Optimizing Sparse Matrix Vector Multiplication on SMPs *

Eun-Jin Im and Katherine Yelick †

Abstract

We describe optimizations of sparse matrix-vector multiplication on uniprocessors and SMPs. The optimization techniques include register blocking, cache blocking, and matrix reordering. We focus on optimizations that improve performance on SMPs, in particular, matrix reordering implemented using two different graph algorithms. We present a performance study of this algorithmic kernel, showing how the optimization techniques affect absolute performance and scalability, how they interact with one another, and how the performance benefits depend on matrix structure.

1 Introduction

Sparse matrix-vector multiplication is an important computational kernel used in scientific computation, signal and image processing, document retrieval, and many other applications. In general, the problem is to compute $y = \alpha A \times x + \beta y$, for sparse matrix A, dense vectors x and y, and scalars α and β. Sparse matrix algorithms tend to run much more slowly than their dense matrix counterparts. For example, on 167 MHz Ultrasparc I, the naive implementation of sparse matrix-vector multiplication runs at 25 Mflops/s for a 1000×1000 dense matrix represented in sparse format. This performance is heavily dependent on the nonzero structure of the matrix, and can be as low as 5 Mflops/s for matrices with a lower ratio of nonzero. For comparison, a naive implementation of dense matrix-vector multiplication runs at 38 Mflops/s, and the vendor-supplied routine runs at 58 Mflops/s. The primary reason for this performance difference is poor data locality in access to the source vector x in the sparse case.

We are building a toolbox called Sparsity for automatically producing optimized sparse matrix vector kernels on uniprocessors and shared memory multiprocessors (SMPs). As part of the development of Sparsity, we have performed a study of memory hierarchy optimization techniques and their benefits on SMPs. We measure the performance advantages for matrices taken from range of application domains, including Finite Element methods, linear programming, and document retrieval (e.g., Latent Semantic Indexing). In this paper we describe the performance results on one or more nodes of an Ultrasparc SMP. The optimization techniques fall into three categories: register blocking, cache blocking, and matrix reordering. We describe each of these here and give some highlights of the performance results.

*This work was supported in part by the Advanced Research Projects Agency of the Department of Defense (F30602-95-C-0136 and N00600-93C-2481), by the Department of Energy (DE-FG03-94ER25206 and W-7405-ENG-48), by the Army Research Office (DAAH04-96-1-0079 and DAA655-98-1-0153), and by the National Science Foundation (CDA-9401156). In addition, the equipment for this research was provided in part by Sun Microsystems, Intel, IBM, and NERSC/LBNL and Hewlett-Packard.

†Computer Science Division, University of California, Berkeley, CA.

2 Related Work

The interface to the routine for dense matrix-vector multiplication is standardized in the Basic Linear Algebra Subprograms (BLAS) [2]; most hardware vendors provide optimized BLAS libraries, highly tuned for their own architectures. Unlike the dense matrix calculation, there are no highly optimized sparse matrix routines supplied by vendors, because there is not yet a standard interface for sparse matrices, and there are diverse formats that different users find suitable for their applications. A standardization effort for sparse matrix routines is currently going on in BLAS Technical (BLAST) Forum [1], but this does not include plans to implement highly optimized versions for specific platforms. As a result of an effort to provide a generic sparse matrix operation library, the NIST sparseBLAS [10] provides generic routines and TNT (Template Numerical Toolkit) [9] provides a generic matrix/vector classes. BlockSolve [8] and Aztec [4] are parallel iterative solvers that include the implementation of optimized sparse matrix operations.

3 Register Blocking

Register blocking reorganizes the sparse matrices into a set of fixed-size dense blocks by finding a small block size which fits into the target machine's register set, and also fits well to the original sparse matrix, meaning the number of extra zero elements stored to compose dense blocks is reasonably small. This optimization improves the reuse of values of vector x within block, eliminates some loop overheads, and improves the ability of a compiler to perform instruction scheduling to overlap memory operations.

Figure 1 shows the raw performance of register blocked multiplication on different block sizes of a dense matrix in sparse format. This is likely to be an upper bound on the performance of sparse matrices in the same format. Figure 2 shows the effective performance of register blocked multiplication on the scientific computing matrices. (By *effective performance*, we mean the Mflop rate counting only the flops that are required by the unblocked code.) The register block size is dependent on the nonzero structure of sparse matrix, and we used a heuristic to find a block size that generates small number of zero element to fill dense blocks [5].

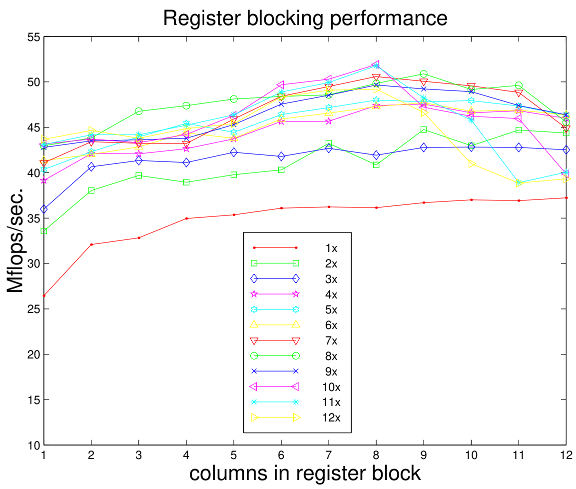

FIG. 1. **Performance of register blocked multiplication** *taken from 167 MHz Ultrasparc for* 1000×1000 *dense matrix represented in sparse format. Each line is for a fixed number of rows, varying the number of columns from 1 to 12.*

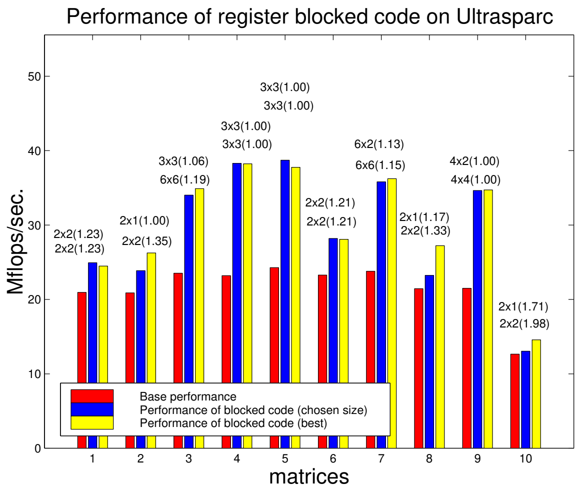

Fig. 2. **Performance of register blocked multiplication on 10 sparse matrices from scientific applications taken on a 167 MHz Ultrasparc.** *The first bar is the base performance and the second bar is the performance of the register blocked matrix for a block size chosen by our model, and the third bar is the best performance over all the block sizes considered. The numbers on the top of the bars are block size and fill overhead in the parenthesis for the second bars (bottom) and the third bars (top).*

matrix	size(N)	nonzeros	Application area
1	23560	484K	Airfoil eigenvalue calculation
2	41092	1.7M	2D PDE problem
3	30237	1.5M	Automobile frame stiffness matrix
4	13965	1.0M	FEM stiffness matrix
5	24696	1.8M	FEM stiffness matrix
6	52329	2.7M	Engine block stiffness matrix
7	54870	2.7M	Structure from shuttle rocket booster
8	4134	94K	Chemical process separation
9	62424	1.7M	Unstructured Euler solver
10	26068	177K	Device simulation

Fig. 3. **Scientific computing matrices used in the experiments.** *The table shows the matrix dimension, number of nonzeros, and the application area of the matrix.*

Figure 3 illustrates the matrices used in the experiment. In our experience, register blocking tends to be effective on matrices from physical simulations, where the matrix often contains a large number of small fixed-size dense sub-blocks, but not for linear programming or document retrieval.

Performance improvement from register blocking on an SMP are similar to the uniprocessor results, since register blocking does not affect parallelism. In section 5 we will combine the register blocking technique with matrix reordering used to to improve scalability.

4 Cache Blocking

As an extension of the register blocking idea, we consider an optimization called *cache blocking*. The idea of this optimization is to keep c_{cache} elements of vector x in the cache

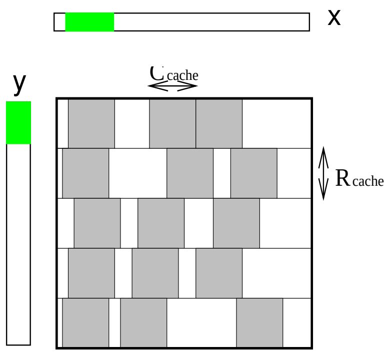

FIG. 4. **Alignment of cache-blocks in sparse matrix** *The grey areas are sparse matrix blocks that contain nonzero elements in the* $c_{cache} \times r_{cache}$ *rectangle. The white areas are the areas that contain no nonzero elements.*

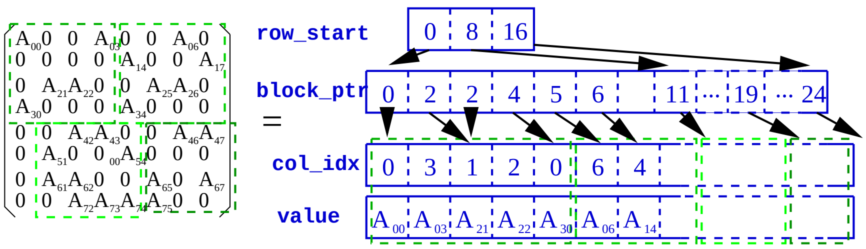

FIG. 5. **Storage format of cache-blocked sparse matrix** *The nonzero elements of* $r_{cache} \times c_{cache}$ *blocks are stored in contiguous locations. The cache blocks consist of block_ptr, col_idx and value arrays. The block_ptr array keeps track of starting points of each of* r_{cache} *rows. The row_start index array stores the indices into block_ptr array for every* r_{cache}*-th row.*

while an $r_{cache} \times c_{cache}$ block of matrix A is multiplied to this portion of vector x. That is, we limit the vector products so that the elements of vector x can all be kept in cache and re-used for the vector product for the next row. A matrix with equal size cache blocks identified is illustrated in figure 4.

Unlike register blocking, creating dense $r_{cache} \times c_{cache}$ blocks by filling in zeros is not practical, because expanding $r_{cache} \times c_{cache}$ sparse matrices to dense matrices will incur excessive storage and computation overhead. For that reason, the blocks in the cache blocked matrix are stored as sparse matrices in the implementation of *static cache blocking*. The sparse matrix is reorganized by changing the order of the nonzero elements of sparse matrix in the storage as shown in figure 5. We have also considered a variant of cache blocking which we call *dynamic cache blocking* in which the representation is left unchanged, but a set of r_{cache} pointers are used to keep track of blocks dynamically. However, the additional pointer manipulation and control required for dynamic cache blocking made it less useful than the static case.

We measured the performance of cache blocked matrix-vector multiplication for the matrices in the scientific computing set on a 167 MHz Ultrasparc I, which has 512K bytes of off-chip L2 cache. The block size was chosen empirically as $16K \times 16K$. Unfortunately, cache blocking shows a slight degradation in performance for all of these matrices from numerical simulations. However, for a matrix that arises in a document retrieval algorithm called Latent Semantic Indexing (LSI), cache blocking dramatically improves performance.

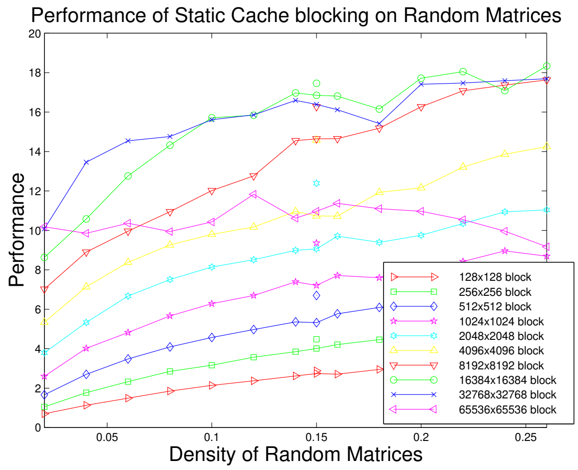

FIG. 6. **Performance of cache-blocked multiplication on random matrices** *measured on* $64K \times 64K$ *random matrices with different densities 0.02–0.26%. Each line represents different cache block sizes, and the separate points at 0.15% are the performance of LSI matrix.*

The unblocked code runs at 5.8 Mflops/s on an Ultrasparc, while the cache blocked code runs at 18 Mflops/s, giving a speedup of 3.1. It is interesting to note that register blocking for the LSI matrix showed no benefit.

The nonzero pattern of the LSI matrix is unusual compared to most scientific applications, in that it has little discernible structure.[1] Combined with the fact that the size of the matrix is very large, the performance of multiplication on LSI matrix before the optimization is very low (5.8 Mflops/sec.) relative to the other matrices (10–25 Mflops/sec.).

As further evidence that cache blocking is effective on matrices with evenly distributed nonzeros, we also measured the performance on random matrices. The results are shown in figure 6, with the x-axis varying the density of nonzero elements between 0.02% and 0.26%. The size of the random matrix was $64K \times 64K$, and the performance was measured for different cache block sizes. The performance of LSI multiplication for the same cache block sizes are shown in the same figure as separate points above $x = 0.15\%$, the density of the LSI matrix. The performance characteristics of the LSI matrix are very similar to those of a random matrix.

We also ran cache blocked multiplication on 8-way Ultrasparc SMP. We tried several strategies in applying cache blocking on an SMP.

- C1: Rows in the same block are distributed evenly to participating processors.
- C2: Each block of rows are assigned to each processor.
- C3: The calculation starts from the diagonal block in the previous setting (C2).
- C4: Each block of columns are assigned to each processor.

The assignment of computation to processors is illustrated in figure 7 for the 4 processor case . In C3, assignment is same to that of C2, but the calculation starts from the diagonal

[1]Sparse matrix-vector multiplication is the kernel of the LSI algorithm. Our matrix came from NERSC/LBNL in collaboration with the Inktomi company, and is a subset of real data from the web. They use an algorithm different from LSI, which is also based on sparse matrix-vector multiplication.

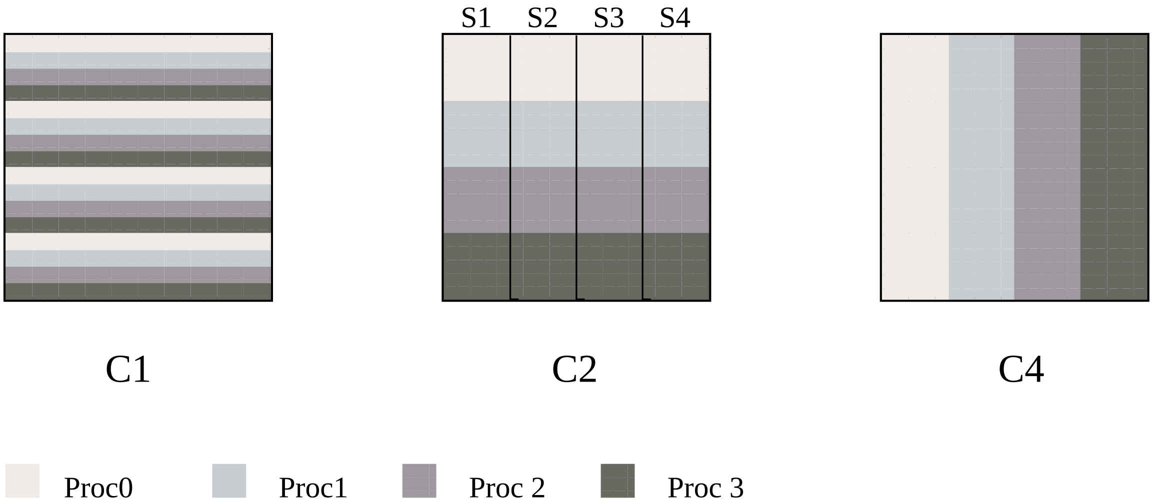

Fig. 7. **Configuration of cache blocking on SMP**

block, i.e., proc0 begins from S1, proc1 from S1, and so on, to make each processor read different parts of source vector.

The performance is shown in figure 8. In the left graph, we show the performance per processor for the multiplication of LSI matrix. The base performance, which is the performance on a uniprocessor without any optimizations, is also shown for comparison. The portion of the LSI matrix used in this experiment is $10K \times 256K$, with 3.8M nonzero elements. The C1 configuration runs no faster than the base performance because the long rows do not fit in the cache. As a variation on configuration C1, we used block sizes that were limited to 16K columns per block, and showed a performance of 17-18 Mflops/s per processor. Configurations C2 and C3 exhibit the same performance, since the data layout is the same; we might expected some slight benefits of C3 over C2, because each processor was accessing different parts of memory in C3, but these effects are negligible. Configuration C4 scales very well, showing the same performance per processor as the cache blocked implementation on a uniprocessor. In configuration C2, C3, and C4, the reason why the performance is low at small number of processors (1 to 4), is that its row size of cache block is too wide (number of columns / number of processors) to fit in cache. When we limit the row size to approximately 16K, and also evenly divide the number of columns across the processors, we see 16-17 Mflops/s per processor for any number of processors.

The right graph shows the performance for matrix number 5 from the scientific computing set, a finite element method matrix. The dimension of this matrix is 24696×24696 with 1.7M nonzero elements. Unlike the LSI matrix, the row length is not long enough to cause problems fitting in cache. Although this matrix exhibited no benefits of cache blocking on a uniprocessor, on an SMP we see that configurations C2 and C3 maintain almost linear speedup, and are better than configurations C1 and C4. In C4, extra synchronization is needed to accumulate partial calculations before updating the destination vector, which makes it slower.

5 Matrix Reordering

Matrix reordering changes the order of rows and columns within A which in turn changes the memory access pattern to vector x. A matrix can be reordered to reduce cache misses and

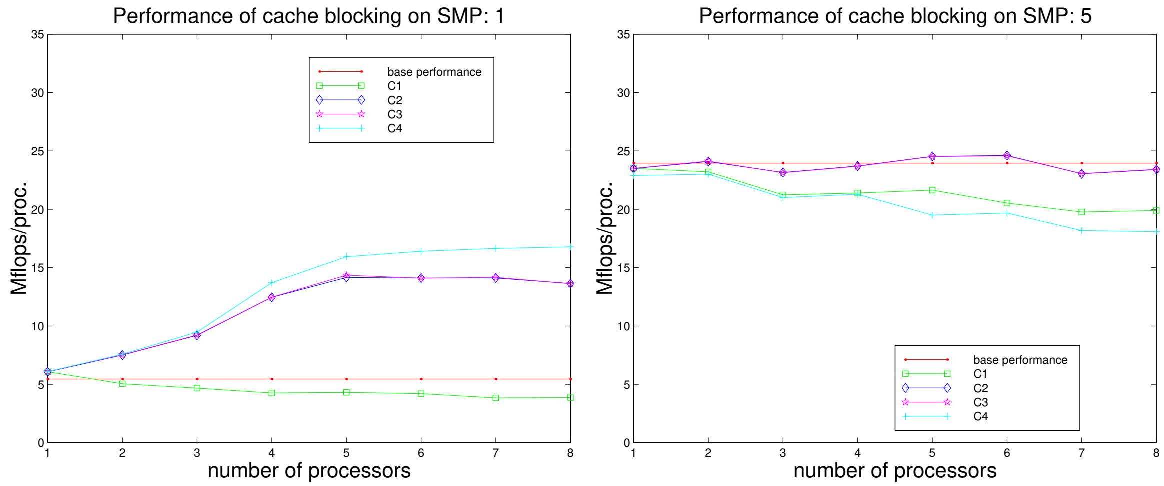

FIG. 8. **Performance of cache-blocked multiplication on 8-way Ultrasparc SMP.** *The left figure shows the performance of multiplication on LSI matrix and right figure shows the performance of multiplication on the 5th matrix from the scientific computing matrices.*

memory coherence traffic across the bus or crossbar of an SMP. Because we are interested in reorderings that are general enough to handle matrices from linear programming and document retrieval, neither of which are symmetric, we extended some existing techniques to the nonsymmetric case.

First, we used a graph numbering scheme, *Reverse Cuthill-McKee* (RCM) [7], in which graph nodes are numbered in reverse order according to Breadth First Search. This is a heuristic that aims to reduce the bandwidth of the symmetric matrix, by moving the nonzeros close to each other (near the diagonal) and therefore improving locality of access to data within the source and destination vectors. The original RCM algorithm considers the symmetric matrix as an adjacent matrix of the graph, i.e., nodes i and j are connected if A_{ij} and A_{ji} are nonzero. We extend the algorithm to non-symmetric, rectangular matrices by differentiating row nodes and column nodes.

We also applied graph partitioning algorithms using the hMETIS [6] package developed at University of Minnesota, which implements a multi-level partitioning algorithm on hypergraphs with tunable parameters based on work by Hendrickson and Leland [3]. In a hypergraph, a set of nodes are connected with a hyperedge. We can partition rows of the matrix by considering each column to be a hyperedge whose nonzero elements compose a set of nodes in the hyperedge. Similarly, the columns are partitioned by considering each row as hyperedge. The hypergraph partitioning was implemented in hMETIS package using multilevel algorithm.

First we applied these two reordering techniques on a uniprocessor implementation using the scientific computing matrices. On an Ultrasparc, RCM bandwidth reduction improves the performance of matrix-vector multiplication on some matrices that arise in linear programming problems by 18 %. Prior to reordering, the linear programming matrices have very wide bandwidth and no locally dense blocks. The RCM reordering significantly reduced the matrix bandwidth, and in doing so, also changed the memory access pattern to source vector x to enhance cache access.

Secondly, we compared the performance improvement and scalability of multiplication on an SMP. We experimented with the scientific computing matrices, and we combined

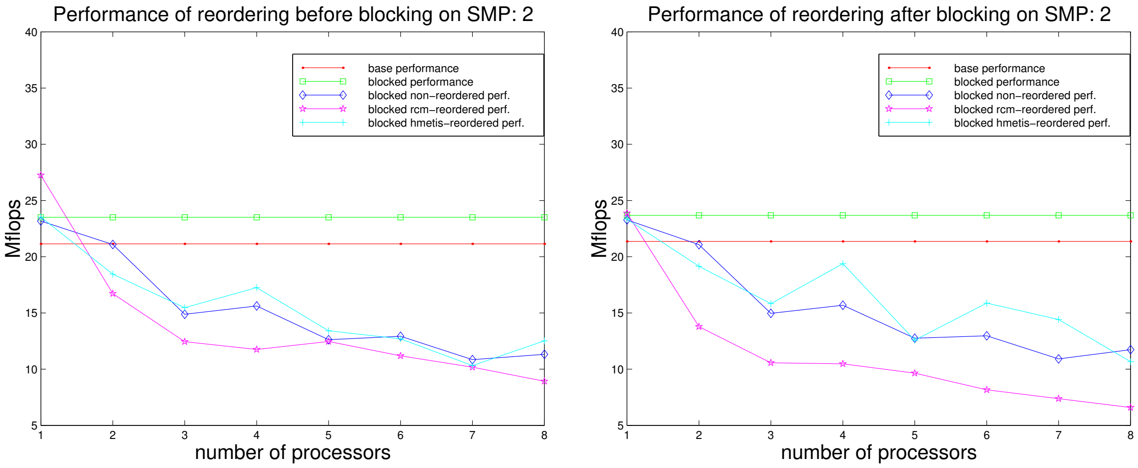

FIG. 9. **Performance per processor of multiplication on 8-way Ultrasparc SMP** *Left graph shows performance when reordering is applied before register blocking and the right graph shows performance when reordering is applied after register blocking is done. Two horizontal lines in both graphs are base performance on uniprocessor, one for non-blocked matrix, and the other for blocked performance. So, we expect this line when we have efficiency of 1.*

register blocking and matrix reordering in different orders – reordering was applied before and after register blocking. One can expect partitioning the matrix first generates more clusters in the matrix, making more dense blocks, but on the other hand, blocking first makes it possible to keep existing dense blocks during reordering. We observed that the graph partitioning algorithm generally generates better reorderings than RCM as shown in figure 9. For matrices with natural dense blocks, partitioning after blocking was better than partitioning before blocking.

6 Conclusions

We described optimization study of sparse matrix-vector multiplication on a uniprocessor and SMP. The optimizations are register blocking, cache blocking and matrix reordering. We also tried the combination of register blocking and matrix reordering.

Each of the three optimization techniques shows a noticeable performance benefit for matrices from some application domain. The matrix structure for which each is beneficial, though, is quite different. Register blocking is most effective when the matrix contains a natural dense blocking factor that also matches the number of registers in the machine; these arise most frequently in finite element applications. Cache blocking is only effective on matrices with nearly random nonzero structure; we found it to be extremely useful in document retrieval and in synthetic matrices with a random nonzero structure. For the SMP case, we showed that assigning processors across the row blocks is better for matrices from FEM, while assigning processors across the column blocks is better for the LSI matrix, where the matrix is much wider. Matrix reordering improved the uniprocessor performance of a matrix from linear programming whose bandwidth is very wide, but had no effect on uniprocessor performance for the scientific matrices. When combining register blocking and reordering on an SMP, we showed that the hypergraph partition reorders the matrix better than RCM, and that blocking before reordering was better because it keeps existing blocks.

The results presented here have shown the importance of good data structure organization for sparse matrices on both uniprocessors and SMPs. Unfortunately, there is no single format that is best for all machines or all matrices. Instead, we have identified some application domains and matrix characteristics that can be used to choose the representation. Our ultimate goal is to use this optimization study to build a toolbox for automatically generating highly optimized sparse matrix vector multiplication codes based on matrix and machine characterisitics.

References

[1] BLAST Forum, *Documentation for the Basic Linear Algebra Subprograms (BLAS)*, Aug. 1997. http://www.netlib.org/utk/papers/sparse.ps.

[2] J. Dongarra, J. D. Croz, S. Hammarling, and R. J. Hanson, *An extended set of FORTRAN Basic Linear Algebra Subprograms*, ACM Trans. Math. Soft., 14 (1988), pp. 1–17.

[3] B. Hendrickson and R. Leland, *A multilevel algorithm for partitioning graphs*, Tech. Rep. SAND93-1301, Sandia National Laboratories, 1993.

[4] S. A. Hutchinson, J. N. Shadid, and R. S. Tuminaro, *Aztec user's guide: Version 1.1*, Tech. Rep. SAND95-1559, Sandia National Laboratories, 1995.

[5] E.-J. Im and K. Yelick, *Model-based memory hierarchy optimizations for sparse matrices*, Oct. 1998. Workshop on Profile and Feedback-Directed Compilation.

[6] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, *hMETIS A Hypergraph Partitioning Package*, University of Minnesota, Jan. 1998.

[7] W.-H. Liu and A. H. Sherman, *Comparative analysis of the Cuthill-McKee and the reverse Cuthill-McKee ordering algorithnms for sparse matrices*, in SIAM J. Numerical Analysis, 1976, pp. 198–213.

[8] P. Plassmann and M. T. Jones, *BlockSolve95 users manual: Scalable library software for the parallel solution of sparse linear systems*, Tech. Rep. ANL-95/48, Argonne National Laboratory, 1995.

[9] R. Pozo, *Template numerical toolkit (TNT)*, 1997. http://math.nist.gov/tnt.

[10] R. Pozo and K. Remington, *NIST Sparse BLAS*, 1997. http://math.nist.gov/spblas.