High-Performance Algebraic Multigrid Solver Optimized for Multi-Core Based Distributed Parallel Systems

Jongsoo Park[1], Mikhail Smelyanskiy[1], Ulrike Meier Yang[2], Dheevatsa Mudigere[1], and Pradeep Dubey[1]

[1]Parallel Computing Lab, Intel Corporation

[2]Center for Applied Scientific Computing, Lawrence Livermore National Laboratory

ABSTRACT

Algebraic Multigrid (AMG) is a linear solver, well known for its linear computational complexity and excellent parallelization scalability. In addition its use also leads to a significantly reduced amount of global reductions, compared to Krylov-based solvers, even when used as a preconditioner to Krylov methods. As a result, AMG is expected to be a solver of choice for emerging extreme scale systems capable of delivering hundred Pflops and beyond. Taking advantage of a large amount of node level concurrency requires significant optimizations of underlying algorithms: from optimizing cache locality and reducing branching overhead, to extracting thread/SIMD parallelism. While node level performance of AMG is generally limited by memory bandwidth, achieving high bandwidth efficiency is challenging due to highly sparse irregular computation, such as triple sparse matrix products, sparse-matrix dense-vector multiplications, independent set coarsening algorithms, and smoo-thers such as Gauss-Seidel. We develop and analyze a highly optimized AMG implementation, based on the well-known HYPRE library. Compared to the HYPRE baseline implementation, our optimized implementation achieves 2.0× speedup on a recent Intel Haswell Xeon processor. Combined with our other multi-node optimizations, this translates into up to 3.7× speedups when weak-scaled to a 64-node system. In addition, our implementation achieves 1.3× speedup compared to AmgX, NVIDIA's high-performance implementation of AMG, running on K40c.

1. INTRODUCTION

Unprecedented growth of the compute capability of high performance systems in the last few decades has pushed the envelope of the most challenging scientific problems, from quantum chemistry, to computational finance, to more recently, big data analytics. Solving large sparse linear systems of equations forms the backbone of many scientific problems and takes a significant portion of the run time. Thus, it is important to use highly scalable algorithms to fully harness the increasing capability of computing systems.

Many linear solver algorithms, such as conjugate gradient or GMRES [1], exhibit poor weak or strong scaling, because the number of iterations to reach the same level of accuracy increases with larger problems and because of inherent global synchronization, such as all-reduce. Alternatively, multigrid solvers are well known for their linear computational complexity[1] and excellent scalability. As a result, such solvers are well suited for the emerging extreme scale systems which can deliver 100+ Pflops of performance [2].

There are two types of multigrid solvers. *Geometric multigrid* uses the grid of a problem, whereas *algebraic multigrid* is applied directly to the linear system matrix. As such, AMG is attractive because it approaches the asymptotic complexity and scalability of geometric multigrid, while enabling the solution of more unstructured problems.

There has been a large body of work on parallelizing the original AMG method [3]. One of the earlier parallel implementations is BoomerAMG [4], an unstructured multigrid solver in the HYPRE library. A lot of effort has been put into improving its scalability, with regard to the time per iteration as well as the number of iterations by improving coarsening algorithms and interpolation operators [5–7].

BoomerAMG was initially designed for distributed-memory architectures, and later extended to hybrid MPI-OpenMP parallelism [8]. Still, AMG solvers need to adapt to the trend that a large portion of the concurrency occurs within each chip. There is a CUDA implementation of AMG called AmgX that has been optimized for NVIDIA GPUs [9]. In [9], results are shown which state that AmgX is on average a couple of times faster than HYPRE running on multi-core Xeon processors. This result is partly understandable considering that the performance of AMG is memory bandwidth bound and an NVIDIA GPU typically has higher memory bandwidth.

However, our paper shows that once optimized for modern x86 multi-core processors, HYPRE AMG running on a Xeon processor can outperform AmgX running on an NVIDIA GPU. We exemplify that not only the raw memory bandwidth provided by hardware but also its efficient utilization is important. We showcase a series of optimizations that can also be applied to other sparse matrix applications. A large fraction of the applied optimizations target cache locality, thus also improving effective bandwidth utilization.

Specifically, this paper makes the following contributions.

- We present an AMG solver implementation highly optimized for modern x86 multi-core processors that can benefit many applications that use AMG. Because our implementation is based on the widely-used HYPRE library, it can also benefit its user base.
- We document optimizations that can be useful for other sparse solver libraries such as Trilinos [10], PETSc [11], and future

[1]In multigrid both the number of iterations to convergence and the time per iteration step are constant or near-constant as the problem size increases.

SC '15, November 15-20, 2015, Austin, TX, USA

DOI: http://dx.doi.org/10.1145/2807591.2807603

sparse matrix applications/libraries that aim to optimize for modern multi-core processors. Examples include reordering of matrix rows and columns that optimize cache locality and branching overhead, and an efficient assembly of matrix rows received from other MPI ranks in matrix-matrix operations.

- We compare the performance of our optimized implementation with AmgX and the baseline HYPRE. Our optimized BoomerAMG running on one socket of 14-core Xeon E5-2697 v3 at 2.6 GHz outperforms AmgX running on K40c by 1.3× despite of the gap in memory bandwidth (54 GB/s vs. 249 GB/s STREAM bandwidth [12]). Compared to the baseline HYPRE, our optimized implementation improves the single-node performance by a factor of 2.

- We evaluate our optimized implementation up to 128 nodes in both weak and strong scenarios. We observe that setup phase has worse scalability than solve phase. We also show that our optimizations reduce performance gap between the currently popular multipass interpolation and more numerically robust 2-stage extended+i interpolations.

The rest of this paper is organized as follows. §2 overviews AMG algorithm and related work. §3 and §4 present a series of optimizations, first the ones for multi-core processors, and then those for multi-node scaling. §5 quantifies the impact of optimizations and compares the performance of our implementation with that of the baseline HYPRE and AmgX. §6 concludes and discusses future work.

2. ALGEBRAIC MULTIGRID AND RELATED WORK

*Multigrid method*s are effective scalable solvers and well suited for high performance computers, since, when properly designed, they can solve a linear system with n unknowns in $O(n)$ computations. They achieve this optimality by eliminating errors that cannot be removed with a few smoothing steps in the current grid resolution via coarse-grid correction on successively coarser grids. *Algebraic multigrid* (AMG), differs from geometric multigrid in that it does not use the actual grid, but instead is applied directly to the linear system $Ax = b$, enabling it to also solve unstructured problems.

Algebraic multigrid consists of a *setup phase* and a *solve phase*. In the setup phase, the *coarse grid variables*, *interpolation operators* P_l, restriction operators R_l (often $R_l = P_l^T$), and the *coarse grid matrices* $A_{l+1} = R_l A_l P_l$ are determined for $l = 0, 1, ..., m$ levels, where $A_0 = A$. During the solve phase, one or two steps of a *smoother*, i.e., a generally simple iterative method such as Jacobi or Gauss-Seidel, are applied at each grid level. The improved guess is then restricted to the next coarser level until the coarsest level is reached, which can be solved with a direct method or approximated with a few smoothing steps. The solution or approximation of the coarse grid solve is then interpolated back up, level by level, to the finest level, applying smoothing again on each level. The complete cycle, which is called a "V-cycle", is then repeated until the desired convergence tolerance is reached. There are two important measures that determine the quality of an AMG algorithm. The first is the *convergence factor*, which indicates how fast the method converges. The second is the *operator complexity*, which affects the number of operations and the memory usage. Operator complexity is defined as the sum of the number of non-zeros of A_l over $l = 0, ..., m$ divided by the number of non-zeros of A. An AMG solver can be scalable (i.e., $O(n)$ computations for n unknowns) when the number of iterations to converge is $O(1)$ and the operator complexity is $O(n)$.

We define a few notations here that will be used in the subsequent sections. Point j is a neighbor of i if and only if there is a non-zero a_{ij}. Point j strongly influences i if and only if $-a_{ij} \geq \alpha \max_{k \neq i}(-a_{ik})$, where α is the strength threshold. This strong influence relation is used to select coarse points. The selected coarse points are retained in the next coarser level, and the remaining fine points are dropped. Let C_l and F_l be the coarse and fine points selected at level l, and let n_l be the number of grid points at level l ($n_0 = n$). Then, $n_l = |C_l| + |F_l|, n_{l+1} = |C_l|$, A_l is a $n_l \times n_l$ matrix, and P_l is a $n_l \times n_{l+1}$ matrix.

There has been a lot of research on variants of AMG since the development of the first AMG method in the 80s [3]. A detailed summary can be found in [13]. One of the issues of the original classical AMG method is that — while it converges fast — it often generates excessive operator complexities, especially for three-dimensional problems. This problem is exacerbated for parallel implementations of AMG. Consequently, efforts were made to coarsen more aggressively to reduce operator complexities, e.g. [6]. More *aggressive coarsening* leads to often considerably reduced convergence, since it violates conditions required for classical interpolation. Convergence can be improved again by combining more aggressive coarsening with *long distance interpolation* [14].

Alternatively, *aggregation* AMG coarsens by aggregating points to obtain the points used on the next level [15, 16] rather than splitting into coarse and fine points as in *classical* AMG. Aggregation AMG typically leads to faster setup and lower operator complexity, but often at the expense of a sub-optimal asymptotic convergence rate [13]. The reduced convergence can be compensated by a method called *smoothed aggregation* AMG [17], which often leads to a high operator complexity [18]. The convergence of *unsmoothed aggregation* AMG can also be improved using K-cycles [19]. Because a K-cycle is more expensive than a V-cycle, this approach adds complexity to the solve phase.

There has been a lot of research on GPU implementations of AMG and comparisons with CPU-based implementations [9, 18, 20–22]. Unsmoothed aggregation AMG has been particularly popular for GPUS due to its typically lower operator complexity that suits limited GDDR memory capacity [9, 18, 22]. NVIDIA AMGX in particular uses unsmoothed aggregation AMG without K-cycles. This approach often converges slower than classical AMG. The focus of this paper is not comparing different variants of AMG algorithms, but a fair comparison of HYPRE running on a CPU with AmgX running on a GPU using as similar settings as possible. Therefore, our comparison uses classical AMG for both libraries.

In the experiments presented here, PMIS coarsening is used, due to its high parallelism and since it is also used in AmgX's version of classical AMG. It is combined with extended+i interpolation, since this often leads to better convergence than the other distance-two interpolation operators [7]. As a smoother, we mainly use hybrid Gauss-Seidel, i.e. Gauss-Seidel within a task, but Jacobi across parallel tasks, since this generally leads to better convergence than the completely parallel Jacobi smoother, but still provides sufficient parallelism compared to lexicographical Gauss-Seidel smoothing. Multi-color or block multi-color Gauss-Seidel [23] is another smoother that provides high parallelism, which has been particularly popular for GPUs [24] and implemented in AmgX [25]. For a more detailed discussion on parallel smoothers, see [26].

3. OPTIMIZATIONS FOR MULTI-CORE PROCESSORS

While there has been a large body of research on achieving $O(n)$ algorithmic scalability of AMG, its scalable parallel implementation is critical in practice for two reasons. First, one often wants to reduce time to solution, and second, solving a large problem often requires the memory of more than a few compute nodes. The optimized parallel implementation of AMG poses unique challenges, since AMG, compared to other solvers, consists of a diverse set of subroutines and irregular, unstructured computation. Addressing these parallelization challenges both for multi-core and multi-node architectures is the focus of this paper. This section presents our parallel optimizations for multi-core processors.

3.1 Setup Phase

In the setup phase, we focus on the two most time consuming steps: the triple sparse matrix product used for construction of the coarse grid operator, as well as the construction of the interpolation operator.

3.1.1 Triple Sparse Matrix Product

We construct the grid operator at level $l+1$, A_{l+1}, by $R_l \cdot A_l \cdot P_l$, where R_l is the restriction operator, A_l is the fine grid operator, and P_l is the interpolation operator, respectively at level l. This triple sparse matrix product is also called Galerkin coarse grid operator or RAP product. In most cases, $R = P^T$, and thus we typically compute $P_l^T A_l P_l$.

Building Block SpGEMM: The building block of this triple sparse matrix product is the sparse matrix-matrix multiplication (SpGEMM). A classical SpGEMM algorithm is proposed by Gustavson [27], and its optimized implementation on an x86 architecture is recently described by Patwary et al. [28]. Our implementation is also based on [28].

Among the improvements upon the original Gustavson's algorithm, the one that has the biggest impact in the context of AMG is reading the input matrix only once rather than twice. Specifically, one obstacle to efficient SpGEMM is that the size of the output matrix is unknown a priori. Therefore, traditional implementations of SpGEMM inspect input matrices in a preprocessing step to count the number of non-zeros of each output matrix row. Then, memory is allocated for the output matrix, and each thread is set to the memory location where it can start populating its portion of the multiplication result. This is followed up by the actual multiplication step where the input matrices are read again. In contrast, our implementation pre-allocates a large enough chunk of memory to each thread, which then stores the multiplication results to its assigned chunk. When all threads are finished, we copy the disjoint memory chunks from each thread to a contiguous region of memory allocated for the final result. This approach eliminates one read of row pointers and column indices in the input matrices at the expense of an additional copy of the output matrix. This is beneficial because reading the second input matrix involves expensive non-contiguous accesses, while copying results into the output matrix is contiguous. Furthermore, in AMG, the output matrix A_{l+1} is typically a couple of times smaller than A_l; thus saving one input matrix read more than offsets the cost of of copying output matrix. Note that this optimization relies on efficient virtual memory management in modern operating systems that allow pre-allocating a large chunk of memory without significant overhead and often lazily bind a physical page to a virtual page only when it is actually accessed.

SpGEMM is not the only sparse matrix operation in AMG where the size of the output matrix is unknown a priori. Other routines in the setup phase that require the determination of the size of the output matrix are the construction of the strength matrix and the

for each row i in matrix R **do**
  $\vec{B}_i \leftarrow \vec{0}, \vec{C}_i \leftarrow \vec{0}$
  for each non-zero r_{ij} in $\vec{R}_i$ **do**
    for each non-zero a_{jk} in $\vec{A}_j$ **do**
      b_{ik} += $r_{ij} \cdot a_{jk}$
  for each non-zero b_{ij} in $\vec{B}_i$ **do**
    for each non-zero p_{jk} in $\vec{P}_j$ **do**
      c_{ik} += $b_{ij} \cdot p_{jk}$

(a) Our optimized implementation

for each row i in matrix R **do**
  $\vec{C}_i \leftarrow \vec{0}$
  for each non-zero r_{ij} in $\vec{R}_i$ **do**
    for each non-zero a_{jk} in $\vec{A}_j$ **do**
      temp $\leftarrow r_{ij} \cdot a_{jk}$
      for each non-zero p_{kl} in $\vec{P}_k$ **do**
        c_{ik} += temp$\cdot p_{kl}$

(b) Alternative way of fusion in the baseline HYPRE

Figure 1: Pseudo code of triple matrix product $R \cdot A \cdot P$ with fused sparse matrix-matrix multiplications

interpolation operators, where we apply similar optimizations.

SpGEMM Fusion: Using this efficient SpGEMM implementation as a building block, we further optimize the triple sparse matrix product by fusing the two SpGEMM operations together to improve cache reuse. A straightforward way of computing RAP is first finishing $B = RA$ then starting $C = BP$. Instead, immediately after computing B_i, the ith row of the temporary matrix B, we compute C_i. In this way, when computing C_i, we are likely to access row B_i out of cache, as opposed to streaming it from memory, as in the unfused original implementation. The pseudo code is shown in Fig. 1(a). In this code, we denote matrix rows as if they are dense vectors for illustration purposes, while, in reality, they are sparse vectors. The accumulation to a sparse vector can be implemented using an auxiliary *marker array*, which will be explained shortly. The baseline HYPRE uses an alternative way of fusion shown in Fig. 1(b). While this approach further reduces space required for the temporary matrix B, it involves redundant floating point operations and memory accesses. Suppose non-zeros r_{11}, r_{12}, a_{11}, a_{21}, and p_{11}. The code in Fig. 1(a) computes b_{11} by $r_{11}a_{11} + r_{12}a_{21}$ then computes c_{11} by $b_{11}p_{11}$, a total of 4 floating point operations. The code in Fig. 1(b) computes c_{11} by $r_{11}a_{11}p_{11} + r_{12}a_{21}p_{11}$, which are 5 floating point operations. We measure that our new fusion approach requires on average 1.73$\times$ fewer floating point operations in the finest level triple matrix product for the matrices evaluated in §5.2.

Reordering of the Interpolation Matrix: In classical AMG, the interpolation function for error values at coarse points is the identity. Therefore, we can permute the $n_l \times n_{l+1}$ interpolation matrix P at multigrid level l to the form of $\begin{bmatrix} I_{n_{l+1}} \\ P_F \end{bmatrix}$, where the block with the first n_{l+1} rows is an identity matrix. Then, we can rewrite the coarse grid construction as follows:

$$\begin{aligned} RAP &= \begin{bmatrix} I \; P_F^T \end{bmatrix} \begin{bmatrix} A_{CC} \; A_{CF} \\ A_{FC} \; A_{FF} \end{bmatrix} \begin{bmatrix} I \\ P_F \end{bmatrix} \\ &= A_{CC} + P_F^T A_{FC} + \left(A_{CF} + P_F^T A_{FF} \right) P_F. \end{aligned}$$

Therefore, we only need a triple matrix product for the $(n_l - n_{l+1})^2$ submatrix A_{FF}. This optimization is particularly effective for matrices that lead to large operation complexities, where $\frac{n_{l+1}}{n_l}$ is high. The overhead of permuting interpolation matrices is easily amortized because the permutation also speeds up interpolation construction and solve phase as will be shown in the subsequent sections.

Software Prefetching: Since the R matrix is accessed contiguously, the hardware prefetcher effectively captures its spatial locality. The challenge remains in the non-contiguous access of the P matrices and especially the larger A matrices. While we are working on the j_1th row of A that corresponds to non-zero r_{ij_1}, we prefetch the j_2th row of A where r_{ij_2} is the next non-zero in ith row of R in software. Due to the indirections in the sparse matrix data structure, this access pattern is not captured by the hardware prefetcher of the current processors. We also unroll the innermost loop 8 times to facilitate prefetching cache line by line, which also helps instruction level parallelism.

Branching Overhead in Sparse Accumulation: We observe that branching is a significant performance bottleneck from the Vtune profile results. Branching is also an obstacle to vectorization in the current x86 SIMD instructions. SpGEMM is branch heavy because of accumulation in the sparse vectors. Specifically, the sparse vector $B_i = R_i A$ is computed by a weighted sum of sparse vectors $A_{j_1}, A_{j_2}, \ldots$, where R_i has non-zeros in columns j_1, j_2, and so on. An idiom of accumulating multiple sparse vectors is using an auxiliary marker array (denote this as `marker`). The content of `marker[i]` is the location in the output sparse vector where the ith element should be accumulated to. If this is the first time to accumulate the ith element, this information can also be obtained from `marker`. The marker array can be viewed as the inverse mapping of column indices in compressed sparse row format. The pseudo code below shows SpGEMM with $C = AB$. If `marker[k]` is smaller than `C.rowptr[i]`, it is the first time to accumulate C_{ik}. Then, we append k to `C.colidx`, set `marker[k]` to the current number of non-zero, `nnz`, and increment `nnz`. Otherwise, `marker[k]` points to the location of `C.values` where we should accumulate.

```
marker[:] ← -1
for each row i of A do
    C.rowptr[i] ← nnz
    for each non-zero a_ij in A_i do
        for each non-zero b_jk in B_j do
            if marker[k] < C.rowptr[i] then
                C.colidx[nnz] ← k
                C.values[nnz] ← a_ij · b_jk
                marker[k] ← nnz
                ++nnz
            else
                C.values[marker[k]] += a_ij · b_jk
```

This idiom is one of the most efficient ways of implementing an abstraction called sparse accumulator (SPA) that can be used as a building block of various sparse matrix operations as in MATLAB [29]. Accumulation of sparse vectors can also be viewed as a set union operation where values associated with the same key are reduced with addition. The marker array is essentially used as a hash table through which set union operations are done. This idiom also appears in other AMG setup routines such as coarsening and the construction of the interpolation matrix. This implies that branching can be a bottleneck in these other sparse matrix operations as well. We estimate this branching overhead by running a version of the triple matrix product where `rowptr` and `colidx` are already populated. This version has less branching overhead and can be used for repeatedly computing matrix products with the same non-zero patterns [27]. We observe on average 2.1× speedups, which shows potential for optimizing the branching overhead.

3.1.2 Interpolation Construction

This step constructs an $n_l \times n_{l+1}$ interpolation operator matrix P where n_l is the number of grid points in the finer level l and n_{l+1} is the number of grid points in the coarser level. Extended+i interpolation [7] is a distance-two interpolation that can compensate for convergence deterioration resulting from the use of more aggressive coarsening like PMIS, and can be computed via the following formula:

$$w_{ij} = -\frac{1}{\tilde{a}_{ii}}\left(a_{ij} + \sum_{k\in F_i^s} a_{ik}\frac{\bar{a}_{kj}}{b_{ik}}\right), j \in \hat{C}_i, \tag{1}$$

with

$$\tilde{a}_{ii} = a_{ii} + \sum_{n\in N_i^w \setminus \hat{C}_i} a_{in} + \sum_{k\in F_i^s} a_{ik}\frac{\bar{a}_{ki}}{b_{ik}},$$

$$b_{ik} = \sum_{l\in \hat{C}_i\cup\{i\}} \bar{a}_{kl},\ \ \bar{a}_{kl} = \begin{cases} 0, & \text{if } \mathrm{sign}(a_{kk}) = \mathrm{sign}(a_{kl}) \\ a_{kl}, & \text{otherwise,} \end{cases}$$

where N_i is the set of neighbors of i, S_i is the set of neighbors of i that strongly influence i, F_i^S contains the fine points in S_i, C_i^S the coarse points in S_i, $N_i^w = N_i \setminus (F_i^S \cup C_i^S)$, and $\hat{C}_i = C_i^S \cup \bigcup_{j\in F_i^S} C_j^S$.

In a distance-two interpolation we interpolate a point i not only from i's strongly influencing neighbors but also their respectively strongly influencing neighbors. In this respect, extended+i interpolation is similar to SpGEMM: when we multiply matrix A with B, for a given row i in A, we not only access each of i's neighbors j that corresponds to a non-zero a_{ij} but also accesses neighbors of j that correspond to non-zeros in the jth row of B.

Similarly to the coarse operator construction, the size of the resultant interpolation matrix is unknown a priori. Therefore, we apply the same technique of pre-allocating a large chunk of memory.

Also similarly to coarse operator construction, the interpolation operator construction has frequent if-else branches. In addition to a marker array checking for sparse accumulation, extended+i interpolation needs to distinguish fine points, coarse points with non-negative coefficients, and coarse points with negative coefficients, as can be seen from the above equation. We renumber coarse and fine points so that coarse points precede fine points, and permute matrices accordingly. Recall that this permutation is also used in the coarse operator construction, and it also helps smoothing operations that will be described in § 3.2. While we are permuting A, we also partition the coarse point columns in each row into those with non-negative coefficients and the others. As a result, each row will have three partitions: coarse point columns with non-negative coefficients, coarse point columns with negative coefficients, and fine point columns. This three way partitioning (i.e., partial sorting) requires only one sweep of data with $O(n)$ complexity rather than $O(nlogn)$ of full sorting, where n is the number of non-zeros in a matrix row.

Despite of these similarities with coarse operator construction, the actual computation performed in the interpolation is quite different as can be seen from Equation 1. The computation in the interpolation matrix construction is arranged in a way that the b_{ik} term for a given i and $k \in F_i^S$ is evaluated only once. To optimize for memory bandwidth, we fuse the interpolation construction with the interpolation truncation. The interpolation matrix is often truncated to keep the operator complexity small. For matrix row i, we set the truncation threshold to $min(\texttt{trunc\_fact} \times a_{i(1)}, a_{i(\texttt{max\_elmts})}$, where $a_{i(1)}$ is the largest absolute value of the non-zeros and $a_{i(\texttt{max\_elmts})}$ is the `max_elmts`th largest absolute value

```
copy x⃗ to temp_x
[is:ie) ← range of points this thread works on
for i in [is:ie) do                          ▷ Old hybrid GS for fine points
    if i is a fine point then
        acc ← b[i]
        for j in [rowptr[i]:rowptr[i+1]) do
            if j ∈ [is:ie) then
                acc −= x[colidx[j]]
            else
                acc −= temp_x[coidx[j]]
        x[i] ← acc
```

(a) The baseline

```
copy x⃗ to temp_x
[is_f:ie_f) ← range of fine points this thread works on
for i in [is_f:ie_f) do                      ▷ New hybrid GS for fine points
    acc ← b[i]
    for j in [rowptr[i]:extptr[i]) do
        acc −= x[colidx[j]]
                     ▷ extptr[i]: the first index belong to other threads
    for j in [extptr[i]:rowptr[i+1]) do
        acc −= temp_x[colidx[j]]
    x[i] ← acc
```

(b) Optimized hybrid GS with reordering

Figure 2: Pseudo code of hybrid Gauss-Seidel smoothing for fine-grid points. Smoothing for coarse points is similar.

of the non-zeros in row i. Non-zeros whose absolute values are below that this threshold is truncated. Typical values of the parameters `trunc_fact` and `max_elmts` are 0.1 and 4, respectively, that are used in §5. Instead of writing the entire interpolation matrix and then truncate it, we apply truncation to each interpolation matrix row immediately after the row is constructed.

3.2 Solve Phase

Smoothing: The interpolation construction in the setup phase permutes the operator matrix so that coarse points precede fine points as presented in §3.1.2. This also helps avoiding branches and improves spatial locality in smoothing. AMG often incorporates C-F smoothing where we apply smoothing first to the coarse points and then to the fine points in pre-smoothing and vice versa in post-smoothing [26]. Instead of checking if it is a coarse point for each row, we simply iterate over the coarse point range in the permuted matrix, and similarly for the fine points. In addition to reducing branching overhead, it helps the hardware prefetcher to be more effective.

Before the solve phase, we partition the non-zeros of lower and upper diagonals within each row of A_l. This allows us to skip the upper diagonals when the output vector for smoothing is initialized as zeros, which is common for pre-smoothing of coarse points. When using hybrid Gauss-Seidel smoothing, a 3-way partitioning is used to further separate out columns belonging to other threads. In hybrid smoothing, the output vector is copied to a temporary buffer, and we read the temporary buffer for columns belonging to other threads to honor write-after-read dependencies. By separating out columns belonging to other threads, we further reduce the branching overhead. Fig. 2 shows how hybrid Gauss-Seidel smoothing can be optimized using these reordering techniques.

Interpolation and Restriction: Interpolation and restriction also account for a significant fraction of solve time. These operations are implemented as sparse matrix vector multiplications (SpMV); interpolation multiplies P, and restriction multiplies $R = P^T$. As in the coarse operator construction, we exploit that P at level l can be permuted so that the first n_{l+1} rows are an identity matrix. Therefore, in the SpMVs for interpolation and restriction, we only need to access the remaining $(n_l - n_{l+1}) \times n_{l+1}$ matrix, saving memory bandwidth. In the baseline HYPRE, the transpose of P is computed for every restriction. We instead keep $R = P^T$ created for the coarse grid construction to reduce the transpose overhead.

3.3 Other Optimizations

We also apply the following relatively simple optimizations. While simple, these optimizations have a substantial impact on the performance. Some of these optimizations, in particular those related to OpenMP parallelizations, have not previously been incorporated because HYPRE AMG has focused more on multi-node scaling using MPI.

- **Strength matrix creation:** The final matrix creation is parallelized using prefix-sum.
- **PMIS coarsening:** We use the parallel random number generator in the Intel Math Kernel Library.
- **Matrix transpose:** We parallelize the matrix transpose using a parallel counting sort. The load balancing is maintained by partitioning rows among the threads in a way that each thread works on a similar number of non-zeros.
- **Fusion of SpMV and inner-product:** We fuse the sparse-matrix dense-vector multiplication (SpMV) with the inner product when computing the residual norm. When the output vector of SpMV is only used for computing its inner product, we can save the memory bandwidth of writing the output vector.

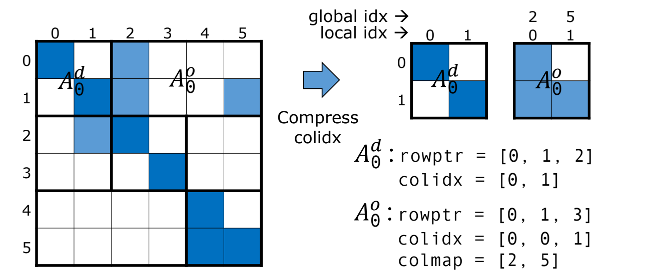
4. OPTIMIZATIONS FOR MULTIPLE NODES

This section presents optimizations we applied for multi-node scaling. To provide the background, we start with the distributed matrix representation in HYPRE.

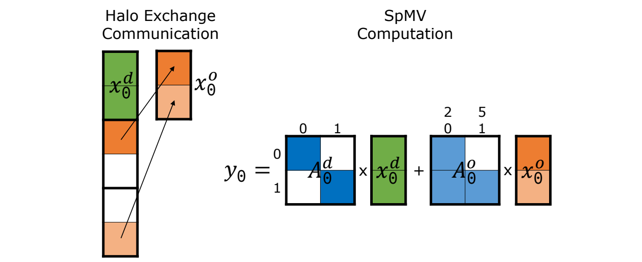
4.1 Distributed Matrix in HYPRE

In HYPRE, a distributed matrix is partitioned among MPI ranks by ranges of rows (more details in [30]). For example, as shown in Fig. 3(a), a matrix with 6 rows is distributed among 3 MPI ranks so that rank 0 owns the first 2 rows, and rank 1 owns the second 2 rows, and so on. Rank p maintains two local compressed sparse row (CSR) matrices to represent its portion, one matrix that corresponds to the rank's block diagonal portion (denoted as A_p^d) and another that corresponds to its block off-diagonal portion (denoted as A_p^o). In our example in Fig. 3(a), rank 0 has A_0^d that represents the first 2×2 block diagonal portion of the distributed matrix A, and A_0^o that represents the remaining portion of the first 2 rows. In the block off-diagonal matrix, column indices are "compressed" so that we can easily index the external vector elements that will be stored in a contiguous location after gathering from other ranks. In Fig. 3(a), A_0^o has non-zeros only in columns 2 and 5, and we renumber them as 0 and 1. We build `colmap` array that inverse maps to the original global column indices.

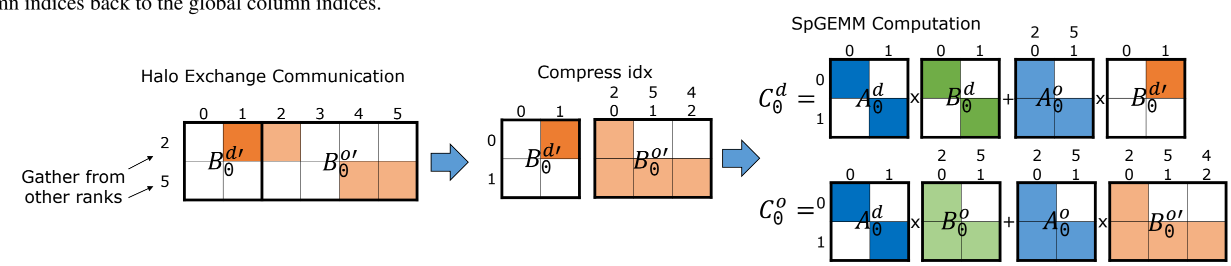
The reason for this compression should become clear with the example of the sparse matrix vector multiplication in Fig. 3(b). Based on the values stored in `colmap`, we gather vector elements necessary for SpMV but stored in other ranks. We call this MPI communication halo exchange because the elements that are gathered correspond to halo (i.e., boundary) points that have a connection with the given MPI rank. The gathered external vector elements are stored in a contiguous location of a vector and their indices



(a) An example distributed matrix in HYPRE. The matrix is partitioned by rows. Each MPI rank p stores its block diagonal portion A_p^d and its block off-diagonal portion A_p^o separately, both in compressed sparse row format. In the block off-diagonal matrix, column indices are compressed to facilitate operations such as SpMV. The array `colmap` maps the compressed local column indices back to the global column indices.



(b) SpMV operation of $y = A \cdot x$. Rank 0 gathers $x[2]$ and $x[5]$ that are needed for SpMV but belong to other ranks (`colmap` tells us which elements we should gather). We call this MPI communication halo exchange. They are copied to a contiguous location as a separate vector x_0^o, and this vector is multiplied with A_0^o using the local SpMV routine. The halo exchange is overlapped with computation of $A_0^d \cdot x_0^d$.



(c) The SpGEMM operation of $C = A \cdot B$ (B has the same sparsity pattern as A for simple illustration). Rank 0 gathers the third and sixth rows of B that are needed for SpGEMM but belong to other ranks, and assembles as a matrix denoted as B_0'. Because its block off-diagonal portion, $B_0^{o'}$, has additional column 4 that does not exist in B_0^o, we append an entry to `colmap`.

Figure 3: An example distributed matrix (a), SpMV (b) and SpGEMM operations (c).

within the vector matches with the compressed local column indices. Therefore, we can reuse the same local SpMV routine for both the block diagonal and off-diagonal parts.

Distributed SpGEMM proceeds similarly but with more challenges. Suppose that we compute $C = A \cdot B$ as in Fig. 3(c). By looking at `colmap`, rank p determines matrix rows needed for SpGEMM but stored in other ranks, then gathers these rows using MPI communication and assembles a matrix we denote as B_p'. Note that this halo exchange step involves gathering matrix rows rather than gathering vector elements as in SpMV, hence leading to a larger communication volume. In addition, the portion of matrix B_p' gathered from other ranks can contain off-diagonal columns that do not exist in B_p^o. For example, in Fig. 3(c), rank 0 gathers row 5 of B from rank 2, and this row has column 4 that does not exist in rank 0's portion of matrix B. Therefore, we need to renumber the indices of off-diagonal columns. In Fig. 3(c), we append column 4 to `colmap` and assign a local column index 3 to it.

We identify that this renumbering accounts for a significant fraction of various routines used in the setup phase such as coarse operator construction, interpolation construction, and matrix transpose, hence a bottleneck in multi-node scaling. Recall that extended+i interpolation traverses neighbors of neighbors, and, therefore it needs a similar halo exchange of matrix rows (rather than exchange of vector elements) and accompanied renumbering pattern. There are two reasons why this renumbering for SpGEMM-like operation takes a considerable amount of time. First, SpGEMM has substantially more column indices to renumber than SpMV does. SpGEMM needs to renumber neighbors of neighbors, while SpMV only needs to renumber neighbors. Second, for each matrix, we apply SpGEMM-like operations such as coarse grid construction and

hash_table $H \leftarrow \emptyset$ ▷ H is thread-private
for each jth non-zeros in matrix B_p' in parallel **do**
  $c \leftarrow B_p'$.colidx[j]
  if $c \notin$ { rows that p owns } and $c \notin B_p$.colmap **then**
    $H \cup = c$
B_p'.colmap ← sort and eliminate duplicates of Hs in parallel
reverse_colmap ← hash table for reverse of B_p.colmap
for each jth non-zeros in matrix B_p' in parallel **do**
  $c \leftarrow B_p'$.colidx[j]
  if $c \notin$ { rows that p owns } and $c \notin B_p$.colmap **then**
    B_p'.colidx[j] ← reverse_colmap[c] + B_p.colmap.size()

Figure 4: Optimized implementation of column index renumbering

interpolation only once. This is in contrast to SpMV that are repeatedly applied per iteration in the solve phase, where the renumbering cost can be amortized. The next section presents our optimization applied to column index renumbering to address these challenges.

4.2 Parallelization of Column Index Renumbering

The column index renumbering roughly translates into a problem of sorting while eliminating duplicates. A slight variation is that rank p only renumbers new column indices that do not already exist in B_p. A straight forward implementation would append all new column indices to an ordered set, but parallelization of this approach would involve a concurrent binary tree that has limited scalability. Instead, as in the pseudo code shown in Fig. 4, each thread builds a thread-private hash table of the column indices for the portion of the matrix assigned to the thread. Here, we exploit

Table 1: Evaluation Settings

	HYPRE	AmgX
Version	2.10.0b (2015.1.22)	2014.12.22
Compiler	Intel compiler 15.0.2	CUDA 6.5
Processor	Xeon E5-2697 v3 (HSW)	Tesla K40c
Parallelism	1 Socket × 14 Cores × 2-wide SMT × 4-wide SIMD	15 multiprocessors 2,880 CUDA cores
Memory	32 GB	12 GB
Clock	2.6 GHz	876 MHz
On-chip stores	32KB private L1$ 256KB private L2$ 35,840KB shared L3$	64KB constant mem 48KB shared mem 1,536KB shared L2$
STREAM triad BW	54 GB/s	249 GB/s (ECC off)

the locality of common matrices arising from scientific problems: adjacent rows share many non-zeros in common columns. Because of this locality, each thread-private hash table filters out a large fraction of duplicated column indices without incurring synchronization overhead. Later, we merge the thread-private hash tables into a single sorted array, B'_p.colmap, using parallel merge sort [31] with a modification that also eliminates duplicates. Then, we construct a hash table that maintains the reverse mapping of B'_p.colmap. This hash table is actually a collection of thread-private hash tables partitioned over input ranges. We equally partition the B'_p.colmap among threads and have each thread construct its own reverse mapping with hash table. Since the input forward mapping is already sorted without duplicates, each thread will construct a hash table for disjoint input ranges. Finally, we renumber each new column index, by first doing a binary search to find which hash table to look up, and then looking up the selected hash table. Alternatively, we can binary search B'_p.colidx for each new column index without constructing the reverse mapping. However, constructing the reverse mapping reduces the time complexity of each lookup from $O(log\ n)$ to $O(log\ t)$, where n is the size of B'_p.colmap and t is the number of threads.

4.3 Eliminate Unnecessary MPI Data Transfers in Interpolation Construction

As can be seen in Equation 1, to compute a non-zero in the resultant interpolation matrix w_{ij}, in addition to the ith row of A, we need to access the kth row of A where $k \in F_i^S$. This kth row can be located in a remote MPI rank. Instead of fetching the entire kth row, we filter out many of its non-zeros that are not used for interpolation construction. For example, we only access a_{kj} such that $j \in \hat{C}_i$ or $j = i$. We also do not need a_{kj} such that its sign is same as a_{ik}. §5.3 will show that this optimization leads to a significant reduction of MPI data transfers.

4.4 Other Optimizations

During the solve phase, we repeat the same pattern of exchanging data among MPI ranks. Instead of repeatedly calling the same set of `MPI_Isends` and `MPI_Irecvs` in each data exchange, we create persistent communication requests before the solve phase and simply call a single `MPI_Startall` for each exchange. Persistent communication amortizes various set up costs. For example, we can reuse data structures generated for lower level protocols like InfiniBand verbs. The MPI run-time can also handshake with the network interface hardware in one transaction instead of one for each `Isend/recv`.

5. EVALUATION

This section evaluates the performance of our optimized HYPRE AMG implementation. We quantify and analyze its single-node performance on the latest x86 multi-core processor, compare it with the NVIDIA AmgX implementation, and finally study its scalability on a multi-node system.

Table 2: Sparse matrices used in single-node experiments. `Lap2d_2000` can be generated from AMG2013 benchmark. `Lap3d_128` is from HPCG benchmark [32]. The other matrices are from University of Florida Collection [33].

	rows	nnz/row
1. `2cubes_sphere`	101,492	9
2. `G2_circuit`	150,102	5
3. `G3_circuit`	1,585,478	5
4. `StocF-1465`	1,465,137	14
5. `apache2`	715,176	7
6. `atmosmodd`	1,270,432	7
7. `atmosmodj`	1,270,432	7
8. `atmosmodl`	1,489,752	7
9. `ecology2`	999,999	5
10. `lap2d_2000`	4,000,000	5
11. `lap3d_128`	2,097,152	27
12. `parabolic_fem`	525,825	7
13. `thermal2`	1,228,045	7
14. `tmt_sym`	726,713	5

Table 3: AMG parameter settings for single-node evaluation

Solver	Standalone AMG (i.e., not as preconditioner)
Cycle	V, `max_levels`=7
Coarsening	Classical, PMIS [4], `str_thr`=0.25 or 0.6*, `max_row_sum`=0.8
Interpolation	Extended+i [7] with truncation options `trunc_fact`=0.1, `max_elmts`=4
Smoother	Hybrid GS (HYPRE), `GS` (AmgX)
Relative tolerance	1e-7

*Selected the one for faster time to solution for each matrix.

5.1 Setup

5.1.1 Single-Node

We evaluate HYPRE on a Haswell generation Intel Xeon processor and compare its performance with NVIDIA AmgX running on K40c. Detailed specifications are listed in Table 1. In the Xeon processor, we use a thread affinity setting of `KMP_AFFINITY=granularity=fine,co` We do not use hyper-threading[2]. We use non-complex double precision numbers for all our experiments. We use the latest versions of both software packages and the latest compiler/run-time that these packages support. We observe considerable speedups in

[2]Hyper-threading helps hide latency. Since our optimizations already reduce latency, for example from reducing branch misprediction by pre-sorting matrix elements, we see no speedup from hyper-threading.

Table 4: AMG parameter settings for multi-node evaluation

Solver	Flexible GMRES [34] with AMG preconditioner
Cycle	V, `max_levels`=16

Coarsening/ Interpolation	`ei(4)`. same options as single-node
	`2s-ei(444)`. Top MG level: aggressive PMIS and 2-stage extended+i interpolation [14] with truncation at every stage, Other levels: `ei(4)`
	`mp`. Top level: aggressive PMIS and multipass [35] interpolation, Other levels: `ei(4)`

Smoother	Hybrid GS
Relative tolerance	1e-7 (weak scaling), 1e-5 (strong scaling)

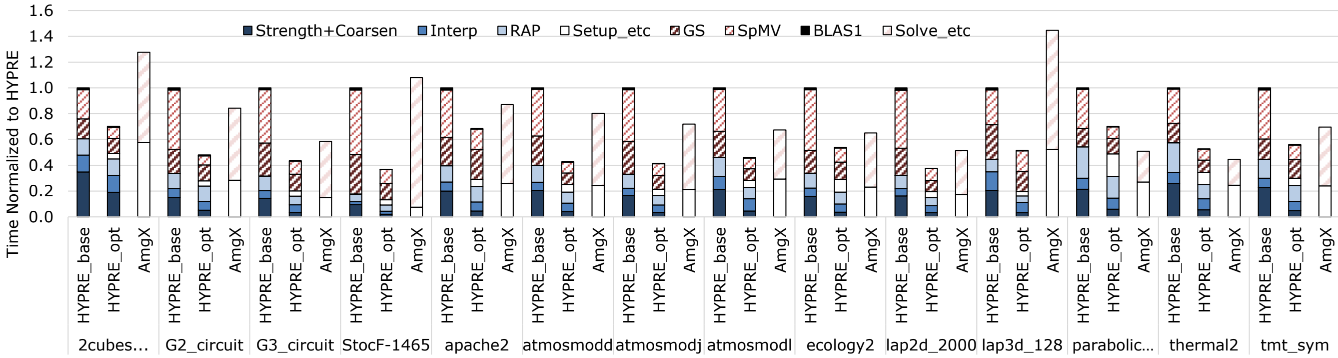


Figure 5: Single-node performance comparison of the baseline HYPRE 2.10.0b (`HYPRE_base`), our optimized HYPRE (`HYPRE_opt`), and AmgX. Times are normalized to the time to solution of `HYPRE_base`. `Strength+Coarsen:` strength matrix creation and PMIS coarsening, `Interp:` interpolation construction, `RAP:` Galerkin triple matrix product, `Setup_etc:` other setup times including pre-processing of reordering, `GS:` Gauss-Seidel smoothing, `SpMV:` sparse matrix vector multiplication, `BLAS1:` vector scaling, addition, and inner-products, `Solve_etc:` other solve times.

HYPRE version 2.10.0b released on January 2015 compared to the prior versions such as the one used in comparison with AmgX [9]. Table 1's last row also shows STREAM triad performance in GB/s, which, in the first order of approximation, provides an upper-bound on achievable performance of AMG. Thus, according to the STREAM benchmark performance, AmgX is expected to be more than 4× faster than HYPRE, as long as both software packages achieve similar efficiency with respect to memory bandwidth.

The AMG implementation can be guided by a rich set of parameters, and its convergence, setup time, and time per each solve iteration can widely vary for different parameters. Therefore, for fair comparison of both software packages, we made an extra effort to use as similar parameters as possible. These parameters are listed in Table 3. Because HYPRE implements the classical AMG, we use the classical AMG option in AmgX. We use the University of Florida collection matrices [33] evaluated in NVIDIA's comparison [9]. We add 2D Laplace (5-point discretization) and a 3D Laplace (27-point discretization) matrix, which are from AMG2013 [36] and HPCG benchmarks [32], respectively. These matrices are listed in Table 2.

The settings used for our single-node experiments do not necessarily result in the fastest time to solution (we use a relative residual norm reduction of 1e-7 as the stopping criterion). For example, our single-node experiments do not use AMG as a preconditioner of a Krylov solver such as GMRES, which is often faster, in order to reveal AMG performance only. Rather than the fastest time to solution, the focus of our single-node experiments is a fair comparison, which can be quantified by operator complexities. When operator complexities are similar, two AMG solvers are likely to generate similar outputs such as interpolation and coarse grid operator matrices in the setup phase. Among the evaluated matrices, the difference in operator complexity ranges between -14–2% (averages -0.2% and standard deviations 4%).

While operator complexities quantify the similarity of setup phase outputs, it is hard to do so for the solve phase because of the lack of details on how AmgX smoothers are implemented. We find that the `GS` option consistently provides a faster time to solution than the `MULTICOLOR_GS` option in AmgX, but it is not clear exactly what the `GS` option implements. We believe this is not a lexicographical GS since its limited parallelism is not suitable for GPUs [24].

5.1.2 Multi-Node

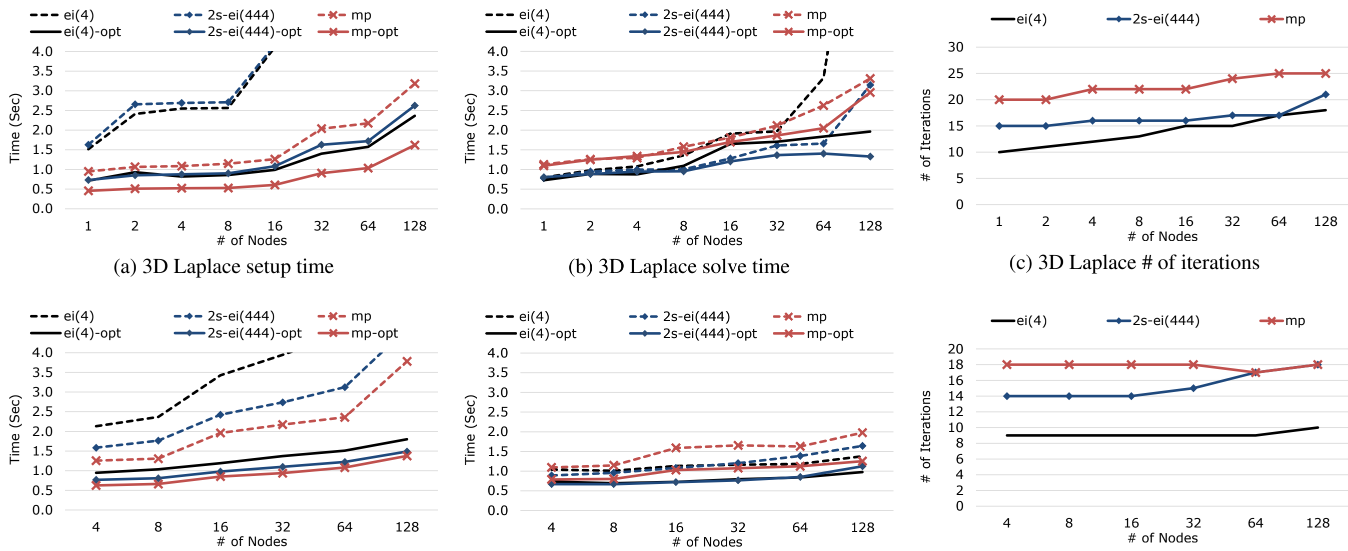
In contrast to the single-node experiments, our multi-node experiments use the best performing settings we were able to find, which are listed in Table 4. We do not compare with multi-node AmgX results due to the lack of access to large enough GPU clusters. We use AMG as the preconditioner of Flexible GMRES solver, and compare multiple coarsening and interpolation settings that have considerable impact on the overall AMG performance. Among various interpolation settings evaluated, we present 3 representative ones: the default recommended setting used for our single-node experiments (`ei(4)`) and two other settings with aggressive coarsening [35] applied to the top MG levels (`2s-ei(444)` with 2-stage extended+i and `mp` with multipass interpolation). Aggressive coarsening with long range interpolation is an important tool to maintain the scalability of AMG with respect to both convergence factor and operator complexity. While multipass interpolation [35] has been often used for its simplicity, it has been shown that 2-stage extended+i interpolation exhibits more robust numerical scalability for a wider range of problems.

We study multi-node scalability on the Endeavor cluster, which has the same Haswell generation Intel Xeon processor used for the single-node evaluation. Each Endeavor cluster compute node has 2 such Xeon processors and 64 GB of memory. These compute nodes are connected with an FDR Infiniband fabric with fat-tree topology. We use Intel MPI 5.0.2.044, and run 1 MPI rank per processor (2 ranks per node) to optimize for NUMA.

We measure weak scaling with two input sets (3D Laplace matrices and the default semi-structured matrices in AMG2013 benchmark [36]). We apply strong scaling to a problem that models an elliptical PDE for permeability fields in reservoir simulation, generated geostatistically using sequential Gaussian simulations [37]. While this problem models a Poisson-like equation, it involves highly discontinuous coefficients and is thus not well conditioned. We use a tolerance of 1e-5 for the stopping criterion to reflect the accuracy requirement of a typical application solving the equation.

5.2 Single Node Performance and Comparison with AmgX

Fig. 5 shows the time to solution normalized to that of the baseline HYPRE (`HYPRE_base`). Our optimized version (`HYPRE_opt`) is on average 2.0 and 1.3× faster than `HYPRE_base` and AmgX, respectively. We first compare `HYPRE_opt` with `HYPRE_base`. We verify that, when the baseline random number generator is used in PMIS coarsening, `HYPRE_opt` results in the identical number of iterations and the final residual norm for all matrices. The results shown in Fig. 5 are with parallel random number generation in



(a) 3D Laplace setup time
(b) 3D Laplace solve time
(c) 3D Laplace # of iterations
(d) AMG2013 semi-structured setup time
(e) AMG2013 semi-structured solve time
(f) AMG2013 # of iterations

Figure 6: Weak scaling multi-node performance (a-c) 3D Laplace matrix with 27-pt discretization from HPCG benchmark [32], ∼27 non-zeros per row, $96^3 \simeq 0.9$M rows and ∼0.27 GB per rank. (d-e) The semi-structured input from AMG2013 benchmark [36], `r`=32 and `pooldist=1` (generates realistic inputs and requires ≥8 ranks), ∼8 non-zeros per row, ∼1.6M rows and 0.15 GB per rank.

MKL, and the number of iterations differs by 2% on average. Because the solve and setup time can contribute to the overall time to solution differently depending on the context, we break down solve and setup times. For example, while solving individual linear systems requires one setup for every solve, in time dependent problems, setup will be called only occasionally.

In strength matrix creation and PMIS coarsening, we obtain on average 6.1 and 3.1× speedups, respectively. The speedups in the interpolation operator construction are not as big except for 3D Laplace matrices. In fact, it slightly slows down for small matrices such as `2cubes_sphere` and `G2_circuit` because the time for partially sorting each matrix row is not sufficiently amortized. However, as will be shown, our partial sorting optimization clearly benefits larger matrices evaluated for multi-node scaling. In the triple matrix product used for coarse operator construction (`RAP`), our memory optimizations described in §3.1.1 provide on average 1.4× speedup. The speedup is in general higher for larger matrices, which is more important target for AMG as a scalable solver algorithm.

In `HYPRE_base`, SpMV is the most time consuming kernel of the solve phase, and transposing the interpolation matrix accounts for a large fraction of it. By keeping the transpose of the interpolation matrix that is generated during the setup phase and using it for the transposed-matrix vector multiplication to avoid transposing the matrix for every restriction, we achieve an average of 3.7× speedup in SpMV. Hybrid Gauss-Seidel smoothing (`GS`) is also sped up by 1.2× on average due to reordering of matrices. This speedup accounts for the reordering overhead that is included in `Setup_etc`. We also evaluate lexicographical GS based on an efficient point-to-point synchronization [38], and fusion of lexicographical GS and SpMV [39]. Lexicographical GS provides 1.26× faster convergence on average. However, in the scenario of one setup per every solve, the faster convergence does not compensate for its limited parallelism and higher pre-processing overhead for building dependency graphs, except for matrices with high inherent parallelism such as `lap3d_128` and `parabolic_fem`. If we consider a scenario where setup cost can be amortized significantly, we observe that lexicographical GS can be faster for 5 matrices — `G3_circuit`, `StocF-1465`, `lap3d_128`, `parabolic_fem`, and `thermal2`.

We now compare `HYPRE_opt` with NVIDIA's AmgX. Because AmgX only provides setup and solve times without further breakdown, we mark its setup and solve times as `Setup_etc` and `Solve_etc`. Even with similar operator complexities, AmgX consistently requires more iterations, 1.3× on average. We cannot exactly point out the reason for this due to the lack of more detailed information on its smoother. We believe however this is because the smoother option `GS` we used invokes a hybrid GS that can lead to worse convergence with higher concurrency. Multi-color GS smoothing with option `MULTICOLOR_GS` provides on average 1.4× faster convergence, but its setup and solve time is 1.2 and 2.8× higher than `GS` on average, respectively. The setup time of AmgX is on par with `HYPRE_opt`, 1.1× faster on average. The solve time of AmgX on the other hand is 2.1× slower. Even if we compute per iteration time to isolate the effect from the convergence drop, the solve time is still on average 1.6× slower.

5.3 Multi-Node Weak Scaling Performance

Having shown that our optimized implementation greatly improves the single-node performance of HYPRE AMG and is competitive with AmgX, this section shows its scalability to multiple nodes. Fig. 6 shows weak scaling results up to 128 compute nodes.

We compare multiple coarsening and interpolation schemes. Multipass interpolation (`mp`) provides faster setup, but extended+i based interpolations (`ex(4)` and `2s-ei(444)`) converges faster, thus providing faster solve, for the inputs evaluated. Therefore, the frequency of setup will determine which interpolation scheme will be overall the fastest. `HYPRE_opt` improves the best setup times (with `mp`) by 2.0 and 2.7× on 128 nodes for the two inputs evaluated, respectively. The best solve times among different interpolation schemes are improved by 2.1 and 1.5×. Note that our optimizations reduce the gap between setup time of `mp` and that of other interpolation schemes. Multipass interpolation has been proposed earlier than the 2-stage interpolations, and thus more heavily opti-

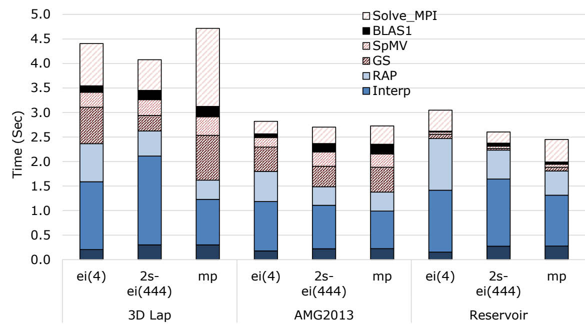
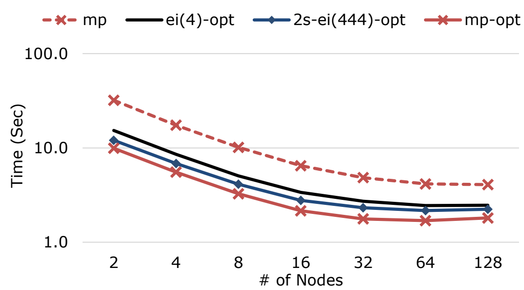
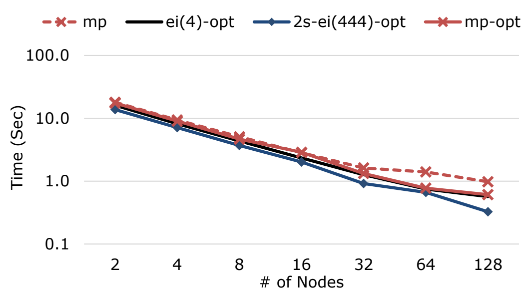


Figure 7: breakdowns of total (setup+solve) time of HYPRE_opt on 128 nodes.



(a) Setup time



(b) Solve time

Figure 8: Strong scaling multi-node performance. 7 non-zeros per row, 128M rows, 10 GB total. Note that y-axis is in log scale.

mized in the baseline HYPRE. However, 2-stage interpolations are more numerically robust as shown in [14]. The time breakdown in Fig. 7 shows that 2-stage aggressive coarsening trade-offs longer interpolation construction time for shorter `RAP` and solve time.

For ideal weak scaling, AMG should exhibit a constant time-to-solution as the number of nodes increases. In practice, the performance deviates from the ideal due to two factors. First, the number of iterations can gradually increase because of non-ideal coarsening and interpolation. For example, AMG often trade-offs convergence for faster setup time [4]. Fig. 6(c) shows that the number of iterations to convergence gradually increases for the 3D Laplace matrix for all 3 interpolation schemes evaluated, while Fig. 6(f) shows that the number of iterations mostly stays constant for the semi-structured input from AMG2013 benchmark.

Second, the setup time and the time per iteration of the solve phase can increase due to non-scaling parts of computation or communication. Since this factor is more pronounced in strong scaling scenarios, it will be discussed in the next section.

5.4 Multi Node Strong Scaling Performance

Figure 8 shows strong scaling results of the reservoir simulation input with different interpolation schemes. The number of iterations to converge stays constant at 8, 10, and 14, for `ei(4)`, `2s-ei(444)`, and `mp`, respectively. We only show the fastest interpolation scheme (`mp`) for the baseline HYPRE to simplify the graph.

Even though our optimizations improve the setup time significantly, the general trend still remains: the scaling of the setup is less ideal than that of the solve. This indicates that setup scalability will be a challenge for extreme scale AMG solvers, especially in strong scaling settings. Among the setup routines, interpolation construction and RAP product exhibit the worst scalability. The speedup of interpolation construction from 2 to 128 nodes ranges from 4.5 to 6.4, depending on the interpolation scheme. The same of RAP product ranges from 4.2 to 5.0. In `HYPRE_base`, `Interp` and `RAP` spend more than half of their time in MPI communication and renumbering column indices of the received non-zeros. Efficient parallel renumbering of column indices (§4.2) speeds up `RAP` by factors of 2.6 and 3.5 for the two inputs on 128 nodes with `ei(4)`, respectively. In addition to the optimization in renumbering, `Interp` incorporates eliminating unnecessary MPI communication (§4.3), reducing the communication volume by more than 3× for both inputs. This leads to 8.8 and 2.8× speedups of interpolation construction with `ei(4)` on 128 nodes.

Even though the solve time scales better, when 128 nodes are used, we still observe that the solve phase spends >60% of its time in MPI communication. `Solve_MPI` bar in Fig. 7 includes halo exchange and all-reduce times, and the halo exchange in distributed SpMV and hybrid GS accounts for more than 80%. On 128 nodes, we observe 1.8 and 1.7× speedups of halo exchanges by using persistent communication. When we strong scale to 128 nodes, MPI messages in halo exchanges become less than 100 KB long. We measure less than 1 GB/s effective uni-directional bandwidth per node for these messages, about 1/6 of the peak expected from the Infiniband fabric in Endeavor cluster. This scalability issue in the AMG solve phase is also studied by Gahvari et al. [40], and they suggest to reduce the communication volume by trading it for redundant computation or by new coarsening and interpolation schemes to create operators with less communication.

6. CONCLUSION AND FUTURE WORK

This paper presents an AMG implementation based on the widely used HYPRE library optimized for x86 multi-core processors. On a single node, our implementation outperforms the baseline HYPRE AMG and NVIDIA's AmgX by 2.0 and 1.3×, respectively. Our optimized implementation provides similarly high speedups compared to the baseline HYPRE AMG in multi-node settings, especially when numerically robust long-range interpolation schemes are used.

This paper also lays out interesting future work. First, *achieving setup scalability is more challenging than solve scalability*, in particular when constructing the interpolation. By incorporating techniques such as aggressive coarsening, long-range interpolation, and interpolation truncation, we can reduce operator complexities, and reduce the communication volume in restriction and prolongation. These techniques however make interpolation construction more complex and thus take longer than the RAP product, which has been perceived as the main bottleneck in the setup phase. Therefore, more optimization efforts are needed for interpolation construction. Second, we observe that *the accumulation of sparse vectors is a common pattern accounting for a significant fraction of the setup time*, including interpolation construction and RAP products. It will be interesting to see speedups of the sparse accumulation from the upcoming AVX-512 instructions such as `vcompressd`. Lastly, *optimizations like reordering and fusion require changes beyond the* *scope of kernels*, which can hamper the modular design of sparse solver libraries. This calls for a sparse solver library design that accommodates both modularity and inter-kernel optimizations, and programming system supports such as domain specific compiler optimizations.

Acknowledgements

The authors first would like thank Robert Falgout for the discussion that led to this paper. We also thank Abdulrahman Manea at Stanford and Jason Sewall at Intel for providing the reservoir simulation data used in our strong scaling experiments. We also thank Intel Endeavor team for their quick and competent support for the cluster.

References

[1] Y. Saad and M. H. Schultz, "GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 7, no. 3, 1986.

[2] Argonne National Laboratory, "Aurora Argonne Leadership Computing Facility," http://aurora.alcf.anl.gov.

[3] A. Brandt, S. F. McCormick, and J. W. Ruge, "Algebraic multigrid (AMG) for automatic multigrid solutions with application to geodetic computations," Report, Inst. for Computational Studies, Fort Collins, Colo., 1982.

[4] V. E. Henson and U. M. Yang, "BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2000.

[5] K. Stüben, "An Introduction to Algebraic Multigrid," in *Multigrid*, 2001.

[6] U. M. Yang, "Parallel algebraic multigrid methods - high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, 2006.

[7] H. D. Sterck, R. Falgout, J. Nolting, and U. M. Yang, "Distance-Two Interpolation for Parallel Algebraic Multigrid," *Numerical Linear Algebra with Applications*, vol. 15, 2008.

[8] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of Scaling Algebraic Multigrid across Modern Multicore Architectures," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2011.

[9] M. Naumov, "AmgX: Scalability and Performance on Massively Parallel Platforms," http://www.siam.org/meetings/ex14/14-naumov-slides.pdf, 2014, SIAM Workshop on Exascale Applied Mathematics Challenges and Opportunities.

[10] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, A. G. Salinger, H. K. Thornquist, R. S. Tuminaro, J. M. Willenbring, A. Williams, and K. S. Stanley, "An Overview of the Trilinos Project," *ACM Transactions on Mathematical Software*, vol. 31, no. 3, 2005.

[11] S. Balay, S. Abhyankar, M. F. Adams, J. Brown, P. Brune, K. Buschelman, V. Eijkhout, W. D. Gropp, D. Kaushik, M. G. Knepley, L. C. McInnes, K. Rupp, B. F. Smith, and H. Zhang, "PETSc Web page," http://www.mcs.anl.gov/petsc, 2014. [Online]. Available: http://www.mcs.anl.gov/petsc

[12] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream.

[13] K. Stüben, "A review of algebraic multigrid," *Computational and Applied Mathematics*, vol. 128, 2001.

[14] U. M. Yang, "On Long Range Interpolation Operators for Aggressive Coarsening," *Numerical Linear Algebra with Applications*, vol. 17, no. 2-3, 2010.

[15] P. Vaněk, J. Mandel, and M. Brezina, "Algebraic Multigrid on Unstructued Meshes," *University of Colorado at Denver, CCM Report 34*, 1994.

[16] B. D. Braess, "Towards Algebraic Multigrid for Elliptic Problems of Second Order," *Computing*, vol. 55, 1995.

[17] P. Vaněk, J. Mandel, and M. Brezina, "Algebraic Multigrid by Smoothed Aggregation for Second and Fourth Order Elliptic Problems," *Computing*, vol. 56, 1996.

[18] R. Gandham, K. Esler, and Y. Zhang, "A GPU accelerated algebraic multigrid method," *Computers and Mathematics with Applications*, vol. 68, 2014.

[19] Y. Notay, "An aggregation-based algebraic multigrid method," *Electronic Transactions on Numerical Analysis*, vol. 37, no. 6, 2010.

[20] G. Haase, M. Liebmann, C. C. Douglas, and gernot Plank, "A Parallel Algebraic Multigrid Solver on Graphics Processing Units," *High Performance Computing and Applications*, 2010.

[21] M. Wagner, K. Rupp, and J. Weinbub, "A Comparison of Algebraic Multigrid Preconditioners using Graphics Processing Units and Multi-Core Central Processing Units," in *Society for Computer Simulation International Symposium on High Performance Computing*, 2012.

[22] 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, 2012.

[23] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[24] E. Phillips and M. Fatica, "A CUDA implementation of the High Performance Conjugate Gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, 2014.

[25] "AmgX Reference Manual," 2014.

[26] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid Smoothers for Ultra-Parallel Computing," *SIAM J. on Sc. Computing*, vol. 33, 2011.

[27] F. G. Gustavson, "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition," *ACM Transactions on Mathematical Software*, vol. 4, no. 3, 1978.

[28] M. M. A. Patwary, N. R. Satish, N. Sundaram, J. Park, M. J. Anderson, S. Gautam, D. Das, S. G. Pudov, V. O. Pirogov, and P. Dubey, "Parallel Efficient Sparse Matrix-Matrix Multiplication on Multicore Platforms," in *International Supercomputing Conference (ISC)*, 2015.

[29] J. R. Gilbert, C. Moler, and R. Schreiber, "Sparse Matrices in MATLAB: Design and Implementation," *SIAM Journal on MAtrix Analysis and Application*, vol. 13, no. 1, 1992.

[30] R. D. Falgout, J. E. Jones, , and U. M. Yang, "Pursuing Scalability for hypre's Conceptual Interfaces," *ACM Transaction on Mathematical Software*, vol. 31, no. 3, 2005.

[31] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *International Conference on Management of Data (SIGMOD)*, 2010.

[32] J. Dongarra and M. A. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems," Sandia National Laboratories, Tech. Rep. 4744, 2013.

[33] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 15, no. 1, 2011, http://www.cise.ufl.edu/research/sparse/matrices.

[34] Y. Saad, "A Flexible Inner-outer Preconditioned GMRES Algorithm," *SIAM Journal on Scientific Computing*, vol. 14, no. 2, 1993.

[35] K. Stüben, *Algebraic multigrid (AMG): an introduction with applications*. GMD-Forschungszentrum Informationstechnik, 1999.

[36] Lawrence Livermore National Lab, "AMG2013," https://codesign.llnl.gov/amg2013.php.

[37] N. Remy, A. Boucher, and J. Wu, *Applied Geostatistics with SGeMS: A User's Guide*. Cambridge University Press, 2011.

[38] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver," in *International Supercomputing Conference (ISC)*, 2014.

[39] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2014.

[40] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms," in *International Conference on Supercomputing (ICS)*, 2011.