Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks

Aydın Buluç*
aydin@cs.ucsb.edu

Jeremy T. Fineman†
jfineman@csail.mit.edu

Matteo Frigo‡
matteo@cilk.com

John R. Gilbert*
gilbert@cs.ucsb.edu

Charles E. Leiserson†‡
cel@mit.edu

*Dept. of Computer Science
University of California
Santa Barbara, CA 93106

†MIT CSAIL
32 Vassar Street
Cambridge, MA 02139

‡Cilk Arts, Inc.
55 Cambridge Street, Suite 200
Burlington, MA 01803

Abstract

This paper introduces a storage format for sparse matrices, called ***compressed sparse blocks (CSB)***, which allows both Ax and $A^{\mathrm{T}}x$ to be computed efficiently in parallel, where A is an $n \times n$ sparse matrix with $nnz \geq n$ nonzeros and x is a dense n-vector. Our algorithms use $\Theta(nnz)$ work (serial running time) and $\Theta(\sqrt{n}\lg n)$ span (critical-path length), yielding a parallelism of $\Theta(nnz/\sqrt{n}\lg n)$, which is amply high for virtually any large matrix. The storage requirement for CSB is esssentially the same as that for the more-standard compressed-sparse-rows (CSR) format, for which computing Ax in parallel is easy but $A^{\mathrm{T}}x$ is difficult. Benchmark results indicate that on one processor, the CSB algorithms for Ax and $A^{\mathrm{T}}x$ run just as fast as the CSR algorithm for Ax, but the CSB algorithms also scale up linearly with processors until limited by off-chip memory bandwidth.

Categories and Subject Descriptors

F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*computations on matrices*; G.4 [**Mathematics of Computing**]: Mathematical Software—*parallel and vector implementations*

General Terms

Algorithms, Design, Experimentation, Performance, Theory

Keywords

Compressed sparse blocks, compressed sparse columns, compressed sparse rows, matrix transpose, matrix-vector multiplication, multithreaded algorithm, parallelism, span, sparse matrix, storage format, work.

This work was supported in part by the National Science Foundation under Grants 0540248, 0615215, 0712243, 0822896, and 0709385, and by MIT Lincoln Laboratory under contract 7000012980.

1 Introduction

When multiplying a large $n \times n$ sparse matrix A having nnz nonzeros by a dense n-vector x, the memory bandwidth for reading A can limit overall performance. Consequently, most algorithms to compute Ax store A in a compressed format. One simple "tuple" representation involves storing each nonzero of A as a triple consisting of its row index, its column index, and the nonzero value itself. This representation, however, requires storing $2\,nnz$ row and column indices, in addition to the nonzeros. The current standard storage format for sparse matrices in scientific computing, ***compressed sparse rows (CSR)*** [32], is more efficient, because it stores only $n+nnz$ indices or pointers. This reduction in storage of CSR compared with the tuple representation tends to result in faster serial algorithms.

In the domain of parallel algorithms, however, CSR has its limitations. Although CSR lends itself to a simple parallel algorithm for computing the matrix-vector product Ax, this storage format does not admit an efficient parallel algorithm for computing the product $A^{\mathrm{T}}x$, where A^{T} denotes the transpose of the matrix A — or equivalently, for computing the product $x^{\mathrm{T}}A$ of a row vector x^{T} by A. Although one could use ***compressed sparse columns (CSC)*** to compute $A^{\mathrm{T}}x$, many applications, including iterative linear system solvers such as biconjugate gradients and quasi-minimal residual [32], require both Ax and $A^{\mathrm{T}}x$. One could transpose A explicitly, but computing the transpose for either CSR or CSC formats is expensive. Moreover, since matrix-vector multiplication for sparse matrices is generally limited by memory bandwidth, it is desirable to find a storage format for which both Ax and $A^{\mathrm{T}}x$ can be computed in parallel without performing more than nnz fetches of nonzeros from the memory to compute either product.

This paper presents a new storage format called ***compressed sparse blocks (CSB)*** for representing sparse matrices. Like CSR and CSC, the CSB format requires only $n+nnz$ words of storage for indices. Because CSB does not favor rows over columns or vice versa, it admits efficient parallel algorithms for computing either Ax or $A^{\mathrm{T}}x$, as well as for computing Ax when A is symmetric and only half the matrix is actually stored.

Previous work on parallel sparse matrix-vector multiplication has focused on reducing communication volume in a distributed-memory setting, often by using graph or hypergraph partitioning techniques to find good data distributions for particular matrices ([7, 38], for example). Good partitions generally exist for matrices whose structures arise from numerical discretizations of partial differential equations in two or three spatial dimensions. Our work, by contrast, is motivated by multicore and manycore architectures, in

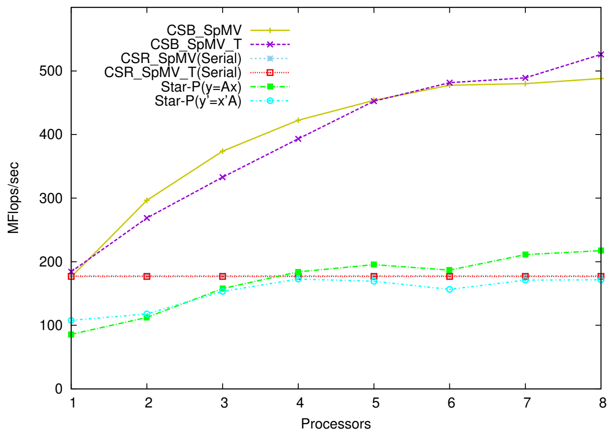


Figure 1: Average performance of Ax and $A^{\mathrm{T}}x$ operations on 13 different matrices from our benchmark test suite. CSB_SpMV and CSB_SpMV_T use compressed sparse blocks to perform Ax and $A^{\mathrm{T}}x$, respectively. CSR_SpMV (Serial) and CSR_SpMV_T (Serial) use OSKI [39] and compressed sparse rows without any matrix-specific optimizations. Star-P (y=Ax) and Star-P (y'=x'A) use Star-P [34], a parallel code based on CSR. The experiments were run on a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors.

which parallelism and memory bandwidth are key resources. Our algorithms are efficient in these measures for matrices with arbitrary nonzero structure.

Figure 1 presents an overall summary of achieved performance. The serial CSR implementation uses plain OSKI [39] without any matrix-specific optimizations. The graph shows the average performance over all our test matrices except for the largest, which failed to run on Star-P [34] due to memory constraints. The performance is measured in Mflops (Millions of FLoating-point OPerationS) per second. Both Ax and $A^{\mathrm{T}}x$ take $2\,nnz$ flops. To measure performance, we divide this value by the time it takes for the computation to complete. Section 7 provides detailed performance results.

The remainder of this paper is organized as follows. Section 2 discusses the limitations of the CSR/CSC formats for parallelizing Ax and $A^{\mathrm{T}}x$ calculations. Section 3 describes the CSB format for sparse matrices. Section 4 presents the algorithms for computing Ax and $A^{\mathrm{T}}x$ using the CSB format, and Section 5 provides a theoretical analysis of their parallel performance. Section 6 describes the experimental setup we used, and Section 7 presents the results. Section 8 offers some concluding remarks.

2 Conventional storage formats

This section describes the CSR and CSC sparse-matrix storage formats and explores their limitations when it comes to computing both Ax and $A^{\mathrm{T}}x$ in parallel. We review the work/span formulation of parallelism and show that performing Ax with CSR (or equivalently $A^{\mathrm{T}}x$ with CSC) yields ample parallelism. We consider various strategies for performing $A^{\mathrm{T}}x$ in parallel with CSR (or equivalently Ax with CSC) and why they are problematic.

The compressed sparse row (CSR) format stores the nonzeros (and ideally only the nonzeros) of each matrix row in consecutive memory locations, and it stores an index to the first stored element of each row. In one popular variant [14], CSR maintains one floating-point array $val[nnz]$ and two integer arrays, $col\_ind[nnz]$ and $row\_ptr[n]$ to store the matrix $A = (a_{ij})$. The row_ptr array stores the index of each row in val. That is, if $val[k]$ stores matrix element a_{ij}, then $row\_ptr[i] \le k < row\_ptr[i+1]$. The col_ind array stores the column indices of the elements in the val array. That is, if $val[k]$ stores matrix element a_{ij}, then $col\_ind[k] = j$.

The compressed sparse column (CSC) format is analogous to CSR, except that the nonzeros of each column, instead of row, are stored in contiguous memory locations. In other words, the CSC format for A is obtained by storing A^{T} in CSR format.

The earliest written description of CSR that we have been able to divine from the literature is an unnamed "scheme" presented in Table 1 of the 1967 article [36] by Tinney and Walker, although in 1963 Sato and Tinney [33] allude to what is probably CSR. Markowitz's seminal paper [28] on sparse Gaussian elimination does not discuss data structures, but it is likely that Markowitz used such a format as well. CSR and CSC have since become ubiquitous in sparse matrix computation [13, 16, 17, 21, 23, 32].

The following lemma states the well-known bound on space used by the index data in the CSR format (and hence the CSC format as well). By index data, we mean all data other than the nonzeros — that is, the row_ptr and col_ind arrays.

Lemma 1. *The CSR format uses $n \lg nnz + nnz \lg n$ bits of index data for an $n \times n$ matrix.*

For a CSR matrix A, computing $y \leftarrow Ax$ in parallel is straightforward, as shown in Figure 2. Procedure CSR_SPMV in the figure computes each element of the output array in parallel, and it does not suffer from race conditions, because each parallel iteration i writes to a single location $y[i]$ which is not updated by any other iteration.

```
CSR_SPMV(A, x, y)
1  n ← A.rows
2  for i ← 0 to n − 1 in parallel
3     do y[i] ← 0
4        for k ← A.row_ptr[i] to A.row_ptr[i+1] − 1
5           do y[i] ← y[i] + A.val[k] · x[A.col_ind[k]]
```

Figure 2: Parallel procedure for computing $y \leftarrow Ax$, where the $n \times n$ matrix A is stored in CSR format.

We shall measure the complexity of this code, and other codes in this paper, in terms of ***work*** and ***span*** [10, Ch. 27]:

- The ***work***, denoted by T_1, is the running time on 1 processor.
- The ***span***,[1] denoted by T_∞, is running time on an infinite number of processors.

The ***parallelism*** of the algorithm is T_1/T_∞, which corresponds to the maximum possible speedup on any number of processors. Generally, if a machine has somewhat fewer processors than the parallelism of an application, a good scheduler should be able to achieve linear speedup. Thus, for a fixed amount of work, our goal is to achieve a sufficiently small span so that the parallelism exceeds the number of processors by a reasonable margin.

The work of CSR_SPMV is $\Theta(nnz)$, assuming, as we shall, that $nnz \ge n$, because the body of the outer loop starting in line 2 executes for n iterations, and the body of the inner loop starting in line 4 executes for the number of nonzeros in the ith row, for a total of nnz times.

The span of CSR_SPMV depends on the maximum number nr of nonzeros in any row of the matrix A, since that number determines the worst-case time of any iteration of the loop in line 4. The n iterations of the parallel loop in line 2 contribute $\Theta(\lg n)$ to the span, assuming that loops are implemented as binary recursion. Thus, the total span is $\Theta(nr + \lg n)$.

The parallelism is therefore $\Theta(nnz/(nr + \lg n))$. In many common situations, we have $nnz = \Theta(n)$, which we will assume for estimation purposes. The maximum number nr of nonzeros in any

[1] The literature also uses the terms ***depth*** [3] and ***critical-path length*** [4].

```
CSR_SPMV_T(A,x,y)

1  n ← A.cols
2  for i ← 0 to n − 1
3    do y[i] ← 0
4  for i ← 0 to n − 1
5    do for k ← A.row_ptr[i] to A.row_ptr[i+1] − 1
6         do y[A.col_ind[k]] ← y[A.col_ind[k]] + A.val[k] · x[i]
```

Figure 3: Serial procedure for computing $y \leftarrow A^{\mathrm{T}}x$, where the $n \times n$ matrix A is stored in CSR format.

row can vary considerably, however, from a constant, if all rows have an average number of nonzeros, to n, if the matrix has a dense row. If $nr = O(1)$, then the parallelism is $\Theta(nnz/\lg n)$, which is quite high for a matrix with a billion nonzeros. In particular, if we ignore constants for the purpose of making a ballpark estimate, we have $nnz/\lg n \approx 10^9/(\lg 10^9) > 3 \times 10^7$, which is much larger than any number of processors one is likely to encounter in the near future. If $nr = \Theta(n)$, however, as is the case when there is even a single dense row, we have parallelism $\Theta(nnz/n) = \Theta(1)$, which limits scalability dramatically. Fortunately, we can parallelize the inner loop (line 4) using divide-and-conquer recursion to compute the sparse inner product in $\lg(nr)$ span without affecting the asymptotic work, thereby achieving parallelism $\Theta(nnz/\lg n)$ in all cases.

Computing $A^{\mathrm{T}}x$ serially can be accomplished by simply interchanging the row and column indices [15], yielding the pseudocode shown in Figure 3. The work of procedure CSR_SPMV_T is $\Theta(nnz)$, the same as CSR_SPMV.

Parallelizing CSR_SPMV_T is not straightforward, however. We shall review several strategies to see why it is problematic.

One idea is to parallelize the loops in lines 2 and 5, but this strategy yields minimal scalability. First, the span of the procedure is $\Theta(n)$, due to the loop in line 4. Thus, the parallelism can be at most $O(nnz/n)$, which is a small constant in most common situations. Second, in any practical system, the communication and synchronization overhead for executing a small loop in parallel is much larger than the execution time of the few operations executed in line 6.

Another idea is to execute the loop in line 4 in parallel. Unfortunately, this strategy introduces race conditions in the read/modify/write to $y[A.col\_ind[k]]$ in line 6.[2] These races can be addressed in two ways, neither of which is satisfactory.

The first solution involves locking column $col\_ind[k]$ or using some other form of atomic update.[3] This solution is unsatisfactory because of the high overhead of the lock compared to the cost of the update. Moreover, if A contains a dense column, then the contention on the lock is $\Theta(n)$, which completely destroys any parallelism in the common case where $nnz = \Theta(n)$.

The second solution involves splitting the output array y into multiple arrays y_p in a way that avoids races, and then accumulating $y \leftarrow \Sigma_p y_p$ at the end of the computation. For example, in a system with P processors (or threads), one could postulate that processor p only operates on array y_p, thereby avoiding any races. This solution is unsatisfactory because the work becomes $\Theta(nnz+Pn)$, where the last term comes from the need to initialize and accumulate P (dense) length-n arrays. Thus, the parallel execution time is $\Theta((nnz+Pn)/P) = \Omega(n)$ no matter how many processors are available.

A third idea for parallelizing $A^{\mathrm{T}}x$ is to compute the transpose explicitly and then use CSR_SPMV. Unfortunately, parallel transposition of a sparse matrix in CSR format is costly and encounters exactly the same problems we are trying to avoid. Moreover, every element is accessed at least twice: once for the transpose, and once for the multiplication. Since the calculation of a matrix-vector product tends to be memory-bandwidth limited, this strategy is generally inferior to any strategy that accesses each element only once.

Finally, of course, we could store the matrix A^{T} in CSR format, that is, storing A in CSC format, but then computing Ax becomes difficult.

To close this section, we should mention that if the matrix A is symmetric, so that only about half the nonzeros need be stored — for example, those on or above the diagonal — then computing Ax in parallel for CSR is also problematic. For this example, the elements below the diagonal are visited in an inconvenient order, as if they were stored in CSC format.

3 The CSB storage format

This section describes the CSB storage format for sparse matrices and shows that it uses the same amount of storage space as the CSR and CSC formats. We also compare CSB to other blocking schemes.

For a given ***block-size parameter*** β, CSB partitions the $n \times n$ matrix A into n^2/β^2 equal-sized $\beta \times \beta$ square ***blocks***[4]

$$A = \begin{pmatrix} A_{00} & A_{01} & \cdots & A_{0,n/\beta-1} \\ A_{10} & A_{11} & \cdots & A_{1,n/\beta-1} \\ \vdots & \vdots & \ddots & \vdots \\ A_{n/\beta-1,0} & A_{n/\beta-1,1} & \cdots & A_{n/\beta-1,n/\beta-1} \end{pmatrix},$$

where the block A_{ij} is the $\beta \times \beta$ submatrix of A containing elements falling in rows $i\beta, i\beta+1, \ldots, (i+1)\beta-1$ and columns $j\beta, j\beta+1, \ldots, (j+1)\beta-1$ of A. For simplicity of presentation, we shall assume that β is an exact power of 2 and that it divides n; relaxing these assumptions is straightforward.

Many or most of the individual blocks A_{ij} are ***hypersparse*** [6], meaning that the ratio of nonzeros to matrix dimension is asymptotically 0. For example, if $\beta = \sqrt{n}$ and $nnz = cn$, the average block has dimension $\sqrt{n}$ and only c nonzeros. The space to store a block should therefore depend only on its nonzero count, not on its dimension.

CSB represents a block A_{ij} by compactly storing a triple for each nonzero, associating with the nonzero data element a row and column index. In contrast to the column index stored for each nonzero in CSR, the row and column indices lie within the submatrix A_{ij}, and hence require fewer bits. In particular, if $\beta = \sqrt{n}$, then each index into A_{ij} requires only half the bits of an index into A. Since these blocks are stored contiguously in memory, CSB uses an auxiliary array of pointers to locate the beginning of each block.

More specifically, CSB maintains a floating-point array $val[nnz]$, and three integer arrays $row\_ind[nnz]$, $col\_ind[nnz]$, and $blk\_ptr[n^2/\beta^2]$. We describe each of these arrays in turn.

The *val* array stores all the nonzeros of the matrix and is analogous to CSR's array of the same name. The difference is that CSR stores *rows* contiguously, whereas CSB stores *blocks* contiguously. Although each block must be contiguous, the ordering among blocks is flexible. Let $f(i,j)$ be the bijection from pairs of block indices to integers in the range $0, 1, \ldots, n^2/\beta^2 - 1$ that describes the ordering among blocks. That is, $f(i,j) < f(i',j')$ if and

[2] In fact, if $nnz > n$, then the "pigeonhole principle" guarantees that the program has at least one race condition.

[3] No mainstream hardware supports atomic update of floating-point quantities, however.

[4] The CSB format may be easily extended to nonsquare $n \times m$ matrices. In this case, the blocks remain as square $\beta \times \beta$ matrices, and there are nm/β^2 blocks.

only if A_{ij} appears before $A_{i'j'}$ in *val*. We discuss choices of ordering later in this section.

The *row_ind* and *col_ind* arrays store the row and column indices, respectively, of the elements in the *val* array. These indices are relative to the block containing the particular element, not the entire matrix, and hence they range from 0 to $\beta - 1$. That is, if $val[k]$ stores the matrix element $a_{i\beta+r, j\beta+c}$, which is located in the rth row and cth column of the block A_{ij}, then $row\_ind = r$ and $col\_ind = c$. As a practical matter, we can pack a corresponding pair of elements of *row_ind* and *col_ind* into a single integer word of $2\lg\beta$ bits so that they make a single array of length *nnz*, which is comparable to the storage needed by CSR for the *col_ind* array.

The *blk_ptr* array stores the index of each block in the *val* array, which is analogous to the *row_ptr* array for CSR. If $val[k]$ stores a matrix element falling in block A_{ij}, then $blk\_ptr[f(i, j)] \le k < blk\_ptr[f(i, j)+1]$.

The following lemma states the storage used for indices in the CSB format.

Lemma 2. *The CSB format uses* $(n^2/\beta^2)\lg nnz + 2\,nnz\lg\beta$ *bits of index data.*

Proof. Since the *val* array contains *nnz* elements, referencing an element requires $\lg nnz$ bits, and hence the *blk_ptr* array uses $(n^2/\beta^2)\lg nnz$ bits of storage.

For each element in *val*, we use $\lg\beta$ bits to represent the row index and $\lg\beta$ bits to represent the column index, requiring a total of $nnz\lg\beta$ bits for each of *row_ind* and *col_ind*. Adding the space used by all three indexing arrays completes the proof. □

To better understand the storage requirements of CSB, we present the following corollary for $\beta = \sqrt{n}$. In this case, both CSR (Lemma 1) and CSB use the same storage.

Corollary 3. *The CSB format uses* $n\lg nnz + nnz\lg n$ *bits of index data when* $\beta = \sqrt{n}$. □

Thus far, we have not addressed the ordering of elements within each block or the ordering of blocks. Within a block, we use a Z-Morton ordering [29], storing first all those elements in the top-left quadrant, then the top-right, bottom-left, and finally bottom-right quadrants, using the same layout recursively within each quadrant. In fact, these quadrants may be stored in any order, but the recursive ordering is necessary for our algorithm to achieve good parallelism within a block.

The choice of storing the nonzeros within blocks in a recursive layout is opposite to the common wisdom for storing dense matrices [18]. Although most compilers and architectures favor conventional row/column ordering for optimal prefetching, the choice of layout within the block becomes less significant for sparse blocks as they already do not take full advantage of such features. More importantly, a recursive ordering allows us to efficiently determine the four quadrants of a block using binary search, which is crucial for parallelizing individual blocks.

Our algorithm and analysis do not, however, require any particular ordering among blocks. A Z-Morton ordering (or any recursive ordering) seems desirable as it should get better performance in practice by providing spatial locality, and it matches the ordering within a block. Computing the function $f(i, j)$, however, is simpler for a row-major or column-major ordering among blocks.

Comparison with other blocking methods

A blocked variant of CSR, called ***BCSR***, has been used for improving register reuse [24]. In BCSR, the sparse matrix is divided into small dense blocks that are stored in consecutive memory locations. The pointers are maintained to the first block on each row of blocks. BCSR storage is converse to CSB storage, because BCSR stores a sparse collection of dense blocks, whereas CSB stores a dense collection of sparse blocks. We conjecture that it would be advantageous to apply BCSR-style register blocking to each individual sparse block of CSB.

Nishtala *et al.* [30] have proposed a data structure similar to CSB in the context of cache blocking. Our work differs from theirs in two ways. First, CSB is symmetric without favoring rows over columns. Second, our algorithms and analysis for CSB are designed for parallelism instead of cache performance. As shown in Section 5, CSB supports ample parallelism for algorithms computing Ax and $A^{\mathrm{T}}x$, even on sparse and irregular matrices.

Blocking is also used in dense matrices. The Morton-hybrid layout [1,27], for example, uses a parameter equivalent to our parameter β for selecting the block size. Whereas in CSB we store elements in a Morton ordering within blocks and an arbitrary ordering among blocks, the Morton-hybrid layout stores elements in row-major order within blocks and a Morton ordering among blocks. The Morton-hybrid layout is designed to take advantage of hardware and compiler optimizations (within a block) while still exploiting the cache benefits of a recursive layout. Typically the block size is chosen to be 32×32, which is significantly smaller than the $\Theta(\sqrt{n})$ block size we propose for CSB. The Morton-hybrid layout, however, considers only dense matrices, for which designing a matrix-vector multiplication algorithm with good parallelism is significantly easier.

4 Matrix-vector multiplication using CSB

This section describes a parallel algorithm for computing the sparse-matrix dense-vector product $y \leftarrow Ax$, where A is stored in CSB format. This algorithm can be used equally well for computing $y \leftarrow A^{\mathrm{T}}x$ by switching the roles of row and column. We first give an overview of the algorithm and then describe it in detail.

At a high level, the CSB_SPMV multiplication algorithm simply multiplies each "blockrow" by the vector x in parallel, where the ith ***blockrow*** is the row of blocks $(A_{i0}A_{i1}\cdots A_{i,n/\beta-1})$. Since each blockrow multiplication writes to a different portion of the output vector, this part of the algorithm contains no races due to write conflicts.

If the nonzeros were guaranteed to be distributed evenly among block rows, then the simple blockrow parallelism would yield an efficient algorithm with n/β-way parallelism by simply performing a serial multiplication for each blockrow. One cannot, in general, guarantee that distribution of nonzeros will be so nice, however. In fact, sparse matrices in practice often include at least one dense row containing roughly n nonzeros, whereas the number of nonzeros is only $nnz \approx cn$ for some small constant c. Thus, performing a serial multiplication for each blockrow yields no better than c-way parallelism.

To make the algorithm robust to matrices of arbitrary nonzero structure, we must parallelize the blockrow multiplication when a blockrow contains "too many" nonzeros. This level of parallelization requires care to avoid races, however, because two blocks in the same blockrow write to the same region within the output vector. Specifically, when a blockrow contains $\Omega(\beta)$ nonzeros, we recursively divide it "in half," yielding two subblockrows, each containing roughly half the nonzeros. Although each of these subblockrows can be multiplied in parallel, they may need to write to the same region of the output vector. To avoid the races that might arise due to write conflicts between the subblockrows, we allocate a temporary vector to store the result of one of the subblockrows and allow the other subblockrow to use the output vector. After both subblockrow multiplications complete, we serially add the temporary vector into the output vector.

To facilitate fast subblockrow divisions, we first partition the blockrow into "chunks" of consecutive blocks, each containing at

```
CSB_SPMV(A, x, y)
1   for i ← 0 to n/β − 1 in parallel      // For each blockrow.
2     do Initialize a dynamic array R_i
3        R_i[0] ← 0
4        count ← 0                        // Count nonzeroes in chunk.
5        for j ← 0 to n/β − 2
6          do count ← count + nnz(A_ij)
7             if count + nnz(A_i,j+1) > Θ(β)
8               then // End the chunk, since the next block
                        // makes it too large.
9                       append j to R_i      // Last block in chunk.
10                      count ← 0
11       append n/β − 1 to R_i
12       CSB_BLOCKROWV(A, i, R_i, x, y[iβ . . (i+1)β − 1])
```

Figure 4: Pseudocode for the matrix-vector multiplication $y \leftarrow Ax$. The procedure CSB_BLOCKROWV (pseudocode for which can be found in Figure 5) as called here multiplies the blockrow by the vector x and writes the output into the appropriate region of the output vector y. The notation $x[a\,..\,b]$ means the subarray of x starting at index a and ending at index b. The function $nnz(A_{ij})$ is a shorthand for $A.blk\_ptr[f(i,j)+1] - A.blk\_ptr[f(i,j)]$, which calculates the number of nonzeros in the block A_{ij}. For conciseness, we have overloaded the $\Theta(\beta)$ notation (in line 7) to mean "a constant times β"; any constant suffices for the analysis, and we use the constant 3 in our implementation.

most $O(\beta)$ nonzeros (when possible) and $\Omega(\beta)$ nonzeros on average. The lower bound of $\Omega(\beta)$ will allow us to amortize the cost of writing to the length-β temporary vector against the nonzeros in the chunk. By dividing a blockrow "in half," we mean assigning to each subblockrow roughly half the chunks.

Figure 4 gives the top-level algorithm, performing each blockrow vector multiplication in parallel. The "**for**... **in parallel do**" construct means that each iteration of the **for** loop may be executed in parallel with the others. For each loop iteration, we partition the blockrow into chunks in lines 2–11 and then call the blockrow multiplication in line 12. The array R_i stores the indices of the last block in each chunk; specifically, the kth chunk, for $k > 0$, includes blocks $(A_{i,R_i[k-1]+1} A_{i,R_i[k-1]+2} \cdots A_{i,R_i[k]})$. A chunk consists of either a single block containing $\Omega(\beta)$ nonzeros, or it consists of many blocks containing $O(\beta)$ nonzeros in total. To compute chunk boundaries, just iterate over blocks (in lines 5–10) until enough nonzeros are accrued.

Figure 5 gives the parallel algorithm CSB_BLOCKROWV for multiplying a blockrow by a vector, writing the result into the length-β vector y. In lines 22–29, the algorithm recursively divides the blockrow such that each half receives roughly the same number of chunks. We find the appropriate middles of the chunk array R and the input vector x in lines 22 and 23, respectively. We then allocate a length-β temporary vector z (line 24) and perform the recursive multiplications on each subblockrow in parallel (lines 25–27), having one of the recursive multiplications write its output to z. When these recursive multiplications complete, we merge the outputs into the vector y (lines 28–29).

The recursion bottoms out when the blockrow consists of a single chunk (lines 12–21). If this chunk contains many blocks, it is guaranteed to contain at most $\Theta(\beta)$ nonzeros, which is sufficiently sparse to perform the serial multiplication in line 20. If, on the other hand, the chunk is a single block, it may contain as many as $\beta^2 \approx n$ nonzeros. A serial multiplication here, therefore, would be the bottleneck in the algorithm. Instead, we perform the parallel block-vector multiplication CSB_BLOCKV in line 18.

If the blockrow recursion reaches a single block, we perform a parallel multiplication of the block by the vector, given in Figure 6.

```
CSB_BLOCKROWV(A, i, R, x, y)
11  if R.length = 2         // The subblockrow is a single chunk.
12    then ℓ ← R[0] + 1                 // Leftmost block in chunk.
13         r ← R[1]                     // Rightmost block in chunk.
14         if ℓ = r
15           then // The chunk is a single (dense) block.
16                start ← A.blk_ptr[f(i, ℓ)]
17                end ← A.blk_ptr[f(i, ℓ) + 1] − 1
18                CSB_BLOCKV(A, start, end, β, x, y)
19           else // The chunk is sparse.
20                multiply y ← (A_iℓ A_i,ℓ+1 ··· A_ir) x serially
21         return
    // Since the block row is "dense," split it in half.
22  mid ← ⌈R.length/2⌉ − 1              // Divide chunks in half.
    // Calculate the dividing point in the input vector x.
23  xmid ← β · (R[mid] − R[0])
24  allocate a length-β temporary vector z, initialized to 0
25  in parallel
26    do CSB_BLOCKROWV(A, i, R[0 . . mid], x[0 . . xmid − 1], y)
27    do CSB_BLOCKROWV(A, i, R[mid . . R.length − 1],
                       x[xmid . . x.length − 1], z)
28  for k ← 0 to β − 1
29    do y[k] ← y[k] + z[k]
```

Figure 5: Pseudocode for the subblockrow vector product $y \leftarrow (A_{i\ell} A_{i,\ell+1} \cdots A_{ir})x$. The **in parallel do**... **do**... construct indicates that all of the **do** code blocks may execute in parallel. The procedure CSB_BLOCKV (pseudocode for which can be found in Figure 6) calculates the product of the block and the vector in parallel.

The block-vector multiplication proceeds by recursively dividing the (sub)block M into quadrants M_{00}, M_{01}, M_{10}, and M_{11}, each of which is conveniently stored contiguously in the Z-Morton-ordered *val*, *row_ind*, and *col_ind* arrays between indices *start* and *end*. We perform binary searches to find the appropriate dividing points in the array in lines 34–36.

To understand the pseudocode, consider the search for the dividing point s_2 between $M_{00}M_{01}$ and $M_{10}M_{11}$. For any recursively chosen $dim \times dim$ matrix M, the column indices and row indices of all elements have the same leading $\lg\beta - \lg dim$ bits. Moreover, for those elements in $M_{00}M_{01}$, the next bit in the row index is a 0, whereas for those in elements in $M_{10}M_{11}$, the next bit in the row index is 1. The algorithm does a binary search for the point at which this bit flips. The cases for the dividing point between M_{00} and M_{01} or M_{10} and M_{11} are similar, except that we focus on the column index instead of the row index.

After dividing the matrix into quadrants, we execute the matrix products involving matrices M_{00} and M_{11} in parallel (lines 37–39), as they do not conflict on any outputs. After completing these products, we execute the other two matrix products in parallel (lines 40–42).[5] This procedure resembles a standard parallel divide-and-conquer matrix multiplication, except that our base case of serial multiplication starts at a matrix containing $\Theta(dim)$ nonzeros (lines 29–32). Note that although we pass the full length-β arrays x and y to each recursive call, the effective length of each array is halved implicitly by partitioning M into quadrants. Passing the full arrays is a technical detail required to properly compute array indices, as the indices $A.row\_ind$ and $A.col\_ind$ store offsets within the block.

The CSB_SPMV_T algorithm is identical to CSB_SPMV, except that we operate over blockcolumns rather than blockrows.

[5] The algorithm may instead do M_{00} and M_{10} in parallel followed by M_{01} and M_{11} in parallel without affecting the performance analysis. Presenting the algorithm with two choices may yield better load balance.

CSB_BLOCKV$(A, start, end, dim, x, y)$

// $A.val[start..end]$ is a $dim \times dim$ matrix M.
28 **if** $end - start \le \Theta(dim)$
29 **then** // Perform the serial computation $y \leftarrow y + Mx$.
30 **for** $k \leftarrow start$ **to** end
31 **do** $y[A.row\_ind[k]] \leftarrow y[A.row\_ind[k]]$ $+ A.val[k] \cdot x[A.col\_ind[k]]$
32 **return**
33 // Recurse. Find the indices of the quadrants.
34 binary search $start, start+1, \ldots, end$ for the smallest s_2 such that $(A.row\_ind[s_2]$ & $dim/2) \ne 0$
35 binary search $start, start+1, \ldots, s_2 - 1$ for the smallest s_1 such that $(A.col\_ind[s_1]$ & $dim/2) \ne 0$
36 binary search $s_2, s_2+1, \ldots, end$ for the smallest s_3 such that $(A.col\_ind[s_3]$ & $dim/2) \ne 0$
37 **in parallel**
38 **do** CSB_BLOCKV$(A, start, s_1 - 1, dim/2, x, y)$ // M_{00}.
39 **do** CSB_BLOCKV$(A, s_3, end, dim/2, x, y)$ // M_{11}.
40 **in parallel**
41 **do** CSB_BLOCKV$(A, s_1, s_2 - 1, dim/2, x, y)$ // M_{01}.
42 **do** CSB_BLOCKV$(A, s_2, s_3 - 1, dim/2, x, y)$ // M_{10}.

Figure 6: Pseudocode for the subblock-vector product $y \leftarrow Mx$, where M is the list of tuples stored in $A.val[start..end]$, $A.row\_ind[start..end]$, and $A.col\_ind[start..end]$, in recursive Z-Morton order. The & operator is a bitwise AND of the two operands.

5 Analysis

In this section, we prove that for an $n \times n$ matrix with nnz nonzeros, CSB_SPMV operates with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$ when $\beta = \sqrt{n}$, yielding a parallelism of $\Omega(nnz/\sqrt{n}\lg n)$. We also provide bounds in terms of β and analyze the space usage.

We begin by analyzing block-vector multiplication.

Lemma 4. *On a $\beta \times \beta$ block containing r nonzeros,* CSB_BLOCKV *runs with work $\Theta(r)$ and span $O(\beta)$.*

Proof. The span for multiplying a $dim \times dim$ matrix can be described by the recurrence $S(dim) = 2S(dim/2) + O(\lg dim) = O(dim)$. The $\lg dim$ term represents a loose upper bound on the cost of the binary searches. In particular, the binary-search cost is $O(\lg z)$ for a submatrix containing z nonzeros, and we have $z \le dim^2$, and hence $O(\lg z) = O(\lg dim)$, for a $dim \times dim$ matrix.

To calculate the work, consider the degree-4 tree of recursive procedure calls, and associate with each node the work done by that procedure call. We say that a node in the tree has height h if it corresponds to a $2^h \times 2^h$ subblock, i.e., if $dim = 2^h$ is the parameter passed into the corresponding CSB_BLOCKV call. Node heights are integers ranging from 0 to $\lg \beta$. Observe that each height-h node corresponds to a distinct $2^h \times 2^h$ subblock (although subblocks may overlap for nodes having different heights). A height-h leaf node (serial base case) corresponds to a subblock containing at most $z = O(2^h)$ nonzeros and has work linear in this number z of nonzeros. Summing across all leaves, therefore, gives $\Theta(r)$ work. A height-h internal node, on the other hand, corresponds to a subblock containing *at least* $z' = \Omega(2^h)$ nonzeros (or else it would not recurse further and be a leaf) and has work $O(\lg 2^h) = O(h)$ arising from the binary searches. There can thus be at most $O(r/2^h)$ height-h internal nodes having total work $O((r/2^h)h)$. Summing across all heights gives total work of $\sum_{h=0}^{\lg \beta} O((r/2^h)h) \le r \sum_{h=0}^{\lg \beta} O(h/2^h) = O(r)$ for internal nodes. Combining the work at internal nodes and leaf nodes gives total work $\Theta(r)$. $\square$

The next lemma analyzes blockrow-vector multiplication.

Lemma 5. *On a blockrow containing n/β blocks and r nonzeros,* CSB_BLOCKROWV *runs with work $\Theta(r)$ and span $O(\beta \lg(n/\beta))$.*

Proof. Consider a call to CSB_BLOCKROWV on a row that is partitioned into C chunks, and let $W(C)$ denote the work. The work per recursive call on a multichunk subblockrow is dominated by the $\Theta(\beta)$ work of initializing a temporary vector z and adding the vector z into the output vector y. The work for a CSB_BLOCKROWV on a single-chunk subblockrow is linear in the number of nonzeros in the chunk. (We perform linear work either in line 20 or in line 18 — see Lemma 4 for the work of line 18.) We can thus describe the work by the recurrence $W(C) \le 2W(\lceil C/2 \rceil) + \Theta(\beta)$ with a base case of work linear in the nonzeros, which solves to $W(C) = \Theta(C\beta + r)$ for $C > 1$. When $C = 1$, we have $W(C) = \Theta(r)$, as we do not operate on the temporary vector z.

To bound work, it remains to bound the maximum number of chunks in a row. Notice that any two consecutive chunks contain at least $\Omega(\beta)$ nonzeros. This fact follows from the way chunks are chosen in lines 2–11: a chunk is terminated only if adding the next block to the chunk would increase the number of nonzeros to more than $\Theta(\beta)$. Thus, a blockrow consists of a single chunk whenever $r = O(\beta)$ and at most $O(r/\beta)$ chunks whenever $r = \Omega(\beta)$. Hence, the total work is $\Theta(r)$.

We can describe the span of CSB_BLOCKROWV by the recurrence $S(C) = S(\lceil C/2 \rceil) + O(\beta) = O(\beta \lg C) + S(1)$. The base case involves either serially multiplying a single chunk containing at most $O(\beta)$ nonzeros in line 20, which has span $O(\beta)$, or multiplying a single block in parallel in line 18, which also has span $O(\beta)$ from Lemma 4. We have, therefore, a span of $O(\beta \lg C) = O(\beta \lg(n/\beta))$, since $C \le n/\beta$. $\square$

We are now ready to analyze matrix-vector multiplication itself.

Theorem 6. *On an $n \times n$ matrix containing nnz nonzeros,* CSB_SPMV *runs with work $\Theta(n^2/\beta^2 + nnz)$ and span $O(\beta \lg(n/\beta) + n/\beta)$.*

Proof. For each blockrow, we add $\Theta(n/\beta)$ work and span for computing the chunks, which arise from a serial scan of the n/β blocks in the blockrow. Thus, the total work is $O(n^2/\beta^2)$ in addition to the work for multiplying the blockrows, which is linear in the number of nonzeros from Lemma 5.

The total span is $O(\lg(n/\beta))$ to parallelize all the rows, plus $O(n/\beta)$ per row to partition the row into chunks, plus the $O(\beta \lg(n/\beta))$ span per blockrow from Lemma 5. $\square$

The following corollary gives the work and span bounds when we choose β to yield the same space for the CSB storage format as for the CSR or CSC formats.

Corollary 7. *On an $n \times n$ matrix containing $nnz \ge n$ nonzeros, by choosing $\beta = \Theta(\sqrt{n})$,* CSB_SPMV *runs with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$, achieving a parallelism of $\Omega(nnz/\sqrt{n}\lg n)$.* $\square$

Since CSB_SPMV_T is isomorphic to CSB_SPMV, we obtain the following corollary.

Corollary 8. *On an $n \times n$ matrix containing $nnz \ge n$ nonzeros, by choosing $\beta = \Theta(\sqrt{n})$,* CSB_SPMV_T *runs with work $\Theta(nnz)$ and span $O(\sqrt{n}\lg n)$, achieving a parallelism of $\Omega(nnz/\sqrt{n}\lg n)$.* $\square$

The work of our algorithm is dominated by the space of the temporary vectors z, and thus the space usage on an infinite number of processors matches the work bound. When run on fewer processors however, the space usage reduces drastically. We can analyze the space in terms of the ***serialization*** of the program, which corresponds to the program obtained by removing all **parallel** keywords.

Lemma 9. *On an $n \times n$ matrix, by choosing $\beta = \Theta(\sqrt{n})$, the serialization of* CSB_SPMV *requires $O(\sqrt{n}\lg n)$ space (not counting the storage for the matrix itself).*

Proof. The serialization executes one blockrow multiplication at a time. There are two space overheads. First, we use $O(n/\beta) = O(\sqrt{n})$ space for the chunk array. Second, we use β space to store the temporary vector z for each outstanding recursive call to CSB_BLOCKROWV. Since the recursion depth is $O(\lg n)$, the total space becomes $O(\beta \lg n) = O(\sqrt{n}\lg n)$. □

A typical work-stealing scheduler executes the program in a depth-first (serial) manner on each processor. When a processor completes all its work, it "steals" work from a different processor, beginning a depth-first execution from some unexecuted parallel branch. Although not all work-stealing schedulers are space efficient, those maintaining the ***busy-leaves property*** [5] (e.g., as used in the Cilk work-stealing scheduler [4]) are space efficient. The "busy-leaves" property roughly says that if a procedure has begun (but not completed) executing, then there exists a processor currently working on that procedure or one of its descendants procedures.

Corollary 10. *Suppose that a work-stealing scheduler with the busy-leaves property schedules an execution of* CSB_SPMV *on an $n \times n$ matrix with the choice $\beta = \sqrt{n}$. Then, the execution requires $O(P\sqrt{n}\lg n)$ space.*

Proof. Combine Lemma 9 and Theorem 1 from [4]. □

The work overhead of our algorithm may be reduced by increasing the constants in the $\Theta(\beta)$ threshold in line 7. Specifically, increasing this threshold by a constant factor reduces the number of reads and writes to temporaries by the same constant factor. As these temporaries constitute the majority of the work overhead of the algorithm, doubling the threshold nearly halves the overhead. Increasing the threshold, however, also increases the span by a constant factor, and so there is a trade-off.

6 Experimental design

This section describes our implementation of the CSB_SPMV and CSB_SPMV_T algorithms, the benchmark matrices we used to test the algorithms, the machines on which we ran our tests, and the other codes with which we compared our algorithms.

Implementation

We parallelized our code using Cilk++ [9], which is a faithful extension of C++ for multicore and shared-memory parallel programming. Cilk++ is based on the earlier MIT Cilk system [20], and it employs dynamic load balancing and provably optimal task scheduling. The CSB code used for the experiments is freely available for academic use at `http://gauss.cs.ucsb.edu/~aydin/software.html`.

The *row_ind* and *col_ind* arrays of CSB, which store the row and column indices of each nonzero within a block (i.e., the lower-order bits of the row and column indices within the matrix A), are implemented as a single ***index array*** by concatenating the two values together. The higher-order bits of *row_ind* and *col_ind* are stored only implicitly, and are retrieved by referencing the *blk_ptr* array.

The CSB blocks themselves are stored in row-major order, while the nonzeros within blocks are in Z-Morton order. The row-major ordering among blocks may seem to break the overall symmetry of CSB, but in practice it yields efficient handling of block indices for look-up in $A.blk\_ptr$ by permitting an easily computed look-up function $f(i, j)$. The row-major ordering also allowed us to count the nonzeros in a subblockrow more easily when computing $y \leftarrow Ax$. This optimization is not symmetric, but interestingly, we achieved similar performance when computing $y \leftarrow A^{\mathrm{T}}x$, where we must still aggregate the nonzeros in each block. In fact, in almost half the cases, computing $A^{\mathrm{T}}x$ was faster than Ax, depending on the matrix structure.

The Z-Morton ordering on nonzeros in each block is equivalent to first interleaving the bits of *row_ind* and *col_ind*, and then sorting the nonzeros using these bit-interleaved values as the keys. Thus, it is tempting to store the index array in a bit-interleaved fashion, thereby simplifying the binary searches in lines 34–36. Converting to and from bit-interleaved integers, however, is expensive with current hardware support,[6] which would be necessary for the serial base case in lines 29–32. Instead, the kth element of the index array is the concatenation of *row_ind*$[k]$ and *col_ind*$[k]$, as indicated earlier. This design choice of storing concatenated, instead of bit-interleaved, indices requires either some care when performing the binary search (as presented in Figure 6) or implicitly converting from the concatenated to interleaved format when making a binary-search comparison. Our preliminary implementation does the latter, using a C++ function object for comparisons [35]. In practice, the overhead of performing these conversions is small, since the number of binary-search steps is small.

Performing the actual address calculation and determining the pointers to x and y vectors are done by masking and bit-shifting. The bitmasks are determined dynamically by the CSB constructor depending on the input matrix and the data type used for storing matrix indices. Our library allows any data type to be used for matrix indices and handles any type of matrix dynamically. For the results presented in Section 7, nonzero values are represented as double-precision floating-point numbers, and indices are represented as 32-bit unsigned integers. Finally, as our library aims to be general instead of matrix specific, we did not employ speculative low-level optimizations such as software prefetching, pipelining, or matrix-specific optimizations such as index and/or value compression [25, 40], but we believe that CSB and our algorithms should not adversely affect incorporation of these approaches.

Choosing the block size β

We investigated different strategies to choose the block size that achieves the best performance. For the types of loads we ran, we found that a block size slightly larger than $\sqrt{n}$ delivers reasonable performance. Figure 7 shows the effect of different blocksizes on the performance of the $y \leftarrow Ax$ operation with the representative matrix Kkt_power. The closest exact power of 2 to $\sqrt{n}$ is 1024, which turns out to be slightly suboptimal. In our experiments, the overall best performance was achieved when β satisfies the equation $\lceil \lg\sqrt{n} \rceil \le \lg\beta \le 3 + \lceil \lg\sqrt{n} \rceil$.

Merely setting β to a hard-coded value, however, is not robust for various reasons. First, the elements stored in the index array should use the same data type as that used for matrix indices. Specifically, the integer $\beta - 1$ should fit in 2 bytes so that a concatenated *row_ind* and *col_ind* fit into 4 bytes. Second, the length-β regions of the input vector x and output vector y (which are accessed when multiplying a single block) should comfortably fit into L2 cache. Finally, to ensure speedup on matrices with evenly distributed nonzeros, there should be enough parallel slackness for the parallelization across blockrows (i.e., the highest level parallelism). Specifically, when β grows large, the parallelism is roughly bounded by $O(nnz/(\beta \lg(n/\beta)))$ (by dividing the work and span from Theorem 6). Thus, we want $nnz/(\beta \lg(n/\beta))$ to be "large enough," which means limiting the maximum magnitude of β.

We adjusted our CSB constructor, therefore, to automatically select a reasonable block-size parameter β. It starts with $\beta =$

[6] Recent research [31] addresses these conversions.

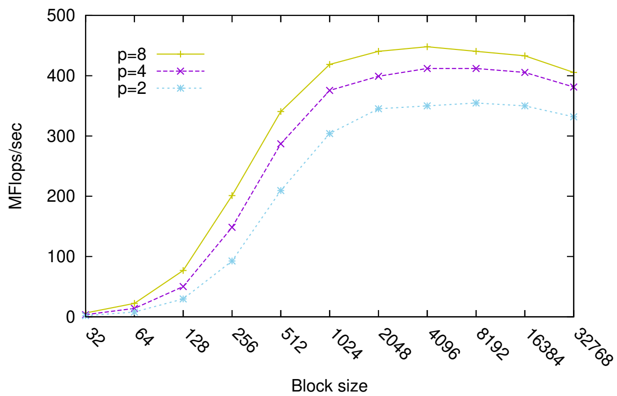


Figure 7: The effect of block size parameter β on SpMV performance using the Kkt_power matrix. For values $\beta > 32768$ and $\beta < 32$, the experiment failed to finish due to memory limitations. The experiment was conducted on the AMD Opteron.

$3 + \lceil \lg \sqrt{n} \rceil$ and keeps decreasing it until the aforementioned constraints are satisfied. Although a research opportunity may exist to autotune the optimal block size with respect to a specific matrix and architecture, in most test matrices, choosing $\beta = \sqrt{n}$ degraded performance by at most 10%–15%. The optimal β value barely shifts along the x-axis when running on different numbers of processors and is quite stable overall.

An optimization heuristic for structured matrices

Even though CSB_SpMV and CSB_SpMV_T are robust and exhibit plenty of parallelism on most matrices, their practical performance can be improved on some sparse matrices having regular structure. In particular, a block diagonal matrix with equally sized blocks has nonzeros that are evenly distributed across blockrows. In this case, a simple algorithm based on blockrow parallelism would suffice in place of the more complicated recursive method from CSB_BlockV. This divide-and-conquer within blockrows incurs overhead that might unnecessarily degrade performance. Thus, when the nonzeros are evenly distributed across the blockrows, our implementation of the top-level algorithm (given in Figure 4) calls the serial multiplication in line 12 instead of the CSB_BlockRowV procedure.

To see whether a given matrix is amenable to the optimization, we apply the following "balance" heuristic. We calculate the imbalance among blockrows (or blockcolumns in the case of $y \leftarrow A^{\mathrm{T}}x$) and apply the optimization only when no blocks have more than twice the average number of nonzeros per blockrow. In other words, if $\max(nnz(A_i)) < 2 \cdot \mathrm{mean}(nnz(A_i))$, then the matrix is considered to have balanced blockrows and the optimization is applied. Of course, this optimization is not the only way to achieve a performance boost on structured matrices.

Optimization of temporary vectors

One of the most significant overheads of our algorithm is the use of temporary vectors to store intermediate results when parallelizing a blockrow multiplication in CSB_BlockRowV. The "balance" heuristic above is one way of reducing this overhead when the nonzeros in the matrix are evenly distributed. For arbitrary matrices, however, we can still reduce the overhead in practice. In particular, we only need to allocate the temporary vector z (in line 24) if both of the subsequent multiplications (lines 25–27) are scheduled in parallel. If the first recursive call completes before the second recursive call begins, then we instead write directly into the output vector for both recursive calls. In other words, when a blockrow multiplication is scheduled serially, the multiplication procedure detects this fact and mimics a normal serial execution, without the use of temporary vectors. Our implementation exploits an undocumented feature of Cilk++ to test whether the first call has completed before making the second recursive call, and we allocate the temporary as appropriate. This test may also be implemented using Cilk++ reducers [19].

Sparse-matrix test suite

We conducted experiments on a diverse set of sparse matrices from real applications including circuit simulation, finite-element computations, linear programming, and web-connectivity analysis. These matrices not only cover a wide range of applications, but they also greatly vary in size, density, and structure. The test suite contains both rectangular and square matrices. Almost half of the square matrices are asymmetric. Figure 8 summarizes the 14 test matrices.

Included in Figure 8 is the load imbalance that is likely to occur for an SpMV algorithm parallelized with respect to columns (CSC) and blocks (CSB). In the last column, the average (mean) and the maximum number of nonzeros among columns (first line) and blocks (second line) are shown for each matrix. The sparsity of matrices can be quantified by the average number of nonzeros per column, which is equivalent to the mean of CSC. The sparsest matrix (Rajat31) has 4.3 nonzeros per column on the average while the densest matrix has about 73 nonzeros per column (Sme3Dc and Torso). For CSB, the reported mean/max values are obtained by setting the block dimension β to be approximately $\sqrt{n}$, so that they are comparable with statistics from CSC.

Architectures and comparisons

We ran our experiments on three multicore superscalar architectures. Opteron is a ccNUMA architecture powered by AMD Opteron 8214 (Santa Rosa) processors clocked at 2.2 GHz. Each core of Opteron has a private 1 MB L2 cache, and each socket has its own integrated memory controller. Although it is an 8-socket dual-core system, we only experimented with up to 8 processors. Harpertown is a dual-socket quad-core system running two Intel Xeon X5460's, each clocked at 3.16 GHz. Each socket has 12 MB of L2 cache, shared among four cores, and a front-side bus (FSB) running at 1333 MHz. Nehalem is a single-socket quad-core Intel Core i7 920 processor. Like Opteron, Nehalem has an integrated memory controller. Each core is clocked at 2.66 GHz and has a private 256 KB L2 cache. An 8 MB L3 cache is shared among four cores.

While Opteron has 64 GB of RAM, Harpertown and Nehalem have only 8 GB and 6 GB, respectively, which forced us to exclude our biggest test matrix (Webbase2001) from our runs on Intel architectures. We compiled our code using `gcc 4.1` on Opteron and Harpertown and with `gcc 4.3` on Nehalem, all with optimization flags `-O2 -fno-rtti -fno-exceptions`.

To evaluate our code on a single core, we compared its performance with "pure" OSKI matrix-vector multiplication [39] running on one processor of Opteron. We did not enable OSKI's preprocessing step, which chooses blockings for cache and register usage that are tuned to a specific matrix. We conjecture that such matrix-specific tuning techniques can be combined advantageously with our CSB data structure and parallel algorithms.

To compare with a parallel code, we used the matrix-vector multiplication of Star-P [34] running on Opteron. Star-P is a distributed-memory code that uses CSR to represent sparse matrices and distributes matrices to processor memories by equal-sized blocks of rows.

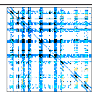
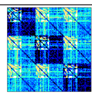
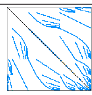
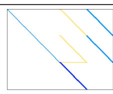
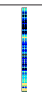
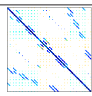
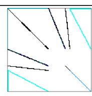
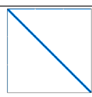
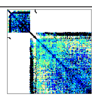
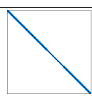
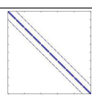
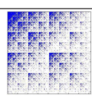
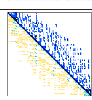
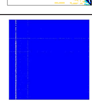
Name Description	Spy Plot	Dimensions Nonzeros	CSC (mean/max) CSB (mean/max)
Asic_320k circuit simulation		$321\text{K} \times 321\text{K}$ $1,931\text{K}$	6.0 / 157K 4.9 / 2.3K
Sme3Dc 3D structural mechanics		$42\text{K} \times 42\text{K}$ $3,148\text{K}$	73.3 / 405 111.6 / 1368
Parabolic_fem diff-convection reaction		$525\text{K} \times 525\text{K}$ $3,674\text{K}$	7.0 / 7 3.5 / 1,534
Mittelmann LP problem		$1,468\text{K} \times 1,961\text{K}$ $5,382\text{K}$	2.7 / 7 2.0 / 3,713
Rucci Ill-conditioned least-squares		$1,977\text{K} \times 109\text{K}$ $7,791\text{K}$	70.9 / 108 9.4 / 36
Torso Finite diff, 2D model of torso		$116\text{K} \times 116\text{K}$ $8,516\text{K}$	73.3 / 1.2K 41.3 / 36.6K
Kkt_power optimal power flow, nonlinear opt.		$2.06\text{M} \times 2.06\text{M}$ 12.77M	6.2 / 90 3.1 / 1,840
Rajat31 circuit simulation		$4.69\text{M} \times 4.69\text{M}$ 20.31M	4.3 / 1.2K 3.9 / 8.7K
Ldoor structural prob.		$952\text{K} \times 952\text{K}$ 42.49M	44.6 / 77 49.1 / 43,872
Bone010 3D trabecular bone		$986\text{K} \times 986\text{K}$ 47.85M	48.5 / 63 51.5 / 18,670
Grid3D200 3D 7-point finite-diff mesh		$8\text{M} \times 8\text{M}$ 55.7M	6.97 / 7 3.7 / 9,818
RMat23 Real-world graph model		$8.4\text{M} \times 8.4\text{M}$ 78.7M	9.4 / 70.3K 4.7 / 222.1K
Cage15 DNA electrophoresis		$5.15\text{M} \times 5.15\text{M}$ 99.2M	19.2 / 47 15.6 / 39,712
Webbase2001 Web connectivity		$118\text{M} \times 118\text{M}$ 1,019M	8.6 / 816K 4.9 / 2,375K

Figure 8: Structural information on the sparse matrices used in our experiments, ordered by increasing number of nonzeros. The first ten matrices and Cage15 are from the University of Florida sparse matrix collection [12]. Grid3D200 is a 7-point finite difference mesh generated using the Matlab Mesh Partitioning and Graph Separator Toolbox [22]. The RMat23 matrix [26], which models scale-free graphs, is generated by using repeated Kronecker products [2]. We chose parameters $A = 0.7$, $B = C = D = 0.1$ for RMat23 in order to generate skewed matrices. Webbase2001 is a crawl of the World Wide Web from the year 2001 [8].

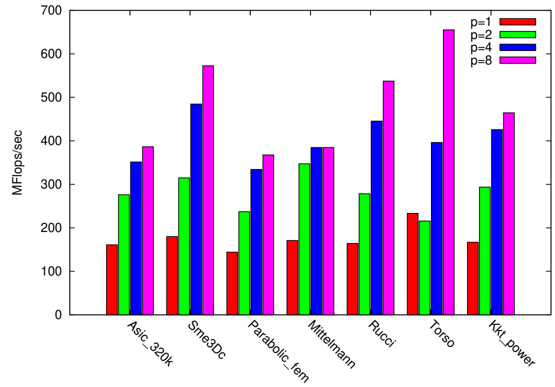


Figure 9: CSB_SPMV performance on Opteron (smaller matrices).

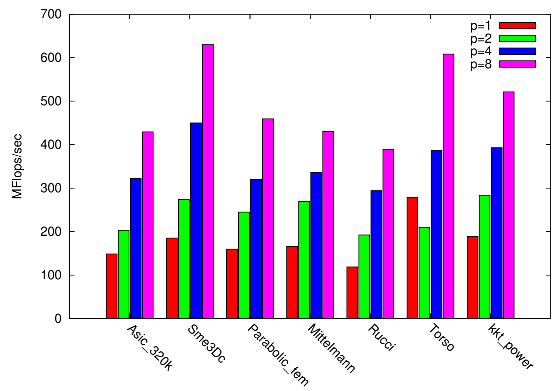


Figure 10: CSB_SPMV_T performance on Opteron (smaller matrices).

7 Experimental results

Figures 9 and 10 show how CSB_SPMV and CSB_SPMV_T, respectively, scale for the seven smaller matrices on Opteron, and Figures 11 and 12 show similar results for the seven larger matrices. In most cases, the two codes show virtually identical performance, confirming that the CSB data structure and algorithms are equally suitable for both operations. In all the parallel scaling graphs, only the values $p = 1, 2, 4, 8$ are reported. They should be interpreted as performance achievable by doubling the number of cores instead

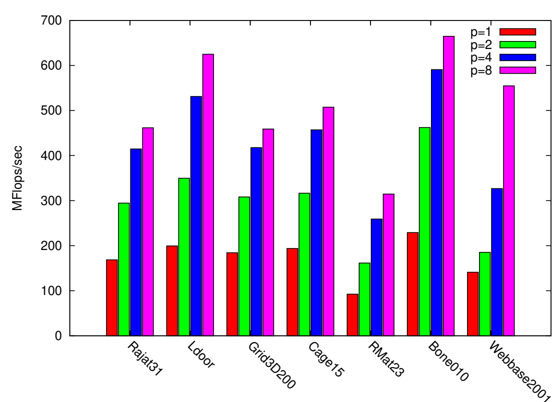


Figure 11: CSB_SPMV performance on Opteron (larger matrices).

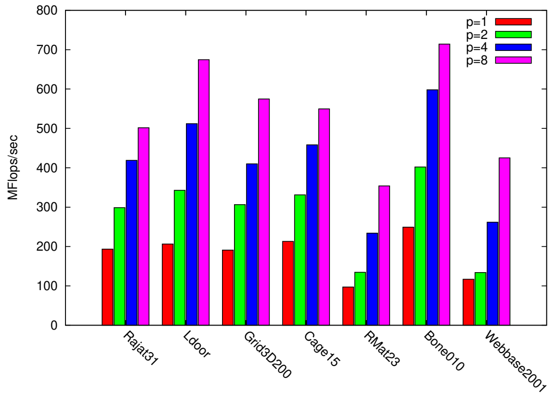


Figure 12: CSB_SpMV_T performance on Opteron (larger matrices).

Processors	CSB_SpMV 1–7	CSB_SpMV 8–14	CSB_SpMV_T 1–7	CSB_SpMV_T 8–14
$P = 2$	1.65	1.70	1.44	1.49
$P = 4$	2.34	2.49	2.07	2.30
$P = 8$	2.75	3.03	2.81	3.16

Figure 13: Average speedup results for relatively smaller (1–7) and larger (8–14) matrices. These experiments were conducted on Opteron.

of as the exact performance on p threads (e.g., $p = 8$ is the best performance achieved for $5 \le p \le 8$).

In general, we observed better speedups for larger problems. For example, the average speedup of CSB_SpMV for the first seven matrices was 2.75 on 8 processors, whereas it was 3.03 for the second set of seven matrices with more nonzeros. Figure 13 summarizes these results. The speedups are relative to the CSB code running on a single processor, which Figure 1 shows is competitive with serial CSR codes. In another study [41] on the same Opteron architecture, multicore-specific parallelization of the CSR code for 4 cores achieved comparable speedup to what we report here, albeit on a slightly different sparse-matrix test suite. That study does not consider the $y \leftarrow A^{\mathrm{T}}x$ operation, however, which is difficult to parallelize with CSR but which achieves the same performance as $y \leftarrow Ax$ when using CSB.

For CSB_SpMV on 4 processors, CSB reached its highest speedup of 2.80 on the RMat23 matrix, showing that this algorithm is robust even on a matrix with highly irregular nonzero structure. On 8 processors, CSB_SpMV reached its maximum speedup of 3.93 on the Webbase2001 matrix, indicating the code's ability to handle very large matrices without sacrificing parallel scalability.

Sublinear speedup occurs only after the memory-system bandwidth becomes the bottleneck. This bottleneck occurs at different numbers of cores for different matrices. In most cases, we observed nearly linear speedup up to 4 cores. Although the speedup is sublinear beyond 4 cores, in every case (except CSB_SpMV on Mittelmann), we see some performance improvement going from 4 to 8 cores on Opteron. Sublinear speedup of SpMV on superscalar multicore architectures has been noted by others as well [41].

We conducted an additional experiment to verify that performance was limited by the memory-system bandwidth, not by lack of parallelism. We repeated each scalar multiply-add operation of the form $y_i \leftarrow y_i + A_{ij}x_j$ a fixed number t of times. Although the resulting code computes $y \leftarrow tAx$, we ensured that the compiler did not optimize away any multiply-add operations. Setting $t = 10$ did not affect the timings significantly—flops are indeed essentially free—but, for $t = 100$, we saw almost perfect linear speedup up to 16 cores, as shown in Figure 14. We performed this experiment with Asic_320k, the smallest matrix in the test suite, which should

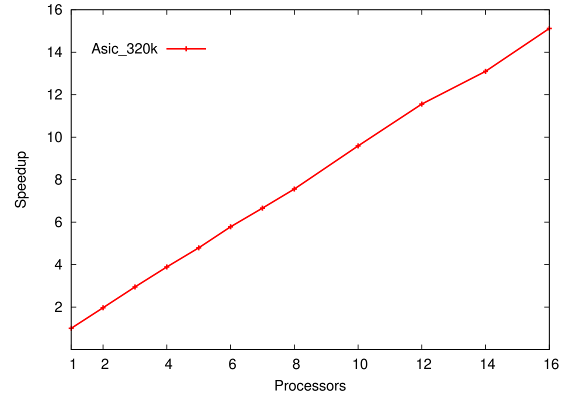


Figure 14: Parallelism test for CSB_SpMV on Asic_320k obtained by artificially increasing the flops per byte. The test shows that the algorithm exhibits substantial parallelism and scales almost perfectly given sufficient memory bandwidth.

exhibit the least parallelism. Asic_320k is also irregular in structure, which means that our balance heuristic does not apply. Nevertheless, CSB_SpMV scaled almost perfectly given enough flops per byte.

The parallel performance of CSB_SpMV and CSB_SpMV_T is generally not affected by highly uneven row and column nonzero counts. The highly skewed matrices RMat23 and Webbase2001 achieved speedups as good as for matrices with flat row and column counts. An unusual case is the Torso matrix, where both CSB_SpMV and CSB_SpMV_T were actually slower on 2 processors than serially. This slowdown does not, however, mark a plateau in performance, since Torso speeds up as we add more than 2 processors. We believe this behavior occurs because the overhead of intrablock parallelization is not amortized for 2 processors. Torso requires a large number of intrablock parallelization calls, because it is unusually irregular and dense.

Figure 15 shows the performance of CSB_SpMV on Harpertown for a subset of test matrices. We do not report performance for CSB_SpMV_T, as it was consistently close to that of CSB_SpMV. The performance on this platform levels off beyond 4 processors for most matrices. Indeed, the average Mflops/sec on 8 processors is only 3.5% higher than on 4 processors. We believe this plateau results from insufficient memory bandwidth. The continued speedup on Opteron is due to its higher ratio of memory bandwidth (bytes) to peak performance (flops) per second.

Figure 16 summarizes the performance results of CSB_SpMV for the same subset of test matrices on Nehalem. Despite having only 4 physical cores, for most matrices, Nehalem achieved scaling up to 8 threads thanks to hyperthreading. Running 8 threads was necessary to utilize the processor fully, because hyperthreading fills the pipeline more effectively. We observed that the improvement from oversubscribing is not monotonic, however, because running more threads reduces the effective cache size available to each thread. Nehalem's point-to-point interconnect is faster than Opteron's (a generation old Hypertransport 1.0), which explains its better speedup values when comparing the 4-core performance of both architectures. Its raw performance is also impressive, beating both Opteron and Harpertown by large margins.

To determine CSB's competitiveness with a conventional CSR code, we compared the performance of the CSB serial code with plain OSKI using no matrix-specific optimizations such as register or cache blocking. Figures 17 and 18 present the results of the comparison. As can be seen from the figures, CSB achieves similar serial performance to CSR.

In general, CSR seems to perform best on ***banded matrices***, all

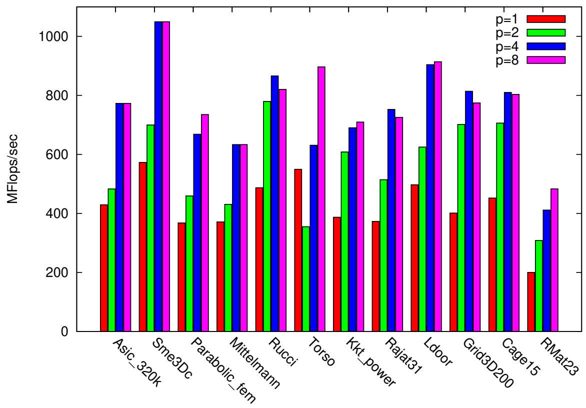


Figure 15: CSB_SpMV performance on Harpertown.

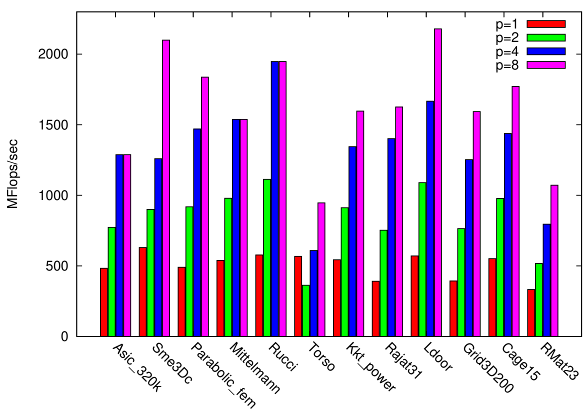


Figure 16: CSB_SpMV performance on Nehalem.

of whose nonzeros are located near the main diagonal. (The maximum distance of any nonzero from the diagonal is called the matrix's ***bandwidth***, not to be confused with memory bandwidth.) If the matrix is banded, memory accesses to the input vector x tend to be regular and thus favorable to cacheline reuse and automatic prefetching. Strategies for reducing the bandwidth of a sparse matrix by permuting its rows and columns have been studied extensively (see [11, 37], for example). Many matrices, however, cannot be permuted to have low bandwidth. For matrices with scattered nonzeros, CSB outperforms CSR, because CSR incurs many cache misses when accessing the x vector. An example of this effect occurs for the RMat23 matrix, where the CSB implementation is almost twice as fast as CSR.

Figure 19 compares the parallel performance of the CSB algorithms with Star-P. Star-P's blockrow data distribution does not afford any flexibility for load-balancing across processors. Load balance is not an issue for matrices with nearly flat row counts, including finite-element and finite-difference matrices, such as Grid3D200. Load balance does become an issue for skewed matrices such as RMat23, however. Our performance results confirm this effect. CSB_SpMV is about 500% faster than Star-P's SpMV routine for RMat23 on 8 cores. Moreover, for any number of processors, CSB_SpMV runs faster for all the matrices we tested, including the structured ones.

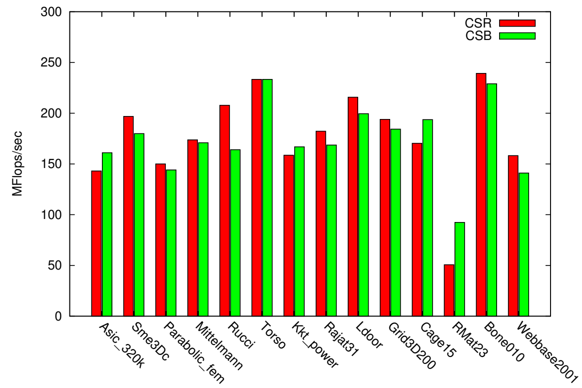


Figure 17: Serial performance comparison of SpMV for CSB and CSR.

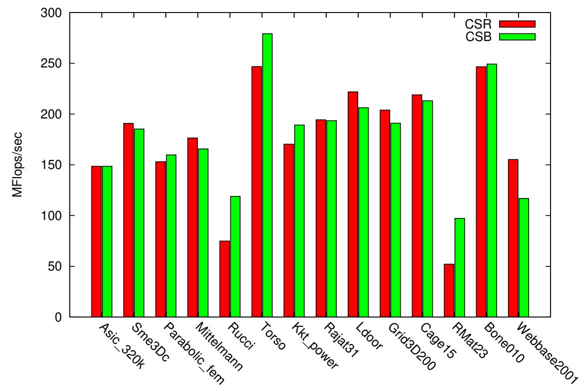


Figure 18: Serial performance comparison of SpMV_T for CSB and CSR.

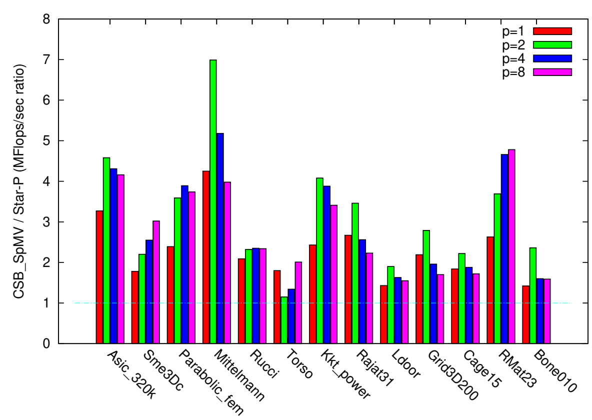


Figure 19: Performance comparison of parallel CSB_SpMV with Star-P, which is a parallel-dialect of Matlab. The vertical axis shows the performance ratio of CSB_SpMV to Star-P. A direct comparison of CSB_SpMV_T with Star-P was not possible, because Star-P does not natively support multiplying the transpose of a sparse matrix by a vector.

8 Conclusion

Compressed sparse blocks allow parallel operations on sparse matrices to proceed either row-wise or column-wise with equal facility. We have demonstrated the efficacy of the CSB storage format for SpMV calculations on a sparse matrix or its transpose. It remains to be seen, however, whether the CSB format is limited to SpMV calculations or if it can also be effective in enabling parallel algorithms for multiplying two sparse matrices, performing LU-, LUP-, and related decompositions, linear programming, and a host of other problems for which serial sparse-matrix algorithms currently use the CSC and CSR storage formats.

The CSB format readily enables parallel SpMV calculations on a symmetric matrix where only half the matrix is stored, but we were unable to attain one optimization that serial codes exploit in this situation. In a typical serial code that computes $y \leftarrow Ax$, where $A = (a_{ij})$ is a symmetric matrix, when a processor fetches $a_{ij} = a_{ji}$ out of memory to perform the update $y_i \leftarrow y_i + a_{ij}x_j$, it can also perform the update $y_j \leftarrow y_j + a_{ij}x_i$ at the same time. This strategy halves the memory bandwidth compared to executing CSB_SPMV on the matrix, where $a_{ij} = a_{ji}$ is fetched twice. It remains an open problem whether the 50% savings in storage for sparse matrices can be coupled with a 50% savings in memory bandwidth, which is an important factor of 2, since it appears that the bandwidth between multicore chips and DRAM will scale more slowly than core count.

References

[1] M. D. Adams and D. S. Wise. Seven at one stroke: results from a cache-oblivious paradigm for scalable matrix algorithms. In *MSPC*, pages 41–50, New York, NY, USA, 2006. ACM.

[2] D. Bader, J. Feo, J. Gilbert, J. Kepner, D. Koester, E. Loh, K. Madduri, B. Mann, and T. Meuse. HPCS scalable synthetic compact applications #2. Version 1.1.

[3] G. E. Blelloch. Programming parallel algorithms. *CACM*, 39(3), Mar. 1996.

[4] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *PPoPP*, pages 207–216, Santa Barbara, California, July 1995.

[5] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *JACM*, 46(5):720–748, Sept. 1999.

[6] A. Buluç and J. R. Gilbert. On the representation and multiplication of hypersparse matrices. In *IPDPS*, pages 1–11, 2008.

[7] U. Catalyurek and C. Aykanat. A fine-grain hypergraph model for 2D decomposition of sparse matrices. In *IPDPS*, page 118, Washington, DC, USA, 2001. IEEE Computer Society.

[8] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley. Stanford webbase components and applications. *ACM Transactions on Internet Technology*, 6(2):153–186, 2006.

[9] Cilk Arts, Inc., Burlington, MA. *Cilk++ Programmer's Guide*, 2009. Available from `http://www.cilk.com/`.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[11] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 24th National Conference*, pages 157–172, New York, NY, USA, 1969. ACM.

[12] T. A. Davis. University of Florida sparse matrix collection. *NA Digest*, 92, 1994.

[13] T. A. Davis. *Direct Methods for Sparse Linear Systems*. SIAM, Philadelpha, PA, 2006.

[14] J. Dongarra. Sparse matrix storage formats. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[15] J. Dongarra, P. Koev, and X. Li. Matrix-vector and matrix-matrix multiplication. In Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst, editors, *Templates for the Solution of Algebraic Eigenvalue Problems: a Practical Guide*. SIAM, 2000.

[16] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, New York, 1986.

[17] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman. Yale sparse matrix package I: The symmetric codes. *International Journal for Numerical Methods in Engineering*, 18:1145–1151, 1982.

[18] E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.

[19] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin. Reducers and other Cilk++ hyperobjects. In *SPAA*, Calgary, Canada, 2009.

[20] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *SIGPLAN*, pages 212–223, Montreal, Quebec, Canada, June 1998.

[21] A. George and J. W. Liu. *Computer Solution of Sparse Positive Definite Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[22] J. R. Gilbert, G. L. Miller, and S.-H. Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[23] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM J. Matrix Anal. Appl*, 13:333–356, 1991.

[24] E.-J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[25] K. Kourtis, G. Goumas, and N. Koziris. Optimizing sparse matrix-vector multiplication using index and value compression. In *Computing Frontiers (CF)*, pages 87–96, New York, NY, USA, 2008. ACM.

[26] J. Leskovec, D. Chakrabarti, J. M. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *PKDD*, pages 133–145, 2005.

[27] K. P. Lorton and D. S. Wise. Analyzing block locality in Morton-order and Morton-hybrid matrices. *SIGARCH Computer Architecture News*, 35(4):6–12, 2007.

[28] H. M. Markowitz. The elimination form of the inverse and its application to linear programming. *Management Science*, 3(3):255–269, 1957.

[29] G. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Canada, Mar. 1966.

[30] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[31] R. Raman and D. S. Wise. Converting to and from dilated integers. *IEEE Trans. on Computers*, 57(4):567–573, 2008.

[32] Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, Philadelpha, PA, second edition, 2003.

[33] N. Sato and W. F. Tinney. Techniques for exploiting the sparsity of the network admittance matrix. *IEEE Trans. Power Apparatus and Systems*, 82(69):944–950, Dec. 1963.

[34] V. Shah and J. R. Gilbert. Sparse matrices in Matlab*P: Design and implementation. In *HiPC*, pages 144–155, 2004.

[35] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, third edition, 2000.

[36] W. Tinney and J. Walker. Direct solutions of sparse network equations by optimally ordered triangular factorization. *Proceedings of the IEEE*, 55(11):1801–1809, Nov. 1967.

[37] S. Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM J. Research and Development*, 41(6):711–726, 1997.

[38] B. Vastenhouw and R. H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Rev.*, 47(1):67–95, 2005.

[39] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16(1):521+, 2005.

[40] J. Willcock and A. Lumsdaine. Accelerating sparse matrix computations via data compression. In *ICS*, pages 307–316, New York, NY, USA, 2006. ACM.

[41] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.