October 1990 Report No. STAN-CS-90- 1330

Parallel ICCG on a Hierarchical Memory **Multiprocessor-** Addressing the Triangular Solve Bottleneck

by

Edward Rothberg, Anoop Gupta

Department of Computer Science

Stanford University

Stanford, California 94305

Parallel ICCG on a Hierarchical Memory Multiprocessor — Addressing the Triangular Solve Bottleneck

Edward Rothberg and Anoop Gupta
Department of Computer Science
Stanford University
Stanford, CA 94305

September 12, 1990

Abstract

The incomplete Cholesky conjugate gradient (ICCG) algorithm is a commonly used iterative method for solving large sparse systems of equations. In this paper, we study the parallel solution of sparse triangular systems of equations, the most difficult aspect of implementing the ICCG method on a multiprocessor. We focus on shared-memory multiprocessor architectures with deep memory hierarchies. On such architectures we find that previously proposed parallelization approaches result in little or no speedup. The reason is that these approaches cause significant increases in the amount of memory system traffic as compared to a sequential approach. Increases of as much as a factor of 10 on four processors were observed. In this paper we propose new techniques for limiting these increases, including data remappings to increase spatial locality, new processor synchronization techniques to decrease the use of auxiliary data structures, and data partitioning techniques to reduce the amount of inter-processor communication. With these techniques, memory system traffic is reduced to as little as one sixth of its previous volume. The resulting speedups are greatly improved as well, although they are still much less than linear. We discuss the factors that limit further speedups. We present both simulation results and results of experiments on an SGI 4D/340 multiprocessor.

1 Introduction

Iterative methods are frequently used to solve the large sparse linear systems that arise when numerically solving partial differential equations. They require much less storage space than direct methods, and they are usually more efficient for the matrices generated in practice. The primary difficulty with iterative methods, in general, is the issue of convergence. The successive approximations to the solution that an iterative method generates may not approach the true solution to the system quickly enough. One method that has been shown to have good convergence properties, both theoretically and empirically, and that is used quite frequently in practice is the preconditioned conjugate gradient method. A particularly popular preconditioner is the incomplete Cholesky preconditioner. However, even with an efficient iterative method like the incomplete Cholesky conjugate gradient (ICCG) method, the solution of large sparse systems of equations is still an extremely time-consuming computation. In this paper, we explore the use of high-performance shared-memory multiprocessors to speed up this computation.

The computation performed within each iteration of the ICCG method involves several substeps, including a sparse matrix vector multiplication and the solution of two sparse triangular systems, and a number of DAXPY's and dot products. While most of these substeps are straightforward to parallelize, the solution of the sparse triangular systems is much more difficult. Since these triangular system solves account for roughly half of the per-iteration nmtime, it is important to effectively parallelize them. A number of methods for solving sparse triangular systems on parallel machines have been proposed [2, 9, 10, 14]. We study the performance of the two primary methods, the *level-scheduled method* [2], and the *self-scheduled method* [14]. We find that although these methods were originally reported to give substantial speedups, neither produces good speedups on the Silicon Graphics 4D/340, a high-performance, 4 processor, shared-memory multiprocessor with a deep memory hierarchy. The best speedups obtained with either of these two methods on 4 processors were between 0.6 and

1.1 for a variety of matrices. Through detailed multiprocessor simulation, we study these **two** methods in order to better understand their behavior and to evaluate modifications to these approaches.

We find that a major obstacle in achieving high performance with these two methods is the enormous amounts of memory system traffic they generate. The parallel approaches running on 4 processors generate as much as 10 times more traffic than the sequential code. This increase leads to more processor time spent waiting for cache misses to be serviced and also leads to saturation of the shared multiprocessor bus. We find that the increase in traffic is primarily due to a loss of spatial locality, which decreases the effectiveness of the caches. With these parallel system solving methods, a processor frequently accesses a particular memory location, but does not subsequently access adjacent locations. Spatial locality is not the only source of increased traffic; we identify a number of other sources, including traffic associated with the use of auxiliary data structures and communication of computed values between processors. We suggest a number of modifications that address each of these sources of increased traffic. The result of these modifications is substantially higher performance; we obtain speedups of between 1.7 and 2.2 on 4 processors. We describe the obstacles that still remain after our modifications.

The rest of this paper is organized as follows. Section 2 gives background material. It first discusses the computation necessary to perform the ICCG method. It then describes the machine on which we perform our experiments, and discusses the matrices we use as benchmarks. Section 3 describes the algorithm for solving a triangular system on a sequential machine and then describes the level-scheduled and self-scheduled methods for performing the computation on a parallel machine. Performance figures are presented for each of these parallel methods. In section 4, we study the performance of the parallel methods in further depth and discuss the problems that are present in each. We describe a number of modifications that improve parallel performance. In section 5 we discuss the problems that remain after these modifications are made, and we also give a brief discussion. Conclusions are presented in section 6.

2 Background

We begin by presenting some background material, including a brief discussion of the ICCG computation and a description of the parallel machine and the benchmark matrices we use to evaluate performance.

2.1 The ICCG method

In this subsection, we briefly describe the incomplete Cholesky conjugate gradient method for solving a sparse system $Au = f$, where A is an n x n sparse matrix, f is a vector of length n, and u is an unknown vector, also of length n. We discuss only the computation involved in the ICCG method; for a description of the conjugate gradient algorithm, the reader is referred to [6, 11, 12].

The conjugate gradient method was originally developed by Hestenes and Stiefel [11]. It is an iterative method; an approximation $u^{(i)}$ is refined at each iteration until the approximation is sufficiently close to the true solution, at which point the method is said to have converged. Convergence can be improved by *preconditioning* the system, where the conditioned system $M^{-1}Au = M^{-1}f$ is solved instead. A popular preconditioner is the incomplete Cholesky preconditioner [12], where an incomplete factorization of A, $A \approx L\,DL^T$, is used to determine the preconditioning matrix $M = LDL^T$. The no-fill incomplete factor L is determined by performing a Cholesky factorization of A while discarding any component of the factor that would result in fill in L. *The* incomplete Cholesky preconditioner roughly doubles the amount of work that must be done per iteration of the conjugate gradient method. Convergence is typically reached in many fewer than half as many iterations, thus making the preconditioner a valuable enhancement to the conjugate gradient method.

The ICCG method has two primary phases. The preprocessing phase is performed once. The main computation that must be done during this phase is the incomplete factorization of A into $L\,DL^T$. Once preprocessing has been completed, then the iteration phase is performed. This phase iteratively improves the approximation to the solution. Since the ICCG method typically spends most of its time in the iteration phase, in this paper we concentrate on the parallelism available in this phase. Our goal is to minimize the cost of a single iteration. A number of the modifications we propose increase the amount of time spent in preprocessing, but decrease the amount of time per iteration.

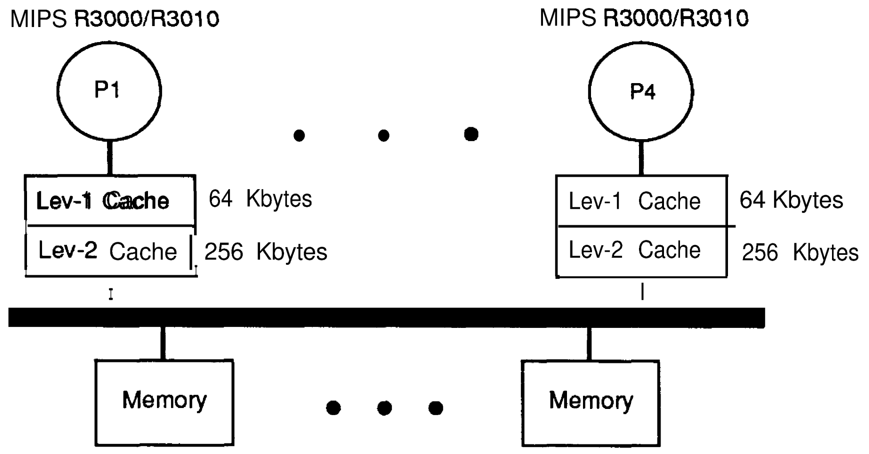


Figure 1: Overview of the SGI 4D/340 multiprocessor architecture.

The computation associated with each iteration of the non-preconditioned conjugate gradient involves one matrix-vector multiply, three inner products, and two DAXPYs. Preconditioning adds a system solve $Mx = b$ to each iteration, where M is the preconditioning matrix. For the incomplete Cholesky preconditioner, $M = LDL^T$, so the system to be solved is $LDL^T x = b$. This is done by solving three systems, $Ly = b$, $Dz = y$, and $L^T x = z$. Note that the systems involving L are both triangular, while the one involving diagonal matrix D is trivially solvable.

If we consider performing the incomplete Cholesky conjugate gradient method on a parallel machine, we see that most of the computation is easily distributed. The matrix-vector multiply is not difficult to parallelize, nor are the inner products or the DAXPYs. The most difficult component to distribute is the pair of triangular systems solves that result from the incomplete Cholesky preconditioner. These system solves comprise roughly half of the total ICCG runtime, so they must be effectively parallelized if we are to obtain significant parallel speedups for the ICCG computation as a whole.

2.2 Multiprocessor Architecture and Benchmark Matrices

The multiprocessor we use to compare the performance of parallel triangular system solving approaches is the SGI 4D/340 [4]. This machine is a 4 processor, bus-based shared-memory multiprocessor. Each processor has two levels of cache memory, and the caches are kept coherent with an invalidation-based protocol [1]. The high-level organization of the machine is shown in Figure 1. The processors are 33MHz MIPS R3000/3010 pairs, and are nominally rated at 27 MIPS and 4.9 double-precision UNPACK MFLOPS.

A reference that misses in both levels of a processor's cache is filled from main memory or from the cache of another processor. The cache line size is 16 words[1] and the cost of servicing a miss is roughly 50 cycles. The cache lines are quite long in order to amortize the large fixed cost associated with servicing a miss. Large cache lines can greatly reduce the effective cache miss cost of a program that exhibits *spatial locality,* that is if there is a high likelihood that data items physically near the current miss location will be referenced soon after. The fact that programs typically exhibit a high degree of spatial locality, combined with the fact that the fixed cost associated with servicing a cache miss will continue to increase as processors become faster, make it quite likely that cache lines will be large in future machines as well.

The shared bus on the SGI 4D/340 has a peak bandwidth of roughly 67 MBytes/second. Four processors are easily capable of saturating the bus. We found that the sequential code for solving triangular systems generated roughly 17 MBytes per second of traffic on the bus, so we would not expect more than four processors to be productively employed on the same bus for this problem. We would also expect four to be productive only if the parallel and sequential codes generate roughly the same amount of traffic.

In order to obtain a more detailed understanding of the performance of both the sequential and parallel

[1]More accurately, four adjacent 4-word cache lines are loaded into the cache on a miss.

Table 1: Benchmark matrices.

Name	Description	Rows	Avg. non-zeroes per row of L
GRID100	5-point approximation of Laplacian	10,000	1.98
BCSSTK23	Globally Triangular Building	3,134	6.71
BCSSTK15	Module of an Offshore Platform	3,948	14.42

codes, we also study performance using the Tango multiprocessor simulator [7]. The machine architecture we simulate is similar to but not identical to the architecture of the SGI 4D/340. Our simulated machine has the same processors as the 4D/340, and has a 64 KByte cache, but it has no second level cache. The cache lines are 16 words long. The cost of a cache miss is 50 cycles, identical to the cost of a second-level cache miss on the 4D/340. We present numbers for simulated parallel executions on up to 8 processors, more than the machine on which we perform our experiments contains. We do this to make the effects we are demonstrating more apparent and also to give some indication as to how the programs might behave on a machine with more processors.

Table 1 describes the benchmark matrices we use to evaluate the performance of parallel triangular system solving methods. The matrices are drawn from the Boeing/Harwell test set. For our study, we perform a zero-fill incomplete factorization of these matrices to arrive at L. Note that we have chosen matrices with widely varying degrees of sparsity. For space reasons, we have chosen only 3 benchmark matrices. We have studied a number of other matrices, however, and the results were quite similar to those for the three matrices we have chosen.

3 Parallel Solution of Sparse Triangular Systems

We now focus on the parallel solution of a sparse triangular system of equations. The solution of such a system is quite a simple computation. The following pseudo-code solves the system $Lx = b$, where L is lower triangular.

```
for i = 1 to $n$ do
    $x_i = (b_i - \sum_{j:j \neq i \wedge L_{ij} \neq 0} L_{ij} * x_j)/L_{ii}$
```

(Note that if the counter, i, were to go from n down to 1, instead of from 1 up to n, then this pseudo-code would solve an upper triangular system.) Since L is lower triangular, j in the summation is always less than i, and thus the sum involves only x_j 's that have already been computed. Since the incomplete factorization in the ICCG computation produces triangular matrices with unit diagonals, we assume from now on that $L_{ii} = 1$.

The sparse matrix L is typically stored using the following sparse storage scheme. A vector a stores the non-zeroes of L, by row. A vector ja stores the column number for each non-zero in a. A vector ia stores a pointer into a for each row, indicating where the non-zeroes for that row begin. Thus, the above pseudo-code, when applied to this specific storage scheme and our unit diagonal assumption, leads to the following code:

```
1.  for i = 1 to $n$ do
2.      $x(i) = b(i)$
3.      for j = $ia(i)$ to $ia(i + 1) - 1$ do
4.          $x(i) = x(i) - u(j) * x(ja(j))$
```

We now consider how this algorithm can be parallelized. One potential source of concurrency is the j loop, the loop over the non-zeroes of row i. However, the number of non-zeroes in one row of L is typically much too small to allow sufficient distribution of work. For example, matrices arising from the 5-point discretization of a rectangular region contain only 2 non-zeroes per row of L, much too few to provide significant concurrency. The other, more readily exploitable source of concurrency is the i loop, the loop over rows of the matrix. It is clearly not the case that all iterations of this loop can be performed in parallel; the result x(i) of iteration i is often needed by some later iteration. However, given a sparse L, the dependencies between iteration i and some later iteration can be determined in advance and can be exploited to permit concurrent computation of unrelated $x(i)$.

The dependencies between the x(i) can be derived in a straightforward manner from the non-zeroes of L. If L_{ij} is non-zero, then clearly x(i) depends on x(j). We can build a graph of dependencies, where each row is a node and a directed edge is present from node j to node i for each non-zero entry L_{ij}. The resulting graph is clearly acyclic, since the edges always go from a node to some higher numbered node. The resulting dependency graph can then be topologically sorted, such that each node is placed in some discrete level and all edges go from a node in a lower level to a node in a higher level. Note that the assignment of rows to levels is not unique. A row could be placed, for example, in the earliest possible level. Alternatively, it could be placed in the latest possible level. Anderson [2] studied a number of different methods for assigning rows to levels, and concluded that the the level assignment had little effect on the performance of the parallel method we now discuss. We place a row in the earliest possible level.

Given a division of the rows of L into a set of levels, the computation within a single level can be distributed among a number of processors, since the rows within a particular level do not depend on each other. This observation leads to *the level-scheduled method* [2] for solving sparse triangular system in parallel. Processors perform the computation corresponding to the rows of one level, then they do a barrier to synchronize, and then the processors move on to the next level. The dependencies are enforced by the barrier synchronizations: a level is not begun until all rows from previous levels have been completed, thus all of the dependencies associated with the level being processed will have been resolved.

Another method for the parallel solution of sparse triangular systems, *the self-scheduled method* [14], does not rely on the global synchronization of all processors to resolve data dependencies. Instead, this method explicitly tests each dependency to make sure that it has been resolved. Each time a processor wishes to use some x(j), it first checks an auxiliary data structure *ready(j)* to determine whether x(j) is ready to be used. If *ready*(j) has not been set, then the processor waits until *ready(j)* is set before proceeding. Once a processor computes an x(i), then it sets *ready(i)*. Note that this method introduces a substantial amount of overhead into the computation. Recall that the inner loop of the sequential computation involves the multiplication of some x(i) by a non-zero of L. The parallel self-scheduled method adds a test of *ready(i)* to this inner loop.

The two parallel codes both function as follows. A permuted index vector stores the row numbers, in order of increasing level. A free processor performs the work associated with the first unprocessed row in the permuted index set. In the level-scheduled method, a barrier is performed whenever a level boundary is reached in the permuted indices. Rows can be assigned to processors in two ways, dynamically or statically. In the dynamic method, a shared counter is used to distribute rows. When a processor is free to perform another task, it atomically increments the counter in order to obtain the index of a new row. In the static method, row indices are distributed before the computation begins. A static row assignment must be done so as to balance the load among the processors. This assignment is typically done by assigning the rows of the level-ordered list to the processors in a round-robin fashion. The dynamic scheme has the advantage of better load balancing, while the static scheme removes the contention for the shared counter.

If we consider the two triangular system solves that must be performed within each iteration of the ICCG method, we note that both involve L, but one uses L itself and the other uses the transpose of L. If L is stored by rows, then L^T can be read from the same data structure, but it would be stored by columns. While it is possible to develop a parallel triangular system solving code that works with matrices stored by columns, Hammond and Schreiber show in [10] that doing so results in substantial losses in performance on a multiprocessor (they used a Sequent Balance). The reason is that the above row-oriented code performs multiple concurrent reads of previously computed x entries. A column-oriented approach, on the other hand, performs multiple concurrent *writes* to x entries that have not yet been computed. The concurrent reads can be performed without mutual exclusion between the processors, but the writes cannot. Each write must therefore be protected with a lock, resulting in substantial overheads. For higher performance, the matrix L is therefore replicated and stored in two different formats, one by rows and the other by columns. In this way, a row-based triangular system solving code can be used to perform both of the triangular system solves of the ICCG method.

A similar point comes up when considering the sparse matrix-vector multiplication that arises in the conjugate gradient method. Since the A matrix involved in this multiplication is symmetric, it can be stored in a packed manner, where only the lower triangle of A is kept. However, Hammond and Schreiber show that parallel performance is substantially decreased if A is stored in this way. The issue is again one of overheads associated with mutually exclusive accesses. An efficient parallel program must store the full A matrix. Overall, an efficient parallel code requires roughly twice as much storage as a sequential code. The sequential code can store L and the lower triangle of A, while the parallel code must store L, L^T, and all of A.

The performance data presented in this paper, for both the parallel and the sequential methods, are all obtained from codes that flush the processor caches before beginning the system solves. The caches are flushed in order to more accurately reflect the state that they would be in when a triangular system solve $Lx = b$ is performed within the larger context of the ICCG computation. Between one invocation of this triangular system solve and the next, the ICCG method performs a matrix-vector multiply using the matrix A, it performs a number of operations involving dense vectors, and it also performs a triangular system solve using the matrix L^T. None of these other operations involve any of the data structures associated with the triangular system solve $Lx = b$. We assume that the data structures associated with A, L^T, and other accessed data structures are sufficiently large that by the time the system $Lx = b$ needs to be solved again, all data structures associated with this computation will have been flushed out of the cache. Actually, b will most likely be present in the cache, but we ignore this fact to simplify the analysis. Since b plays a small part in the computation, the impact of this simplification is minimal.

We have implemented versions of each of the methods described above for solving triangular systems. Figure 2 presents the speedups obtained on the SGI 4D/340 multiprocessor for the three benchmark matrices, relative to an efficient sequential implementation. The obtained speedups are quite small; the largest is roughly 1 .1 on 4 processors.

4 Why Are the Speedups So Small?

In this section, we investigate the reasons for the poor speedups obtained with existing parallel methods. We consider the operations involved in the parallel sparse triangular system solve computation, and we consider how these operations interact with the architecture of the machine performing the computation.

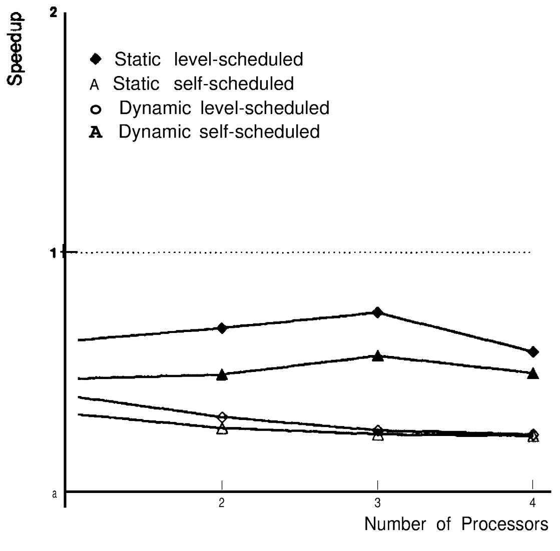
4.1 Static versus dynamic methods

The results in Figure 2 clearly show that the statically scheduled methods perform significantly better than the dynamically scheduled methods. This is due to the extremely fine grain of the computation, coupled with the fact that the machine has no hardware support for distributed loops. In order to obtain a row in both of the dynamic schemes, a processor must first acquire a lock, then increment a global counter, and finally release the lock. Acquiring the lock requires roughly 20 machine cycles, assuming the lock is free. Once a processor obtains the lock, it must fetch the value of the global counter and increment it. Because of the invalidation-based cache coherence protocol [1], the current value of the global counter is only available in the cache of the processor that last incremented it. Since the counter is actively modified by all of the processors, a processor wishing to obtain a new row typically has to fetch the contents of the counter from the cache of another processor, requiring roughly 50 cycles. Releasing the lock does not cost the releasing processor any time, since the processor can proceed after it has issued the release. However, the lock is not available to any other processor until roughly 20 cycles after it has been released.

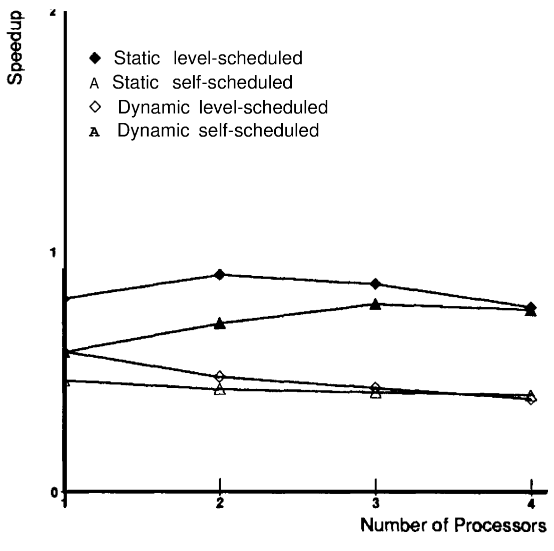
If we compare the costs of performing the above mutual exclusive accesses to the counter to the amount of work done once a row is obtained, we see why so much performance is lost. If we divide the total runtime of the sequential triangular system solving code for a particular matrix by the number of rows in that matrix, we find that a single row requires very little time to process. For matrices GRID100, BCSSTK23, and BCSSTK15, respectively, a row requires 100, 180, and 330 cycles of computation. Clearly, contention for the counter is a substantial problem. While it is possible that hardware support for distributed loops would improve the performance of the dynamic methods, we show in the next subsection that there are other reasons to nrefer static methods.

4.2 Spatial locality

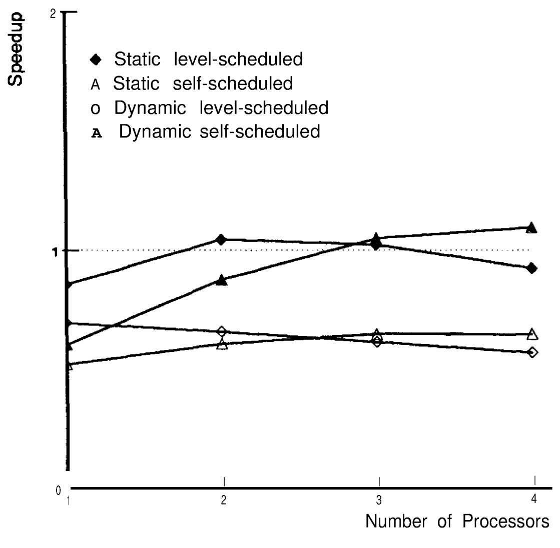
In this subsection we use simulation to investigate the reasons for the poor speedups obtained with existing parallel methods. We first consider the costs involved in fetching the L matrix as the number of processors is increased. These costs are measured by simulating each of the parallel approaches with our multiprocessor simulator, and keeping track of how many cache misses result from accesses to the matrix L. Since the cost of



GRID100



BCSSTK23



BCSSTK15

Figure 2: Speedups for matrices GRID100, BCSSTK23, and BCSSTK15.

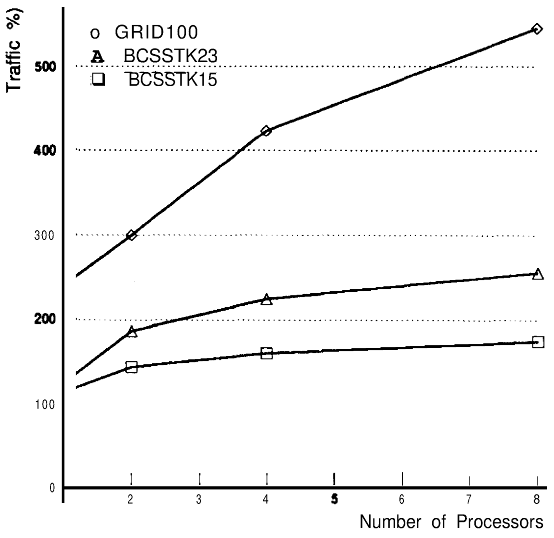


Figure 3: Data traffic due to matrix L, as a percentage of the size of L, for static level-scheduled method.

fetching L from main memory is the dominant cost in the sequential solution of triangular systems, accounting for between 60% and 85% of total memory traffic and between 50% and 75% of overall runtime for the three matrices, it will certainly play an important role in determining the performance of a parallel code as well.

In Figure 3 we show the amount of data fetched when reading L, as a percentage of the size of L, using between 1 and 8 processors. These traffic figures come from the static level-scheduled method, but the figures are virtually identical for the other methods. To put these figures in better perspective, we note that the sequential triangular system solving code generates between 102% and 103% of the size of L in traffic for the three matrices.

It is clear that the parallel code, even when executed on a single processor, generates more traffic than the sequential code. Furthermore, the amount of data traffic increases as the number of processors is increased. This traffic translates into more idle time spent waiting for cache misses to be serviced, and more traffic on the shared bus. These observations can be explained if we consider the spatial locality of the parallel computation. As discussed previously, one of the main purposes of long cache lines is to amortize the latency associated with fetching a data item from main memory. This practice is only effective if the program exhibits a high degree of spatial locality. Returning to our triangular system solving codes, consider the layout of the matrix L in memory. The non-zeroes for row i are immediately adjacent to those of row i + 1, and a single cache line can contain non-zeroes from a number of adjacent rows. Thus, when a fetch is done to a memory location for row i, the memory locations fetched in the same cache line belong to row i or to rows adjacent to i. Since the sequential triangular system solving code begins with row 1 and accesses each subsequent row in succession, this code immediately uses the extra locations brought in due to the large cache line size. Not surprisingly, the amount of data traffic due to L in the sequential code is roughly equal to the size of the matrix.

Now let us consider the pattern of accesses to L for the parallel methods. We first consider the case where the parallel method is run on a single processor. Recall that the rows are accessed in a permuted order in the parallel code; the rows of one level are accessed before those of the next level. The level numbering of rows will not in general be the same as the row numbering of the original matrix. Consider the impact of this different order on the spatial locality of the computation. An access to row i may bring in non-zeroes from rows adjacent to row i in the original matrix. However, these locations from adjacent rows may not be accessed until much later in the parallel code, and consequently they might be displaced from the cache before they are accessed. A very sparse matrix suffers more from this effect. Consider, for example, the matrix GRID100. Each row of this matrix contains only 2 non-zeroes. Since a cache line contains 8 non-zeroes, only one-fourth of the entries brought in on a cache miss will be accessed immediately. Matrices with more non-zeroes per row will use more data immediately, thus making better use of the cache line, but they will still suffer from this effect. This

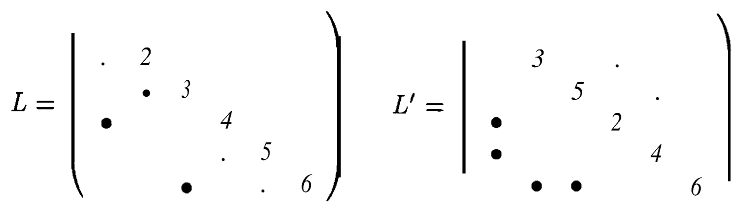


Figure 4: Non-zero structure of a matrix *L*, before and after reordering.

decrease in spatial locality explains the disparity between the amount of data traffic for the sequential approach and the amount for the parallel methods run on one processor.

Now let us consider what happens to spatial locality when more than one processor is used. A processor that accesses the non-zeroes of a row i may bring in non-zeroes from adjacent rows. However, in the parallel case, these other non-zeroes may belong to a row that is assigned to a different processor and is therefore never accessed by this processor. These non-zeroes represent wasted data traffic. Clearly, the probability that two adjacent rows are assigned to the same processor is lower when there are more processors. Thus, we observe an increase in the amount of data traffic due to L as the number of processors is increased. Note that an increase in traffic would also be observed for accesses to the x and b vectors, since these data structures suffer from the same effect.

4.3 Restoring spatial locality in the parallel methods

As mentioned previously, an increase in data traffic seriously degrades parallel performance. We must therefore consider methods for restoring the spatial locality that was present in the sequential code but has been lost in the parallel approaches. Our goal is to arrange the data so that it is accessed in a regular, contiguous manner, as was the case in the sequential code. One way to obtain such behavior is to rearrange the non-zeroes of L so that the rows processed by a particular processor are stored together, in the order in which they are accessed. The sparse matrix data structure we have been using does not allow such a reordering, since it assumes that the non-zeroes of row i + 1 are placed immediately after those of row i. We therefore augment the data structure with a new $ia - last$ structure. The non-zeroes of a row i can then be found in positions $ia(i)$ through $ia - lust(i)$ in a. We performed such a reordering as a preprocessing step for the two statically scheduled methods, and simulated the resulting approaches. The data traffic was substantially reduced, but it was still much higher than that of the sequential code. The reason is that the non-zeroes of L are only part of the problem. Spatial locality is still missing in accesses to other important data structures, including iu, $ia - dust$, x, and b.

In order to restore locality in these other structures as well, we propose that the rows and columns of the matrix be symmetrically reordered, so that the row numbers of the rows assigned to a particular processor are contiguous and appear in the order in which the rows are processed. The reordering can again be done as a preprocessing step for the two statically scheduled methods. For an example of such a reordering, consider the matrix L of Figure 4. Assume that the odd-numbered rows of this matrix are assigned to processor A and the even-numbered rows are assigned to processor B. Matrix L' shows the matrix after a reordering in which the rows assigned to processor A appear before those assigned to processor B. Note that the resulting matrix is no longer lower triangular. This change has no effect on the dependencies between the rows, though. We have simply assigned different numbers to the rows.

After such a reordering is performed, each processor works with a contiguous set of rows. Accesses to all data structures proceed as they do in the sequential case, with processors always accessing the entries related to row i + 1 immediately after accessing those of row i. The effect of this reordering on traffic due to L can be seen in the simulation results of Figure 5. Clearly, this modification has eliminated the superfluous data traffic related to matrix L that was introduced by the original parallel programs. The effect of this reduction in traffic on the performance of the SGI 4D/340 can be seen in Figure 6, where we present parallel speedups for the reordered matrices.

A similar modification cannot be made when the assignment of rows to processors is done dynamically. If

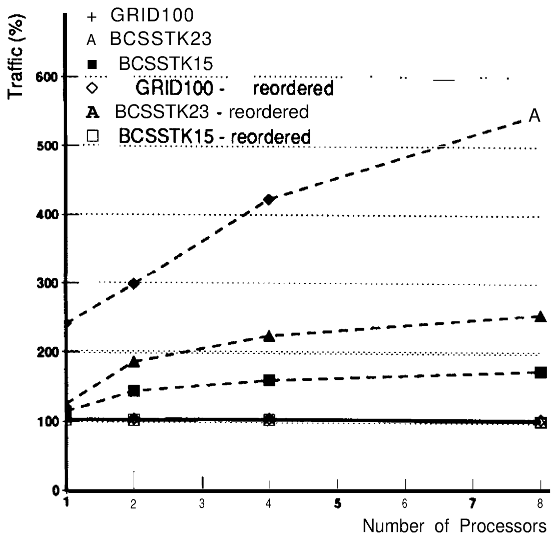


Figure 5: Data traffic due to matrix L, before and after reordering.

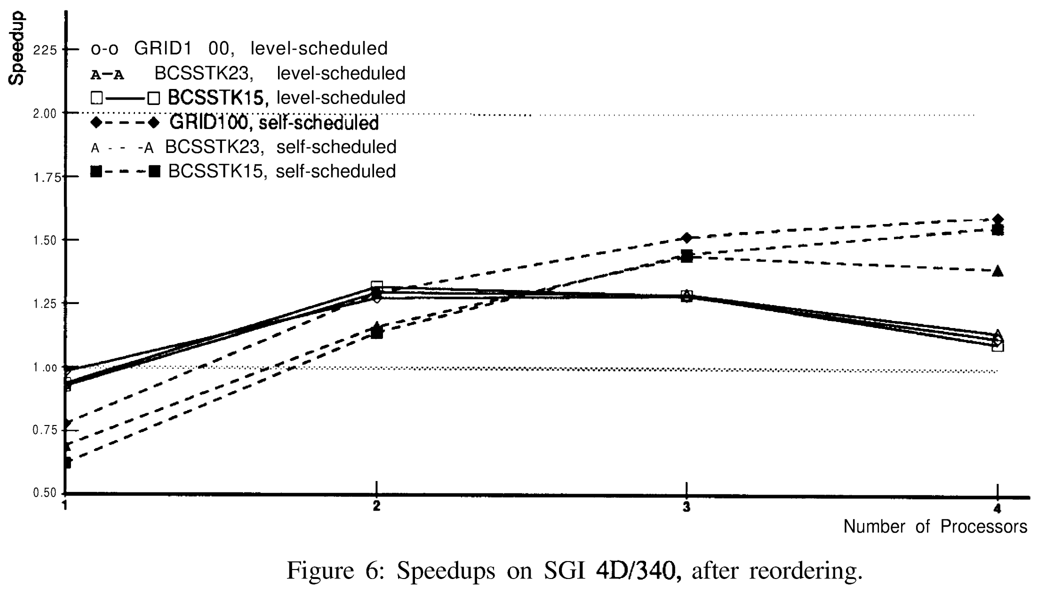
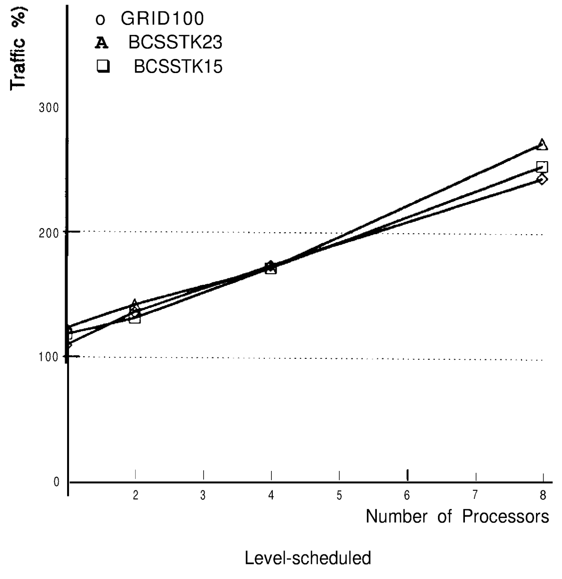


Figure 6: Speedups on SGI 4D/340, after reordering.



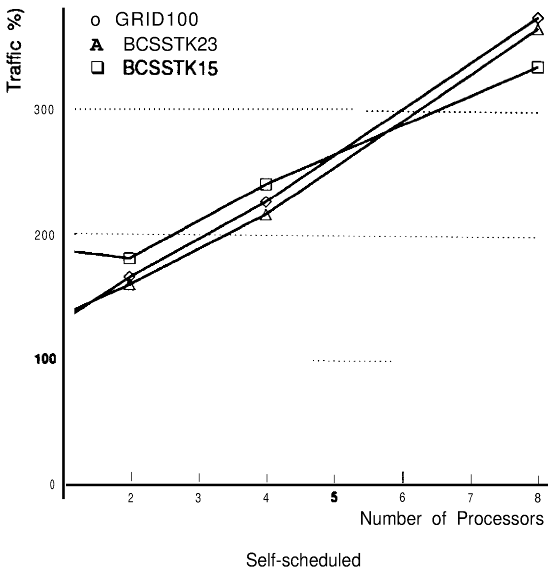


Figure 7: Total data traffic, as a percentage of total data set size.

it is not known a priori which rows are accessed by a particular processor, then it is not possible to arrange the rows so that spatial locality is preserved. We therefore conclude that a static assignment of rows to processors has significant performance advantages over a dynamic approach on a machine with large cache lines.

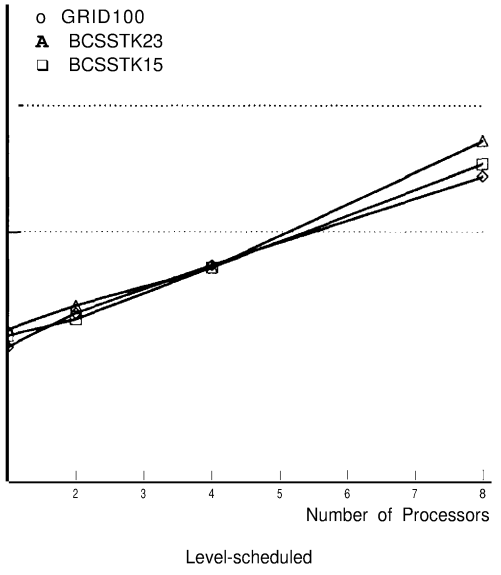
4.4 Implicit self-scheduling

In the last subsection, we studied the data traffic associated with the L matrix. Since traffic due to L has been effectively minimized by the reordering modification, we now consider other sources of data traffic. In Figure 7 we show the total amounts of data traffic generated by the static level-scheduled and self-scheduled schemes, expressed as a percentage of the total size of all data associated with the system solve[2]. By all data, we mean the matrix L and the vectors x and b. Since the processor caches are empty when the system solve begins, the percentage clearly can not be less than 100%. One would hope, however, that it would not be much above this level. The sequential code, for example, generates 106%, 108%, and 107% of the size of the data set in data traffic, for matrices GRID100, BCSSTK23, and BCSSTK15, respectively.

The first thing to note about these graphs is that the self-scheduled method generates substantially more traffic than the level-scheduled method. The primary reason for this extra *traffic* is the use of *the ready* data structure. Recall that this structure indicates when a particular data item has been computed and is ready to be used. This data structure brings about extra traffic for three reasons. The first reason is probably the most obvious; this structure increases the size of the data set. The processors must access this structure in addition to the structures holding the actual triangular system data. The lowest possible ratio of data traffic to data set size is no longer 100%. This factor, however, is quite minor. The *ready* data structure only increases the data set size by 2% to 10%.

The second factor has to do with interference in the processor cache. In the level-scheduled method, the calculation of a single entry of x involves accesses to a row of L and to some set of entries of x. In the self-scheduled method, accesses are made to some set of entries in *ready* as well. Problems result if the accessed locations map to the same place in the cache. If three separate structures are accessed, then the chances are much greater that two of them interfere.

[2]The anomaly in the traffic figure for the single processor case of the self-scheduled code run on matrix BCSSTK15 is due to interference between two data structures in the direct-mapped cache. While the effect of cache interference is typically small, it can sometimes become significant when the accessed locations of two different data structures end up mapping to the same location in the cache.



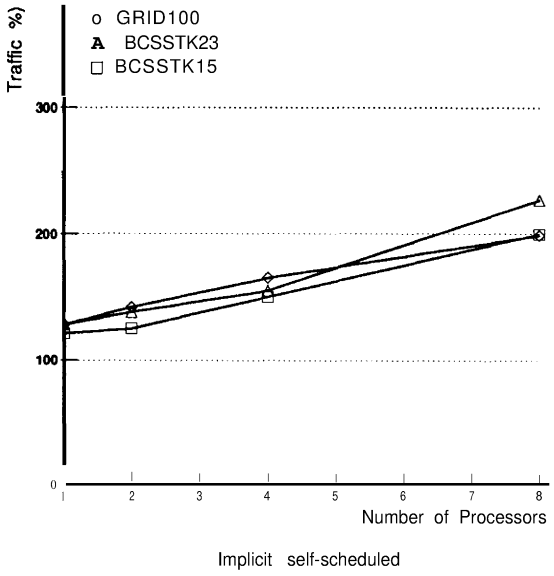


Figure 8: Total data traffic, as a percentage of total data set size, for implicit self-scheduled method.

The third factor has to do with the size of the cache lines. The smallest piece of information that can be passed between one processor and another is a single cache line. If one processor has completed an entry in x, it sets an entry in *ready* to one. In order for another processor that is waiting for that entry to learn of its availability, it must fetch the entire cache line containing *ready(i)*. The cache line will in general be much larger than the single entry of *ready(i)* that is needed. The processor must then fetch the cache line containing *x(i). Thus,* even though the *ready* structure is smaller than the vector x, the traffic due to each is comparable. We show in the next section that traffic due to x is a substantial fraction of overall traffic.

Fortunately, the data traffic impact of the *ready* data structure is avoidable. In fact, *the ready* structure can be eliminated entirely. Recall that this structure indicates when a particular entry in x is ready to be used: once x(i) has been computed, then *ready(i)* is set to one. Consider the computation of entry x(i) in the self-scheduled method. Once a processor begins computing the value, it does nothing else until the value has been computed. If the intermediate values of the entry are kept in a local variable and the final value is written into x(i) only once the computation is complete, then the location in memory holding x(i) goes immediately from an invalid state to a state where it holds its final value. *Thus, x(i) can serve the* purpose of *ready(i). The* IEEE 754 Standard for Binary Floating-Point Arithmetic [5] provides a value, called *NaN* (Not a Number), that can be used to indicate that x(i) is not yet a valid floating-point number. We can store *NaN* into each entry of x before the triangular system solve begins. Now, instead of setting *ready(i)* to 1 to indicate that x(i) is ready to be used, we can simply store a valid value into x(i). The simulation data traffic results for this method, which we call *the implicit self-scheduled method, are* presented in Figure 8. In Figure 9 we present the speedups for the implicit self-scheduled method on the SGI 4D/340, as compared to those of the level-scheduled method.

Before the implicit self-scheduling modification, the self-scheduled method was slightly slower than the level-scheduled method when run on two or fewer processors, and slightly faster on more than two. Now it is slower only when run on a single processor, and it is now substantially faster on more than two processors. The implicit self-scheduled method is slower than the level-scheduled method on a small number of processors due to the overheads associated with testing x elements to determine if they are ready to be used, as mentioned previously. It surpasses the performance of the level-scheduled method as the number of processors is increased because of two factors, both related to the global synchronization that must be done after each level in the level-scheduled method. The first factor is the communication overhead incurred when the global synchronization is performed in the level-scheduled method. Synchronization in the implicit self-scheduled method is done through computed x values. As can be seen from Figure 8, the traffic for the level-scheduled method is now somewhat

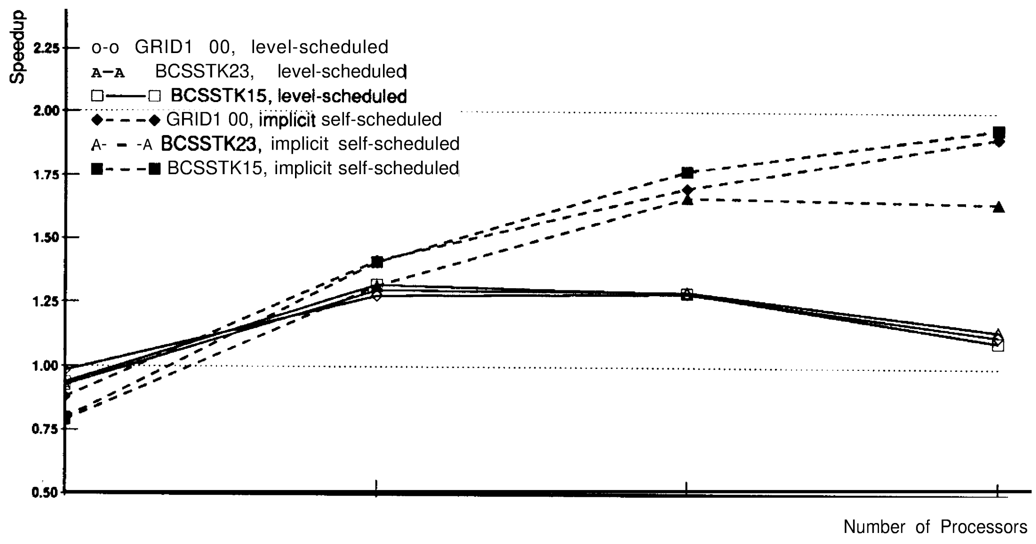


Figure 9: Speedups for level-scheduled and implicit self-scheduled methods.

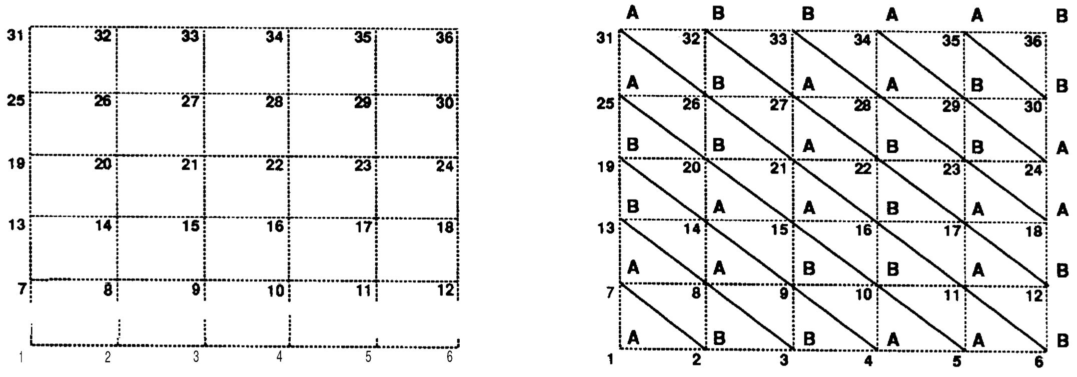


Figure 10: A small 5-point grid, and its two-processor mapping.

higher than that of the implicit self-scheduled method. The other factor is the load balancing problem that result from the global barrier. The self-scheduled method only waits when it needs a value that has not yet been computed. The level-scheduled method, on the other hand, must wait for all processors to finish a level before going on to the next one.

4.5 Traffic due to x

While data traffic has been substantially reduced by the modifications made so far, the traffic still increases as the number of processors is increased. The modifications we made in a previous subsection removed the traffic increase due to L, b, and auxiliary data structures. While the traffic increase can be explained partly as a product of the synchronization traffic in the case of the level-scheduled method, that does not explain the increase in traffic for the self-scheduled method. The increase is actually due to the communication of elements of x between processors.

Consider the 5-point grid problem of Figure 10. The grid on the left represents a natural ordering of the points in the grid, and the grid on the right represents the mapping of grid points to processors that would be done by a round-robin assignment on two processors. The letter next to each grid point indicates the processor to which that point is assigned. The diagonal lines show the different levels of the computation. In the L corresponding

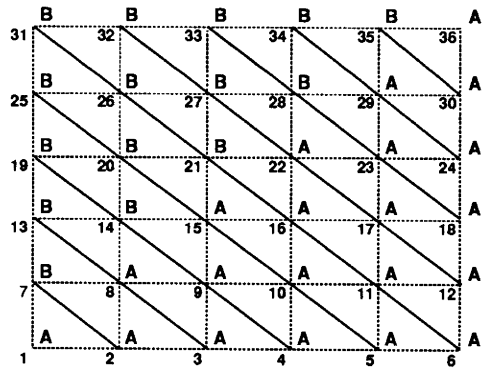
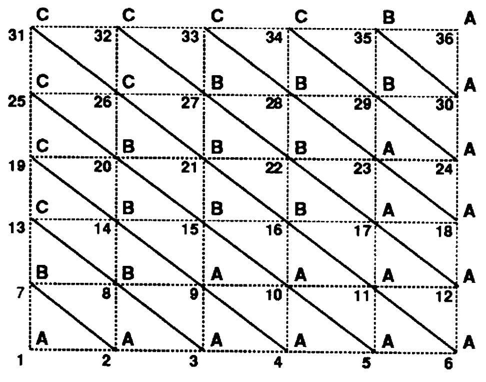



Figure 11: Processor mappings for two and three processor partitions.

to an incomplete factorization of the matrix of the grid, a point depends only on its lower numbered neighbors. Thus, for example, point 8 depends only on points 2 and 7.

Now consider the parallel solution of a system involving this L, using the processor mapping of Figure 10. Processor A begins by computing the value of x at point 1, placing the computed value in its cache. Now processor A computes x at point 7, and processor B computes x at point 2. Note that x(2) depends on $x(1)$. Thus, processor B must fetch the value of $x(1)$ from the cache of processor A. Looking ahead, when processor A computes the value of x(8) in the next level, it must fetch the value of x(2) from the cache of processor B. Overall, every entry of x computed by processor A, with the exception of a few boundary points, must be communicated to processor B, and vice versa.

Looking at the case of a general sparse matrix, consider a single row of L and the corresponding entry in x. This x entry will be needed by a number of future rows, thus it will need to be communicated to the processors that process these future rows. If rows are assigned to processors in a round-robin manner, then it is quite likely that this x entry will need to be communicated to a number of other processors. Indeed, if the number of rows that use the entry is large compared to the number of processors, then we would expect that this entry would need to be communicated to all other processors. Consider matrix BCSSTK15, for example, where each entry of x is used on average by more than 14 other rows. When the system arising from this matrix is solved on 2 processors, 99.6% of all entries of x must be communicated between one processor and the other. When solved on 4 processors, each entry of x is communicated on average to 2.90 other processors, which is again nearly equal the worst case. We would expect that when the number of uses of an element of x is large in comparison to the number of processors, then for a round-robin assignment the number of elements communicated between processors would be roughly equal $(P - 1) * n$, where P is the number of processors and n is the number of rows in the matrix. If the number of uses of an entry is smaller than the number of processors, then the number of elements that must be communicated is limited by the number of uses. For example, the communication necessary in solving the system arising from matrix GRID100 would be at most $2 * n$, regardless of the number of processors used.

Let us now quantify the impact of the communication of x entries on overall traffic. Our simulations show that the x vector is responsible for 19%, 19%, and 12% of all system solve traffic for matrices GRID100, BCSSTK23, and BCSSTK15, respectively, when run on one processor. When run on four processors, it is responsible for 34% 30% and 26%. On eight processors, x is responsible for 37%, 43%, and 36%. Clearly, the round-robin assignment of rows to processors results in a rapidly increasing communication load, as each x entry must be communicated to more and more processors.

We now consider the potential benefits that might come from a better assignment of rows to processors. Returning to our simple grid example, consider the mappings of nodes to processors of Figure 11. In these mappings, we partition the graph into a number of regions, such that the majority of edges in the graph go between two nodes in the same region while still keeping the load well balanced among the processors. By performing such a partitioning, we substantially reduce the number of entries of x that must be communicated between processors. For the case of 2 processors and an arbitrary size grid, only the $2\sqrt{n}$ entries along the boundary between the two regions would need to be communicated between one processor and the other. Communication

Table 2: Ratio of x traffic, with and without partitioning heuristic.

Name	P=2		P=4		P=8	
	Element ratio	Traffic ratio	Element ratio	Traffic ratio	Element ratio	Traffic ratio
GRID100	50.0	5.3	23.5	4.5	11.5	3.5
BCSSTK23	2.5	1.3	2.4	1.4	2.5	1.9
BCSSTK15	2.0	1.5	2.5	1.9	3.1	2.4

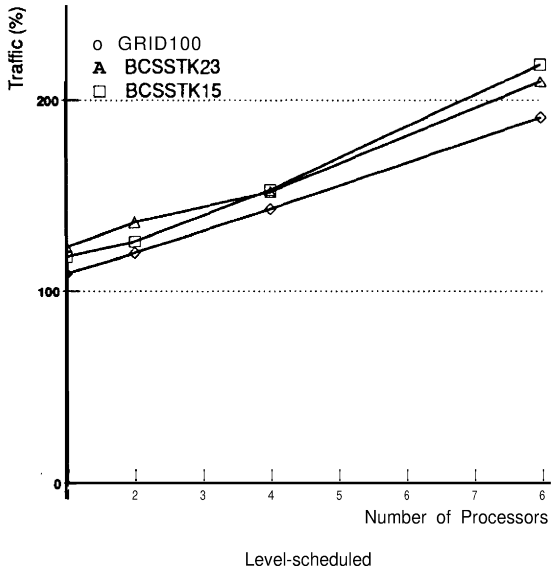
would thus be reduced from n for a round-robin mapping to roughly $2\sqrt{n}$ for a subregion mapping. For an arbitrary number of processors, using a similar partitioning, there would be $P-1$ boundaries between subregions, each of which would have roughly $2\sqrt{n}$ entries along it. Thus we would expect communication to decrease from $2n$ to roughly $2(P-1)\sqrt{n}$.

When considering general sparse matrices, the partitioning problem becomes much more difficult. Without the very simple structure of a grid, and without any knowledge of the structure of the problem that gave rise to the matrix, we must instead attempt to solve a general optimization problem. We must attempt to minimize the number of edges crossing between nodes in different partitions, while at the same time balancing the load among the partitions at each level of the graph. Actually, the number of edges crossing between partitions is not the exact measure we are interested in. If we assume that an entry only needs to be communicated once between any pair of processors, then any set of edges that goes from a number of nodes in one partition to a single node in another partition should be counted as a single edge. This optimization problem appears to be quite difficult. A simpler version of this problem, where the load balancing constraint is removed, can be shown to be NP-complete [8].

We have experimented with simple heuristics for finding good partitionings. Our heuristics build the partitioning one level at a time, attempting to minimize the amount of communication necessary to complete the level, assuming that the assignments for all previous levels are fixed. The heuristics attempts to minimize communication within a single level by placing a node on the same processor as the nodes from previous levels that it uses. Since all edges originating from nodes at one level terminate at nodes of previous levels and these previous nodes have already been assigned to processors, it is possible to determine which processors contain the x values used by a particular node in the current level. The success of these heuristics clearly depends on how many different processors own these nodes. If the majority of these nodes are owned by a single processor, then placing the current node on this processor will generate little communication. Otherwise, a large amount of communication will be necessary.

Our first heuristic simply considers the nodes in the current level one by one, and assigns each one to the processor that owns the most previous nodes used by the node. In order to balance the load within a level, we require that each processor be assigned roughly the same number of nodes at each level. Each processor therefore has an upper limit on the number of nodes that can be assigned to it, and that processor is removed from consideration once this limit has been reached. Our heuristic assigns nodes that only use nodes assigned to a single processor first, in order to make sure that these nodes are not kept off of this processor because of the load balancing limit.

This simple heuristic gives the same partition on grid problems as the one we discussed earlier in this section. We also tried this heuristic on matrices BCSSTK23 and BCSSTK15. Table 2 presents the results of this heuristic partitioning. The *element ratio* column shows the improvement in the number of times elements of x must be communicated between processors. *The traffic ratio* column shows the effect of this improvement on actual data traffic, taking caching effects into account. We see, for example, that for matrix GRID100 on 2 processors the heuristic decreases the number of times elements of x must be communicated by a factor of 50, yet it only yields a factor of 5 reduction in communication traffic due to x. This discrepancy can be understood by recalling that our machine has 16-word cache lines. Thus, 8 entries of x fit in a single cache line. With the heuristic partitioning, only one element of x per level of the grid need be communicated. However, it is not possible to communicate a single entry; the smallest unit of communication is a single cache line. Thus, we must move 8 times as much data between processors as is necessary. Without the heuristic partitioning, every element of x must be communicated between the two processors, so little wasted data movement occurs. If all eight entries in one cache line come from the level most recently computed, then they are all needed by the



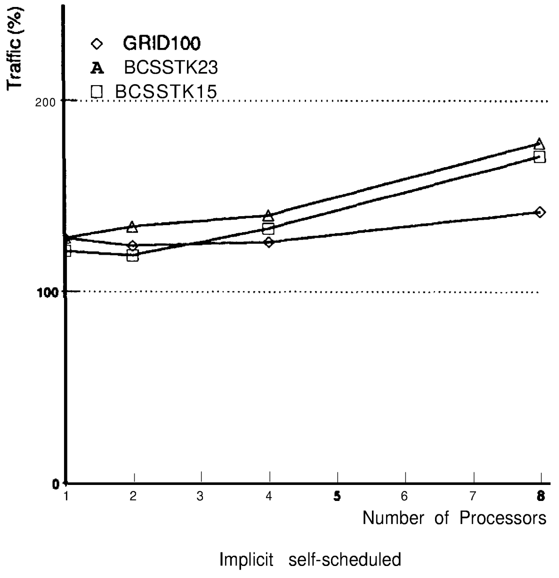


Figure 12: Total data traffic, as a percentage of total data set size, after partitioning.

other processor in the next level.

We now present, in Figure 12, simulation results for total data traffic generated after our heuristic partitioning is performed, using both the level-scheduled and the implicit self-scheduled methods. We also present in Figure 13 the speedups obtained on the SGI 4D/340 after the partitioning. Not surprisingly, the performance improvement is greatest for matrix GRID100, where the reduction in communication was the largest.

One limitation of the above heuristic is that it always assumes that an x entry assigned to one processor is not available on any other processor. This assumption is not always true; once an x entry has been communicated from one processor to another, then that entry will most likely be available in the caches of both processors from that point on. Our initial heuristic can be modified to take this into account. We can simply assign a node to the processor that has the most entries used by the node in its cache, regardless of whether the entry is cached because the caching processor owns the entry or because that entry was communicated to the processor earlier in the computation. The resulting heuristic further decreases the communication of x entries for BCSSTK23 and BCSSTK15 by roughly 15%, and decreases data traffic for these two matrices by slightly less than 15%. The new heuristic gives the same partitionings as the previous one for the grid problem. One disadvantage of this heuristic as compared to the previous one is that it requires substantially more runtime to find a partitioning. The SGI 4D/340 runtimes for the triangular system solve using the two heuristics are virtually identical, indicating that the communication of x entries is a sufficiently small component of runtime after the reductions from the first heuristic that further reductions have little effect on overall performance on the SGI 4D/340.

5 Discussion

Although the speedups we have obtained after our modifications are substantially larger than those of the original parallel methods, they are still relatively small. The best speedup obtained is roughly 2.2 with the implicit self-scheduled method on 4 processors. We believe that these speedups are limited by four factors. First, the code on which the parallel self-scheduled method is based is less efficient than a purely sequential code. The self-scheduled method adds a test to the innermost loop of the sequential code. It is clear from the speedup figures for the self-scheduled method on one processor that this factor accounts for a 20% loss of efficiency. Another factor is the saturation of the shared bus. As more processors are added, the bus becomes more heavily used, causing processors servicing cache misses to spend more time waiting for access to the bus. The result

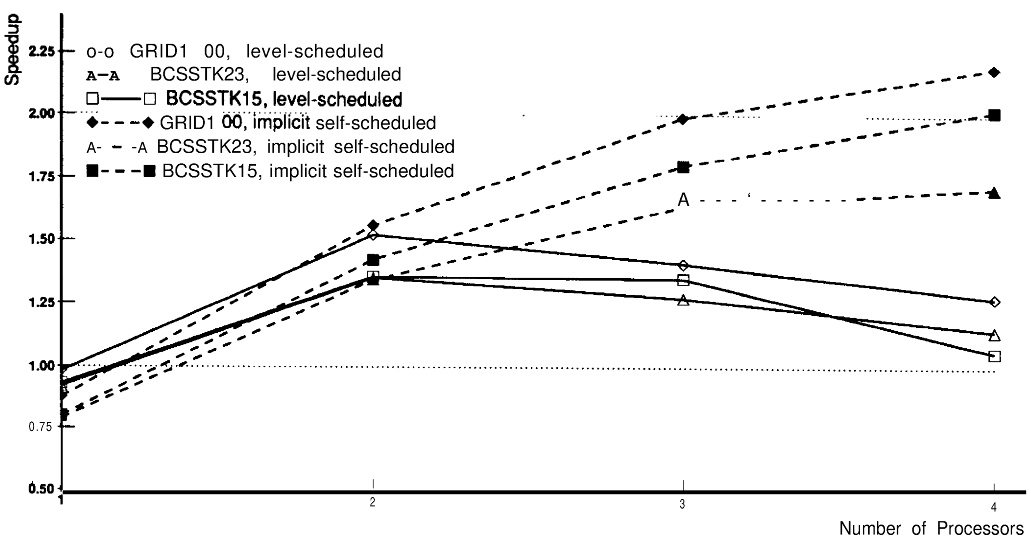


Figure 13: Speedups, after partitioning.

is an increase in the cost of a cache miss, and thus an increase in the amount of time processors spend sitting idle. Another related factor is the increase in traffic that results from the communication of entries of x when more processors are used. While this increase contributes to the saturation of the bus, it is also a significant contributor to the number of cache misses that must be serviced, and consequently to the amount of processor idle time. The final factor which may limit speedup is the amount of concurrency available in the problem. Processors may spend time waiting for x entries to become available. We do not currently know how large a role each of the last three factors plays in the less than perfect speedups we observed.

We now briefly consider the reasons for the large speedups reported in earlier papers. These speedups were many times those we observed on the SGI 4D/340 using the identical methods. Previous work on solving sparse triangular systems in parallel [3, 10,141 was performed on three different parallel machines, the Encore Multimax, the Sequent Balance, and the Alliant FX/8. We consider the Encore and Sequent machines first. Neither of these machines is particularly powerful in floating-point computation. The processors in these machines are rated at less than 100 LINPACK KFLOPS each. The MIPS R3000/R3010 processors of the SGI 4D/340, on the other hand, are rated at 4.9 MFLOPS each, or roughly 50 times the speed of the slower machines. However, the memory systems of these three multiprocessors all have roughly the same memory bandwidths and produce fetched data in comparable amounts of time. Thus, on the Encore and Sequent machines, memory system costs are minimal compared to the costs of performing floating-point operations. Large speedups can be obtained by simply distributing the floating-point work. The memory system traffic resulting from the distribution has little effect on runtimes. This situation, where memory system costs are dwarfed by the cost of performing floating-point arithmetic, no longer holds in current machines, and we see no reason to believe that it will hold in future machines either.

The processors of the Alliant FX/8 are relatively powerful in floating-point computation. They are slightly slower than the R3000/R3010 for scalar computations and slightly faster for vectorizable work. The Alliant achieves large speedups primarily because of its shared cache design; all eight processors share a single cache. One important implication of this design is that a data item fetched into cache by one processor is available immediately to all other processors. Thus, the loss of spatial locality due to processors fetching data that they do not use that was observed on our machine is not an important factor on the Alliant. Communication of the entries of the x vector is not a problem on the Alliant either. When an entry is computed, it is written into the shared cache, and thus is available to the other processors without any communication costs. Unfortunately, the shared cache design of the Alliant does not scale well as processor speeds are increased. It will be quite difficult to design a shared first-level cache that will support the bandwidth and latency requirements of 8 future high-performance processors. Many recent microprocessors, for example, run sufficiently fast that they cannot afford the latencies associated with going off-chip to their caches, so their caches are placed on-chip. While it will be possible for future multiprocessors to share an external second-level cache among a number of processors, we believe that the such a design would benefit from the new techniques discussed in this paper.

We now briefly consider a number of architectural factors that limit the performance of parallel sparse triangular solvers, and how they might be overcome. One obvious limitation of the SGI 4D/340 in performing this computation is bus bandwidth. While bandwidth appears adequate for 4 processors, our simulations results indicate that the bus would certainly not support any more than 4 processors, nor would it support faster processors. We believe that the bandwidth necessary to support more or faster processors could be provided without an increase in memory latency or an inordinate increase in cost through the use of a faster bus clock speed and a split-transaction protocol. If sufficient memory bandwidth were available, then the rate of the system solve would be limited only by the cost of communicating x entries between processors and by the concurrency available in the problem. We hope to further investigate how much performance can be obtained when more bandwidth is available.

Performance is also limited by the latency of the cache misses to the system solve data structures. Clearly this is a limitation for sequential as well as parallel approaches. While it is not reasonable to expect all data to fit in the fast levels of the memory hierarchy, it is reasonable to expect that future machines will be better able to hide this latency, using techniques such as data streaming and program-directed prefetch to bring soon-to-be-used data into the processor cache while the processor is working on something else [13]. Such capabilities will certainly present interesting new considerations for parallel approaches.

Another factor that limits the performance, specific to the implicit self-scheduled method, is the large overhead of the frequent tests for valid floating-point values. We believe that, given a small amount of additional hardware support, this method could be much faster. Rather than testing each x entry to determine whether it is a valid floating-point value, we could instead assume that it is valid and use it in the computation. If our assumption is incorrect and the entry is *NaN,* then the result of any computation involving this entry will also be *NaN.* After every nth iteration (for some n) in the computation of $x(i)$, the running value of $x(i)$ can be tested to determine whether a *NaN* has been encountered. If not, then the running value can be checkpointed and the computation can proceed. If a *NaN* was encountered, then the computation can start over from the last checkpointed value. Such an approach has the potential to significantly reduce the overhead of the implicit self-scheduled method. While this approach also has the potential to waste computation, we believe that a balance could be struck between the efficiency of the sequential code and the possibly wasted computation. This method is unfortunately impractical on the SGI 4D/340 because all operations involving *NaN* values are handled in software, and are thus many times slower than normal floating-point operations.

Another interesting issue concerns the cache coherency protocol. The SGI machine uses an invalidation-based protocol [1], where a cache line is invalidated in the cache if a write by another processor is observed to any location on that cache line. Thus, a write to an entry of x by one processor invalidates the line containing that entry in the caches of all other processors. The other processors must then fetch the contents of that cache line from the cache of the processor that performed the write. If an update protocol were used instead, where the cache line is updated to reflect the effect of the write, then the communication of x entries would become substantially less expensive. The benefit must, of course, be traded against the added cost of implementing an update protocol instead of the simpler invalidation protocol.

Finally, we note that while this paper has focused exclusively on the incomplete Cholesky conjugate gradient algorithm, the results would apply to a number of other algorithms as well. Both the Gauss-Seidel and SOR algorithms, for example, exhibit dependencies that are almost identical to those of a triangular system solve. In each, the nodes of the graph representation of the matrix depend only on lower numbered nodes and the nodes that can be computed in parallel can be partitioned into a series of levels based on these dependencies. We believe that these other algorithms would exhibit problems similar to those observed for parallel triangular system solving when executed on a hierarchical memory multiprocessor, and that the techniques described here should be of help.

6 Conclusions

In this paper we have considered the most demanding aspect of performing the incomplete Cholesky conjugate method on a multiprocessor, the triangular system solve. We studied two previously described methods for solving this problem, the level-scheduled and self-scheduled methods. We found that both provided little speedup on a multiprocessor with a deep memory hierarchy, a characteristic we believe will become more common as processors continue to get faster.

Upon further investigation of the behavior of these codes, we found that one important reason for their poor performance was that they caused a substantial decrease in the spatial locality of the computation as the number of processors was increased. This decrease in locality resulted in a significantly larger number of cache misses, and the resulting increased traffic on the shared bus caused large amounts of processor idle time spent waiting for misses to be serviced. We proposed a matrix reordering technique that restored the spatial locality. Since this technique is only applicable when the rows of the matrix are assigned statically to the processors, we concluded that a static allocation is preferable on machines with large cache lines.

Another source of lost performance, specific to the self-scheduled method, was the use of an additional data structure, a *ready* array, to indicate when a particular data element is ready to be used. This array was responsible for a significant amount of data traffic, traffic that was not present in the sequential code or in the level-scheduled code. We eliminated this structure by using the data itself to indicate when an element is ready to be used, rather than using an auxiliary structure. We also studied the amount of traffic generated by accesses to the x vector. We proposed a simple heuristic that greatly reduced this traffic for the benchmark matrices by keeping rows that accessed the same sets of x elements together on the same processor.

After all of the above modifications were made, the resulting codes for solving sparse triangular systems in parallel achieved substantially higher performance. Speedups were between 0.6 and 1 .1 on four processors before the modifications. After the modifications, they increased to between 1.7 and 2.2. The speedups were not perfect, due to a number of factors including limited bus bandwidth, synchronization overheads, interprocessor communication costs, and limited problem concurrency. However, the speedups were sufficiently large to justify the use of a parallel machine on this problem. This work has demonstrated that the performance of a parallel triangular system solving code is highly dependent on the amount of data traffic that the parallel approach generates. The parallel code must be written so that the locality of the computation is preserved.

This work has also shown that the performance of a parallel ICCG code would be severely limited by existing parallel triangular system solve approaches, which give no speedup on parallel machines with deep memory hierarchies. Since the triangular system solve steps comprise roughly half of the ICCG computation, overall speedups would be limited to a factor of at most two. Our work makes larger overall speedups possible when using a modest number of processors. It remains to be seen how these speedups will scale when the number of processors is increased.

Acknowledgments

This research is supported by DARPA contract N00014-87-0828. Anoop Gupta is also supported by an NSF Presidential Young Investigator award.

References

[1] Archibald, J., and Baer, J.-L., "An economical solution to the cache coherence problem", *Proc. of the 15th Annual Int. Sym. on Computer Architecture, 355-362, 1985.*

[2] Anderson, E., *Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations,* CSRD Technical Report 805, Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois, August, 1988.

[3] Anderson, E., and Saad, Y., "Solving sparse triangular linear systems on parallel computers", *International Journal of High Speed Computing,* 1: *73-95, 1989.*

[4] Baskett, F., Jermoluk, T., and Solomon, D., "The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second", *COMPCON 88, 468-47* 1, 1988.

[5] Cody, W.J., et al, "A proposed radix- and word-length-independent standard for floating-point arithmetic", *IEEE Computer,* 16, 1984.

[6] Concus, P., Golub, G.H., and O'Leary, D.P., "A generalized conjugate gradient method for the numerical solution of elliptic partial differential equations", In G.H. Golub, editor, *Studies in Numerical Analysis.* pages 179-198. The Mathematical Association of America, 1984.

[7] Davis, H., Goldschmidt, S., and Hermessy, J., *Tango: A multiprocessor simulation and tracing system,* Technical Report CSL-TR-90-439, Stanford University, 1990.

[8] Garey, M.R., and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness,* W.H. Freeman, 1979.

[9] Greenbaum, A., *Solving sparse triangular linear systems using FORTRAN with parallel extensions on the NYU Ultracomputer prototype,* Technical Report 99, Courant Institute, New York University, 1986.

[10] Hammond, S.W., and Schreiber, R., *Efficient ZCCG on a shared memory multiprocessor,* RIACS Technical Report 89.24, Research Institute for Advanced Computer Science, 1989.

[1 1] Hestenes, M.R., and Stiefel, E., "Methods of conjugate gradients for solving linear systems", *J. Res. National Bureau of Standards,* (49):409-436, *1952.*

[12] Meijerink, J.A., and Van der Vorst, H.A., "An iterative solution method for linear equation systems of which the coefficient matrix is a symmetric M-Matrix", *Math. Comp.,* 31:148-162, 1977.

[13] Mowry, T., and Gupta, A., "Tolerating latency through software-controlled prefetching in scalable shared-memory multiprocessors", Technical Report, Stanford University, to appear.

[14] Saltz, J.H., Mirchandaney, R., and Baxter, D., *Run-time parallelization and scheduling of loops,* Technical Report ICASE 88-70, ICASE, NASA Langley Research Center, 1988.