Systematic Reduction of Data Movement in Algebraic Multigrid Solvers

Hormozd Gahvari*, William Gropp*, Kirk E. Jordan†, Martin Schulz‡
and Ulrike Meier Yang‡

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Computational Science Center, IBM TJ Watson Research Center, Cambridge, MA 02142
‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551

{gahvari,wgropp}@illinois.edu, kjordan@us.ibm.com, {schulzm,umyang}@llnl.gov

***Abstract*—Algebraic Multigrid (AMG) solvers find wide use in scientific simulation codes. Their ideal computational complexity makes them especially attractive for solving large problems on parallel machines. However, they also involve a substantial amount of data movement, posing challenges to performance and scalability. In this paper, we present an algorithm that provides a systematic means of reducing data movement in AMG. The algorithm operates by gathering and redistributing the problem data to reduce the need to move it on the communication-intensive coarse grid portion of AMG. The data is gathered in a way that ensures data locality by keeping data movement confined to specific regions of the machine. Any decision to gather data is made systematically through the means of a performance model. This approach results in substantial speedups on a multicore cluster when using AMG to solve a variety of test problems.**

I. Introduction

The rising scale of HPC systems not only requires applications to match the increased level of concurrency available in the system, but also imposes new constraints and limitations, in particular in terms of resilience and power consumption. The latter is directly tied to data movements, which are responsible for a majority of the power consumed in a system. To address these challenges and to successfully exploit future architectures, we therefore need new algorithmic approaches that specifically target the reduction of data movements and at the same time offer new avenues for resiliency.

In our work we approach this topic from the algorithmic side focusing on Algebraic Multigrid (AMG) methods, a class of solvers for large, sparse linear systems of equations that finds use in a wide range of scientific applications. AMG has the ideal property of having a computational complexity that is linear in the number of unknowns, when it works well, and has shown excellent weak scaling to the size of current high-end systems, such as IBM Blue Gene/L [1] and Blue Gene/P [2]. However, for architectures with wide multicore nodes, AMG is starting to run into scaling bottlenecks, which are directly connected to its algorithmic approach. AMG obtains its optimal computation complexity by using smaller "coarse grid" problems to accelerate the solution of the original "fine grid" problem. Since the number of nonzeros per row for the matrices on the coarser levels grows, communication complexity also increases significantly, leading to a large number of messages. Making things worse, the number of nodes involved in the communication only decreases slowly, in particular on systems with wide multicore nodes, since often at least one core per node still participates at coarser grid levels. However, the large number of processes at coarser grid levels can be aggregated and the resulting set of tasks can be either executed on a subset of nodes (agglomeration approach) or several copies of this set of tasks redundantly across various subsets of nodes (redundant approach). If these strategies are applied at the correct level, the communication requirements at coarser levels are significantly reduced, and in the extreme case even eliminated by aggregating the remaining coarser computation onto single, potentially replicated, processes.

In this paper we focus on the use of redundancy, since it not only reduces communication requirements, but also offers potentials for implicit resiliency due to the redundant nature of the computation at coarser levels. Our algorithm partitions the problem domain into chunks and distributes these chunks to subsets of the involved processes. By carefully selecting which processes get which chunks, communication can be made to occur only in localized fashion.

One of the critical aspects of this approach is the decision about the right level to switch to redundant cycling. In our approach we guide this decision at runtime based on a performance model of AMG, which we extended to include redundant cycling. This enables us to automatically tune our approach to the input set without loss of generality.

In particular, this paper makes the following contributions:

- We explore a new algorithmic approach for AMG that reduces overall communication using redundant data distributions,
- We extend an existing performance model to cover redundant data distribution schemes, and
- We use this model to dynamically determine the appropriate level for switching to the redundant scheme.

Overall, our algorithm combined with our novel model guided dynamic level adaptation is up to 3x faster when using AMG to solve a variety of realistic problem cases.

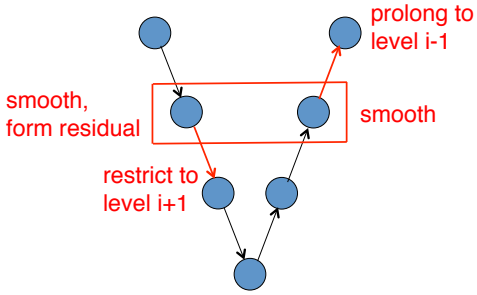

Fig. 1. A multigrid V-cycle, with the fundamental operations at a given level highlighted. A smoother application is followed by the formation of the residual, which is restricted to the next coarsest grid. Upon receiving the coarse grid correction, there is then another round of smoothing followed by interpolation of the result to the next finest grid.

II. Algebraic Multigrid

Multigrid methods are widely used when solving large equation systems on parallel machines, since they are optimal, i.e., they can solve a linear system with N unknowns with $O(N)$ computational work. They achieve this by performing part of the work that other solvers would perform on the original "fine grid" problem on smaller "coarse grid" problems instead. This process is known as coarse grid correction. After the application of an inexpensive smoother, such as a Jacobi or Gauss-Seidel iteration on the fine grid, which results in an approximate solution, that approximation is corrected on the next coarsest grid. The correction can be obtained either through direct solution or another round of smoothing followed by a further recursive coarse grid correction. The use of multiple grids gives the process the term multigrid; we consider here its simplest form, the V-cycle, in which the work proceeds from the finest grid to the coarsest grid and then back to the finest grid (see Figure 1). We number the grids from finest to coarsest; if there are L grids, the finest grid is level 0, and the coarsest grid is level $L-1$.

The first multigrid methods were geometric in nature, and were used to solve problems on structured grids. AMG extends multigrid to cover problems on unstructured grids such as finite element meshes. All that is required is a linear system $Ax = b$. The grid information is inferred from the graph of A. This requires AMG to operate in two phases. The first is a setup phase, in which the hierarchy of grids including various operators is determined. At each level, the grid points that will remain on the next coarsest grid are selected, followed by the formation of an operator which restricts the residual on that level to the next coarsest grid and a prolongation operator, which interpolates the correction back up from that grid. Here, as is often the case in practice, the restriction interpolation is chosen to be the transpose of the interpolation operator. The solve operator for the next grid is then formed by multiplying the restriction, the solve and the interpolation operator of the current level. Once the hierarchy of grids has been set up, AMG switches to the solve phase. In the solve phase, a series of multigrid cycles, such as V-cycles, are applied to the hierarchy of grids generated in the setup phase until the residual error is below a given threshold. More information about AMG can be found in [3]; a good overview of multigrid methods in general is available in [4].

There are a number of implementations of AMG, which offer access to a wide variety of coarsening and interpolation schemes for generating the hierarchy of grids and a number of smoothers for use with the solve phase. In our experiments, we use the BoomerAMG solver [5] in the hypre software library [6]. We use HMIS coarsening [7] with extended+i interpolation [8] truncated to at most 4 elements per row. For 3D problems, we also use aggressive coarsening with multipass interpolation [9] on the first level. These schemes were developed over a number of years to keep the hierarchy of operators from being overly complex while also maintaining good numerical convergence. Overly complex operators most often result in severely degraded parallel performance [7]. For the smoother, we use hybrid Gauss-Seidel, which is comprised of Gauss-Seidel iteration between process boundaries and Jacobi iteration across process boundaries. More information about parallel smoothers for AMG can be found in [10].

Sparse matrices in BoomerAMG are stored in the ParCSR matrix data structure. In this data structure, if there are p MPI processes, the matrix A is partitioned by rows into matrices A_k, $K = 0, \ldots, p-1$. A_k is stored locally as two sequential CSR (compressed sparse row) sparse matrices, D_k and O_k. D_k contains all entries in A_k whose column indices point to rows stored on process k. O_k contains the remaining entries, which have column indices that point to rows stored on other processes. Computing a matrix-vector product (MatVec) Ax involves computing $A_k x = D_k x^D + O_k x^O$ on each process, where x^D is the portion of x stored locally and x^O is the portion of x that needs to be sent by other processes. More detail is available in [11].

The efficient implementation of multigrid methods on parallel machines has been an area of research for quite some time, leading to the development of several variants with reduced data movement. Much of this work is centered around the basic idea of accumulating and redistributing the problem data onto a subset of the processes at a particular level of the multigrid cycle and performing the rest of the cycle on those processes only. One option to implement such a scheme is to redundantly distribute the data onto those processors. Then each one has the same data, meaning that they no longer have to communicate with each other during the redundantly treated levels. Gropp [12] found this approach to be beneficial for geometric multigrid in some cases. A later study of this approach for AMG [13], which was added to the hypre library starting with version 2.8.0b [14], found that it could yield substantial speedups, but also noted that the benefits diminished at scale. Womble and Young [15] used a more gradual, bottom-up approach to redundancy in geometric multigrid, where pairs of communicating processors with few enough remaining unknowns replicated their data, but did not consider data locality.

Another variant of the data gathering approach uses plain agglomeration, i.e., it involves simply concentrating the data replicated by the redundant approach onto a subset of the involved processes without replication, performing the rest of the cycle there, and then redistributing the result to the originally involved processes. Such an approach has been used by Nakajima [16] for the coarse grid solve, in Sandia's ML smoothed aggregation AMG solver [17] and by Sampath and Biros [18] for an octree-based geometric multigrid solver to deal with convergence degradation and/or load imbalance.

Our focus here is on the overall reduction of data movement. While future plans include the investigation of both the agglomeration and redundant approaches, we focus first on the redundant variant, due to its potential for data recovery and improved fault tolerance in AMG. In the next section, we present our algorithm for reducing data movement in AMG, which relies on a performance model to decide the amount and level of data gathering to be performed. This model can provide the necessary information to find the best tradeoff between reduced data movement and excess computation. Additionally, our algorithm is designed to gather data within specific portions of the machine to improve data locality, further reducing the amount of data movement.

III. Algorithm

Our algorithm takes the basic redundant approach of [12] and [13], but significantly enhances scalability by distributing smaller portions of the problem data to each process. Since, as a consequence, this will not be the entire problem data, the redundant phase still requires communication, but there will be far fewer messages to send. To further reduce communication, the algorithm also allows for the data to be gathered in specific regions of the machine to improve data locality.

We call our algorithm the chunks algorithm because the data is gathered into a set of chunks that are distributed to the involved processes in such a way that communication only occurs between processes that own different chunks, with no communication occurring between processes that own the same chunk, as illustrated in Figure 2 for 4 chunks with 3 processes each. All processes that own the same chunk have the exact same portion of the problem data. For the agglomeration approach, each chunk would only consist of one process. As one of the key contributions, we tie our algorithm to a performance model that allows the algorithm to dynamically decide when to switch to redundancy to best improve performance. The model can also be applied to the agglomeration approach.

A. Algorithm Details

We implemented the chunks algorithm as an addition to hypre, to take advantage of the infrastructure already in place for the basic redundant algorithm. Any decision to switch to the chunks algorithm is made during the AMG setup phase just after a coarse grid is formed, before beginning the coarsening process that forms the next coarsest grid. If a decision is made to switch, we form two sets of MPI communicators:

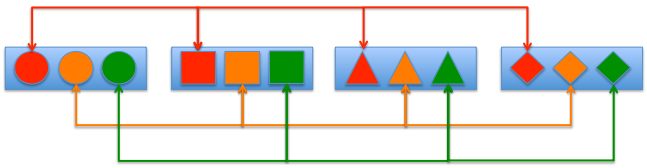

Fig. 2. Illustration of chunk data distribution with 4 chunks (blue blocks) and 12 processes (red, orange, and green shapes). Each chunk is owned by 3 processes depicted as having the same shape. The communication pattern can be regularized as shown by the arrows, with processes in a color group communicating only with each other.

collective communicators, which gather data to form chunks, and point-to-point communicators, to handle communication between chunks. We create one collective communicator for each chunk, and one point-to-point communicator for each cluster, which we define to be a group of processes in different chunks that will need to exchange data during the solve phase. Clusters are depicted as groups of processes of the same color in Figure 2. After the communicators are formed, all processes within a chunk perform `MPI_Allgatherv` operations to acquire the matrix data for their chunk. Then, new parallel matrices are formed over processes in each cluster. Once this is complete, each matrix is treated as the finest level operator for a new coarse solver object, and the parallel AMG setup routine is called for each of the new matrices. If there is just one chunk, then it is the basic redundant case. In this case, the collective communicator is `MPI_COMM_WORLD` (or a smaller communicator consisting of just the active processes if some have dropped out), and there is no point-to-point communication.

In the solve phase, we perform additional `MPI_Allgatherv` operations in the cycle during the switchover to the redundant phase to split and reorganize the right hand side and the solution vector into chunks. The solve cycle is then called on the coarse solver object and the reorganized right hand side and solution vector. Once that portion of the cycle is complete, the appropriate pieces of the result are copied from the chunks solution vector to the non-chunks solution vector. This step requires no communication, as the processes in each chunk will already have all data needed stored locally.

B. Data Placement Strategy

Though the chunks algorithm can be run with any set of collective and point-to-point communicators that allows for each process in a chunk to be part of its own cluster of communicating processes, to maximize the benefits of the chunks algorithm, we must place data in a way that keeps communication within well-defined units of the machine, as this preserves locality. Such considerations are going to be even more important on future machines, with data movement becoming more and more expensive relative to computation in terms of both performance and power. How to best do this depends on the mapping of processes to nodes in the machine. Here, we present strategies for how to do this for

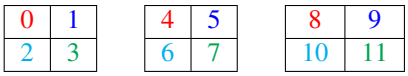

Fig. 3. Assignment of processes to collective and point-to-point communicators for chunks algorithm for the case of 12 processes and 4 chunks with a block mapping of processes to nodes. Processes with the same color are in the same collective communicator; processes in the same box are in the same point-to-point communicator.

Fig. 4. Assignment of processes to collective and point-to-point communicators for chunks algorithm for the case of 12 processes and 4 chunks with a cyclic mapping of processes to nodes. Processes with the same color are in the same collective communicator; processes in the same box are in the same point-to-point communicator.

the typical default mappings, block and cyclic, of processes to nodes in a way that ensures that all communication during the chunks stage takes places within nodes. Assume there are C chunks that divide the number of processes evenly, and $N = \frac{P}{C}$ clusters. In each placement strategy, C is taken to be the number of processes per node. The key is then to have every chunk represented among the processes that are on the same node.

1) Block Mapping: If there is a block mapping of tasks to nodes, we make the process with rank r belong to chunk $r \mod C$. The processes that will need to combine data are the sets $\{r | r \mod C = i\}$ for $i = 0, \ldots, N-1$, which form the collective communicators. The point-to-point communicators will then be the groups of processes numbered $\{0, 1, \ldots, C-1\}$, $\{C, C+1, \ldots, 2C-1\}$, $\ldots$, $\{(N-1)C, (N-1)C+1, \ldots, NC-1\}$. Figure 3 illustrates this for 12 processes and 4 chunks.

2) Cyclic Mapping: If there is a cyclic mapping of tasks to nodes, the communicators for the block mappings will be transposed. The collective communicators are numbered $\{0, 1, \ldots, N-1\}$, $\{N, N+1, \ldots, 2N-1\}$, $\ldots$, $\{(C-1)N, (C-1)N+1, \ldots, CN-1\}$, and the point-to-point communicators are the sets of process ranks $\{r | r \mod N = i\}$ for $i = 0, \ldots, C-1$. Figure 4 illustrates this for 12 processes and 4 chunks.

C. Data Gathering Automation with the Performance Model

We use a performance model of the AMG solve cycle to automate the decision to switch to the chunks algorithm. Our model is based on a simple latency-bandwidth model for communication and then adds a communication distance term and penalties to take into account multicore issues and limited bandwidth. It is able to capture the performance of AMG on a number of different machines and under MPI-only and hybrid MPI/OpenMP programming models while making use of machine parameters that can be determined from simple measurements and/or the topology of the interconnect, with cycle time prediction accuracies of over 90% in many cases; the details of how are in our past work on this subject [19]–[21]. Before describing how we use the model to automate the decision to switch to the chunks algorithm, we give an overview of the model when running entirely MPI tasks, one per core, on the machine, since this is what we ran in our experiments.

1) Model Overview: Our model of the AMG cycle works by breaking it down into its fundamental operations. Let T^i_{solve} be the time spent at level i in the cycle. The breakdown is

$$T^i_{\text{solve}} = T^i_{\text{smooth}} + T^i_{\text{restrict}} + T^i_{\text{interp}},$$

where T^i_{smooth} is the time spent smoothing on level i, T^i_{restrict} is the time spent restricting from level i to level $i-1$, and T^i_{interp} is the time spent interpolating from level i to level $i+1$. In a cycle with L levels, the total time spent is

$$T^{\text{AMG}}_{\text{solve}} = \sum_{i=0}^{L-1} T^i_{\text{solve}}.$$

We treat the individual steps in terms of the linear algebra operations involved. At each level, the smoothing time involves one MatVec to form the residual, and two applications of the smoother. A smoother application is a similar operation to a MatVec, so we treat it in the same fashion. The restriction time involves one MatVec with the restriction operator (the transpose of the interpolation operator in our experiments), and the interpolation time involves one MatVec with the interpolation operator. To enable us to derive equations for these operations, we define the following parameters. For our baseline communication model, we model the time to send an n-element message as $T_{\text{send}} = \alpha + n\beta$, where α is the communication start-up cost and β is the per-element send cost. α covers both software overhead and latency involved in message passing, and β is tied to the available bandwidth. We then define P to be the number of processes, and t_i to be the time per floating-point operation at level i. The remaining parameters, which cover the operators, are:

- C_i – number of unknowns on level i
- $s_i, \hat{s}_i$ – average number of nonzero entries per row in the level i solve and interpolation operators, respectively
- $p_i, \hat{p}_i$ – maximum number of sends over all processes in the level i solve and interpolation operators, respectively
- $n_i, \hat{n}_i$ – maximum number of elements sent over all processes in the level i solve and interpolation operators, respectively

We now express the time spent in each step as a function of α and β to give us our baseline model. The time spent smoothing at level i is

$$T_{\text{smooth}}(\alpha, \beta) = 6\frac{C_i}{P} s_i t_i + 3(p_i\alpha + n_i\beta).$$

The time spent restricting from level i to level $i+1$ is given by

$$T^i_{\text{restrict}}(\alpha, \beta) = \begin{cases} 2\frac{C_{i+1}}{P}\hat{s}_i t_i + \hat{p}_i\alpha + \hat{n}_i\beta & \text{if } i < L-1 \\ 0 & \text{if } i = L-1. \end{cases}$$

The time spent interpolating from level i to level $i-1$ is given by

$$T^i_{\text{interp}}(\alpha, \beta) = \begin{cases} 0 & \text{if } i = 0 \\ 2\frac{C_{i-1}}{P}\hat{s}_{i-1} t_i + \hat{p}_{i-1}\alpha + \hat{n}_{i-1}\beta & \text{if } i > 0. \end{cases}$$

We augment this baseline model with additional terms and penalties to reflect issues observed on real machines. The penalties can be "on" or "off" depending on the architecture, and the best fit to a particular machine will have some penalties in effect and others not in effect. We first add a γ term that represents the delay per hop to take into account messages traveling long distances. This change is reflected in the baseline model by replacing α with

$$\alpha(h) = \alpha(h_m) + (h - h_m)\gamma,$$

where h is the number of hops a message travels, and h_m is the smallest possible number of hops a message can travel in the network. h is assumed to be the diameter of the network within the job's partition to account for routing delays. h_m is 1 in a torus or mesh network, and 2 in fat-tree networks where a message has to travel through 1 switch, which involves using two links.

Another issue is limited bandwidth. Message passing applications have difficulty achieving the full bandwidth provided by the hardware in ideal conditions, and this bandwidth is in turn rarely achieved under non-ideal conditions. Another source of limited bandwidth is contention from messages sharing links. We take both of these into account with a penalty to β. Let $B_{\max}$ be the peak aggregate per-node bandwidth, and B be the measured bandwidth corresponding to β, which is $B = \frac{8}{\beta}$ with β the time to send one double-precision floating-point value. The penalty for being unable to achieve the full hardware bandwidth is the fraction $\frac{B_{\max}}{B}$. To account for link contention, let m be the number of messages in the network, and l be the number of links available to the job. The penalty for this is $\frac{m}{l}$. The overall penalty is obtained by multiplying β by the sum of both of these terms: $\beta \leftarrow \left(\frac{B_{\max}}{B} + \frac{m}{l}\right)\beta$.

Multicore nodes introduce two more issues: the possibility of increased contention between cores when accessing the interconnect, and increased contention in switches caused by the extra message load these cores inject into the network. We model this by multiplying the $\alpha(h_m)$ and γ terms by $\left\lceil c\frac{P_i}{P}\right\rceil$. Here, c is the number of cores per node, and P_i is the number of active processes on level i (active processes are processes that have not dropped out of the computation). The resulting penalized terms are $\alpha \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \alpha(h_m)$ and $\gamma \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \gamma$.

2) Making the Decision to Switch to Chunks: When deciding whether or not to switch to chunks at a given level l, we use the model to predict T_{noswitch}, the time spent at this level if we do not switch, and T_{switch}, the time spent if we switch. If the latter is less, then we switch. We try different numbers of chunks, testing from the smallest power of two less than p_l down to a minimum number, which is 1 in the general case and the number of MPI tasks per node if we are using one of the data placement strategies designed to keep communication on-node, selecting for T_{switch} the number of chunks that results in the smallest time.

Assuming we get correct values for T_{switch} and T_{noswitch}, it is in fact best to switch at the first level at which $T_{\text{switch}} < T_{\text{noswitch}}$. If we switch at a finer level $\hat{l} < l$, then we will have made a suboptimal decision due to a slowdown at levels $\hat{l}$ through $l-1$ combined with the same improvements on levels l through $L-1$ that we would have obtained from the original switch. If we switch at a coarser level $\tilde{l} > l$, the decision will again be suboptimal because switching at level l would give the same improvements as switching at level $\tilde{l}$ combined with improvements at levels l through $\tilde{l}-1$ that switching at level $\tilde{l}$ would not have obtained.

To determine T_{noswitch}, we use the model to predict the time of five matrix-vector multiplications using the existing level l solve operator. Three of them represent the operations normally done with the solve operator, smoothing and residual formation. The other two represent restriction and interpolation. We have to approximate these with the solve operator, as at the stage in the setup where we have to make the decision, the restriction and interpolation operators have yet to be formed. The expression for T_{noswitch} in terms of the baseline model is

$$T_{\text{noswitch}}(\alpha, \beta) = 10\frac{C_l}{P}s_l t_l + 5(p_l\alpha + n_l\beta).$$

We assume the network parameters required by the model for making a runtime decision are supplied from machine measurements. We cannot assume this for the computation rate, however, as it can vary greatly depending on the size of the solve operator and the pattern and number of nonzero entries in it [22]; this is why it was allowed to vary in the original model. We instead measure it as follows. We perform 10 sequential MatVecs with the D_k matrix from the ParCSR data structure on each process, and divide the observed time by the number of flops performed, i.e., 20 times the number of nonzero entries in D_k. We exclude processes that have no data and report a time per flop of zero. We take the maximum reported value over all processes to be t_l. However, if the measured value for t_l is greater than that of the one for t_{l-1}, we set $t_l = t_{l-1}$ and set $t_k = t_{l-1}$ for all levels $k > l$. This happens because processes close to running out of data will exhibit measurements with an abnormally high time per flop due to measuring primarily overhead instead of flops, while in reality it decreases with decreasing matrix dimension and increasing density [22], trends that both hold when progressing from fine to coarse in AMG.

To determine T_{switch}, we use the model to predict the time of five matrix-vector multiplications with a redistributed solve operator and two all-gather operations, one to gather the problem data and one to gather the right-hand side. Let C be the number of chunks. The all-gather operations are over sets of $\frac{P}{C}$ processes; we assume they gather the problem data over a binary tree and then broadcast that data along the same tree. Counting from the root, each stage of gathering data on the tree involves sends that are approximately of size $\frac{C_l}{2C}, \frac{C_l}{4C}, \frac{C_l}{8C}, \ldots$. In accordance with our communication model, we charge the amount of data sent as $\frac{C_l}{C}\left(\frac{1}{1-\frac{1}{2}} - 1\right) = \frac{C_l}{C}$. The broadcast step involves sending approximately $\frac{C_l}{C}\left\lceil \log_2 \frac{P}{C}\right\rceil$ units of data. Using the baseline α-β model, the expression for the all-gather

time is

$$T_{\text{allgather}}(\alpha,\beta) = 2\left\lceil \log_2 \frac{P}{C} \right\rceil \alpha + 2\frac{C_l}{C}\left(1 + \left\lceil \log_2 \frac{P}{C} \right\rceil\right)\beta.$$

For the redistributed operator, every element in the model for a MatVec is subject to change except for the network parameters. We divide the number of unknowns and nonzero entries in the original operator equally among the chunks, and the amount of data sent per message equally among the number of communication partners in the nonredistributed operator. The number of communication partners each process has in the redistributed operator cannot be known a priori except in the fully redundant case. We assume this is $C-1$, as this is the maximum number of communication partners a process could have, unless there are fewer chunks than communication partners. Then, we assume the number of partners is unchanged from the original operator. To reflect this in the model, we introduce the modified value $p_l^* = \min\{p_l, C-1\}$ for the maximum number of sends in the redistributed level l solve operator. The computation rate t_l that was measured for the original operator is not going to be the same either; we let t_l^* be the value for the redistributed operator. Then, in terms of the baseline model,

$$\begin{aligned} T_{\text{switch}}(\alpha,\beta) =& 10\frac{C_l}{C}s_l t_l^* + 5p_l^*\left(\alpha + \frac{n_l}{p_l}\right) \\ &+ 2T_{\text{allgather}}(\alpha,\beta). \end{aligned}$$

However, obtaining an actual value for t_l^* is impractical, since we cannot measure the changed rate for the redistributed operator without first performing the redistribution. We instead handle it as follows. We first classify the local pre-chunks MatVec operation as one of the following three categories, a system explained in more detail in [22]:

- Small: the matrix and the source vector fit in cache
- Medium: the source vector fits in cache, but the matrix does not
- Large: the source vector does not fit in cache

The time per flop undergoes significant jumps when moving from a smaller category to a larger one [22]. If this is caused by data gathering, it will have a dramatic effect on the resulting performance of the chunks algorithm. Therefore, when determining the number of chunks, we exclude all values that cause the local problem classification to increase in size. For all other values, we assume $t_l^* = t_l$ for the purposes of determining T_{switch}.

We must reiterate here the importance of having a performance model that takes features on modern parallel machines that make data movement more costly into account. Not adequately penalizing interprocessor communication will result in too conservative of a decision to switch to chunks due to on-processor computation being more expensive relative to communication in the model than it actually is. Not taking the cache into account will result in too eager of a decision to switch to chunks because on-processor computation then becomes too inexpensive relative to interprocessor communication.

IV. Results

We tested our algorithm on Hera, a multicore Linux cluster at Lawrence Livermore National Laboratory. Hera consists of 800 compute nodes, with four quad-core 2.3 GHz AMD Opteron processors per node and a L2 cache size of 512 KB per core as well as 2MB shared L3 cache per processor. The nodes are connected by a DDR Infiniband interconnect organized as a two-level fat-tree. The network parameters needed for the performance model were measured by microbenchmarking and/or calculated from the network topology. All of the penalties added to the baseline model are in effect. More detail is available in [20].

We used five different test problems to illustrate the applicability of the chunks algorithm to multiple problems. Our suite consists of three 3D problems on a cubic domain and two 2D problems on a square domain. For the 3D problems, we used a problem size of $30 \times 30 \times 30$ points per core and ran on 64, 512, and 4096 cores. For the 2D problems, we used a problem size of 150×150 points per core, and ran on 64, 256, and 1024 cores.

7pt Laplace: Our first 3D test problem is a 3-dimensional 7-point Laplace problem on a cube.

27pt Stencil: The second test problem is a 3D diffusion problem with a 27-point stencil on a cube.

Convection-Diffusion: The final 3D problem is a nonsymmetric problem, derived with finite differences on a regular grid from the partial differential equation

$$-u_{xx} - u_{yy} - u_{zz} + c(u_x + u_y + u_z) = 1,$$

with $c = 1$.

9pt Laplace: Our first 2D test problem is a Laplace problem on a regular grid with a 9-point stencil, derived from a finite element discretization.

Rotated Anisotropy: Our final problem is a 2-dimensional rotated anisotropy on a uniform square:

$$-(c^2 + \epsilon s^2)u_{xx} + 2(1-\epsilon)scu_{xy} - (s^2 + \epsilon c^2)u_{yy} = 1,$$

where $c = \cos\gamma, s = \sin\gamma, \gamma = 60^\circ$ and $\epsilon = 0.01$.

For each problem, we ran a combination of one AMG setup and 10 V-cycles ten times, averaging the reported times to avoid impact caused by noise. We used a cyclic mapping of MPI tasks per node, and the cyclic collective and point-to-point communicators for the chunks algorithm, keeping communication during the chunks stage entirely within nodes.

Table I presents the results for the original algorithm (No Chunks), the chunks algorithm with the performance model used to guide when to make the switch, and the speedup achieved. In all cases, the number of chunks used was 16. We also determined speedups for the total times from the setup time plus the number of solve cycles needed when using the iterative solver GMRES(5) [23] preconditioned with AMG to solve each problem to a tolerance of 10^{-6}, which are given in Table II. The specific level at which the switch is made for each problem is given in Table III. The speedups were substantial for all 3-dimensional cases, almost always over

TABLE I
RESULTS OF CHUNKS VS. ORIGINAL ALGORITHM FOR EACH TEST PROBLEM.

Problem	Setup Times			Cycle Times		
7pt Laplace	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	1.04 s	4.32 s	10.76 s	50.8 ms	138.4 ms	179.2 ms
Chunks	0.76 s	1.40 s	3.76 s	34.4 ms	67.9 ms	114.7 ms
Speedup	1.37	3.08	2.86	1.48	2.04	1.56
27pt Stencil	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	1.14 s	4.20 s	9.87 s	74.7 ms	149.4 ms	186.7 s
Chunks	0.68 s	1.53 s	3.58 s	48.9 ms	77.7 ms	130.3 ms
Speedup	1.68	2.74	2.76	1.53	1.92	1.43
Convection-Diffusion	64 Cores	512 Cores	4096 Cores	64 Cores	512 Cores	4096 Cores
No Chunks	0.95 s	3.94 s	10.62 s	46.5 ms	119.0 ms	167.9 ms
Chunks	0.79 s	1.69 s	3.58 s	35.8 ms	82.7 ms	96.1 ms
Speedup	1.20	2.33	2.97	1.30	1.44	1.75
9pt Laplace	64 Cores	256 Cores	1024 Cores	64 Cores	256 Cores	1024 Cores
No Chunks	0.47 s	0.98 s	1.35 s	25.9 ms	41.6 ms	46.7 ms
Chunks	0.28 s	0.65 s	0.94 s	22.7 ms	35.6 ms	35.3 ms
Speedup	1.68	1.51	1.44	1.14	1.17	1.32
Rotated Anisotropy	64 Cores	256 Cores	1024 Cores	64 Cores	256 Cores	1024 Cores
No Chunks	0.47 s	0.98 s	1.83 s	38.7 ms	58.0 ms	78.2 ms
Chunks	0.32 s	0.71 s	1.14 s	26.4 ms	40.6 ms	54.7 ms
Speedup	1.47	1.38	1.60	1.46	1.43	1.43

TABLE II
SPEEDUPS FOR TOTAL TIMES ACHIEVED BY CHUNKS COMPARED TO ORIGINAL ALGORITHM.

3D Problem	64 Cores	512 Cores	4096 Cores
7pt Laplace	1.36	2.74	2.56
27pt Stencil	1.62	2.47	2.40
Convection-Diffusion	1.24	2.04	2.70
2D Problem	64 Cores	256 Cores	1024 Cores
9pt Laplace	1.43	1.28	1.40
Rotated Anisotropy	1.49	1.40	1.55

40%. The setup phase in particular showed great improvement, with most speedups well over 2x, and much improved weak scalability. We also saw improvements for the 2-dimensional cases, which generally have much lower computational and communication complexities and therefore less potential for performance improvement.

V. CONCLUSIONS

We have successfully introduced a new algorithm that improves the performance of AMG through the reduction of data movement. The algorithm uses redundant data distribution to reduce and regularize the communication pattern of AMG, and enables communication to take place within well-defined and localized units of the machine. We have coupled our algorithm with a performance model that can decide dynamically at runtime when switching to the new algorithm is beneficial. In our experiments, we achieved speedups of up to 3x in the setup phase and 2x in the solve phase.

Current trends in computer architecture forecast an increasing role for our algorithm in preparing AMG for larger-scale machines. Per-chip and per-node core counts are expected to increase, with some researchers [24] envisioning a future with thousands of cores per chip. The chunks algorithm enables adaptation to this future. As the size of the problems being solved on larger machines increases, so can the number of chunks, to keep the problem size per core from being too large, while still keeping interprocess communication entirely within chips or nodes. In addition, the algorithm can be extended in a tree-like fashion by applying the simple chunks algorithm recursively to each chunk.

There are many avenues for future work. We are interested in the potential redundancy brings for resiliency and further algorithmic innovation. We expect to use the extra data to recover more quickly from faults. Furthermore, we will explore extensions to this approach that enable us to perform different computations on the duplicated pieces of data in parallel and then recombine them in a way that accelerates convergence. This idea was once popular [25], but was later found not to have enough benefit to overcome the cost of the extra work [26]. However, with the chunks algorithm able to place the duplicated pieces in places of the machine where work on each piece will not interfere with work on other pieces, we are hoping to see some form of performance gain from such an approach. Additionally, power is a growing concern and the reduction in data movement realized by this work is a significant first step to reducing power requirements. We will explore this issue further and investigate how to create a hybrid algorithm between agglomeration and redundancy to reduce power usage further without a loss in performance or resilience.

Another interesting topic is the interaction of the chunks algorithm with changes to the programming model. For simplicity, we considered an MPI-only programming model in this paper, but hybrid models that combine MPI with other programming models are becoming increasingly popular, most notably hybrid MPI/OpenMP for multicore clusters. The use of such a programming model has shown significant performance gains as well as increased scalability on several architectures [27], but will not entirely address the performance challenges it faces on coarse grids, particularly when dealing with systems at extreme scales. The combination of the chunks algorithm, which addresses the coarse grid difficulties, and

TABLE III
NUMBER OF LEVELS IN THE AMG HIERARCHY, ALONG WITH LEVEL AT WHICH THE SWITCH TO CHUNKS WAS MADE, FOR EACH PROBLEM AND PROCESS COUNT. AS A REMINDER, THE LEVELS ARE NUMBERED STARTING FROM 0.

3D Problem	64 Cores		512 Cores		4096 Cores	
	Num. Levels	Switch Level	Num. Levels	Switch Level	Num. Levels	Switch Level
7pt Laplace	7	2	8	3	9	4
27pt Stencil	7	1	8	3	8	3
Convection-Diffusion	7	2	8	3	9	4
2D Problem	64 Cores		256 Cores		1024 Cores	
	Num. Levels	Switch Level	Num. Levels	Switch Level	Num. Levels	Switch Level
9pt Laplace	9	5	9	6	10	6
Rotated Anisotropy	11	6	11	7	12	7

a hybrid programming model, is therefore very promising. In summary, the chunks algorithm and the future research opportunities it opens have significant potential for ensuring the scalability of AMG to next generation parallel machines.

ACKNOWLEDGEMENTS

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-SC0004131, and performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-587832). Neither Contractor, DOE, or the U.S. Government, nor any person acting on their behalf: (a) makes any warranty or representation, express or implied, with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in this document.

REFERENCES

[1] R. D. Falgout, "An introduction to algebraic multigrid," *Computing in Science and Engineering*, vol. 8, pp. 24–33, 2006.

[2] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Scaling hypre's Multigrid Solvers to 100,000 Cores," in *High-Performance Scientific Computing: Algorithms and Applications*, M. W. Berry, K. A. Gallivan, E. Gallopoulos, A. Grama, B. Philippe, Y. Saad, and F. Saied, Eds. Springer, 2012, pp. 261–279.

[3] K. Stüben, "An introduction to algebraic multigrid," in *Multigrid*, U. Trottenberg, C. Oosterlee, and A. Schüller, Eds. San Diego, CA: Academic Press, 2001, pp. 413–528.

[4] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*, 2nd ed. SIAM, 2000.

[5] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, April 2002.

[6] "*hypre*: High performance preconditioners," http://www.llnl.gov/CASC/hypre/.

[7] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing complexity in parallel algebraic multigrid preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, pp. 1019–1039, 2006.

[8] H. De Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Numerical Linear Algebra With Applications*, vol. 15, pp. 115–139, April 2008.

[9] U. M. Yang, "On long-range interpolation operators for aggressive coarsening," *Numerical Linear Algebra With Applications*, vol. 17, pp. 453–472, April 2010.

[10] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid Smoothers for Ultra-Parallel Computing," *SIAM Journal on Scientific Computing*, vol. 33, pp. 2864–2887, 2011.

[11] R. D. Falgout, J. E. Jones, and U. M. Yang, "Pursuing Scalability for *hypre*'s Conceptual Interfaces," *ACM Transactions on Mathematical Software*, vol. 31, pp. 326–350, September 2005.

[12] W. Gropp, "Parallel Computing and Domain Decomposition," in *Fifth Conference on Domain Decomposition Methods for Partial Differential Equations*, T. Chan, D. Keyes, G. Meurant, J. Scroggs, and R. Voigt, Eds. SIAM, 1992, pp. 349–361.

[13] A. H. Baker, R. D. Falgout, H. Gahvari, T. Gamblin, W. Gropp, K. E. Jordan, T. V. Kolev, M. Schulz, and U. M. Yang, "Preparing Algebraic Multigrid for Exascale," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-533076, March 2012.

[14] "*hypre* Reference Manual, Version 2.8.0b," https://computation.llnl.gov/casc/hypre/download/hypre-2.8.0b_ref_manual.pdf.

[15] D. E. Womble and B. C. Young, "A model and implementation of multigrid for massively parallel computers," *International Journal of High Speed Computing*, vol. 2, pp. 239–255, 1990.

[16] K. Nakajima, "New Strategy for Coarse Grid Solvers in Parallel Multigrid Methods using OpenMP/MPI Hybrid Programming Models," in *2012 International Workshop on Programming Models and Applications for Multicores and Manycores*, New Orleans, LA, February 2012, pp. 93–102.

[17] M. W. Gee, C. M. Siefert, J. J. Hu, R. S. Tuminaro, and M. G. Sala, "ML 5.0 Smoothed Aggregation User's Guide," Sandia National Laboratories, Tech. Rep. SAND2006-2649, February 2007.

[18] R. S. Sampath and G. Biros, "A parallel geometric multigrid method for finite elements on octree meshes," *SIAM Journal on Scientific Computing*, vol. 32, pp. 1361–1392, 2010.

[19] 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 *25th ACM International Conference on Supercomputing*, Tucson, AZ, June 2011, pp. 172–181.

[20] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP," in *41st International Conference on Parallel Processing*, Pittsburgh, PA, September 2012.

[21] ——, "Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned," in *3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, Salt Lake City, UT, November 2012.

[22] H. Gahvari, "Benchmarking Sparse Matrix-Vector Multiply," Master's thesis, University of California, Berkeley, December 2006.

[23] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

[24] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, December 2006.

[25] P. O. Frederickson and O. A. McBryan, "Parallel Superconvergent Multigrid," in *Multigrid Methods: Theory, Applications, and Supercomputing*, S. F. McCormick, Ed. Marcel Dekker, 1988, pp. 195–210.

[26] L. R. Matheson and R. E. Tarjan, "Parallelism in Multigrid Methods: How Much Is Too Much?" *International Journal of Parallel Programming*, vol. 24, pp. 397–432, 1996.

[27] A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang, "Challenges of Scaling Algebraic Multigrid across Modern Multicore Architectures," in *25th IEEE Parallel and Distributed Processing Symposium*, Anchorage, AK, May 2011, pp. 275–286.