CACHING–EFFICIENT MULTITHREADED FAST MULTIPLICATION OF SPARSE MATRICES

Peter D. Sulatycke and Kanad Ghose
Department of Computer Science
State University of New York, Binghamton, NY 13902–6000
{sulat, ghose}@cs.binghamton.edu

Abstract

Several fast sequential algorithms have been proposed in the past to multiply sparse matrices. These algorithms do not explicitly address the impact of caching on performance. We show that a rather simple sequential cache–efficient algorithm provides significantly better performance than existing algorithms for sparse matrix multiplication. We then describe a multithreaded implementation of this simple algorithm and show that its performance scales well with the number of threads and CPUs. For 10% sparse, 500 X 500 matrices, the multithreaded version running on 4–CPU systems provides more than a 41.1–fold speed increase over the well–known BLAS routine and a 14.6 fold and 44.6–fold speed increase over two other recent techniques for fast sparse matrix multiplication, both of which are relatively difficult to parallelize efficiently.

Keywords: sparse matrix multiplication, caching, loop interchanging

1. Introduction

The need to efficiently multiply two sparse matrices is critical to many numerical applications that require solutions of large linear systems. These include climate modeling, molecular dynamics, CFD solvers etc. [CuWi 85], [GoVL 89], [PTVF 92]. The multiplication of sparse matrices presents some interesting challenges for optimizing the overall performance. In the past the main goals were to optimize the storage requirements and to cut down the number of unnecessary multiplications. Today, with cache miss latencies dominating the latencies of operations by almost an order of magnitude (e.g., 3 cycle latency for a fused multiply–and–add instruction vs. 20+ cycles for handling a cache miss), the optimization goals must additionally target a reduction of the overall number of cache misses. In this paper, we show how simple loop interchange based modifications of the classical matrix multiplication loops can promote better caching leading to sparse matrix multiplication times that are significantly better than the times obtained from some relatively recent techniques for sparse matrix multiplication. (Note that the sparse matrix multiplication problem is different from the problem of multiplying a sparse matrix with a vector; caching efficient solutions are known for the latter.) We then proceed on to parallelize our sequential version for running on a small scale symmetric multiprocessor. The conversion of the sequential, cache–efficient version into a parallel multithreaded version is extremely straightforward and the multithreaded version requires no data synchronization across the threads.

The naive technique for multiplying two 2–dimensional N X N matrices, A and B, whether they are sparse or non–sparse is to use a triply–nested loop:

```
for (i = 0; i++; i < N)
    for (j = 0; j++; j < N)
        for (k = 0; k++; k < N)
            C[i,j] += A[i,k]* B[k,j];  (L1)
```

If this code is applied to sparse matrices, two inefficiencies result. First, zero–valued elements are accessed and second, a multiplication is performed regardless of the fact that one or both of the multiplicands is a a zero–valued element.

The first inefficiency has been generally addressed by using compact data structures to store only the non–zero elements of the matrices to be multiplied. In [FAC 89], several such data structures are compared in some depth. In [Gus 78], a sparse row–wise representation of sparse matrices is used, together with a scheme for eliminating a multiplication when one of the elements being multiplied was a one or a zero.

In [PDZ 92], a compact representation of sparse matrix and a multiplication algorithm based on this representation is suggested. This algorithm (hereafter called the Park algorithm for convenience) was specifically designed to handle banded and triangular matrices while being space efficient. The algorithm begins by converting both matrix A and B into a compressed form that keeps track of continuous non–zero row segments of the matrices. The data structure stores all non–zero elements in one array and then the row number, starting column and ending column of all segments in another array. If the segments are long then the amount of space used by the data structure is kept to a minimum. The next step in the algorithm is to transpose matrix B. Now the dot product between each row of matrix A and every row of matrix B may be efficiently calculated. This is done by only multiplying together the parts of the segments from matrix A and B that overlap. This is analogous to ANDing two bit vectors together. The calculation of overlap involves a considerable amount of conditional logic. Finally the resulting product is then written into a likewise compressed matrix C. This algorithm speeds up sparse matrix multiplication of banded matrices because of two reasons: only non–zero elements are multiplied and cache misses are kept to a minimum within the bands. When the input matrices are not banded the size of overlapping segments is reduced and cache misses increase accordingly. This results in poor computational speed for non–banded matrices. In [PDZ 92], a speedup by a factor of up to 40 is reported for this algorithm over a naive implementation (which used the basic loop L1 described in Section 1).

Another fast algorithm for multiplying sparse matrices is described in the widely used Numerical Recipes text [PTVF 92]. The basic idea behind this algorithm is quite similar to the one in [PDZ 92]. Here both matrix A and B are compressed, matrix B is transposed and dot products are taken of the corresponding rows. What differs in the two algorithms is the data structures used and how this influences the calculation of the dot products. In the Numerical Recipes algorithm the data structure used does not store segments but instead stores all the column numbers of non–zero elements, along with the data values. This data structure allows the dot product to be calculated at the individual element level instead of the segment level. Each column number from a row of matrix A is individually compared against the current column number from a row of matrix B. Depending on the result of the comparison the current column numbers of matrix A and/or B are incremented. This is very much like the process of merging two arrays together. This algorithm results in the same cache misses to the data values as in the Park et al. algorithm due to similar access patterns. What differs is the sensitivity of the comparison code to the length and number of segments. The Park algorithm is optimized for a few long segments while the Numerical Recipes algorithm is less sensitive to segment number and length.

Generally, in most of these techniques, no attention has been paid on the impact of caching on the performance of sparse matrix multiplication techniques. Exceptions to this are all concerned with the multiplication of a sparse matrix with a vector – these techniques cannot, however, be generalized easily to handle the multiplication of two sparse matrices. The sparse matrix–vector multiplication technique proposed by Toledo [Tol 96] uses cache access reordering, inspired by the work of Das et al in [DMS+ 94] to speed up matrix–vector multiplication. Another work along similar lines is reported in [TeJa 92]. Here, the cache access pattern for a direct–mapped cache during the multiplication of a sparse matrix and a vector is modeled and a blocking technique that blocks the diagonal elements of a banded sparse matrix is suggested as a means for promoting efficient caching. The matrix multiplication routine in the BLAS package [DDDH 90] uses blocking to promote the use of caches; it is basically designed to handle non–sparse matrices. The need to optimize the use of the cache and general guidelines for doing so have been described in [LeWo 94], but to date these techniques have not been used in published literature to speed up the multiplication of sparse matrices and design multi–threaded applications for the same purpose.

2. Caching Issues in Matrix Multiplication

Consider the normal 3–nested loop (L1) for multiplying two matrices. Assume that a row–order allocation scheme is used for the arrays. In this code, the elements of C are accessed row by row, with a stride of one within each row. The access patterns for the elements of A and B are different. For a given (i, j) pair, accesses to A are made with a stride of one within the i–th row, while accesses to B are made in the j–th column, with a stride of one in column order. This corresponds to a stride of N in the row–order layout of B. The access patterns for the three matrices are summarized in the following:

a) Each element of C is accessed N times consecutively. Furthermore, the elements are accessed in row order with a stride of one.

b) Each row of A is accessed N times successively in row order with a stride of one. The rows are visited in order.

c) Each column of B is visited N times in succession: this amounts to accessing the columns with a stride of N in the row order memory layout. The columns of A are visited in succession.

Consider now the execution of this code on a processor with a set–associative cache with a line size of L (in bytes). The number of consecutive elements of the arrays that will fit into a cache line is thus L/w, where w is the size of each element of the arrays. When row order accesses are made with unit stride, in the worst case, the first access to a row element within a cache line will result in a miss, while the remaining (L/w – 1) accesses within the line will result in cache hits. In contrast, assuming N > L/w (i.e., N consecutive elements in row order do not fit into a cache line), accesses to elements with a stride of N will all result in misses in the worst case. If we assume that the cache access time on a hit or to detect a miss is t_c and miss service time to be t_m, the total time in accessing the arrays during the execution of the code above is:

$$T_{access} = 2 * N^2 * (t_c + w/L * t_m)$$

(for C, assuming C[i,j] is held in a register for each inner loop)
+ $N^2 * (N * t_c + w/L * t_m)$
(for A, making the optimistic assumption that a row of A remains cached after the first access to the row)
+ $N^3 * (t_c + t_m)$
(for B, assuming that a column does not remain cached as we iterate over the same column, since the access stride is N >> 1)

If we assume that $N = 500$, $t_c = 2$ cycles, $t_m = 20$ cycles, $L = 32$ bytes and $w = 8$ bytes (64–bit float), all of which are typical, we have:

$T_{access} =$ $3.5 * 10^6 + 251.2 * 10^6 + 2750 * 10^6$ cycles $= 3004.7 * 10^6$ cycles

In contrast, assuming a fused multiply–and–add function unit, that has a latency of 4 cycles (a typical value) for implementing the operations in the inner loop, the total latency of the multiply and add operations is:

$T_{op} =$ $N^3 * t_{mpy\&add} = 500 * 10^6$ cycles

Note that the data access time is dominated by the miss handling times. As the contemporary trend of the CPU cycle time decreasing at a faster rate than the cache miss handling time continues to grow, the execution time of the above code fragment will become increasingly dominated by the cache miss handling time.

If the simple loop L1 is used for multiplying sparse matrices, the value of T_{op} *for useful operations* becomes smaller, while the access times remains the same (a data element needs to be accessed whether it has a zero value or not), making efficient cache accessing more critical to sparse matrix multiplication.

3. Variations of the Standard Matrix Multiplication for Sparse Matrices

In our variations, we use a compressed version of the two sparse input matrices to avoid unnecessary multiplication and to exploit caching more effectively. For each sparse matrix, three distinct one–dimensional arrays are used to store the following information:

(a) An array of the non–zero data values in row–major order (i.e., row by row, the non–zero data values for row #1, followed by the non–zero data values for row #2 etc.)

(b) An array of the column indices for the non–zero elements, again in row major order (i.e., the column numbers of the non–zero elements of row #1, followed by the column numbers of the non–zero elements of row #2 etc.)

(c) An array of the counts of the number of non–zero elements in a row on a row by row basis.

The use of these three distinct arrays improves the chances of keeping the information necessary to access an array element in distinct cache lines. The type definition for the compressed form in C is:

```
typedef {   float   *data;
            int     *index;
            int     *len;
    /* num. of nonzero elements in a row */
        } compressed;
```

In our first variation (called *IKJ*) of the standard matrix multiplication technique, we first interchange the two inner loop (iterating on j and k, respectively) of the original loop (L1) to get:

```
for (i = 0; i++; i < N)
    for (k = 0; k++; k < N)
        for (j = 0; j++; j < N)
            C[i,j] += A[i,k] * B[k,j]; (L2)
```

Lets now examine how the cache is exercised during the execution of the above code. First note that successive writes to the result array (C) are confined in row order, the same order in which the arrays are laid out in memory. As the inner loops (k and j) execute, repeated accesses will be made to the *row* of C being produced, likely resulting in cache hits. These two factors increase the chances of cache hits on accessing the array C. Second, note that the accesses to the elements of array A are made in row order and with unit stride, but A is sparse, so compressing A makes efficient use of the cache. The particular looping structure used for this variation requires a row order traversal of B, with unit stride *within the compressed row order* layout, again improving the chance of cache hits. Furthermore. the elements of B are used in row order. (Unlike A[i, k], the values of the elements of B used during the j–iteration change from one iteration to the next.) The C–like code is as follows:

```
for (i=0; i< matrixa–>row; i++) {
    indexbc = 0;
    /* index of matrix B compressed */
    for (k=0; k< matrixa–>col; k++) {
        indexa = i*matrixa–>col + k;
        /* index of matrix A */
        len = matrixbc–>len[k];
        /* number of non–zero elements */
        if(matrixa–>data[indexa]!= 0) {
            for (j=0; j< len; j++) {
            indexc =
            matrixbc–>index[indexbc+ j];
            /* row of c determined by row
            of a, col of c determined by
            col of b */
            matrixc–>data[i*matrixc–>col
                + indexc] +=
            matrixa–>data[indexa]*
            matrixbc–>data[indexbc + j];
            }
        } /*endif not zero */
        indexbc += len;
    }}
```

Our next variation (called *KIJ*) is again quite similar to the previous variation. In this case, we interchange the outermost and innermost loops of L1 to get:

```
for (k = 0; k++; k < N)
    for (i = 0; i++; i < N)
        for (j = 0; j++; j < N)
            C[i,j] += A[i,k] * B[k,j]; (L3)
```

For reasons similar to that for the IKJ version, A and B are both compressed. Notice first that accesses to A are *not* made in row order thus making little or no use of caching. Transposing A will remedy this situation and it was observed that the overhead for transposing A generally balances its benefits. The access patterns for A (transposed) and B are similar to that of IKJ. Unlike IKJ, this variation accesses the elements of C rather inefficiently. For every value of the iterator k, the C[i, j] elements are accessed repeatedly row by row. The chances of finding a C[i, j] in the cache (for a given i, j pair) from one value of k to the next is quite small, so caching is not well exploited in accessing C.

Three other loop–interchanged versions of the original loop L1 are possible, but it can be seen that these variations do not exploit caches effectively, so they are not discussed in this paper.

We implemented our two variations (hereafter called *method IKJ*, *method KIJ*) of the standard matrix multiplication loop on a Sun SPARC 20, with 50 MHz. CPUs (using gcc with –o3 option) and compared them against some of the recent techniques for sparse matrix multiplication, namely the Park technique [PDZ 92], the BLAS routine [DDDH 90] and the algorithm from the often used Numerical Recipes text [PTVF 92]. Our main reason for doing so was to identify a good sequential algorithm that is amenable to an efficient multithreaded implementation. We used pseudo–random generators to create a randomly sparse or banded sparse input matrices. All matrices, other than the ones used in the plots for execution time vs. sparsity used a density (or, equivalently, a sparsity) of 10%. For the banded sparse matrices, the width of the band was limited to 1/6–th of the width of the matrices, with elements placed randomly around the diagonal. Our approach was to take the average of six runs on a given set of inputs. Code was used to flush the caches in–between runs.

Figure 1 depicts the average execution time of the various techniques being compared against the value of N, where each of the input A and B sparse matrices are N X N. As observed in Figure 1, for randomly sparse matrices, methods IKJ and KIJ produce significant speedups over all other methods and over all matrix sizes. Method IKJ completes 138 times faster than the naive algorithm (the L1 loop) for 100x100 matrices and 254 times faster for 1000x1000 matrices. (The execution times for the naive algorithm are not depicted in the graphs.) Compared to its nearest competitor, the Numerical Recipes algorithm, method IKJ produces speedups of 4.8 to 6.1 over the same range. This increase in speedup with the increase in matrix size is due to the fact that method IKJ and KIJ produce less cache misses for the same increase in size compared to the other methods.

Figure 1 also depicts the execution time versus N for banded sparse matrices. Method IKJ still outperforms the other techniques. As expected the Park algorithm performs much better than in the randomly scattered data case. This is caused by the fact that banded data will produce a small number of relatively large data segments. Another interesting feature – not easily observable from this graph – is the improved performance of the other methods relative to method IKJ. This was caused by the reduced number of cache misses resulting from the more tightly packed banded data. Method IKJ does not benefit from this because it has already optimized cache access.

The main conclusion from Figure 1 is that method IKJ provides the best performance among all the sequential techniques compared here, irrespective of the nature of the distribution of the elements in the sparse matrices. Although not shown separately in the graphs (but included as part of the execution time shown), the time needed to compress matrices A and B was a small fraction of the total execution time, in the range of 3% to 5%. *The transpose times for the Park algorithm and the Numerical recipes algorithm are* ***not*** *part of the execution times shown.*

Figure 2 depicts the variation of execution time with the sparsity of a 500 X 500 randomly scattered sparse matrices. As observed in Figure 2, methods IKJ and KIJ produce significant speedups over all other methods and over all matrix densities. Method IKJ produces speedups that range from 1086 to 103 times the naive algorithm over the 3–15% density range. Over the same density range method IKJ even performs 9 to 4 times faster than the method from Numerical Recipes. As expected the speedup advantage that method IKJ has over all other methods decreases as density increases. This is caused by the fact that denser data will produce less cache misses for the other methods.

Figure 2 also depicts the variation of execution time with the sparsity of a 500 X 500 banded sparse matrices. (Note that the density within the band is 6 times the density shown on the horizontal axis, since the band is 1/6–th of the matrix width.) This graph is notable for the marked improvement in the Park algorithm as density increases and segments become longer. In fact, when the band is at 90% density, method IKJ only outperforms the Park algorithm by 15%. As expected all other algorithms show some improvement over their execution of random data.

The results from Figures 1 and 2 clearly indicate that method IJK is the sequential algorithm of choice for implementing the multiplication of sparse matrices – banded or randomly

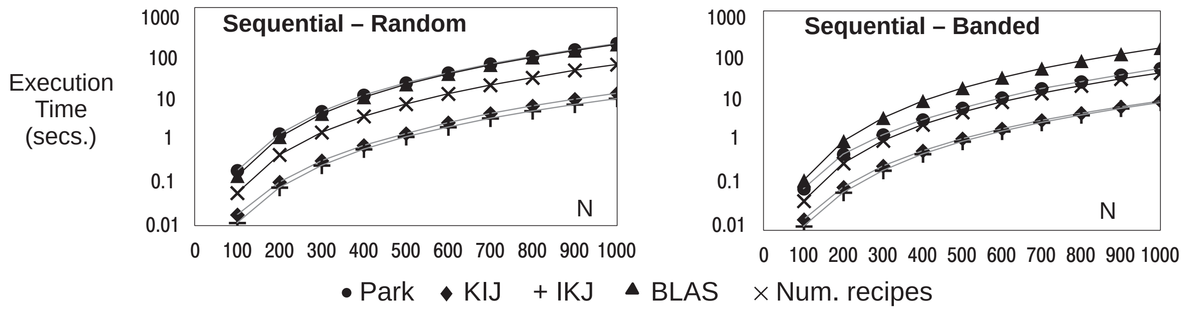

Figure 1: Execution Time versus Sparse Matrix Size for (a) Randomly Scattered Elements and (b) Banded Matrix

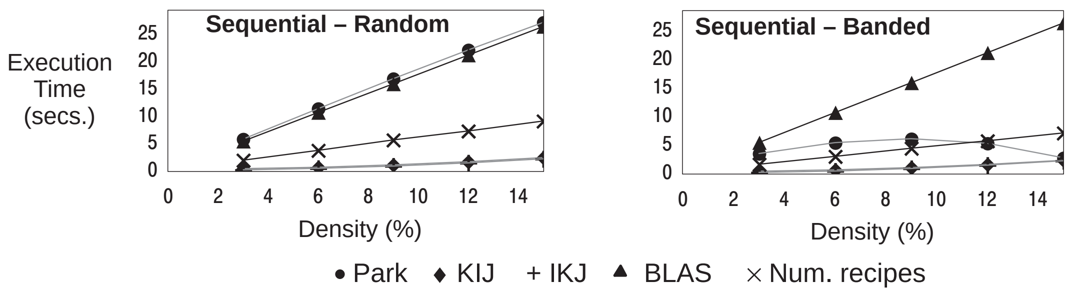

Figure 2: Execution Time vs. Sparse Matrix Density for 500 X 500 (a) Randomly Scattered and (b) Banded Sparse Matrices

scattered. In the next section we show how the IJK method can be easily parallelized into multithreaded code.

4. Parallel IKJ Method

The parallel versions of the IKJ method are implemented using the Sun Solaris threads on SPARC 20 multiprocessing workstations. Each CPU module within the SPARC 20s have a superscalar CPU with on–chip L1 caches and an external L2 cache. Up to 4 such CPU modules are available within a SPARC 20. We implemented several variations, changing the nature of load distribution (static vs. dynamic) and inter–thread synchronization. In all of these variations, *num_threads* refers to the number of computation threads used. A single thread was used to compress the input matrices, since the compression time is only 3% to 5% of the sequential execution time. Of course, with an increase in the number of threads, a parallel implementation of the compression is likely to be beneficial.

We now discuss the parallel implementation of the IKJ method. The data access pattern of this particular method makes it extremely easy to parallelize. Accesses to the input arrays need not be synchronized; rows from these matrices can be cached without the fear of any invalidation in the L2 caches. Most importantly, method IKJ produces a row of the result matrix C for each row of A used. Thus, if the unit of work assigned to a thread is a row of A, no synchronization is necessary for the writes to C from the various threads. This also means that the writes from the threads do not cause any invalidations of the L2 caches of the CPUs, since no processor caches a row of C that is being generated by another CPU – delays due to invalidations are thus absent.

Our first multithreaded implementation of method IKJ (called *PIKJ_Slabs*) uses a static allocation of the rows of matrix A (compressed) to independent threads. Here, each thread is statically allocated {N/num_threads} consecutive rows, where N is the number of rows in A (or B, and also equal to the number of columns).

Our second multithreaded implementation of method IKJ (called *PIKJ_RRobin*) uses a different form of load balancing than PIKJ_Slabs. Here thread #n is assigned rows with numbers of the form (n + m * num_threads), where m equals 0, 1, 2, 3, ... (N/num_threads – 1). The main reason for this particular static allocation is to achieve better load balancing among the threads when the sparse data happen to be skewed from an ideal random distribution.

The third parallel implementation of method IKJ uses dynamic load balancing (called *PIKJ_Dyn*). Here a mutex protected counter is used to allocate one row of matrix A at a time to each thread. Again, no synchronization is needed for accessing the matrices A, B or C. Dynamic load balancing is expected to produce better execution times since the threads are better load balanced, but this comes at the cost of the synchronization overhead over the lock protecting the row counter. We now present our experimental results for the parallel implementations of the IKJ technique. Our basic technique for generating the matrices and doing the runs remains the same as for the sequential versions – we take the average of 6 runs.

Figure 3 depicts how the execution time of the parallel implementations change with the size of the N X N matrices for four threads, one thread running on each CPU, for the randomly scattered and banded sparse matrices with a density of 10%. The reason why the dynamically load balanced version (*PIKJ_Dyn*) does better than the others is a little subtle. The threads are created sequentially with a time lag between the creation of the consecutive threads. As a result, in the statically load–assigned versions (*PIKJ_Slabs*, *PIKJ_RRobin*), the first thread finishes up earlier and waits for the last thread to finish (assuming that all threads have roughly the same amount of work to do), because of the random distribution of the sparse elements. In contrast, in the dynamically load balanced version, if any thread finishes earlier, it will try to get a new chunk of work, which eventually causes the overall execution time to go down. For the randomly scattered sparse matrices, *PIKJ_RRobin* does slightly better than *PIKJ_Slabs* – possibly due to some skewing in the generated data (due to the use of the less–than–ideal pseudo–random generators for generating the indices of non–zero elements). For the banded matrices, *PIKJ_RRobin* does relatively better than *PIKJ_Slabs* (compared to the randomly scattered case). This is due to the fact that the first few rows and the last few rows of the banded sparse matrices have fewer elements than the other rows. Consequently, *PIKJ_RRobin* does a better job at load balancing than *PIKJ_Slabs*, leading to better execution times. Although not clear from Figure 3, on a case by case basis, the execution times for the banded matrices are slightly lower than the execution times of the randomly scattered case – this is a consequence of the fact that better cache performance is obtained in the banded case, where the data density within the bands are effectively 6 times of the density within the entire matrix in the randomly scattered case.

Figure 4 shows how the parallel execution time of the multithreaded implementations of method IKJ (PIKJ_*) varies with the density for 500 X 500 randomly scattered and banded sparse matrices. Here, the trends are similar to the sequential versions. The main implications of these graphs is that the execution times of the parallel implementations go up fairly linearly with the densities, as expected. The difference in execution times among the three variations in the banded and randomly scattered cases are explained in the same manner as for the graphs of Figure 3.

Figure 5 depicts how the execution time of the parallel implementations of method IKJ varies with the number of CPUs. In this case, 10% sparse, 500 X 500 matrices are used for the inputs, with the elements randomly scattered. The less–than linear speedup can be attributed to at least three factors: to the finite time needed to create the threads sequentially; to the time needed to compress matrix B sequentially and to possible cache conflicts. Nevertheless, the 4–CPU performance represents a 41.1 fold improvement over BLAS, a 44.6–fold improvement over Park's method, and a 14.6–fold speedup over the Numerical Recipes algorithm, both of which are difficult to parallelize efficiently.

Figure 5 also depicts how the execution time for multiplying two 10% sparse matrices with randomly scattered elements scale with the number of threads on a 4 CPU system. The small decrease in execution times when we move from 4 threads to 8 threads is probably due to better load balancing resulting from a smaller chunk of work being assigned to each thread. With further increase in the number of threads, the parallel execution time goes up slowly, possibly due to the scheduling overhead of threads. The main inference to be drawn from Figure 5 is that all the parallel implementation of the IKJ method should scale well with the number of threads and can potentially provide better performance on systems with more than 4 CPUs.

5. Conclusions

As the CPU clock rates increase in relation to the cache miss handling latency, the execution time of many data intensive applications become dominated by the cache miss handling time. Consequently, algorithms that tend to exploit caching more effectively are needed to handle this trend. We showed that the use of simple loop interchanging and a compressed format for representing sparse data items speeds up the sequential execution time (which *includes the array compression time*) of the simple matrix multiplication loop drastically, making it outperform some fairly recent sequential algorithms for sparse matrix multiplication. This simple algorithm (the IKJ method) makes more efficient use of caching and can be parallelized rather efficiently. Our experimental results show that for 10% sparse, 500 X 500 matrices, the multithreaded version running on 4–CPU systems provides more than a 545–fold speed increase over the naive sequential version and a 41.1 fold speedup over BLAS. A 14.6 fold and 44.6–fold speed increase are also achieved against two recent techniques for fast sparse matrix multiplication. We expect that the speedups achieved by our techniques to increase as the relative cache miss handling times increase in future platforms. The IKJ method thus represents a very attractive way of implementing sparse matrix multiplication on small scale symmetric multiprocessors.

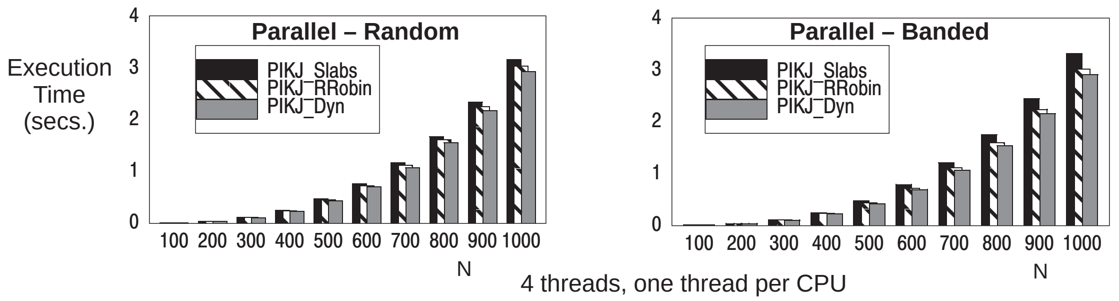

Figure 3: Parallel Execution Time versus Sparse Matrix Size for (a) Randomly Scattered Elements and (b) Banded Sparse Matrices

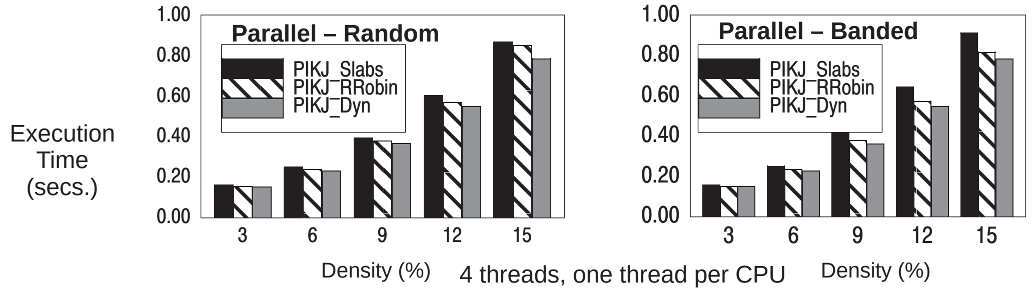

Figure 4: Parallel Execution Time versus Sparse Matrix Density 500 X 500 for (a) Randomly Scattered and (b) Banded Sparse Matrices

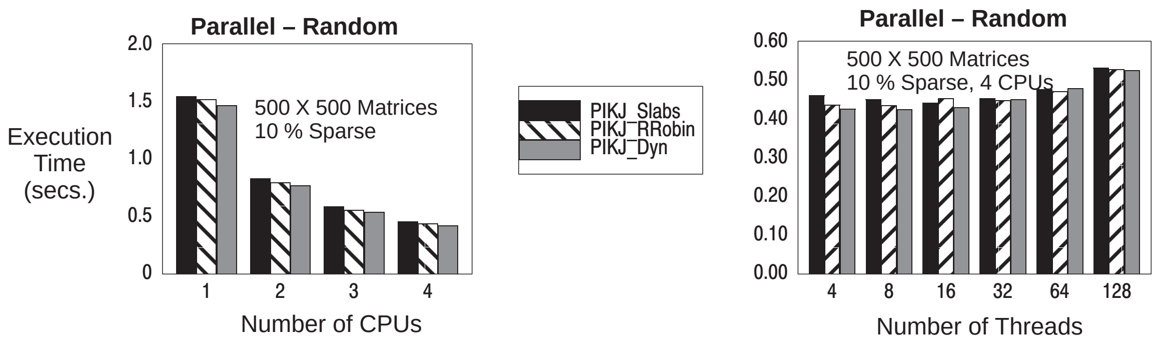

Figure 5: Variation in Parallel Execution Time with the Number of CPUs and Threads

References

[CuWi 85] J.K. Cullum, R.A. Willoughby. Lanczos, “Algorithms for Large Symmetric Eignenvalue Computations”. Vol. 1 (birkhauser, Boston 1985).

[DDDH 90] Dongarra, J.J., Du Croz, J., Duff, I.S. and Hammarling, S., “A set of Level 3 Basic Linear Algebra Subprograms”, ACM Trans. Math. Software, Vol. 16 (Algorithm 679), pp. 1–28.

[DMS+ 94] R.Das, D.J. Mavriplis, J. Saltz, S. Gupta, R. Ponnusamy. “The Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives”. AIAA Journal, 32(3): 489–496, 1994

[FAC 89] P. Felice, A. Agnifili, E. Clementini. “Data Structures for Compact Sparse Matrices Representation”. Advanced Engineering Software, Vol. 11, No. 2, (1989) 75–83

[GoVL 89] G.H. Golub, C.F. Van Loan. *Matrix Computations*. 2nd ed. (Johns Hopkins Univ. Press, Baltimore, 1989)

[Gus 78] F.G. Gustavson. “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition”. ACM Trans. on Math. Software, Vol. 4, No. 3, Sept. 1978, 250–269

[LeWo 94] A.R. Lebeck, D.A. Wood. “Cache Profiling and the SPEC Benchmarks: A Case Study”. IEEE Computer, Oct. 1994, 15–26

[PTVF 92] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery. Numerical Recipes, *The Art of Scientific Computing.* 2nd ed. (Cambridge University Press, 1992)

[PDZ 92] S.C. Park, J.P. Draayer, S.Q. Zheng. “Fast Sparse Matrix Multiplication”. Computer Physics Comm. 70 (1992) 557–568.

[Tol 96] S. Toledo. “Improving Memory–System Performance of Sparse Matrix–Vector Multiplication”. Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing, March 1997.