Submitted for publication.

Multiprocessor Runtime Support for Fine-Grained, Irregular DAGs

Frederic T. Chong* Shamik D. Sharma† Eric A. Brewer‡ Joel Saltz§
ftchong@ai.mit.edu shamik@cs.umd.edu brewer@lcs.mit.edu saltz@cs.umd.edu

Massachusetts Institute of Technology
University of Maryland

Abstract

We examine multiprocessor runtime support for fine-grained, irregular directed acyclic graphs (DAGs) such as those that arise from sparse-matrix triangular solves. We conduct our experiments on the CM-5, whose lower latencies and active-message support allow us to achieve unprecedented speedups for a general multiprocessor. Where as previous implementations have maximum speedups of less than 4 on even simple banded matrices, we are able to obtain scalable performance on extremely small and irregular problems. On a matrix with only 5300 rows, we are able to achieve scalable performance with a speedup of almost 35 for 128 processors.

We achieve these speedups with non-matrix-specific methods which are applicable to any DAG. We compare a range of run-time preprocessed and dynamic approaches on matrices from the Harwell-Boeing benchmark set. Although precomputed data distributions and execution schedules produce the best performance, we find that it is challenging to keep their cost low enough to make them worthwhile on small, fine-grained problems.

Additionally, we find that a policy of frequent network polling can reduce communication overhead by a factor of three over the standard CM-5 policies. We present a detailed study of runtime overheads and demonstrate that send and receive processor overhead still dominate these applications on the CM-5. We conclude that these applications would highly benefit from architectural support for low-overhead communication.

1 Introduction

Our goal is to execute fine-grained, irregular directed acyclic graphs (DAGs) efficiently on general-purpose, distributed-memory multiprocessors. We explore a range of runtime methods, some of which involve *runtime preprocessing*, an approach shown to be effective with iterative computations [SMC91]. We test our methods on an important real-world application, sparse-matrix triangular solves. In order to maintain generality, we do not use matrix-specific optimizations, yet we achieve unprecedented performance.

Our methods depend heavily upon active messages [E+92] and are closely related to the dataflow paradigm [ACM88]. Active messages greatly reduce overhead by providing user-level communications. We also take full advantage of active message handlers to decrease synchronization and data-copying costs. These mechanisms allow us to take a data-driven approach with fine-grained

*Fred Chong is supported in part by an Office of Naval Research Graduate Fellowship and ARPA contract N00014-91-J-1698

†Shamik Sharma is supported in part by EPRI (RP3103-06), ARPA (NAG-1-1485), ONR (SC292-1-22913), and NSF (ASC 9213821).

‡Eric Brewer is supported in part by the National Science Foundation, grant CCR-8716884; by ARPA, contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM.

§Joel Saltz is supported in part by EPRI (RP3103-06), ARPA (NAG-1-1485), ONR (SC292-1-22913), and NSF (ASC 9213821).

```
do i = 1, n
  x(i) = b(i)
  do j = low(i), high(i)
    x(i) = x(i) - t(j)*x(column(j))
  end do
end do
```

Figure 1: An example of a sparse triangular solve

synchronization. This approach, coupled with reasonable hardware support, facilitates much of our speedups.

Surprisingly, there have been few studies of irregular, fine-grained DAGs in the dataflow community. Rubin [Rub92] has studied regular, fine-grained DAGs arising from a conjugate gradient problem. While Rubin performed his study on the Monsoon dataflow machine, Yeung and Agarwal [YA93] have studied the same problem on Alewife, a shared-memory, SPARC-based multiprocessor.

Chakrabarti and Yelick [CY93] have studied the Gröebner-basis problem, which results in an irregular DAG. However, this computation is not as fine-grained as the applications we examine. The Gröebner study used consumer-driven techniques, using multithreading to hide the latency of fetching data. Since our application performance can be critical-path bound, we take a data-driven approach instead.

The remainder of the paper is organized as follows: In Section 2, we will describe sparse triangular solves and why they are important. In Section 3, we will describe our experimental platform. Section 4 contains the bulk of our experimental results, as well as describing the runtime methods used in the experiments. Section 5 discusses the costs and benefits of run-time preprocessing. Section 6 describes why the network should be frequently polled. Section 7 presents a detailed breakdown of our application overheads. Given these overheads, we extrapolate the benefits of architectural support for lower-overhead communication in Section 8. We then present our concluding remarks.

2 Sparse Triangular Solves

For this study, we selected sparse-matrix triangular solve as the source of our fine-grained, irregular DAGs. While parallel computation has generally focused on the easy problems of dense matrix computation, sparse matrices represent a much larger class of scientific computation. Sparse systems arise from simulations of integrated circuits, electric fields, physical structures, power grids, and just about any other real-world system.

Sparse triangular solves involve finding a vector $\vec{x}$ in an equation of the form ($\mathbf{T}\vec{x} = \vec{b}$), where $\mathbf{T}$ is either an upper or lower triangular matrix and $\vec{b}$ is a vector of values. The code for such a computation is given in Figure 1. Figure 2 illustrates the DAG formed by a sparse, lower-triangular matrix. Note that the dependencies can be quite irregular. Additionally, each arc in the DAG represents only one multiply-accumulate operation. This extremely fine-grain computation, coupled irregular dependencies, makes sparse triangular solves a challenging benchmark for the runtime strategies we will support.

Sparse triangular solves are important in solving linear systems of equations of the form $\mathbf{A}\vec{x} = \vec{b}$, where $\vec{x}$ is a vector of unknowns and $\vec{b}$ is a vector of values. There are various techniques for solving such systems of equations. In direct methods, this system is *completely* factored into $\mathbf{LU}\vec{x} = \vec{b}$ where $\mathbf{L}$ is a lower-triangular matrix and $\mathbf{U}$ is an upper-triangular matrix. The system can then be solved through a straightfoward forward substitution ($\mathbf{L}\vec{y} = \vec{b}$) and a backward substitution ($\mathbf{U}\vec{x} = \vec{y}$). In preconditioned iterative methods, the matrix $\mathbf{A}$ is *incompletely* factored into lower and upper triangular matrices $\mathbf{L}$ and $\mathbf{U}$, which are then used in iterative triangular solves to obtain a solution for $\vec{x}$.

Sparsity patterns of the factored triangular matrices used in direct methods are very different from the incompletely factored matrices. The completely factored matrices have substantially greater

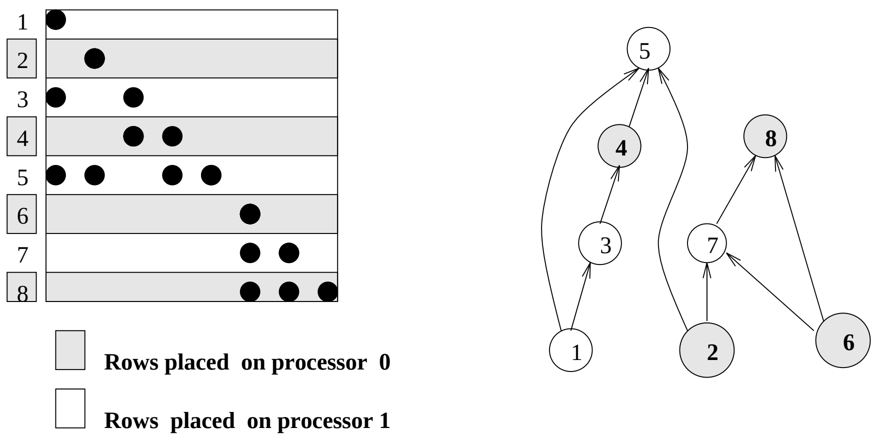

Figure 2: A lower triangular matrix (left) and the resulting forward subsitution DAG (right). The filled circles on the left represent nonzeros in the matrix. The nodes of the DAG correspond to rows of the matrix.

---

fill-in than the incomplete factors.

Our experimental results are based on incompletely factored matrices. We perform parallelization at row-granularity (i.e all computation pertaining to a row is assigned to a single processor). This is a very fine granularity level, with only a few floating point operations per row. With completely factored matrices, we find that even at this level of granularity, parallelism is limited. This is because fill-in during factorization causes the sparsity patterns of completely factored matrices to be very different from those of incompletely factored matrices. In the final version of this paper, we will describe our experiments with solving completely factored matrices at an even finer level of granularity (by distributing accumulate and multiply operations across processors).

Efficient parallel implementation of triangular solves has been a difficult problem. This problem is due to the extremely small grain of the computation. Each row depends only on a few other rows, and each dependency involves a single multiply-accumulate (generally double-precision floating point). Speedups are rarely reported for sparse triangular solves. Heath and Raghavan [HR93] report good results for factorization on the Intel iPSC/860, but their solves exhibit constant or decreasing performance as the number of processors increases. They obtain speedups of no more than 3 or 4 regardless of the number of processors, even for banded matrices. They downplay the importance of the solve by noting that the running time of the factorization phase dominates that of the solve phase. Lucas, Blank, and Tiemann [LBT87] report similar results. While the time spent in a single triangular solve is much less than the time spent in factorization, there are many applications where the solve time is important. Often there are tens to hundreds of solves performed using a single factorization. This is true not only for preconditioned iterative methods, also for applications using direct methods, where the same factorization of **A** may be used to solve for different right hand side vectors $\vec{b}$. For example, the time spent performing solves is equal to or double that of the time spent factoring in the sequential execution of the ETMSP power grid code [ETM93].

We would like to note that in applications where the solve is repeated several times, it may be more efficient to invert the triangular matrix, converting the solve into an easily parallelizable matrix vector multiplication [AS92]. It is possible to eliminate fill-in during the inversion [PA92]. We have not investigated this technique in this paper.

[0] We estimated speedups assuming a sequential solve speed similar to that of uncached DAXPY on a single iPSC/860 node [Moy91].

3 Experimental Platform

In this section, we describe the multiprocessor hardware and communications software that we use in our experiments.

3.1 CM-5

We perform our experiments on the CM-5 [Thi93a]. Each node of the CM-5 has a 33 MHz SPARC processor as well as four vector units. The CM-5 has a fat-tree-based network [L^+92] which provides fairly uniform communications delays and sustainable bandwidth. It also has a control network for global operations. To keep our study architecturally general, we do not use the vector units, nor do we use the control network.

3.2 Strata

For efficient communication and accurate timing, we made extensive use of the Strata communications library [BB94]. Strata is a CMMD-compatible [Thi93b] package with high-performance communication and extensive support for timing and debugging. Section 6 discusses the impact of using Strata's communication primitives.

The standard CMMD timers on the CM-5 require kernel calls for each timing event, which cost hundreds of cycles. Strata, however, provides user-level access to the cycle counter on the network interface of every CM-5 node. The overhead for timing with the cycle counter is about 18 cycles: Strata subtracts out the overhead to achieve interval timing with single-cycle resolution. Although we can accurately time a single interval, timing many intervals inflates the execution time and may affect the ordering of interprocessor events. Given the fine-grain nature of the application, adding 18 cycles per timing can be substantial. In cases where this inflation was significant, we instead used the mean difference between code with and without the property of interest. Section 7 covers this technique in more detail.

In general, we timed events that lasted at most one-tenth of the 16.6 milliseconds between timer interrupts. This ensures that the cycle counts are rarely affected by the interrupts, time-slicing, or other processes. [BK94] covers CM-5 timing methodology in more detail.

4 Runtime Methods

We adopt a data-driven approach to DAG computation. Active messages provide an efficient mechanism to communicate the data and synchronize the computation. At any given time, several tasks in the DAG may be ready to execute. The next task may be selected dynamically at compute-time. However, we may also adopt preprocessed schemes, commonly known as *inspector-executor* strategies [SMC91].

A preprocessing step (called an inspector) is used to optimize the execution of the actual computation step (called the executor). The runtime cost of the one-time preprocessing step is amortized over all executions of the executor. Depending on how many times the executor step is carried out, preprocessing of varying complexity and quality can be carried out. We have examined various preprocessing schemes and demonstrate the performance impact of each on sparse triangular solves.

As with any computation, parallelizing a DAG involves determining a "good" distribution of data and computation across processors - a good distribution being one that minimizes communication and maximizes parallelism. Unlike computations without dependencies, parallelizing a DAG also involves determining a computational ordering of the tasks allocated to each processor. Furthermore, each computation must be scheduled to execute only when all incoming dependencies have been satisfied. This requires the specification of suitable synchronization points in the computation. We call the computational structure resulting from distribution, ordering, and synchronization of tasks a *computational schedule*. Determining a computational schedule requires runtime preprocessing, wherein the dependencies are examined before the actual execution of the DAG. We have examined different preprocessing schemes and show the performance impacts of each on our application.

4.1 Task Distribution

For task distribution, we have considered three schemes - block distribution, cyclic distribution and Dominant Sequence Clustering (DSC) [GY92]. We considered block and cyclic distributions of nodes across processors because of their simplicity. While these distributions do not optimize either communication or load balance, their computational preprocessing costs are neglibile. The performance of these regular distributions provides us with a lower bound against which we can compare the performance of other distributions. At the opposite end of the spectrum there are computation-intensive task clustering schemes which look at the entire DAG in order to minimize the critical path length of the DAG [Sar89] [Yan93]. For example, Dominant Sequence Clustering, obtains an allocation by incrementally reducing the critical path length through successive clustering steps. Comparisons show that DSC outperforms most other static-task allocation scheme in both processing time and quality of computation schedules. DSC was intended for statically clustering and scheduling static task graphs at runtime, and is computationally very expensive. We used DSC in our as a runtime preprocessing step in order to determine the performance gains of using a sophisticated, albeit costly, clustering scheme.

4.2 Ordering

Once nodes of the DAG have been distributed across processors, we then need to determine an order in which local elements will be computed. Not all orderings are valid: node B cannot be scheduled before node A if there is a dependence from node A to node B. Determining an optimal ordering is an NP-complete problem. Nonetheless, various heuristics can be applied to obtain schedules of varying quality.

One way to achieve a good ordering without any preprocessing cost is to employ a dynamic data-driven scheme in which each incoming message triggers the appropriate computation. On the CM-5, this can be done via active message handlers. For example, in triangular solves, the computation at each node represents an accumulation. The final incoming data element triggers another active message which sends the computed data element to all other nodes that need the result. The data driven scheme automatically ensures that all dependencies are respected.

Instead of adopting a data-driven method, we can obtain an ordering via preprocessing. For example, global level scheduling, as in [Sal90], partitions the nodes of the DAG into *levels* $L_1 \ldots L_n$, such that the nodes in level L_i depend only on nodes in $L_1 \ldots L_{i-1}$. Each processor orders its local nodes based on their levels. The resulting schedule is called a *level schedule*. Partitioning the nodes into levels involves a topological sort over the global DAG.

The topological sort, needed to obtain a global level scheduling, has the same fine-grained irregular computational characteristics as a triangular solve. Consequently, the runtime methods we describe for the triangular solve can also be applied to parallelize the topological sort. Currently, however, we perform level scheduling sequentially.

While level scheduling is a reasonable heuristic, it does not try to schedule rows in an order that can overlap computation with communication. For example, the PYRROS package [Yan93], which generates the DSC clusters, also provides a task ordering based on the Ready Critical Path heuristic. This heuristic tries to schedule those tasks on the critical path first so that dependent nodes can be executed earlier. This schedule is much more costly to compute than level scheduling, but given a DSC partition, it can be expected to provide a good ordering.

We have also tried combining dynamic data-driven with prescheduled task distributions. In this hybrid scheme, we use a precomputed DSC mapping and schedule of tasks to processors, but incoming data elements can activate any ready task if the next DSC-scheduled row is not ready.

Note that DSC runtime approaches imply both a task distribution and a schedule.

4.3 Presence Counters

Each node of a DAG must wait until all of the data along its incoming dependencies have arrived before performing its computation and forwarding the result along its outgoing dependencies. In order to ensure such synchronization, we use *presence counters* [ACM88]. A presence counter is used to count the number of items that have arrived. If the computation to be performed on incoming data elements is non-commutative, the presence counter must be placed at the beginning of each task to check for the arrival of all data before computation. On the other hand, if the computation is commutative (as in the accumulates of a triangular solve), the presence counter can be placed at the end of the task to detect completion and indicate forwardability. Checking the presence counter at the end of the computation enhances parallelism since useful computation can be carried out instead of idling while waiting for data. Presence counters are necessary even when tasks are prescheduled since data arriving from off-processor may suffer unbounded delays. This phenomenon is because most modern multiprocessor networks, including the CM-5, do not guarantee delivery within a well-defined amount of time.

Consequently, all of our implementations use presence counters. Most use one counter per matrix row, but the coarse-grain, level-scheduled version attempts to reduce overhead by keeping one counter for many rows. This savings, however, is at the cost of reduced parallelism, since every row sharing the same counter must wait for the presence counter, even if some of them are ready early. We have examined implementations, with both early and late presence checking. We will also precisely measure the cost of maintaining presence counters in Section 7.

4.4 Methods Tested

In this section, we describe several versions of sparse triangular solve implementations. Each of these versions is intended to demonstrate the effectiveness of particular aspects of the runtime support.

Dynamic blocked: This version uses a block distribution of rows based on original node numbers. This is a dynamic implementation without any prescheduled ordering of rows. The computation is ordered by a ready queue, resulting in an order similar to breadth-first traversal of the DAG.

Dynamic cyclic: This version is identical to the one above, except that it uses a cyclic distribution of tasks.

Fine-grained Level-scheduled cyclic: This version distributes rows by cyclicly and uses level-scheduling to order the tasks on each processor. It uses a presence-counter for each row. The check is done at the start to detect when all required data has arrived.

Coarse-grained Level-scheduled cyclic : Rows are distributed in a cyclic fashion. Rows are ordered using level-scheduling and all rows on a processor that belong to the same level are placed in a *group*. Each group of rows shares a presence counter. The presence counter is used to detect readiness. Incoming data is buffered until all data required for execution of the next group has arrived. This scheme is aimed at reducing the overhead of presence-counter checking, but suffers from reduced parallelism.

Prescheduled DSC strict: This is a dynamic implemantaion which uses a precomputed DSC distributon and ordering. The schedule is followed strictly. If a prescheduled row is not ready to be computed, we poll the network until the required data arrives.

Prescheduled DSC hybrid: The rows in this scheme are distibuted according to a DSC distribution. Howeever, this scheme does not follow the DSC ordering strictly. When the next row in the schedule is not ready, we allow other rows to computed if they are ready for execution.

4.5 Benchmarks

We chose four power grid matrices from the Harwell-Boeing [DGL92] benchmark set for our experiments. These sparse matrices come from the BCSPWR set, and represent parts of the Western US Power Network and the Eastern US Power Network. Our matrices were incompletely factored, with the sparsity pattern being identical to that of the original matrix.

In Table 1, we have tabulated the relevant characteristics of these matrices. As can be seen from the table, the matrices are symmetric, very sparse and relatively small. *Max Speedup* is an upper bound calculated by the number of rows divided by the critical path. *DSC Speedup* is the speedup (up to 128 processors) predicted by the DSC scheduler, which uses an ideal graph model incorporating simple communication and computation costs on the CM5.

Table 1: The benchmark BCSPWR matrix set

	Rows	NonZeros	Max Speedup	DSC Speedup
bcspwr05	274	3102	49.2	10.5
bcspwr06	1454	10134	103.9	20.6
bcspwr07	1612	11157	115.1	21.9
bcspwr10	5300	40716	481.8	77.3

4.6 Results

Figure 3 summarizes the performance of all of our approaches on each of our benchmarks. The times for one processor come from an optimized sequential code running on a single CM-5 SPARC node. For some of the slower schemes, this explains the rise in execution time from one processor to four processors.

The performance of the dynamic block and cyclic approaches are similar except for bcspwr10. The block distribution is worse in this case because the sparse triangular solve tends to proceed in a wavefront from the first row towards the last row. Our block distribution assigns rows in linear blocks. This distribution can sequentialize this wavefront. The level scheduling schemes provide similar or worse performance. The coarse-grained-level scheme provided little gain over the fine-grained version. As we shall see in Section 7, presence counter overhead is a relatively small fraction of total overhead.

In general, the prescheduled DSC-strict approach performs the best. Since our matrices are relatively small and possess limited parallelism, performance differences are most pronounced in the smaller numbers of processors where DSC can do a better job of optimizing the critical path. As the number of processors increases, DSC has a harder time distributing load without adding communication cost to critical paths. The exception is bcspwr10, our largest matrix, which does possess enough parallelism for all schemes to do fairly well. The fact that the DSC-hybrid approach does not perform as well as the strict approach suggests that very little time is spent waiting for the next scheduled row when another row could be worked on. The added overhead of the hybrid approach clearly negates any gain from decreased wait time.

Overall, the speedups for both dynamic and preprocessed schemes are quite good for such small, fine-grained problems.

5 Preprocessing Costs

In this section, we compare the cost of preprocessing a DAG to the speedups gained by the resulting schedule and partition. We discover that it is very difficult to make the cost of prescheduling worthwhile at our small computational grain and small problem sizes.

In examining the costs, we must consider both the time to compute the DAG distribution and execution schedule and the time to move the data from its initial distribution to the new distribution. In general, the redistribution of data can be considered a global permutation. Global permutations on the CM-5 can be performed at roughly 2.5 Mbytes per second per processing node [BK94]. The upper triangular bcspwr10 matrix, for example, has 5300 rows, each represented by approximately 24 bytes. There is also a symmetric lower triangular matrix which we must store separately. In the 8 processor case, this gives each node approximately 32 Kbytes to contribute to the permutation. This means that we can complete our permutation in about 13 ms.

Using DSC, bcspwr10 on 8 processors runs about 2.1 ms faster than the dynamic runtime using a cyclic distribution. Iterative methods can take from tens to thousands of iterations to converge. Assuming that we repeat our solve 100 to 1000 times, and multiplying by 2 for both forward and backward solve, we get a total savings of 420 to 4200 ms. Given the low cost of the global permutation, we can apply most of this time against our preprocessing algorithm.

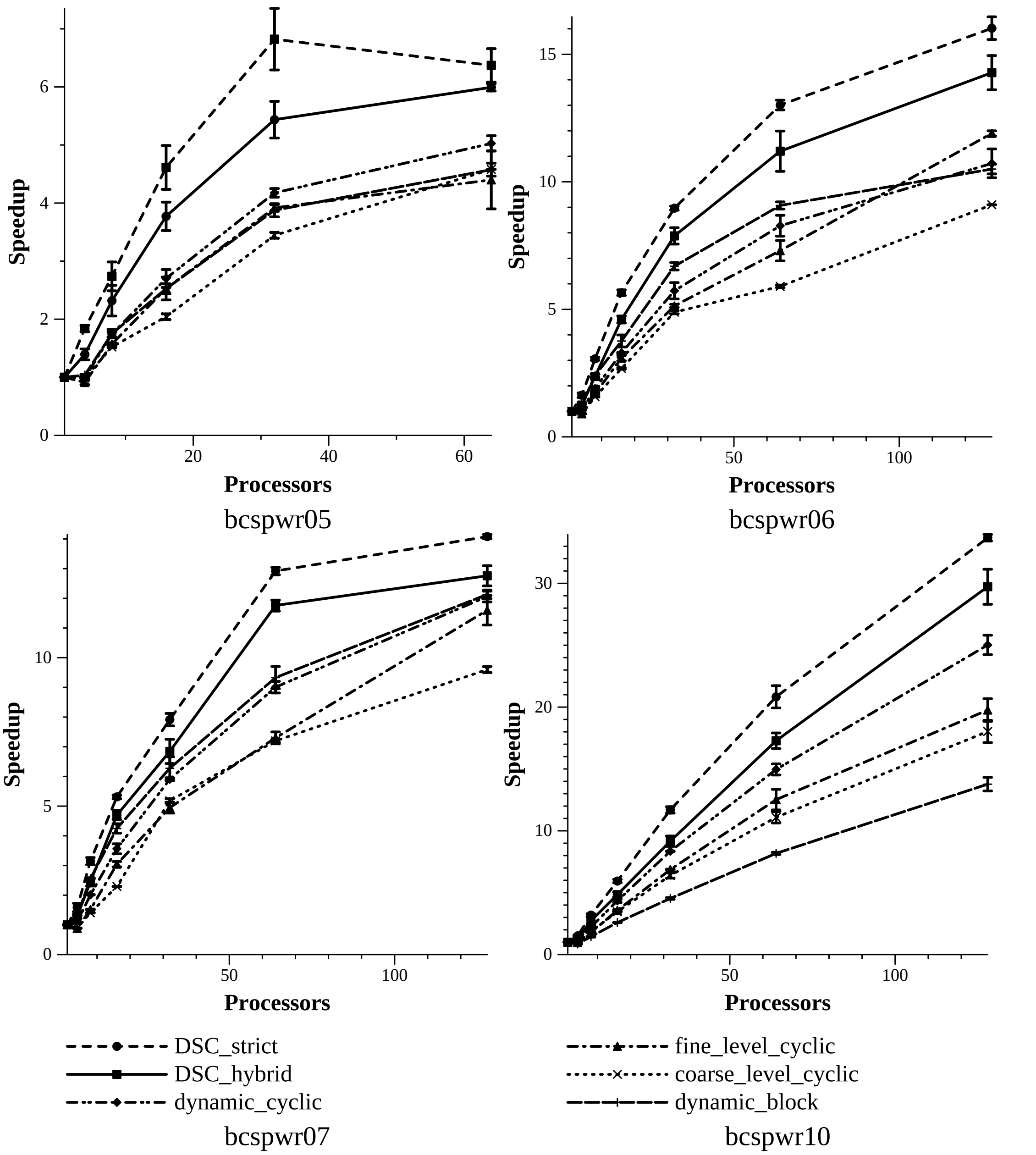

Figure 3: Performance curves for forward substitution using a range of runtime methods on four benchmark matrices. Note that the scales differ between graphs in order to provide maximal resolution between curves within each graph.

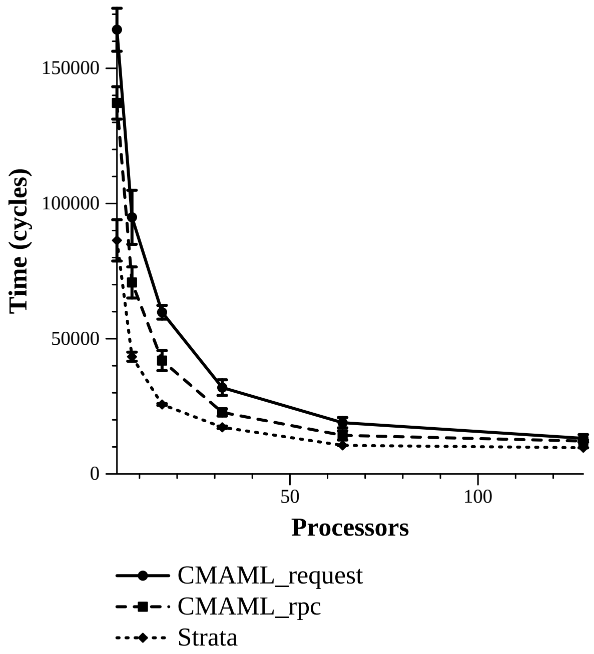

Figure 4: Performance curves for bcspwr07 with static DSC runtime using various send-with-polling routines. Cycle time is 30 ns.

DSC was intended for compile-time use and we only have a sequential implementation to measure. The running time of DSC on a SPARC 1+ (similar in power to a CM-5 SPARC node) on bcspwr10 is on the order of 20 seconds. Even assuming a parallel implementation with perfect speedup on 8 processors, the number of iterations required to make DSC worthwhile is likely to be on the high end.

Fortunately, some applications have a static non-zero structure through different factorizations. Although the values of the matrix we are solving may change, the structure stays the same and we can use the same schedule and data mapping. This may allow us to assume benefits for many thousands of iterations for the cost of one preprocessing computation.

Nonetheless, we conclude that we need even simpler prescheduling algorithms and may need to apply them to larger problems with larger grain size. Level scheduling is significantly cheaper but does not calculate a task distribution. In the final version of this paper, we will take a closer look at alternative schemes and their costs.

6 Network Polling

Given the fine-grained nature of the application, we expected active-message overhead to be a substantial fraction of total overhead. Somewhat unexpected was the importance of network-polling policy.

Figure 4 plots the performance of bcspwr07 with the DSC-prescheduled-strict runtime using three different versions of active messages. The `CMAML_rpc` version uses `CMAML_rpc` for sending packets, which only polls if the outgoing packet is blocked. As a result, a processor can go for a long time without polling. This in turn causes other processors to stall until a poll occurs. The key problem is the relative independence of the ability to inject and the need to poll (to allow others to make progress).

We attempt to remedy this problem with our second version, which uses `CMAML_request` to inject packets. `CMAML_request` polls on every injection. Unfortunately, this routine was designed for request-reply protocols, not efficient use of the network. It only injects messages on one side of the network, halving network capacity for our applications. Additionally, both sides of the network are polled, but one side will always be empty. The resulting performance is worse than that of **CMAML_rpc**.

The Strata send routine, however, was designed for efficient use of the network. This version polls on every injection and also uses both the left and right networks, which increases both throughput and network capacity. Strata also has slightly lower overheads than the CMAML versions. Strata reduces communications overhead in our application up to a factor of three relative to the **CMAML_rpc** version.

The change from CMAML to Strata increases *application* performance by 25 to 30 percent. The importance of polling and the resulting improvements are consistent with the results in [BK94]. Given the limited capacity of the network, it is critical to use both sides of the network, which provides twice as much buffering. Secondarily, it is critical to poll on every injection to avoid stalling other processors. Low-cost interrupts avoid the latter problem, but do not exist on the CM-5 and also would require additional overhead to ensure atomicity.

7 Overheads

In this section, we isolate the overheads in the DSC-strict runtime implementation. With only two multiply-accumulates per dependency, our application is extremely fine grained. Not only does this make both computation and communication overheads highly significant, it also makes the code extremely difficult to measure. Even with lightweight cycle accesses to the cycle counter through Strata, we had to keep timing to a minimum in order to avoid altering the behavior of our codes. As a result, we used a series of incrementally varying codes to isolate relevant overheads.

7.1 Local Overheads

We first focused upon local overheads by comparing the performance of two versions of our parallel code on a single CM-5 SPARC node with that of our optimized sequential solver on the same node. One version of the parallel code contained presence counters and checks, and the other had them removed.

Figure 5 contains the results of these experiments for each of our benchmark matrices. The difference between the parallel code without presence counters and the sequential code constitutes the overhead of using row-based datastructures with local and non-local outgoing dependency information. The difference ranges from 5 to 20 percent of the sequential time, but we expect the actual impact to be on the low end of the range. This is because the higher percentages arise from the cache behavior of the larger matrices. However, when the actual application is run in parallel, the matrices will be distributed across multiple processors, greatly reducing their local working sets.

The difference between the parallel code without presence counters and the code with counters represents the cost of maintaining presence information. This cost is about 30 percent of the sequential time.

With these figures, we expect our total local overhead to be about 40 percent of our sequential time. Given that our application efficiencies range from 15 to 40 percent, we are left with substantial non-local overheads to examine.

7.2 Non-Local Overheads

As we might expect, a portion of our efficiency is lost to idle time caused by critical paths in our schedule. However, we found that a major portion of our overheads comes from send and receive overhead.

Messages received by our application invoke a receive handler which adds the incoming data to the appropriate matrix row and increments two presence counters. Note that the data is multiplied with the appropriate coefficient on the sender's side before communication.

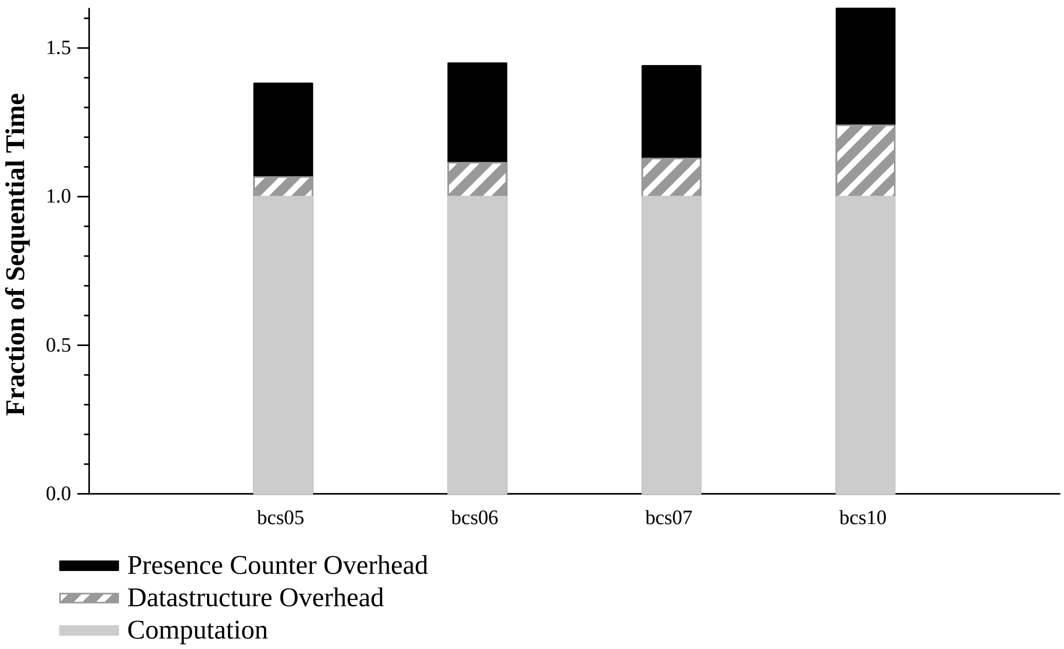

Figure 5: CM-5 single node performance of our parallel code is broken down for each of our benchmarks. Performance is given as a ratio to the single node performance of an optimized sequential implementation. "Computation" equals 1.0 by definition.

We measured the time for a message to be received and the resulting execution of its handler. We then subtracted the time to execute the handler to obtain the send overhead, which is 81 cycles. The Strata send routine that we use in our applications takes 66 cycles with an uncongested network. Adding our overheads together, this gives us a total *best-case* send and receive overhead of 147 cycles per message.

Now we can calculate the *minimum* overhead in our applications due to send and receive by counting the number of messages sent and received. Figure 6 gives our results for each of our applications. Note that the overhead is substantial and that the data is very consistent, with overheads increasing as matrix size increases. With overheads as nearly as high as 40 percent, it is clear that substantial performance improvements could result from architectures with decreased send and receive overhead.

8 Lower-Overhead Architectures

Although the CM-5 and active messages deliver much lower overheads than earlier systems such as the iPSC/860, send and receive overhead still limits performance on fine-grained applications. Newer systems promise to deliver much lower overheads which directly translates to much higher performance on our applications. The T3D [T3D93], for example, can send a message in 4 cycles and receive a message with similar overhead. The Alewife [ACJ+91] shared-memory multiprocessor has a memory controller which reduces processor overhead to similar levels.

The StarT [Bec93] architecture has an on-processor message interface and can send 256 bits in one clock cycle and can receive 128 bits in one cycle. The J-Machine [D+92] first demonstrated such processor-interface integration, but was not implemented to support the floating point operations necessary scientific applications such as our benchmarks.

With low-overhead architectures, we can expect send and receive overhead of only 3 cycles per 24-byte message in our application. With careful polling policies or inexpensive message interrupts, our applications should not be limited by the capacity of most modern networks. Therefore, a

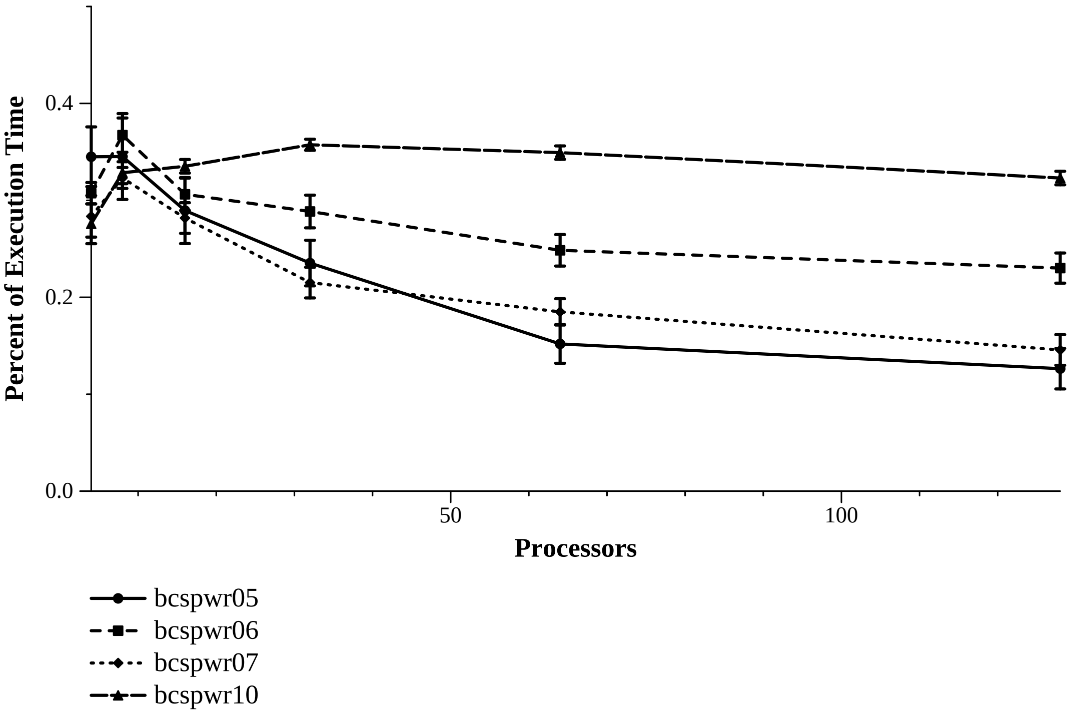

Figure 6: Communication overhead of the DSC-prescheduled-strict runtime.

reduction in communication overhead should directly translate to performance improvements in our applications. If the CM-5 had this kind of messaging support, our applications would run 15 to 40 percent faster.

9 Conclusion

Using both dynamic and run-time preprocessing methods, we have demonstrated unprecedented speedups for extremely fine-grained, irregular DAGs arising from triangular solves of realistic sparse matrices. While previous implementations have reported low efficiencies and bounded speedups of about 4, our solves exhibit performance that scales to speedups of almost 35 on 128 processors for matrices as small as 5300 rows.

In comparing our range of runtime methods, we find that the DSC algorithm produces the best performance. However, the small run length of our problems will make it difficult to implement such algorithms such that their costs are less than their benefits.

Our applications require a frequent message-polling policy to achieve good performance. This policy is not available from the standard CMMD library, but is provided by the Strata communications library.

Finally, our detailed study of overheads revealed that send and receive overheads are still a significant factor on the CM-5, constituting from 15 to 40 percent of our application times. We conclude that our applications would significantly benefit from architectural support for lower-overhead communications.

10 Acknowledgments

We would like to thank Joel Phillips and Jacob White for discussions on iterative methods; Mustafa Uysal for providing factored matrices; Tao Yang, Apostolos Gerasoulis, and Jia Jiao for providing and supporting the PYRROS code which performs DSC; and Stuart Fiske and Debby Wallach for their comments on drafts of this paper. We are also grateful to AHPCRC Minnesota, for providing access to their CM-5.

References

[ACJ+91] Anant Agarwal, David Chaiken, Kirk Johnson, David Kranz, John Kubiatowicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, and Dan Nussbaum. The MIT Alewife machine: a large-scale distributed-memory multiprocessor. MIT/LCS/TM 454, MIT, 545 Technology Sq., Cambridge, MA 02139, June 1991.

[ACM88] Arvind, David E. Culler, and Gino K. Maa. Assessing the benefits of fine-grained parallelism in dataflow programs. In *Supercomputing '88*. IEEE, 1988.

[AS92] Fernando Alvarado and Robert Schreiber. Optimal parallel solution of sparse triangular systems. *SIAM Journal of Scientific and Statistical Computation*, 1992. to appear.

[BB94] Eric A. Brewer and Robert D. Blumofe. Strata: A high-performance communications library. Technical Report (to appear). MIT Laboratory for Computer Science, January 1994.

[Bec93] Michael J. Beckerle. Overview of the START(*T) multithreaded computer. In *Proceedings of IEEE COMPCON*, February 1993.

[BK94] Eric A. Brewer and Bradley C. Kuszmaul. How to get good performance from the CM-5 data network. In *International Parallel Processing Symposium*, Cancun, Mexico, April 1994. To Appear.

[CY93] Soumen Chakrabarti and Katherine Yelick. Implementing an irregular application on a distributed memory architecture. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 169–178, San Diego, California, May 1993.

[D+92] William J. Dally et al. The message-driven processor: A multicomputer processing node with efficient mechanisms. *IEEE Micro*, pages 23–39, April 1992.

[DGL92] Ian S. Duff, Roger G. Grimes, and John G. Lewis. User's guide for the Harwell-Boeing sparse matrix collection. Technical Report TR/PA/92/86, CERFACS, 42 Ave G. Coriolis, 31057 Toulouse Cedex, France, October 1992.

[E+92] Thorsten von Eicken et al. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, Queensland, Australia, May 1992.

[ETM93] Analysis of performance accelerator running ETMSP. Technical Report TR-102856, Performance Processors, Inc., Palo Alto, California 94301, October 1993. Research Project 8010-31.

[GY92] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16:276–291, 1992.

[HR93] Michael T. Heath and Padma Raghavan. Distributed solution of sparse linear systems. Technical Report UIUCDCS-R-93-1793, Department of Computer Science, University of Illinois, Urbana, IL 61801, February 1993.

[L+92] Charles E. Leiserson et al. The network architecture of the connection machine CM-5. In *Symposium on Parallel Architectures and Algorithms*, pages 272–285, San Diego, California, June 1992. ACM.

[LBT87] Robert F. Lucas, Tom Blank, and Jerome J. Tiemann. A parallel solution method for large sparse systems of equations. *IEEE Transactions on Computer-Aided Design*, CAD-6(6):981–991, November 1987.

[Moy91] Steven A. Moyer. Performance of the iPSC/860 node architecture. Technical Report IPC-TR-91-007, Institute for Parallel Computation, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA 22903, May 1991.

[PA92] Alex Pothen and Fernando Alvarado. A fast rerdering algorithm for parallel sparse triangular solution. *SIAM Journal of Scientific and Statistical Computation*, 13:645–653, 1992.

[Rub92] Norman Rubin. Data flow computing and the conjugate gradient method. Technical Report MCRC-TR-25, Motorola Cambridge Research Center, 1992.

[Sal90] Joel H. Saltz. Aggregation methods for solving sparse triangular systems on multiprocessors. *SIAM Journal of Scientific and Statistical Computing*, 11(1):123–144, Jan 1990.

[Sar89] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. MIT Press, 1989.

[SMC91] Joel H. Saltz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Transactions on Computers*, pages 603–611, 1991.

[T3D93] CRAY T3D system architecture overview. Technical report, Cray Research, Inc., 900 Lowater Road, Chippewa Falls, WI 54729, September 1993.

[Thi93a] Thinking Machines Corporation, Cambridge, MA. *CM-5 Technical Summary*, November 1993.

[Thi93b] Thinking Machines Corporation, Cambridge, MA. *CMMD Reference Manual (Version 3.0)*, May 1993.

[YA93] Donald Yeung and Anant Agarwal. Experience with fine-grain synchronization in MIMD machines for preconditioned conjugate gradient. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, pages 187–197, San Diego, California, May 1993.

[Yan93] Tao Yang. Scheduling and code generation for parallel architectures. Technical Report DCS-TR-299, Rutgers, May 1993.