Software Interleaving *

Ricardo Bianchini Mark E. Crovella Leonidas Kontothanassis Thomas J. LeBlanc

{ricardo,crovella,kthanasi,leblanc}@cs.rochester.edu

Department of Computer Science
University of Rochester
Rochester, New York 14627
(716) 275-5426

Abstract

In this paper, we investigate the costs and benefits of implementing memory interleaving in software. As our main contribution, we compare software memory interleaving to row-major allocation and logarithmic broadcasting. Our analysis demonstrates the clear superiority of software interleaving over row-major allocation in the presence of memory contention. Our analysis also indicates that the choice between software interleaving and logarithmic broadcasting is less clear, as it depends both on the type of synchronization used and the number of processors. We conclude that, on large-scale multiprocessors, software memory interleaving and lock-based synchronization is the most effective combination for reducing memory contention in matrix computations.

1 Introduction

In order to effectively exploit large-scale shared-memory multiprocessors, we must eliminate *all* bottlenecks that limit scalability. One such bottleneck is limited memory or interconnection network bandwidth. Even well-designed machines can exhaust the available bandwidth when a program issues an excessive number of remote memory accesses or when remote accesses are distributed non-uniformly. While techniques for improving locality of reference are often successful at reducing the number of remote references, a non-uniform distribution of references may still result.

A non-uniform distribution of remote accesses can be found in several classes of applications. For example, in many parallel graph algorithms, such as transitive closure and Warshall-Floyd's all-pairs shortest paths, a single processor writes a row of the adjacency matrix that is then read by every other processor. Many linear algebra algorithms, including a straightforward parallelization of Gaussian elimination and LU decomposition, exhibit this same structure. In each case, a non-uniform distribution of accesses results, wherein most processors require simultaneous access to a single memory module.

A non-uniform distribution of accesses can cause contention both in the interconnection network and at remote memories. Although network contention may decrease performance under certain circumstances, the expected increases in interconnection network bandwidth (particularly with the use of optical networks) lead us to believe that contention for memory modules should almost always dominate. In addition, any technique successful at alleviating memory contention is likely as a result to alleviate interconnection network contention. Thus, memory contention alleviation is the key to significant performance improvement.

Several techniques have been developed for reducing memory contention. Linear algebra algorithms can exploit the properties of numerical equations to improve locality of reference, and as a side-effect eliminate most non-uniform memory addressing [4]. How-

*This research was supported under NSF CISE Institutional Infrastructure Program Grant No. CDA-8822724, and ONR Contract No. N00014-92-J-1801 (in conjunction with the DARPA HPCC program, ARPA Order No. 8930). Ricardo Bianchini is supported by Brazilian CAPES and NUTES/UFRJ fellowships. Mark Crovella is partially supported by a DARPA Research Assistantship in Parallel Processing administered by the Institute for Advanced Computer Studies, University of Maryland.

0-8186-6427-4/94 $04.00 © 1994 IEEE

ever, this approach may introduce significant complexity in the algorithm, and does not generalize to other classes of applications, such as the graph algorithms used in our study. Many other techniques for alleviating the effects of a non-uniform distribution of memory accesses assume special hardware support, such as multi-stage interconnection networks with combining of memory references [5, 7], interleaved memory [7], eager sharing [8], and eager combining [1]. Although these techniques are known to reduce or eliminate memory contention, the associated hardware can be both complex and expensive, and may depend on particular properties of the interconnection network. Thus, general software solutions are an attractive alternative.

Two software techniques for alleviating contention are software combining trees [9] and data replication. Software combining trees are analogous to hardware combining networks, and incorporate logarithmic broadcasting. We can also limit memory contention by replicating data across multiple memory modules. By distributing the requests for data evenly among the copies, we can reduce or eliminate memory contention for the original copy.

Each of the techniques described above is general enough to use in any program. However, our investigation of memory contention in programs for solving linear algebra and graph problems suggests that techniques devoted specifically to parallel matrix computations [6] can also be very effective at alleviating contention. In this paper, we evaluate the effectiveness of memory interleaving implemented in software. This technique is motivated by the observation that memory contention in matrix computations is typically caused by simultaneous access to a single row of the matrix by multiple processors. If matrices are allocated among memories by rows, simultaneous access to any part of a row requires that processors contend for a single memory module. Memory interleaving spreads memory accesses across several memory modules when multiple processors access a single row of the matrix.

We seek to characterize the source and extent of memory contention in SPMD matrix computations, quantify the costs and benefits of software memory interleaving, and evaluate the tradeoffs between software interleaving and logarithmic broadcasting on large shared-memory multiprocessors.

2 Methodology

We simulate a large-scale multiprocessor (up to 200 processors) based on a multi-stage interconnection network executing our example applications. Our simulations consist of two distinct steps: a trace collection process, and a trace analysis process. The trace-collection step uses Tango [2] to simulate a multiprocessor with (infinite) write-back caches. The traces generated by Tango contain the data references that missed in the local cache of each processor, and all synchronization events.

Our analyzer process takes as input an address trace produced by Tango, and simulates execution of the references in the trace on a distributed shared memory multiprocessor. The analyzer respects the synchronization behavior of the application as represented by the synchronization events contained in the trace. We simulate hardware barriers by allowing all synchronizing processors to leave a barrier at the same time. Lock and unlock operations introduce a short execution delay, 5 cycles. Synchronization events are not allowed to cause contention in our model, although they are critical in maintaining the relative timing of events during trace analysis.

In our machine model, a memory module can process only one request at a time. Requests arriving when the module is busy are rejected and must be reissued. Our analyzer measures contention for memory at the page level; thus each 4KB page is treated as a separate memory module to which requests may be directed. We treat each page as a separate memory module so as to simulate an ideal page placement policy in which contention caused by simultaneous accesses to multiple pages does not occur.

Our simulations assume a cache line size of 64 bytes, a fixed network latency of 36 processor cycles, and local memory latency of 10 processor cycles per cache line. In the absence of contention, a remote memory request requires a request message, a reply message, and memory service time, or 82 cycles total. Each request rejected due to contention suffers a 72 cycle penalty, corresponding to an immediate re-issue of the request. Our assumption that network latency is fixed (i.e., there is no network contention) allows us to isolate the effects of memory contention from network contention. This assumption corresponds to a machine architecture where the network bandwidth per node is much larger than the memory bandwidth. In addition, including network contention in our simulations would assign some of the contention we observe to the network rather than the memory, but would not be likely to affect the tradeoffs we consider here.

Our simulation parameters are somewhat optimistic. Throughout the paper, we present results demonstrating the performance effect of changing each of our parameters.

Our application workload consists of four parallel programs: two linear algebra applications (Gaussian elimination and matrix inversion), two graph algorithms (transitive closure and all-pairs shortest paths).

Our linear algebra applications are similar in that, during each phase of the computation, processors need access to a pivot row of the matrix describing the linear equations. This pivot row is written by one processor and then read by every other processor. In our graph applications, processors also require access to a pivot row of the adjacency matrix (where each vertex is connected to each other vertex with probability 0.5) representing the graph. The pivot row is written by one processor and then read by many other (possibly all) processors. In all the applications, the elements of a matrix row are allocated to consecutive addresses in a single memory module, so all processors direct a request to the same memory module after synchronizing.

The input to all our applications is a 512 × 512 matrix, except for all-pairs which takes a 400 × 400 matrix as input. Synchronization is implemented with locks in Gaussian elimination and transitive closure, while barrier synchronization is used for matrix inversion and all-pairs shortest paths.

3 Effects and Source of Contention

3.1 The Effects of Memory Contention

Table 1 shows how memory contention affects the running time of our applications. For Gaussian elimination and all-pairs shortest paths, memory contention causes the running time to increase with an increase in processors. In fact, moving from 50 to 200 processors increases the running time of these applications by a factor of 2-3, rather than cutting the running time by a factor of 4. The situation is not quite as bleak in the case of matrix inversion, where 100 processors perform slightly better than 50 processors; however, 200 processors perform no better than 50 processors. Transitive closure is the only program that benefits from an increase in processors, although doubling the number of processors from 50 to 100 only improves performance by a factor of 1.8, and multiplying the number of processors by 4 only improves performance by a factor of 2.4. It is important to note that, for the inputs used in our simulations, these programs have good locality of reference and load balancing properties, and achieve good speedup when contention is not considered. Thus, for all of these programs, memory contention is the major obstacle to effective speedup.

Application	Running Time		
	50 procs	100 procs	200 procs
Gaussian elim	7.4	8.5	15.6
Matrix inversion	26.1	21.7	26.0
Transitive closure	21.7	12.3	9.0
All pairs	43.0	71.3	136.8

Table 1: Running time (in millions of cycles) under row-major allocation.

The effects of contention are magnified even more if we relax some of our optimistic assumptions. For example, if we double the memory latency to 20 processor cycles, the effect of contention is even more pronounced. On 200 processors, 92% of the misses in Gaussian elimination suffer contention (up from 84%), the average remote reference latency increases to 2910 cycles (up from 1546), and the running time increases to 28.8 M cycles (up from 15.6 M cycles). Similarly, if we keep memory latency at 10 cycles and reduce the cache line size to 32 bytes, then 90% of the misses in Gaussian elimination suffer contention, the average remote latency increases slightly to 1571 cycles, and the running time increases dramatically to 30.9 M cycles (since we have doubled the number of remote references). If we both double the memory latency and reduce the cache line size to 32 bytes, then the average remote latency increases to 2904 cycles, and the running time increases to 55.2 M cycles. These results suggest that under less optimistic (and perhaps more realistic) assumptions, memory contention is likely to be an extremely serious problem in the large-scale shared-memory machines we consider.

3.2 The Source of Memory Contention

From the results presented in the previous section, it is obvious that all of our example programs suffer from memory contention. Our hypothesis was that the major component of the performance degradation observed in our experiments was due to simultaneous access to a single row of the matrix, as opposed to a accesses to a single element. We validated this hypothesis with a simple experiment in which we simulated Gaussian elimination on 50 processors, using a matrix that was allocated so that elements within

the same row were placed in different pages. This allocation strategy reduced the average remote access latency from 164 cycles to 83 cycles, which is near optimal. This experiment confirms that the memory contention seen in our examples is due primarily to simultaneous access to the elements of a row, all of which reside in one memory module.

We can also see from our examples that synchronization plays an important role in memory contention. All-pairs shortest paths experiences the worst contention by far, in part because our implementation uses barriers to implement the parallel loop. Transitive closure is similar in structure, but we used locks in its implementation. By using barriers in the all-pairs shortest paths program, we force all processes to access the same row at the same time on every iteration of the outermost loop, thereby increasing contention. To confirm the role of barrier synchronization as a root cause of memory contention in all-pairs shortest paths, we implemented the program using locks instead of barriers on 50 processors. The average latency of a remote memory access fell from 2764 cycles to 247 cycles, and the running time decreased from 43M cycles to 14.4M cycles. It is clear from this experiment that barriers exacerbate the problem of memory contention.[1]

We conclude from these experiments that the major source of contention in our application programs is due to synchronized access to the elements of a single row of the matrix, all of which reside in a single page (or memory module). Although relaxing synchronization constraints (by replacing barriers with locks) helps to reduce contention, we still observe substantial performance degradation due to contention in large-scale machines. In the next section we consider an alternative data allocation strategy designed to address this problem.

4 Reducing Contention with Software Interleaving

Our experiments in the previous section suggest that the main cause of memory contention in our example programs is the row-major allocation we used for matrices. Row-major allocation places an entire row of the matrix in a single page (or memory module), so that access to the row by multiple processors results in memory contention. Since none of our example programs access a matrix by columns, one obvious way to alleviate memory contention is to allocate the matrices in column-major order. That way, each element of a row resides in a different memory module. However, column-major allocation merely trades memory contention for additional cache misses (due to false sharing), and does not solve the performance problem. We require an allocation strategy that has the spatial locality properties of row-major allocation, and the memory contention properties of column-major allocation. *Software interleaving* has both properties.

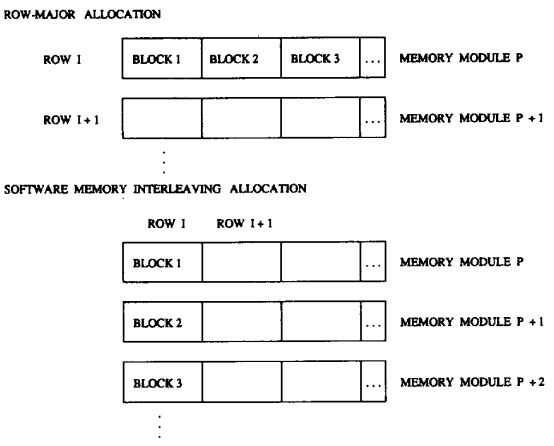

Figure 1: Software Interleaving Matrix Allocation.

4.1 Software Interleaving

In software interleaving, we divide each row of the input matrix into cache blocks, and map the cache blocks of a single row into different memory modules. Figure 1 shows how matrices are allocated under software interleaving.

Software interleaving is a specific instance of the more general data allocation strategy, referred to as *block scattered decomposition* [3], in which the size of the block is determined by the architecture's cache line size. In effect, we use column-major allocation of cache blocks, rather than column-major allocation of elements. Since no cache block contains elements from multiple rows, we eliminate the additional cache misses due to false sharing in column-major allocation. Since the cache blocks of a single row map to different memory modules, no memory contention occurs when multiple processors simultaneously access different cache blocks of the same row.

Software interleaving can be implemented easily by

[1] Note that the effects of memory contention are greater in the lock-based implementation of all-pairs shortest paths than in the lock-based implementation of transitive closure, since there are many more cache misses in all-pairs shortest paths.

the compiler, as it only requires the strip-mining loop transformation, and a slightly more complicated addressing of the interleaved data structures. A smart compiler can also determine whether or not to transform a program based on an analysis of its parallel loops. Note that software interleaving avoids modifying the allocation of work to processors and the synchronization determined by the original loop structure.

Software interleaving has a tremendous effect on the average latency of remote accesses observed by our sample parallel programs. For Gaussian elimination, the average remote access latency on 200 processors is 82 cycles, which is optimal. The results for transitive closure are also close to optimal. Average latency for matrix inversion under software interleaving increases slightly with an increase in processors, but still manages a 6-10 fold decrease in average latency when compared with row-major allocation. And even though all-pairs shortest paths still suffers from contention, which results in an average remote access latency of 366 cycles on 200 processors, software interleaving improves the average remote access latency by a factor of 18 to 33.

This decrease in remote access latency produces a corresponding improvement in running time, as seen in Table 2. Under software interleaving, each of our applications runs faster with an increase in processors. For Gaussian elimination and transitive closure, doubling the number of processors cuts the running time nearly in half. Additional processors also improve the running time of matrix inversion, although not in the same proportion. Even all-pairs shortest paths continues to exhibit improved running time with an increase in processors, although the performance improvements offered by 200 processors are insignificant. The speedup of matrix inversion and all-pairs shortest paths is limited by the use of barrier synchronization; too many processors waste cycles waiting for a barrier.

Software interleaving is also effective at reducing contention under less optimistic assumptions than those used in the majority of our experiments. For example, even if we double the memory latency to 20 cycles, software interleaving eliminates most memory contention in Gaussian elimination. On 200 processors, only 0.78% of the misses suffer contention, the average latency of remote accesses is only 95 cycles, and the running time only increases by 15%. The same observation applies if we reduce the cache line size to 32 bytes. For Gaussian elimination on 200 processors with a cache line size of 32 bytes, only 0.23% of the remote references suffer from contention, the average remote latency is 82 cycles, and the running time is only 4.0 M cycles. (By way of comparison, Gaussian elimination under row-major allocation takes 30.9 M cycles on 200 processors when the cache line size is 32 bytes.) If we both double the memory latency and reduce the cache line size to 32 bytes, then only 0.9% of the remote references suffer from contention, the average remote latency rises slightly to 98 cycles (where the minimum is now 92 cycles), and the running time increases to 4.7 M cycles. Thus, the enormous performance advantages of software interleaving are relatively insensitive to memory latency and cache line size.

Application	Running Time		
	50 procs	100 procs	200 procs
Gaussian elim	7.7	4.5	2.96
Matrix inversion	25.3	15.3	10.1
Transitive closure	21.3	11.8	6.4
All pairs	15.4	10.5	10.3

Table 2: Running time (in millions of cycles) under software interleaving.

The conclusion that software interleaving can effectively eliminate the effects of contention holds even if we allocate multiple data rows to a memory module (rather than assign each row of the matrix to a separate page, and treat each page as a memory module). As long as consecutive rows are allocated in different memory modules, there is no significant contention for data within a memory module other than the contention measured in our simulations.

As a final observation, we note that Gaussian elimination runs slightly faster on 50 processors under row-major allocation than under software interleaving. In this case, the additional addressing costs of software interleaving outweigh the benefits associated with reducing memory contention. We will examine those costs in the next section.

4.2 Overhead in Software Interleaving

As we discussed earlier, software interleaving transforms the code by applying strip-mining to certain loops. The effect of strip-mining is to replace one loop with two, thereby increasing loop overhead. This overhead is not present when using row-major allocation, and therefore increases the running time of any program using software interleaving, unless offset by a reduction in memory contention.

Table 3 illustrates the tradeoff between the over-

Application	Running Time			
	RM-NC	RM-C	SI-NC	SI-C
Gaussian elim	2.4	15.6	2.7	3.0
Matrix inversion	7.7	26.0	8.7	10.1
Transitive closure	6.1	9.0	6.3	6.4
All pairs	4.0	136.8	4.4	10.3

Table 3: Running time (in millions of cycles) with and without contention on 200 procs.

head associated with software interleaving (SI) and the memory contention associated with row-major allocation (RM). In the table, the NC and C suffixes stand for "no contention" and "contention", respectively. In the absence of memory contention (that is, under the assumption that a memory module can satisfy any number of requests simultaneously), all of our programs execute 3-15% faster on 200 processors using row-major allocation, due to the overhead associated with software interleaving. When memory contention is included, software interleaving clearly dominates, improving performance by an order of magnitude in the case of all-pairs shortest paths. Recall from Tables 1 and 2 that software interleaving performs significantly better on 50 processors only for those programs with a large amount of contention (matrix inversion and all-pairs shortest paths). For programs with lower contention levels, software interleaving performs either slightly better (transitive closure) or slightly worse (Gaussian elimination) than row-major allocation on 50 processors. These data suggest that it is not always obvious how to resolve the tradeoffs involved. In the next section we analyze these tradeoffs to determine the circumstances under which to use software interleaving.

5 Determining When to Use Software Interleaving

The previous section presented examples of the benefits of software interleaving, and mentioned some of the tradeoffs associated with the technique. This section develops analytical models that explain why software interleaving usually outperforms row-major allocation, and under what circumstances software interleaving outperforms logarithmic broadcasting.

In each case, it is necessary to consider the two kinds of producer-consumer synchronization separately: barrier synchronization and lock synchronization. Under barrier synchronization, we assume that each task begins trying to access a new matrix row immediately after the barrier. This leads to a different analysis from lock synchronization, in which tasks access rows after a lock is set. Under lock synchronization, conflicts in accessing a matrix row are less frequent.

The metric we will use in our comparison is the increase in running time over the optimal case, which has no memory contention and no additional instruction overhead. We measure the running time of the optimal case by simulating the simplest program (row-major allocation) on a system with infinite memory bandwidth (but nonzero memory latency).

Our purpose in performing these analyses is not to develop highly detailed models that can be used to predict the performance of programs. We focus instead on simple models that provide insight into reasons for preferring one technique over another, and that serve as a means of verifying our understanding of the tradeoffs involved.

5.1 Modeling Software Interleaving and Row-Major Allocation

For a given cache line size and matrix size, the loop overhead introduced by strip mining is a constant number of cycles. These cycles are distributed among the various processors, and therefore have a decreasing effect on running time as we increase the number of processors. The contention effects under software interleaving depend on the form of synchronization. If processes are loosely synchronized (as is the case when we use locks), then the overhead introduced by software interleaving is almost entirely attributed to loop overhead as follows:

$$SI(P) = \frac{L}{P} + K_1$$

where L is the execution time of the additional instructions introduced by strip mining, and P is the number of processors (assuming good load balance). K_1, which is typically small relative to L, represents the small amount of contention that still occurs under lock synchronization. We find that the quantity K_1 is fixed for each of our programs.

Software interleaving can suffer from memory contention when using barrier synchronization, but only for the first cache line of a row. Subsequent accesses to the same row are skewed by the serial access to the first cache line. The overhead of software interleaving in this case is:

$$SI(P) = \frac{L}{P} + RTP$$

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
SI Gauss	7.7	4.5	3.0
Opt All pairs	12.4	6.7	4.0
SI All pairs	15.4	10.5	10.3

Table 4: Running time of Gauss and All pairs (in millions of cycles) under software interleaving, compared to optimal.

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
RM Gauss	7.4	8.5	15.6
Opt All pairs	12.4	6.7	4.0
RM All pairs	43.0	71.3	136.8

Table 5: The running time of Gauss and All pairs (in millions of cycles) under row-major allocation, compared to optimal.

where R is the number of rows in the matrix, and T is the transfer time of a cache line (82 cycles).

As seen in Table 4, our experimental results agree with this analysis. For Gaussian elimination, we measure L as approximately 50M cycles and K_1 as approximately 300,000 cycles. For all-pairs shortest paths, we measure L as approximately 70M cycles; from the program, we know that R is 400, and as noted above, $T = 82$. These parameters result in good agreement with the data in all cases.

In contrast, row-major allocation adds no additional loop overhead. However, it suffers serious contention under both barrier and lock synchronization. Under barrier synchronization, all processors contend for the entire row. Since all rows are eventually required by all processors, row-major allocation under barrier synchronization adds overhead equal to the cost of transferring the entire matrix, times P. This is because the last processor to receive a row will get it after $P-1$ other row transfers have completed. Under barrier synchronization, all the other processors will be forced to wait for the last processor at the next barrier, so all are slowed equally. In other words:

$$RM(P) = \frac{M}{E}TP$$

where M is the number of elements in the entire matrix, and E is the number of elements per cache line.

Under lock synchronization, contention occurs due to random conflicts between processors, as before. However, random conflicts are more common, since processors access a single module repeatedly while transferring a row, and the demand for a particular row tends to be greatest immediately after it is produced. In fact, we can determine from the characteristics of our simulated machine that under row-major allocation, it only takes 8 processors transferring rows to saturate a memory module. Since the network trip lasts for 72 cycles, but the memory access itself only takes 10 cycles (which we will call *service time*), no more than 7 consecutive memory accesses can occur during a network trip.

Beyond a certain number of processors, we can expect that at any point in time, at least one memory module is saturated. This observation holds because there are only a fixed number of memories in use; adding more processors adds to the number of requests sent to each memory. The delay caused by a memory module's saturation is eventually propagated to all processes, since each processor (in addition to consuming rows) is producing a row that eventually the other processors will need.

Thus, although it is difficult to model the random contention for memory when the number of processors is small, we can provide an estimate of overhead when the number of processors is large. This estimate is based on the assumption that at any point in time, some module is saturated. We can then see that each additional processor adds an additional service time to the transfer of each cache line, since the additional processor will likely access the module while it is saturated. This means that each additional processor adds the cost of an entire matrix's memory service time, or 10 cycles times the number of cache lines in an entire matrix. So we estimate the overhead of row-major allocation, for large P, and lock synchronization, as:

$$RM(P) = \frac{M}{E}C(P-\theta)$$

where C is the memory's service time (10 cycles), and θ is the threshold number of processors beyond which the system shows memory saturation.

As seen in Table 5, our experimental results for row-major allocation generally confirm our analysis. For all-pairs shortest paths, where $M = 400^2$, our predictions are about 30% too high; however, these running times are extremely long and our model predicts them well enough for comparison purposes with software interleaving. For Gaussian elimination, we determine by inspecting the data that memory satu-

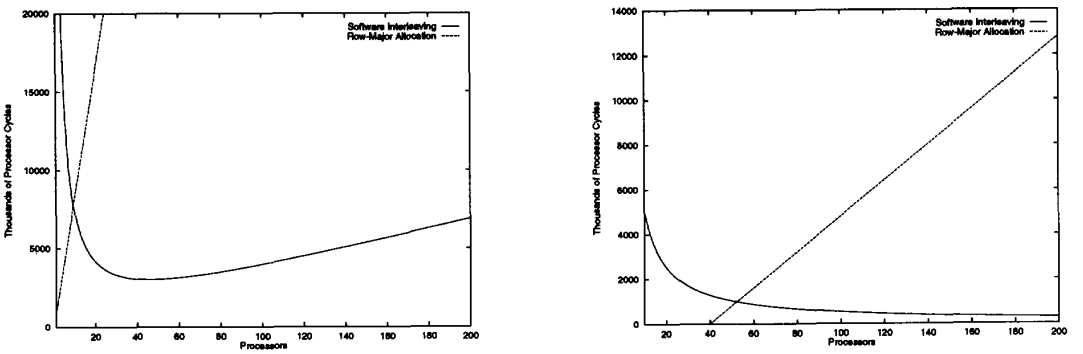

Figure 2: Overhead of Row-Major Allocation compared to Software Interleaving for Barrier (Left) and Lock Synchronization (Right)

ration is reached at about 40 processors, so $\theta = 40$; also, since pivot rows only constitute the upper half of the matrix in Gaussian elimination, $M = 512^2/2$. Our model of overhead for lock synchronization is then quite accurate.

Using this analysis, we can determine when the extra cost of software interleaving is worth paying in exchange for the reduction in contention that it provides. Figure 2 shows plots of the analytic models developed above, for the cases of all-pairs shortest paths (on the left) and Gaussian elimination (on the right). The all-pairs graph shows that under the high contention costs of barrier synchronization, software interleaving is preferable even on as few as 10 processors. Beyond about 50 processors, the cost of software interleaving begins to rise, but at a slower rate than the cost of row-major allocation. This trend reflects the difference between contending for the first cache line of the row in the interleaving case, and contending for the entire row in the row-major case.

The analytic models for lock synchronization in Gaussian elimination are plotted on the right side of Figure 2. Since contention under lock synchronization starts more slowly than under barriers, more processors are required before software interleaving is preferred over row-major, but the same basic effect is observed: beyond some number of processors (in this case about 50) software interleaving is always preferable.

5.2 Comparing Software Interleaving and Logarithmic Broadcasting

The previous section showed that, as the number of processors increases, eventually there comes a point when it is more profitable to use software interleaving over row-major allocation. However, to adequately assess when to use software interleaving, we must compare it to the best known software alternative: logarithmic broadcasting.

We implemented two versions of broadcasting for the row-major Gaussian elimination program. The two versions differ in terms of who drives the broadcast, the producer or the consumers of the data. As our consumer-driven implementation performed significantly better, it is the only one we will present results for.

As pointed out in the last section, 8 processors reading a row can saturate a memory module when the memory latency is 10 cycles and the network latency is 72 cycles; however, as long as the number of processors contending is less than 8, each processor is delayed only a small amount. Thus, in our simulated machine, logarithmic broadcasting should not use a tree of degree greater than 8. With this assumption, logarithmic broadcasting can completely eliminate contention when used with lock synchronization. This is because the condition in which some memory module is always saturated does not occur, as it did under simple row-major allocation. Memory modules do not saturate since the complete broadcast of each row is implemented using a much larger set of memory modules, and the number of processors accessing a single module will never be greater than the degree of the tree.

For this reason we can estimate the cost of logarithmic broadcasting under lock synchronization as a constant, which is equal to the extra instructions and synchronization necessary to implement the technique. Thus,

$$LB(P) = K_2$$

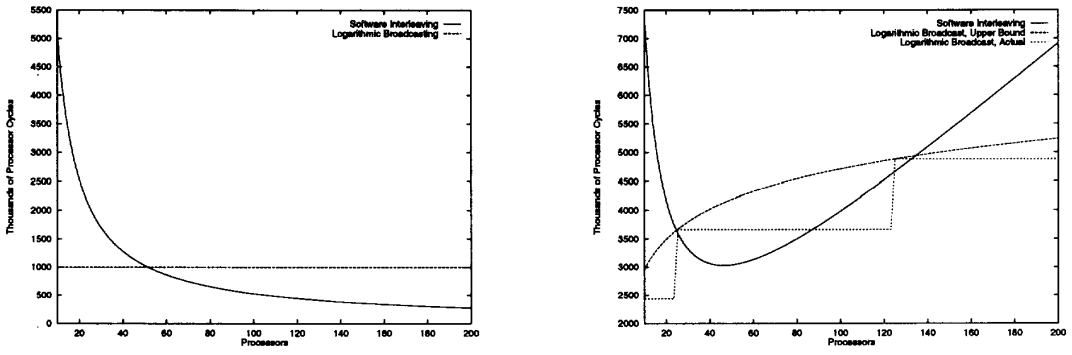

Figure 3: Overhead of Logarithmic Broadcasting compared to Software Interleaving for Lock (Left) and Barrier Synchronization (Right)

where K_2 depends on the specific program. Interestingly, in the programs we studied, K_2 was significant; for example, in Gaussian elimination, $K_2 = 1.0$M cycles. This occurs partly due to the synchronization needed to access broadcast buffers. Ideally each row would have a broadcast buffer on each processor, but that would require expanding the memory usage of the program by a factor of P, which is impractical. Since the amount of buffer space used for row broadcast on each processor must be bounded, buffer space must be re-used, which requires synchronization.

In contrast, under barrier synchronization, the cost of logarithmic broadcasting is not independent of P. The broadcast of each row requires d steps, where $d+1$ is the depth of the broadcast tree.[2] For a tree of degree r, each step requires r row transfers. The first row causes a delay equal to its transfer time; the other rows cause a delay equal only to their memory service times (as discussed earlier in this section). Thus we can estimate the overhead of logarithmic broadcasting under barrier synchronization as:

$$LB(P) = d\frac{M}{E}T + d(r-1)\frac{M}{E}C$$

where d is proportional to $\lceil \log_r P \rceil$.

In our experiments we held d equal to 3, while we varied r to attain the lowest possible value consistent with $d = 3$. For the 50 processor case, we set $r = 4$; for $P = 100$, $r = 5$; and for $P = 200$, $r = 6$. Table 6 shows the results of our experiments with All pairs and Gaussian elimination under logarithmic broadcasting, and compares them to their ideal cases. The table shows that $K_2 = 1.0$M cycles is a good estimate of the constant overhead for Gaussian elimination under logarithmic broadcasting. It also shows that our estimate of the overhead due to logarithmic broadcasting under barriers in all-pairs shortest paths is fairly accurate.

Application	Running Time		
	50 procs	100 procs	200 procs
Opt Gauss	6.5	3.7	2.4
Broad Gauss	7.4	4.7	3.4
Opt All pairs	12.4	6.7	4.0
Broad All pairs	15.9	10.3	7.8

Table 6: The running time of Gauss and All pairs (in millions of cycles) under logarithmic broadcasting, compared to optimal.

Figure 3 shows how the two techniques compare. The comparison for lock synchronization is on the left, while the comparison for barrier synchronization is on the right. For lock synchronization, beyond about 50 processors, software interleaving performs better than logarithmic broadcasting. This is because the fixed overhead under software interleaving is lower than that under logarithmic broadcasting. Since contention is much less severe under lock synchronization, the extra cycles required to implement logarithmic broadcasting are more expensive than necessary; software interleaving is preferable due to its simplicity.

The situation is different for barrier synchronization, as shown on the right side of Figure 3. This figure shows the overhead of software interleaving compared to logarithmic broadcasting using a tree of fixed degree (equal to 5). The step-function nature of the log-

[2] For a tree of degree r, the depth of the broadcast tree is roughly $\lceil \log_r P \rceil$, although details of how the tree is constructed can change this value by 1 in some cases. In all our experiments, $d = 3$.

arithmic broadcasting curve is due to changes in the depth of the tree as the number of processors increases. The figure also shows an upper bound on logarithmic broadcasting to show that as P grows large, logarithmic broadcasting eventually outperforms software interleaving everywhere. This figure shows that under barrier synchronization contention is so severe that the linearly increasing costs of accessing the first cache line in each row under software interleaving eventually grow larger than the logarithmically increasing costs of broadcast.

Figure 3 shows that for large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but software interleaving is best when using lock synchronization. It also shows that for small numbers of processors, the situation is reversed: software interleaving is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization.

6 Conclusions

In this paper we used detailed simulations of application kernels to show that memory contention can substantially degrade the performance of SPMD computations on large-scale shared-memory multiprocessors based on multi-stage interconnection networks. We showed that under row-major allocation, memory contention is due to synchronized access to entire rows of a matrix, rather than simultaneous accesses to isolated data elements. We also showed that software interleaving dramatically reduces memory contention, and therefore performs much better than row-major allocation on large-scale machines.

We analyzed the costs associated with software interleaving and logarithmic broadcasting, and showed how the choice between these two techniques for alleviating memory contention depends both on the type of synchronization used and the number of processors. For large numbers of processors, logarithmic broadcasting is best when using barrier synchronization, but software interleaving is best when using lock synchronization. For small numbers of processors, the situation is reversed: software interleaving is best when using barrier synchronization, while logarithmic broadcasting is best when using lock synchronization. Since the use of barrier synchronization exacerbates memory contention, we conclude that software interleaving and lock-based synchronization is the most effective combination for reducing memory contention in SPMD matrix computations on large-scale machines.

References

[1] R. Bianchini and T. J. LeBlanc. Eager Combining: A coherency protocol for increasing effective network and memory bandwidth in shared-memory multiprocessors. In *Proceedings of the 6th Symposium on Parallel and Distributed Processing*, October 1994.

[2] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing*, pages II–99 – II–107, August 1991.

[3] J. Dongarra, R. van de Geijn, and D. Walker. A look at scalable dense linear algebra libraries. In *Proceedings of the 1992 Scalable High Peformance Computing Conference*, pages 372–379, 1992.

[4] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel algorithms for dense linear algebra computations. *SIAM Review*, 32(1):54–135, March 1990.

[5] A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir. The NYU ultracomputer - designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb 1983.

[6] J. M. Ortega and C. H. Romine. The ijk forms of factorization methods ii. parallel systems. *Parallel Computing*, 7:149–162, 1988.

[7] G. F. Pfister, W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, and J. Weiss. The IBM research parallel processor prototype (RP3): Introduction and architecture. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 764–771, Aug 1985.

[8] Larry Wittie and Creve Maples. MERLIN: Massively parallel heterogeneous computing. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I–142 – I–150, August 1989.

[9] Pen-Chung Yew, Nian-Feng Tzeng, and Duncan H. Lawrie. Distributing hot-spot addressing in large-scale multiprocessors. *IEEE Transactions on Computers*, C-36(4):388–395, April 1987.