A Compressed, Divide and Conquer Algorithm for Scalable Distributed Matrix-Matrix Multiplication

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

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

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

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 dense, as in the case of AMG matrix hierarchies. In addition, we implement a lossless data compression method to reduce the communication cost. We combine this with an efficient communication pattern during distributed-memory GEMM to provide 2.24 times (on average) better performance than the state-of-the-art library PETSc. Additionally, we show that the performance and scalability of our method surpass PETSc even more when the density of the matrix increases. We demonstrate the efficacy of our methods by comparing our GEMM with PETSc on a wide range of matrices.

CCS CONCEPTS

• **Computing methodologies → Parallel computing methodologies**; **Distributed algorithms**; • **General and reference → Performance**; • **Mathematics of computing → Mathematical software performance**.

HPCAsia 2021, January 20–22, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8842-9/21/01…$15.00
https://doi.org/10.1145/3432261.3432271

KEYWORDS

sparse matrix product, dense matrix multiplication, GEMM, algebraic multigrid, AMG, numerical linear algebra, parallel computing, matrix compression, Golomb-Rice encoding

ACM Reference Format:
Majid Rasouli, Robert M. Kirby, and Hari Sundar. 2021. A Compressed, Divide and Conquer Algorithm for Scalable Distributed Matrix-Matrix Multiplication. In *The International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia 2021), January 20–22, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3432261.3432271

1 INTRODUCTION

Matrix-matrix multiplication (GEMM) is a key linear algebra primitive commonly used by the computational and data science communities. Examples include operations used as part of the setup phase of algebraic multigrid methods (AMG) [6], an example that we will use in this work to demonstrate the effectiveness of our methods, as well as large-scale graph analytics, where a linear algebra formulation is used, such as triangle counting [1], graph clustering [15], breadth first search [7], amongst others [9]. While there are several highly-tuned distributed-memory matrix libraries available, they usually target either sparse [2, 4] or dense [10] 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. 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 dominant cost of a distributed GEMM is usually the communication part. The situation gets even worse when the matrices become denser. This causes high idle time for processors waiting until the communication is finished. To address this situation, we have implemented a lossless data compression. Before sending the data, processors compress it. Then, after receiving the data, they decompress it and use it. Following this method, we show significant reduction in the communication cost and improved scalability for the GEMM operation.

We demonstrate the effectiveness of our algorithms and data-structures by comparing with PETSc [2] and demonstrate performance comparable to, and in some cases better than, 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.

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;
- A lossless data compression for data exchange; 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 where variable sparsity patterns are encountered. We provide a brief review of related work in §1.2. In §2 we discuss the different strategies used to improve the performance and scalability of GEMM. 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.

1.1 Background - AMG

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. During the setup phase, a hierarchy of matrices is computed based on the matrix A. The hierarchy then will be used in the solve phase to get the solution x. The dominant cost of the setup phase is the matrix-matrix multiplications needed to compute the coarse-grid approximation (Galerkin approximation). The multiplications gets even more costly in deeper levels of the hierarchy, because sparse matrix products cause fill-in, i.e. increasing the number of nonzeros. This is where a GEMM that performs efficient for a wide range of sparsity rates becomes important and the scalability of the whole AMG would greatly depend on it.

While AMG is highly attractive due to its black-box nature [6, 16, 17], it does not scale well due to the loss of sparsity at coarser levels arising from the Galerkin approximation [14], leading to poor scalability, especially at the coarser levels. In this paper, we develop a GEMM algorithm that would perform well for all ranges of sparsity rates, for which AMG would be a great application case.

1.2 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 [18] 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 [3], Buluc and Gilbert present algorithms for parallel sparse GEMM using a two-dimensional block data distributions with serial hypersparse kernels. Gremse *et al.* [8] present a promising algorithm using iterative row merging to improve the performance on GPUs. Similarly, Saule *et al.* [13] 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.

2 METHODS

In this section, we present our divide and conquer GEMM algorithm. We first explain our 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. In addition, we present our compression method to further reduce the communication cost.

2.1 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.

To perform the multiplication

$$C = A \times B \tag{2}$$

we keep splitting the matrices horizontally and vertically (Figure 1) based on row size and column size of the matrices, until we can fit the result of the multiplication in a pre-allocated buffer.

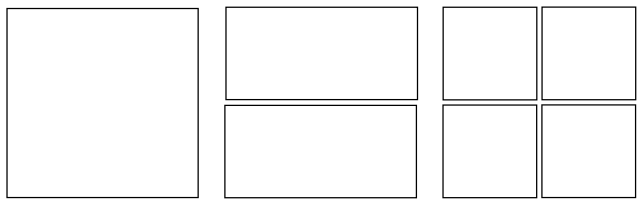


Figure 1: A basic scheme that shows splitting the matrix first horizontally, then vertically.

The recursive function, RECURS_GEMM, includes three cases:

(1) Case 1: Stop the recursion and perform the multiplication.
(2) Case 2: A is horizontal ($row\ size \leq col\ size$). Split A column-wise and B row-wise.
(3) Case 3: A is vertical ($row\ size > col\ size$). Split A row-wise and B column-wise.

2.1.1 Case 1. For this part, we have tried three methods:

(1) Dense buffer;
(2) Intel MKL's *mkl_dcsrmultcsr*; and
(3) Intel MKL's *mkl_sparse_spmm*.

First we explain our dense buffer implementation, since the main idea of the three methods is the same. Our goal is to fit the multiplication result of blocks of A and B in a dense buffer. We represent 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

$$row\ size\ of\ A_{ij} \times column\ size\ of\ B_{lk}. \tag{3}$$

Therefore, Equation (3) can be used as the naive choice to decide when to stop the recursion, but Figure 2 shows why that is not a good choice. If we use Equation (3) 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.

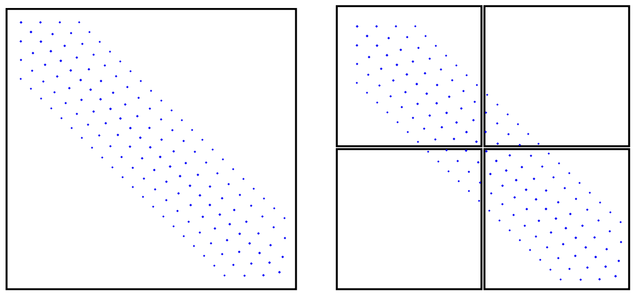


Figure 2: Using row and column sizes to decide when to stop the recursion is not efficient, because the top left block is the same size as the top right one, but it should be divided to more sub-blocks.

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

$$\text{NNZ\_MAT\_SIZE} := A\_nnz\_row \times B\_nnz\_col \tag{4}$$

is set. Our code has a profiling function that allows us to empirically determine the appropriate threshold by running several test cases. This is machine dependent and needs to be done only once on a new machine.

We allocate a memory block of size of the threshold once, before starting the matrix product, to use for all the multiplications that we do in the recursive calls.

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, we 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.

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.

As the next steps, to perform the multiplication when we stop the splitting, we have tried two of the Intel's MKL sparse-sparse matrix multiplication functions, *mkl_sparse_spmm* and *mkl_dcsrmultcsr*. The former, *mkl_sparse_spmm*, is the newer function in Sparse BLAS part of MKL. This function does not have much memory management. So, we cannot follow our approach to allocate a buffer first, then reuse that to store the results of small blocks of A and B. On the other hand, *mkl_dcsrmultcsr* is perfect for our algorithm. To use it, the memory for the result matrix C should be pre-allocated. Also, there is no function overhead to use this function, like the previous case.

One adjustment that we needed to use this function was related to the sparse format. Our matrices are stored in the CSC format, but this function only works on CSR matrices. Since the transpose of a matrix stored in the CSC format, would be in CSR, we use the following fact and pass the matrices A and B in the opposite order to the function and have the result matrix C back in CSC format:

$$A_{CSC} \times B_{CSC} = (B_{CSR}^T \times A_{CSR}^T)^T = C_{CSR}^T = C_{CSC}. \quad (5)$$

Comparing the three methods mentioned in this section, *mkl_dcsrmultcsr* was significantly faster than the other two functions. We use this method for the rest of the paper.

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 3). 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 1). The results of the two multiplications need to be added together at the end. This results in there being entries in the result matrix that have 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.

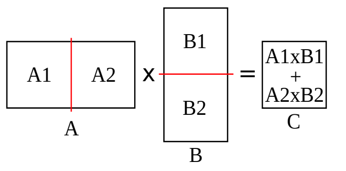


Figure 3: Case 2: When A is horizontal, split A by column and B by row. Call the recursive function twice.

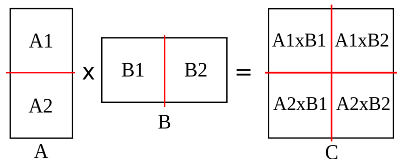


Figure 4: Case 3: When A is vertical, split A by row and B by column. Call the recursive function four times.

Algorithm 1 Case 2: C = RECURS_GEMM2(A, B)

```
Input: A, B
Output: C
1: $(A1, A2) = \text{SPLIT\_BY\_COL}(A)$
2: $(B1, B2) = \text{SPLIT\_BY\_ROW}(B)$
3: $C \leftarrow \text{RECURS\_GEMM}(A1, B1)$
4: $C \leftarrow \text{RECURS\_GEMM}(A2, B2)$
```

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 and B by column (Figure 4). This time the RECURS_GEMM will be called four times (Algorithm 2). 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 2 Case 3: C = RECURS_GEMM3(A, B)

```
Input: A, B
Output: C
1: $(A1, A2) = \text{SPLIT\_BY\_ROW}(A)$
2: $(B1, B2) = \text{SPLIT\_BY\_COL}(B)$
3: $C \leftarrow \text{RECURS\_GEMM}(A1, B1)$
4: $C \leftarrow \text{RECURS\_GEMM}(A2, B1)$
5: $C \leftarrow \text{RECURS\_GEMM}(A1, B2)$
6: $C \leftarrow \text{RECURS\_GEMM}(A2, B2)$
```

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.

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

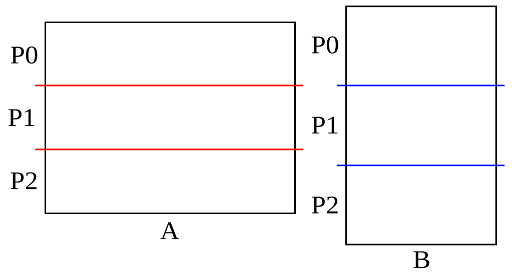


Figure 5: Partitioning of the matrices across the processors in row blocks.

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 RECURS_GEMM if the data is available locally. In this section, we explain how the communication is done to perform $C = A \times B$ in a distributed fashion. Then we demonstrate how we reduce the communication cost by applying an integer compression method.

2.2.1 Overlapped Communication. Matrices are partitioned across multiple processors by row blocks (Figure 5). 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 3 shows how the communication is done. We use *MPI* for communicating sub-matrices between processors. *nprocs* is the total number of processors and *myrank* is each processor's rank. *Isend* and *Irecv* are MPI's non-blocking send and receive commands, which means sending and receiving data gets started but the program does not stop for it to finish. Instead, it runs the next commands until it reaches the *wait* command.

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 is done so that 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. Another advantage of using an overlapped approach will be explained in the compression part.

2.2.2 Compression. We have implemented our GEMM algorithm in C++. The matrices are stored in the CSC (Compressed Sparse Column) format, in which 3 arrays are needed: one to store row indices, the second one to have scan on columns and the third one to store values of the matrix entries. We use data-type `int` for the first two arrays and `double` for the values.

Algorithm 3 $C_i = A_i \times B$

```
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
```

The communication cost is usually the dominant cost to perform our GEMM, especially when the matrices become denser. To reduce the communication cost, we designed a lossless compression method to reduce the size of the `int` arrays. Then, each processor after receiving the sub-matrix, decompresses it and uses it.

We designed a compression method based on the Golomb-Rice encoding [12] algorithm, which is a lossless integer compression method. First, we choose a fixed integer number M. Then we can write each integer number a as:

$$a = q \times M + r \tag{6}$$

in which, q is the quotient of a/M and r is the remainder. Now, we have 2 smaller integer numbers q and r. We store r in bit arrays, but do not encode q. For any given integer array, the goal is to choose an M which is not very big but makes the quotient (q) zero for most entries of the array.

By choosing M as a power of 2, we can make divisions by M possible by bit operations. If $M = 2^k$, then a/M is equivalent to $a >> k$. The same is true for multiplications. $q \times M$ can be done by $q << k$.

We need to divide by M for compression and multiply by M for decompression, which now can be done very fast by shifting bits. Also, we can store r in k bits and use 1 bit to know if q was zero or non-zero (we call that the $q - bit$). Finally, if q was non-zero, we set the $q - bit$ to 1 and store q as a `short` integer, which usually takes 2 bytes (half of `int`).

We can have a better compression rate, if more q's are zero. To achieve that, to compress an array A, instead of compressing each entry, we do so on the differences of the entries. So, to compress

$A[0],\ A[1],\ A[2],\ A[3], \ldots$

we apply the compression on:

$A[0],\ A[1] - A[0],\ A[2] - A[1],\ A[3] - A[2], \ldots.$

We have implemented a part to check the array that we want to compress, and decide between two values for k, 7 or 15. The reason is that, we need $k + 1$ bits to store the remainder part and the $q - bit$, and it is more efficient to do the compression and decompression if $k+1$ is a multiple of 8. Because each byte is 8 bits and processing complete bytes is faster than doing so on two or more partial bytes. To make it more clear, let us assume k is 5. Then, to write a compressed number, we need to write 6 bits of a byte. To compress the next number we need to do the same on the last 2 bits of the current byte and 4 bits of the next byte. Processing partial bytes adds a big processing time to our operations, so we keep whole bytes for the compression method.

If the difference between consequent entries in the arrays are big, then using $k = 7$ may result in high number of non-zero q's, because any difference which is higher than $M = 2^7 = 128$ will have a non-zero q. That's the reason we also check $k = 15$. In this case, any number less than $M = 2^{15} = 32768$ has a zero q, so we do not need to store that quotient.

In the best case scenario ($k = 7$ and zero q), we would need $7 + 1 = 8$ bits (1 byte) to store integers (which are 4 bytes). In this case, we save %75. In the worst case ($k = 15$ and non-zero q), we would need $15 + 1 = 16$ bits (2 bytes) for the remainder and 2 bytes to store the quotient, which would be the same size as `int`.

Algorithm 4 shows the communication algorithm after applying the compression method. We see another advantage of using the overlapped communication here, to hide the decompression cost.

Table 1 shows how much we save by applying our compression method on 8 matrices from the SuiteSparse Matrix Collection. We present the result of our compression method on 335 matrices from that collection in the nex section.

```
Algorithm 4 $C_i = A_i \times B$
Input: $A_i, B$
Output: $C_i$ (result of $A_i \times B$)
 1: $B\_send \leftarrow compress(B_i)$
 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:     $B_j \leftarrow decompress(B\_send)$
 6:     $C_i \leftarrow$ RECURS_GEMM($A_i, B_j$)
 7:     wait for Isend and Irecv to finish
 8:     $swap\_pointers(B\_send, B\_recv)$
 9: end for
10: locally sort $C_i$ and add duplicates
```

Table 1: Sample compression rates on 8 matrices from the Florida Matrix Collection

Matrix ID	saving %
1	63
1213	56
1257	73
1402	56
1403	53
1412	50
1421	51
1883	50

3 NUMERICAL RESULTS

Our platform is called Saena. It 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 Frontera at the Texas Advanced Computing Center (TACC). The configuration of each compute node is described below:

- Processors:
  - Intel Xeon Platinum 8280 ("Cascade Lake")
  - Number of cores: 28 per socket, 56 per node.
  - Clock rate: 2.7Ghz ("Base Frequency")
  - "Peak" node performance: 4.8TF, double precision
- Memory: $DDR - 4$ memory, 192GB/node
- Network: Mellanox InfiniBand, HDR-100

For these experiments we have used matrices from the SuiteSparse Matrix Collection (formerly known as the Florida Matrix Collection) [5]. We have used the symmetric matrices that have at least 50k number of rows and columns. For the compression experiment, we have used 335 matrices. For the PETSc comparison experiments, we have used 264 of them, because of some limitations, such as lack of enough memory (for matrices with very high number of nonozeros) or the inability of our software to partition the irregular cases between processors before calling GEMM. The symmetry condition is not a requirement for our method. We need to pass the transpose the right-hand matrix to our GEMM function. For simplicity, we have chosen the symmetric cases.

To perform the multiplication, we multiply the 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.

As the first experiment, we show the efficacy of our compression method. We perform the compression on a matrix in the CSC (Compressed Sparse Column) format, which has two integer arrays for the indices and one floating point array for the values. We perform the compression only on the integer arrays.

Figure 6 shows the compression rate on 335 matrices. We use the term compression rate as the percentage we reduce the data size, so $\frac{original\ size - compressed\ size}{original\ size}$. The x axis shows the least compression rate and the y axis shows the cumulative percentage of the matrices that we can compress up to any rate. Point (x, y) on the line tells us we can compress $\%x$ of the matrices to at least $\%y$ compression rate. For instance, we can gain at least %30 compression rate on all the matrices, %70 compression rate on almost %20 of the matrices and we can not compress any matrix higher than %75 compression rate. These numbers show the rate only for integer arrays of the matrix, and does not include the floating point array. The total compression rate that we gain on the whole matrix is in the $\%20 - 25$ range.

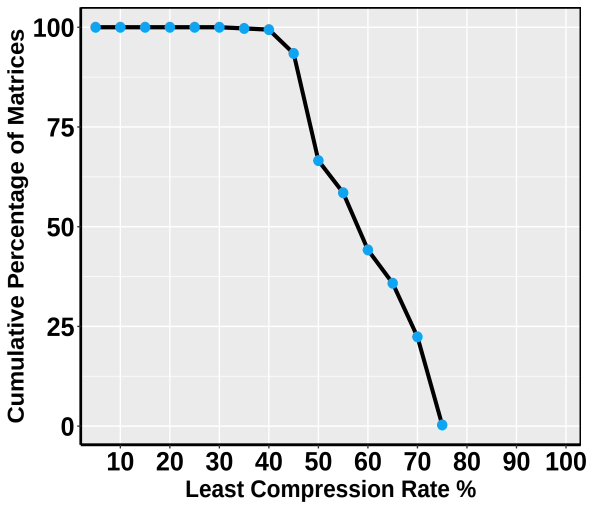


Figure 6: This plot shows the compression rate on 335 matrices. It shows how much we can reduce the data size by using our compression method.

Figure 7 shows the average speed-up of one GEMM on 264 matrices when using our method, comparing with PETSc. These experiments are done on 32 nodes, each with 28 MPI tasks. The horizontal dashed line at 1 shows the base for the comparison. Any dot above that line shows the speed-up we gain by using our method and any dot below that shows we lose performance. 207 matrices are above the red line, so we gain speed-up on %78 of the cases. The average gain is $2.24x$, so on average we gain more than twice execution time boost.

The majority of the cases that our method is slower, are the matrices that are very close to diagonal matrices. Our method, because of its divide and conquer nature, would spend time on any sub-blocks of the matrix, even the ones that only have very few nonzeros, which causes our method to be slower than PETSc. The cases that we are significantly ahead of PETSc, are the ones that have nonzeros far from the diagonal of the matrix. Our algorithm shows its capabilities in these cases, because it is agnostic about on what parts of the matrix it is performing the multiplication.

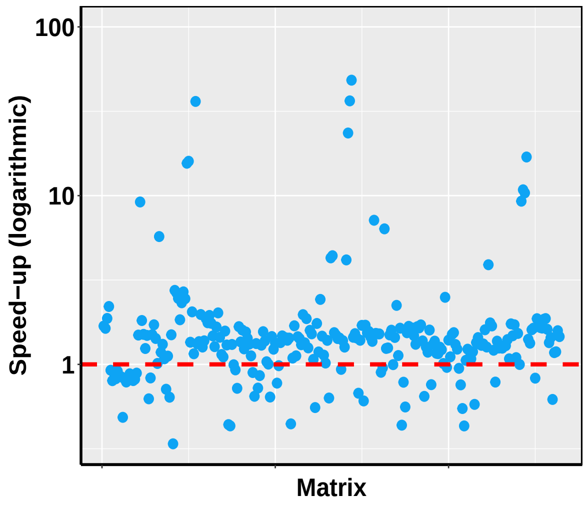
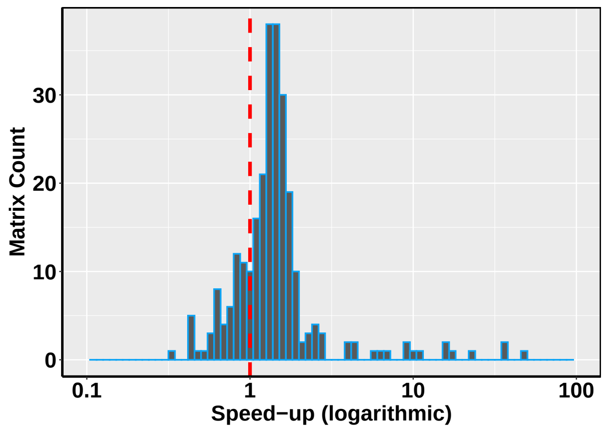


Figure 7: This plot shows the average speed-up of one GEMM on 264 matrices when using our method, comparing with PETSc. Dots above the red dashed line show the matrices that benefit from our multiplication method. GEMM is slower by using our method on the ones below the red line.

Figure 8 shows the histogram plot based on the same results. The x axis shows the speed-up we gain or lose and the height of each bar shows the number of matrices with that boost rate. Any bar on the right side of the red dashed line shows the gain and any bar on the left shows the cases with performance loss.

Our application, as explained in Section 1, is Algebraic Multigrid. The previous two plots show how effective our method is at the finest level of the multigrid hierarchy, if those matrices are used in an AMG solver. At the coarser levels of multigrid (levels 2 and above), the matrices get denser and our method performs even better on them. In the following plots, we compare one GEMM execution time using our method and PETSc in the cases that our method was slower than PETSc. To have a fair comparison, we first create a multigrid hierarchy ($5levels$) using our software. Then we call GEMM on those matrices, by both PETSc and our method. By following this heuristic, instead of creating the hierarchy separately, we can compare the GEMM operation on the exact same matrices at all levels of the hierarchy.

Figures 9 - 11 show GEMM comparison on three matrices for which our method was slower than PETSc. They



m

Figure 8: This plot is the histogram plot of speed-up gain or loss using our method comparing with PETSc. The matrices that are on the right side are the ones that perform GEMM faster based on our method and the ones on the left are slower than PETSc.

show GEMM at 5 levels of the hierarchy. Level 1 is the finest level. We observe that our method out-performs PETSc in the coarse levels, except one case. Also, we can see that the total matrix multiplication time is shorter by using our method.

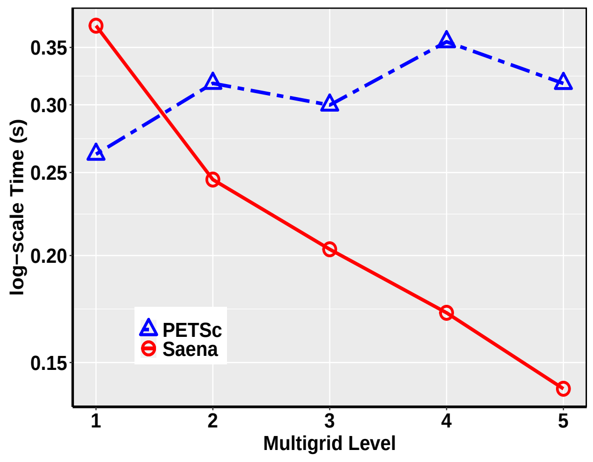


Figure 9: This plot compares the average time of one GEMM on a matrix (*ldoor*) for which our method is slower at the finest level. Our method out-performs PETSc at levels $2-5$**.**

To compare strong scaling, we have chosen 4 matrices from different speed-up gain or loss range. They are marked by the dashed lines in Figure 12. The strong scaling of one GEMM time for those matrices are shown with the same color in Figure 13. The solid lines show the time for our method

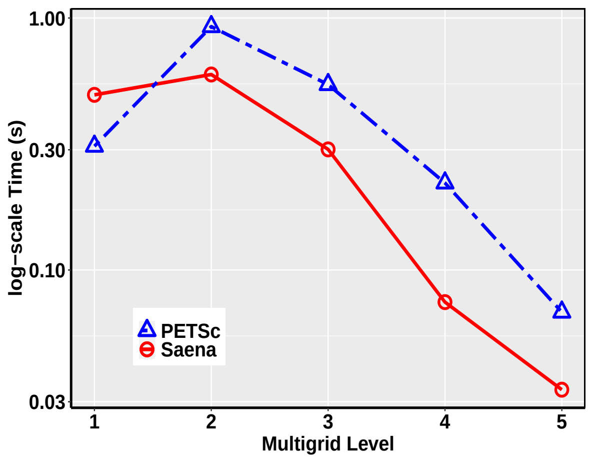


Figure 10: This plot compares the average time of one GEMM on a matrix (*Hook*_1498) for which our method is slower at the finest level. Our method out-performs PETSc at levels $2-5$**.**

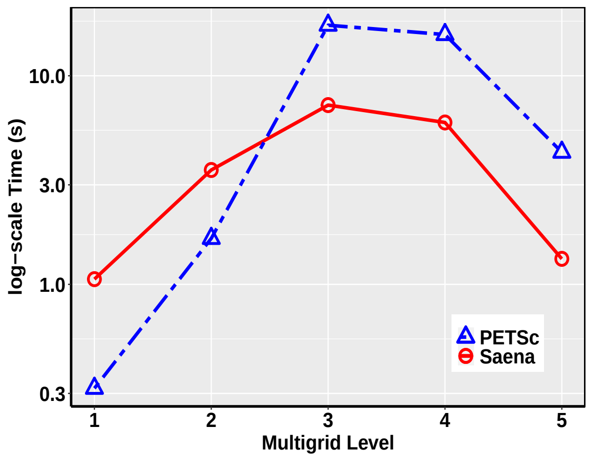


Figure 11: This plot compares the average time of one GEMM on a matrix (*Bump*_2911) for which our method is slower at the finest level. Our method out-performs PETSc at levels $3-5$**.**

(Saena), while the dashed lines show the PETSc execution times. Each matrix is shown with the same color for both our method and PETSc.

As the final plot, we compare the strong scaling at 2 levels of the multigrid (levels 1 and 4) for a case that our method was slower at the finest level. We can observe from Figure 14 that the performance and scalability of our method improves significantly at the coarser level.

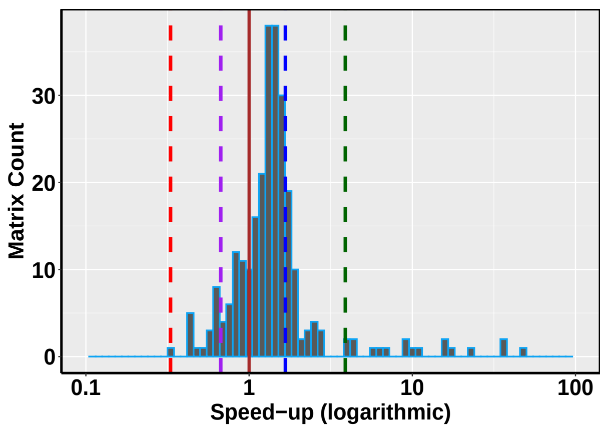


Figure 12: This plot shows the 4 matrices (the dashed lines) that we have chosen from different speed-up gain or loss range, for the strong scaling plot.

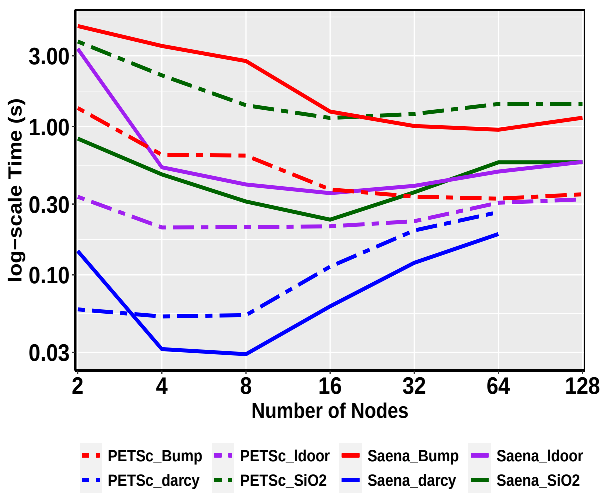


Figure 13: This plot shows the strong scaling of one GEMM time for 4 matrices. The solid lines show the time for our method (Saena), while the dashed lines are for PETSc. The two timings for the same matrix have the same color.

4 CONCLUSION

We have presented a divide and conquer approach to improve the performance and scalability 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 algorithm. Our compression method also worked very well for reducing the data

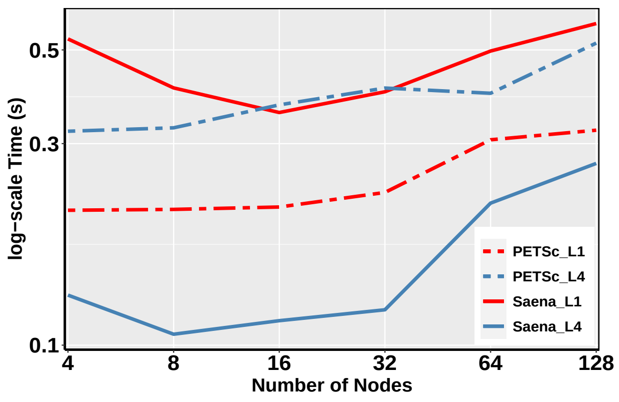


Figure 14: This plot shows the strong scaling for a matrix at levels 1 and 4. The solid lines show the timing for Saena (our method), while the dashed lines are for PETSc. Each level is shown with same color.

size and consequently the communication cost. We demonstrated performance gains from using our method and compared our multiplication with the in-built function 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.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation grant CCF-1704715 and Army Research Office W911NF1510222 (Program Manager Dr. Mike Coyle). This research used resources of the Extreme Science and Engineering Discovery Environment (XSEDE), allocation number *PHY*20033.

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] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Lisandro Dalcin, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Dave A. May, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, Stefano Zampini, Hong Zhang, and Hong Zhang. 2017. PETSc Web page. http://www.mcs.anl.gov/petsc. (2017). http://www.mcs.anl.gov/petsc

[3] A. Buluç 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

[4] Aydin Buluç and John R Gilbert. 2011. The Combinatorial BLAS: design, implementation, and applications. *The International Journal of High Performance Computing Applications* 25, 4 (2011), 496–509. https://doi.org/10.1177/1094342011403516

[5] Timothy A. Davis and Yifan Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1, Article 1 (Dec. 2011), 25 pages. https://doi.org/10.1145/2049662.2049663

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

[7] 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.

[8] F. Gremse, A. Höfter, 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

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

[10] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. 2013. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Software* 39, 2 (2013), 13:1–13:24. https://doi.org/10.1145/2427023.2427030

[11] M. Rasouli, V. Zala, R. M. Kirby, and H. Sundar. 2018. Improving Performance and Scalability of Algebraic Multigrid through a Specialized MATVEC. In *2018 IEEE High Performance extreme Computing Conference (HPEC).* 1–7. https://doi.org/10.1109/HPEC.2018.8547580

[12] R. Rice and J. Plaunt. 1971. Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data. *IEEE Transactions on Communication Technology* 19, 6 (December 1971), 889–897. https://doi.org/10.1109/TCOM.1971.1090789

[13] 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.

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

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

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

[17] 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.

[18] 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