A Divide and Conquer Distributed Matrix-Matrix Multiplication

Majid Rasouli
School of Computing
University of Utah
Salt Lake City, Utah, USA
rasouli@cs.utah.edu

Robert M. Kirby
School of Computing
University of Utah
Salt Lake City, Utah, USA
kirby@sci.utah.edu

Hari Sundar
School of Computing
University of Utah
Salt Lake City, Utah, USA
hari@cs.utah.edu

***Abstract*—Matrix-matrix multiplication (GEMM) is a widely used linear algebra primitive common in scientific computing and data sciences. While several highly-tuned libraries and implementations exist, these typically target either sparse or dense matrices. The performance of these tuned implementations on unsupported types can be poor, and this is critical in cases where the structure of the computations is associated with varying degrees of sparsity. One such example is Algebraic Multigrid (AMG), a popular solver and preconditioner for large sparse linear systems. In this work, we present a new divide and conquer sparse GEMM, that is also highly performant and scalable when the matrix becomes very dense, as in the case of AMG matrix hierarchies. We combine this with an efficient communication pattern during distributed-memory GEMM to provide performance comparable to state-of-the-art sparse matrix libraries like PETSc. Additionally, the performance and scalability of our method surpass PETSc when the density of the matrix increases. We demonstrate the efficacy of our methods by comparing our GEMM with PETSc for different levels of sparsity.**

***Index Terms*—Matrix-Matrix Product, GEMM, Algebraic Multigrid, AMG, Linear Algebra, Iterative Solver, Preconditioner, Sparse, Dense**

I. Introduction

Matrix-matrix multiplication (GEMM) is a key linear algebra primitive commonly used by the computational and data science communities. Examples include operations part of the setup phase of algebraic multigrid methods (AMG) [1], an example that we will use heavily in this work to demonstrate the effectiveness of our methods. It is also common for large-scale graph analytics, where a linear algebra formulation is used, such as triangle counting [2], graph clustering [3], breadth first search [4], amongst others [5]. While there are several highly-tuned distributed-memory matrix libraries available, they usually target either sparse [6], [7] or dense [8] matrices. Unfortunately, the performance and scalability of these libraries is sub-optimal for matrices that are unsupported. For the case of AMG and for graph algorithms where the linear algebra formulation necessitates a GEMM, the resulting matrices can lose sparsity and become potential bottleneck for performance and scalability if the underlying GEMM implementation is unable to handle the loss of sparsity. The main contribution of this work is the development of a scalable distributed-memory GEMM algorithm that is able to be performant for varying levels of sparsity. We achieve this by developing a new divide-and-conquer GEMM that recursively divides the matrices vertically and horizontally. This progressively makes the matrices skinny such that the resulting product matrix block ($C = AB$) is dense. The splitting and merging of the matrices are done efficiently leveraging the sparse structure of the graphs, and aim to identify and expose dense blocks in the resulting product, for which we have implemented efficient data-structures. These product blocks are then combined in an efficient manner to produce the resulting product matrix C in a sparse format.

The divide and conquer approach improves memory access for deep-memory hierarchies, and adapts to varying levels of sparsity in a natural manner. Denser matrices will end the recursion sooner, but otherwise are identical to the behavior for sparser matrices. This enables our GEMM to perform in a predictable fashion independent of the density of the sparse matrix. We demonstrate the effectiveness of our algorithms and data-structures by comparing with PETSc [6] and demonstrate performance comparable to PETSc for sparse matrices. In contrast, while the performance and scalability of PETSc suffers when the matrices become denser, our GEMM demonstrates excellent scalability even for these cases. We use the example of building an AMG grid hierarchy to evaluate our methods. Specifically, an AMG grid hierarchy is built using a Galerkin approximation by the multiplication of three sparse matrices. This leads to increasing loss of sparsity at coarser levels. Fig. 1 from [9] shows the loss of sparsity in three levels of AMG.

The main **contributions** of this work are,

- A new divide and conquer algorithm for GEMM that is able to perform efficiently for a wide range of sparsity patterns,
- A new communication pattern to improve the parallel scalability of GEMM, and
- A thorough evaluation and scalability study to demonstrate the effectiveness of the proposed methods.

The rest of the paper is organized as follows. In the next section, we provide background into AMG to help the readers understand the target application. We chose AMG as the application because we wanted to consider realistic scenarios

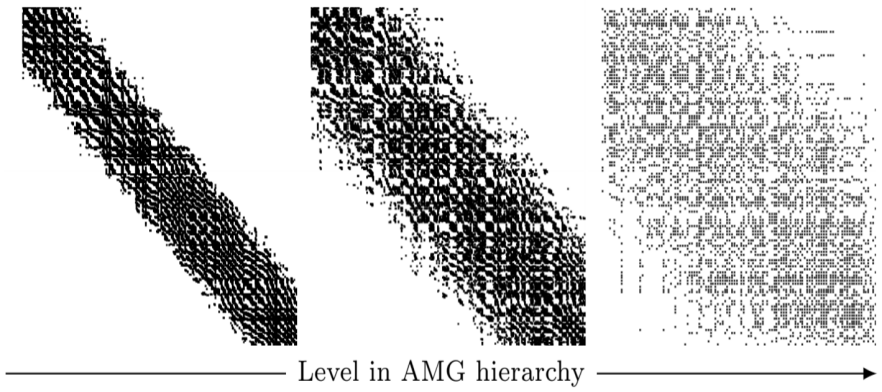

Fig. 1. This figure shows an example of three levels of an AMG hierarchy (levels 3, 4 and 5). The loss of sparsity is noticable from this figure.

where variable sparsity patterns are encountered. We provide a brief review of related work in §I-B. In §II we discuss the different strategies used to improve the performance and scalability of GEMM. In §III we show the strong and weak scaling of our methods and compare our method with PETSc. Finally, we conclude the paper in §IV.

A. Background - AMG

The solution of elliptic partial differential equation (PDE) operators forms the foundation of the mathematical models of several engineering applications. In solid mechanics, the stiffness matrix derived from linear elasticity represents an elliptic operator that, when discretized with the finite element method (FEM) yields a symmetric positive definite system. In fluid mechanics, the viscous terms and the pressure components of the incompressible Navier-Stokes equations, when discretized, often lead to symmetric positive definite systems. For the solution of such large-scale systems that need to be solved in parallel, iterative solvers with $\mathcal{O}(n)$ complexity and mesh-independent convergence are preferred. The most popular methods in this regard have been the multigrid methods– both geometric and algebraic. The geometric multigrid (GMG) methods when applied to matrices generated from regular (often evenly-spaced) discretizations of elliptic operators [10]–[13]. The mathematical predictability of the benefits of the GMG approach combined with the ability to exploit the regularity of the data structures (in terms of indexing, coarsening, etc.) has made GMG, especially in conjunction with preconditioned conjugate gradient (PCG) [14], [15], the solver of choice when solving engineering applications that require large-scale parallel solution approaches. The story is more mixed when one moves to unstructured discretizations and corresponding to algebraic multigrid (AMG), the communities alternative to GMG for such problems.

While AMG is very attractive due to its black-box nature [1], [16], [17], it does not scale as well as GMG [18]. This is primarily due to the loss of sparsity at coarser levels arising from the Galerkin approximation [19], leading to poor scalability, especially at the coarser levels. Additionally, while both geometric and algebraic variants require an initial **setup** phase, the cost of the setup is significantly higher for AMG, making it less attractive unless multiple solves are performed for the same operator. This is clearly unattractive for dynamic systems where the operator is changing rapidly, such as systems with h or p enrichment. In such cases, while the cost of an AMG solve might be lower than say that of CG, the overall cost i.e., setup + solve might be more than using asymptotically inferior solvers. In most cases, the scalability of setup phase is also poor and typically worse than that of the solve phase. This has limited the attractiveness of AMG for large systems. In this work, we develop an efficient sparse matrix multiplication (GEMM) algorithm to improve the performance and scalability of AMG. As will be explained in the following section, a sparse GEMM is the dominant part of the AMG setup. While several of our optimizations apply to sparse GEMM in general, and can be more broadly applied, our strategies for reducing inter-process communication make use of the special structure of GEMM encountered during AMG setup.

We start with a brief description of our AMG framework. AMG has been a popular method for solving the large-scale and often sparse linear system one obtains from discretization of elliptic partial differential equations. The linear system can be written as

$$Ax = b \tag{1}$$

in which, $A \in R^{n \times n}$, x and $b \in R^n$. AMG consists of a setup and a solve phase. The first step of the setup phase is to aggregate the nodes of the equivalent graph (G) of the matrix A. Every row of the matrix A is considered as a node in the graph G and there is an edge between nodes i and j if entry (i, j) is nonzero in A. After aggregation, some nodes will be chosen as *roots* and the rest of the nodes of the graph will be assigned to them. Multiple aggregation methods for AMG have been introduced, such as [20], [21], [22], [23]. For this paper, *maximal independent set* from [20], with some modifications, is used.

Given a linear system we have n nodes in the graph G and m nodes are chosen as the *roots*. We compute prolongation P and Restriction R operators using the *roots*. The prolongation

operator has two applications. It can interpolate a vector $v \in R^m$ to $v' \in R^n$, such that $m < n$. The other application is creating a coarse version of the operator A using the Galerkin approximation:

$$A_c = R \times A \times P \tag{2}$$

such that $A_c \in R^{m \times m}$. This operation is called *coarsening*. The restriction operator is used similarly. The triple GEMM in (2) is the dominant cost of the AMG setup, especially when done in parallel. Improving the efficiency and scalability of this step in the main contribution of this work.

Progressively coarser versions of the matrix are created during the setup phase corresponding to a hierarchy (i.e. multi-level or "multigrid") of data structures. An AMG hierarchy of $L+1$ levels consists of three categories of operators: coarse matrices (As), prolongation matrices (Ps) and restriction matrices (Rs).

The coarse matrices for each level are created similar to A_c:

$$As[l+1] = Rs[l] \times As[l] \times Ps[l],\ l = 0, 1, ..., L$$

such that $As[0]$ is the finest matrix A. Again note that this is very expensive even at the coarser levels as the matrices get denser at the coarser levels. Our GEMM maintains good performance and scalability across all levels of the multigrid hierarchy.

The next phase of AMG is the solve phase. To solve $Ax = b$, we start with an initial guess for x. The solution is computed in a recursive function *vcycle* (Algorithm 1). Regular smoothers are used in the relaxation part, such as Jacobi, Chebyshev, etc. Then, the residual r is computed. Next, r is taken to the coarser level by using the restriction operator (R). The function recurses until it reaches the coarsest level ($L+1$). At that level, the system will be solved directly. The solution of that system is actually the error, which will be interpolated by P. After that, the solution will be corrected by subtracting the interpolated error from it. Finally, the solution will be smoothed again.

Algorithm 1 vcycle($g, x,\ b,\ l$)

Input: $grid\ g$, b, x, and $level\ (l)$
Output: $solution\ (x)$

1: **if** $l = L + 1$ **then**
2: $\quad x \leftarrow direct\ solver(g[L+1],\ x,\ b)$
3: **else**
4: $\quad x \leftarrow Smoother(g, x,\ b,\ l)$
5: $\quad r \leftarrow As[l] \times x - b$
6: $\quad r_c \leftarrow Rs[l] \times r$
7: $\quad y_c \leftarrow vcycle(g, x,\ r_c,\ l+1)$
8: $\quad y \leftarrow Ps[l] \times y_c$
9: $\quad x \leftarrow x - y$
10: $\quad x \leftarrow Smoother(g, x,\ b,\ l)$
11: **end if**

Smoothed Aggregation AMG (SA-AMG) [16] is a modified version of AMG, in which the prolongation and restriction operators are smoothed to improve the convergence of AMG. For this paper, the improved version of SA-AMG in [19] is used.

B. Related Work

While significant research has been done on improving the efficiency and scalability of sparse matrix-vector products, sparse GEMM in comparison has received far less attention. Yuster and Zwick [24] provide a theoretically nearly optimal algorithm for multiplying sparse matrices, but rely on fast rectangular matrix multiplication. Consequently the approach is currently of only theoretical value. In [25], Buluc and Gilbert present algorithms for parallel sparse GEMM using a two-dimensional block data distributions with serial hypersparse kernels. Gremse *et al.* [26] present a promising algorithm using iterative row merging to improve the performance on GPUs. Similarly, Saule *et al.* [27] evaluate the performance of sparse matrix multiplication kernels on the Intel Xeon Phi. Most AMG implementations have relied on standard sparse GEMM implementations without any special considerations for the structure of the matrices generated within AMG. This work attempts to fill this gap.

II. Methods

In this section, we present our GEMM algorithm. We first discuss the choice of the stop condition for the recursive function and how matrices are being divided to smaller blocks. Then, We explain how the communication is being done in an overlapped distributed fashion to help the recursive function scale better.

A. Matrix-Matrix Multiplication

We design a divide and conquer approach to perform GEMM in a node-local fashion. The key idea is to perform simple tasks while recursing, having efficient memory access, and to perform the multiplication for small chunks where the resulting matrix can fit into an appropriate cache. For clarity of presentation, we assume that the data is available locally and discuss it as a serial implementation. Shared memory parallelism is added in a straightforward manner. The distributed part is explained in the next section. The matrices are stored in the *CSC* sparse format.

To perform the multiplication

$$C = A \times B \tag{3}$$

we keep splitting the matrices horizontally and vertically (Figure 2) based on row size and column size of A, until we can fit the result of the multiplication in a dense buffer.

The recursive function, RECURS_GEMM, includes three cases:

1) Case 1: Stop the recursion and perform the multiplication.
2) Case 2: A is horizontal. Split the blocks.
3) Case 3: A is vertical. Split the blocks.

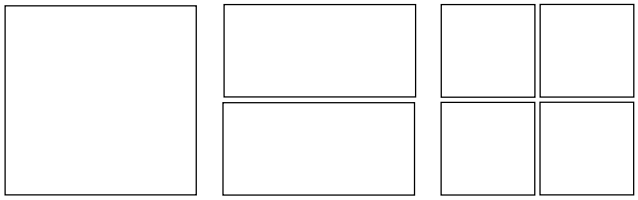

Fig. 2. A basic scheme that shows splitting the matrix first horizontally, then vertically.

1) Case 1: First we discuss when we decide to stop the recursive function. Our goal is to fit the multiplication result of blocks of A and B in a dense buffer. We show the blocks of A and B as A_{ij} and B_{lk}. We use two indices here because the matrices get divided both horizontally and vertically. The size of the dense buffer to store $A_{ij} \times B_{lk}$ is

$$row\ size\ of\ A_{ij} \times column\ size\ of\ B_{lk} \tag{4}$$

Therefore, Equation (4) can be used as the naive choice to decide when to stop the recursion, but Figure 3 shows why that is not a good choice. If we use Equation (4) for this example, the splitting process for the top two blocks of the matrix stops at the same step because they have the same size, but to have a more efficient method the top left block should be divided to more blocks than the top right block.

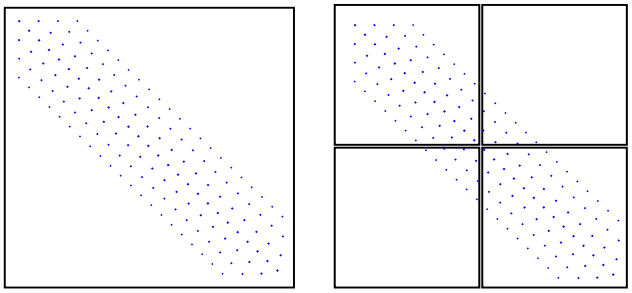

Fig. 3. Using row and column sizes to decide when to stop the recursion is not efficient, because the top left block is the same size as the top right one, but it should be divided to more sub-blocks.

Furthermore, by splitting sparse matrices recursively, we will have more and more zero rows and columns in the resulting blocks. So, using row size and column size of the blocks is not very helpful. Instead, we use nonzero rows and nonzero columns.

At the start of the recursive function, we compute the number of nonzero rows of A (A_nnz_row) and nonzero columns of B (B_nnz_col). A threshold for

$$\text{NNZ\_MAT\_SIZE} := A\_nnz\_row \times B\_nnz\_col \tag{5}$$

is set. Our algorithm has a profiling step where it empirically determines the appropriate threshold by running several test cases. On the machines we used, $20M$ was chosen as the threshold. We continue splitting the matrices until the threshold is reached, and then we preform the multiplication.

We have implemented two methods to perform the multiplication:

1) dense buffer
2) hashmap

When performing the multiplication, at least one of the matrices, typically the output, needs random access as it is accumulating the results. Given that the divide and conquer approach has reduced the size of the output matrix, the first approach is to keep a temporary buffer for dense matrix storage. Each nonzero of B is multiplied by its corresponding nonzero of A and the result will be added to the corresponding index in the dense matrix. As long as the dense matrix is small enough to fit within the L2 cache, we should get good performance. At the end of the multiplication, we traverse the dense matrix and extract the non-zeros as a sparse matrix. This approach works well as long as the resulting matrix is dense. We allocate a memory block of size of the threshold before starting the matrix product. When we reach the stop condition for each recursive call, we preform the following steps:

1) Initialize the first NNZ_MAT_SIZE entries to 0.
2) Perform the multiplication and add the result entries to the buffer matrix.
3) Extract nonzeros from the dense matrix and add them to C.

When the ratio of nonzeros to NNZ_MAT_SIZE is low, it becomes inefficient to traverse the whole dense matrix and check for nonzeros in the final stage. To solve this issue, we use an efficient hashmap to achieve similar results without the $\mathcal{O}(n^2)$ overhead of extracting the non-zeros from the dense matrix. The entry's index is the key and its value is the hashmap's value. When we want to add the multiplication of nonzeros of A and B to the hashmap, we check if the index exists in the map. If it exists the value is being added to the existing one's. Otherwise a new entry will be added to the hashmap. Clearly, there is an overhead to this approach that needs to be balanced against the overheads associated with the dense representation.

To measure the effectiveness of our method in a practical situation, we have implemented the RECURS_GEMM function in our Algebraic Multigrid solver, called *Saena*, and performed GEMM twice to compute the coarse matrix of the 3D Poisson Problem.

In Figure 4, we compare the two methods for computing coarse matrix $Ac = R \times A \times P$, in which A is the 3D Poisson problem of size $216k$. For $0 \leq$ NNZ_MAT_SIZE $\leq 10M$, in $1M$ steps, we compare the two methods in order to develop an efficient hybrid algorithm. For instance, the first point indicates that the dense structure is better than the hashmap approach in 1245 cases for the multiplications such that $0 \leq$ NNZ_MAT_SIZE $< 1M$. For the lower range the dense representation is better and for the higher range the hashmap is significantly faster. Figure 5 shows the same experiment for matrix ID 1882 from SuiteSparse (Florida) Matrix Collection, which is a sparse matrix of size $1M$ and $5M$ nonzero.

A combination of these two methods would give us the best performance across different matrix structures and densities. The dense method is being used for the lower range and the hashmap for the higher range. We have done a series of experiments to determine the threshold when to switch between the two methods. Figures 4 and 5 suggest to use the dense structure method when $0 \leq$ NNZ_MAT_SIZE $< 4M$ and use hashmap for the rest. We noticed that when hashmaps are better, the difference time between the two methods on average is higher. In other words, on average, n times performing hashmap is faster than n times using the dense structure. So empirically, we found $1M$ to be a good estimate for switching between the two methods.

Figure 6 compares the hybrid method with the basic two methods. We have compared the three approaches on different sizes of 3D Poisson problem, ranging from $8k$ to almost half a million. For instance, for the case where the matrix is of size $512k$, performing the triple matrix product takes $291s$ if only hashmap is used for Case 1, takes $72s$ if only the dense structure is used and finally takes almost $17s$ when the hybrid approach is utilized.

2) Case 2: When A is horizontal, i.e. its row size is less than or equal to its column size, we halve A by column based on its column size (Figure 7). Since row size of B equals column size of A, we halve B by row, so it will be a split similar to A, but horizontally. Then, the RECURS_GEMM will be called twice, once on $A1$ and $B1$, and again on $A2$ and $B2$ (Algorithm 2). The results of the two multiplications need to be added together at the end. It means, there will be entries for the result matrix with the same index. We call these entries *duplicates*. Since there will be numerous nested recursive calls, we avoid doing adding duplicates at this stage. After the starting recursive function is finished, we sort C and then add the duplicates only once at the end.

Algorithm 2 Case 2: $C =$ RECURS_GEMM2(A, B)

Input: A, B
Output: C
1: $(A1, A2) =$ SPLIT_BY_COL(A)
2: $(B1, B2) =$ SPLIT_BY_ROW(B)
3: $C \leftarrow$ RECURS_GEMM$(A1, B1)$
4: $C \leftarrow$ RECURS_GEMM$(A2, B2)$

3) Case 3: When A is vertical, i.e. its row size is greater than its column size, we halve A by row and B by column (Figure 8). This time the RECURS_GEMM will be called four times (Algorithm 3). Although we have 4 recursive calls in this case, but there is no duplicates at the end, which makes this case more efficient than Case 2 for the total time, because we have a smaller set of entries to sort and add the duplicates.

Algorithm 3 Case 3: $C =$ RECURS_GEMM3(A, B)

Input: A, B
Output: C
1: $(A1, A2) =$ SPLIT_BY_ROW(A)
2: $(B1, B2) =$ SPLIT_BY_COL(B)
3: $C \leftarrow$ RECURS_GEMM$(A1, B1)$
4: $C \leftarrow$ RECURS_GEMM$(A2, B1)$
5: $C \leftarrow$ RECURS_GEMM$(A1, B2)$
6: $C \leftarrow$ RECURS_GEMM$(A2, B2)$

We have also implemented splitting based on the number of nonzeros. In *Case 2*, we split A in a way to have half of nonzeros in $A1$, and the other half in $A2$. The same split is used for B. In *Case 3*, we do the same, but separately for both A and B. We compare these two splitting methods in the last section.

4) All together: When all three cases work together, we have Case 2 and Case 3, that aim to divide the matrices into skinny matrices such that the resulting matrix is small. Then by using a hybrid multiplication algorithm, we get these results. These results are then accumulated and merged together. From a memory access perspective, the accumulation and merging required for Case 2 and 3 is structured access to the matrix, with the only random access happening during Case 1. This makes the overall algorithm very efficient.

B. Communication

In the previous section we explained how to perform RECURS_GEMM if the data is available locally. In this section, we explain how the communication is done to perform

$$C = A \times B \tag{6}$$

in a distributed fashion for general (non necessarily symmetric) matrices A and B. Also, we propose a method to improve the performance and scalability of RECURS_GEMM if B is symmetric.

Matrices are partitioned across multiple processors by row blocks (Figure 9). Since matrices A and B may have different number of rows, they may not be partitioned the same way. We avoid communicating A in our method and only communicate B, to avoid any communication at the end of the multiplication. Algorithm 4 shows how the communication is done. It is an overlapped implementation, so while the processors are communicating the data, the multiplication is being executed on the available data from the previous processor (so executing RECURS_GEMM between the *Isend-Irecv* part and *wait*). This way we save a portion of the time that the communication takes and use it to do the multiplication. The other advantage of our communication algorithm is having each processor to communicate only with their neighbors.

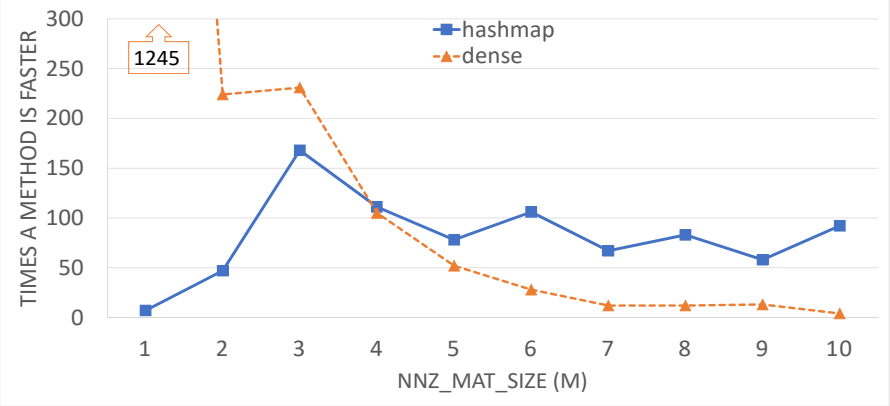

Fig. 4. Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is the 3D Poisson problem of size $216k$. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

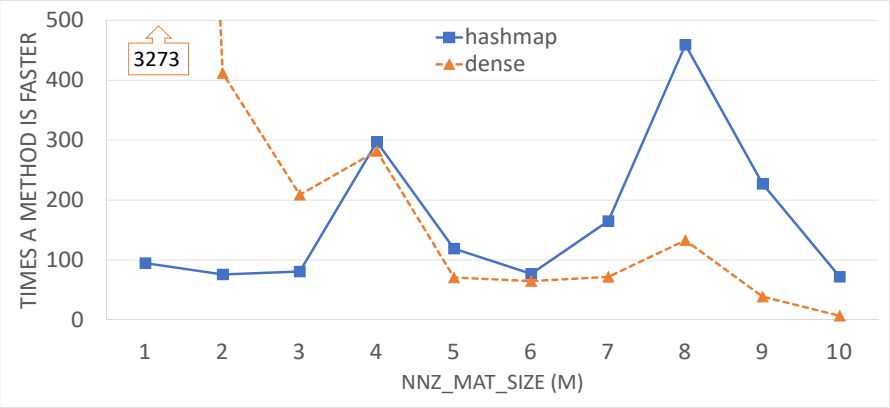

Fig. 5. Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is matrix ID 1882 from SuiteSparse (Florida) Matrix Collection. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

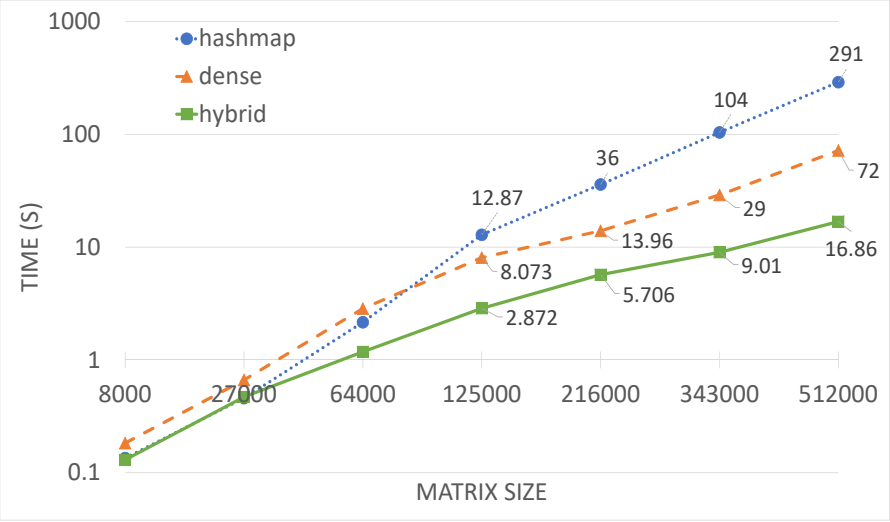

Fig. 6. Comparison between the three methods to do Case 1: only using hashmap, only using the dense structure, and the hybrid method. They are used in Case 1 part of RECURS_GEMM to compute the first coarse matrix (the triple multiplication) of 7 matrices (3D Poisson) of different sizes.

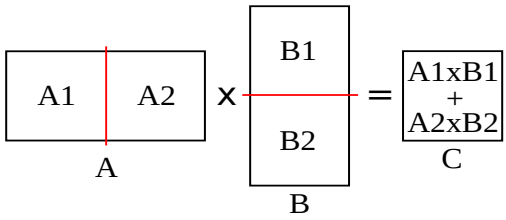

Fig. 7. Case 2: When A is horizontal, split A by column and B by row. Call the recursive function twice.

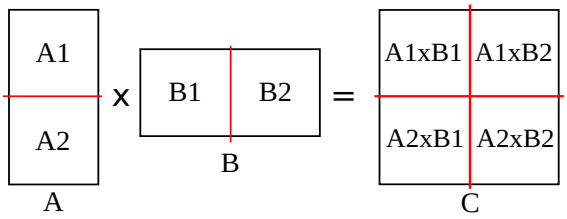

Fig. 8. Case 3: When A is vertical, split A by row and B by column. Call the recursive function four times.

Algorithm 4 $C_i = A_i \times B$

Input: A_i, B
Output: C_i (result of $A_i \times B$)

1: $B\_send \leftarrow B_i$
2: **for** $k = myrank : myrank + nprocs$ **do**
3: $Isend(B\_send)$ to left neighbor
4: $B\_recv \leftarrow$ Irecv(remote B) from right neighbor
5: $C_i \leftarrow$ RECURS_GEMM(A_i, B_send)
6: wait for Isend and Irecv to finish
7: $swap\_pointers(B\_send, B\_recv)$
8: **end for**
9: locally sort C_i and add duplicates

Now, we explain how to improve our algorithm to compute $A \times B$, when B is symmetric. Since the number of columns of A equals the number of rows of B (to be able to multiply them), we assume the same division of rows of B on A's columns (blue lines), only to show corresponding parts of A and B that should be multiplied. To compute entry C_{ij}, we need to multiply row i of A with column j of B and add them together. For that, the blocks of A and B with the same color in Figure 10 should be multiplied, and only after multiplying all those blocks we have all the duplicates to add together and have the final value for entry C_{ij} (Line 9 in Algorithm 4).

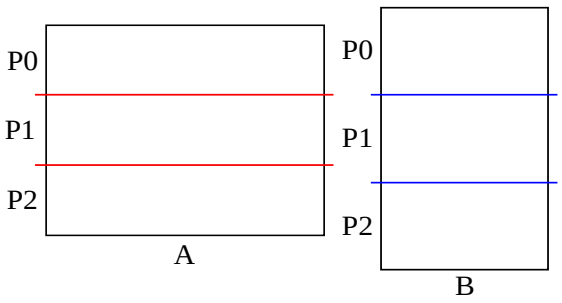

Fig. 9. Partitioning of the matrices across the processors in row blocks.

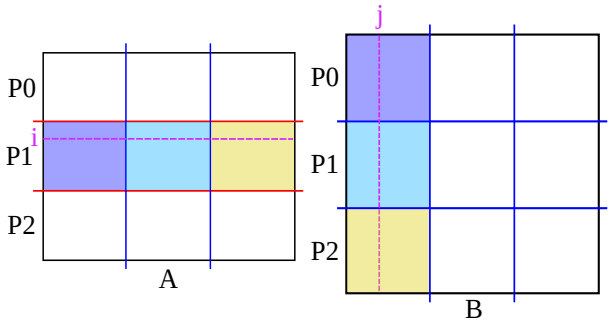

Fig. 10. Column j of B is stored on different processors, so to compute entry C_{ij} we need to multiply the parts of A and B with the same color.

When B is symmetric, instead of working with B_i's, we consider their local transpose BT_i in Figure 11.

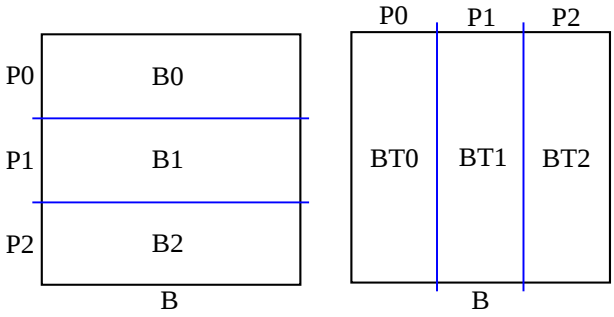

Fig. 11. When B is symmetric, we use local transpose of its row blocks.

If we partition B in column blocks, then we don't even need B to be symmetric, but since we want to consider the same partitioning method for all the matrices in our algorithm and we don't want to change the partition for one matrix to do GEMM only, we assume B is symmetric.

Algorithm 5 $C_i = A_i \times B$, when B is symmetric.

Input: A_i, B
Output: C_i (result of $A_i \times B$)

1: $B\_send \leftarrow$ local transpose of B_i
2: **for** $k = myrank : myrank + nprocs$ **do**
3: $Isend(B\_send)$ *to left neighbor*
4: $B\_recv \leftarrow$ Irecv(remote B) from right neighbor
5: $C_{ik} \leftarrow$ RECURS_GEMM(A_i, B_send)
6: *wait for Isend and Irecv to finish*
7: $swap\_pointers(B\_send, B\_recv)$
8: locally sort C_{ik} and add duplicates
9: **end for**
10: put all C_{ik}'s into C_i

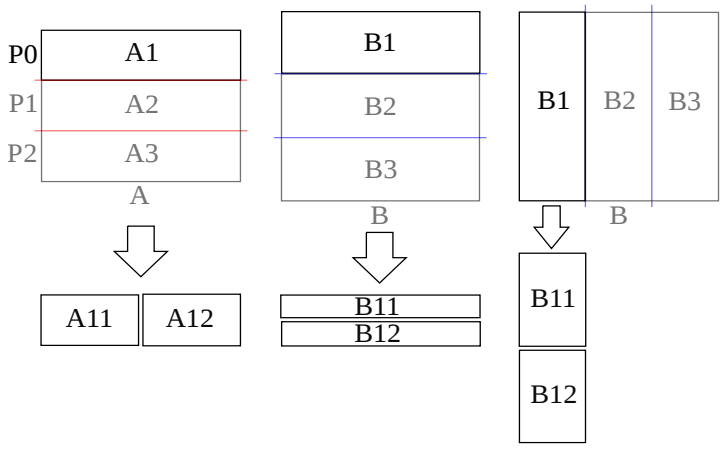

Fig. 12. Comparing splitting B based on Algorithm 4 (middle) and Algorithm 5 (right)

There are at least two advantages for using Algorithm 5 instead of Algorithm 4. First, we can sort subsets of entries of C_i separately and add duplicates (compare line 9 of Algorithm 4 with line 8 of Algorithm 5, which is cheaper than doing that on the final matrix. The other advantage is having better divided blocks of B for RECURS_GEMM. The way the splitting procedure is being executed, it is more efficient to have matrix B horizontal if A is vertical and vice versa. Figure 12 compares both ways of dividing matrix B on processor $P0$. Algorithm 4 makes B skinnier, but Algorithm 5 helps to reach the threshold faster.

III. Numerical Results

Our code is written in `C++` using `MPI` and `OpenMP` and is freely available on github (url withheld for review) under an MIT license. All experiments were conducted on the RMACC Summit Supercomputer at the University of Colorado, Boulder (via XSEDE). Each node has 24 cores and it uses Intel Xeon E5-2680 with $4.84GB$ of memory per core. All experiments were run in the hybrid `MPI+OpenMP` mode.

For these experiments we have multiplied a banded matrix with itself, assuming that the matrix is being multiplied with a separate matrix, so not using any information from the left-hand side matrix for the right-hand side one.

Figure 13 shows the weak scaling for two banded matrices: one with $24k$ on each node and the other one with $100k$ on each node. Our solver scales better when there is a smaller block of the matrix on each core, which happens when we use more processors or when the matrix is smaller. So, for the same matrix if we use more processors, we will have better performance.

Figure 14 is the strong scaling for four banded matrices of the same size ($192k$), but with different bandwidth. The legend shows the density ($\frac{nonzero}{size^2}$) of each matrix. Our solver scales consistently for different density values.

Figure 15 compares the scaling time for when we split the matrices based on matrix size and when we split based on number of nonzeros. The one that uses the matrix size is more scalable than the other one.

Figure 16 compares the strong scaling between our solver with PETSc. For the matrices with lower density (more sparse) PETSc performs better when using higher number of processes, but for denser matrices our solver is faster. In multigrid hierarchy, the coarse matrices get denser as we go to lower levels, so it becomes expensive to perform GEMMat those levels, so switching to our algorithm for lower levels of multigrid would improve the performance and scalibility significantly.

IV. Conclusion

We have presented a divide and conquer approach to improve the performance and scalibility of GEMM. Our GEMM has a very good performance and is scalable even when the matrix becomes very dense, as in the case of AMG matrix hierarchies. We have also designed an overlapped communication method to improve the efficiency of our recursive algorithm. We demonstrated performance gains from using our methods and compared our multiplication with the in-built functions within PETSc. In our future work, we want to further improve our performance and scalability and also focus on using sparsification algorithms to ensure the sparsity of coarser levels in the AMG application.

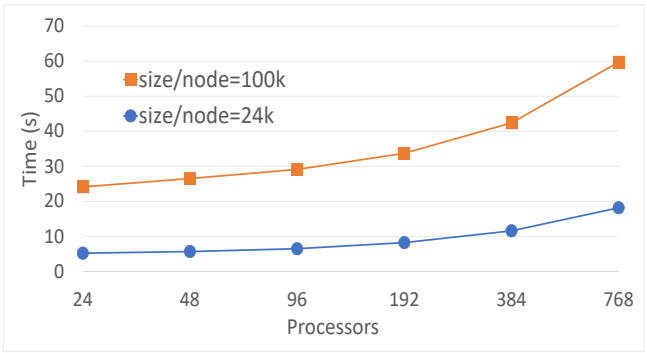

Fig. 13. Weak scaling for two banded matrices: blue line shows the one with $24k$ on each node ($1k$ on each core) and red line shows a larger one with $100k$ on each node (4166 on each core).

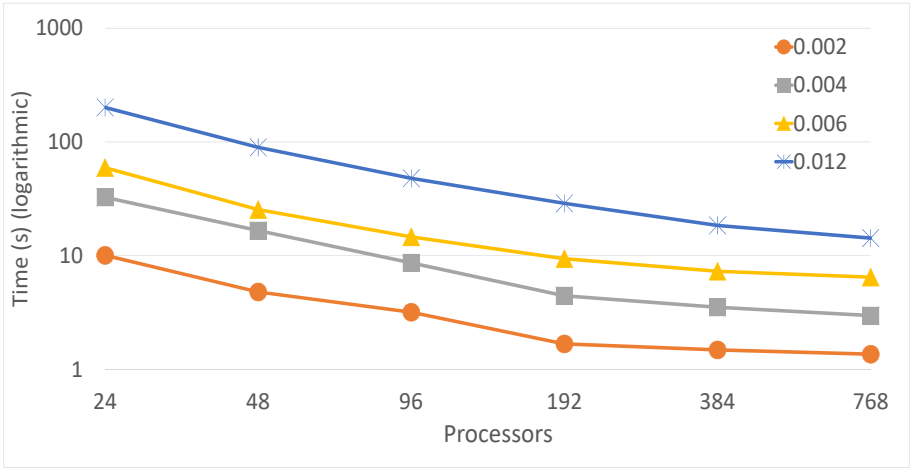

Fig. 14. Strong scaling for four banded matrices of the same size ($192k$), but with different bandwidth. The legend shows the density ($\frac{nonzero}{size^2}$) of each matrix.

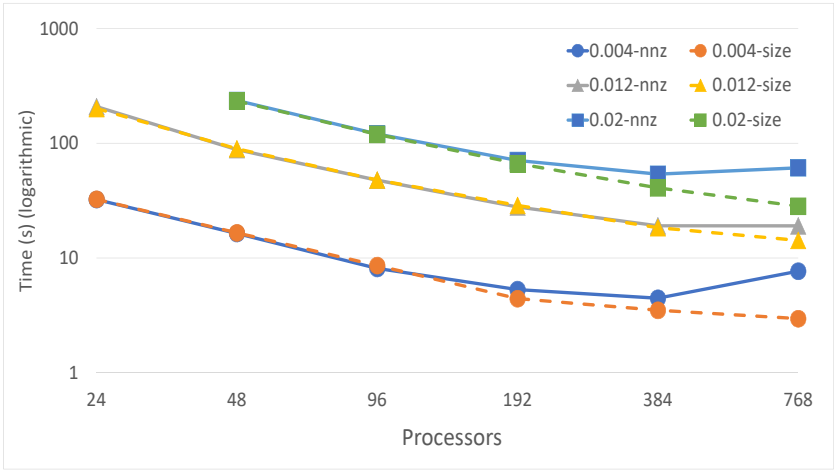

Fig. 15. Comparison of the the strong scaling when using two different splitting strategies: based on matrix size and based on number of nonzeros. The legend shows the density of each matrix together with the splitting strategy. The solid and dash lines show the splitting based on nonzeros and matrix sizes, respectively. The lines for the cases with the same densities have the same marker (e.g. square) to be easily comparable.

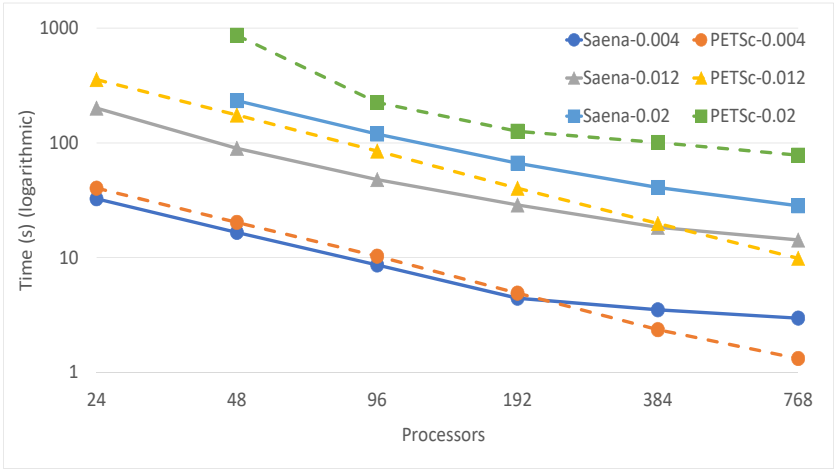

Fig. 16. Comparison of the the strong scaling between our solver (solid lines) and PETSc (dash lines). The legend shows the density of each matrix. The lines for the cases with the same densities have the same marker (e.g. square) to be easily comparable.

REFERENCES

[1] J. E. Dendy, Jr., "Black box multigrid," *Journal of Computational Physics*, vol. 48, no. 3, pp. 366–386, 1982.

[2] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 2015, pp. 804–811.

[3] S. M. Van Dongen, "Graph clustering by flow simulation," Ph.D. dissertation, 2000.

[4] J. R. Gilbert, S. Reinhardt, and V. B. Shah, "A unified framework for numerical and combinatorial computing," *Computing in Science & Engineering*, vol. 10, no. 2, pp. 20–25, 2008.

[5] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[6] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, D. A. May, L. C. McInnes, K. Rupp, B. F. Smith, S. Zampini, H. Zhang, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2017. [Online]. Available: http://www.mcs.anl.gov/petsc

[7] A. Buluç and J. R. Gilbert, "The combinatorial blas: design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496–509, 2011.

[8] J. Poulson, B. Marker, R. A. van de Geijn, J. R. Hammond, and N. A. Romero, "Elemental: A new framework for distributed memory dense matrix computations," *ACM Transactions on Mathematical Software*, vol. 39, no. 2, pp. 13:1–13:24, 2013.

[9] A. Bienz, R. D. Falgout, W. Gropp, L. N. Olson, and J. B. Schroder, "Reducing parallel communication in algebraic multigrid through sparsification," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. S332–S357, 2016.

[10] Y. Maday and R. Muñoz, "Spectral element multigrid. II. Theoretical justification," *Journal of scientific computing*, vol. 3, no. 4, pp. 323–353, 1988.

[11] J. H. Bramble and X. Zhang, "The analysis of multigrid methods," in *Handbook of numerical analysis, Vol. VII*, ser. Handb. Numer. Anal., VII. Amsterdam: North-Holland, 2000, pp. 173–415.

[12] S. C. Brenner, "Smoothers, mesh dependent norms, interpolation and multigrid," *Applied Numerical Mathematics*, vol. 43, no. 1-2, pp. 45–56, 2002, 19th Dundee Biennial Conference on Numerical Analysis (2001).

[13] A. Gholami, D. Malhotra, H. Sundar, and G. Biros, "Fft, fmm, or multigrid? a comparative study of state-of-the-art poisson solvers for uniform and nonuniform grids in the unit cube," *SIAM Journal on Scientific Computing*, vol. 38, no. 3, pp. C280–C306, 2016. [Online]. Available: https://doi.org/10.1137/15M1010798

[14] D. Braess, "On the combination of the multigrid method and conjugate gradients," in *Multigrid Methods II*, W. Hackbusch and U. Trottenberg, Eds. Berlin: Springer–Verlag, 1986, pp. 52–64.

[15] O. Tatebe and Y. Oyanagi, "Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines," in *Supercomputing'94. Proceedings*. IEEE, 1994, pp. 194–203.

[16] P. Vanek, J. Mandel, and M. Brezina, "Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems," Denver, CO, USA, Tech. Rep., 1995.

[17] P. Vaněk, M. Brezina, J. Mandel *et al.*, "Convergence of algebraic multigrid based on smoothed aggregation," *Numerische Mathematik*, vol. 88, no. 3, pp. 559–579, 2001.

[18] H. Sundar, G. Biros, C. Burstedde, J. Rudi, O. Ghattas, and G. Stadler, "Parallel geometric-algebraic multigrid on unstructured forests of octrees," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 43:1–43:11. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389055

[19] E. Treister and I. Yavneh, "Non-galerkin multigrid based on sparsified smoothed aggregation," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. A30–A54, 2015.

[20] N. Bell, S. Dalton, and L. N. Olson, "Exposing fine-grained parallelism in algebraic multigrid methods," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C123–C152, 2012.

[21] Y. Notay, "An aggregation-based algebraic multigrid method," *Electronic transactions on numerical analysis*, vol. 37, no. 6, pp. 123–146, 2010.

[22] H. Guillard and P. Vanek, "An aggregation multigrid solver for convection-diffusion problems on unstructured meshes." Tech. Rep., 1998.

[23] Y. Notay, "Aggregation-based algebraic multilevel preconditioning," *SIAM J. Matrix Analysis Applications*, vol. 27, no. 4, pp. 998–1018, 2006.

[24] R. Yuster and U. Zwick, "Fast sparse matrix multiplication," *ACM Trans. Algorithms*, vol. 1, no. 1, pp. 2–13, Jul. 2005. [Online]. Available: http://doi.acm.org/10.1145/1077464.1077466

[25] A. Buluç and J. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012. [Online]. Available: https://doi.org/10.1137/110848244

[26] F. Gremse, A. Höfter, L. Schwen, F. Kiessling, and U. Naumann, "Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging," *SIAM Journal on Scientific Computing*, vol. 37, no. 1, pp. C54–C71, 2015. [Online]. Available: https://doi.org/10.1137/130948811

[27] E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Performance evaluation of sparse matrix multiplication kernels on intel xeon phi," in *Parallel Processing and Applied Mathematics*, R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Waśniewski, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 559–570.