for CFAI

Parallel Conjugate Gradient: Effects of Ordering Strategies, Programming Paradigms, and Architectural Platforms

LEONID OLIKER AND XIAOYE LI

NERSC, MS 50F, Lawrence Berkeley National Laboratory. Berkeley, CA 94720

GERD HEBER

Cornell Theory Center. 638 Rhodes Hall, Cornell University, Ithaca, NY 14853

RUPAK BISWAS

MRJ Technology Solutions, MS T27A-1, NASA Ames Research Center, Moffett Field, CA 94035

Abstract

The Conjugate Gradient (CG) algorithm is perhaps the best-known iterative technique to solve sparse linear systems that are symmetric and positive definite. A sparse matrix-vector multiply (SPMV) usually accounts for most of the floating-point operations within a CG iteration. In this paper, we investigate the effects of various ordering and partitioning strategies on the performance of parallel CG and SPMV using different programming paradigms and architectures. Results show that for this class of applications, ordering significantly improves overall performance, that cache reuse may be more important than reducing communication, and that it is possible to achieve message passing performance using shared memory constructs through careful data ordering and distribution. However, a multithreaded implementation of CG on the Tera MTA does not require special ordering or partitioning to obtain high efficiency and scalability.

Keywords: Ordering algorithms, unstructured meshes, sparse matrices, distributed and shared memory, multithreading

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. Structured grids are the most natural

way to discretize such a computational domain; however, 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 storage requirements to explicitly store the connectivity information for every point in the mesh). They also facilitate dynamic grid adaptation to efficiently solve problems with evolving physical features such as shock waves, vortices, detonations, shear layers, and crack propagation.

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. Within each iteration of CG, the sparse matrix vector multiply (SPMV) is usually the most expensive operation.

On uniprocessor machines, numerical solutions of such complex, real-life problems can be extremely time consuming, a fact driving the development of increasingly powerful parallel multiprocessor supercomputers. The unstructured, dynamic nature of many systems worth simulating, however, makes their efficient parallel implementation a daunting task. This is primarily due to the load imbalance created by the dynamically changing nonuniform grids and the irregular data access patterns. These cause significant communication at runtime, leaving many processors idle and adversely affecting the total execution time.

Furthermore, modern computer architectures, based on deep memory hierarchies, show acceptable performance only if users care about the proper distribution and placement of their data [2]. 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. The traditional 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 this paper, we investigate the effects of various ordering and partitioning strategies on the performance of CG and SPMV using different programming paradigms and architectures. In particular, we use the reverse Cuthill-McKee [3] and the self-avoiding walks [5] ordering strategies, and the METIS [7] partitioner. We examine parallel implementations of CG using MPI, shared-memory compiler directives, and multithreading, on three state-of-the-art parallel supercomputers: a Cray

T3E, an SGI Origin2000, and a Tera MTA. Results show that ordering improves performance significantly, and that cache reuse can be more important than reducing communication. However, the multithreaded implementation does not require any special ordering or partitioning to obtain high efficiency and scalability.

2 Partitioning and Linearization

Space-filling curves have been demonstrated to be an elegant and unified linearization approach for certain problems in N-body and FEM simulations, mesh partitioning, and other graph-related areas [4, 8, 9, 10, 12]. The linearization of a higher-dimensional spatial structure, i.e. its mapping onto a one-dimensional structure, is exploited in two ways: First, the locality preserving nature of the construction fits elegantly into a given memory hierarchy, and second, the partitioning of a contiguous linear object is trivial. For our experiments, we pursued both strategies with some modifications. In the following, we briefly describe the two classes of enumeration techniques and the general-purpose graph partitioner which were used.

2.1 Cuthill-McKee Algorithms (CM)

The particular enumeration of the vertices in a 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. Cuthill and McKee [3] suggested a simple algorithm based on ideas from graph theory. Starting from a vertex of minimal degree, levels of increasing distance from that vertex are first constructed. The enumeration is then performed level-by-level with increasing vertex degree (within each level). Several variations of this method have been suggested, the most popular being reverse Cuthill-McKee (RCM) where the level construction is restarted from a vertex of minimal degree in the final level. In many cases, it has been shown that RCM improves the profile of the resulting matrix. The class of CM algorithms are fairly straightforward to implement and largely benefit by operating on a pure graph structure, i.e. the underlying graph is not necessarily derived from a triangular mesh.

2.2 Self-Avoiding Walks (SAW)

These were proposed recently [5] as a mesh-based (as opposed to geometry-based) technique with similar application areas as space-filling curves. A SAW 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 CM, 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.

2.3 Graph Partitioning (e.g. METIS)

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. Perhaps the most popular is METIS [7] that belongs to the class of multilevel partitioners. METIS reduces the size of the graph by collapsing vertices and edges using a heavy edge matching scheme, applies a greedy graph growing algorithm for partitioning the coarsest graph, and then uncoarsens it back using a combination of boundary greedy and Kernighan-Lin refinement to construct a partitioning for the original graph. Partitioners strive to balance the computational workload among processors while reducing interprocessor communication. Improving cache performance is not a typical objective of most partitioning algorithms.

3 Sparse Matrix-Vector Multiplication and Conjugate Gradient

Sparse matrix-vector multiplication (SPMV) is one of the most heavily-used kernels in large-scale numerical simulations. To perform a SPMV, $y \leftarrow Ax$, we assume that the nonzeros of matrix A are stored in the Compressed Row Storage (CRS) format [1]. The dense vector x is stored sequentially in memory with unit stride. Various numberings of the mesh elements/vertices result in different nonzero patterns of A, which in turn cause different access patterns for the entries of x. Moreover, on a distributed-memory machine, they imply different amounts of communication.

The Conjugate Gradient (CG) algorithm is the oldest and best-known Krylov subspace method used to solve the linear system $Ax = b$. The method starts from an initial guess of the vector x. It then successively generates approximate solutions in the Krylov subspace, and search directions

Compute $r_0 = p_0 = b - Ax_0$ for some initial guess x_0

for $j = 0, 1, \ldots,$ until convergence

$\alpha_j = (r_j, r_j)/(Ap_j, p_j)$

$x_{j+1} = x_j + \alpha_j p_j$

$r_{j+1} = r_j - \alpha_j Ap_j$

$\beta_j = (r_{j+1}, r_{j-1})/(r_j, r_j)$

$p_{j+1} = r_{j+1} + \beta_j p_j$

endfor

Figure 1: The Conjugate Gradient algorithm.

used in updating the approximate solution and residual. The algorithm [11] is outlined in Figure 1. Each iteration of CG involves one SPMV for Ap_j, three vector updates (AXPY) for x_{j+1}, r_{j+1}, and p_{j+1}, and three inner products (DOT) for the update scalars α_j and β_j which make the generated sequences satisfy certain orthogonality conditions. For a symmetric and positive definite linear system, these conditions imply that the distance between the approximate solution and the true solution is minimized.

Suppose the matrix A is of order n and has nnz nonzeros. Then, one SPMV involves $O(nnz)$ floating-point operations, while AXPY and DOT involve only $O(n)$ floating-point operations. Thus, for many practical matrices, SPMV dominates the other two operations. This is demonstrated by the results given in Section 4. Note that both AXPY and DOT are insensitive to mesh orderings.

4 Experimental Results

Our experimental test mesh consists of a two-dimensional Delaunay triangulation, generated by the Triangle [13] 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. This sparsity pattern is representative of the matrices obtained from discretizing PDEs on three-dimensional meshes. 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 our test matrix, the SPMV computation accounts for approximately 87% of the total number of floating-point operations within each CG iteration.

4.1 Distributed-Memory Implementation

In our experiments, we use the parallel SPMV and CG routines in Aztec [6], implemented using MPI. The matrix A is partitioned into blocks of rows, with each block assigned to one processor. The associated components of vectors x and b are distributed accordingly. Communication may be needed to transfer some components of x. For example, in $y \leftarrow Ax$, if y_i is updated on processor p_1, $A_{ij} \neq 0$, and x_j is owned by processor p_2, then p_2 must send x_j to p_1. In general, a processor may need more than one x-component from another processor. It is thus more efficient to combine several x-components into one message so that each processor sends no more than one message to another processor. This type of optimization can be performed in a pre-processing phase. The other two operations, AXPY and DOT in the CG algorithm, are easily parallelized — AXPY requires only local computations, whereas DOT requires a local sum followed by a global sum reduction.

Three routines within Aztec are of particular interest to us: `AZ_transform`, which initializes the data structures and the communication schedule for SPMV, `AZ_matvec_mult`, which performs the matrix-vector multiply, and `AZ_cg`, which solves a linear system using the CG algorithm. In Table 1, we report the runtimes of the `AZ_matvec_mult` and `AZ_cg` routines on the Cray T3E at NERSC. Each T3E node consists of a 450 MHz DEC Alpha processor (900 Mflops peak theoretical floating-point speed), 96 KB secondary cache, and is interconnected to other nodes through a 3D

	ORIG		METIS		RCM		SAW	
P	SPMV	CG	SPMV	CG	SPMV	CG	SPMV	CG
8	0.5622	8.6519	0.4758	7.6617	0.3812	6.1853	0.1708	2.9158
16	0.3252	5.0929	0.2682	2.9092	0.1927	3.1979	0.0861	1.4912
32	0.1990	3.1667	0.0870	1.4677	0.0951	1.6615	0.0442	0.7948
64	0.1191	1.9287	0.0559	0.9614	0.0451	0.8816	0.0283	0.4616

Table 1: Runtimes (in seconds) of `AZ_matvec_mult` (SPMV) and `AZ_cg` (CG) using different orderings on the Cray T3E.

torus. It was not possible to run our test problem on less than 8 processors of the T3E due to memory constraints.

For the key kernel routine `AZ_matvec_mult`, SAW is always about twice as fast as RCM. In turn, RCM is about 1.5 times faster than METIS on 16 or fewer processors, and about the same on 32 or more processors. Note that when using 32 or more processors, METIS is twice as fast as ORIG (the natural ordering from Triangle). For `AZ_cg`, SAW is again about twice as fast as RCM. However, we do not see a clear advantage of RCM over METIS for this routine. Both RCM and METIS are twice as fast as ORIG on large number of processors. Finally, METIS, RCM, and SAW, all demonstrate excellent scalability (more than 75% efficiency) up to the 64 processors that were used for these experiments, but ORIG seems less scalable (only about 56% efficiency). As expected, there is a strong correlation between the performance of CG and the underlying SPMV for all test cases.

Table 2 shows the pre-processing times spent in `AZ_transform`. The times for METIS, RCM, and SAW are comparable, and are usually an order of magnitude larger than the corresponding times for `AZ_matvec_mult`. The `AZ_transform` times show some scalability up to 32 processors. However, for ORIG, the times are two to three orders of magnitude larger, and show very little scalability. Clearly, the ORIG ordering is too inefficient and unacceptable on distributed-memory machines.

P	ORIG	METIS	RCM	SAW
8	504.2407	2.8290	2.3703	2.0227
16	547.9423	1.4553	1.3299	1.1568
32	333.6689	0.8403	0.8640	0.8041
64	150.0072	0.4224	0.7763	0.5368

Table 2: Runtimes (in seconds) of `AZ_transform` using different orderings on the Cray T3E.

To better understand the various partitioning and ordering algorithms, we have built a simple performance model to predict the parallel runtime of `AZ_matvec_mult`. First, using the T3E's hardware performance monitor, we collected the average number of cache misses per processor. This is reported in Table 3, and shows that SAW has the fewest number of cache misses. In comparison, RCM, METIS, and ORIG have between two and three times that number. Second, we gathered statistics on the average communication volume and the maximum number of messages per processor, both of which are also shown in Table 3. Notice that METIS transfers the least

P	Avg. Cache Misses (10^6)				Avg. Communication (10^6 bytes)				Max. Message Count			
	ORIG	METIS	RCM	SAW	ORIG	METIS	RCM	SAW	ORIG	METIS	RCM	SAW
8	3.6842	3.0340	3.7490	2.0042	3.2275	0.0107	0.0308	0.0488	7	3	2	6
16	2.0072	1.3305	1.9049	0.9706	2.3643	0.0108	0.0315	0.0362	15	4	2	9
32	1.0597	0.6576	1.0172	0.5073	1.4918	0.0092	0.0316	0.0302	31	5	2	11
64	0.6011	0.3581	0.5150	0.2900	0.8285	0.0079	0.0316	0.0229	63	6	2	16

Table 3: Locality and communication statistics for `AZ_matvec_mult`.

amount of data, whereas RCM has the fewest number of messages.

In our model, we estimate the total parallel runtime T as

$$T = T_f + T_m + T_c \ ,$$

where, T_f, T_m, and T_c are the estimated per-processor times to perform floating-point operations, to service the cache misses, and to communicate the x vector. Given that a floating-point operation requires 1/900 microseconds and that each cache miss latency is 0.08 microseconds (both from product documentation), and assuming that the MPI bandwidth and latency are 50 MB/second and 10 microseconds (both from measurement), respectively, we can estimate the total runtime based on the information in Table 3.

P	ORIG			METIS		
	T (deviation)	T_m/T	T_c/T	T (deviation)	T_m/T	T_c/T
8	0.3666 (-35%)	0.80	0.18	0.2501 (-47%)	0.97	0.00
16	0.2117 (-35%)	0.76	0.22	0.1103 (-58%)	0.96	0.00
32	0.1170 (-41%)	0.72	0.26	0.0547 (-37%)	0.96	0.01
64	0.0667 (-44%)	0.72	0.27	0.0298 (-46%)	0.96	0.01
P	RCM			SAW		
	T (deviation)	T_m/T	T_c/T	T (deviation)	T_m/T	T_c/T
8	0.3077 (-19%)	0.97	0.00	0.1686 (-1%)	0.95	0.01
16	0.1566 (-19%)	0.97	0.00	0.0821 (-5%)	0.94	0.01
32	0.0838 (-12%)	0.97	0.01	0.0432 (-2%)	0.94	0.02
64	0.0428 (-5%)	0.96	0.02	0.0248 (-12%)	0.93	0.03

Table 4: Predicted runtimes for `AZ_matvec_mult` on the T3E. In the column of total time T, the percentage deviation from the measured time is given in parenthesis.

Table 4 shows the predicted total time T, and the ratios T_m/T and T_c/T. In parenthesis, we also give the percentage deviation of the predicted time from the measured runtime (that are reported in Table 1). The maximum deviation from the measured runtimes is -58%, which gives us a fair degree of confidence in our model. The results in Table 4 clearly indicate that servicing the cache misses is extremely expensive and requires more than 93% of the total time for METIS, RCM, and SAW, and 72–80% for ORIG (which has relatively more communication). Although SAW and RCM both incur more communication than METIS (in terms of the average message volume as shown in Table 3), their total runtimes are significantly less. This illustrates that for our combination of applications and architectures, improving cache reuse can be more important than reducing interprocessor communication.

4.2 Shared-Memory Implementation

The shared-memory version of CG was implemented on the Origin2000, which is a SMP cluster of nodes each containing two 250 MHz MIPS R10000 processors and local memory. The hardware makes all memory equally accessible from a software standpoint, by sending memory requests through routers located on the nodes. Access time to memory is nonuniform, depending on how far away the memory lies from the processor. The topology of the interconnection network is a hypercube, bounding the maximum number of memory hops to a logarithmic function of the number of processors. Each processor also has a relatively large 4 MB secondary cache, where only it can fetch and store data. If a processor refers to data that is not in cache, there is a delay while a copy of the data is fetched from memory. When a processor modifies a word of data, all other copies of the cache line containing that word are invalidated.

This version of the parallel CG code was written using SGI's native pragma directives, which create IRIX threads. A rewrite to OpenMP would require minimal programming effort but has not been done at this time. Each processor is assigned an equal number of rows in the matrix. The parallel SPMV and AXPY routines do not require explicit synchronizations, since they do not contain concurrent writes. Global reduction operations are required for DOT and the convergence tests. Two basic implementation approaches described below were taken.

The SHMEM strategy naively assumes that the Origin2000 is a flat shared-memory machine. Arrays are not explicitly distributed among the processors, and nonlocal data requests are handled by the cache coherent hardware. Alternatively, the CC-NUMA strategy addresses the underlying distributed-memory nature of the machine by performing an initial data distribution. Sections of the

sparse matrix are appropriately mapped onto the memories of their corresponding processors using the default "first touch" data distribution policy of the Origin2000. The computational kernels of both the SHMEM and CC-NUMA implementations are identical, and simpler to implement than the MPI version. Table 5 shows the SPMV and CG runtimes using both approaches with the ORIG, RCM, and SAW orderings of the mesh. We also present the runtime of CG using an MPI implementation on the Origin2000 with the SAW ordering, as a basis for comparison.

	SHMEM					
	ORIG		RCM		SAW	
P	SPMV	CG	SPMV	CG	SPMV	CG
1	2.224	46.911	1.489	37.183	1.460	36.791
2	1.249	28.055	0.852	21.867	0.831	21.772
4	1.425	30.637	0.935	25.350	0.915	24.751
8	0.922	16.836	0.572	14.431	0.572	14.121
16	1.047	16.348	0.635	15.516	0.645	15.548
32	1.072	16.653	0.664	15.350	0.641	15.423
64	0.747	10.809	0.323	7.782	0.324	8.450

	CC-NUMA						MPI
	ORIG		RCM		SAW		SAW
P	SPMV	CG	SPMV	CG	SPMV	CG	CG
1	2.224	46.911	1.489	37.183	1.460	36.791	
2	1.218	27.053	0.851	21.454	0.829	21.229	23.145
4	0.879	17.608	0.421	10.651	0.410	10.593	7.880
8	0.535	9.824	0.220	5.575	0.216	5.516	3.815
16	0.326	6.205	0.115	2.845	0.113	2.872	1.926
32	0.197	3.584	0.061	1.548	0.060	1.514	1.075
64	0.118	2.365	0.028	0.885	0.026	0.848	0.905

Table 5: Runtimes (in seconds) for different orderings running in SHMEM and CC-NUMA modes on the SGI Origin2000. The CG runtimes for an MPI implementation on the Origin2000 with the SAW ordering is also given for comparison.

Notice that the CC-NUMA implementation shows significant performance gains over SHMEM. This is expected since the Origin2000 is a distributed-memory system, and therefore should be treated as such. As the number of processors increases, the runtime difference between the two

approaches becomes more dramatic, achieving an order of magnitude improvement when using more than 16 processors. Proper data distribution becomes increasingly important for larger numbers of processors since the corresponding communication overhead grows nonuniformly. Within the CC-NUMA approach, the RCM and SAW ordering schemes dramatically reduce the runtimes compared to ORIG, indicating that an intelligent ordering algorithm is necessary to achieve good performance and scalability on distributed shared-memory systems. There is little difference in parallel performance between RCM and SAW because both ordering techniques reduce the number of secondary cache misses and the non-local memory references of the processors. Recall however that on the T3E, SAW was about twice as fast as RCM. This discrepancy in performance is probably due to the larger cache size of the Origin2000 that reduces the beneficial effects of smart ordering.

The last two columns of Table 5 compare the CC-NUMA and MPI implementations of CG on the Origin2000 using the SAW ordering. Notice that the runtimes are very similar, even though the programming methodologies of these two approaches are quite different. These results indicate that for this class of applications, it is possible to achieve message passing performance using shared memory constructs, through careful data ordering and distribution.

4.3 Multithreaded Implementation

The Tera MTA is a supercomputer recently installed at the San Diego Supercomputing Center (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 among the active streams at every clock tick, while executing a pipelined instruction.

The uniform shared memory of the MTA is flat, and physically distributed across hundreds of banks that are connected through a 3D toroidal network to the processors. All memory addresses are hashed by the hardware so that apparently adjacent words are actually distributed across different memory banks. Because of the hashing scheme, it is impossible for the programmer to control data placement. This enhances programmability compared to standard cache-based multiprocessor systems. Rather than using data caches to hide latency, the MTA processors use multithreading to tolerate latency. If a thread is waiting for its memory reference to complete, the processor executes instructions from other threads. Performance thus depends on having a large number of concurrent computation threads.

Lightweight synchronization among threads is provided by the memory itself. Each word of physical memory contains a full-empty bit, which enables fast synchronization via load and store instructions without operating system intervention. Synchronization among threads may stall one of the threads, but not the processor on which the threads are running, since each processor may run many threads. Explicit load balancing across loops is also not required since the dynamic scheduling of work to threads provides the ability of keeping the processors saturated, even if different iterations require varying amounts of time to complete. Once a code has been written in the multithreaded model, no additional work is required to run it on multiple processors, since there is no difference between uni- and multiprocessor parallelism.

The multithreaded implementation of CG is trivial, requiring only MTA compiler directives. Since the data structures are dynamically allocated pointers, special pragma assertions were used to indicate that there are no loop-carried dependencies. The compiler was thus able to automatically parallelize the appropriate loop segments. Load balancing is implicitly handled by the operating system which dynamically assigns rows to threads. The reduction operations for DOT and the convergence test were handled automatically as well. Otherwise, special synchronization constructs were not required since there are no other possible race conditions in the multithreaded CG. It is important to highlight that no special ordering was necessary to achieve parallel performance.

Results using 60 streams per processor are presented in Table 6. Notice that both CG and the underlying SPMV achieve high scalability of over 90%. This indicates that there is enough instruction level parallelism in CG to tolerate the relatively high overhead of memory access. There is a slight drop in performance between four and eight processors. As we increase the number of processors, the number of active threads increase proportionately while the runtimes become very small. As a result, a greater percentage of the overall time is spent on thread management, causing a decrease in efficiency. We look forward to continuing our experiments as more processors become

	ORIG	
P	SPMV	CG
1	0.378	9.86
2	0.189	5.02
4	0.095	2.53
8	0.051	1.35

Table 6: Runtimes (in seconds) for the original ordering on the Tera MTA.

available on the Tera MTA.

5 Summary and Future Work

In this paper, we examined three different parallel implementations of the Conjugate Gradient (CG) algorithm using three leading programming paradigms and architectures. The MPI version of the code on the T3E uses the Aztec [6] library, where we compared the parallel performance of ordering the sparse matrix using reverse Cuthill-McKee (RCM) [3], self-avoiding walk (SAW) [5], and the METIS partitioner [7]. Results showed that all three schemes greatly improve the parallel performance of CG compared to the naive natural ordering. In addition, we demonstrated that traditional graph partitioners, which focus on minimizing edge cuts, are not necessarily the best tools for partitioning sparse matrices on multiprocessor systems. Using RCM or SAW as an ordering/partitioning strategy results in a faster CG than METIS, due to better cache reuse. A performance model was also presented which predicts the expected sparse matrix-vector multiply (SPMV) runtime as a function of both cache misses and communication overhead. Within each CG iteration, the SPMV is usually the most expensive operation.

A shared memory implementation of CG on the Origin2000 showed that ordering algorithms dramatically improve parallel performance. This is because the Origin2000 is a distributed-memory architecture, so proper data distribution is required even when programming in shared memory mode. A direct comparison with an MPI implementation indicated that it is possible to achieve message passing performance using shared memory constructs for this class of applications through careful data ordering and distribution. Finally, results of a multithreaded implementation of CG on the Tera MTA indicated that special ordering and/or partitioning schemes are not required on the MTA to obtain high efficiency and scalability.

We plan to port the distributed-memory implementation of CG onto the newly installed RS/6000 SP machine at NERSC. This system consists of 256 two-CPU SMP nodes, and is the first commercial implementation of the POWER3 microprocessor. In addition, we will examine the effects of partitioning the sparse matrix using METIS, and subsequently performing RCM or SAW orderings on each subdomain. Combining both schemes should minimize interprocessor communication and significantly improve data locality. Future research will focus on evaluating the effectiveness of the parallel Jacobi-Davidson eigensolver, when various orderings are applied to the underlying sparse matrix. A multithreaded version of the Jacobi-Davidson algorithm will be implemented on the Tera

MTA. We also intend to extend the SAW algorithm to three dimensions and modify it to efficiently handle adaptively refined meshes in a parallel environment.

Acknowledgements

The work of the first two authors was supported by 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. The work of the third author was partially supported by the National Science Foundation under grant numbers NSF-CISE-9726388 and NSF-MIPS-9707125 while the author was at the University of Delaware. The work of the fourth author was supported by National Aeronautics and Space Administration under contract number NAS 2-14303 with MRJ Technology Solutions.

References

[1] R. Barrett, M. Berry, T.F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the solution of linear systems: Building blocks for the iterative methods.* SIAM, Philadelphia, 1994.

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

[3] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proc. ACM National Conference*, pages 157–172, New York, 1969.

[4] M. Griebel and G. Zumbusch. Hash-storage techniques for adaptive multilevel solvers and their domain decomposition parallelization. *AMS Contemporary Mathematics Series*, 218:279–286, 1998.

[5] G. Heber, R. Biswas, and G.R. Gao. Self-avoiding walks over adaptive unstructured grids. In *Parallel and Distributed Processing (LNCS 1586)*, pages 968–977. Springer-Verlag, Berlin, 1999.

[6] S.A. Hutchinson, L.V. Prevost, J.N. Shadid, and R.S. Tuminaro. Aztec User's Guide, Version 2.0 Beta. Technical Report SAND95-1559, Sandia National Laboratories, Albuquerque, 1998.

[7] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Scientific and Statistical Computing*, 20:359–392, 1998.

[8] C.-W. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular problems. *J. of Supercomputing*, 10:119–140, 1995.

[9] M. Parashar and J.C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proc. 29th Hawaii International Conference on System Sciences*, pages 604–613, Maui, 1996.

[10] J.R. Pilkington and S.B. Baden. Dynamic partitioning of non-uniform structured workloads with space-filling curves. *IEEE Trans. on Parallel and Distributed Systems*, 7:288–300, 1996.

[11] Y. Saad. *Iterative methods for sparse linear systems.* PWS Publishing Company, Boston, 1996.

[12] J. Salmon and M.S. Warren. Parallel, out-of-core methods for fast evaluation of long-range interactions. In *Proc. 8th SIAM Conference on Parallel Processing for Scientific Computing*, Minneapolis, 1997.

[13] J.R. Shewchuk. Triangle: Engineering a 2D quality mesh generator and delaunay triangulator. In *Applied Computational Geometry: Towards Geometric Engineering (LNCS 1148)*, pages 203–222. Springer-Verlag, New York, 1996.