A Divide and Conquer Matrix-Matrix Multiplication Algorithm to Speedup Algebraic Multigrid Setup

Majid Rasouli[1,2], Robert M. Kirby[1,3], and Hari Sundar[1,4]

[1] School of Computing, University of Utah,
Salt Lake City, UT, USA
[2] rasouli@cs.utah.edu
[3] kirby@cs.utah.edu
[4] hari@cs.utah.edu

Abstract. Algebraic Multigrid (AMG) is a popular solver and preconditioner for large sparse linear systems, especially the ones obtained from the discretization of elliptic operators. A key bottleneck for the scalability of the setup phase of AMG is the Matrix-matrix multiplication (MatMult) required for computing the sparse grid representation. While significant work has been done on optimizing matrix-matrix multiplication, in this work, we optimize it in the context of the matrix multiplications encountered in the setup phase of AMG. Specifically, we designed a new divide and conquer sparse MatMult, that is scalable and makes use of the specific structure of the restriction and prolongation matrices. This combined with an efficient communication pattern during parallel MatMult provides excellent performance for our AMG setup. We compare our solver with PETSc to demonstrate the performance improvements gained using our methods.

Keywords: Matrix-Matrix Product · Algebraic Multigrid · AMG · Iterative Solver · Sparse · Preconditioned Conjugate Gradient · Preconditioner

1 Introduction

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, 3, 4, 7]. 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) [2, 16], 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 [6, 19, 18], it does not scale as well as GMG [14]. This is primarily due to the loss of sparsity at coarser levels arising from the Galerkin approximation [17], 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.

1.1 Background

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 [1], [12], [9], [11]. For this paper, *maximal independent set* from [1], 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: $x \leftarrow direct\ solver(g[L + 1],\ x,\ b)$
3: **else**
4: $x \leftarrow Smoother(g, x,\ b,\ l)$
5: $r \leftarrow As[l] \times x - b$
6: $r_c \leftarrow Rs[l] \times r$
7: $y_c \leftarrow vcycle(g, x,\ r_c,\ l + 1)$
8: $y \leftarrow Ps[l] \times y_c$
9: $x \leftarrow x - y$
10: $x \leftarrow Smoother(g, x,\ b,\ l)$
11: **end if**

Smoothed Aggregation AMG (SA-AMG)[19] 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 [17] 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 [20] 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 [5], Buluc and Gilbert present algorithms for parallel sparse MATMULT using a two-dimensional block data distributions with serial hypersparse kernels. Gremse *et al.* [8] the authors 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 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. We then explain further modifications to the triple matrix multiplication (2) that are part of the AMG setup process.

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. We have implemented MATMULT in a recursive fashion. We split the matrices recursively in two ways: split by half based on the matrix size and based on

number of nonzeros. First we explain the algorithms performing splitting based on the matrix sizes.

The recursive function, RECURS_MATMULT, includes three cases:

1. Case 1: Stop the recursion: perform the multiplication.
2. Case 2: A is horizontal.
3. Case 3: A is vertical.

Case 1 By splitting sparse matrices recursively, we will have more and more zero rows and columns in the resulting sub-matrices. So, using row size and column size of those sub-matrices 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 NNZ_MAT_SIZE $= A\_nnz\_row \times B\_nnz\_col$ 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. Then, we preform the multiplication. We have implemented two methods for this case: dense data structure and 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.

When NNZ_MAT_SIZE gets larger, 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.

In Figure 1, 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 hashmap in 1245 cases for all 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

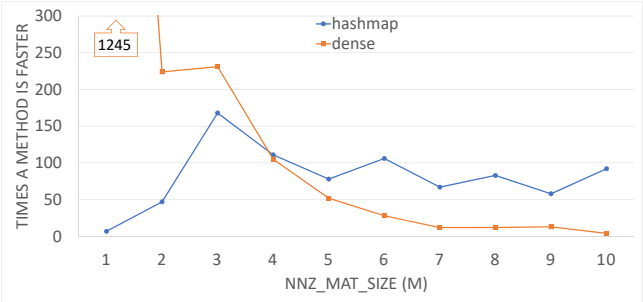

Fig. 1: Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is the 3D Poisson problem of size $216k$. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

faster. Figure 2 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 1 and 2 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.

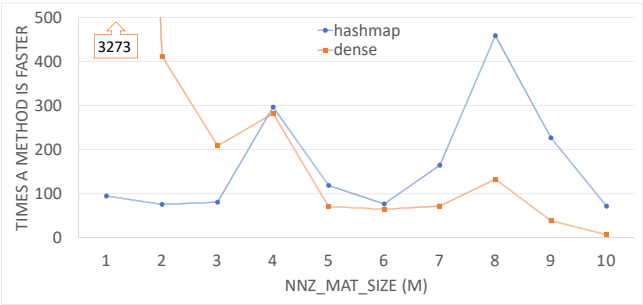

Fig. 2: Comparison between dense structure method with hashmap to compute coarse matrix $Ac = R \times A \times P$, in which A is matrix ID 1882 from SuiteSparse (Florida) Matrix Collection. The plot shows the number of times each method is faster than the other one in intervals of $1M$ for NNZ_MAT_SIZE.

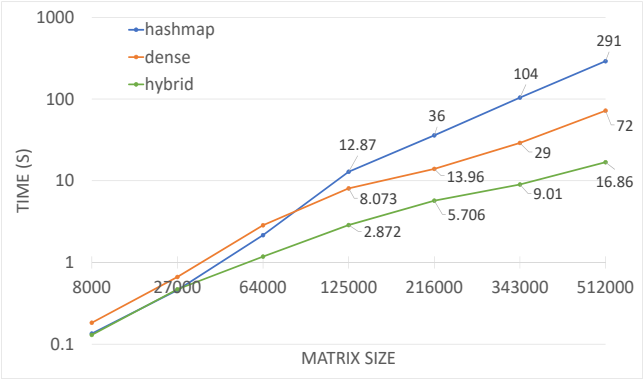

Fig. 3: Comparison between the three methods to do Case 1: only using hashmap, only using the dense structure, and the hybrid method. They are used as Case 1 to compute the first coarse matrix (the triple multiplication) on 7 matrices (3D Poisson) of different sizes.

Figure 3 compares the hybrid method with the basic two methods. We have compared the 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 $512k$, computing the first coarse matrix (performing the triple matrix) 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 for Case 1.

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 4-left). 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 will need to be added together at the end. It means, there will be entries for the result matrix with the same index, e.g. multiple entries with $(10, 20)$ as the index. We call these entries *duplicates*. Since there will be numerous nested recursive calls, we avoid doing adding duplicate 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.

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 4-right). 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 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

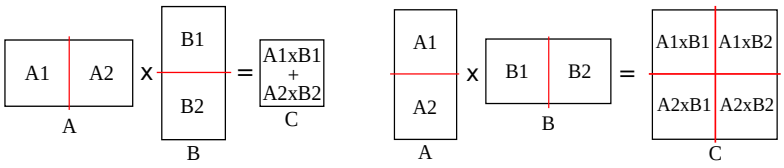

Fig. 4: Left figure shows Case 2: When A is horizontal, split A by column and B by row. Call the recursive function twice. Right figure shows Case 3: When A is vertical, split A by row and B by column. Call the recursive function four times.

Algorithm 2 Case 2: $C = \text{RECURS\_MATMULT2}(A, B)$

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

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: $C = \text{RECURS\_MATMULT3}(A, B)$

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

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.

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 AMG Matrix-Multiplication: Communication

As noted earlier, to compute the coarse matrix Ac, a triple matrix multiplication is performed:

$$Ac = R \times A \times P \tag{3}$$

with $R = P^T$ in the Galerkin approach, which we use in our implementation. In this section, we explain how the communication is setup to compute the coarse matrix Ac, exploiting properties of the matrix structure specific to AMG. Matrices are partitioned across multiple processors by row blocks (Figure 5). Matrices A and P have the same number of rows and consequently are partitioned the same way. R has fewer number of rows and has a different partition. This is because coarsening need not be uniform across processes. Consequently the partitioning of the rows of R could be different from that of A and P. The triple matrix product is performed in two parts: first $A \times P$ is computed, followed by $R \times B$, where $B = A \times P$ is the result of the first multiplication. In other words, this is equivalent to performing matrix-matrix multiplications (MATMULT) twice.

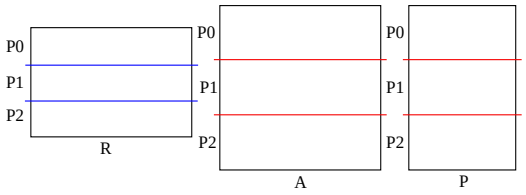

Fig. 5: Partitioning of the matrices across the processors in row blocks. A and P have the same partition but R's is different.

Part 1 In this part, we explain how to compute $B := A \times P$. We assume the same partition of rows of A on also its columns (red lines) since A is square. For columns of P we use the partition of rows of R (blue lines in Figure 6-left). Without loss of generality, let us focus on how to perform MATMULT on processor $P1$. To compute entry $B(i, j)$, we need to multiply row i of A with column j of P and add them together (since we are working with sparse matrices, we only consider the nonzeros here).

Column j of P is distributed between all the processors. Therefore we need to communicate all nonzeros of that column and then perform $B_{ij} = \sum_k A_{ik} P_{kj}$. Since that can lead to significant communication, we make use of the fact that R is the transpose of P and is already available locally because of the multigrid

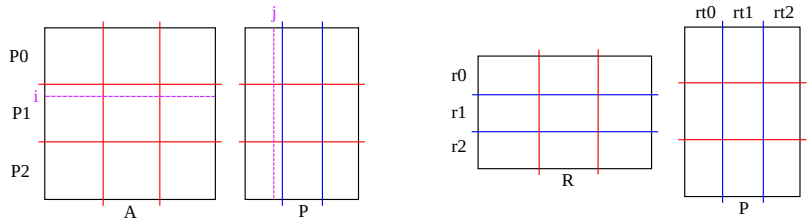

Fig. 6: The left figure shows how the matrices are split into sub-blocks in Part 1. Column j of P is stored on different processors. The right figure shows how row blocks of R are transpose of column blocks of P.

hierarchy. We note that column blocks of P (e.g. $r0$ in Figure 6) are actually row blocks of R transposed ($rt0$, which is transpose of $r0$).

We have implemented this part in an overlapped fashion; first we do the send and receive commands (to communicate rt blocks) and while this communication is being done, we perform MATMULT, so saving time for the communication (Algorithm 4). B_i in the algorithm is the row block of matrix B on processor i and B_{ik} is the sub-block result of multiplying A_i with rt_k.

Algorithm 4 Part 1: $B_i = A_i \times P$

Input: A_i, R
Output: B_i (result of $A_i \times P$)
1: $R1 \leftarrow$ transpose of R_i (locally)
2: **for** $k = myrank : myrank + nprocs$ **do**
3: $R2 \leftarrow Irecv(transpose\ of\ Rblock)\ from\ right\ neighbor$
4: $Isend(R1)\ to\ left\ neighbor$
5: $B_{ik} \leftarrow$ RECURS_MATMULT$(A_i, R1)$
6: $wait\ for\ Isend\ and\ Irecv\ to\ finish$
7: $swap(R1, R2)$
8: **end for**
9: locally sort B_i and add duplicates

Part 2 Now we explain how to do the second MATMULT: $R \times B$. The blocks of R on processor $P1$ in, Figure 7 (left), should be multiplied with the corresponding blocks of B with the same color. In contrary to the previous part, we already don't have the transpose of the right-hand side matrix and performing the transpose in parallel is expensive. Instead, we change the way the multiplication is being done and perform a cheaper communication round at the end.

In this case we have the transpose of the left-hand side matrix, so we use P instead of R to do this multiplication. We only perform the multiplication of the green blocks on processor $P1$ in Figure 7 (right), which can be done locally. In the previous case each sub-block of the local left-hand side is multiplied by only

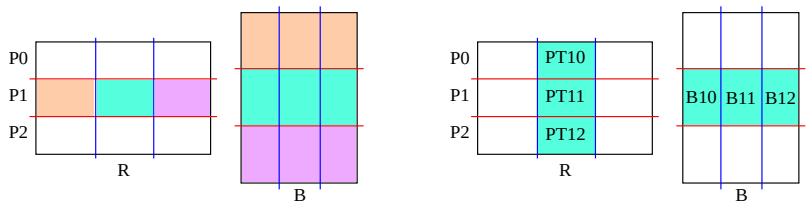

Fig. 7: Left: Part 2: $R \times B$. This figure shows which sub-blocks of R should be multiplied by which sub-blocks of B. Right: Local transpose of blocks of P is used instead of R. Each green sub-block of the transpose of P is multiplied by all green sub-blocks of B.

one sub-block of the local right-hand side. But in this method, each sub-block of local P is being multiplied by the entire row block of B, e.g. $PT10$ is multiplied by all $B10$, $B11$ and $B12$, and so on.

The difference for final result between this case and the naïve Case 2 is that, the final entries are not on the correct processors, so an additional data exchange needs to be performed at the end to restore the entries on the correct processes. We sort the entries locally at the end and add the duplicates together. Finally, we sort them globally (using HykSort [15]) and again add the duplicates together (Algorithm 5).

Algorithm 5 Part 2: $Ac = R \times B$

Input: P_i, B_i
Output: Ac_i
1: $PT_i \leftarrow$ transpose of P_i (locally)
2: **for** $k = 0 : nprocs$ **do**
3: $\quad Ac_i \leftarrow$ RECURS_MATMULT(PT_{ik}, B_i)
4: **end for**
5: locally sort Ac_i and add duplicates
6: globally sort Ac_i and add duplicates

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 8 (left) 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 8 (right) 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.

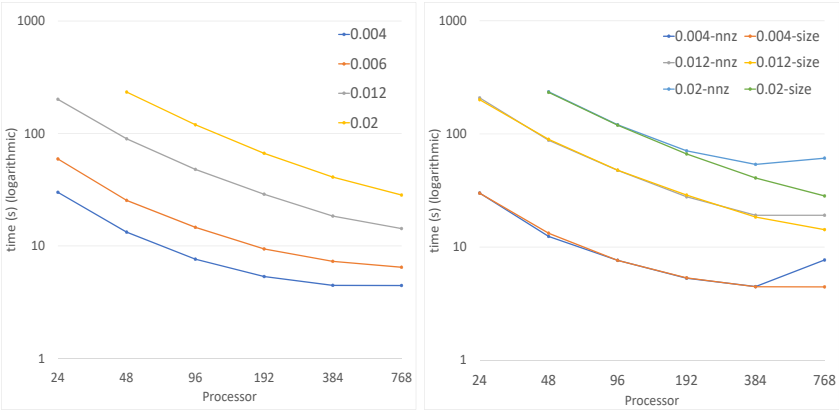

Fig. 8: Left: Strong scaling for four banded matrices of the same size ($192k$), but with different bandwidth. The legend shows the density ($\frac{nonzero}{size^2}$) of each matrix. Right: Comparison of the the strong scaling when using two different splitting strategies: based on matrix size and based on number of nonzeros. The legend shows the density of each matrix together with the splitting strategy.

Figure 9 (left) 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 MATMULTat those levels, so switching to our algorithm for lower levels of multigrid would improve the performance and scalibility significantly.

Figure 9 (right) 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.

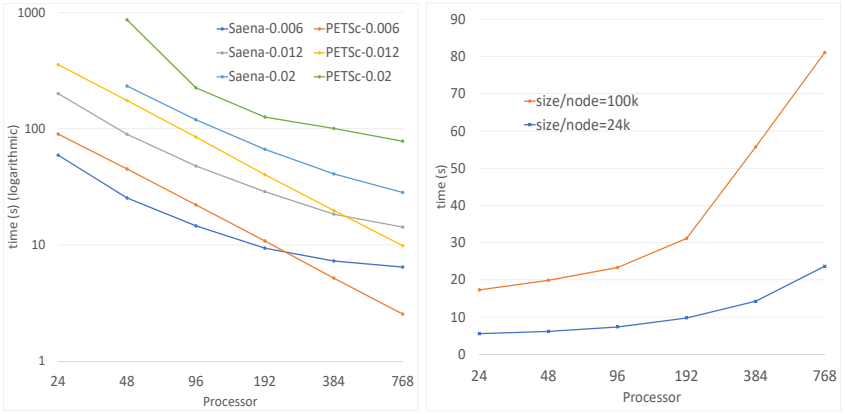

Fig. 9: Left: Comparison of the the strong scaling between our solver and PETSc. The legend shows the density of each matrix. Right: Weak scaling for two banded matrices: blue line shows the one with $24k$ on each node ($1k$ on each core) and red line shows a larger one with $100k$ on each node (4166 on each core).

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. Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM Journal on Scientific Computing **34**(4), C123–C152 (2012)
2. Braess, D.: On the combination of the multigrid method and conjugate gradients. In: Hackbusch, W., Trottenberg, U. (eds.) Multigrid Methods II. pp. 52–64. Springer–Verlag, Berlin (1986)
3. Bramble, J.H., Zhang, X.: The analysis of multigrid methods. In: Handbook of numerical analysis, Vol. VII, pp. 173–415. Handb. Numer. Anal., VII, North-Holland, Amsterdam (2000)
4. Brenner, S.C.: Smoothers, mesh dependent norms, interpolation and multigrid. Applied Numerical Mathematics **43**(1-2), 45–56 (2002), 19th Dundee Biennial Conference on Numerical Analysis (2001)
5. Buluç, A., Gilbert, J.: Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments. SIAM Journal on Scientific Computing **34**(4), C170–C191 (2012). https://doi.org/10.1137/110848244, https://doi.org/10.1137/110848244

6. Dendy, Jr., J.E.: Black box multigrid. Journal of Computational Physics **48**(3), 366–386 (1982)
7. Gholami, A., Malhotra, D., Sundar, H., Biros, G.: 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), C280–C306 (2016). https://doi.org/10.1137/15M1010798, https://doi.org/10.1137/15M1010798
8. Gremse, F., Höfter, A., Schwen, L., Kiessling, F., Naumann, U.: Gpu-accelerated sparse matrix-matrix multiplication by iterative row merging. SIAM Journal on Scientific Computing **37**(1), C54–C71 (2015). https://doi.org/10.1137/130948811, https://doi.org/10.1137/130948811
9. Guillard, H., Vanek, P.: An aggregation multigrid solver for convection-diffusion problems on unstructured meshes. Tech. rep. (1998)
10. Maday, Y., Muñoz, R.: Spectral element multigrid. II. Theoretical justification. Journal of scientific computing **3**(4), 323–353 (1988)
11. Notay, Y.: Aggregation-based algebraic multilevel preconditioning. SIAM J. Matrix Analysis Applications **27**(4), 998–1018 (2006)
12. Notay, Y.: An aggregation-based algebraic multigrid method. Electronic transactions on numerical analysis **37**(6), 123–146 (2010)
13. Saule, E., Kaya, K., Çatalyürek, Ü.V.: Performance evaluation of sparse matrix multiplication kernels on intel xeon phi. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Waśniewski, J. (eds.) Parallel Processing and Applied Mathematics. pp. 559–570. Springer Berlin Heidelberg, Berlin, Heidelberg (2014)
14. Sundar, H., Biros, G., Burstedde, C., Rudi, J., Ghattas, O., Stadler, G.: Parallel geometric-algebraic multigrid on unstructured forests of octrees. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 43:1–43:11. SC '12, IEEE Computer Society Press, Los Alamitos, CA, USA (2012), http://dl.acm.org/citation.cfm?id=2388996.2389055
15. Sundar, H., Malhotra, D., Biros, G.: Hyksort: A new variant of hypercube quicksort on distributed memory architectures. In: Proceedings of the 27th International ACM Conference on International Conference on Supercomputing. pp. 293–302. ICS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2464996.2465442, http://doi.acm.org/10.1145/2464996.2465442
16. Tatebe, O., Oyanagi, Y.: Efficient implementation of the multigrid preconditioned conjugate gradient method on distributed memory machines. In: Supercomputing'94. Proceedings. pp. 194–203. IEEE (1994)
17. Treister, E., Yavneh, I.: Non-galerkin multigrid based on sparsified smoothed aggregation. SIAM Journal on Scientific Computing **37**(1), A30–A54 (2015)
18. Vaněk, P., Brezina, M., Mandel, J., et al.: Convergence of algebraic multigrid based on smoothed aggregation. Numerische Mathematik **88**(3), 559–579 (2001)
19. Vanek, P., Mandel, J., Brezina, M.: Algebraic multigrid by smoothed aggregation for second and fourth order elliptic problems. Tech. rep., Denver, CO, USA (1995)
20. Yuster, R., Zwick, U.: Fast sparse matrix multiplication. ACM Trans. Algorithms **1**(1), 2–13 (Jul 2005). https://doi.org/10.1145/1077464.1077466, http://doi.acm.org/10.1145/1077464.1077466