Efficient Sparse Matrix-Matrix Multiplication on Multicore Architectures*

Adam Lugowski† John R. Gilbert‡

May 8, 2014

Abstract

We describe a new parallel sparse matrix-matrix multiplication algorithm in shared memory using a quadtree decomposition. Our preliminary implementation is nearly as fast as the best sequential method on one core, and scales well to multiple cores.

1 Introduction

Sparse matrix-matrix multiplication (or *SpGEMM*) is a key primitive in some graph algorithms (using various semirings) [9] and in numeric problems such as algebraic multigrid [15]. Multicore shared memory systems can solve very large problems [16], or can be part of a hybrid shared/distributed memory high-performance architecture.

Two-dimensional decompositions are broadly used in state-of-the-art methods for both dense [17] and sparse [1, 2] matrices. Quadtree matrix decompositions and algorithms have a long history [5, 6, 14, 19, 20], including recursive matrix multiplication [18].

We propose a new sparse matrix data structure and the first highly-parallel sparse matrix-matrix multiplication algorithm designed specifically for shared memory.

2 Quadtree Representation

Our basic data structure is a 2D quadtree matrix decomposition. Unlike previous work that continues the quadtree until elements become leaves, we instead terminate the quadtree early and store the elements in large leaf blocks. This arrangement brings the best of both worlds; the quadtree provides isolation and chunking, and the large leaf blocks provide locality and a way to amortize tree costs.

There are many answers to the question of when to stop subdivision. We use a simple strategy: subdivide until either leaf *nnz* or leaf size in bytes is below a threshold. This threshold can be fixed or dynamically chosen to provide sufficient parallelism for a particular matrix on a particular machine. The former approach aims at efficient utilization of fixed resources such as caches, while the latter method aims to minimize the number of hypersparse blocks and total per-block overhead.

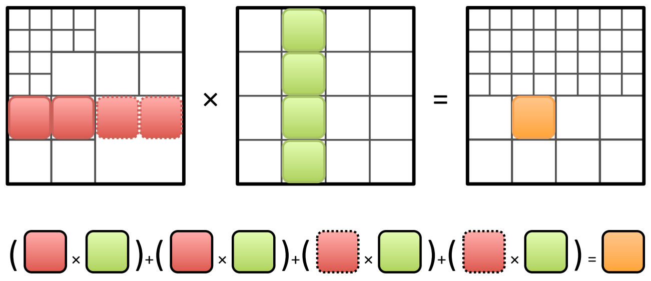


Figure 1: Computation of a result block using a list of pairwise block multiplications.

Inner blocks form the internal nodes of the quad tree. Each inner block is a container for four other blocks. Each child can be null, a leaf, or another inner block, and represents one quadrant of the parent inner block. Note that subdivisions always occur on powers of 2; hence, position in an inner block implies the high-order bits of row and column indices of the children. This allows the leaves to use smaller indices than the matrix dimensions appear to require. We do not, however require the matrix to have dimensions that are powers of 2.

The leaf blocks store the matrix elements in $(row, col, value)$ triples form. Row and column indices can be 8, 16, 32 or 64-bit unsigned integers, where the minimum index size that fits the block dimensions is chosen at runtime. The type of the values is defined by the user.

A shadow block is a block that provides a view of a subset of a TriplesBlock's elements. This is useful when the blocks of two different quadtrees need to be matched. Depending on the two trees' decompositions, an inner block may be matched with a leaf block. If this is undesirable, we may perform a *shadow subdivision* of the leaf block.

In a shadow subdivision a new inner block is created and populated with four shadow blocks that together return the same data as the original TriplesBlock. The original TriplesBlock's elements are scanned once, and the shadow each one belongs to is determined with a simple bit comparison of its row and column indices. A shadow block doesn't own its data; rather it is a view of a part of another leaf block. Its data structure is a pointer to the original TriplesBlock and an array of offsets of each element. It and its parent inner block are

*Supported by Contract #618442525-57661 from Intel Corp. and Contract #8-482526701 from the DOE Office of Science.

†CS Dept., UC Santa Barbara, alugowski@cs.ucsb.edu

‡CS Dept., UC Santa Barbara, gilbert@cs.ucsb.edu

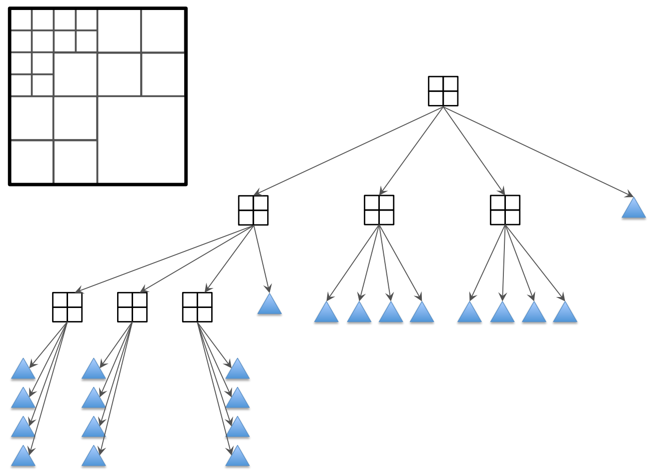


Figure 2: Quadtree of an adjacency matrix of a power law graph. This is matrix A in our running example in Figure 6.

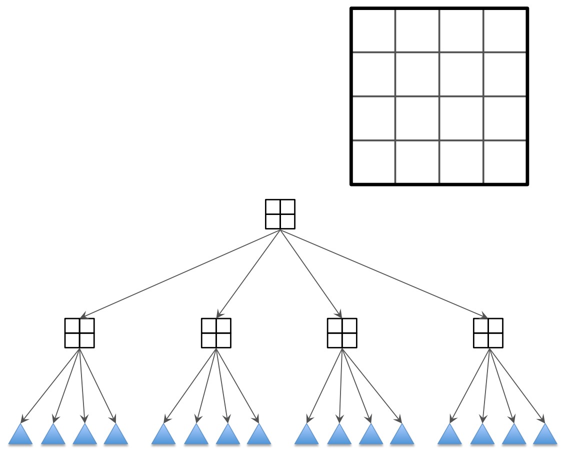


Figure 3: Quadtree of an adjacency matrix of an Erdős-Rényi graph. This is matrix B in our running example in Figure 6.

considered temporary and are expected to be destroyed by the end of the operation that created them. For the purposes of read-only algorithms, a shadow block *is* a leaf block.

In our implementation, a shadow block with nnz nonzeros consists of an $O(nnz)$ space array of indices into the original TriplesBlock. Another possible scheme is to partially sort the TriplesBlock into four quadrants, which allows each shadow block to simply be an $O(1)$ space begin and end bound. This method has two problems. First, the partial sort is more expensive than a scan. Second, the original TriplesBlock is no longer in pure column order, which makes accessing its elements both more expensive and more complicated when this block is part of several tasks. Both problems can be solved by using Z-Morton [12] order instead of column order, which allows arbitrarily deep subdivisions. Z-Morton order, however, does not provide $O(1)$ lookups by row or column indices, which makes the sparse multiplication kernels asymptotically more expensive.

3 Pair-List Matrix Multiplication Algorithm

The quadtree decomposition suggests a natural recursive SpGEMM algorithm: recursively evaluate the following:

$$\begin{aligned} C_4 &= (A_1 \times B_1) + (A_2 \times B_3) \\ C_2 &= (A_1 \times B_2) + (A_2 \times B_4) \\ C_3 &= (A_3 \times B_1) + (A_4 \times B_3) \\ C_4 &= (A_3 \times B_2) + (A_4 \times B_4) \end{aligned} \tag{3.1}$$

This algorithm has a serious flaw, however. Each level of the recursion introduces an SpAdd operation in addition to the recursive multiplies. When thought of as a DAG of tasks, the multiplies are the leaves of a large tree of SpAdds. The number of SpAdds each block is involved in is equal to its depth in the tree. Unfortunately, there is no known method to perform an SpAdd in time proportional to only the FLOPs required. Instead, the total time of all additions is proportional to total FLOPs plus the size of the operands times the height of the tree. The add tree therefore imposes an unwanted log factor and becomes a significant bottleneck. Our algorithm reformulates the operations such that the **SpAdds can be inlined into the leaf multiplies**.

The algorithm consists of a **symbolic phase and a computational phase**. The *symbolic phase* generates an execution strategy, and the *computational phase* carries out that strategy. Each phase is itself a set of parallel tasks. **We are willing to temporarily reorganize data on-the-fly, and discard the changes after use**. This extra work does not add to the asymptotic complexity.

The source of parallelism of both phases comes from the recursive structure of the quadtree of C. Each internal node yields a symbolic phase task, and each leaf yields a computational phase task.

We choose to formulate a DAG of tasks and let a scheduling framework map those tasks to threads. Our algorithm does not perform scheduling; rather, we use a standard scheduling framework such as TBB, Cilk, or OpenMP.

3.1 Symbolic Phase

The symbolic phase divides computation of $C = A \times B$ into compute tasks such that each compute task *owns* (is the only writer to) a particular block of C and is supplied with a list of all the operands it needs to perform the multiplication.

Let C_{own} be a leaf block in C, and *pairlist* be the list of pairs of leaf blocks from A and B whose block inner product is C_{own}:

$$C_{own} = \sum_{i=1}^{|pairlist|} A_i \times B_i \tag{3.2}$$

The blocks A_i and B_i may be original leaf blocks or shadow blocks. The symbolic phase recursively determines all the C_{own} and corresponding *pairlist*. Equation (3.2) still contains additions, but in Section 3.3 we describe a method to evaluate (3.2) without explicit SpAdd steps.

To provide intuition for what we wish to accomplish, consider a dense $\beta \times \beta$ grid of blocks instead of a quadtree. The result matrix will contain β^2 blocks, each one the result of a block inner product between the corresponding block row of A and block column of B. The i^{th} block in the block row of A is matched with the i^{th} block in the block column of B in this block inner product. Therefore, we describe this block inner product with a list, named *pairlist*, with length β of pairs of blocks.

We now wish to accomplish the same task, but with two differently structured quadtrees of blocks instead of a dense grid. Different pairlists can have blocks of different sizes, though all the blocks in one pairlist are the same size. An element of an input matrix may participate in several pairlists with different block sizes, via shadow blocks.

The PairList algorithm's symbolic phase recursively determines all the C_{own} and corresponding *pairlist*. We begin with $C_{own} \leftarrow C$, and $pairlist \leftarrow [(A, B)]$.

If *pairlist* consists only of leaf blocks, spawn a compute task with C_{own} and *pairlist*.

If all the blocks in *pairlist* are divided, we divide C_{own} into four children with one quadrant each and recurse, rephrasing divided $C = A \times B$ using (3.2):

$$\begin{aligned} C_1 &= [(A_1, B_1), (A_2, B_3)] \\ C_2 &= [(A_1, B_2), (A_2, B_4)] \\ C_3 &= [(A_3, B_1), (A_4, B_3)] \\ C_4 &= [(A_3, B_2), (A_4, B_4)] \end{aligned} \tag{3.3}$$

In total, each recursive call receives a C_{own} and an entire list of pairs of blocks. For every pair in *pairlist*, insert two pairs into each child's *pairlist* according to the respective line in (3.3). Each child's *pairlist* is twice as long as the parent's *pairlist*, but totals only 4 sub-blocks to the parent's 8.

If *pairlist* includes both divided blocks and leaf blocks, we temporarily divide the leaves until all blocks in *pairlist* are equally divided. This temporary division creates shadow blocks as described in Section 2. Shadow subdivision resolves any differences in quadtree depth between the operands. It allows the symbolic phase to recurse until only leaves remain, which lets the compute phase only operate on leaves. See Figure 5 for an example. The shadow blocks persist only until the end of the SpGEMM.

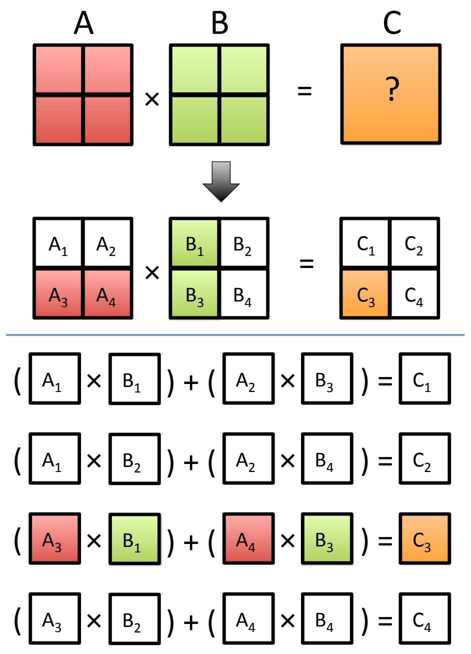


Figure 4: Illustration of Equation (3.3).

3.2 Symbolic Phase Example We illustrate the symbolic phase of a multiplication of two matrices by tracing how two result blocks' pair lists are generated. We use the running example in Figure 6. Operand A is more dense in a corner as might appear in an adjacency matrix of a power law graph. Operand B shows a uniform subdivision, as might appear in an adjacency matrix of an Erdős-Rényi [4] graph. Their respective quadtrees are illustrated in Figures 2 and 3.

In the figures, leaf blocks and compute tasks are denoted with rounded corners; shadow blocks and shadow subdivisions are denoted with dotted lines.

Both traces share the same root symbolic task. This task is initialized with the full problem: $pairlist = [(A, B)]$ and $C_{own} = C$. It sees that all blocks in *pairlist* are subdivided, so the recursive case applies. C_{own} is subdivided and a matching *pairlist* is generated according to (3.2) (as illustrated in Figure 4). Four new symbolic tasks are spawned, one for each newly divided C_{own} child. Our two traces diverge here; each

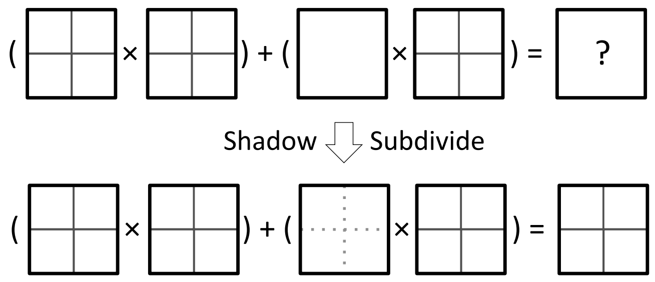


Figure 5: Division mismatch: a leaf block is paired with an inner block. A shadow subdivision of the leaf block yields an inner block that resolves the mismatch and allows another recursive step.

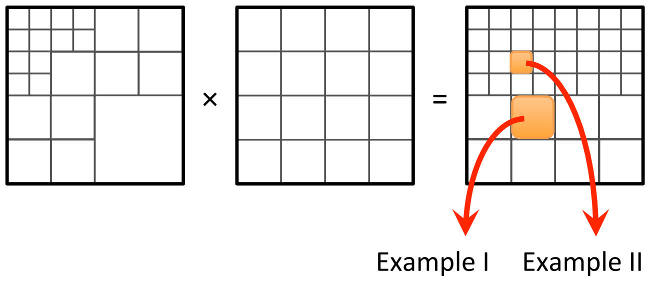


Figure 6: The running example. We wish to multiply an RMAT matrix with an adjacency matrix of an Erdős-Rényi graph. The quadtree for the RMAT is shown in Figure 2, and the ER in Figure 3.

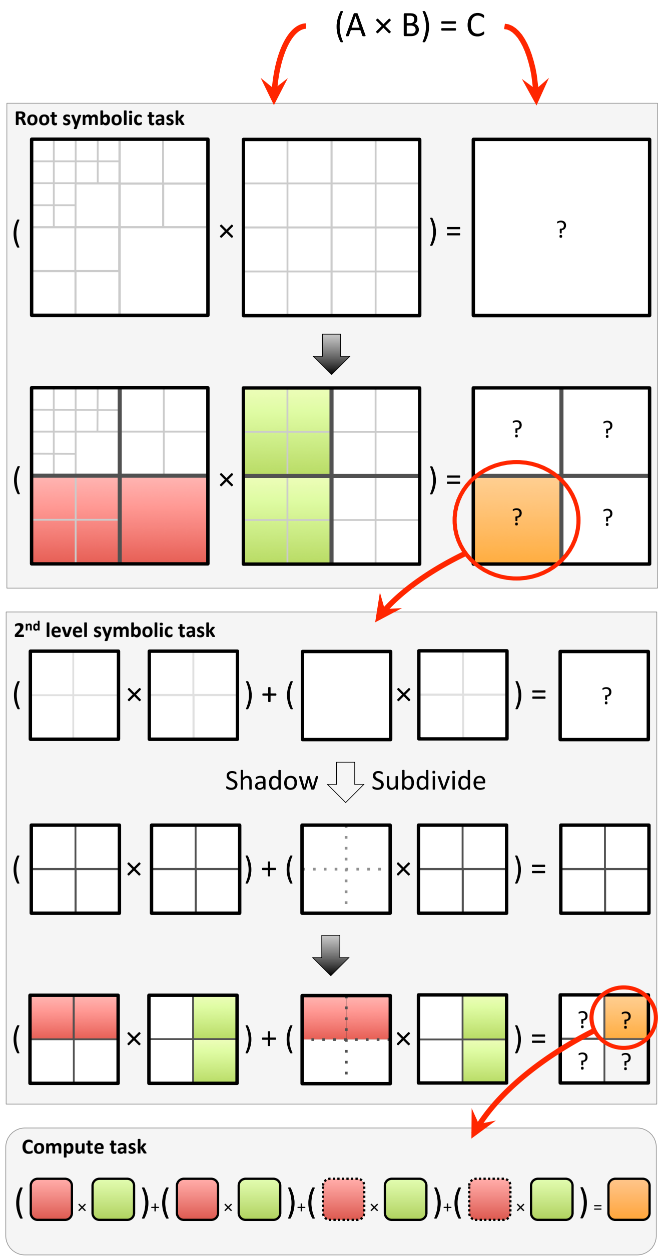


Figure 7: Example Trace I: The root symbolic task applies the recursive case. The next recursive symbolic task has a mix of inner block and leaves, so performs a shadow subdivide. The next recursion are all leaf tasks, so are turned into compute tasks.

one follows the recursive call on a different child.

Example Trace I follows the third (bottom left) child. It is fully illustrated in Figure 7. The second level symbolic task has a *pairlist* that consists of three inner blocks and one leaf. This requires a shadow subdivision of the leaf. The recursion then continues, spawning four more symbolic tasks. Each one of these four consists of only leaves, so they simply spawn compute tasks.

Example Trace II follows the first (top left) child of the root symbolic task. This trace is fully illustrated in Figure 8. The second level symbolic task has a *pairlist* that consists of all inner blocks, so the recursive case is trivially applied again. This spawns four more symbolic tasks, and we choose to follow the fourth (bottom right) child. This third level symbolic task has a *pairlist* with one inner block and seven leaves. The leaves must be shadow subdivided so another recursive case can be applied. These recursive children contain only leaves. Some are original leaves, corresponding to the most dense part of A. The rest are shadows, both from less dense parts of A, and from the generally less dense B. The final recursion, then, can spawn compute tasks.

3.3 Computational Phase This phase consists of tasks that each compute one block inner product (3.2). We present the final approach in Algorithm 1 and describe it below. Observe that each compute task is lock-free because it only reads from the blocks in *pairlist* and only writes to C_{own}.

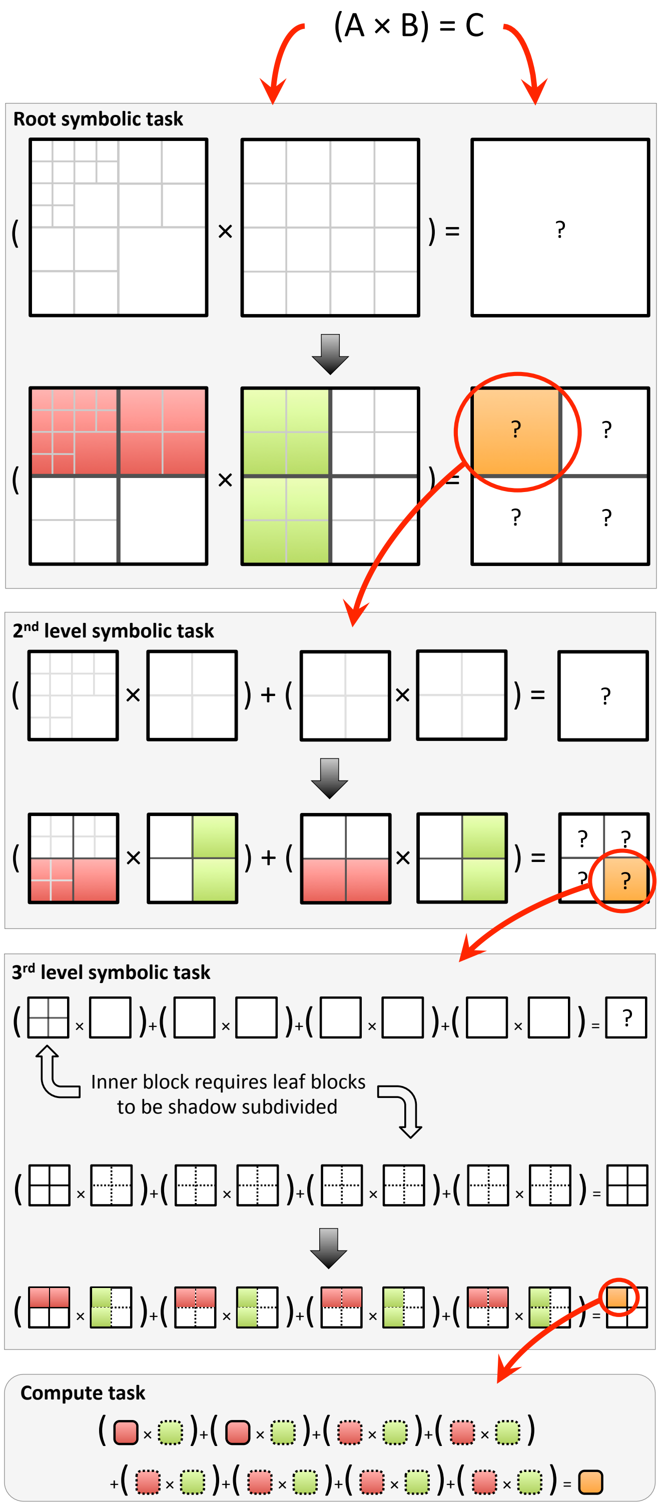


Figure 8: Example Trace II: Trace that requires 3 levels of symbolic tasks.

We extend Gustavson's sequential sparse matrix multiplication algorithm [8]. Gustavson computes the product of column j of B and A using a "sparse accumulator", or *SPA*. The SPA can be thought of as a dense auxiliary vector, or hash map, that efficiently accumulates sparse updates to a single column of C_{own}. Gustavson's algorithm reads both A and B column-by-column, but their columns are selected differently. The algorithm reads the non-empty columns of B in order, but performs random lookups of columns in A. To facilitate these access patterns for our $(row, col, value)$ triples storage, we *organize* the column-sorted triples. A *column organizer* is an auxiliary structure that allows quick access to particular columns of a block. **Due to different access patterns for blocks A and B, we organize them differently**.

Algorithm 1 Compute Task's Multi-Leaf Multiply

```
Require: $C_{own}$ and $pairlist$
Ensure: Complete $C_{own}$
  for all $(A_b, B_b)$ in $pairlist$ do
    organize $A_b$ columns with hash map or CSC
    organize $B_b$ columns into list
  end for
  merge all $B$ organizers into $combined\_B\_org$
  for all (column $j$, $PairList_j$) in $combined\_B\_org$ do
    $SPA \leftarrow \{\}$
    for all $(A_b, B_b)$ in $PairList_j$ do
      for all non-null $k$ in column $j$ in $B_b$ do
        accumulate $B_b[k, j] \times A_b[:, k]$ into $SPA$
      end for
    end for
    copy contents of $SPA$ to $C_{own}[:, j]$
  end for
```

The first type of column organizer is designed for constant-time lookup of a particular column i in A. We provide two methods to achieve this. The first is a hash map with an entry $i \rightarrow (offset_i, length_i)$ for each non-empty column i. The second is a CSC-like array of offsets of the first element of a column. Both offer $O(1)$ lookups of a particular column i, but the CSC-like method trades a faster constant for $O(n)$ space.

The second type of column organizer allows iteration over non-empty columns B. We generate a list of tuples $(j, offset_j, length_j)$.

All column organizers are generated with a single scan of only the column indices. Therefore each one takes linear time to generate. For maximum parallelism, the organizers can be generated in each compute task. This means each block is organized many times, once by each compute task it is used in. This cost is negligible for small to medium matrices, but can be greatly reduced by caching the organizers.

The column organizers allow us to efficiently use Gustavson's algorithm on our triples to evaluate the multiplies in (3.2). We show that if all pairwise block multiplies in a computational task are performed simultaneously then they can be interleaved in such a fashion that **the addition step is inlined into the multiply step**.

The key to this inlining is the SPA. Gustavson uses the SPA to accumulate the sparse updates to a single column j of C_{own}. Observe that in a blocked algorithm every non-null column j in any B in the pair list will lead to its own SPA for column j of that pair's partial result. The add step's only function is to accumulate all the partial column j results into one. Our key contribution is do all updates to column j together, allowing us to use the same SPA for them all. Since there are no further updates to column j, no add step is necessary.

Another way to picture this process is to observe that the A blocks represent a short-and-fat slice of the matrix A, and the B blocks represent a tall-and-skinny slice of the matrix B. C_{own} is the inner product of these two slices. When the slices are thought of as whole matrices, this inner product already handles the addition properly. Our contribution can be thought of as virtually merging the A and B blocks into such slices.

Our addition to Gustavson is a mechanism that combines columns j from all blocks B_i in *pairlist* to present a view of the entire column j from matrix B. This *organizer combiner* is like the second column organizer, but generalized to cover the non-empty columns in *all* blocks B instead of just one. We accomplish this with a structure that combines the B organizers with the property that all entries of column j are together.

We supply two ways to implement an organizer combiner. First is an ordered multi-map of $j \rightarrow (B_{source}, \mathit{offset})$. We fill the multimap from each B organizer. The second is a dense 2D array of the same entries as the multi-map values. This method escapes a $\log n$ insert time at the cost of higher space usage.

Our extensions to Gustavson therefore consist of column organizers, a column organizer combiner, and finally an interleaving of inner products of multiple block pairs.

We draw the reader's attention to a pattern in our auxiliary data structures: we provide two versions for each structure that requires random access. The traditional implementations of these structures use a dense array (like CSC column pointers or a dense vector SPA), and are the only part of the QuadMat data structure and SpGEMM implementation that depend on the matrix dimensions m or n. This approach works superbly for matrices with dimensions small enough for these structures to fit in available memory. However, we wish to break this dependency in order to support huge matrix dimensions. We therefore always provide an alternative structure that has the same $O()$ time complexity (but with a higher constant) that does not depend on the matrix dimensions. The choice of which version to use is made at runtime.

Our dense SPA is similar to the traditional one [7]. It consists of two arrays of length m. The first, *vals* is the actual values (such as doubles). The second is an array of full/empty bits. Lastly, a *used_elements* array lists the i where $\mathit{vals}[i]$ is full. Our alternative SPA implementation uses a hashmap $i \rightarrow (\mathit{val})$ instead of the dense arrays.

3.4 Post Processing The symbolic and compute phases produce a valid result, but this result might not be subdivided appropriately. If this is undesirable, a *post-processing* phase can correct the problem.

If a resulting block is too dense, *i.e.* its $\mathit{nnz} > \mathit{threshold}$, it needs to be subdivided. A subdivision resembles a shadow subdivide, but the result is permanent. This subdivision can be done by a single task as soon as the compute task finishes building the result.

If a resulting block is too sparse, *i.e.* the sum nnz of it and its siblings $\leq \mathit{threshold}$, it needs to be coalesced with its siblings in the quadtree. Coalescing is the opposite of subdivision and can only be attempted after all children of a result inner block are computed. Coalescing also needs to be performed recursively up the quadtree; it is possible that the entire result matrix is nearly empty and needs to be coalesced into a single block.

4 Choice of Division Threshold

QuadMat has a tuning parameter in the form of the subdivision strategy. In this work it is the value of the division threshold as explained in Section 2.

In our experiments, we decided to avoid hand-tuning individual SpGEMM problems by using a one-size-fits-all algorithm to choose a threshold for a particular problem. We only allow ourselves to use information known at the start of the problem, namely the processing environment and the dimensions and nonzero count of the operands. An optimal algorithm is a matter of ongoing research, but for the purposes of these experiments we make an educated guess and choose a division threshold=max($50k$, $\mathit{largest\_nnz}/80$).

The choice of division threshold has wide ramifications. The threshold determines the parallelism of the computation. At one extreme, if we set the threshold=nnz, the entire matrix is one single leaf block and potential parallelism is 1. At the other extreme we have a very small threshold with immense potential parallelism due to the fact that the compute blocks are independent. This, however, leads to an increase in the number of blocks and block overhead, mainly column organization, and an increase in the likelihood that each block is *hypersparse* ($nnz \ll n$).

The increased cost of column organization is mainly due to the fact that this preliminary work does not yet implement organizer caching. Observe that each block is used in many compute tasks. Without caching, each compute task performs its own organization of its operands. This leads to duplicate work, and becomes significant on some problems with small thresholds. This is the primary reason why we chose a relatively large threshold. When ongoing work in caching is complete we expect to be able to remove this restriction.

A smaller division threshold also leads to each block becoming less dense. To illustrate, assume that matrix M with dimensions n has an average of c nonzeros per column, or $nnz = cn$. As we divide each column into b blocks, each block owns c/b column nonzeros. As b increases with the division threshold, the nonzero count of each block approaches 0 and the block becomes hypersparse.

Hypersparse blocks have two important consequences. First, the dense structures (organizers, SPA) that depend on n and not on nnz become inefficient. CombBLAS solves this problem by using a *Doubly-Compressed Sparse Columns* (DCSC) datastructure, which is CSC with the column pointers compressed.

Second, hypersparse block inner products have lower utility for every lookup into A. Recall that the heart of the compute phase is "accumulate $B_b[k, j] \times A_b[:, k]$ into *SPA*". In an undivided M each nonzero with row k in B will look up column k in A once. This column may be empty (a miss), but assume it has c nonzeros. The algorithm then accumulates all c elements into the SPA. If we do the same on a divided M, column k is now in b parts. In total, there will now be b lookups instead of one, but the same number of accumulation operations to amortize the cost.

The hypersparse effect can be reduced with prevention and mitigation. Prevention means increasing the division threshold. In practice this likely means that the optimal division threshold is the maximum one that provides enough potential parallelism. This implies a threshold that depends on the number of threads used; we did not pursue this in our reported experiments. In qualitative experiments, however, we notice an increase in performance on low thread counts with higher thresholds.

The hypersparse effect can be mitigated by reducing the cost of a lookup miss. If the lookup is in cache then it can incur minimal penalty. Ongoing organizer work should address this with a hierarchal organizer (similar to DCSC) that allows many lookup misses to fail quickly using the same (cached) memory locations. A smaller threshold results in smaller blocks, and therefore a larger portion of the organizer can be in cache.

5 Experiments and Comparisons

5.1 Experimental Design We implemented our algorithm in C++, using the Threading Building Blocks (TBB) framework [13] for task parallelism. We compare it to the fastest serial and parallel codes available. We use an Intel Westmere-EX machine with four E7-8870 @ 2.40GHz processors for a total of 40 physical cores and 80 threads. The machine has 256 GB RAM.

5.1.1 Codes We compare against the leading serial code, CSparse [3], and the parallel code Combinatorial BLAS [2]. For this paper, we only consider SpGEMM kernels.

CSparse is a small sparse matrix package written in C. It includes implementations for a wide range of sparse matrix algorithms that are either asymptotically optimal or fast in practice. The primary drawback to CSparse is that it is single threaded. Nevertheless, it is considered a leading sparse matrix code and offers a strong benchmark.

The Combinatorial BLAS (CombBLAS) is a library written in highly-templated C++ and MPI that offers a small set of linear algebraic kernels that can be used as building blocks for the most common graph-analytic algorithms. Graph abstractions can be built on top of its sparse matrices, taking advantage of its existing best practices for handling parallelism in sparse linear algebra. The main data structures are sparse matrices and vectors which are distributed in a two-dimensional processor grid for scalability. This means that the CombBLAS requires a square number of processes. CombBLAS is written for distributed memory, but we compare our shared-memory code with it as it is a leading parallel SpGEMM code.

5.1.2 Datasets We present a set of problems that consist of a single sparse matrix multiplication $A \times B$ or a triple product $A \times B \times C$. We generate three types of matrices, and two randomly permuted variants, to serve as the base of our problem set as described in Table 1.

Kronecker product (RMAT) matrices [11] approximate a power-law degree distribution among vertices. We use quadrant edge probabilities of $[.57, .19, .19, .05]$ and a fill factor of 16. We also symmetrize the matrix to model an undirected graph. Each RMAT is labeled with its *scale*, where the dimensions of the matrix are

Table 1: Dataset categories. Each SpGEMM problem's name specifies the matrix used and the operation. The matrix name is a concatenation of *Base*, *Scale*, and *RP* from this table. The operation is denoted by a *suffix* from Section 5.1.3.

Base	Scale	Randomly Permuted	Matrix Dim.	Approx. *nnz*
Flat random:				
ER	18 or 20		2^{scale}	$32 * 2^{scale}$
Power law random:				
rmat	16 or 18		2^{scale}	$32 * 2^{scale}$
Power law random (randomly permuted):				
rmat	16 or 18	RP	2^{scale}	$32 * 2^{scale}$
3D structured mesh:				
torus3D	150 or 200		$scale^3$	$7 * scale^3$
3D structured mesh (randomly permuted):				
torus3D	150 or 200	RP	$scale^3$	$7 * scale^3$
Algebraic multigrid:				
AMG	150 or 200		$scale^3$	$27 * scale^3$

$n = 2^{scale}$. The maximum possible *nnz* is $32n$; however due to a large number of collisions in the dense regions the actual number can be substantially less.

We generate adjacency matrices for Erdős-Rényi graphs with similar vertex and edge counts to our RMAT graphs. Each ER graph has $n = 2^{scale}$ vertices and about $32n$ edges.

A 3D torus mesh serves to represent 3D geometic mesh applications. A mesh size of d contains d^3 vertices, each with a connection to its six neighbors and itself. Therefore, the sparse matrix has dimension d^3 with $7d^3$ nonzeros.

Finally we consider a simple algebraic multigrid application. We consider a 3D rectahedral mesh of dimension d, with d^3 cells, which performs a linear combination of its 27 neighbors.

Each dataset has a scale parameter as described. For the RMAT and torus datasets we also include a randomly-permuted variant, denoted with a *RP* suffix. This variant shows the effect of nonzero distribution. To ensure compatibility with all codes, all datasets only contain numeric elements of type *double*.

5.1.3 Problems

We generate SpGEMM problems from the datasets in several ways, each marked by a distinct suffix to the dataset name:

1. Suffix `_sq`: We square the matrix.
2. Suffix `_perm`: We randomly permute the matrix rows by left multiplying it by a generated random permutation matrix.
3. Suffix `_sub`: We select half the rows and half the columns of the matrix by a triple product. This operation is also called SpRef.
4. Suffix `_cont`: Finally, we generate a set of three matrices that approximate the contraction step of algebraic multigrid. We contract a dimension d matrix with d^3 cells to a dimension $d/2$ matrix with $d^3/8$ cells. This entails a $R \times A \times P$ triple product.

The complete set of problems is described in Tables 2 and 3 in the Appendix.

5.1.4 Measurements

For each problem we calculate the number of non-zero arithmetic operations (floating-point multiplies and additions) that occur. We then run each code/number of cores combination and record the elapsed time.

This data allows us to make a variety of comparisons. We can determine serial efficiency by looking at the $p = 1$ results. We can determine strong scaling by comparing increasing processor counts on the same problem, or weak scaling by comparing larger generated problems on the same number of processors. We can also compare to the serial CSparse code to determine when parallelism becomes profitable.

Additionally, we probe QuadMat by profiling its behavior on one core.

5.2 Results

We ran QuadMat with blocksize threshold=max($50k$, *largest_nnz*/80), with a naïve index caching implementation, no post-processing phase, and only dense versions of auxiliary data structures. The raw elapsed times for each problem are listed in Tables 4 and 5 in the Appendix.

We analyze QuadMat's performance from several angles. First, we get a broad overview of the performance of all codes by comparing them to each other. We then explore the effect of nonzero distribution on the runtimes, and the effect of threshold choice on scalability. Finally we profile QuadMat execution.

5.3 Code Comparisons

The purpose of a shared-memory parallel code is to perform a task faster than a sequential code. In this vein we get a broad performance overview of both parallel codes by comparing the speedup each offers compared to CSparse. In Figure 9 we plot the speedup (or slowdown) of 1, 4, 16, 36, and 64-thread runs compared to single-threaded CSparse on each problem in our set.

We see many strengths of QuadMat and some weaknesses. QuadMat's strongest performance is on ER and

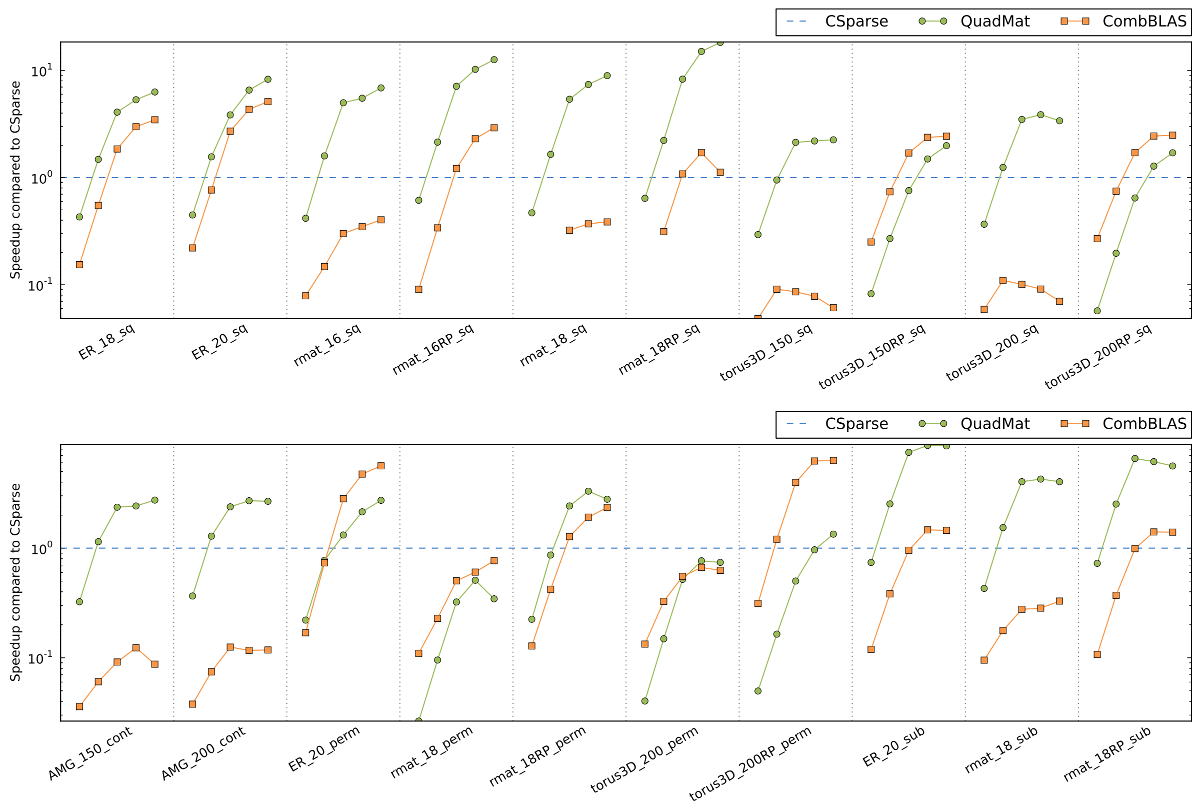


Figure 9: **(Preliminary)** Speedup compared to CSparse for CombBLAS and QuadMat on 1, 4, 16, 36, and 64 threads. Y-axis is in log scale. Note that the machine has 40 cores, so the 64 thread results are using multiple threads per core.

RMAT matrix squares, and the AMG contraction and submatrix extraction triple products. In 13 out of 20 problems QuadMat matches CSparse performance with four cores or fewer. QuadMat shows good speedup on the remaining problems, and does not match the CSparse sequential time on only two out of 20 problems. This shows that there are clearly some significant bottlenecks remaining.

We plot the same data as absolute values, namely FLOPS (or nonzero arithmetic operations per second) achieved. Figure 13 plots the same 1, 4, 16, 36, and 64-thread runs for CombBLAS and QuadMat, but they can now be directly compared to the FLOPS achieved by single-threaded CSparse. We observe that on some problems all codes suffer reduced FLOPS, while all are faster on others. The gap is large, two orders of magnitude.

5.3.1 Effects of Nonzero Distribution

We compare the effect of nonzero distribution on the various codes. This is most evident when the same problem is available in a highly structured and a randomly permuted form, namely torus squares. CSparse and QuadMat both perform better on the structured version, CombBLAS on the randomized version. There are two primary reasons for this.

Both CSparse and QuadMat use a dense lookup table for the columns (CSC and CSC-like dense organizer, respectively). This makes sequential reads of the columns very efficient. This locality is lost when the matrix is randomly permuted, and FLOPS performance approaches that of the ER squares.

The hypersparse algorithm used by CombBLAS does not allow it to benefit from this locality as much, so it is less affected by its loss. On the other hand, CombBLAS uses a uniform block decomposition so the narrow-banded torus gives a very unbalanced computational load. The random permutation provides a nearly uniform nonzero distribution which allows CombBLAS to scale very well. Indeed we see this effect in all problems; CombBLAS performs well on problems that offer good load balancing and less well on ones that do not.

While load balance has a much weaker effect on QuadMat, we observe that QuadMat struggles when the left factor is much more sparse and random than the right factor, such as the permutation problems.

To help explain why sparse left factors are a performance bottleneck, we measure the observed utility of A organizer lookups. As described in Section 4, our inner product computation performs lookups into A's column organizer. The cost of each miss (empty column) is amortized by the number of nonzero elements discovered by hits. Each hit discovers at least one element.

We instrumented QuadMat to measure the total number of organizer lookups, the number of hits, and the number of nonzeros discovered through each hit. Dividing the latter by the total number of lookups gives us the lookup utility. Note that these measured numbers are specific to each particular block decomposition and will change with a different division threshold. See Table 6 in the Appendix.

We quickly observe a pattern. QuadMat has good computational performance on problems with high lookup utility and poor performance on problems with low lookup utility. Indeed the worst performing permutation problems have terrible lookup utility because nearly all lookups miss (due to the sparseness of the permutation matrix) and the hits discover the minimum one element. This is the hypersparse block effect.

CombBLAS is not affected by poor lookup utility because its hypersparse sequential kernel does not perform lookups. In ongoing work we try to get the best of both worlds. We mitigate the cost of the misses by switching to a DCSC-like organizer on hypersparse blocks. Our design also permits us to selectively perform the hypersparse algorithm on some block pairs then combine that result with results using our Gustavson-derived kernel.

5.3.2 Strong Scaling We are interested in what our code does on the same problem when it is given more resources. In Figure 10 we plot the speedup of QuadMat on two to 36 cores. On a single socket laptop with 4 cores and 8 threads we see excellent scaling even with two threads per core, but on our larger machine we see much less benefit from multiple threads per core.

We observe excellent scaling with 2 and 4 threads on all problems, and good scaling with 9 threads on most problems. Thread counts above 9 bring mixed performance; most problems continue scaling; some stay about the same. We hypothesize two reasons: insufficient parallelism and memory effects.

Our profile statistics in Table 6 include the total compute task work (total number of seconds) and the span (longest individual task). The ratio of those two times is our potential parallelism. We see that for our chosen division threshold, some problems (particularly AMG contraction) are indeed constrained by insufficient potential parallelism.

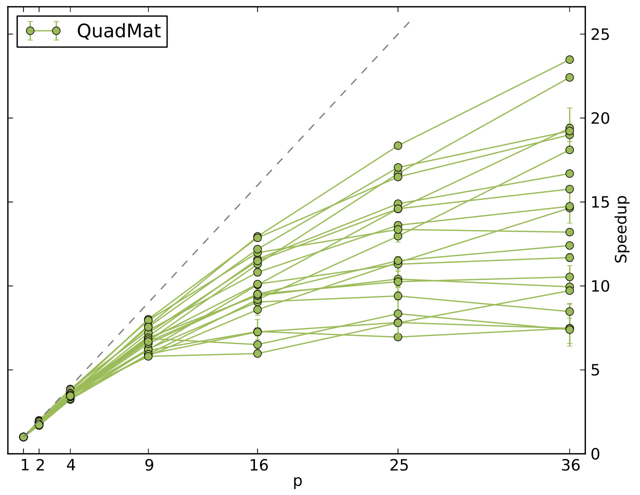


Figure 10: **(Preliminary)** Strong scaling of normal QuadMat. Each line shows the speedup for a particular problem when more threads are used.

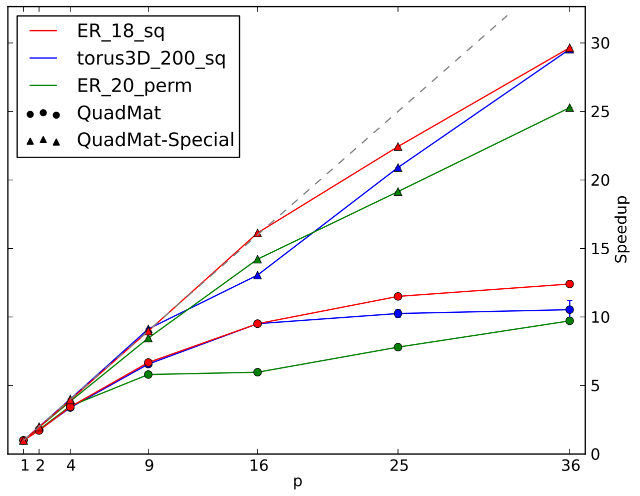


Figure 11: **(Preliminary)** Strong scaling comparison of normal QuadMat with a special version with increased arithmetic intensity to show impact of memory effects.

To explore memory effects, we performed a set of runs in which we artificially inflated the cost of arithmetic operations by looping them 5,000 times. This drastically reduces the effects of memory latency, bandwidth, and caches. Figure 11 shows the results for three problems, comparing the speedup of the normal code with the one with inflated arithmetic.

The vastly improved scaling of the code with inflated arithmetic shows that memory effects have a significant impact on strong scaling.

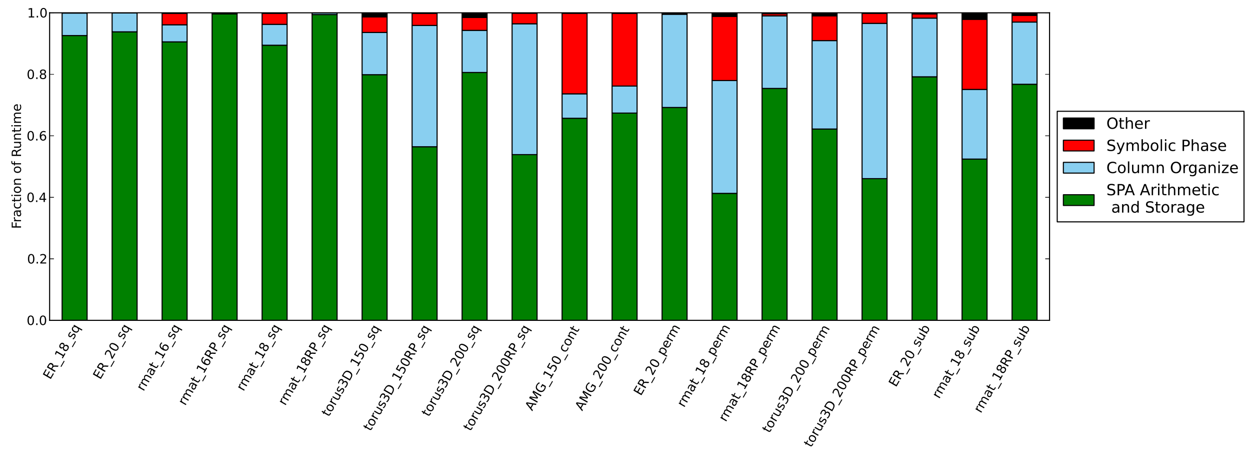


Figure 12: **(Preliminary)** Breakdown of time spent in each part of the algorithm on a single core. The green 'SPA Arithmetic & Storage' portion represents the inner block product computation. The blue 'Column Organize' proportion accounts for the time to generate and combine column organizers. The red 'Symbolic Phase' is dominated by shadow block creation. Miscellaneous code such as destructors and TBB overhead go into the black 'Other' portion.

We wish to bring the reader's attention to a hidden pitfall of shared memory algorithms that perform memory allocation in threaded kernels. Main memory is a shared resource, therefore its allocation must be done in a thread-safe manner. The naïve approach, locking, introduces a serialization hidden to the algorithm designer. One solution is an allocator based on thread-private heaps. TBB provides such an allocator [10].

5.3.3 Profiling We explore the efficiency of our algorithm and implementation through profiling. We compiled a special profiled binary which records the time spent in each phase of the algorithm. We are particularly interested in the amount of time taken by overhead in our design: the symbolic phase (dominated by shadow block creation) and column organization. We profile every problem in our problem set on one core in Figure 12.

The profile data shows that the symbolic phase, dominated by shadow block creation, is not a significant portion of the runtime. The time spent in the symbolic phase is less than 5% of runtime in all but four problems; the maximum is 25%.

Recall that this preliminary implementation includes only a naïve implementation of organizer caching. The need for efficient organization and organizer reuse is suggested by the profile data; column organization accounts for between $15 - 45\%$ of runtime for all but ER and RMAT square problems.

6 Discussion and Future Work

Our results show that, despite room for improvement, our algorithm has excellent performance and scaling. It offers significant speedup on some problems, and we have strong leads on how to improve the cases where it does not.

Interestingly, the problems that QuadMat excels on are also the ones that are sometimes considered the most difficult in the graph community: ones with a small number of high-degree vertices.

Our continuing work includes two main improvements that should significantly reduce or eliminate QuadMat's weaknesses: organizer caching and a hierarchical A-side organizer. These improvements should provide more latitude in automatically choosing a good division threshold.

Our algorithm has potential to be extended in several ways.

We envision a triple product primitive that does not materialize the entire intermediate product at any one time. This can be accomplished by merging the two SpGEMMs' symbolic phases. When done carefully with added destructor tasks, the portions of the intermediate product needed for a portion of the second SpGEMM can be materialized, used, and destroyed.

We also believe that the quadtree intermediate structure and triples leaf storage enables computing $A^T \times B$ with similar complexity to $A \times B$.

Additionally, we plan to take advantage of the block decomposition to use serialization coupled with compression algorithms for savings in both memory and memory bandwidth.

We may be able to save extra post-processing work by merging the subdivide or coalesce step with the compute phase. This is a great application for auto-tuning, as the appropriate choice needs to be made at runtime and according to the actual workload.

We also emphasize that our leaf blocks provide a triples *interface*, but do not mandate triples storage as an implementation. This enables features such as dense blocks or generator blocks that emit triples but do not store them.

7 Conclusion

In conclusion, we summarize the key contributions of the design of our quadtree sparse matrix multiplication algorithm:

- A method for elimination of explicit SpAdd operations that offers a significant reduction in work for block-based SpGEMM.
- A split between symbolic and computational phases with temporary on-the-fly data reorganization for simpler operations.
- An algorithm description that divides work into small tasks that can be scheduled on any number of threads by third-party frameworks.
- A quadtree of triples blocks datastructure that has significant flexibility with manageable overhead.
- A preliminary implementation that demonstrates these benefits.

8 Acknowledgments

We wish to acknowledge Sam Williams and Aydın Buluç for their help and inspiration of parts of the experiments.

References

[1] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proc. 21st Symp. on Parallelism in Algorithms and Arch.*, 2009.

[2] A. Buluç and J.R. Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *Intl. J. High Perf. Computing Appl.*, 25(4):496–509, 2011.

[3] T. A Davis. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, Sept 2006.

[4] P. Erdős and A Rényi. On the evolution of random graphs. In *Publication of the Mathematical Institute of the Hungarian Academy of Sciences*, pages 17–61, 1960.

[5] Jeremy D. Frens and David S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. *SIGPLAN Not.*, 32(7):206–216, June 1997.

[6] Jeremy D. Frens and David S. Wise. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 144–154, New York, NY, USA, 2003. ACM.

[7] J. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992.

[8] F. G. Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Trans. Math. Softw.*, 4(3):250–269, 1978.

[9] J. Kepner and J. R. Gilbert, editors. *Graph Algorithms in the Language of Linear Algebra.* SIAM, 2011.

[10] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in Intel Threading Building Blocks. *Intel Technology Journal*, 11(4):309 – 322, 2007.

[11] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using Kronecker multiplication. In *Proc. 9th Principles and Practice of Knowledge Disc. in Databases*, pages 133–145, 2005.

[12] Guy M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Canada, 1966.

[13] C. Pheatt. Intel threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[14] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, 1984.

[15] Y. Shapira. *Matrix-based Multigrid: Theory and Applications.* Springer, 2003.

[16] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. *SIGPLAN Not.*, 48(8):135–146, February 2013.

[17] R. A. Van De Geijn and J. Watts. Summa: Scalable universal matrix multiplication algorithm. *Concurrency: Practice and Experience*, 9(4):255–274, 1997.

[18] David S. Wise. Representing matrices as quadtrees for parallel processors: Extended abstract. *SIGSAM Bull.*, 18(3):24–25, August 1984.

[19] David S. Wise and John Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9(3):282 – 296, 1990.

[20] David S. Wise, Jeremy D. Frens, Yuhong Gu, and Gregory A. Alexander. Language support for Morton-order matrices. *SIGPLAN Not.*, 36(7):24–33, June 2001.

9 Appendix

...

Table 2: The Problems - Matrix Squares. Colors in the visual representation of nonzero distribution indicate density. Green and red hues represent more nonzeros. All matrices here and in Table 3 share the same color scale.

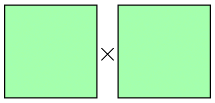
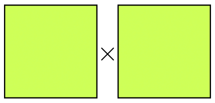
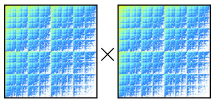
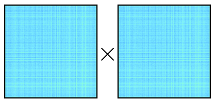
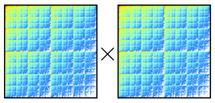
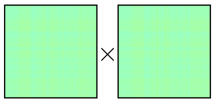
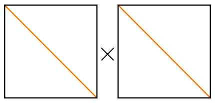
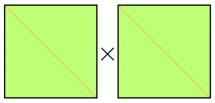
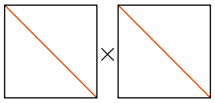
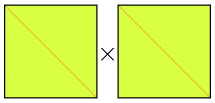
Name	Factors		Product	Non-Zero Arithmetic Ops.
ER_18_sq		$262K \times 262K$, $nnz = 8.39M$ $262K \times 262K$, $nnz = 8.39M$	$262K \times 262K$ $nnz = 268M$	$269M$
ER_20_sq		$1.05M \times 1.05M$, $nnz = 33.6M$ $1.05M \times 1.05M$, $nnz = 33.6M$	$1.05M \times 1.05M$ $nnz = 1.07G$	$1.07G$
rmat_16_sq		$65.5K \times 65.5K$, $nnz = 1.83M$ $65.5K \times 65.5K$, $nnz = 1.83M$	$65.5K \times 65.5K$ $nnz = 365M$	$2.15G$
rmat_16RP_sq		$65.5K \times 65.5K$, $nnz = 1.83M$ $65.5K \times 65.5K$, $nnz = 1.83M$	$65.5K \times 65.5K$ $nnz = 365M$	$2.15G$
rmat_18_sq		$262K \times 262K$, $nnz = 7.65M$ $262K \times 262K$, $nnz = 7.65M$	$262K \times 262K$ $nnz = 3.04G$	$16.1G$
rmat_18RP_sq		$262K \times 262K$, $nnz = 7.65M$ $262K \times 262K$, $nnz = 7.65M$	$262K \times 262K$ $nnz = 3.04G$	$16.1G$
torus3D_150_sq		$3.38M \times 3.38M$, $nnz = 23.6M$ $3.38M \times 3.38M$, $nnz = 23.6M$	$3.38M \times 3.38M$ $nnz = 84.4M$	$246M$
torus3D_150RP_sq		$3.38M \times 3.38M$, $nnz = 23.6M$ $3.38M \times 3.38M$, $nnz = 23.6M$	$3.38M \times 3.38M$ $nnz = 84.4M$	$246M$
torus3D_200_sq		$8.00M \times 8.00M$, $nnz = 56.0M$ $8.00M \times 8.00M$, $nnz = 56.0M$	$8.00M \times 8.00M$ $nnz = 200M$	$584M$
torus3D_200RP_sq		$8.00M \times 8.00M$, $nnz = 56.0M$ $8.00M \times 8.00M$, $nnz = 56.0M$	$8.00M \times 8.00M$ $nnz = 200M$	$584M$

Table 3: The Problems - Algebraic Multigrid Contractions, Permutations, and Submatrix Extractions. Colors in the visual representation of nonzero distribution indicate density. Green and red hues represent more nonzeros. All matrices here and in Table 2 share the same color scale.

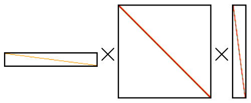
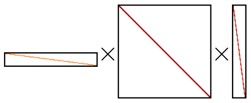
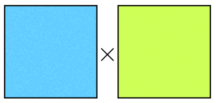
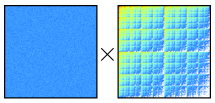
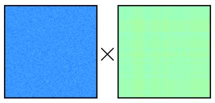
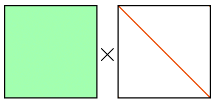
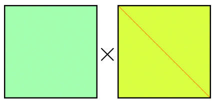
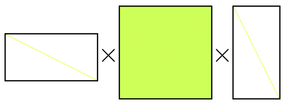
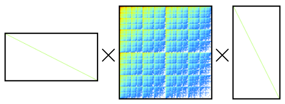
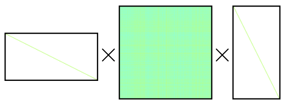
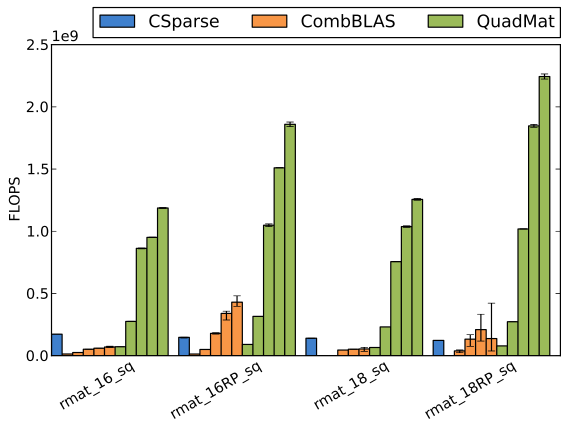
Name	Factors	Product	Non-Zero Arithmetic Ops.
AMG_150_cont	 $422K \times 3.38M$, $nnz = 3.38M$ $3.38M \times 3.38M$, $nnz = 90.7M$ $3.38M \times 422K$, $nnz = 26.8M$	$422K \times 422K$ $nnz = 11.4M$	$571M$
AMG_200_cont	 $1.00M \times 8.00M$, $nnz = 8.00M$ $8.00M \times 8.00M$, $nnz = 215M$ $8.00M \times 1.00M$, $nnz = 63.7M$	$1.00M \times 1.00M$ $nnz = 27.1M$	$1.36G$
ER_20_perm	 $1.05M \times 1.05M$, $nnz = 1.05M$ $1.05M \times 1.05M$, $nnz = 33.6M$	$1.05M \times 1.05M$ $nnz = 33.6M$	$33.6M$
rmat_18_perm	 $262K \times 262K$, $nnz = 262K$ $262K \times 262K$, $nnz = 7.65M$	$262K \times 262K$ $nnz = 7.65M$	$7.65M$
rmat_18RP_perm	 $262K \times 262K$, $nnz = 262K$ $262K \times 262K$, $nnz = 7.65M$	$262K \times 262K$ $nnz = 7.65M$	$7.65M$
torus3D_200_perm	 $8.00M \times 8.00M$, $nnz = 8.00M$ $8.00M \times 8.00M$, $nnz = 56.0M$	$8.00M \times 8.00M$ $nnz = 56.0M$	$56.0M$
torus3D_200RP_perm	 $8.00M \times 8.00M$, $nnz = 8.00M$ $8.00M \times 8.00M$, $nnz = 56.0M$	$8.00M \times 8.00M$ $nnz = 56.0M$	$56.0M$
ER_20_sub	 $524K \times 1.05M$, $nnz = 524K$ $1.05M \times 1.05M$, $nnz = 33.6M$ $1.05M \times 524K$, $nnz = 524K$	$524K \times 524K$ $nnz = 8.39M$	$25.2M$
rmat_18_sub	 $131K \times 262K$, $nnz = 131K$ $262K \times 262K$, $nnz = 7.65M$ $262K \times 131K$, $nnz = 131K$	$131K \times 131K$ $nnz = 4.24M$	$9.98M$
rmat_18RP_sub	 $131K \times 262K$, $nnz = 131K$ $262K \times 262K$, $nnz = 7.65M$ $262K \times 131K$, $nnz = 131K$	$131K \times 131K$ $nnz = 1.88M$	$5.67M$

Table 4: **Preliminary** Matrix Square elapsed time in seconds, mean of 5 runs. The machine has 40 cores capable of 80 concurrent threads.

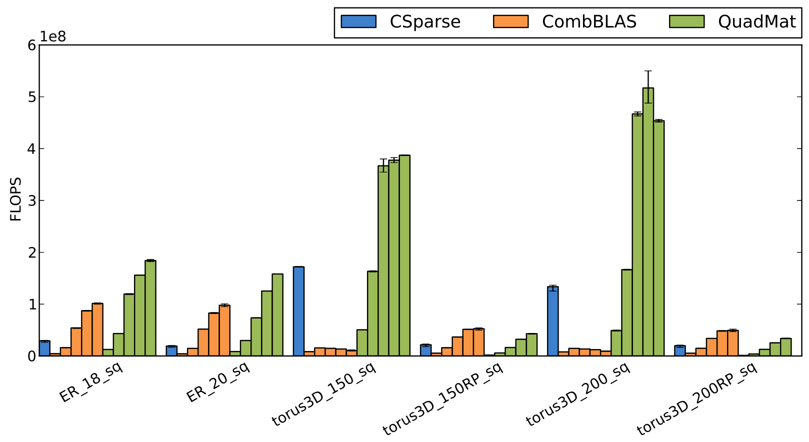
		ER_18_sq	ER_20_sq	rmat_16_sq	rmat_16RP_sq	rmat_18_sq	rmat_18RP_sq	torus3D_150_sq	torus3D_150RP_sq	torus3D_200_sq	torus3D_200RP_sq
CSparse	1p	9.20	56.2	12.4	14.6	115.	131.	1.43	11.4	4.37	29.4
CombBLAS	1p	59.7	255.	158.	161.			29.6	45.5	74.2	109.
	4p	16.7	73.4	84.1	42.9		418.	15.8	15.5	39.8	39.4
	9p	8.39	35.0	65.4	20.0	577.	161.	10.6	9.16	27.2	23.8
	16p	4.97	20.7	41.4	12.0	355.	121.	16.7	6.71	43.4	17.2
	25p	3.82	15.9	40.2	8.23	342.	67.9	6.73	5.55	18.2	14.0
	36p	3.08	13.0	35.7	6.32	309.	76.8	18.3	4.80	47.9	12.0
	64p	2.65	11.0	30.8	4.99	297.	117.	23.5	4.68	62.5	11.8
QuadMat	1p	21.4	126.	29.8	23.7	244.	204.	4.88	138.	11.9	516.
	2p	12.5	73.1	15.3	13.3	138.	117.	2.89	80.6	6.82	282.
	4p	6.21	36.0	7.80	6.80	69.5	58.9	1.51	42.2	3.51	150.
	9p	3.20	21.5	3.75	3.40	34.0	26.8	.823	23.3	1.81	76.8
	16p	2.25	14.6	2.49	2.05	21.3	15.8	.672	15.1	1.25	45.6
	25p	1.86	11.0	2.23	1.59	16.7	11.1	.624	10.7	1.16	30.9
	36p	1.73	8.57	2.26	1.42	15.5	8.71	.652	7.64	1.13	23.0
	64p	1.46	6.79	1.81	1.15	12.8	7.17	.636	5.74	1.29	17.3
	80p	1.40	6.55	1.61	1.22	11.0	7.31	.653	5.12	1.08	16.5

Table 5: **Preliminary** Algebraic Multigrid Contraction, Permutation, and Submatrix Extraction elapsed time in seconds, mean of 5 runs. The machine has 40 cores capable of 80 concurrent threads.

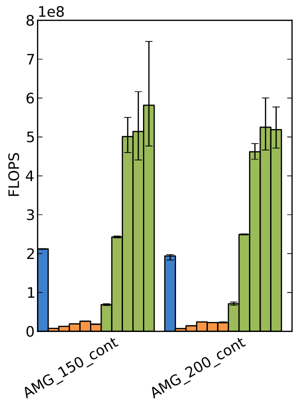
		AMG_150_cont	AMG_200_cont	ER_20_perm	rmat_18_perm	rmat_18RP_perm	torus3D_200_perm	torus3D_200RP_perm	ER_20_sub	rmat_18_sub	rmat_18RP_sub
CSparse	1p	2.69	6.99	5.40	.583	.681	3.99	17.3	2.69	.503	.452
	1p	75.1	185.	31.9	5.31	5.32	30.0	55.3	22.5	5.29	4.22
	4p	44.5	94.0	7.35	2.55	1.62	12.2	14.3	7.02	2.84	1.22
	9p	29.3	74.3	3.25	1.86	.873	8.83	6.98	3.93	2.61	.686
CombBLAS	16p	29.4	55.9	1.91	1.16	.535	7.24	4.35	2.81	1.82	.456
	25p	21.8	53.5	1.41	1.09	.435	6.51	3.29	2.21	1.85	.382
	36p	21.8	59.8	1.14	.965	.356	6.00	2.77	1.83	1.77	.322
	64p	30.8	59.3	.958	.758	.290	6.37	2.74	1.85	1.53	.323
	1p	8.30	19.1	24.5	22.0	3.04	99.0	347.	3.63	1.17	.622
	2p	4.45	10.7	13.7	11.7	1.53	52.8	203.	2.06	.616	.331
	4p	2.35	5.44	6.97	6.11	.788	26.8	105.	1.06	.327	.179
	9p	1.35	2.78	4.21	2.92	.413	12.4	55.9	.535	.170	.0996
QuadMat	16p	1.14	2.94	4.09	1.81	.281	7.70	34.5	.359	.125	.0688
	25p	1.19	2.29	3.13	1.29	.223	6.01	23.8	.322	.113	.0662
	36p	1.11	2.58	2.52	1.14	.206	5.21	17.9	.311	.118	.0734
	64p	.982	2.61	1.98	1.69	.244	5.39	12.9	.314	.125	.0805
	80p	1.25	2.64	1.90	2.11	.241	5.53	12.2	.342	.134	.0896



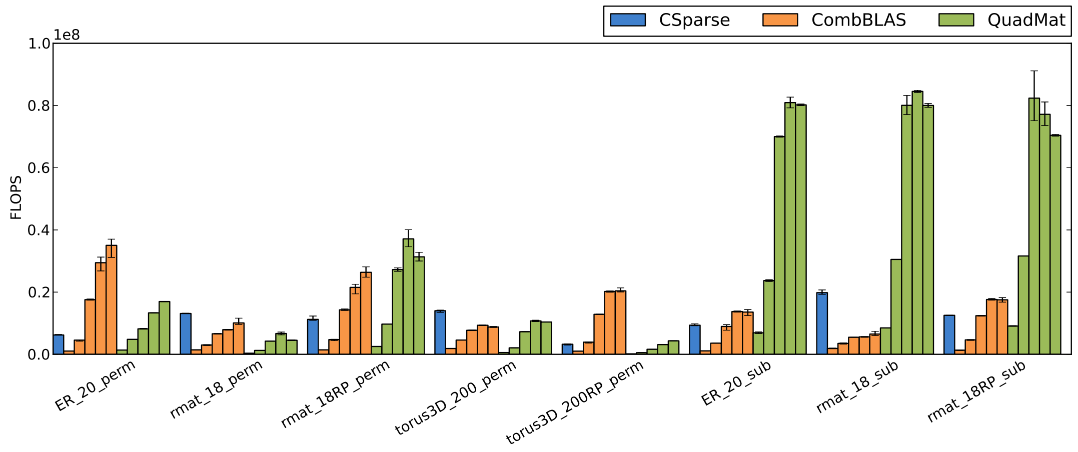
(a) RMAT matrix squares.



(b) Other matrix squares.



(c) Algebraic multigrid contractions.



(d) Permutations and submatrix extractions.

Figure 13: **(Preliminary)** FLOPS, or nonzero arithmetic operations per second, for each of the problems listed in Tables 2 and 3. Each set of five CombBLAS and QuadMat bars correspond to 1, 4, 16, 36 and 64 threads, while the CSparse bar is a single thread. The machine has 40 cores capable of 80 concurrent threads. The height of each bar indicates the mean of 5 runs; the error bars mark the fastest and slowest runs.

Table 6: **Preliminary** Problem statistics extracted using an instrumented build of QuadMat run with one thread. Detailed analysis of this data is in Sections 5.3.1 and 5.3.2. The division threshold is chosen to balance parallelism with minimization of total block count (reduce hypersparse blocks). The same *very preliminary* choice algorithm is used for all problems. Relatively poor QuadMat performance on some problems is explained by two factors. Poor scaling can be due to insufficient potential parallelism (threshold too large). Poor computational performance (torus squares, all permutations and submatrix extractions) is due to low A organizer lookup utility (threshold too small).

	ER_18_sq	ER_20_sq	rmat_16_sq	rmat_16RP_sq	rmat_18_sq	rmat_18RP_sq	torus3D_150_sq	torus3D_150RP_sq	torus3D_200_sq	torus3D_200RP_sq
Block Division Threshold	104850	419424	50000	50000	95639	95639	295312	295312	700000	700000
Total Comp. Tasks (Work)	$21.7s$	$122s$	$26.1s$	$25.2s$	$236s$	$202s$	$4.85s$	$133s$	$11.6s$	$471s$
Max Comp. Task (Span)	$0.0971s$	$0.634s$	$0.224s$	$0.437s$	$0.867s$	$0.948s$	$0.031s$	$0.315s$	$0.0621s$	$0.893s$
Potential Parallelism	223.7	191.8	116.3	57.8	271.8	213.2	156.3	423.2	186.0	527.1
A Organizer Lookups	1.34×10^8	5.37×10^8	8.06×10^7	1.46×10^7	6.33×10^8	1.22×10^8	7.53×10^7	6.14×10^8	1.78×10^8	1.74×10^9
Hits	86.5%	86.5%	69.6%	96.3%	63.4%	93.7%	74.5%	24%	69.7%	20.5%
A *nnz* / Hit	2.31	2.31	22.4	89.1	23.9	83.3	2.95	1.12	3.15	1.1
A *nnz* / Lookup	2	2	15.6	85.7	15.1	78.1	2.2	0.269	2.2	0.226

	AMG_150_cont	AMG_200_cont	ER_20_perm	rmat_18_perm	rmat_18RP_perm	torus3D_200_perm	torus3D_200RP_perm	ER_20_sub	rmat_18_sub	rmat_18RP_sub
Block Division Threshold	1133988	2690984	419424	95639	95639	700000	700000	419424	95639	95639
Total Comp. Tasks (Work)	$6.29s$	$15.3s$	$23s$	$17.6s$	$2.9s$	$86.6s$	$339s$	$3.34s$	$0.912s$	$0.615s$
Max Comp. Task (Span)	$0.215s$	$0.45s$	$0.114s$	$0.0226s$	$0.0284s$	$0.0177s$	$0.401s$	$0.0108s$	$0.0045s$	$0.00319s$
Potential Parallelism	29.2	33.9	201.1	780.7	102.1	4906.9	846.7	310.0	202.8	192.6
A Organizer Lookups	3.94×10^7	9.5×10^7	5.37×10^8	6.09×10^8	1.22×10^8	6.89×10^9	1.73×10^9	4.19×10^6	4.65×10^6	1.05×10^6
Hits	79.3%	77.3%	6.25%	1.26%	6.25%	0.813%	3.23%	86.5%	18%	28.3%
A *nnz* / Hit	6.85	6.92	1	1	1	1	1	2.31	5.08	6.32
A *nnz* / Lookup	5.43	5.35	0.0625	0.0126	0.0625	0.00813	0.0323	2	0.912	1.79