Table of Contents

Improving Memory-System Performance of Sparse Matrix-Vector Multiplication*

Sivan Toledo†

Abstract

Sparse matrix-vector multiplication is an important kernel that often runs inefficiently on superscalar RISC processors. This paper describes techniques that increase instruction-level parallelism and improve performance. The techniques include reordering to reduce cache misses originally due to Das et al., blocking to reduce load instructions, and prefetching to prevent multiple load-store units from stalling simultaneously. The techniques improve performance from about 40 Mflops (on a well-ordered matrix) to over 100 Mflops on a 266 Mflops machine.

1 Introduction

Sparse matrix-vector multiplication is an important computational kernel in many iterative linear solvers. Unfortunately, on many computers this kernel runs slowly relative to other numerical codes, such as dense matrix computations. This extended abstract proposes new techniques for improving the performance of sparse matrix-vector multiplication on superscalar RISC processors. We experimentally analyzes these techniques, as well as techniques that have been proposed by others, to show that they can improve performance by more than a factor of two on many matrices.

Three main factors contribute to the poor performance of sparse matrix-vector multiplication codes on modern superscalar RISC processors, such as the one shown in Figure 1. The code assumes that the matrix is stored in a *compressed-row* format, but the same considerations apply to other storage formats that support general sparsity patterns. The N nonzeros of the n-by$-n$ matrix A are compressed into a single vector `a` using a rowwise ordering, and the column indices of these nonzeros are compressed into an integer vector `colindex`. The vector `rowptr` stores the first index of each row of A in the vectors `a` and `colindex`, and its last element contains $N+1$. The inner loop of the code loads `a(jp)` and `colind(jp)`, loads `x(colind(jp))`, and performs one multiply-add operation. While `a` and `colind` are loaded using a stride-1 access pattern, `x(colind(jp))` may be any element of `x`.

First, lack of data locality causes a large number of cache misses. Typically, accesses to the arrays that represent the sparse matrix A (`a`, `rowptr`, and `colind` in Figure 1) have no temporal locality whatsoever, but have good spatial locality. That is, there is no data reuse, but accesses are in a stride-1 loop. Accesses to the dense vector x being multiplied reuse data, but the access pattern depends on the sparsity structure of A. One technique that can reduce the number of cache misses is to reorder the matrix to reduce the number of cache misses on x. This technique was proposed by Das et al. [5], analyzed in certain cases by Temam and Jalby [9] and further investigated by Burgess and Giles [4]. We study this technique further in this paper, and we also show that the effectiveness of the new techniques that we propose depends on it.

A second factor that limits performance is the tendency of multiple load/store functional units to miss on the same cache line. Many superscalar RISC processors have at least two load-store units. On such processors, when one unit is stalled due to a cache miss, the other unit(s) can continue to load data from the cache. Unfortunately, in stride-1 loops the other units soon try to load data

*Most of this work was done while the author was a postdoctoral fellow at the IBM T. J. Watson Research Center, Yorktown Heights, New York.

†Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304.

```
do i = 1,n
  sum = 0.0D0
  do j = rowptr(i), rowptr(i+1)-1
    sum = sum + a(jp) * x(colind(jp))
  end do
  y(i) = sum
end do
```

FIG. 1. *A sparse matrix-vector multiplication code for matrices stored in a compressed-row format.*

from the cache line that caused the first miss. Consequently, all units are often stalled on the same cache line. The miss is compulsory because the accesses have no temporal locality, so one unit much spend the time waiting for the miss to be serviced. But the misses generated by the other units are not compulsory and we show below how to prevent them using prefetching.

Finally, sparse matrix-vector multiplication codes typically perform a large number of load instruction relative to the number of floating-point operations they perform. This phenomenon is caused by the poor data locality, which makes it difficult to reuse data already in registers, and because the code must load row or column indices in addition to floating-point data. The code in Figure 1, for example, loads three words into registers for each floating-point multiply-add operation. The large number of load instruction places a heavy load on the load/store units that interface the register files to the cache, and on the integer ALU's that compute the addresses to be loaded. (These ALU's are sometimes part of the load/store units and sometimes part of the integer execution units.) On most current processors, these units are often the bottleneck in sparse matrix-vector multiplication. The floating-point units are therefore underutilized. We present below a blocking technique that reduces the number of load instructions. Blocking in sparse matrix-vector multiplication was used in somewhat different forms in [1, 3].

Another inessential but still important factor that limits the performance of the code in Figure 1 is the fact that the column index `colind(jp)` must be converted from an integer index to a byte offset from the beginning of `x`. This conversion, which is required on most processors for indirect addressing, causes the integer ALU's to perform an additional instruction in every iteration.

Although the techniques that we propose can be applied separately, they are most effective when they are combined. In particular, reordering the matrix can enhance or degrade the effect of blocking. Also, without the reduction in the number of cache misses on x that reordering yields, our prefetching technique is ineffective on large matrices.

We have implemented our techniques and evaluated them on an IBM workstation with a POWER2 superscalar RISC processor using a suite of 13 matrices. We describe the techniques in the next section. We summarize our experimental results in Section 3 and present our conclusions in Section 4.

The full paper discusses the applicability of our techniques to other current superscalar RISC processors and presents experimental results on an additional architecture, the Sun UltraSparc I. The full paper also describes more fully the implementation technique that we have used to implement prefetching and proposes a method for hardware assisted prefetching. In addition, the full paper also contains a more complete description of our experimental results, which are only summarized in this extended abstract due to lack of space.

2 Algorithmic Techniques

This section describes the algorithmic techniques that we use to improve the memory-system performance of the sparse matrix-vector multiplication code.

Reducing Cache Misses through Bandwidth Reduction. Das et al. [5] proposed to reorder sparse matrices using a bandwidth-reducing technique in order to reduce the number of cache misses that accesses to `x` generate. Temam and Jalby [9] analyzed the number of cache misses as a function of the bandwidth for certain cache configurations. The technique was investigated further by Burgess

and Giles [4], who extended it to other unstructured-grid computations. Burgess and Giles studied experimentally several reordering strategies, including reverse Cuthill-McKee and a greedy blocking method. The found that reordering improved performance relative to a random ordering, but they did not find a sensitivity to the particular ordering method used.

We have performed additional experiments with a larger set of matrices. Our experiments essentially validates the results of Burgess and Giles. We have found that compared to a random ordering, bandwidth reduction and nested-dissection orderings reduce cache misses and improve performance. Performance can improve by more than a factor of three on large matrices.

When the bandwidth reduction technique is used alone without blocking or prefetching, the particular ordering method does not matter much to the performance of matrix-vector multiplication. Consequently, reordering methods should be selected based on the reordering speed and on the effect on ordering-sensitive preconditioners, such as Incomplete LU. When the bandwidth reduction technique is combined with blocking and prefetching, however, different ordering methods yield different performance results. We have found that in this situation Cuthill-McKee ordering is usually the best choice. It is a fast algorithm and has benefits in preconditioning as well, as explained below in Section 4.

Reducing Load Instructions through Blocking. We reduce the number of load instructions the code performs by splitting the general sparse matrix A into a sum of two or three matrices, some of which are block matrices. It is reasonable to expect the matrix to contain dense blocks, because such matrices arise in many application areas (for example, equations defined on grids with more than one variable per grid point). Multiplying a block sparse matrix by a vector can reduce the number of loads over unblocked multiplication in three different ways. First, blocking enables the code to load fewer row or column indices into registers, because only one index per block is required, rather than one per nonzero. Second, elements of `x` are loaded once and used k times when the matrix is blocked into k-by-l blocks (this is a form of register blocking). Third, since the POWER2 processor has a load-quad instruction that loads two consecutive floating-point doublewords into two registers, 1-by-2 blocks allow the processor to load the two nonzeros in `a` and the two elements of `x` using only two, rather than four, load instructions.

We therefore scan the matrix in a preprocessing phase and extract all the 2-by-2 blocks that we find, and then all the 1-by-2 blocks that we find. The extraction is done in a greedy fashion that extracts all the blocks that we find in a rowwise scanning of the matrix. For 2-by-2 blocks, this greedy algorithms is not always optimal, in the sense that it may find fewer 2-by-2 blocks than possible. The greedy algorithm is optimal for 1-by-2 blocks. To preserve data locality in accesses to `x`, we do not multiply by the 2-by-2 blocks, then 1-by-2 blocks, and then by the remaining unblocked nonzeros. Instead, we process row after row, where in each row i we perform both the multiplication by the unblocked nonzeros in row i and the multiplication with the blocked nonzeros in rows i and $i + 1$.

On processors without a quad-load instruction (practically all other current RISC processors), some of the benefit of 1-by-2 blocks can be realized by replacing them by 2-by-1 blocks. These require only three floating-point loads, because the element of `x` can be loaded once and used twice.

Our approach to blocking is different from previous algorithms, mainly in that our algorithm attempts to find many small completely dense blocks. Other researchers have proposed algorithms that attempt to find larger (and hence fewer) dense blocks and/or blocks that are not completely dense. The full paper contains detailed comparisons between our blocking strategy and those of Agarwal, Gustavson and Zubair [1] and of Balay, Gropp, McInnes and Smith [3].

Prefetching in Irregular Loops. Traditionally, prefetching was considered to be a technique for hiding latency, in the sense that prefetching can prevent memory access latency from degrading performance, as long as memory bandwidth is sufficient to keep the processing units busy (See [2] or [8], for example). In many codes, for example dense matrix multiplication, the ratio of floating-point to load instructions is high. This high ratio allows the algorithm to hide the latency of cache misses by prefetching cache lines early. The fact that the load/store unit is stalled for many cycles is negligible, because there are only few load instruction relative to floating-point instructions.

In matrix-vector multiplication, especially with sparse matrices, the ratio of floating-point to

load instructions is low, below one. The memory bandwidth that is required to keep the processing units is, therefore, high. Since most computers do not have sufficient memory bandwidth, loading data from memory becomes the bottleneck that determines performance in this computation. It is not important here whether the load/store unit stalls early (with prefetching) or late (without), since it is needed all the time.

We can still use prefetching, not to hide latency, but to improve memory bandwidth. As far as we know, this use of prefetching is novel. In particular, we use prefetching to prevent multiple load/store units from stalling on the same cache line. Since most of the cache misses are generated by accesses to two vectors that are accessed in a stride-1 loop, multiple load/store units can miss on the same cache line. One unit misses on the first word in a line and stalls. The second unit tries to load the next word in the vector and stalls too. This can happen even though two vectors are accessed, if the second unit loads a word in cache from the second vector and then misses on the first. When two units stall on the same line, the effective memory-to-cache and cache-to-register bandwidths are reduced. It is possible to avoid this bandwidth reduction using prefetching.

Our strategy is to prefetch elements of `a` and `colind` before they are used. The prefetching instruction always misses and stalls one of the load/store units. The other unit, however, continues without stalling as long as there are no cache misses on `x` and `y`, which are rare once the matrix has been reordered. The details are quite complicated, however, and they are explain in the full paper.

The use of prefetching to prevent multiple load/store units to stall on the same cache line is likely to have a beneficial effect on other RISC processors that have multiple load/store units.

Eliminating Integer to Pointer Conversions. The expression `x(colind(jp))` is compiled into an instruction that loads `colind(jp)` into a register and at least two more instructions that load `x(colind(jp))`. `Colind(jp)` can be loaded in one instruction since `colind` is accessed in a stride-1 loop, so an instruction that loads it and increments the pointer to `colind(jp)` by the size of one integer. Loading `x(colind(jp))` requires two instructions because the instruction that loads a word with a given offset from the beginning of `x` requires a byte offset, not a word offset. `Colind(jp)` must therefore be multiplied by the size of a word in bytes to yield a byte offset.

The multiplication is done by an arithmetic shift instruction that executes in one cycle on one of the integer ALU's. On processors where the integer ALU's also compute the addresses for load instructions, such as IBM's POWER2 and Digital's Alpha 21164, this extra integer instruction places additional overhead on the integer ALU's which are already overloaded with load instructions. We therefore perform a preprocessing phase in which we replace the integer indices with pointers to elements of `x`. We show below that this optimization alone can improve performance by as much as 38%.

3 Experimental Results

In this section we present the results of the extensive experiments we carried out in order to assess the effectiveness of our strategy. We report here on experiments carried out on a superscalar IBM RS/6000 workstation. Some of the details are omitted due to lack of space, as are experiments on another architecture, a Sun UltraSparc I. They are included in the full version of the paper.

The experiments were carried out on an RS/6000 workstation with a 66.5 Mhz POWER2 processor a 256 Kbytes 4-way set-associative data cache, a 256-bit wide memory bus and 512 Mbytes of main memory. The POWER2 processor has 32 architected and 54 physical floating-point registers, two floating-point units, two integer units that also serve as load-store units, and a branch unit. The floating-point units are each capable of executing one multiply-add instruction in every cycle for a peak performance of 266 million floating-point operations per second (Mflops). For data in the cache, the integer units are each capable of loading or storing in one cycle one integer register, one floating-point register, or even two floating-point registers using a so-called quad-load or quad-store instruction.

To put our results in perspective, we note that on this machine the *dense* matrix-vector-multiplication subroutine (DGEMV) in IBM's Engineering and Scientific Subroutine Library, which uses both blocking and prefetching [2], achieves performance of about 170 Mflops when applied to large matrices.

TABLE 1

The characteristics of the test-suite matrices. The Boeing matrices are all stiffness structural engineering matrices from Roger Grimes of Boeing, and the NasGraph matrices are from the 1994 partitioning benchmarks of NASA's Numerical Aerodynamics Simulations Department.

Matrix	Dimension	Nonzeros	Source	Model
bcsstk32	44609	2014701	Boeing	Model of an automobile chassis
msc10848	10848	1229778	Boeing	Test matrix KNUCKLEF8 from MSC/NASTRAN
msc23052	23052	1154814	Boeing	Test matrix SHOCKF8 from MSC/NASTRAN
ct20stif	52329	2698463	Boeing	CT20 engine block
crystk02	13965	968583	Boeing	FEM crystal free vibration
crystk03	24696	1751178	Boeing	FEM crystal free vibration
bcsstk35	30237	1450163	Boeing	Automobile seat frame and body attachment
bcsstk36	23052	1143140	Boeing	Automobile shock absorber assembly
bcsstk37	25503	1140977	Boeing	Track ball
cojack	188384	875446	NasGraph	Cooling water jacket of a BMW engine
hsctl	31736	253816	NasGraph	Large model of a high speed civil transport
pwt	36519	253069	NasGraph	Unstructured grid
rock1	133904	214086	NasGraph	Large model of a rock

We coded most of the algorithms in C and compiled them with options that enables optimizations including loop unrolling and the use of quad-load instructions. We implemented prefetching by slightly modifying the compiler's assembly-language output to introduce prefetching and additional unrolling. We produced and tested ten variants of the matrix-vector multiplication code. The first code code is in C and uses integer column indices and no blocking or prefetching. The other nine all use pointer column indices. These codes test all the combination of blocking (none, 1-by-2, 1-by-2 and 2-by-2) and prefetching (C code with no prefetching, C+assembly code with no prefetching, C+assembly code with prefetching).

The algorithms were tested on a suite of 13 sparse symmetric stiffness matrices, described in Table 1. The matrices are all structurally symmetric, but the code is general and does not exploit symmetry. (A matrix-vector code for symmetric matrices can load fewer coefficients and therefore run more efficiently.) The matrices that were contributed by Roger Grimes of Boeing represent 3-dimensional with several degrees of freedom (variables) per grid point, typically at least 3. They seem to have been ordered using a band or profile ordering algorithm applied to the original grid, so all the degrees of freedom associated with a single grid point remain contiguous. Matrices from the NasGraph collection also represent 3-dimensional models, but they only have one degree of freedom per grid point. As a consequence, they are sparser and do not have contiguous dense blocks. We do not how they were ordered before they were placed in the matrix collection. All the matrices were too large to remain in the cache between multiplications. Since the cache can hold $32,768$ doublewords, in some of the experiments the vector `x` could fit within the cache and remain there between multiplications.

For each matrix, we measured the performance of all combinations of the ten codes and five symmetric orderings. The five orderings are random ordering, a nested-dissection-type ordering (denoted by WGPP), reverse Cuthill-McKee (RCM), Cuthill-McKee (CM), and the original ordering of the matrix as stored in the matrix collection. The WGPP ordering code was written by Anshul Gupta [7].

The Effect of Blocking and Prefetching. Figure 2 shows the performance of the ten codes. The matrices are all ordered in the original ordering from the matrix collections, which proved to be the best or close to best ordering for all of them (see below for more details on the effect of ordering). The most striking feature that emerges from the figure is that the behavior of the nine Boeing matrices is very different from the behavior of the four NasGraph matrices. The difference is mostly due to differences in the sparseness of the matrices.

Each one of the algorithmic improvements that we have introduced boosts performance on the Boeing matrices. Replacing integer indices by pointers increases performance from about 40 Mflops to about 52–55 Mflops. With no blocking, the extra unrolling in the assembly-language code

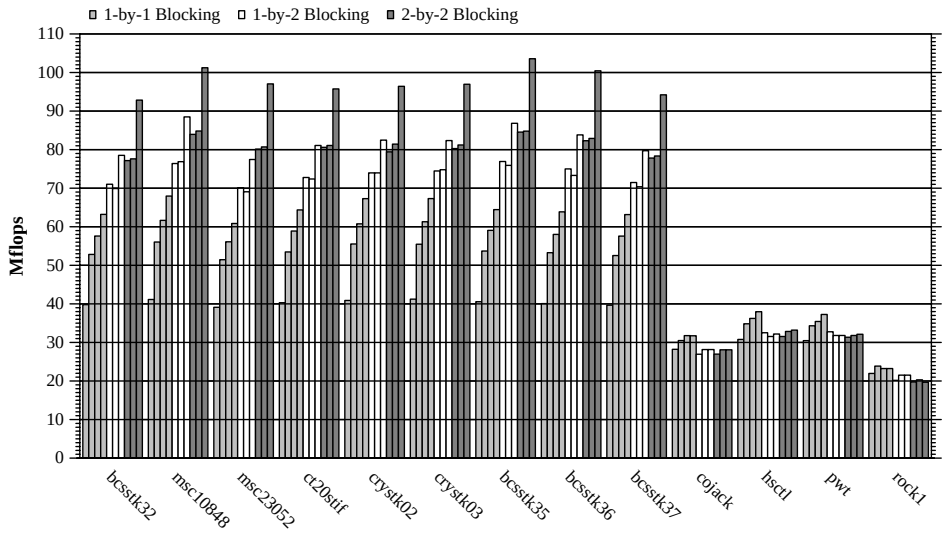

FIG. 2. *The performance in millions of floating-point operations per second (Mflops) of the sparse matrix-vector multiplication codes. The performance on each test matrix is represented by 10 bars, one for each code. The 10 bars are organized into three groups, one for 1-by-1 blocking (i.e., no blocking), one for 1-by-2 blocking, and one for 2-by-2 and 1-by-2 blocking. Each group is represented by a different shade of gray. The leftmost bar with no blocking represents the performance of C code with integer indices. The next three bars, as well as the three bars in the other two groups, represent the performance of C code with pointer indices, assembly language code with pointer indices, and assembly language code with pointer indices and prefetching. The original orderings of the matrices in the matrix collections are used.*

improves performance by 5–6 additional Mflops. Prefetching improves performance by another 5–6 Mflops. With blocking, there is essentially no difference between C and assembly-language versions. The 1-by-2 blocking with no prefetching improves performance to 70–77 Mflops, and prefetching combined with 1-by-2 blocking boosts performance to 78-98 Mflops. The 2-by-2 blocking with no prefetching gives similar performance. Finally, 2-by-2 blocking combined with prefetching yields performance of 93–104 Mflops.

On the NasGraph matrices, using pointers, unrolling, and prefetching all help, but blocking does not improve performance. In fact, in most cases blocking degrades performance. On these matrices, the fastest code yields performance of only 23–39 Mflops, and the differences between the different codes are smaller than the differences on the Boeing matrices. Since the NasGraph matrices are much sparser than the Boeing matrices, blocking the multiplication code introduces overhead without delivering a significant benefit. The most likely reason that the NasGraph matrices do not benefit from blocking is that they represent models with only one degree of freedom per grid point. As a consequence, the underlying graphs have only few cliques, so the number of dense blocks is small no matter how the matrices are ordered.

The Effect of Ordering. In all cases the performance on ordered matrices is better than on randomly permuted matrices. The differences are especially large on large matrices, such as bcsstk32, ct20stif, cojack, and rock1. When these matrices are randomly ordered, the performance ranges from about 10 to 20 Mflops, and neither blocking nor prefetching improves performance. The performance on bcsstk32 and ct20stif improves to almost 60 Mflops when they are reordered, even without blocking and prefetching. As shown before, blocking and prefetching improves performance further to about 95 Mflops. Ordering improves the performance on the sparser cojack and rock1 from less than 10 Mflops to about 25–30 Mflops, but blocking and prefetching do not help.

Without blocking, the various ordering methods yield roughly the same performance. With blocking, different ordering methods yield different performance levels. The random ordering is still always worst, followed by the RCM ordering, by the WGPP ordering, the CM ordering, and finally the original ordering.

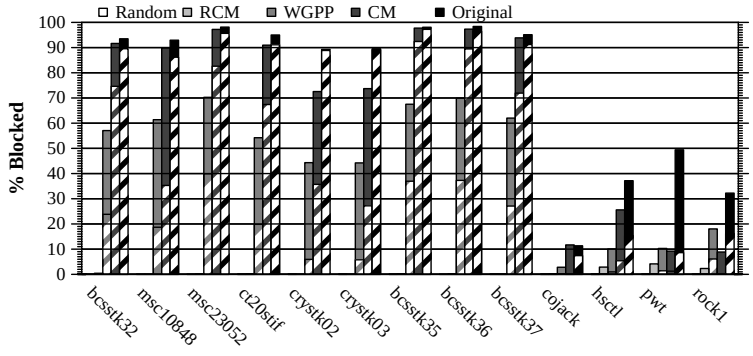

FIG. 3. *The fraction of matrix nonzeros that is blocked in 2-by-2 and 1-by-2 blocks. The graph shows the fraction of nonzeros in blocks for five different orderings of each matrix (the first two usually result in very low levels of blocking). The striped portion of each bar represents the fraction of nonzeros in 2-by-2 blocks, and the solid portion the fraction of the remaining nonzeros that are blocked in 1-by-2 blocks. The levels of blocking when only 1-by-2 blocks are used are slightly higher than the combined levels shown here.*

Figure 3 explains the variations in the performance of the blocked codes with various ordering methods. We analyze mostly the denser Boeing matrices, because the performance impact of blocking on the sparser NasGraph matrices is marginal. The figure shows that the random and reverse-Cuthill-McKee orderings result in matrices with no or very few 2-by-2 and 1-by-2 blocks. Therefore, using a blocked code with a randomly permuted matrix or a matrix which has been RCM ordered does not improve performance. The WGPP ordering creates more blocks. In the Boeing matrices, 44–70% of the nonzeros are in blocks. Usually most of them are in the smaller 1-by-2 blocks. The Cuthill-McKee orderings results in many more nonzeros in blocks, between 73 and 97%. The fraction that is in 2-by-2 blocks is better than with the WGPP ordering, but it is still sometimes less than one half. In the original ordering, even more nonzeros are in blocks, and the vast majority of them are in 2-by-2 blocks.

The differences between the fractions of nonzero blocked in the different orderings can be traced to several factors. The original ordering yields better blocking than all the other orderings probably since it was applied to grid points rather than individual degrees of freedom (variables). Consequently, dense blocks in the matrix that was produced by the grid generator remained dense. Since we applied the other orderings to the matrices, some of these dense blocks disappeared. The differences between the Cuthill-McKee and the reverse-Cuthill-McKee may be due to the fact that our blocking algorithm is greedy and scans the matrix from top to bottom and from left to right within rows.

The Cost of Reordering and Blocking. The Cuthill-McKee and reverse Cuthill-McKee orderings take a factor of between 1 and 3 more than the basic matrix-vector multiplication time with a randomly permuted matrix. They are well worth their cost. The WGPP ordering costs a factor of between 20 and 200 more than matrix-vector vector multiplication. (This ordering code was designed as a fill-reducing mechanism for direct factorizations, which are much more expensive than single matrix-vector multiplications). It is therefore not appropriate for our application.

Blocking usually costs between 4 and 15 times the cost of basic matrix-vector multiplications. A rough calculation shows that if the matrix is used in more than 75 multiplications, blocking reduces the overall running time.

4 Conclusions

We have presented four techniques for accelerating sparse matrix-vector multiplication on superscalar RISC processors. One of the techniques, precomputing addresses for indirect addressing, is trivial but important. The technique of reordering the matrix to reduce its bandwidth and hence reduce cache misses has been proposed by Das et al. [5] and investigated further in two other papers [4, 9]. We have explored it further and found that the Cuthill-McKee yields excellent results

on a variety of matrices. Two other techniques, representing nonzeros in small dense blocks and prefetching to allow cache hits-under-miss processing are novel (others have proposed to represent nonzeros in larger blocks [1, 3]). They, too, improve performance significantly on many matrices. On an IBM workstation, the combined effect of the four techniques can improve performance from about 15 Mflops to over 95 Mflops depending on the size and sparseness of the matrix. Although even basic matrix-vector multiplication codes perform better when the matrices are not extremely sparse (< 10 nonzeros per row), highly optimized codes are even more sensitive to the sparseness of the matrices.

We believe that our techniques would improve performance on other superscalar RISC processors with similar architectural features. This belief has been partially validated by experiments on a Sun UltraSparc I workstations, which are reported in the full paper. The techniques, with the exception of the technique we use to implement prefetching, are portable.

Reordering sparse matrices using the Cuthill-McKee ordering has another benefit in sparse iterative solvers. When a conjugate gradient solver uses an incomplete Cholesky preconditioner, the ordering of the matrix effects the convergence rate. Duff and Meurant [6] compared the convergence rate of incomplete-Cholesky-preconditioned conjugate gradient with 17 different orderings on 4 model problems. In their tests the Cuthill-McKee and the reverse Cuthill-McKee resulted convergence rates that were best or close to best. Although it is possible to use different orderings for preconditioning and matrix-vector multiplication, doing so requires permuting a vector twice in every iteration. This extra step renders each iteration more expensive. Using a Cuthill-McKee or similar ordering for both steps eliminates the need to permute vectors in every iteration, it leads to few cache misses in the matrix-vector multiplication step (and very likely in the preconditioning step as well), it enables blocking, and it accelerates convergence.

Acknowledgments. Fred Gustavson, Bowen Alpern, and Dave Burgess read and commented on early versions of this paper. Their comments helped improve the paper considerably. Thanks to Anshul Gupta for assistance with the use of his matrix reordering package, WGPP. Thanks to Ramesh Agarwal for stimulating discussions.

References

[1] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *A high performance algorithm using pre-processing for sparse matrix-vector multiplication*, in Proceedings of Supercomputing '92, Nov. 1992, pp. 32–41.

[2] R. C. Agarwal, F. G. Gustavson, and M. Zubair, *Improving performance of linear algebra algorithms for dense matrices using algorithmic prefetch*, IBM Journal of Research and Development, 38 (1994), pp. 265–275.

[3] S. Balay, W. Gropp, L. C. McInnes, and B. Smith, *PETSc 2.0 users manual*, Tech. Rep. ANL-95/11, Revision 2.0.15, Argonne National Laboratory, 1996.

[4] D. A. Burgess and M. B. Giles, *Renumbering unstructured grids to improve the performance of codes on hierarchical memory machines*, Tech. Rep. 95/06, Numerical Analysis Group, Oxford University Computing Laboratory, May 1995.

[5] R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, *The design and implementation of a parallel unstructured Euler solver using software primitives*, AIAA Journal, 32 (1994), pp. 489–496.

[6] I. S. Duff and G. Meurant, *The effect of ordering on preconditioned conjugate gradient*, BIT, 29 (1989), pp. 635–657.

[7] A. Gupta, *WGPP: Watson graph partitioning (and sparse matrix ordering) package*, Tech. Rep. RC20453, IBM T.J. Watson Research Center, Yorktown Heights, NY, May 1996.

[8] A. K. Porterfield, *Software Methods for Improvement of Cache Performance on Supercomputer Applications*, PhD thesis, Rice University, May 1989.

[9] O. Temam and W. Jalby, *Characterizing the behavior of sparse algorithms on caches*, in Proceedings of Supercomputing '92, Nov. 1992.