Ordering Schemes for Sparse Matrices using Modern Programming Paradigms

LEONID OLIKER, XIAOYE LI, PARRY HUSBANDS
Lawrence Berkeley National Laboratory
One Cyclotron Rd, Mail Stop 50B-2239
Berkeley, CA 94720 USA
{loliker,xsli,pjrhusbands}@lbl.gov

RUPAK BISWAS
NASA Ames Research Center
Mail Stop T27A-1
Moffett Field CA 94035 USA
rbiswas@nas.nasa.gov

Abstract

The Conjugate Gradient (CG) algorithm is perhaps the best-known iterative technique to solve sparse linear systems that are symmetric and positive definite. In previous work, we investigated the effects of various ordering and partitioning strategies on the performance of CG using different programming paradigms and architectures. This paper makes several extensions to our prior research. First, we present a hybrid (MPI+OpenMP) implementation of the CG algorithm on the IBM SP and show that the hybrid paradigm increases programming complexity with little performance gains compared to a pure MPI implementation. For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. We present MPI results for ILU(0) preconditioned CG (PCG) using the BlockSolve95 library, and show that the initial ordering of the input matrix dramatically affect PCG's performance. Finally, a multithreaded version of the PCG is developed on the Cray (Tera) MTA. Unlike the message-passing version, this implementation did not require the complexities of special orderings or graph dependency analysis. However, only limited scalability was achieved due to the lack of available thread level parallelism.

1 Introduction

The ability of computers to solve hitherto intractable problems and simulate complex processes using mathematical models makes them an indispensable part of modern science and engineering. Computer simulations of large-scale realistic applications usually require solving a set of non-linear partial differential equations (PDEs) over a finite region, subject to certain initial and boundary conditions. Structured grids are the most natural way to discretize such a computational domain since they are characterized by a uniform connectivity pattern. Their regular structure is also well suited for simple ordering techniques. Unfortunately, complicated domains must often be divided into multiple structured grids to be completely discretized, requiring a great deal of human intervention. Unstructured meshes, by contrast, can be generated automatically for applications with complex geometries or those with dynamically moving boundaries but at the cost of higher memory requirements. However, because such meshes are irregularly structured, sophisticated ordering schemes are required to achieve high performance on leading parallel systems. In this paper, we examine the relationship between the ordering of unstructured meshes, and the corresponding parallel performance of the underlying numerical solution.

The process of obtaining numerical solutions to the governing PDEs requires solving large sparse linear systems or eigen systems defined over the unstructured meshes that model the underlying physical objects. The Conjugate Gradient (CG) algorithm is perhaps the best-known iterative technique to solve sparse linear systems that are symmetric and positive definite. The CG algorithm is often used with a preconditioner for systems that are ill-conditioned. Within each iteration of preconditioned CG (PCG), the sparse matrix vector multiply (SPMV) and the inverse of the preconditioning matrix are usually the most expensive operations.

Modern computer architectures, based on deep memory hierarchies, show acceptable performance only if users care about the proper distribution and placement of irregularly structured data [1, 8]. Single-processor performance crucially depends on the exploitation of locality, and parallel performance degrades significantly if inadequate partitioning of data causes excessive communication and/or data migration. An intuitive approach would be to use a sophisticated partitioning algorithm, and then to post-process the resulting partitions with an enumeration strategy for enhanced locality. Although, in that sense, optimizations for partitioning and locality may be treated as separate problems, real applications tend to show a rather intricate interplay of both.

In previous work [9], we investigated the effects of various ordering and partitioning strategies on the performance of CG and SPMV using different programming paradigms and architectures. In particular, we used the reverse Cuthill-McKee [2] and the self-avoiding walks [4] ordering strategies, and the METIS [7] graph partitioner. This paper makes several extensions to our prior research. We present a hybrid (MPI+OpenMP) implementation of the CG algorithm on the IBM SP and show that the hybrid paradigm increases programming complexity with little performance gains compared to a pure MPI implementation. For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. We present MPI results for ILU(0) preconditioned CG (PCG) using the BlockSolve95 [6] library. BlockSolve95 graph colors and reorders the input matrix to achieve high parallelism; however, we found that the initial ordering of the input matrix dramatically affected PCG's performance. Finally, a multithreaded version of the PCG is developed on the Cray (Tera) MTA. This implementation was dramatically less complex than the BlockSolve95's PCG, which required advanced graph dependency analysis and matrix reordering. However, only limited scalability was achieved due to the lack of available thread level parallelism.

2 Preconditioned conjugate gradient

The Conjugate Gradient (CG) algorithm is the oldest and best-known Krylov subspace method used to solve the sparse symmetric positive definite linear system $Ax = b$. The convergence rate of CG depends on the spectral condition number of the coefficient matrix A. For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. In other words, the original system is transformed into another that has the same solution, but with better spectral properties.

Each iteration of the preconditioned CG (PCG) [10] involves one sparse matrix-vector product (SPMV), one solve with preconditioner, three vector updates (AXPY), and three inner products (DOT). For most practical matrices, the SPMV and solve dominate the other operations.

A preconditioner is any kind of modification to the original sparse linear system which makes it easier to solve. One broad class of effective preconditioners is based on *incomplete factorizations* of the matrix A. In this paper, we consider the simplest form of incomplete factorization, called ILU(0), where all the fill elements not at the nonzero positions of A are discarded. Compared with the other ILU variants, ILU(0) is computationally fast and memory efficient. It is quite effective for a reasonable number of practical matrices.

The ILU(0) method contains two steps. First, an incomplete LU factorization of A must be created. This factorization is performed only once, hence its cost can be amortized. Secondly, the lower and upper triangular solves with L and U are performed in each PCG iteration. The triangular solves incur about the same number of operations as the SPMV Ax, because the sparsity patterns of L and U are identical to the lower and upper triangular parts of A. However, a parallel triangular solve tends to be slower than a parallel SPMV, because it has a smaller degree of parallelism. For SPMV, all the components can be obtained independently in parallel. This is not true for a triangular solve. Figure 1 illustrates the lower triangular solve $Lx = b$. The solution of x_i depends on all x_j, $j < i$, unless $l_{ij} = 0$. Thus there are more task dependencies than SPMV, even if L is very sparse. The task dependency graphs change with the matrix ordering; hence, different orderings have different degrees of parallelism.

```
$x = b$
**for** $j = 1, n$
    $x_j = x_j / l_{jj}$
    **for each** $i > j$ and $l_{ij} \neq 0$
        $x_i = x_i - l_{ij} x_j$
**endfor**
```

Figure 1: The lower triangular solve.

3 Partitioning and Linearization

Almost all state-of-the-art computer architectures utilize some degree of memory hierarchy (registers, cache, main memory) that implies data locality is crucial. Various numberings of the mesh elements/vertices result in different nonzero patterns of matrix A which, in turn, cause different access patterns for the entries of the vector x. Moreover, on a distributed-memory machine, they imply different amounts of communication. Some excellent parallel graph partitioning algorithms have been developed and implemented in the last decade that are extremely fast while giving good load balance quality and low edge cuts. With graph partitioning, data locality is enforced by minimizing interprocessor communication, but not at the cache level. In this paper, we have used the popular METIS [7] multilevel partitioner for our experiments.

Linearization techniques play an important role in enhancing cache performance. Over the years, special numbering strategies have been developed to optimize memory usage and locality of sparse matrix computations. The particular enumeration of the vertices in an finite element method (FEM) discretization controls, to a large extent, the sparseness pattern of the resulting stiffness matrix. The bandwidth, or profile, of the matrix has a significant impact on the efficiency of linear systems and eigensolvers. We examine the performance of the reverse Cuthill-McKee (RCM) [2]. This algorithm is fairly straightforward to implement and largely benefits by operating on a pure graph structure, i.e., the underlying graph is not necessarily derived from a triangular mesh.

A Self-Avoiding Walk (SAW) [4] over a triangular mesh is an enumeration of the triangles such that two consecutive triangles (in the SAW) share an edge or a vertex, i.e. there are no jumps in the SAW. It can be shown that walks with more specialized properties exist over arbitrary unstructured meshes, and that there is an algorithm for their construction whose complexity is linear in the number of triangles in the mesh. Furthermore, SAWs are amenable to hierarchical coarsening and refinement, i.e. they have to be rebuilt only in regions where mesh adaptation occurs, and can therefore be easily parallelized. SAW, unlike RCM, is not a technique designed specifically for vertex enumeration; thus, it cannot operate on the bare graph structure of a triangular mesh. This implies a higher construction cost for SAWs, but several different vertex enumerations can be derived from a given SAW.

4 Experimental results

Our experimental test mesh consists of a two-dimensional Delaunay triangulation, generated by the Triangle [11] software package. The mesh is shaped like the letter "A", and contains 661,054 vertices and 1,313,099 triangles. The underlying matrix was assembled by assigning a random value in $(0, 1)$ to each (i, j) entry corresponding to the vertex pair (v_i, v_j), where $1 \leq distance(v_i, v_j) \leq 3$. All other off-diagonal entries were set to zero. This simulates a stencil computation where each vertex needs to communicate with its neighbors that are no more than three edge lengths away. The matrix is symmetric with its diagonal entries set to 40, which makes it diagonally dominant (and hence positive definite). This ensures that the CG algorithm converges successfully. The final sparse matrix A has approximately 39 entries per row and a total of 25,753,034 nonzeros. The CG algorithm converges in 13 iterations, with the unit vector as the right-hand side b and the zero vector as the initial guess for x. For the PCG experiments, the diagonal entries of the matrix were reduced to 10, thus no longer making it diagonally dominant and causing the original CG to fail. The PCG algorithm successfully converged in 10 iterations, given the modified matrix.

4.1 Hybrid CG results

The latest technological advances have allowed increasing numbers of processors to have access to a single memory space in a cost effective manner. As a result, the latest teraflops-scale parallel architectures contain a larger number of networked SMPs. Pure MPI codes should port easily to these systems, since message passing is required among the SMP nodes. However, it is not obvious that message passing within each SMP is the most effective use of the system. A recently proposed programming paradigm combines two layers of parallelism, by implementing OpenMP shared-memory codes within each SMP, while using MPI among the SMP clusters. This mixed programming strategy allows codes to potentially benefit from loop-level parallelism in addition to coarse-grained domain-level parallelism. Although the hybrid programming methodology may be the best mapping to the underlying architecture, it remains unclear whether the performance gains of this approach compensate for the increased programming complexity and the loss of portability.

The hybrid architecture used in our experiments is the IBM SP system, recently installed at the San Diego Supercomputing Center (SDSC). The machine contains 1,152 processors arranged as 144 SMP compute nodes. Each node is equipped with 4 GB of memory shared among its eight 222 MHz Power3 processors, and connected via a crossbar. The crossbar technology reduces bandwidth contention to main memory, compared to traditional shared-bus designs. Each Power3 CPU has an L1 (64 KB) cache which is 128-way set associative, and L2 (4 MB) cache which is four-way set associative with its own private cache bus. All the nodes are connected to each other via a switch interconnect using an omega-type topology. Currently, only four MPI tasks (out of the eight processors) are available within each SMP when using this fast switch. Thus, under the current configuration, the user is required to implement mixed mode programs to utilize all the processors. The next generation switch will alleviate this problem.

For the hybrid implementation of the CG algorithm on the IBM SP, we started with the Aztec MPI library [5] and incrementally added OpenMP parallelization directives. Through the use of profiling, the key loop nests responsible for significant portions of the overall execution were identified. A naive parallelization of all loops can be counterproductive since the overhead of OpenMP can exceed the savings in execution time. Some reorganization of the code, including the use of temporary variables, was necessary to preserve correctness. In all, eight Aztec loops were parallelized with OpenMP directives, the most important being the SPMV routine. To achieve the best possible OpenMP performance, dense vector operations were performed with the threaded vendor-optimized BLAS from the Engineering and Scientific Subroutine Library (ESSL).

Table 1 shows the results of the hybrid CG implementation on the SP, for varying numbers of SMP nodes, MPI tasks, and OpenMP threads. In addition to the ORIG, METIS, RCM, and SAW orderings, we present a new hybrid partitioning/linearization scheme comprised of METIS+SAW(M+S) . Since METIS [7] is well-suited for minimizing interprocessor communication and SAW [4] has been demonstrated to enhance cache locality, combining these two approaches is a potentially promising strategy for hybrid architectures. First, the graph is partitioned into the appropriate number of MPI tasks using METIS. Next, a SAW linearization is applied to each individual subdomain in parallel. Thus, when multiple OpenMP threads process their assigned submatrix, the SAW reordering should improve each processor's cache performance and reduce false sharing.

Notice that when there is only one SMP node and one MPI task, as in {1,1,4} and {1,1,8} (the tuple {x,y,z} denotes {SMP nodes, MPI tasks, OpenMP threads }), the CG code is effectively parallelized using only OpenMP; thus, timings are not presented for the corresponding METIS and METIS+SAW entries. Similarly, when the number of OpenMP threads is one, the parallelization is purely MPI based. Recall that

P	S	M	O	ORG	MET	RCM	SAW	M+S
4	1	1	4	6.47		3.56	3.24	
	1	2	2	6.84	4.96	3.29	3.01	2.99
	1	4	1	7.26	3.99	3.19	2.91	2.90
	2	1	2	6.92	4.80	3.25	2.96	2.92
	2	2	1	7.65	3.88	3.13	2.82	2.80
	4	1	1	7.27	3.87	3.10	2.80	2.77
8	1	1	8	4.38		2.16	1.99	
	1	2	4	4.99	2.92	1.99	1.87	1.84
	1	4	2	6.03	2.42	1.93	1.81	1.78
	2	1	4	4.85	2.76	1.85	1.71	1.67
	2	2	2	5.95	2.23	1.75	1.62	1.58
	2	4	1	6.14	1.89	1.75	1.59	1.57
	4	1	2	5.30	2.12	1.73	1.56	1.53
	4	2	1	6.04	1.80	1.68	1.50	1.49
	8	1	1	5.55	1.77	1.68	1.51	1.45
16	2	1	8	3.37	1.92	1.21	1.13	1.11
	2	2	4	4.12	1.36	1.07	1.01	0.99
	2	4	2	4.78	1.47	1.08	1.01	1.00
	4	1	4	3.68	1.26	0.98	0.92	0.89
	4	2	2	4.52	1.07	0.98	0.89	0.88
	4	4	1	5.18	0.96	0.98	0.91	0.90
	8	1	2	4.15	1.05	0.96	0.89	0.84
	8	2	1	4.53	0.90	0.92	0.84	0.82
32	4	1	8	2.97	0.87	0.65	0.67	0.61
	4	2	4	3.60	0.70	0.61	0.59	0.58
	4	4	2	4.06	0.72	1.12	0.68	0.64
	8	1	4	3.32	0.62	0.59	0.52	0.50
	8	2	2	3.80	0.58	0.59	0.56	0.54
	8	4	1	4.26	0.58	0.60	0.58	0.56
64	8	1	8	2.99	0.47	0.39	0.39	0.37
	8	2	4	3.55	0.45	0.69	0.44	0.40
	8	4	2	3.96	0.46	0.79	0.49	0.46

Table 1: Runtimes (in seconds) of CG using different orderings on the IBM SP with varying numbers of SMP nodes (S), MPI tasks (M), and OpenMP threads (O) .

due to limitations in the current switch architecture of the SDSC's SP, the maximum number of MPI tasks is limited to four on each SMP, and hybrid programming is required to use all the available processors.

The performance of the ordering schemes averaged across all combinations of nodes, tasks, and threads from best to worst are: METIS+SAW, SAW, RCM, METIS, and ORIG. The METIS+SAW strategy consistently outperforms all others; however as was shown in our previous work [9], cache behavior is significantly more important than interprocessor communication for our application. As a result, there is no significant performance difference between the hybrid METIS+SAW strategy and the pure SAW linearization. Nonetheless, we expect algorithms with higher communication requirements to benefit from this dual partitioning/ordering approach. This will be the subject of future research. Overall, these results show that intelligent ordering schemes are extremely important for efficient sparse matrix computations regardless of whether the programming paradigm is OpenMP, MPI, or a combination of both.

To compare hybrid versus pure MPI performance, first examine the METIS+SAW column since it gives the best CG runtimes. Each processor set shows differing results. For example, on 16 processors, the fastest CG implementation is for {8,2,1}, meaning no OpenMP parallelization is triggered. However, on 32 processors, {8,1,4} is the fastest, outperforming

	ORIG		METIS	
P	TriSolve	PCG	TriSolve	PCG
8	14.08	51.87	9.96	13.17
16	8.01	32.96	3.87	5.32
32	5.98	8.83	1.80	2.56
64	6.12	8.30	0.87	1.28

	RCM		SAW	
P	TriSolve	PCG	TriSolve	PCG
8	5.98	8.61	4.86	6.87
16	2.86	4.23	2.74	4.02
32	1.74	2.58	1.31	1.99
64	0.78	1.25	0.81	1.17

Table 2: Runtimes (in seconds) for the triangular solve and the overall PCG using different orderings on the SP.

{8,4,1}. Finally, on 64 processors, using the maximum number of OpenMP threads, as in {8,1,8}, gives the best results. Within each processor set, varying the number of tasks and threads does not result in a significant performance difference. Overall, the hybrid implementation offers no noticeable advantage. This is true for the other ordering schemes as well, as is evident from Table 1. However, since the hybrid paradigm increases programming complexity and adversely affects portability, we conclude that for running iterative sparse solvers on clusters of SMPs, a pure MPI implementation is a more effective strategy. A hybrid PCG implementation is not considered in this paper, and will be the subject of future work.

4.2 Message-passing PCG results

Parallel programming with message passing is the most common and mature approach for high-performance parallel systems. The message-passing PCG experiments in this paper use the BlockSolve95 [6] software library, which is used for solving large, sparse linear systems on parallel platforms that support message-passing with MPI. Although Aztec is a powerful iterative library, it does not provide a global ILU(0) factorization routine. For these experiments we used the IBM SP located at NERSC, which consist of two 200 MHz Power3 processors per SMP node and is otherwise very similar to SDSC's SP described in Section 4.1. BlockSolve95 uses two matrix reordering schemes to achieve scalable performance. First, the graph is reduced by extracting cliques and identical nodes (i-nodes) in the sparse matrix structure, allowing for the use of higher-level BLAS. Next, the reduced graph is colored using an efficient parallel coloring heuristic. Finally, vertices of the same color are grouped and ordered sequentially. As a result, during the triangular solves of the PCG, the unknowns corresponding to these vertices can be solved for in parallel, after the updates from previous color groups have been performed. The number of colors in the graph therefore determines the number of parallel steps in the triangular solve. Since BlockSolve95 reorders the input matrix, we investigate what effect, if any, our ordering strategies have on the parallel performance of PCG.

Table 2 presents the runtimes of the triangular

	ORIG		METIS	
P	Color	Factorize	Color	Factorize
8	116.68	339.94	48.41	107.20
16	75.63	283.71	20.00	46.90
32	46.96	128.30	10.01	23.26
64	28.08	82.63	5.01	11.41

	RCM		SAW	
P	Color	Factorize	Color	Factorize
8	37.87	82.53	33.93	75.31
16	19.01	40.02	17.05	37.22
32	9.59	20.19	8.76	19.79
64	5.39	10.48	4.64	9.57

Table 3: Runtimes (in seconds) for BlockSolve95 graph coloring and matrix factorization using different orderings on the SP

solve and the total PCG using various ordering strategies. Results clearly show that the initial ordering of the matrix plays a significant role in PCG performance, even though the input matrix is further reordered by the BlockSolve95 library. Notice that the triangular solve procedure is responsible for the majority of PCG's computational overhead, and is also sensitive to the initial ordering. For the overall PCG runtime, SAW has a slight advantage over RCM and METIS; however, all three ordering schemes are about an order of magnitude faster than ORIG.

The BlockSolve95 graph coloring and ILU(0) matrix factorization times are presented in Table 3. The initial ordering of the matrix dramatically affects both these pre-processing steps, with SAW producing the best results. Notice from Tables 2 and 3 that the BlockSolve95 library shows scalable performance across all aspects of the PCG computation when intelligent ordering schemes are used.

4.3 Multithreaded PCG results

Multithreading has received considerable attention over the years as a promising way to hide memory latency in high-performance computers, while providing access to a large and uniform shared memory. Using multithreading to build commercial parallel computers is a new concept in contrast to the standard single-threaded microprocessors of traditional supercomputers. Such machines can potentially utilize substantially more of its processing power by tolerating memory latency and using low-level synchronization directives. Cray (formally Tera) has designed and built a state-of-the-art multithreaded computer called the MTA, which is especially well-suited for irregular and dynamic applications. Parallel programmability is considerably simplified since the user has a global view of the memory, and need not be concerned with the data layout.

The Cray MTA is a supercomputer installed about two years ago at SDSC. The MTA has a radically different architecture than current high-performance computer systems. Each 255 MHz processor has support for 128 hardware streams, where each stream includes a program counter and a set of 32 registers. One program thread can be assigned to each stream. The processor switches with no overhead among the active streams at every clock tick even if a thread is not blocked, while executing a pipelined instruction. Previous work [9] presented the CG and SPMV implementation on the MTA and showed that special ordering or partitioning schemes are not required to obtain high efficiency and scalability.

P	Trisolve	PCG
1	71.98	80.34
2	45.74	50.02
4	26.94	29.18
8	16.04	17.29

Table 4: Runtimes (in seconds) for the triangular solve and the overall PCG on the MTA

For the MTA implementation of PCG, we developed a multithreaded version of the lower and upper triangular solves (see Figure 1.) Matrix factorization times are not reported since it is performed only once outside the inner loop. Our multithreaded strategy uses low-level locks to effectively perform an on-the-fly dependency analysis. Recall that to compute the lower triangular solve $Lx = b$, the solution of x_i depends on all x_j , $j < i$, unless $l_{ij} = 0$. First, synchronization locks are applied to all x_j, $j = 1, 2, \ldots n$, to guarantee correct dependency behavior. Threads are then dynamically assigned to solve for each x_i. If a given x_i has a dependency on x_j which has not yet been computed, the attempt to access the blocked memory address of x_j will cause the thread responsible for processing x_i to be temporarily put to sleep. Once a thread successfully solves for x_j, the synchronization lock on that variable is released, causing the runtime system to wake all blocked threads waiting to access the memory address of x_j. Subsequent attempts to access that variable will no longer cause active threads to become blocked. The lightweight synchronization of the MTA allows locks to be effectively used at such a fine granularity. Notice that the multithreaded version of triangular solve is dramatically less complex than the BlockSolve95 implementation described in Section 4.2, which required advanced graph dependency analysis and matrix reordering to achieve high parallelism.

Table 4 presents the performance of PCG with the ORIG ordering on the MTA using 60 streams. Observe that the triangular solve is responsible for most of the computational overhead, and achieved a speedup of approximately 4.5X on eight processors. This limited scalability is due to the lack of available thread level parallelism in our dynamic dependency scheme. A large fraction of the computational threads were blocked at any given time, preventing a full saturation of the MTA processors. Subsequent attempts to optimize the multithreaded code by increasing the number of streams and using more sophisticated orderings strategies caused the MTA to crash due to limitations in its current system software. We plan to revisit the multithreaded PCG once a more mature runtime system becomes available on the MTA.

5 Summary and conclusions

In this paper, we examined the performance of and the programming effort required for the Conjugate Gradient (CG) sparse iterative solver using three leading programming paradigms. A recently proposed hybrid programming paradigm combines two layers of parallelism, by implementing OpenMP shared-memory codes within an SMP, while using MPI among the SMP clusters. We developed the CG algorithm on the IBM SP, by starting with the Aztec [5] MPI library and incrementally adding OpenMP parallelization directives. A new hybrid partitioning/linearization scheme comprised of METIS+SAW was presented, and consistently outperformed the other ordering schemes. Comparing hybrid (MPI+OpenMP) versus pure MPI implementations of CG, we found no significant performance differences between the two schemes. However, since the hybrid paradigm increases programming complexity and adversely affects portability, we conclude that for running iterative solvers on clusters of SMPs, a pure MPI implementation is a more effective strategy.

For ill-conditioned linear systems, it is often necessary to use a preconditioning technique. We presented MPI results for ILU(0) preconditioned CG (PCG) using the BlockSolve95 [6] library. Unlike CG, the runtime of the PCG algorithm is dominated by the triangular solves which are inherently less amenable to parallelization than SPMV. BlockSolve95 graph colors and reorders the input matrix to achieve high parallelism; however, we found that the initial ordering of the input matrix dramatically affected PCG's performance. Overall, the SAW linearization resulted in the best runtimes for all components of PCG, including graph coloring and factorization.

Multithreading has received considerable attention over the years as a promising way to hide memory latency in high-performance computers, while providing access to a large and uniform shared memory. A multithreaded version of the PCG algorithm was developed on the Cray MTA. Here, the triangular solve uses low-level locks to effectively perform a graph dependency analysis at runtime. This implementation was dramatically less complex than the BlockSolve95's PCG, which required advanced graph dependency analysis and matrix reordering. However, only limited scalability was achieved due to the lack of available thread level parallelism in our dynamic dependency scheme, which prevented a full saturation of the MTA processors. In future, we plan to revisit the MTA as a more mature runtime environment becomes available and as more processors are added to the system.

Acknowledgments

The work of the first three authors was supported by the Director, Office of Computational and Technology Research, Division of Mathematical, Information, and Computational Sciences of the U.S. Department of Energy under contract number DE-AC03-76SF00098.

References

[1] D. A. Burgess and M. B. Giles, Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines, *Advances in Engineering Software*, 28 (1997), pp. 189–201.

[2] E. Cuthill and J. McKee, Reducing the bandwidth of sparse symmetric matrices, *Proc. ACM National Conference,* 1969, pp. 157–192.

[3] A. George, Computer implementation of the finite element method, *Stanford University Technical Report STAN-CS-208,* Stanford, CA, 1971.

[4] G. Heber, R. Biswas, and G. R. Gao, Self-avoiding walks over adaptive unstructured grids, *Concurrency: Practice and Experience,* 12 (2000), pp. 85–109.

[5] S. A. Hutchinson, L. V. Prevost, J. N. Shadid, and R. S. Tuminaro, Aztec User's Guide, *Sandia National Laboratories Technical Report SAND95-1559,* Albuquerque, NM, 1998.

[6] M. T. Jones and P. E. Plassmann, BlockSolve95 User's Manual: Scalable Library Software for the Parallel Solution of Sparse Linear Systems, *Argonne National Laboratory Technical Report ANL-95/48,* Chicago, IL, 1995.

[7] G. Karypis and V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Statist. Comput.,* 20 (1998), pp. 359–392.

[8] R. Löhner, Renumbering strategies for unstructured-grid solvers operating on shared-memory, cache-based parallel machines, *Computer Methods in Applied Mechanics and Engineering,* 163 (1998), pp. 95–109.

[9] L.Oliker, X. Li, G. Heber, and R. Biswas Parallel Conjugate Gradient: Effects of Ordering Strategies, Programming Paradigms, and Architectural Platforms *13th International Conference on Parallel and Distributed Computing Systems,* (2000).

[10] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, 1996.

[11] J. R. Shewchuk, Triangle: Engineering a 2d quality mesh generator and Delaunay triangulator, *Applied Computational Geometry: Towards Geometric Engineering, Lecture Notes in Computer Science,* Vol. 1148, Springer-Verlag, Heidelberg, Germany, 1996, pp. 203–222.