Efficient Sparse Matrix-Matrix Multiplication on Multicore Architectures*

Adam Lugowski† John R. Gilbert‡

Abstract

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

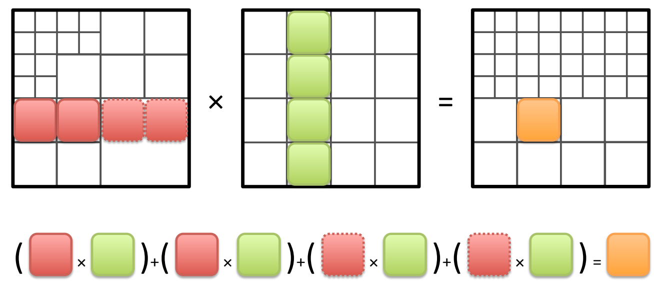

Figure 1: Computation of a result block using a list of pairwise block multiplications.

1 Introduction

Sparse matrix-matrix multiplication (or *SpGEMM*) is a key primitive in some graph algorithms (using various semirings) [5] and numeric problems such as algebraic multigrid [9]. Multicore shared memory systems can solve very large problems [10], 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 [11] and sparse [1] [2] matrices. Quadtree matrix decompositions have a long history [8].

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 only divide a block if its nonzero count is above a threshold. Elements are stored in column-sorted triples form inside leaf blocks. Quadtree subdivisions occur on powers of 2; hence, position in the quadtree implies the high-order bits of row and column indices. This saves memory in the triples. We do not assume a balanced quadtree.

3 Pair-List Matrix Multiplication Algorithm

The algorithm consists of two phases, a *symbolic phase* that generates an execution strategy, and a *computational phase* that carries out that strategy. Each phase is itself a set of parallel tasks. Our algorithm does not schedule these tasks to threads; rather we use a standard scheduling framework such as TBB, Cilk, or OpenMP.

3.1 Symbolic Phase We wish to divide computation of $C = A \times B$ into efficiently composed tasks with sufficient parallelism. The quadtree structure gives a natural decomposition into tasks, but the resulting tree of sparse matrix additions is inefficient. Instead we form a list of additions for every result block, and build the additions into the multiply step. We let C_{own} represent a leaf block in C, and *pairs* the list of pairs of leaf blocks from A and B whose block inner product is C_{own}.

$$C_{own} = \sum_{i=1}^{|pairs|} A_i \times B_i \tag{3.1}$$

The symbolic phase recursively determines all the C_{own} and corresponding *pairs*.

We begin with $C_{own} \leftarrow C$, and $pairs \leftarrow (A, B)$. If *pairs* only consists of leaf blocks, spawn a compute task with C_{own} and *pairs*. If *pairs* includes both divided blocks and leaf blocks, we temporarily divide the leaves until all blocks in *pairs* are equally divided. This temporary division lets each computational task operate on equal-sized blocks; it persists only until the end of the SpGEMM.

Once the blocks in *pairs* are divided, we divide C_{own} into four children with one quadrant each and recurse, rephrasing divided $C = A \times B$ using (3.1):

$$\begin{aligned} C_1 &= [(A_1, B_1), (A_2, B_3)] \\ C_2 &= [(A_1, B_2), (A_2, B_4)] \\ C_3 &= [(A_3, B_1), (A_4, B_3)] \\ C_4 &= [(A_3, B_2), (A_4, B_4)] \end{aligned} \tag{3.2}$$

For every pair in *pairs*, insert two pairs into each child's *pairs* according to the respective line in (3.2). Each child's *pairs* is twice as long as *pairs*, but totals only 4 sub-blocks to the parent's 8.

3.2 Computational Phase This phase consists of tasks that each compute one block inner product (3.1). Each task is lock-free because it only reads from the blocks in *pairs* and only writes to C_{own}. We extend

*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

Gustavson's sequential algorithm [4] in Algorithm 1.

Our addition to Gustavson is a mechanism that combines columns j from all blocks B_i in *pairs* to present a view of the entire column j from B. We then compute the inner product of column j and all blocks A_i 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}.

A and B are accessed differently, so we *organize* their column-sorted triples differently. For constant-time lookup of a particular column i in A, we use a hash map with a $i \rightarrow (\mathit{offset}_i, \mathit{length}_i)$ entry for each non-empty column i. A CSC-like structure is acceptable, but requires $O(m)$ space. We iterate over B's non-empty columns, so generate a list of $(j, \mathit{offset}_j, \mathit{length}_j)$. Both organizers take $O(nnz)$ time to generate. A structure that merges all B_i organizers enables iteration over logical columns that span all B_i.

Algorithm 1 Compute Task's Multi-Leaf Multiply

Require: C_{own} and *pairs*
Ensure: Complete C_{own}
  for all (A_b, B_b) in *pairs* **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 Texte

4 Experiments

We implemented our algorithm in TBB [7] and compared it with the fastest serial and parallel codes available, on a 40-core Intel Nehalem machine. We test by squaring Kronecker product (RMAT) matrices [6] and Erdős-Rényi matrices.

Observe from Table 1 that QuadMat only has a small speed penalty on one core compared to CSparse, but gains with two or more cores.

5 Conclusion

Our algorithm has excellent performance, and has the potential to be extended in several ways. Our next steps include a triple product primitive that does not materialize the entire intermediate product at any one time, and computing $A^T \times B$ with similar complexity to $A \times B$.

Table 1: SpGEMM results on E7-8870 @ 2.40GHz - 40 cores over 4 sockets, 256 GB RAM. Note: CombBLAS is an MPI code that requires a square number of processes.

	Squared Matrix	R_{16}	R_{18}	ER_{18}	ER_{20}
	Each Input nnz	1.8M	7.6M	8.39M	33.6M
	Output nnz	365M	2.96G	268M	1.07G
CSparse [3]	1p	14s	122s	9s	58s
CombBLAS [2]	1p	154s	1597s	64s	248s
	9p	19s	155s	8s	34s
	36p	8s	49s	3s	12s
QuadMat	1p	19s	150s	13s	111s
	2p	10s	87s	8s	66s
	9p	3s	21s	3s	18s
	36p	2s	11s	2s	9s

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

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

[6] 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.

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

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

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

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

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