40% increase in performance. But by far the most important benefit for MICCG3D is the ability to increase parallelism by expressing synchronization at a fine granularity. By employing optimizations to reduce the cost of the computation, memory system, and synchronization failures, we expect the contributions of memory and cycle efficiency to become more significant, but further study is needed to quantify this effect.

8 Acknowledgments

This research was funded in part by NSF grant # MIP-9012773, in part by DARPA contract # N00014-87-K-0825, and in part by a NSF Presidential Young Investigator Award. We would also like to acknowledge Norm Rubin [10] for inspiring us to use the MICCG3D application for our study.

References

[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] 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.

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

[4] 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.

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

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

[7] David Kranz, Beng-Hong Lim, and Anant Agarwal. Low-Cost Support for Fine-Grain Synchronization in Multiprocessors. Multithreading: A Summary of the State of the Art, *Kluwer Academic Publishers*, 1992. Also available as *MIT/LCS/TM 470*, 1992.

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

[9] G. M. Papadopoulos and D. E. Culler. Monsoon: An Explicit Token-Store Architecture. *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pp. 82-91, New York, June 1990.

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

[11] 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.

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

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

[14] 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.