A Divide and Conquer Distributed Matrix-Matrix Multiplication

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

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

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

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 surpasses PETSc when the density of the matrix increases. We demonstrate the efficacy of our methods by comparing our GEMM with PETSc for a variety of matrices with different levels of sparsity.

KEYWORDS

Matrix-Matrix Product, Algebraic Multigrid, AMG, Iterative Solver, Sparse, Preconditioned Conjugate Gradient, Preconditioner

ACM Reference Format:
Majid Rasouli, Robert M. Kirby, and Hari Sundar. 2019. A Divide and Conquer Distributed Matrix-Matrix Multiplication. In *Proceedings of ACM Conference (ICPP'19).* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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) [7], 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 [1], graph clustering [20], breadth first search [9], amongst others [12]. While there are several highly-tuned distributed-memory matrix libraries available, they usually target either sparse [**? ?**] or dense [**?**] 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.

1.1 Background - AMG

Majid ►*I moved this from the old intro. Trim as neccessary.*◄

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 $O(n)$ 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 [4, 5, 8, 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) [3, 18], the solver of choice when solving engineering applications that require large-scale parallel solution approaches.

ICPP'19, August 2019, Kyoto, Japan

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 [7, 21, 22], it does not scale as well as GMG [17]. 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 scalbility of setup phase is also poor and typically worse than that of the solve phase. This is limited the attractiveness of AMG for large systems. In this work, we develop an efficient sparse matrix multiplication (MATMULT) algorithm to improve the performance and scalability of AMG. As will be explained in the following section, a sparse MATMULT is the dominant part of the AMG setup. While several of our optimizations apply to sparse MATMULT in general, and can be more broadly applied, our strategies for reducing inter-process communication make use of the special structure of MATMULT 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 [2], [15], [11], [14]. For this paper, *maximal independent set* from [2], 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 MATMULT 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 MATMULT 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:      x ← direct solver(g[L + 1], x, b)
3:  else
4:      x ← Smoother(g, x, b, l)
5:      r ← As[l] × x − b
6:      r_c ← Rs[l] × r
7:      y_c ← vcycle(g, x, r_c, l + 1)
8:      y ← Ps[l] × y_c
9:      x ← x − y
10:     x ← Smoother(g, x, b, l)
11: end if
```

Smoothed Aggregation AMG (SA-AMG)[22] 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.

1.2 Related Work

While significant research has been done on improving the efficiency and scalability of sparse matrix-vector products, sparse MATMULT in comparison has received far less attention. Yuster and Zwick [23] 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 [6], Buluc and Gilbert present algorithms for parallel sparse MATMULT using a two-dimensional block data distributions with serial hypersparse kernels. Gremse *et al.* [10] the authors present a promising algorithm using iterative row merging to improve the performance on GPUs. Similarly, Saule *et al.* [16] evaluate the performance of sparse matrix multiplication kernels on the Intel Xeon Phi. Most AMG implementation have relied on

standard sparse MATMULT implementations without any special considerations for the structure of the matrices generated within AMG. This work attempts to fill this gap.

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

- A new efficient MATMULT algorithm.
- A new method to communicate data required to do MATMULT specific to Algebraic Multigrid

The rest of the paper is organized as follows. In §2 we discuss the different strategies used to improve the performance and scalability of MATMULT. In §3 we show the strong and weak scaling of our methods and compare our method with PETSc. Finally, we conclude the paper in §4.

2 METHODS

In this section, we present our matrix-matrix multiplication algorithm, the choice of the stop condition for the recursive function and how matrices are being divided to smaller blocks. We then explain how the communication is being done in the distributed fashion to help the recursive function scale better.

2.1 Matrix-Matrix Multiplication

We design a divide and conquer approach to perform the sparse MATMULT 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.

To perform the multiplication

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

we keep splitting the matrices horizontally and vertically (Figure 1) based on row size and column size of A, until we can fit the result of the multiplication in a dense matrix.

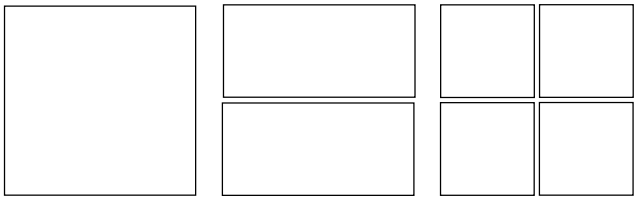

Figure 1: A basic scheme that shows splitting the matrix first horizontally, then vertically.

The recursive function, RECURS_MATMULT, includes three cases:

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

2.1.1 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 matrices gets 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, the obvious choice to decide when to stop the recursion is Equation (4), but Figure 2 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, but the top left block should be divided to more blocks than the top right block.

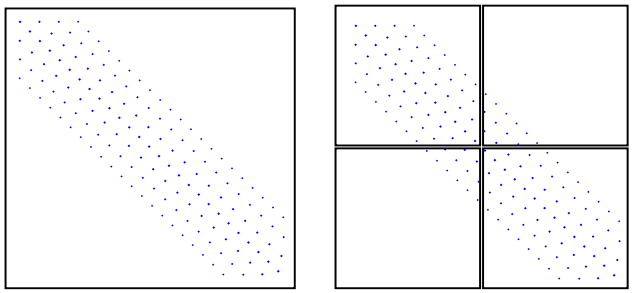

Figure 2: Using row and column sizes to decide when to stop the recursion is not efficient, because the top left block is the same size as the top right one, but it should be divided to more sub-blocks.

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, the number of nonzero rows of A (*A_nnz_row*) and nonzero columns of B (*B_nnz_col*) are being computed. 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 $O(n^2)$ 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 of the effectiveness of our method in a practical situation, we have implemented the RECURS_MATMULT function in our Algebraic Multigrid solver, called *Saena*, and performed MATMULT twice to compute the coarse matrix based on the 3D Poisson Problem.

In Figure 3, 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 4 shows the same experiment for matrix ID 1882 from SuiteSparse (Florida) Matrix Collection, which is a sparse matrix of size $1M$ and $5M$ nonzero.

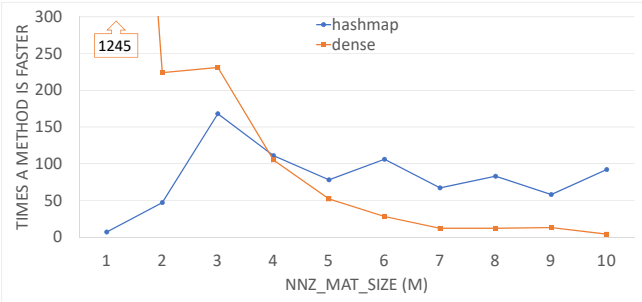

Figure 3: Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is the 3D Poisson problem of size $216k$. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

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 3 and 4 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.

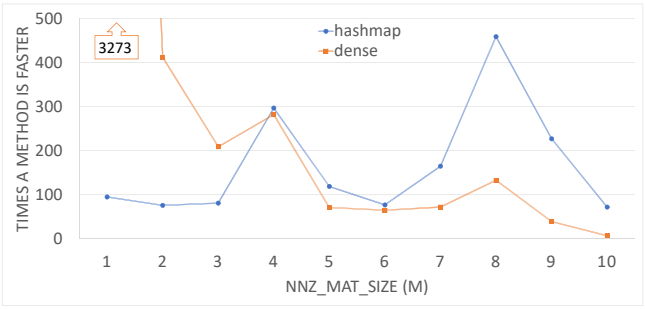

Figure 4: Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is matrix ID 1882 from SuiteSparse (Florida) Matrix Collection. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

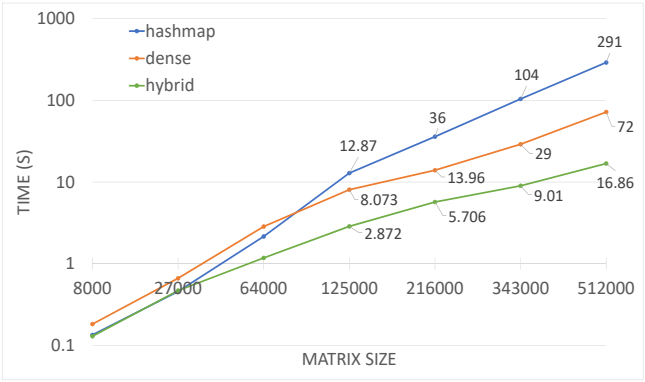

Figure 5: Comparison between the three methods to do Case 1: only using hashmap, only using the dense structure, and the hybrid method. They are used as Case 1 to compute the first coarse matrix (the triple multiplication) on 7 matrices (3D Poisson) of different sizes.

Figure 5 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.

2.1.2 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 6). 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_MATMULT 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 first recursive function is finished, we will perform a sorting and then add the duplicates only once at the end.

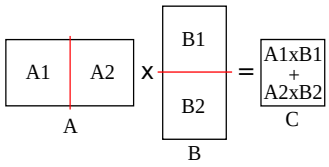

Figure 6: Case 2: When A is horizontal, split A by column and B by row. Call the recursive function twice.

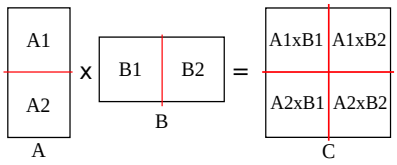

Figure 7: Case 3: When A is vertical, split A by row and B by column. Call the recursive function four times.

Algorithm 2 Case 2: C = RECURS_MATMULT2(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_MATMULT($A1, B1$)
4: $C \leftarrow$ RECURS_MATMULT($A2, B2$)

2.1.3 Case 3. When A is vertical, i.e. its row size is greater than its column size, we halve A by row based on its row size and halve B by column (Figure 7). In contrary to the previous case, the column size of B is not related to row size of A, so they are split independently. This time the RECURS_MATMULT 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 it is faster to do the sorting and adding duplicates at the end on a smaller set of entries.

Algorithm 3 Case 3: C = RECURS_MATMULT3(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_MATMULT($A1, B1$)
4: $C \leftarrow$ RECURS_MATMULT($A2, B1$)
5: $C \leftarrow$ RECURS_MATMULT($A1, B2$)
6: $C \leftarrow$ RECURS_MATMULT($A2, B2$)

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. After a series of experiments we noticed that the splitting method based on size scales better than splitting based on nonzeros.

2.1.4 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.

2.2 Communication

In the previous section we explained how to perform MATMULT 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 show how to improve performance and scalability if B is symmetric.

Matrices are partitioned across multiple processors by row blocks (Figure 8). Since matrices A and B may have different number of rows, they may not be partitioned the same way. We avoid communicating A 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. That's the reason RECURS_MATMULT is between the *Isend-Irecv* part and *wait*. This way we use a portion of the (or sometimes the whole) time that the communication takes, to execute the multiplication.

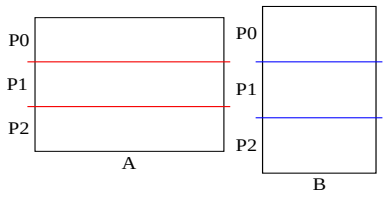

Figure 8: Partitioning of the matrices across the processors in row blocks.

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(B) from right neighbor
5: $C_i \leftarrow$ RECURS_MATMULT(A_i, B_send)
6: wait for Isend and Irecv to finish
7: $swap(B\_send, B\_recv)$
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 9 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).

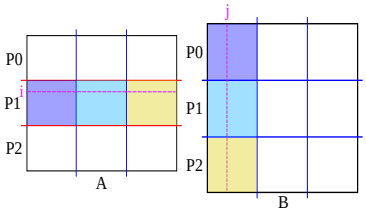

Figure 9: Column j of B is stored on different processors, so to compute entry C_{ij} we need to multiply the parts of A and B with the same color.

Since in this case B is symmetric, we can work with its local transpose. Instead of working with B_i's, we consider their local transpose BT_i in Figure 10.

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 for all the matrices in our algorithm and we don't want to change the partition for one matrix only to perform RECURS_MATMULT, we assumed B being symmetric.

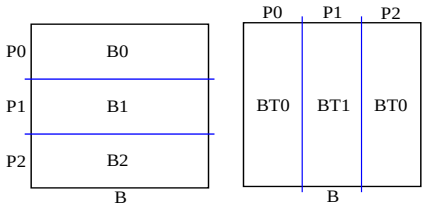

Figure 10: In case of a symmetric B, we use local transpose of its row blocks.

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(local transpose of B) from right neighbor
5: $C_{ik} \leftarrow$ RECURS_MATMULT(A_i, B_send)
6: *wait for Isend and Irecv to finish*
7: $swap(B\_send, B\_recv)$
8: locally sort C_{ik} and add duplicates
9: **end for**
10: put all C_{ik}'s into C_i

There are at least two advantages for changing Algorithm 4 to Algorithm 5. First, we can sort and add duplicates on a subset of entries of C_i (compare line 9 of Algorithm 4 with line 8 of Algorithm 5. The other benefit is having better divided blocks of B after splitting.

3 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 11 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. There is no data in the plot for the matrix with density 0.02 on 24 processors because it was too big to fit in the memory on one node.

Figure 12 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, especially when the matrices are relatively denser.

Figure 13 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

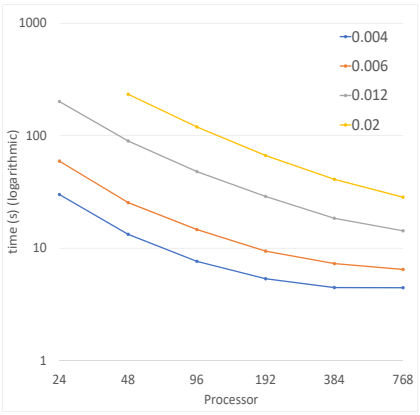

Figure 11: Strong scaling for four banded matrices of the same size ($192k$), but with different bandwidth. The legend shows the density ($\frac{nonzero}{size^2}$) of each matrix.

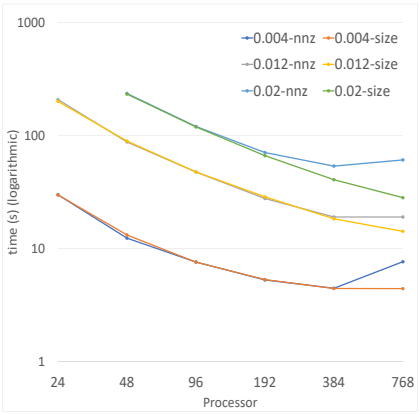

Figure 12: Comparison of the the strong scaling when using two different splitting strategies: based on matrix size and based on number of nonzeros. The legend shows the density of each matrix together with the splitting strategy.

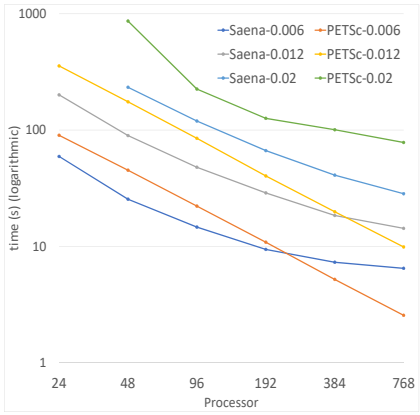

Figure 13: Comparison of the the strong scaling between our solver and PETSc. The legend shows the density of each matrix.

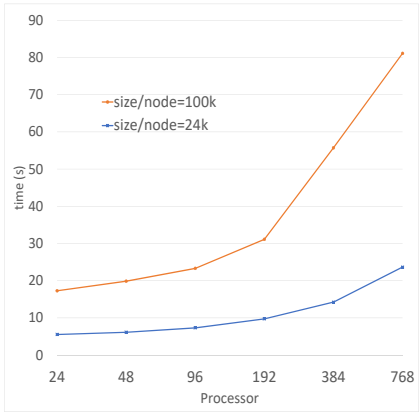

Figure 14: Weak scaling for two banded matrices: blue line shows the one with $24k$ on each node ($1k$ on each core) and red line shows a larger one with $100k$ on each node (4166 on each core).

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 MATMULTat those levels, so switching to our algorithm for lower levels of multigrid would improve the performance and scalibility significantly.

Figure 14 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.

4 CONCLUSION

We have presented a divide and conquer approach to improve the efficiency of matrix-matrix multiplication. We have also designed a low-communication algorithm specific to AMG, to improve the efficiency of the AMG setup phase. 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.

REFERENCES

[1] Ariful Azad, Aydin Buluç, and John Gilbert. 2015. Parallel triangle counting and enumeration using matrix algebra. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. IEEE, 804–811.

[2] Nathan Bell, Steven Dalton, and Luke N Olson. 2012. Exposing fine-grained parallelism in algebraic multigrid methods. *SIAM Journal on Scientific Computing* 34, 4 (2012), C123–C152.

[3] D. Braess. 1986. On the combination of the multigrid method and conjugate gradients. In *Multigrid Methods II*, W. Hackbusch and U. Trottenberg (Eds.). Springer–Verlag, Berlin, 52–64.

[4] J. H. Bramble and X. Zhang. 2000. The analysis of multigrid methods. In *Handbook of numerical analysis, Vol. VII*. North-Holland, Amsterdam, 173–415.

[5] Susanne C. Brenner. 2002. Smoothers, mesh dependent norms, interpolation and multigrid. *Applied Numerical Mathematics* 43, 1-2 (2002), 45–56. 19th Dundee Biennial Conference on Numerical Analysis (2001).

[6] A. BuluÃğ and J. Gilbert. 2012. Parallel Sparse Matrix-Matrix Multiplication and Indexing: Implementation and Experiments. *SIAM Journal on Scientific Computing* 34, 4 (2012), C170–C191. https://doi.org/10.1137/110848244 arXiv:https://doi.org/10.1137/110848244

[7] J. E. Dendy, Jr. 1982. Black box multigrid. *J. Comput. Phys.* 48, 3 (1982), 366–386.

[8] A. Gholami, D. Malhotra, H. Sundar, and G. Biros. 2016. 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* 38, 3 (2016), C280–C306. https://doi.org/10.1137/15M1010798 arXiv:https://doi.org/10.1137/15M1010798

[9] John R Gilbert, Steve Reinhardt, and Viral B Shah. 2008. A unified framework for numerical and combinatorial computing. *Computing in Science & Engineering* 10, 2 (2008), 20–25.

[10] F. Gremse, A. HÃűfter, L. Schwen, F. Kiessling, and U. Naumann. 2015. GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging. *SIAM Journal on Scientific Computing* 37, 1 (2015), C54–C71. https://doi.org/10.1137/130948811 arXiv:https://doi.org/10.1137/130948811

[11] HervÃĪ Guillard and Petr Vanek. 1998. *An Aggregation Multigrid Solver for Convection-diffusion Problems on Unstructured Meshes*. Technical Report.

[12] Jeremy Kepner and John Gilbert. 2011. *Graph algorithms in the language of linear algebra*. SIAM.

[13] Yvon Maday and Rafael Muñoz. 1988. Spectral element multigrid. II. Theoretical justification. *Journal of scientific computing* 3, 4 (1988), 323–353.

[14] Yvan Notay. 2006. Aggregation-Based Algebraic Multilevel Preconditioning. *SIAM J. Matrix Analysis Applications* 27, 4 (2006), 998–1018.

[15] Yvan Notay. 2010. An aggregation-based algebraic multigrid method. *Electronic transactions on numerical analysis* 37, 6 (2010), 123–146.

[16] Erik Saule, Kamer Kaya, and Ümit V. Çatalyürek. 2014. Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi. In *Parallel Processing and Applied Mathematics*, Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Waśniewski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 559–570.

[17] Hari Sundar, George Biros, Carsten Burstedde, Johann Rudi, Omar Ghattas, and Georg Stadler. 2012. Parallel Geometric-algebraic Multigrid on Unstructured Forests of Octrees. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '12)*. IEEE Computer Society Press, Los Alamitos, CA, USA, Article 43, 11 pages. http://dl.acm.org/citation.cfm?id=2388996.2389055

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

[19] Eran Treister and Irad Yavneh. 2015. Non-Galerkin multigrid based on sparsified smoothed aggregation. *SIAM Journal on Scientific Computing* 37, 1 (2015), A30–A54.

[20] Stijn Marinus Van Dongen. 2000. *Graph clustering by flow simulation*. Ph.D. Dissertation.

[21] Petr Vaněk, Marian Brezina, Jan Mandel, et al. 2001. Convergence of algebraic multigrid based on smoothed aggregation. *Numer. Math.* 88, 3 (2001), 559–579.

[22] Petr Vanek, Jan Mandel, and Marian Brezina. 1995. *Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems*. Technical Report. Denver, CO, USA.

[23] Raphael Yuster and Uri Zwick. 2005. Fast Sparse Matrix Multiplication. *ACM Trans. Algorithms* 1, 1 (July 2005), 2–13. https://doi.org/10.1145/1077464.1077466