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.

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] 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 $C_{own}$.

\begin{equation}
C_{own} = \sum_{i=1}^{|\text{pairs}|} A_i \times B_i
\end{equation}

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

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{align}
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{align}

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

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 \( i \rightarrow (\text{offset}_i, \text{length}_i) \) 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 \((j, \text{offset}_j, \text{length}_j)\). Both organizers take \( O(n_{nz}) \) 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 \( \text{combined}_B \_org \)
for all \((\text{column } j, \text{PairList}_j)\) in \( \text{combined}_B \_org \) do
SPA \( \leftarrow \{\} \)
for all \((A_b, B_b)\) in \( \text{PairList}_j \) 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

The entire intermediate product at any one time, and computing \( A^T \times B \) with similar complexity to \( A \times B \).

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.

<table>
<thead>
<tr>
<th></th>
<th>Squared Matrix</th>
<th>Each Input n nz</th>
<th>Output n nz</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( R_{16} )</td>
<td>( R_{18} )</td>
<td>( ER_{18} )</td>
</tr>
<tr>
<td>CSparse [3]</td>
<td>1p</td>
<td>14s</td>
<td>122s</td>
</tr>
<tr>
<td>CombBLAS [2]</td>
<td>9p</td>
<td>36p</td>
<td>10s</td>
</tr>
<tr>
<td>QuadMat</td>
<td>1p</td>
<td>9s</td>
<td>15s</td>
</tr>
<tr>
<td></td>
<td>2p</td>
<td>10s</td>
<td>87s</td>
</tr>
<tr>
<td></td>
<td>9p</td>
<td>3s</td>
<td>21s</td>
</tr>
<tr>
<td></td>
<td>36p</td>
<td>2s</td>
<td>11s</td>
</tr>
</tbody>
</table>

References