LAWRENCE LIVERMORE NATIONAL LABORATORY

LLNL-JRNL-479591

Scaling hypre's multigrid solvers to 100,000 cores

A. H. Baker, R. D. Falgout, T. V. Kolev, U. M. Yang

April 8, 2011

High Performance Scientific Computing: Algorithms and Applications

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Scaling hypre's multigrid solvers to 100,000 cores

Allison H. Baker, Robert D. Falgout, Tzanio V. Kolev, and Ulrike Meier Yang

Abstract The *hypre* software library [15] is a collection of high performance preconditioners and solvers for large sparse linear systems of equations on massively parallel machines. This paper investigates the scaling properties of several of the popular multigrid solvers and system building interfaces in *hypre* on two modern parallel platforms. We present scaling results on over 100,000 cores and even solve a problem with over a trillion unknowns.

1 Introduction

The need to solve increasingly large, sparse linear systems of equations on parallel computers is ubiquitous in scientific computing. Such systems arise in the numerical simulation codes of a diverse range of phenomena, including stellar evolution, groundwater flow, fusion plasmas, explosions, fluid pressures in the human eye, and many more. Generally these systems are solved with iterative linear solvers, such as the conjugate gradient method, combined with suitable preconditioners, see e.g. [17, 19].

A particular challenge for parallel linear solver algorithms is scalability. An application code is scalable if it can use additional computational resources effectively. In particular, in this paper we focus on *weak scalability*, which requires that if the size of a problem and the number of cores are increased proportionally, the computing time should remain approximately the same. Unfortunately, in practice, as simulations grow to be more realistic and detailed, computing time may increase dramatically even when more cores are added to solve the problem. Recent machines

---

Allison Baker · Rob Falgout · Tzanio Kolev · Ulrike Yang
Lawrence Livermore National Laboratory, Center for Applied Scientific Computing, e-mail: {abaker,rfalgout,tzanio,umyang}@llnl.gov.
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-JRNL-479591).

with tens or even hundreds of thousands of cores offer both enormous computing possibilities and unprecedented challenges for achieving scalability.

The *hypre* library was developed with the specific goal of providing users with advanced parallel linear solvers and preconditioners that are scalable on massively parallel architectures. Scalable algorithms are essential for combating growing computing times. The library features parallel multigrid solvers for both structured and unstructured problems. Multigrid solvers are attractive for parallel computing because of their scalable convergence properties. In particular, if they are well-designed, then the computational cost depends linearly on the problem size, and increasingly larger problems can be solved on (proportionally) increasingly larger numbers of cores with approximately the same number of iterations to solution. This natural algorithmic scalability of the multigrid methods combined with the robust and efficient parallel algorithm implementations in *hypre* result in preconditioners that are well-suited for large numbers of cores.

The *hypre* library is a vital component of a broad array of application codes both at and outside of Lawrence Livermore National Laboratory (LLNL). For example, the library was downloaded more than 1800 times from 42 countries in 2010 alone, approaching nearly 10,000 total downloads from 70 countries since its first open source release in September of 2000. The scalability of its multigrid solvers has a large impact on many applications, particularly because simulation codes often spend the majority of their runtime in the linear solve.

The objective of this paper is to demonstrate the scalability of the most popular multigrid solvers in *hypre* on current supercomputers. We present scaling studies for conjugate gradient, preconditioned with the structured-grid solvers PFMG, SMG, SysPFMG, as well as the algebraic solver BoomerAMG, and the unstructured Maxwell solver AMS. Note that previous investigations beyond 100,000 cores focused only on the scalability of BoomerAMG on various architectures and can be found in [5, 2].

The paper is organized as follows. First we provide more details about the overall *hypre* library in Section 2 and the considered multigrid linear solvers in Section 3. Next, in Section 4, we specify the machines and the test problems we used in our experimental setup. We present and discuss the corresponding scalability results in Section 5, and we conclude by summarizing our findings in Section 6.

2 The *hypre* library

In this section we give a general overview of the *hypre* library. More detailed information can be found in the User's Manual available on the *hypre* web page [15].

2.1 Conceptual interfaces

We first discuss three of the so-called *conceptual interfaces* in *hypre*, that provide different mechanisms for describing a linear system on a parallel machine. These interfaces not only facilitate the use of the library, but they make it possible to provide linear solvers that take advantage of additional information about the application.

The **Structured Grid Interface** (`Struct`) is a stencil-based interface that is most appropriate for scalar finite-difference applications whose grids consist of unions of logically rectangular (sub)grids. The user defines the matrix and right-hand side in terms of the stencil and the grid coordinates. This geometric description, for example, allows the use of the PFMG solver, a parallel algebraic multigrid solver with geometric coarsening, described in more detail in the next section.

The **Semi-Structured Grid Interface** (`SStruct`) is essentially an extension of the Structured Grid Interface that can accommodate problems that are mostly structured, but have some unstructured features (e.g., block-structured, composite or overset grids). It can also accommodate multiple variables and variable types (e.g., cell-centered, edge-centered, etc.), which allows for the solution of more general problems. This interface requires the user to describe the problem in terms of structured grid *parts*, and then describe the relationship between the data in each part using either stencils or finite element stiffness matrices.

The **Linear-algebraic Interface** (`IJ`) is a standard linear-algebraic interface that requires that the users map the discretization of their equations into row-column entries in a matrix structure. Matrices are assumed to be distributed across P MPI tasks in contiguous blocks of rows. In each task, the matrix block is split into two components which are each stored in compressed sparse row (CSR) format. One component contains the coefficients that are local to the task, and the second, which is generally much smaller than the local one, contains the coefficients whose column indices point to rows located in other tasks. More details of the parallel matrix structure, called ParCSR, can be found in [10].

2.2 Solvers

The *hypre* library contains highly efficient and scalable specialized solvers as well as more general-purpose solvers that are well-suited for a variety of applications.

The specialized multigrid solvers use more than just the matrix to solve certain classes of problems, a distinct advantage provided by the conceptual interfaces. For example, the structured multigrid solvers SMG, PFMG, and SysPFMG all take advantage of the structure of the problem. As a result, these solves are typically more efficient and scalable than a general-purpose solver alternative. The SMG and PFMG solvers require the use of the `Struct` interface, and SysPFMG requires the `SStruct` interface.

For electromagnetic problems, *hypre* provides the unstructured Maxwell solver, AMS, which is the first provably scalable solver for definite electromagnetic prob-

lems on general unstructured meshes. The AMS solver requires matrix coefficients plus the discrete gradient matrix and the vertex coordinates which can be described with the `IJ` or `SStruct` interface.

For problems on arbitrary unstructured grids, *hypre* provides a robust parallel implementation of algebraic multigrid (AMG), called BoomerAMG. BoomerAMG can be used with any interface (currently not supported through `Struct`), as it only requires the matrix coefficient information.

The *hypre* library also provides common general-purpose iterative solvers, such as the GMRES and Conjugate Gradient (CG) methods. While these algorithms are not scalable as stand-alone solvers, they are particularly effective (and scalable) when used in combination with a scalable multigrid preconditioner.

2.3 Considerations for large-scale computing

Several features of the *hypre* library are required to efficiently solve very large problems on current supercomputers. Here we describe three of these features: support for 64-bit integers, scalable interface support for large numbers of MPI tasks, and the use of a hybrid programming model.

64-bit integer support has recently been added in *hypre*. This support is needed to solve problems in ParCSR format with more than 2 billion unknowns (previously a limitation due to 32-bit integers). To enable the 64-bit integer support, *hypre* must be configured with the `--enable-bigint` option. When this feature is turned on, the user must pass *hypre* integers of type `HYPRE_Int`, which is the 64-bit integer (usually a 'long long int' type in C). Note that this 64-bit integer option converts all integers to 64-bit, which does affect performance and increases memory use.

Scalable interfaces as well as solver algorithms are required for a code utilizing *hypre* to be scalable. When using one of *hypre*'s interfaces, the problem data is passed to *hypre* in its distributed form. However, to obtain a solution via a multigrid method or any other linear solver algorithm, MPI tasks need to obtain nearby data from other tasks. For a task to determine which tasks own the data that it needs, i.e. their communication partners or neighbors, some information regarding the global distribution of the data is required. Storing and querying a global description of the data, which is the information detailing which MPI task owns what data or the *global partition*, is either too costly or not possible when dealing with tens of thousands or more tasks. Therefore, to determine inter-task communication in a scalable manner, we developed new algorithms that employ an *assumed partition* to answer queries through a type of rendezvous algorithm, instead of storing global data descriptions. This strategy significantly reduces storage, communication, and computational costs for the solvers in *hypre* and improves scalability as shown in [4]. Note that this optimization requires configuring *hypre* with the `--no-global-partition` option and is most beneficial when using tens of thousands of tasks.

A **hybrid programming model** is used in *hypre*. While we have obtained good scaling results in the past using an MPI-only programming model, see e.g. [11],

with increasing numbers of cores per node on multicore architectures, the MPI-only model is expected to be increasingly insufficient due to the limited off-node bandwidth and decreasing amounts of memory per core. Therefore, in *hypre* we also employ a mixed or hybrid programming model which combines both MPI and the shared memory programming model OpenMP. The OpenMP code in *hypre* is largely used in loops and divides a loop among k threads into roughly k equal-sized portions. Therefore, basic matrix and vector operations, such as the matrix-vector multiply or dot product, are straightforward, but the use of OpenMP within other more complex parts of the multigrid solver algorithm, such as in parts of the setup phase (described in Section 3), may be non-trivial. The right choice for hybrid MPI/OpenMP partitioning in terms of obtaining optimal performance is dependent on the specific target machine's node architecture, interconnect, and operating system capabilities [5]. See [5] or [2] for more discussion on the performance of BoomerAMG with a hybrid programming model.

3 The multigrid solvers

As mentioned in Section 1, multigrid solvers are algorithmically scalable, meaning that they require O(N) computations to solve a linear system with N variables. This desirable property is obtained by cleverly utilizing a sequence of smaller (or coarser) grids, which are computationally cheaper to compute on than the original (finest) grid. A multigrid method works as follows. At each grid level, a smoother is applied to the system, which serves to resolve the high-frequency error on that level. The improved guess is then transferred to a smaller, or coarser, grid, the smoother is applied again, and the process continues. The coarsest level is generally chosen to be a size that is reasonable to solve directly, and the goal is to eliminate a significant part of the error by the time this coarsest level is reached. The solution to the coarse grid solve is then interpolated, level by level, back up to the finest grid level, applying the smoother again at each level. A simple cycle down and up the grid is referred to as a V-cycle. To obtain good convergence, the smoother and the coarse-grid correction process must complement each other to remove all components of the error.

A multigrid method has two phases: the setup phase and the solve phase. The setup phase consists of defining the coarse grids, interpolation operators, and coarse-grid operators for each of the coarse-grid levels. The solve phase consists of performing the multilevel cycles (i.e., iterations) until the desired convergence is obtained. In the scaling studies, we often time the setup phase and solve phase separately. Note that while multigrid methods may be used as linear solvers, they are more typically used as preconditioners for Krylov methods such as GMRES or conjugate gradient.

The challenge for multigrid methods on supercomputers is turning an efficient serial algorithm into a robust and scalable parallel algorithm. Good numerical properties need to be preserved when making algorithmic changes needed for paral-

lelism. This non-trivial task affects all aspects of a multigrid algorithm, including coarsening, interpolation, and smoothing.

In the remainder of this section, we provide more details for the most commonly-used solvers in *hypre* for which we perform our scaling study.

3.1 PFMG, SMG, and SysPFMG

PFMG [1, 12] is a semicoarsening multigrid method for solving scalar diffusion equations on logically rectangular grids discretized with up to 9-point stencils in 2D and up to 27-point stencils in 3D. It is effective for problems with variable coefficients and anisotropies that are uniform and grid-aligned throughout the domain. The solver automatically determines the "best" direction of semicoarsening, but the user may also control this manually. Interpolation is determined algebraically. The coarse-grid operators are also formed algebraically, either by Galerkin or by the non-Galerkin process described in [1]. The latter is available only for 5-point (2D) and 7-point (3D) problems, but maintains these stencil patterns on all coarse grids, reducing cost and improving performance. Relaxation options are either weighted Jacobi or red/black Gauss-Seidel. The solver can also be run in a mode that skips relaxation on certain grid levels when the problem is (or becomes) isotropic, further reducing cost and increasing performance.

PFMG also has two constant-coefficient modes, one where the entire stencil is constant throughout the domain, and another where the diagonal is allowed to vary. Both modes require significantly less storage and can also be somewhat faster than the full variable-coefficient solver, depending on the machine. The variable diagonal case is the most effective and flexible of the two modes. The non-Galerkin options here are similar to the variable case, but maintain the constant-coefficient format on all grid levels.

SMG [20, 7, 9, 12] is also a semicoarsening multigrid method for solving scalar diffusion equations on logically rectangular grids. It is more robust than PFMG, especially when the equations exhibit anisotropies that vary in either strength or direction across the domain, but it is also much more expensive per iteration. SMG coarsens in the z direction and uses a plane smoother. The xy plane solves in the smoother are approximated by employing one cycle of a 2D SMG method, which in turn coarsens in y and uses x-line smoothing. The plane and line solves are also used to define interpolation, and the solver uses Galerkin coarse-grid operators.

SysPFMG is a generalization of PFMG for solving systems of elliptic PDEs. Interpolation is defined only within the same variable using the same approach as PFMG, and the coarse-grid operators are Galerkin. The smoother is of nodal type and solves all variables at a given point simultaneously.

3.2 BoomerAMG

BoomerAMG [14] is the unstructured algebraic multigrid (AMG) solver in *hypre*. AMG is a particular type of multigrid method that is unique because it does not require an explicit grid geometry. This attribute greatly increases the types of problems that can be solved with multigrid because often the actual grid information may not be known or the grid may be highly unstructured. Therefore, in AMG the "grid" is simply the set of variables, and the coarsening and interpolation processes are determined entirely based on the entries of the matrix. For this reason, AMG is a rather complex algorithm, and it is challenging to design parallel coarsening and interpolation algorithms that combine good convergence, low computational complexity as well as low memory requirements. See [25], for example, for an overview of parallel AMG. Note that the AMG setup phase can be costly, compared to that of a geometric multigrid method. Classical coarsening schemes [8, 18] have led to slow coarsening, especially for 3D problems, resulting in large computational complexities per V-cycle, increased memory requirements and decreased scalability. In order to achieve scalable performance, one needs to use reduced complexity coarsening methods, such as HMIS and PMIS [23], which require distance-two interpolation operators, such as extended+i interpolation [22], or even more aggressive coarsening schemes, which need interpolation with an even longer range [24, 26]. The parallel implementation of long range interpolation schemes generally involves much more complicated and costly communication patterns than nearest neighbor interpolation. Additionally, communication requirements on coarser grid levels can become more costly as the stencil size typically increases with coarsening, which results in MPI tasks having many more neighbors (see, e.g., [13] for a discussion). The AMG solve phase consists largely of matrix-vector multiplies and the application of a (typically) inexpensive smoother, such as hybrid (symmetric) Gauss-Seidel, which applies sequential (symmetric) Gauss-Seidel locally on each core and uses delayed updates across cores. Note that hybrid smoothers depend on the number of cores as well as the distribution of data across tasks, and therefore one cannot expect to achieve exactly the same results or the same number of iterations when using different configurations. However, the number of iterations required to converge to the desired tolerance should be fairly close.

3.3 AMS

The Auxiliary-space Maxwell Solver (AMS) is an algebraic solver for electromagnetic diffusion problems discretized with Nedelec (edge) finite elements. AMS can be viewed as an AMG-type method with multiple coarse spaces, in each of which a BoomerAMG V-cycle is applied to a variationally constructed scalar/vector nodal problems. Unlike BoomerAMG, AMS requires some fine-grid information besides the matrix: the coordinates of the vertices and the list of edges in terms of their vertices (the so-called discrete gradient matrix), which allows it to be scalable and

robust with respect to jumps in material coefficients. More details about the AMS algorithm and its performance can be found in [16].

4 Experimental setup

For the results in this paper, we used version 2.7.1a of the *hypre* software library. In this section, we describe the machines and test runs used in our scaling studies.

4.1 Machine descriptions

The **Dawn** machine is a Blue Gene/P system at LLNL. This system consists of 36,864 compute nodes, and each node contains a quad-core 850 MHz PowerPC 450 processor, bringing the total number of cores to 147,456. All four cores on a node have a common and shared access to the complete main memory of 4.0 GB. This guarantees uniform memory access (UMA) characteristics. All nodes are connected by a 3D torus network. We compile our code using IBM's C and OpenMP/C compilers and use IBM's MPICH2-based MPI implementation.

The **Hera** machine is a multicore/multi-socket Linux cluster at LLNL with 864 compute nodes connected by Infiniband network. Each compute node has four sockets, each with an AMD Quadcore (8356) 2.3 GHz processor. Each processor has its own memory controller and is attached to a quarter of the node's 32 GB memory. While a core can access any memory location, the non-uniform memory access (NUMA) times depend on the location of the memory. Each node runs CHAOS 4, a high-performance computing Linux variant based on Redhat Enterprise Linux. Our code is compiled using Intel's C and OpenMP/C compiler and uses MVAPICH for the MPI implementation.

4.2 Test runs

Because we are presenting a scaling study, and not a convergence study, we chose relatively simple problems from a mathematical point of view. However, these problems are sufficient for revealing issues with scaling performance.

3D Laplace: A 3D Laplace equation with Dirichlet boundary conditions, discretized with seven-point finite differences on a uniform Cartesian grid.

3D Laplace System: A system of two 3D Laplace equations as above, with weak inter-variable coupling at each grid point. Each Laplacian stencil had a coefficient of 6 on the diagonal and -1 on the off-diagonals, and the inter-variable coupling coefficient was 10^{-5}.

3D Electromagnetic Diffusion: A simple 3D electromagnetic diffusion problem posed on a structured grid of the unit cube. The problem has unit conductivity and homogeneous Dirichlet boundary conditions and corresponds to the example code **`ex15`** from the *hypre* distribution.

We use the notation PFMG-*n*, CPFMG-*n*, SMG-*n*, SysPFMG, AMG-*n*, and AMS-*n* to represent conjugate gradient solvers preconditioned respectively by PFMG, constant-coefficient PFMG, SMG, SysPFMG, BoomerAMG, and AMS, where n signifies a local grid of dimension $n \times n \times n$ on each core. We use two different parameter choices for PFMG (and CPFMG), and denote them by appending a '-1' or '-2' to the name as follows:

- PFMG-*n*-1 - Weighted Jacobi smoother and Galerkin coarse-grid operators;
- PFMG-*n*-2 - Red/black GS smoother, non-Galerkin coarse-grid operators, and relaxation skipping.

For AMG, we use aggressive coarsening with multipass interpolation on the finest level, and HMIS coarsening with extended+i interpolation truncated to at most 4 elements per row on the coarser levels. The coarsest system is solved using Gaussian elimination. The smoother is one iteration of symmetric hybrid Gauss-Seidel. For AMS, we use the default parameters of **`ex15`** plus the ℓ_1^*-GS smoother from [3] (option **`-rlx 4`**).

For PFMG, CPFMG, SMG, and AMG, we solve the 3D Laplace problem with global grid size $N = np \times np \times np$, where $P = p^3$ is the total number of cores. For SysPFMG, we solve the 3D Laplace System problem with a local grid size of 40^3. For AMS, we solve the 3D Electromagnetic Diffusion problem, where here n^3 specifies the local number of finite elements on each core.

In the scaling studies, P ranged from 64 to 125,000 on Dawn and 64 to 4096 on Hera, with specific values for p given as follows:

- $p = 4, 8, 12, 16, 20, 24, 28, 32, 36, 40, 44, 48, 50$ on Dawn; and
- $p = 4, 8, 12, 16$ on Hera.

For the hybrid MPI/OpenMP runs, the same global problems were run, but they were configured as in the following table.

Machine	Threads	Problem size per MPI task	Number of MPI tasks
Dawn (smp)	4	$2n \times 2n \times n$	$p/2 \times p/2 \times p$
Hera (4x4)	4	$n \times 2n \times 2n$	$p \times p/2 \times p/2$
Hera (1x16)	16	$2n \times 2n \times 4n$	$p/2 \times p/2 \times p/4$

5 Scaling Studies

In this section, we present the scaling results. We first comment on the solver performance and conclude the section with comments on the times spent to set up the problems using *hypre*'s interfaces. For all solvers, iterations were stopped when the

L^2 norm of the relative residual was smaller than 10^{-6}. The number of iterations are listed in Table 1.

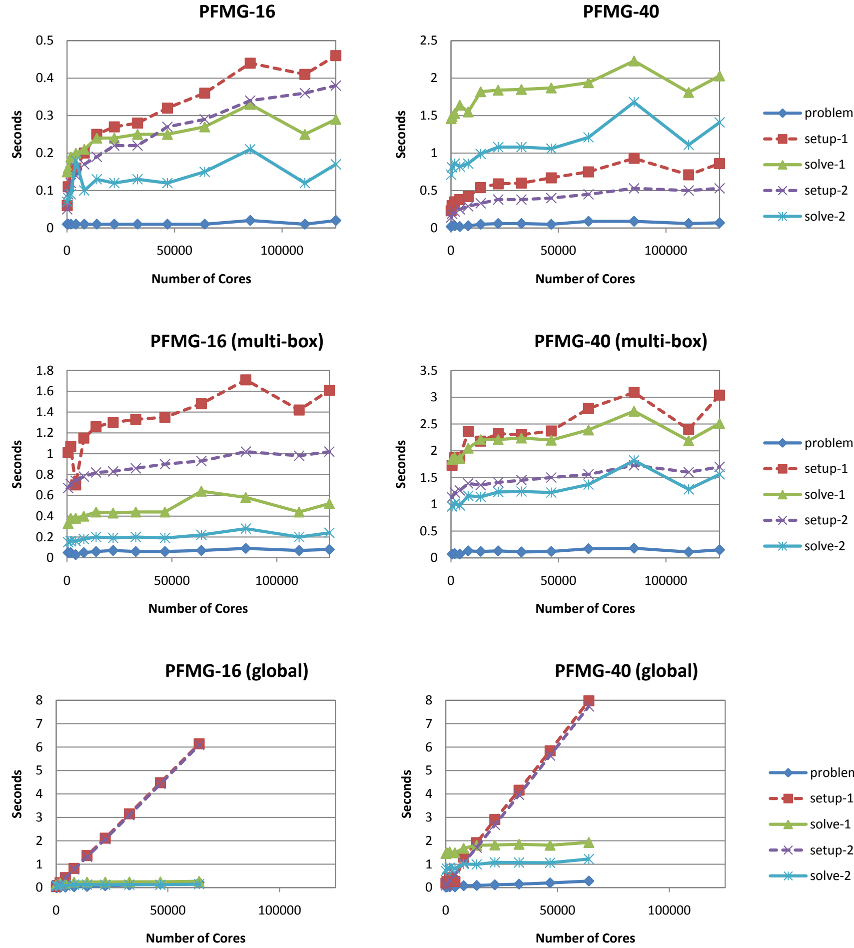

Fig. 1 PFMG results on Dawn for two different local problem sizes, 16^3 (left) and 40^3 (right). The top and bottom rows use one box to describe the local problem, while the second row splits the local problem into $64 = 4 \times 4 \times 4$ boxes. The top two rows use the assumed partition algorithm and the bottom row uses the global partition. Setup and solve phase times are given for two different parameter choices.

In Figure 1, we give results for PFMG using MPI only. Since communication latency speeds are several orders of magnitude slower than MFLOP speeds on todays architectures (Dawn included), communication costs tend to dominate for the smaller PFMG-16 problems. Because of this, the setup phase is slower than the

solve phase, due primarily to the assumed partition and global partition components of the code. The global partition requires $O(P\log P)$ communications in the setup phase, while the current implementation of the assumed partition requires $O((\log P)^2)$ communications (the latter is easier to see in Figure 2). It should be possible to reduce the communication overhead of the assumed partition algorithm in the setup phase to $O(\log P)$ by implementing a feature for coarsening the *box manager* in *hypre* (the box manager serves the role of the *distributed directory* in [4]). For PFMG-40 with the assumed partition, the setup phase is cheaper than the solve phase for the single box case, and roughly the same cost for the multi-box case. For both multi-box cases, describing the data with 64 boxes leads to additional overhead in all phases, but the effect is more pronounced for the setup phase. The problem setup uses the `Struct` interface and is the least expensive component.

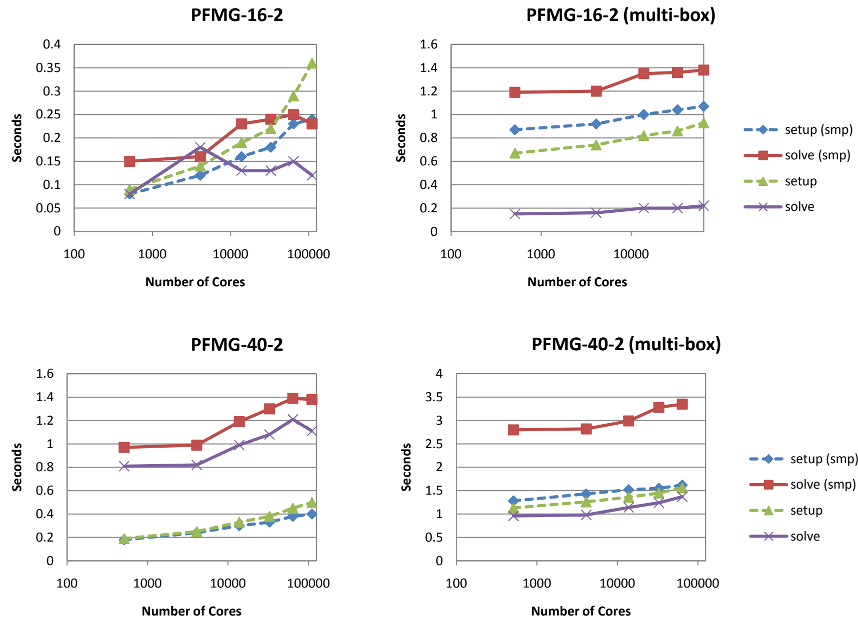

Fig. 2 PFMG results on Dawn comparing MPI and hybrid MPI/OpenMP for parameter choice 2 and two different local problem sizes, 16^3 (top) and 40^3 (bottom). The left column uses one box to describe the local problem and the right column uses 64. The assumed partition is used in all cases, and the results are plotted on a log scale.

In Figure 2, we give results for PFMG, comparing MPI to hybrid MPI/OpenMP. With the exception of the setup phase in the single box cases, the hybrid results are slower than the MPI-only results due most likely to unnecessary thread overhead generated by the OpenMP compiler (the Hera results in Figure 5 show that our hybrid model can be faster than pure MPI). In the single box cases, the $O((\log P)^2)$

communications in the assumed partition algorithm dominates the time in the setup phase, so the pure MPI runs are slower than the hybrid runs due to the larger number of boxes to manage. This scaling trend is especially apparent in the PFMG-16-2 plot. For the multi-box cases, thread overhead is multiplied by a factor of 64 (each threaded loop becomes 64 threaded loops) and becomes the dominant cost.

The constant-coefficient CPFMG solver saves significantly on memory, but it was only slightly faster than the variable-coefficient PFMG case with nearly identical scaling results. The memory savings allowed us to run CPFMG-200-2 to solve a 1.049 trillion unknown problem on 131,072 cores ($64 \times 64 \times 32$) in 11 iterations and 83.03 seconds. Problem setup took 0.84 seconds and solver setup took 1.10 seconds.

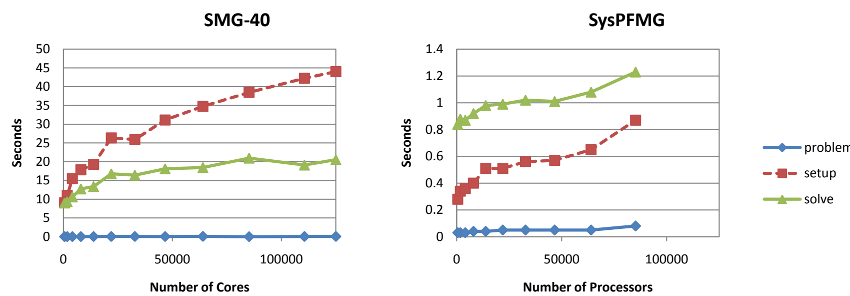

Fig. 3 SMG-40 and SysPFMG results on Dawn using the assumed partition algorithm.

In Figure 3, we show results for SMG-40 and SysPFMG. We see that SMG is a much more expensive solver than PFMG, albeit more robust. We also see that the setup phase scales quite poorly, likely due to the $O((\log P)^3)$ number of coarse grids and our current approach for building a box manager on each grid level. See [12] for more discussion on expected scaling behavior of SMG (and PFMG).

For SysPFMG, we see that the scaling of both the setup and solve phases are not flat, even though the iteration counts in Table 1 are constant. However, this increase is the same increase seen for PFMG-40 at 85,184 cores, so we expect the times to similarly decrease at larger core counts. The problem setup uses the `sstruct` interface and is the least expensive component.

Figure 4 presents the performance of BoomerAMG-CG on Dawn. The two top figures show the performance obtained when using MPI only and the effect of using 32-bit integers versus 64-bit integers. The 32-bit version could only be used to up to 32,768 cores for the 40^3 Laplace problem. We also include results that were obtained using the Sequoia benchmark AMG [21], which is a stand-alone version of BoomerAMG, in which only those integers that are required to have a longer format were converted, an approach that was unfortunately too work extensive to be applied to all of *hypre*. The benchmark results are very close to the 32-bit integer results.

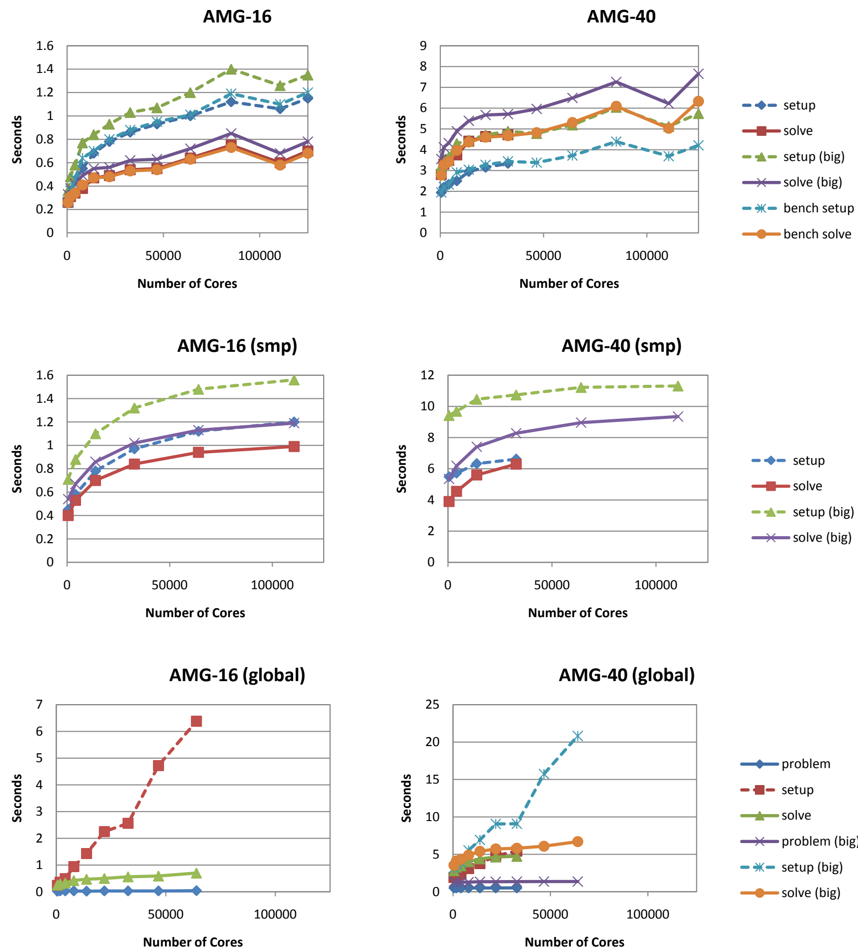

Fig. 4 AMG results on Dawn for two different local problem sizes, 16^3 (left) and 40^3 (right). The top figures show setup and solve times for the 32-bit version of *hypre*, the 64-bit version (big), and the AMG Sequoia benchmark (bench), using MPI only and assumed partitioning. The middle figures show times using the hybrid programming model MPI/OpenMP and assumed partitioning and include the problem setup times. The bottom figures use MPI only and global partitioning.

The two bottom figures show the effect of using a global partition when setting up the communication pattern and its linear dependence on the number of MPI tasks. The problem setup uses the IJ interface and takes very little time compared to solve and setup.

The middle figures show timings obtained using a hybrid OpenMP/MPI programming model with one MPI task per node using 4 OpenMP threads on Dawn. On Dawn, the use of OpenMP for AMG leads to larger run times than using MPI

only and is therefore not recommended. While the solve phase of BoomerAMG is completely threaded, portions of it, like the multiplication of the transpose of the interpolation operator with a vector cannot be implemented as efficiently using OpenMP as MPI with our current data structure. Also, portions of the setup phase, like the coarsening and part of the interpolation, are currently not threaded, which also negatively affects the performance when using OpenMP.

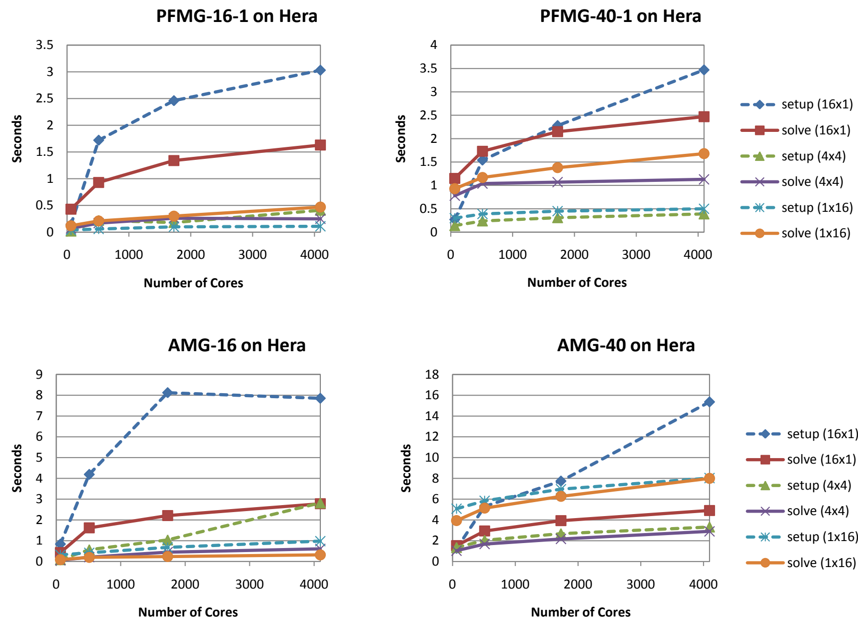

Fig. 5 Setup and solve times on Hera for PFMG and AMG, using an MPI only programming model (16x1), and two hybrid MPI/OpenMP programming models using 4 MPI tasks with 4 OpenMP threads each (4x4) and using 1 MPI task with 16 OpenMP threads (1x16) per node.

On Hera, the use of a hybrid MPI/OpenMP versus an MPI only programming model yields different results, see Figure 5. Since at most 4096 cores were available to us, we used the global partition. We compare the MPI only implementations with 16 MPI tasks per node (16x1) to a hybrid model using 4 MPI tasks with 4 OpenMP threads each (4x4), which is best adapted to the architecture, and a hybrid model using 1 MPI task with 16 OpenMP threads per node (1x16). For the smaller problem with 16^3 unknowns per core, we obtain the worst times using MPI only, followed by the 4x4 hybrid model. The best times are achieved using the 1x16 hybrid model, which requires the least amount of communication, since communication is very expensive compared to computation on Hera and communication dominates over computation for smaller problem sizes. The picture changes for the larger problem with 40^3 unknowns per core. Now the overall best times are achieved using the 4x4

hybrid model, which has less communication overhead than the MPI only model, but is not plagued by NUMA effects like the 1x16 hybrid model, where all memory is located in the first memory module, causing large access times and contention for the remaining 12 cores, see also [6].

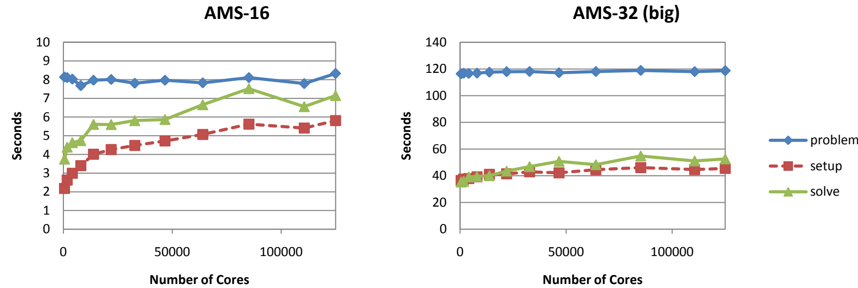

Fig. 6 AMS results on Dawn for two different local problem sizes, 16^3 (left) and 32^3 (right). Shown are the problem generation time and the solver setup and solve times. The AMS-32 problem uses the 64-bit version of *hypre*.

In Figure 6 we show the scalability results for CG with AMS preconditioner applied to the constant coefficient electromagnetic diffusion problem described in the previous section. We consider a coarse (AMS-16) and a 64-bit fine problem (AMS-32) with parallel setups corresponding to 16^3 and 32^3 elements per core. The respective largest problem sizes were around 1.5 and 12 billion on $125,000$ cores. Both cases were run with the assumed partition version of *hypre*.

Though not perfect, both the AMS setup and solve times in Figure 6 show good parallel scalability, especially in the case of AMS-32 where the larger amount of local computations offsets better the communications cost. The slight growth in the solve times can be partially explained by the fact that the number of AMS-CG iterations varies between 9 to 12, for AMS-16, and 10 to 14, for AMS-32.

In Figure 7, we show timings for one iteration (cycle) of a few solvers considered earlier. This removes the effect of the iteration counts given in Table 1 on the overall solve times. We see that the cycle time for AMG-40 is only slightly larger than PFMG-40-1, even though it is a fully unstructured code. This is mainly due to the dominant cost of communication, but it also suggests that improvements may be possible in the PFMG computational kernels where we should be able to take better advantage of structure. PFMG-40-2 is faster than PFMG-40-1 because it maintains 7-pt operators on all grid levels, reducing both communication and computation costs. AMS-16 is the slowest even though the grid is smaller, because it is solving a much harder problem that involves essentially four AMG V-cycle plus additional smoothing. It is interesting to note that all of the curves have almost the same shape, with a slight increase at 85,184 cores. This is most likely due to a poor mapping

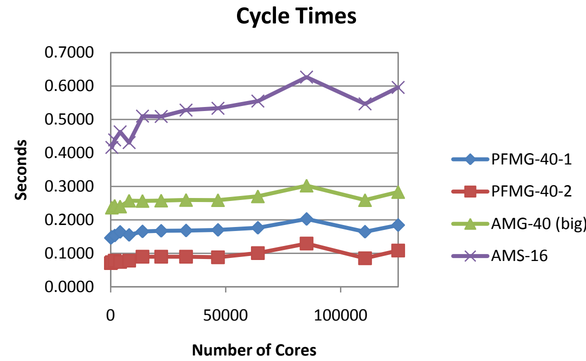

Fig. 7 Cycle times on Dawn for CG with various multigrid preconditioners.

of the problem data to the hardware, which resulted in more costly long-distance communication.

We now comment on the performance improvements that can be achieved when using the struct interface and PFMG or CPFMG over the IJ interface and BoomerAMG for suitable structured problems. For the smaller Laplace problem, PFMG-16-1 is about 2.5 times faster than the 32-version of AMG-16 and the AMG benchmark, and 3 times faster than the 64-bit version. The non-Galerkin version, PFMG-16-2, is about 3 to 4 times faster than AMG. For the larger problem, PFMG-40-2 is about 7 times faster than the 64-bit version of AMG and about 5 times faster than the benchmark.

Finally, we comment on the times it takes to set up the problems via *hypre*'s interfaces. Figures 1, 3, and 4 showed that for many problems, the setup takes very little time compared to setup and solve times of the solvers. Note that the times for the problem setup via the IJ interface in Figure 4 includes setting up the problem directly in the ParCSR data structure and then using the information to set up the matrix using the IJ data interface. Using the IJ interface directly for the problem setup takes only about half as much time.

The problem generation time in Figure 6 corresponds to the assembly of the edge element Maxwell stiffness matrix and load vector, as well as the computation of the rectangular discrete gradient matrix and the nodal vertex coordinates needed for AMS. These are all done with the `SStruct` interface, using its finite element functionality for the stiffness matrix and load vector. Overall, the problem generation time scales very well. Though its magnitude may appear somewhat large, one should account that it also includes the (redundant) computation of all local stiffness matrices, and the penalty from the use of the 64-bit version of *hypre* in AMS-32.

In Figure 8, we give results for the `SStruct` interface. The BCube stencil-based results involve one cell-centered variable and two parts connected by the `GridSetNeighborPart()` routine, while the BCube FEM-based results use a node-

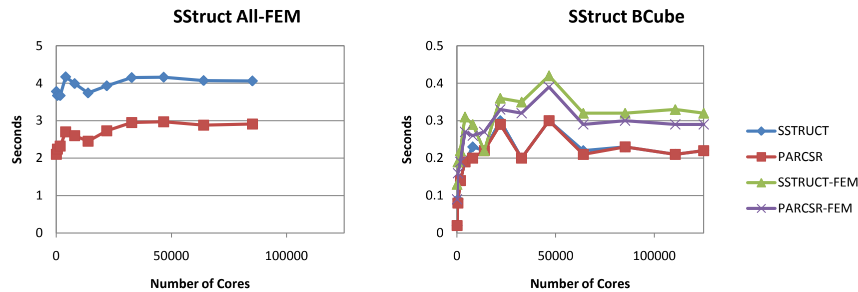

Fig. 8 Scaling results on Dawn for the **`SStruct`** interface, comparing the cost of setting up the **`SSTRUCT`** and **`PARCSR`** object types, and the cost of using a stencil-based or FEM-based approach.

centered variable. The All-FEM results involve 7 different variable types (all of the supported types except for cell-centered) and three parts.

For the BCube example, we see that building the **`SSTRUCT`** and **`PARCSR`** objects takes about the same time for the stencil-based case, with **`SSTRUCT`** taking slightly longer in the FEM-based case. This is probably due to the extra communication required to assemble the structured part of the matrix for non-cell-centered variable types along part boundaries. This difference is even more pronounced in the All-FEM example. We also see from the BCube example that the FEM-based approach is more expensive than the stencil-based approach. This is due to the additional cost of assembling the matrix and also partly due to the fact that the FEM interface does not currently support assembling a box of stiffness matrices in one call to reduce overhead.

6 Concluding Remarks

We performed a scaling study of the interfaces and various multigrid solvers of the *hypre* library. The results show overall good scaling on the IBM BG/P machine Dawn at LLNL. We demonstrated that in order to achieve scalability, it is crucial to use an assumed partition instead of a global partition. On Dawn, using an MPI only programming model gave generally better timings than using a hybrid MPI/OpenMP programming model. However the use of MPI/OpenMP showed improved times on the multicore/multi-socket Linux cluster Hera at LLNL. In the future, we plan to investigate how to reduce communication, to improve the use of threads, as well as to employ more suitable data structures. Our goal is to achieve good performance on exascale computers, which are expected to have millions of cores with memories that are orders of magnitudes smaller than on current machines.

P	PFMG-				CPFMG-				SMG-40	SysPFMG
	16-1	16-2	40-1	40-2	16-1	16-2	40-1	40-2		
64	9	9	10	10	9	9	9	9	5	9
512	9	10	10	11	9	9	10	10	5	9
1728	10	10	10	11	9	9	10	10	5	9
4096	10	10	10	11	10	9	10	10	5	9
8000	10	11	10	11	10	10	10	10	6	9
13824	10	11	11	11	10	10	10	10	6	9
21952	10	11	11	12	10	10	10	11	6	9
32768	10	11	11	12	10	10	10	10	6	9
46656	10	11	11	12	10	10	10	11	6	9
64000	10	11	11	12	10	10	10	10	6	9
85184	10	11	11	13	10	10	10	10	6	9
110592	10	11	11	13	10	10	10	11	6	
125000	10	11	11	13	10	10	10	12	6	

P	AMG-				AMS-	
	16	16 (smp)	40	40 (smp)	16	32
64	12		13			
512	14	14	15	16	9	10
1728	15		17		10	10
4096	15	15	18	18	10	11
8000	16		19		11	11
13824	17	16	21	20	11	11
21952	17		22		11	12
32768	18	17	22	22	11	13
46656	18		23		11	14
64000	19	18	24	23	12	13
85184	19		24		12	14
110592	19	19	24	24	12	14
125000	19		27		12	14

P	AMG-16			AMG-40		
	(16x1)	(4x4)	(1x16)	(16x1)	(4x4)	(1x16)
64	12	12	12	13	13	14
512	14	13	14	15	16	16
1728	15	14	14	17	17	18
4096	15	15	15	18	19	20

Table 1 Iteration counts on Dawn (top two tables) and Hera (bottom table). For all solvers, iterations were stopped when the relative residual was smaller than 10^{-6}.

References

1. S. F. Ashby and R. D. Falgout. A parallel multigrid preconditioned conjugate gradient algorithm for groundwater flow simulations. *Nuclear Science and Engineering*, 124(1):145–159, September 1996. UCRL-JC-122359.
2. A. Baker, R. Falgout, T. Gamblin, T. Kolev, M. Schulz, and U. M. Yang. Scaling algebraic multigrid solvers: On the road to exascale. In *Proceedings of Competence in High Performance Computing (CiHPC 2010)*, 2011. To appear. Also available as LLNL Tech. Report

LLNL-PROC-463941.
3. A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang. Multigrid smoothers for ultra-parallel computing. 2010. (submitted). Also available as a Lawrence Livermore National Laboratory technical report LLNL-JRNL-435315.
4. A. H. Baker, R. D. Falgout, and U. M. Yang. An assumed partition algorithm for determining processor inter-communication. *Parallel Computing*, 32(5–6):394–414, 2006. UCRL-JRNL-215757.
5. A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 2011. LLNL-CONF-458074.
6. A. H. Baker, M. Schulz, and U. M. Yang. On the performance of an algebraic multigrid solver on multicore clusters. In J.M.L.M. Palma et al., editor, *VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 102–115. Springer-Verlag, 2011. http://vecpar.fe.up.pt/2010/papers/24.php.
7. C. Baldwin, P. N. Brown, R. D. Falgout, J. Jones, and F. Graziani. Iterative linear solvers in a 2d radiation-hydrodynamics code: methods and performance. *J. Comput. Phys.*, 154:1–40, 1999. UCRL-JC-130933.
8. A. Brandt, S. F. McCormick, and J. W. Ruge. Algebraic multigrid (AMG) for sparse matrix equations. In D. J. Evans, editor, *Sparsity and Its Applications*. Cambridge University Press, Cambridge, 1984.
9. P. N. Brown, R. D. Falgout, and J. E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Comput.*, 21(5):1823–1834, 2000. Special issue on the Fifth Copper Mountain Conference on Iterative Methods. UCRL-JC-130720.
10. R. Falgout, J. Jones, and U. M. Yang. Pursuing scalability for hypre's conceptual interfaces. *ACM ToMS*, 31:326–350, 2005.
11. R. D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Engineering*, 8(6):24–33, 2006. Special issue on Multigrid Computing. UCRL-JRNL-220851.
12. R. D. Falgout and J. E. Jones. Multigrid on massively parallel architectures. In E. Dick, K. Riemslagh, and J. Vierendeels, editors, *Multigrid Methods VI*, volume 14 of *Lecture Notes in Computational Science and Engineering*, pages 101–107. Springer-Verlag, 2000. Proc. of the Sixth European Multigrid Conference held in Gent, Belgium, September 27-30, 1999. UCRL-JC-133948.
13. 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 *Proceedings of the 25th International Conference on Supercomputing (ICS 2011)*, 2011. To appear. Also available as LLNL Tech. Report LLNL-CONF-473462.
14. V. Henson and U. Yang. BoomerAMG: A parallel algebraic multigrid solver and preconditioner. *Appl. Numer. Math.*, 41:155–177, 2002.
15. *hypre*: High performance preconditioners. http://www.llnl.gov/CASC/hypre/.
16. T. Kolev and P. Vassilevski. Parallel auxiliary space AMG for H(curl) problems. *J. Comput. Math.*, 27:604–623, 2009. Special issue on Adaptive and Multilevel Methods in Electromagnetics. Also available as a Lawrence Livermore National Laboratory technical report UCRL-JRNL-237306.
17. U. Meier and A. Sameh. The behavior of conjugate gradient algorithms on a multivector processor with a hierarchical memory. *J. Comp. Appl. Math.*, 24:13–32, 1988.
18. J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, volume 3 of *Frontiers in Applied Mathematics*, pages 73–130. SIAM, Philadelphia, PA, 1987.
19. Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2nd edition, 2003.
20. S. Schaffer. A semi-coarsening multigrid method for elliptic partial differential equations with highly discontinuous and anisotropic coefficients. *SIAM J. Sci. Comput.*, 20(1):228–242, 1998.
21. Sequoia. ASC sequoia benchmark codes. https://asc.llnl.gov/sequoia/benchmarks/.

22. H. D. Sterck, R. D. Falgout, J. W. Nolting, and U. M. Yang. Distance-two interpolation for parallel algebraic multigrid. *Numer. Linear Algebra Appl.*, 15(2–3):115–139, 2008. Special issue on Multigrid Methods. UCRL-JRNL-230844.
23. H. D. Sterck, U. M. Yang, and J. J. Heys. Reducing complexity in parallel algebraic multigrid preconditioners. *SIAM J. Matrix Anal. Appl.*, 27(4):1019–1039, 2006.
24. K. Stüben. Algebraic multigrid (AMG): an introduction with applications. In U. Hackbusch, C. Oosterlee, and A. Schueller, editors, *Multigrid*. Academic Press, 2000.
25. U. Yang. Parallel algebraic multigrid methods - high performance preconditioners. In A. Bruaset and A. Tveito, editors, *Numerical Solution of Partial Differential Equations on Parallel Computers*, volume 51, pages 209–236. Springer-Verlag, 2006.
26. U. M. Yang. On long range interpolation operators for aggressive coarsening. *Numer. Linear Algebra Appl.*, 17:453–472, 2010. LLNL-JRNL-417371.