Improving Performance and Scalability of Algebraic Multigrid through a Specialized MATVEC

Majid Rasouli*, Vidhi Zala†, Robert M. Kirby‡ and Hari Sundar§
School of Computing, University of Utah
Salt Lake City, Utah
Email: *rasouli@cs.utah.edu, †vidhi.zala@utah.edu, ‡kirby@cs.utah.edu, §hari@cs.utah.edu

***Abstract*—Algebraic Multigrid (AMG) is an extremely popular linear system solver and/or preconditioner approach for matrices obtained from the discretization of elliptic operators. However, its performance and scalability for large systems obtained from unstructured discretizations seem less consistent than for geometric multigrid (GMG). To a large extent, this is due to loss of sparsity at the coarser grids and the resulting increased cost and poor scalability of the matrix-vector multiplication. While there have been attempts to address this concern by designing sparsification algorithms, these affect the overall convergence. In this work, we focus on designing a specialized matrix-vector multiplication (MATVEC) that achieves high performance and scalability for a large variation in the levels of sparsity. We evaluate distributed and shared memory implementations of our MATVEC operator and demonstrate the improvements to its scalability and performance in AMG hierarchy and finally, we compare it with PETSc.**

I. Introduction

Many engineering applications use mathematical models that have as one of their building blocks the solution of elliptic partial differential equation (PDE) operators. 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 large-scale systems that need to be solved in parallel, iterative solvers with $\mathcal{O}(n)$ complexity and mesh-independent convergence are preferred.

One success story in this regard has been geometric multigrid (GMG) methods when applied to matrices generated from regular (often evenly-spaced) discretizations of elliptic operators [1], [2], [3]. 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) [4], [5], 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], [7], [8], it does not scale as well as GMG [9]. This is primarily due to the loss of sparsity at coarser levels arising from the Galerkin approximation [10], leading to poor scalability, especially at the coarser levels. Therefore, although the coarser grids (and correspondingly the discretized operators) are smaller, they take significantly longer to evaluate compared to the full-size or fine-grid problem [11]. While attempts have been made to enforce sparsity at coarser levels [10], this is usually at the cost of reduced rate of convergence. The scalability issue is primarily due to the fact that we are solving progressively denser, yet smaller matrices at the coarser levels. Standard sparse matrix libraries, especially parallel ones, are not able to ensure good performance across the range of sparsities. Indeed, this problem is specific to AMG, and we believe that AMG implementations should have specialized MATVEC implementations that are optimal over a wide range of sparsities. Analyzing the performance of MATVEC within the context of AMG and developing methods to guarantee performance and scalability across all levels is the central contribution of this work.

In §I-A, we give a brief overview of AMG and the specific choices our AMG code makes. This is followed by analysis of different strategies to improve the performance and scalability of the MATVEC in §II. Finally, in §III, experiments for the new MATVEC at different AMG levels using all our proposed methods are presented.

A. 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 [12], [13], [14], [15]. For this paper, *2-distance maximal independent set* from [12], 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:

$$A_c = R \times A \times P$$

such that $A_c \in R^{m \times m}$. This operation is called *coarsening*. The restriction operator is used similarly.

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.

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)[7] 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 [10] is used.

II. Methods

As motivated earlier, most AMG codes typically build on existing sparse matrix codes, that are for general purpose and therefore are unable to achieve the best possible performance and scalability in the multi-level scenario typical of multigrid. In this section, we identify four areas where generic sparse matrix-vector multiplication (MATVEC) can be improved in the context of AMG, and propose strategies to address these. Although, several of these strategies will likely help for generic MATVEC as well, the choices are tuned for AMG, so the benefits might be less pronounced. Additionally, some of these strategies have been explored by other researchers, but their effective application to AMG codes is new and addresses a major scalability and performance bottleneck for most AMG codes [11].

We start by describing how our matrices and vectors are partitioned across the processes, as all subsequent strategies depend on this. Matrix A is partitioned row-wise across processes (MPI tasks), as are vectors v and w in the basic MATVEC, $w = A \times v$.

Consider a generic MATVEC implementation using this structure. Since we consider a row-wise partitioning of A, the diagonal blocks and the corresponding entries of v are stored on the same processor. For non-diagonal blocks, some entries of v corresponding to rows owned by other processes need to be communicated to perform the MATVEC. The expectation is that if A is sufficiently sparse, then the number of remote entries of v communicated are small.

The distributed MATVEC consists of three parts:

1) local loop: multiplication of the diagonal blocks with the local entries of vector v
2) communication: sending and receiving the vector elements required for the multiplication with off-diagonal blocks of the matrix
3) remote loop: multiplication of the off-diagonal blocks with the received entries of vector v

Now, several performance optimizations are possible for these steps, most importantly it is possible to overlap step 2 with step 1. But our focus in this work is to improve the performance and scalability in the context of AMG, and for this it is important to identify two characteristics of the MATVEC applied to AMG. Firstly, the multi-level nature of AMG implies that we are performing the MATVEC at multiple scales, similar to a strong scaling experiment. It is important that the MATVEC is capable of performing in an extreme strong-scaling sense. Secondly, the coarse matrices, $A_c = RAP$, obtained via Galerkin approximation, incur fill-in–effectively reducing the sparsity of coarser matrices. The combination of these two effects is the main bottleneck for large-scale AMG codes. Since, these issues are very specific to AMG and not sparse-matrices in general, we focus only on the AMG-MATVEC and not on the generic MATVEC. Additionally, while there are approaches to *sparsify* the coarse matrices [10], these do affect the overall convergence and the focus

of this work is to improve the performance without making any algorithmic changes.

We now describe the four improvements we propose to the AMG-MATVEC. For each of these improvements, we will demonstrate the effects using matrices from the SuiteSparse Matrix Collection [16], focusing on only the MATVEC. A set of matrices was selected based on their sparsity and size. Several of the results presented in this section use matrix ID 1883. These matrices were specifically selected to highlight the issues faced during large-scale scaling.

The experiments for both this section and Section III are done on Comet. It is a dedicated eXtreme Science and Engineering Discovery Environment (XSEDE) cluster designed by Dell and SDSC (San Diego Supercomputer Center at UC San Diego) delivering 2.76 peak petaflops. The standard compute nodes consist of Intel Xeon E5-2680v3 processors and 128 GB DDR4 DRAM (64 GB per socket).

A. Process Shrinking

In Figure 1, we present the time for MATVEC for different levels of multigrid. Level 0 is the full-size matrix A, and the other levels are obtained via Galerkin approximation as described in §I-A. In this case, the communication of remote values of v is overlapped with the local MATVEC computation.

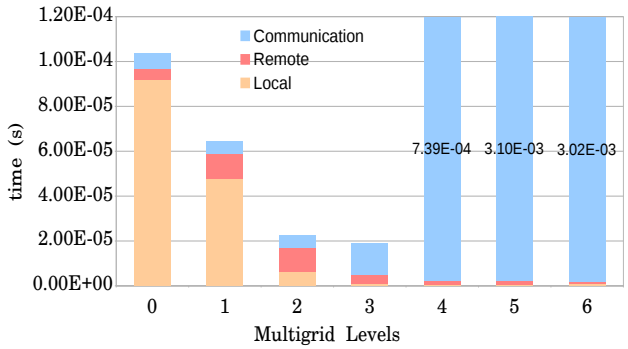

Fig. 1. Average time of one MATVEC for 6 levels of multigrid for matrix ID 1883, on 4 nodes, 96 MPI tasks. The values on levels 4 and 5 bars are the communication time.

We can see that the communication costs rise rapidly after level 2, to the effect that these end up being more expensive than level 0. This is primarily due to increased fill-in at the coarser levels thereby requiring increased data-exchange, as well as an increasing number of processes to exchange data with. In addition, at the coarser levels, the time for the local-loop goes down (strong-scaling), making it harder to efficiently overlap communication with the local-loop. The key idea here is that performing a small MATVEC on a very large number of processes is inefficient, so it is preferable to use only a small subset of processes to perform the MATVEC at the coarser levels, with the number of active processes dependent on the size of the coarser grid. We call this *process shrinking*. Note that it is important to not be over-aggressive with process shrinking, as this could increase the processing time for the local and remote loops. The improvement as a result of process shrinking is shown in Figure 2. Shrinking was applied at levels $3, 4$ and 5.

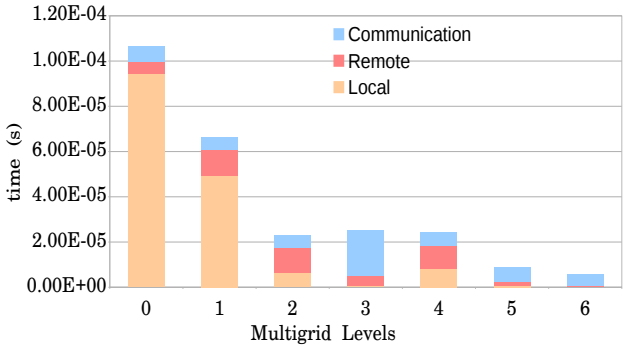

Fig. 2. Average time of one MATVEC for 6 levels of multigrid for matrix ID 1883, after process shrinking at level 4, by a factor 16. (on 4 nodes, 96 cores, 96 MPI tasks,1 `OpenMP` thread)

The process shrinking is done by reducing the number of processes by a factor, let's call it a. If there are p processes at the current level, the ones with rank ak, in which $k = 0, 1, ..., \lfloor \frac{p}{a} \rfloor$, become *roots*. Processes with rank $ak + 1, ak + 2, ..., ak + (a - 1)$ will send their data to rank ak processes, which are their roots.

To decide when the process shrinking should happen, some factors, including some machine-dependent ones, play part. We implement a dummy MATVEC for each multigrid level and evaluate its performance. Based on this approximation of total time, computation time or communication time of the current level comparing with the previous level, the decision is being made about shrinking. If the total time for the coarse MATVEC increases compared to the current level, as a result of increased communication costs, then we enforce process shrinking.

B. Optimizing local and remote Loops

As a result of process shrinking, we observe that the local and remote loops dominate the cost, especially the local loop (see Figure 2). In this section, we propose methods to optimize the local and remote loops. Note that the local and remote loops are *node* local, so this stage only involves shared-memory parallelism, i.e., `OpenMP` in our case. The AMG specific aspect here is that the sparsity–especially of the off-diagonal blocks–changes as we get coarser. We implemented four variations for performing the local loop[1] of MATVEC and evaluated their performance across different AMG levels.

While evaluating $w = Av$, there are two choices regarding the traversal depending on whether multiple passes are made over v or w. If we *read* from v multiple times, we only *write* to w once, and if we *read* from v once, we *write* to w multiple times. The entries of A need to be arranged depending on which traversal we choose. We consider 4 variations ($S_1 - S_4$) on implementing the shared-memory MATVEC depending on this choice.

[1]the local and remote part are similar in structure, except for the sparsity

1) S_1: Column-Major: Our first implementation uses the column-major ordering for A, i.e., we perform a single read on v and multiple writes to w. We start from the first nonzero column of the matrix (say column j) and multiply all entries of that column by $v[j]$, and add to the corresponding entries of w. The loop is parallelized over the entries of v using `OpenMP` with a `reduction` at the end to add local copies of w together (see figure 3).

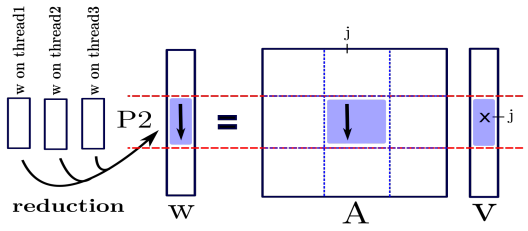

Fig. 3. S_1: Column-wise order is considered on the matrix. v in being iterated once. Local copies of w are computed by each thread followed by a reduction.

2) S_2: Row-Major: Our second implementation uses the entries of A stored in the row-major order. This is the obvious implementation of the MATVEC. The multiplication accumulates in $w[i]$ the product of all nonzero entries of row i of A with the corresponding entries of v, i.e., $w[i] = \sum_{j \in \mathcal{J}_i} A[i,j]v[j]$, where $\mathcal{J}_i$ are the column indices of nonzero entries in row i.

3) S_3: Staggered Row-Major: This is an optimized version of S_2. Note that if the block is dense, the probability of different threads trying to access the same entry of v at the same time is high. In figure 4 (left), three threads are shown as an example. It is possible that all three threads start from the same sub-block of the matrix. So, all of them will try to access the same part of vector v at the same time, causing memory contention. Since, this is a read-only access, the effect is likely minimal, but we still wanted to evaluate the effect. The algorithm is similar to the one in S_2, but the starting blocks for the threads are distributed. Let's say three threads are being used. We will partition the local part of vector v to three parts and the multiplication on each thread starts at its corresponding part of v. Figure 4 illustrates how this algorithms operates.

4) S_4: Column-Major with optimized reduction: Since we observed S_1 performing very poorly, we traced it to the

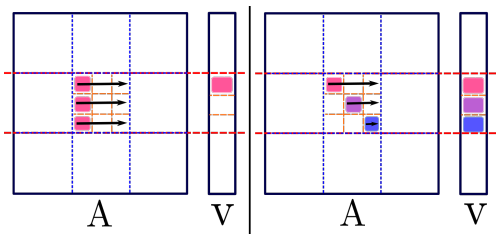

Fig. 4. S_3: The left figure shows all the threads may access the same part of vector v at the same time. The right figure shows changing the starting entry on the threads can reduce the possibility of the access competition.

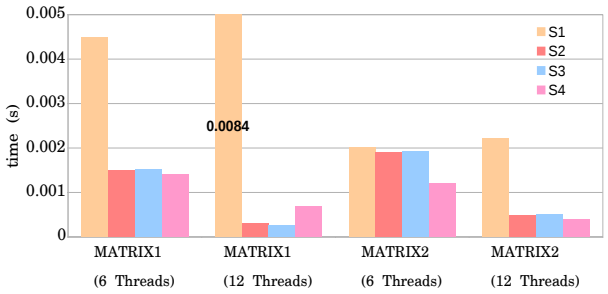

Fig. 5. Comparison between the 4 methods for local part of MATVEC from Section II-B. It is done on 384 cores, 32 MPI tasks, 6 and 12 `OpenMP` threads

vector `reduction`[2], and implemented a parallel reduction ourselves, not relying on the `OpenMP` reduction. This simply adds two intermediate vectors at a time in a binary-tree fashion to perform the vector reduction.

5) Comparing the four variants: Four experiments are done to compare these methods. The matrices are chosen from SuiteSparse Matrix Collection: Matrix1 and Matrix2 IDs are 1883 and 1364, respectively.

Based on our experiments, method S_3 is sometimes slightly better than S_2, but most of the times they are close to each other, so since S_2 is simpler we prefer this one. S_1 is usually the worst implementation. For lower number of threads method S_4 is usually the best one. Higher number of threads helps methods S_2 and S_3 more than the other ones.

The local loop has been discussed in this subsection. To improve the remote loop, the S_4 method is used in our solver. The reason is that to perform MATVEC, whole column j of the matrix should be multiplied by entry j of the vector. If that entry needs to be communicated to this processor, it is more efficient to communicate it once for the whole vector, so the column-major order is the better choice for the remote loop (S_1 and S_4) and S_4 is the improved version of S_1.

C. Load-balance & Communication-cost trade-off

Our implementation like most sparse-matrix implementations, partitions the matrix A across processes by keeping the number of non-zeros (nnz) roughly equal[3] across the partitions. At deeper levels of the AMG hierarchy, as the nnz per line increases, this partitioning scheme results in having a large difference in the number of rows on different processors. The number of vector elements stored on each processor is equivalent to the number of rows of the matrix, which means that the communication costs for some processors increases significantly.

Given that the cost of data movement across the network (inter-node data movement) is significantly higher than computation (intra-node data movement), it would make sense to reduce the cost of communication even if the local work is increased. To this effect, we have implemented a dynamic work-balance algorithm, consisting of two schemes:

[2] $gnu/4.9.2$ is used for this experiment.
[3] we keep all entries of a row on the same process.

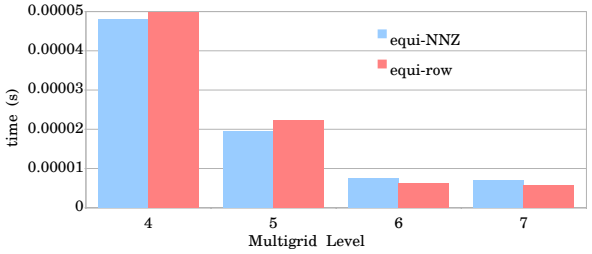

Fig. 6. Comparison of work-balance based on nonzeros with work-balance based on rows after level 4 of matrix A_{1883}. For sparser levels equi-NNZ is better, for denser ones equi-row is faster. This is done on Comet using 1728 MPI tasks.

1) work-balance based on number of nonzeros (equi-NNZ)
2) work-balance based on number of rows (equi-row)

The solver starts with a equipartition on nnzs, and automatically switches to an equipartition on the number of rows based on the density of the coarse matrices. The exact threshold at which this switch happens is empirically determined by profiling on the target machine. During the AMG setup once the coarse matrix is constructed, we determine the density and decide the appropriate partitioning scheme to employ.

Work-balance based on nnz improves the performance of MATVEC loops, and work-balance based on the number of rows reduces the communication cost. At first, the loops are usually the dominant part of MATVEC, so the partitioning is based on the number of nonzeros. At the deeper levels, the communication part becomes the bottleneck, so the solver switches to the second partitioning method. Figure 6 compares the two work-balance schemes after level 4 of the AMG hierarchy. For levels 4 and 5 as we explained the one with the same number of nonzeros works better. After reaching a denser matrix, having the same number of rows on the processes is faster. The decision when to switch from equi-nnz to equi-row is made based on the sparsity of the matrix in the multigrid hierarchy. Check Section III to see an example applying this method.

D. *Sparse to Dense* MATVEC

When the coarse matrices become sufficiently dense, all processes are sending data to all other processes. This has two problems. First, for p processes this amounts to p^2 messages being sent, and secondly each process has to store the entire vector v in memory. Both of these issues can be addressed by switching to a dense representation for the matrix and performing the MATVEC differently. The dense matrix is still partitioned row-wise, so that the partitioning remains the same for v. However, the matrix is stored in blocks corresponding to the partitioning of v. In order to perform the MATVEC we perform $p-1$ rounds of overlapped communication and computation. In the first round, each process multiplies its portion of v with the corresponding block, and accumulates the result in its portion of w. It then sends its v vector to the next process and receives a new v from the previous process, in a cyclic fashion. This continues until the MATVEC is complete after $p-1$ rounds. The communication can be easily overlapped with the local block-MATVEC as shown in Algorithm 2. The dense MATVEC addresses both our problems, as it only requires $\mathcal{O}(n/p)$ storage, where n is the length of v and it only sends $\mathcal{O}(p)$ messages.

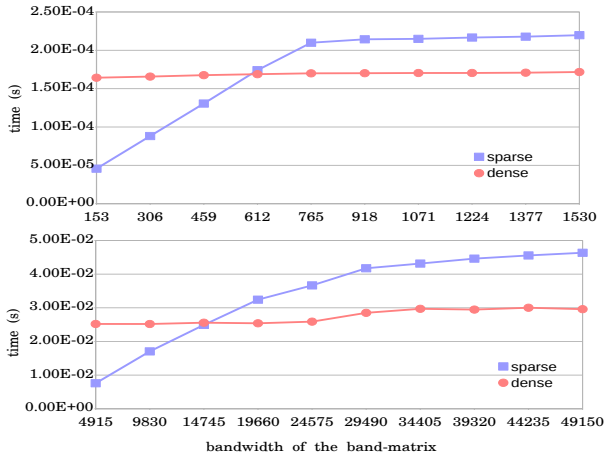

Fig. 7. Comparison of the average time for one sparse and one dense MATVEC for a band matrix with different bandwidth values. up: matrix size: 1536×1536, 1 node, 24 MPI tasks; down: matrix size: 49152×49152, 8 nodes, 192 MPI tasks

Algorithm 2 Overlapped Dense MATVEC

Input: A, v
Output: w
1: $y \leftarrow v$
2: **for** $k = myrank : myrank + nprocs$ **do**
3: $x \leftarrow$ *Irecv* v *from right neighbor*
4: *Isend*(y) *to left neighbor*
5: **for** $i \in$ *local rows* **do**
6: **for** $j \in$ *index set of owner of* v **do**
7: $w[i] \mathrel{+}= A[i,j] * y[j]$
8: **end for**
9: **end for**
10: *wait for Isend and Irecv to finish*
11: $swap(x, y)$
12: **end for**

For this part's experiments (Figure 7), a band matrix is generated and MATVEC is computed for different bandwidth values. So, the matrix size is fixed, but the sparsity of the matrix changes based on its bandwidth. The same process is applied for the same matrix, but in a dense data structure.

Switching to the dense structure happens in the final levels of the hierarchy, in which the size of the matrix is small and also not all processors are active. That is the reason the comparison between the sparse and dense MATVEC are done here on smaller matrices and for fewer number of processors. When the coarse matrices in the multigrid hierarchy pass a sparsity threshold, our solver switches to the dense MATVEC.

E. *Summary*

We choose one of four variants for performing the local MATVEC depending on the machine. We also perform process shrinking based on the approximate MATVEC that is computed

in the setup phase, to ensure good balance between computation and communication. Additionally, we switch between two different load-balancing strategies, and finally switch to a dense representation when sparsity decreases significantly, which again gets decided in the setup part.

III. Numerical Results

In this section, we present experiments and results demonstrating the efficacy of our methods. First we present the improvement in the MATVEC scalability for a typical multigrid hierarchy. We use a combination of the proposed methods based on our experiments on the target machine, and evaluate the MATVEC performance for three scenarios,

1) Applying only loop optimizations, without any shrinking or changes to load-balancing,
2) Applying loop-optimizations and switching to equi-row partitioning and dense MATVEC after level 4
3) Same as #2 above, but additionally shrink processes at levels 3 and 4.

These results are presented in Figure 8, where one can see the benefits of the proposed changes to the MATVEC on the overall performance across all levels. When the matrices in the multigrid levels pass a sparsity threshold at a specific level, load-balance will be switched from equi-NNZ to equi-row for all the levels after that. In addition to that, instead of sparse MATVEC, dense MATVEC will be used beyond that level. Also, process shrinking happens based on a MATVEC approximation at each level, considering the computation and communication costs at the current level and comparing them with the ones of the previous level.

Comparing the first and second approaches, we see that beyond level 5, it is better to use dense MATVEC. Also, we can observe that it can be detrimental to switch to dense-MATVEC too early, *e.g.* at level 4 . Finally, we can observe that combining all these methods removes the bump, and also demonstrates that these methods are successful in improving the performance and scalability significantly.

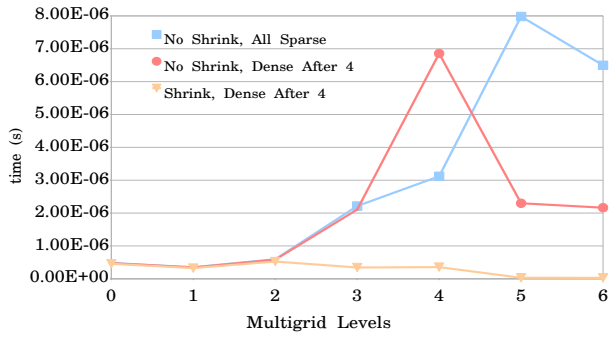

Fig. 8. Average time for one MATVEC for A_{1883} on 1728 cores, 42 nodes. This figure shows how different methods discussed here can improve the performance of MATVEC at different levels of the multigrid hierarchy.

Next, we compare our solver with PETSc (Figure 9). To avoid any bias due to the coarsening strategy used by the AMG methods, we use the multigrid hierarchy generated by our code, and pass the matrix at each level to PETSc to perform MATVEC using PETSc data structures and functions. Since PETSc does not fully support `OpenMP`, to have a fair comparison, we haven't used `OpenMP` for our solver either. Both codes operate using only MPI. While the performance of PETSc and our solver are comparable at the finer grids, the performance of PETSc deteriorates at coarser levels, whereas our optimizations ensure that our performance is stable.

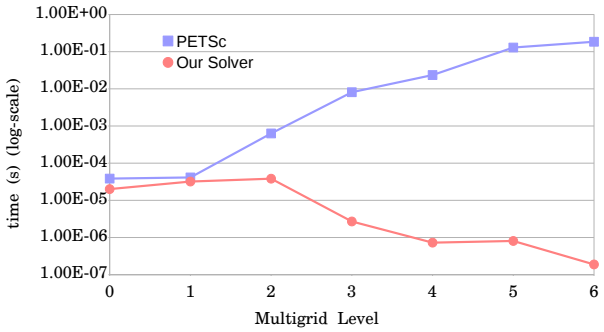

Fig. 9. Comparison between PETSc and our solver: Average time for one MATVEC for A_{1883} at different levels of the multigrid hierarchy, on 1008 MPI tasks.

Finally, Figure 10 shows the weak and strong scaling plots for solving the 3D Poisson problem.

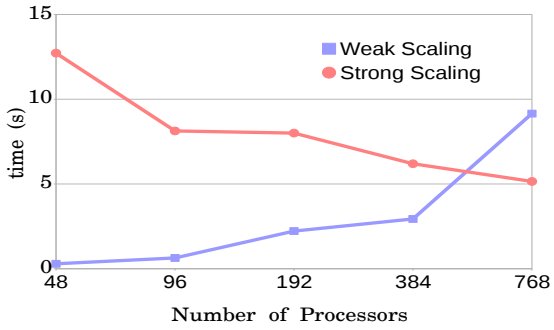

Fig. 10. The strong scaling is done on a 3D Poisson matrix with $4M$ degree of freedom and $27.4M$ nonzeros. For the weak scaling, the biggest matrix has $8M$ degree of freedom and $55M$ nonzeros.

IV. Conclusion

We presented various strategies to improve the performance and scalability of the matrix-vector product in the context of large-scale multiscale iterative algorithms like AMG. Our methods are tuned for the target architecture and the appropriate variant is determined during the setup phase. Our optimizations allow us to keep the cost of MATVEC stable for a wide range of matrix sizes and sparsities. In future work, we would like to improve the performance of our AMG code, especially by incorporating sparsification algorithms. We are also currently working on reducing the setup cost of our AMG code so that an end-to-end comparison with other AMG codes can be made.

V. Acknowledgment

This work was supported in part by the National Science Foundation grants ACI-1464244 and CCF-1643056 and Army Research Office W911NF1510222 (Program Manager Dr. Mike Coyle). This research used resources of the Extreme Science and Engineering Discovery Environment (XSEDE) allocation TG-PHY180002.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[16] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049662.2049663