

Ordering Unstructured Meshes for Sparse Matrix Computations on Leading Parallel Systems

Leonid Oliker¹, Xiaoye Li¹, Gerd Heber², and Rupak Biswas³

¹ NERSC, Lawrence Berkeley National Laboratory, Berkeley, CA 94720, USA
`{oliker,xiaoye}@nersc.gov`

² Cornell Theory Center, Rhodes Hall, Cornell University, Ithaca, NY 14853, USA
`heber@tc.cornell.edu`

³ MRJ/NASA Ames Research Center, Moffett Field, CA 94035, USA
`rbiswas@nas.nasa.gov`

Abstract. Computer simulations of realistic applications usually require solving a set of non-linear partial differential equations (PDEs) over a finite region. The process of obtaining numerical solutions to the governing PDEs involves solving large sparse linear or eigen systems over the unstructured meshes that model the underlying physical objects. These systems are often solved iteratively, where the sparse matrix-vector multiply (SPMV) is the most expensive operation within each iteration. In this paper, we focus on the efficiency of SPMV using various ordering/partitioning algorithms. We examine different implementations using three leading programming paradigms and architectures. Results show that ordering greatly improves performance, and that cache reuse can be more important than reducing communication. However, a multithreaded implementation indicates that ordering and partitioning are not required on the Tera MTA to obtain an efficient and scalable SPMV.

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. For example, one thrust area in the U.S. DoE Grand Challenge projects is to design future accelerators such as the Spallation Neutron Source (SNS). Our colleagues at the Stanford Linear Accelerator Center (SLAC) need to model complex RFQ cavities with large aspect ratios [6]. Unstructured grids are currently used to resolve the small features in a large computational domain; dynamic mesh adaptation will be added in the future for additional efficiency.

The PDEs for electromagnetics are discretized by the FEM method, which leads to a generalized eigenvalue problem $Kx = \lambda Mx$, where K and M are the stiffness and mass matrices, and are very sparse. In a typical cavity model, the

number of degrees of freedom is about one million. For such large eigenproblems, direct solution techniques quickly reach the memory limits. Instead, the most widely-used methods are Krylov subspace methods, such as Lanczos or Jacobi-Davidson. In all the Krylov-based algorithms, sparse matrix-vector multiplication (SPMV) must be performed repeatedly. Therefore, the efficiency of SPMV usually determines the eigensolver speed. SPMV is also one of the most heavily used kernels in large-scale numerical simulations.

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. 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 [1]. 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 partitioning tool like MeTiS [5], and to post-process the resulting partitions with an enumeration strategy for enhanced locality. Although, optimizations for partitioning and locality may then be treated as separate problems, real applications tend to show a rather intricate interplay of both.

In this paper, we investigate the efficiency of SPMV using various ordering/partitioning algorithms. We examine different implementations using different programming paradigms and architectures. Results on state-of-the-art parallel supercomputers show that ordering greatly improves performance, and that cache reuse can be more important than reducing communication. However, a multithreaded implementation indicates that ordering and partitioning are not required on the Tera MTA to obtain an efficient and scalable SPMV.

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. The linearization of a higher-dimensional spatial structure, i.e. its mapping onto a one-dimensional hyperspace, 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 enumeration techniques and the general-purpose graph partitioner which were used. In future, we plan to integrate ordering algorithms into the eigensolver at SLAC and evaluate the overall performance gain.

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 [2] 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. 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 [3] 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 (MeTiS)

Some excellent parallel graph partitioning algorithms have been developed and implemented over the last 5-10 years that are extremely fast while giving good load balance quality and low edge cuts. Perhaps the most popular is MeTiS [5], belonging 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.

3 Experimental Results

To perform a SPMV, $y \leftarrow Ax$, we assume that the nonzeros of matrix A are stored in a compressed sparse row format. 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 dif-

ferent patterns for accessing the entries of x . Moreover, on a distributed-memory machine, they imply different amounts of communication.

Our experimental test mesh consists of a two-dimensional Delaunay triangulation, generated by Triangle [7]. 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 to each (row, column) entry corresponding to the vertex endpoints (v_1, v_2) of the edges in the mesh. This simulates a stencil computation where each vertex needs to communicate with its nearest neighbors. The final matrix is extremely sparse containing only 2,635,207 nonzeros. The number of floating point operations required for a SPMV is twice the number of nonzeros (5,270,414 for our test matrix).

3.1 Distributed-Memory Implementation

In our experiments, we use the parallel SPMV routines in Aztec [4] implemented using MPI. The matrix A is partitioned into blocks of rows, with each block assigned to one processor. Two routines are of particular interest: `AZ_transform` and `AZ_matvec_mult`. The first initializes the data structures and the communication schedule, while the second performs the matrix-vector multiply. In Table 1, we report the runtimes of these two routines on the 450 MHz Cray T3E at NERSC. The original natural ordering (ORIG) is the slowest and clearly unacceptable on distributed-memory machines. For `AZ_matvec_mult`, the key kernel routine, RCM is slightly but consistently faster than SAW, while MeTiS requires almost twice the RCM execution time. However, MeTiS, RCM, and SAW, all demonstrate excellent scalability up to the 64 processors that were used for these experiments. The pre-processing times in `AZ_transform` are more than an order of magnitude larger than the corresponding times for `AZ_matvec_mult` (except for ORIG where it is two to three orders of magnitude larger).

Table 1. Runtimes (in seconds) for different orderings on the Cray T3E

| P | <code>AZ_matvec_mult</code> | | | | <code>AZ_transform</code> | | | |
|----|-----------------------------|-------|-------|-------|---------------------------|-------|-------|-------|
| | ORIG | MeTiS | RCM | SAW | ORIG | MeTiS | RCM | SAW |
| 4 | 0.228 | 0.093 | 0.046 | 0.054 | 181.556 | 0.573 | 0.419 | 0.466 |
| 8 | 0.161 | 0.046 | 0.024 | 0.029 | 207.567 | 0.296 | 0.219 | 0.266 |
| 16 | 0.080 | 0.022 | 0.012 | 0.013 | 140.269 | 0.159 | 0.122 | 0.177 |
| 32 | 0.046 | 0.010 | 0.006 | 0.006 | 21.095 | 0.132 | 0.179 | 0.124 |
| 64 | 0.027 | 0.005 | 0.003 | 0.004 | 6.682 | 0.142 | 0.158 | 0.114 |

To better understand the various partitioning/ordering algorithms, we have built a simple performance model to predict the parallel runtime. First, using the T3E’s hardware performance monitor, we collected the average number of cache misses per processor. This is reported in Table 2. 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

Table 2. Locality and communication statistics of `AZ_matvec_mult`

| P | Avg. Cache Misses (10^6) | | | | Avg. Comm. (64-bit words) | | | | Max. Message Count | | | |
|----|------------------------------|-------|------|------|---------------------------|-------|-----|-----|--------------------|-------|-----|-----|
| | ORIG | MeTiS | RCM | SAW | ORIG | MeTiS | RCM | SAW | ORIG | MeTiS | RCM | SAW |
| 4 | 1.02 | 0.63 | 0.71 | 0.48 | 85323 | 115 | 418 | 585 | 3 | 3 | 2 | 3 |
| 8 | 0.62 | 0.31 | 0.36 | 0.26 | 60425 | 130 | 424 | 524 | 7 | 3 | 2 | 6 |
| 16 | 0.30 | 0.16 | 0.17 | 0.11 | 33084 | 132 | 432 | 385 | 15 | 4 | 2 | 8 |
| 32 | 0.16 | 0.08 | 0.10 | 0.06 | 17139 | 112 | 434 | 319 | 31 | 5 | 2 | 9 |
| 64 | 0.10 | 0.05 | 0.06 | 0.03 | 8717 | 95 | 434 | 241 | 63 | 6 | 2 | 14 |

communication volume and the maximum number of messages per processor, both of which are also shown in Table 2. MeTiS transfers the least amount of data, whereas RCM has the fewest number of messages.

In our model, we estimate the total parallel runtime as $T = T_f + T_m + T_c$, where T_f , T_m , and T_c are the estimated 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 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 2. We found a maximum deviation of 74% from the measured runtimes. The model showed that servicing the cache misses was extremely expensive and required more than 95% of the total time for MeTiS, RCM, and SAW, and almost 80% for ORIG which has relatively more communication.

3.2 Shared-Memory Implementation

The shared-memory version of SPMV was implemented on the Origin2000, which is a SMP cluster of nodes each containing two processors, some local memory, and 4 MB secondary cache per processor. This parallel code was written using SGI's native pragma directives, which create IRIX threads. Each processor is assigned an equal number of rows in the matrix. SPMV proceeds in parallel without the need for any synchronization, since there are no concurrent writes. The 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 were simpler to implement than the MPI version. Table 3 shows the runtime of SPMV using both approaches with the ORIG, RCM, and SAW orderings of the mesh.

Table 3. Runtimes (in seconds) for different orderings running in SHMEM and CC-NUMA modes on the SGI Origin2000

| P | SHMEM | | | CC-NUMA | | |
|----|-------|-------|-------|---------|-------|-------|
| | ORIG | RCM | SAW | ORIG | RCM | SAW |
| 1 | 0.348 | 0.254 | 0.250 | 0.341 | 0.247 | 0.244 |
| 2 | 0.200 | 0.150 | 0.147 | 0.189 | 0.152 | 0.091 |
| 4 | 0.167 | 0.138 | 0.134 | 0.191 | 0.025 | 0.028 |
| 8 | 0.169 | 0.111 | 0.104 | 0.088 | 0.004 | 0.006 |
| 16 | 0.200 | 0.184 | 0.206 | 0.060 | 0.002 | 0.005 |
| 32 | 0.483 | 0.453 | 0.359 | 0.054 | 0.006 | 0.006 |
| 64 | 0.947 | 0.931 | 0.891 | 0.065 | 0.015 | 0.008 |

As expected, the CC-NUMA implementation shows significant performance gain over SHMEM. Within the CC-NUMA approach, RCM and SAW dramatically reduce the runtimes as compared to ORIG, indicating that an ordering algorithm is necessary to achieve good performance on distributed shared-memory systems. There is little difference in parallel performance between RCM and SAW because both reduce the number of secondary cache misses and the non-local memory references of the processors. However, there is a slowdown in performance when using more than 16 processors. This is due to the increased surface-to-volume ratio of the mesh partitions, which cause the overhead of cache coherence and false sharing to grow with the numbers of processors.

3.3 Multithreaded Implementation

The Tera MTA is a supercomputer recently installed at SDSC. The MTA has a radically different architecture than current high-performance computer systems. Each 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. Rather than using data caches to hide latency, the MTA processors use multithreading to tolerate latency. Once a code has been written in the multithreaded model, no additional work is required to run it on multiple processors.

The multithreaded implementation of the SPMV was trivial, requiring only MTA compiler directives. Load balancing is implicitly handled by the operating system which dynamically assigns rows to threads. Special synchronization constructs were not required since there are no possible race conditions in the multithreaded SPMV. For this implementation, no special ordering is required to achieve parallel performance and scalability. Results, shown in Table 4, indicate that there is enough instruction level parallelism in SPMV to tolerate the relatively high overhead of memory access. However, MTA runtimes will generally be slower than traditional cache-based systems for load balanced applications with substantial cache reuse.

Table 4. Runtimes (in seconds) on the Tera MTA using 60 streams per processor

| P=1 | P=2 | P=4 | P=8 |
|--------|--------|--------|--------|
| 0.0812 | 0.0406 | 0.0203 | 0.0103 |

4 Work in Progress

In this paper, we examined different ordering strategies for SPMV, using three leading programming paradigms and architectures. We plan to port the distributed-memory implementation of SPMV onto the newly installed RS/6000 SP machine at NERSC. In addition, we will examine the effects of partitioning the sparse matrix using MeTiS and then 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. Finally, we intend to extend the SAW algorithm to three-dimensional meshes and modify it to efficiently handle adaptively refined meshes in a parallel environment.

Acknowledgements

The first two authors were supported by the Office of Computational and Technology Research, U.S. DoE Division of Mathematical, Information, and Computational Sciences under contract DE-AC03-76SF00098. The third author was supported by NSF under grants CISE-9726388 and MIPS-9707125 while he was at the University of Delaware. The work of the fourth author was supported by NASA under contract NAS 2-14303 with MRJ Technology Solutions.

References

1. Burgess, D.A., Giles, M.B.: Renumbering Unstructured Grids to Improve the Performance of Codes on Hierarchical Memory Machines. *Adv. Engrg. Soft.* **28** (1997) 189–201
2. Cuthill, E., McKee, J.: Reducing the Bandwidth of Sparse Symmetric Matrices. *Proc. ACM Natl. Conf.* (1969) 157–172
3. Heber, G., Biswas, R., Gao, G.R.: Self-Avoiding Walks over Adaptive Unstructured Grids. *Springer-Verlag LNCS* **1586** (1999) 968–977
4. Hutchinson, S.A., Prevost, L.V., Shadid, J.N., Tuminaro, R.S.: *Aztec User's Guide*, v 2.0 Beta. Sandia National Laboratories TR SAND95-1559 (1998)
5. Karypis, G., Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.* **20** (1998) 359–392
6. McCandless, B., Li, Z., Srinivas, V., Sun, Y., Ko, K.: Omega3P: A Parallel Eigensolver for Modeling Large, Complex Cavities. *Proc. Intl. Comput. Accel. Phys. Conf.* (1998)
7. Shewchuk, J.R.: Triangle: Engineering a 2D Quality Mesh Generator and Delaunay Triangulator. *Springer-Verlag LNCS* **1148** (1996) 203–222