Algebraic Multigrid on a Dragonfly Network: First Experiences on a Cray XC30

Hormozd Gahvari[1](✉), William Gropp[2], Kirk E. Jordan[3], Martin Schulz[1], and Ulrike Meier Yang[1]

[1] Lawrence Livermore National Laboratory, Livermore, CA 94551, USA
{gahvari1,schulzm,umyang}@llnl.gov

[2] University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
wgropp@illinois.edu

[3] IBM TJ Watson Research Center, Cambridge, MA 02142, USA
kjordan@us.ibm.com

Abstract. The Cray XC30 represents the first appearance of the dragonfly interconnect topology in a product from a major HPC vendor. The question of how well applications perform on such a machine naturally arises. We consider the performance of an algebraic multigrid solver on an XC30 and develop a performance model for its solve cycle. We use this model to both analyze its performance and guide data redistribution at runtime aimed at improving it by trading messages for increased computation. The performance modeling results demonstrate the ability of the dragonfly interconnect to avoid network contention, but speedups when using the redistribution scheme were enough to raise questions about the ability of the dragonfly topology to handle very communication-intensive applications.

1 Introduction

The network topology of an HPC system has a critical impact on the performance of parallel applications. In recent years, vendors have experimented with a wide range of topologies. A topology that has found wide interest is the dragonfly topology [18]. Introduced several years ago, it has seen its first major deployment in the Cray XC30. As more XC30s and other machines that make use of dragonfly interconnects are deployed, the question of application performance on these machines becomes paramount. How suited is the dragonfly topology for particular applications? What are its advantages and disadvantages? What are its future prospects as machines get even larger?

This paper examines one application, algebraic multigrid (AMG), on an XC30, to see how well it performs on this topology and get a first look at potential hazards it and other applications would face on a dragonfly machine. AMG is a popular solver for large, sparse linear systems of equations with many scientific and engineering applications. It is very attractive for HPC owing to ideal computational complexity, but faces challenges on emerging parallel machines [3,4] that served as motivation for a recent in-depth study [11] into its performance

S.A. Jarvis et al. (Eds.): PMBS 2014, LNCS 8966, pp. 3–23, 2015.
DOI: 10.1007/978-3-319-17248-4_1

and ways to improve it. We present results from that study on modeling the performance of the AMG solve cycle on an XC30 and using the performance model to improve its performance on that architecture.

Our specific contributions are as follows:

- We successfully extend a performance model that previously covered fat-tree and torus interconnects to a dragonfly interconnect.
- We use that model at runtime to guide data redistribution within the AMG solve cycle to improve its performance on a dragonfly machine.
- We point out an important hazard faced by the dragonfly interconnect in a real-world scenario.

The model predicts cycle times to accuracies mostly between 85 and 93 percent in our experiments, and covers both all-MPI and hybrid MPI/OpenMP programming models. The data redistribution involves having processes combine data, trading messages they would send amongst themselves for increased computation. Resulting speedups range from modest to over 2x overall, with the large speedups occurring during the communication-heavy setup phase of AMG or when solving a communication-intense linear elasticity problem. This occurs despite the model rating the XC30 interconnect as being effective overall at avoiding network contention, leading to questions about the ability of the dragonfly interconnect when tasked with handling a large number of messages.

2 Dragonfly Networks

The general principle behind dragonfly interconnects is to keep the minimum hop distance low like a fat-tree, while also providing high bandwidth between nodes and low network contention at less cost [18]. This is accomplished through a generalized two-level design. The core of the network is formed by a number of groups of routers, with each group connected by optical cables to every other group. The routers in each individual group have their own specific topology. This is diagrammed in Fig. 1.

2.1 Implementation on the Cray XC30

The dragonfly implementation on the XC30 is called the Aries interconnect [2]. In the Aries interconnect, the routers in each group are arranged as rows and columns of a rectangle, with all-to-all links across each row and column but not diagonally. There are 16 routers in the horizontal dimension and 6 in the vertical dimension, for a total of 96 routers per group. Four nodes are connected to each router, bringing the number of nodes per group to 384. This is illustrated in Fig. 2.

2.2 Target System

We ran our experiments on Eos, an XC30 at Oak Ridge National Laboratory. Eos consists of 744 compute nodes with two eight-core 2.6 GHz Intel Xeon E5-2670 processors per node. The hardware bandwidth between nodes is 16 GB/s.

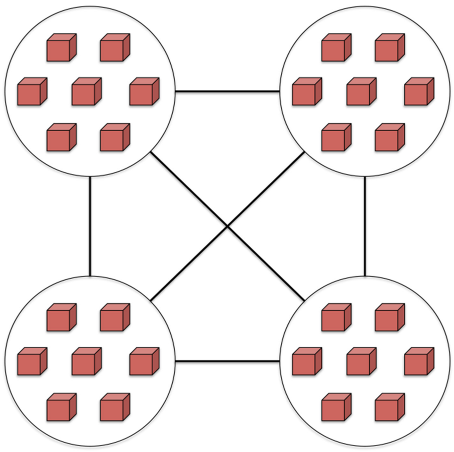


Fig. 1. Dragonfly network basics. Routers (boxes) are in groups (circled), with each group connected to every other group. The routers within groups can be connected in many different ways; no particular topology is shown here.

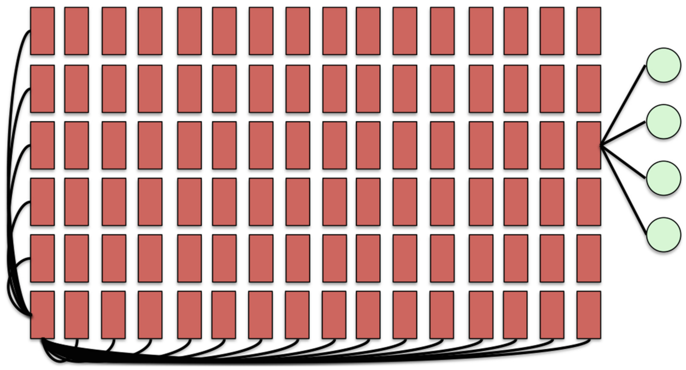


Fig. 2. Group topology in the Aries network, with 16 routers in the horizontal dimension and 6 in the vertical dimension. Each router is connected to every other router in its row and column, which is shown for the router in the lower left-hand corner. Four nodes are connected to each router, which is shown for one of the routers in the rightmost column.

All experiments save for those in Sect. 6.2 use the Intel compiler, version 13.1.3. The MPI implementation is Cray's native MPI. Eos also features simultaneous multithreading (SMT) in the form of Intel Hyper-Threading [19]. This allows for users to run their jobs on up to two times the number of physical cores. However, we do not consider it here, as we have yet to have developed a performance model for this form of SMT.

3 Algebraic Multigrid

The application we focus on in our study is algebraic multigrid (AMG). It is one of the multigrid solvers, which are best known for having a computational cost linear in the number of unknowns being solved. This is very attractive for HPC, where the goal is to solve large problems, and it is therefore of great interest to study the performance of multigrid methods on HPC platforms. Multigrid methods operate by performing some of the work on smaller "coarse grid" problems instead of concentrating it all on the original "fine grid" problem. On each grid, a smoother, typically a simple iterative method like Jacobi or Gauss-Seidel, is applied. Afterwards, a correction is typically solved for on the next coarsest grid, which except for the very coarsest grid involves solving another coarse grid problem. This correction is then applied to accelerate the solution process. The coarsest grid is often solved directly. This particular order of progression through grids, from finest to coarsest and back to finest, is called a V-cycle, which is the most basic multigrid cycle and the one we consider here.

AMG is a means of leveraging multigrid, which was originally developed to solve problems on structured grids, to solve problems with no explicit grid structure, where all that is known is a sparse linear system $A^{(0)}u^{(0)} = f^{(0)}$. This requires AMG to consist of two phases, setup and solve, which are illustrated in Fig. 3. The setup phase involves selecting the variables that will remain on each coarser grid and defining the restriction ($R^{(m)}$) and interpolation ($P^{(m)}$) operators that control the transfer of data between levels. There are a number of algorithms for doing this, and they can be quite complicated. For our experiments, we use the AMG code BoomerAMG [16] in the hypre software library [17] We use HMIS coarsening [7] with extended+i interpolation [6] truncated to at most 4 coefficients per row and aggressive coarsening with multipass interpolation [22] on the finest level. Each coarse grid operator $A^{(m+1)}$ is formed by computing the triple matrix product $R^{(m)}A^{(m)}P^{(m)}$. This operation, particularly for unstructured problems, leads to increasing matrix density on coarse grids, which in turn results in an increasing number of messages being sent among an increasing number of communication partners. These have resulted in substantial challenges to performance and scalability on some machines [3,4], even when using advanced coarsening and interpolation schemes like the ones we use in our experiments, and serve as added motivation for studying AMG on the XC30.

In the solve phase, the primary operations are the smoothing operator and matrix-vector multiplication to form r^m and perform restriction and interpolation. In our experiments, we use hybrid Gauss-Seidel as the smoother. Hybrid Gauss-Seidel uses the sequential Gauss-Seidel algorithm to compute local data within process boundaries, but uses Jacobi smoothing across process boundaries to preserve parallelism. Applying this smoother is a very similar operation to matrix-vector multiplication.

Sparse matrices in BoomerAMG are stored in the ParCSR data structure. A matrix A is partitioned by rows into matrices A_k, $k = 0, 1, \ldots, P-1$, where P is the number of MPI processes. Each matrix A_k is stored locally as a pair

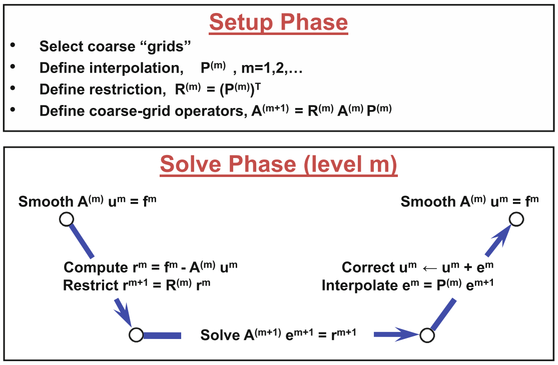

Setup Phase
• Select coarse "grids"
• Define interpolation, $P^{(m)}$, m=1,2,...
• Define restriction, $R^{(m)} = (P^{(m)})^T$
• Define coarse-grid operators, $A^{(m+1)} = R^{(m)} A^{(m)} P^{(m)}$
Solve Phase (level m)
Smooth $A^{(m)} u^m = f^m$
Compute $r^m = f^m - A^{(m)} u^m$
Restrict $r^{m+1} = R^{(m)} r^m$
Solve $A^{(m+1)} e^{m+1} = r^{m+1}$
Interpolate $e^m = P^{(m)} e^{m+1}$
Correct $u^m \leftarrow u^m + e^m$
Smooth $A^{(m)} u^m = f^m$

Fig. 3. Setup and solve phase of AMG.

of CSR (compressed sparse row) matrices D_k and O_k. D_k contains all entries of A_k with column indices that point to rows stored locally on process k, and O_k contains the remaining entries. Matrix-vector multiplication Ax or smoothing requires 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 that needs to be sent from other processes. More detail can be found in [9].

The ability to use a shared memory programming model is provided in BoomerAMG in the form of OpenMP parallelization within MPI processes. This is done using `parallel for` constructs at the loop level, which spawn a number of threads that can each execute a portion of the loop being parallelized. Static scheduling is used, which means the work is divided equally among the threads before the loop starts. The loops parallelized in this fashion are the ones that perform smoother application, matrix-vector multiplication, and the triple matrix product.

4 Performance Model

In previous work [12–14], we developed an accurate performance model for AMG and validated it on a wide range of platforms and network topologies, including Linux clusters, prior Cray machines, and IBM Blue Gene systems. We now expand the model to the dragonfly interconnect and contrast the results.

4.1 Model Specifics

Our model is based on the simple α-β model for interprocessor communication. The time to send a message consisting of n double precision floating-point values is given by

$$T_{\text{send}} = \alpha + n\beta,$$

where α is the communication startup time, and β is the per value send cost. We model computation time by multiplying the number of floating-point operations by a computation rate t_i. We allow this to vary with each level i in the multigrid hierarchy because the operations in an AMG cycle are either sparse matrix-vector multiplication or a smoother application, which is a similar operation. An in-depth study [10] found that the computation time for sparse matrix-vector multiplication varies with the size and density of the matrix, and the operators in an AMG hierarchy have varying sizes and densities. We do not consider the overlap of communication and computation, as there is very little room for this on the communication-intensive coarse grid problems on which our concerns our focused.

We treat the AMG cycle level-by-level. If there are L levels, numbered 0 to $L-1$, the total cycle time is given by

$$T_{\text{cycle}}^{AMG} = \sum_{i=0}^{L-1} T_{\text{cycle}}^{i},$$

where T_{cycle}^{i} is the amount of time spent at level i of the cycle. This is in turn broken down into component steps, diagrammed in Fig. 4, which we write as

$$T_{\text{cycle}}^{i} = T_{\text{smooth}}^{i} + T_{\text{restrict}}^{i} + T_{\text{interp}}^{i}.$$

Smoothing and residual formation, which are combined into T_{smooth}^{i}, are treated as matrix-vector multiplication with the solve operator. Interpolation is treated as matrix-vector multiplication with the interpolation operator. Restriction is treated as matrix-vector multiplication with the restriction operator, which for the purposes of our experiments is the transpose of the interpolation operator.

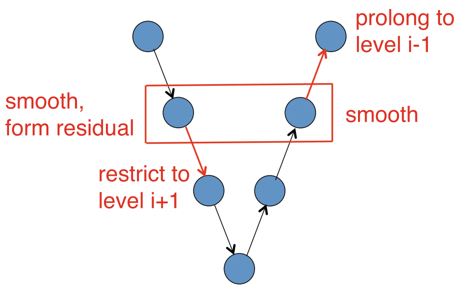


Fig. 4. Fundamental operations at each level of an AMG V-cycle.

To enable us to write expressions for each component operation, we define the following terms to cover different components of the operators that form the multigrid hierarchy:

- P – total number of processes.
- C_i – number of unknowns on grid 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 assume one smoothing step before restriction and one smoothing step after interpolation, which is the default in BoomerAMG. The time spent smoothing on level i is given by

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

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

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

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

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

To this baseline, we add terms and penalties to cover phenomena seen in practice that the α-β model alone does not cover. One such phenomenon is communication distance. While it is assumed that the hop count has a very small effect on communication time, we cannot assume this on coarse grid problems in AMG where many messages are being sent at once. The further a message has to travel, the more likely it is to run into delays from conflicts with other messages. To take this into account, we introduce a communication distance term γ that represents the delay per hop, changing the 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.

Another issue is limited bandwidth, of which we consider two sources. One is the inability to make full use of the hardware. The peak hardware bandwidth is rarely achieved even under ideal conditions, let alone the non-ideal conditions under which applications usually run. The other source of limited bandwidth is network contention from messages sharing links. Let $B_{\max}$ be the peak aggregate per-node hardware bandwidth, and B be the measured bandwidth corresponding to β. Let m be the total number of messages being sent, and l be the number of network links available. Then we multiply β by the sum $\frac{B_{\max}}{B} + \frac{m}{l}$ to take both of these factors into account. The limited hardware bandwidth penalty functions as a baseline, with link contention becoming the dominant factor when it is significant (it might not be significant in certain problems on which the fine grids do not feature much communication).

Multicore nodes are another potential source of difficulties. If the interconnect is not suited to handle message passing traffic from many cores at once, then there can be contention in accessing the interconnect and contention at each hop when routing messages. To capture these effects, we multiply either or both of the terms $\alpha(h_m)$ and γ described earlier by $\lceil t\frac{P_i}{P} \rceil$, where t is the number of MPI tasks per node, and P_i is the number of active processes on level i. Active processes mean ones that still have unknowns in their domains on coarse grids and thus have not "dropped out."

We treat hybrid MPI/OpenMP as follows. The message counts for MPI communication are assumed to change with the number of processes. What we modify explicitly is the computation term t_i. Let b_j be the available memory bandwidth per thread for j threads. We then multiply t_i by $\frac{b_1}{b_j}$. We do this to take into account limited memory bandwidth from threads contending to access memory shared by multiple cores. We expect a slowdown here versus the all-MPI case because there is no longer a definite partitioning of memory when using threads. Our original hybrid/OpenMP model also had a penalty to cover slowdowns from threads being migrated across cores that reside on different sockets [13]; we do not consider this here as it can be readily mitigated by pinning threads to specific cores.

4.2 Adaptation to Dragonfly Networks

The model as presented above is straightforward to adapt to dragonfly networks. It boils down to how to best determine the needed machine parameters. Most of them are readily determined from benchmark measurements, as was the case with other machines. α and β were measured using the latency-bandwidth benchmark in the HPC Challenge suite [8]. α was set to the best reported latency, and β was set to the value corresponding to the best reported bandwidth, which for a reported bandwidth of B bytes per second is $\frac{8}{B}$ for sending double precision floating point data. The t_i terms were measured by performing serial sparse matrix-vector multiplications using the operators for the test problem we used when validating the model; this is further described in Sect. 5.1. The values for b_j needed to evaluate the penalty for hybrid MPI/OpenMP were taken by using the STREAM Triad benchmark [20] and dividing by the number of threads being used.

We determined γ from the measured values of α and β. Starting with the formulation of α as a function of the number of hops h

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

we set $\alpha(h_m)$ to be the measured value of α. If D is the diameter of the network, the maximum latency possible is

$$\alpha(D) = \alpha(h_m) + \gamma(D - h_m).$$

We use the maximum latency reported by the same benchmark we used to measure α as a value for $\alpha(D)$. Then

$$\gamma = \frac{\alpha(D) - \alpha(h_m)}{D - h_m}.$$

For dragonfly interconnects, we set h_m to 2 (reflecting the case where two nodes connected to the same router are communicating). We charge the distance D to each message sent, analogous to the role the height of the tree played for fat-tree interconnects in [12, 13]. Though pessimistic, this distance is charged to reflect the potential impact of routing delays. When counting the number of links available to a message for determining the link contention portion of the limited bandwidth penalty, we use the midpoint of the fewest possible (all the nodes in one group are filled before moving onto the next one) and most possible (each node is in a new group until all groups are in use), as there is no simple geometric formula like there is with a mesh or torus network.

To make the numbers specific to the Aries interconnect, we set D equal to 7; the maximum shortest path between two nodes involves traversing one link to get to the routers in that node's group, two links to find an available connection to reach the next group (not all routers in the Aries interconnect are connected to the optical network), one link to reach that group, two more links to traverse the routers, and then one last link to reach the target node. When counting links for the limited bandwidth penalty, we treat the optical links between groups as four links because they have four times the bandwidth. If there are N nodes in use, and G groups in the network, then the minimum possible number of available links is

$$N + 170\left\lceil \frac{N}{384} \right\rceil + 4\min\left\{ \left\lfloor \frac{N}{384} \right\rfloor, \frac{G(G-1)}{2} \right\},$$

and the maximum possible number of available links is

$$N + 170\min\{N, G\} + 4\min\left\{ N - 1, \frac{G(G-1)}{2} \right\}.$$

In both expressions, the first term accounts for the number of links connecting nodes to routers. The second accounts for the number of router-to-router links in groups, which number $16 \cdot 5 + 6 \cdot 15 = 170$ per group. The third accounts for the number of optical links.

5 Model Validation

5.1 Experimental Setup

For each of our experiments on Eos, the Cray XC30 we are evaluating, we ran 10 AMG solve cycles and measured the amount of time spent in each level, dividing by 10 to get average times spent at each level. For our test problem, we used a

3D 7-point Laplace problem on a cube with $50 \times 50 \times 25$ points per core, as was done in past experiments used to validate this model on other machines [12–14]. The mapping of MPI tasks to nodes was the default block mapping, in which each node is filled with MPI tasks before moving onto the next one. We report results on 1024 and 8192 cores.

Machine parameters for Eos are given in Table 1. How we obtained the values for α, β, and γ was described in Sect. 4.2. We measured t_i by measuring the time for 10 sparse matrix-vector multiplies using the local portion of the solve operator A_i on each level in the MPI-only case and dividing the largest time over all the processes by the number of floating point operations. For $i \geq 3$, we used the value measured for t_2. Per-thread memory bandwidths for the hybrid MPI/OpenMP penalty are in Table 2.

Table 1. Measured machine parameters on Eos.

Parameter	α	β	γ	t_0	t_1	t_2
Value	0.238 µs	0.858 ns	0.416 µs	1.59 ns	0.806 ns	0.545 ns

Table 2. Per thread memory bandwidths on Eos.

No. Threads	1	2	4	8	16
Bandwidth (MB/s)	11106	5335.5	2755.0	1374.8	678.56

5.2 Results

To help us understand the XC30 interconnect, we compared the measured AMG cycle time at each level with what the performance model would predict, with the different penalties turned on and off. Results are plotted in Fig. 5 for the all-MPI case and in Fig. 6 for the hybrid MPI/OpenMP case. In each plot, the measured cycle time at each level is shown as a solid black line. Six different model scenarios are also shown as colored lines with markers, with the best fit solid and the others dotted:

1. Baseline model *(α-β Model).*
2. Baseline model plus distance penalty *(α-β-γ Model).*
3. Baseline model plus distance penalty and bandwidth penalty on β *(β Penalty).*
4. Baseline model plus distance penalty, bandwidth penalty on β, and multicore penalty on α *(α,β Penalties).*
5. Baseline model plus distance penalty, bandwidth penalty on β, and multicore penalty on γ *(β,γ Penalties).*
6. Baseline model plus distance penalty, bandwidth penalty on β, and multicore penalties on α and γ *(α,β,γ Penalties).*

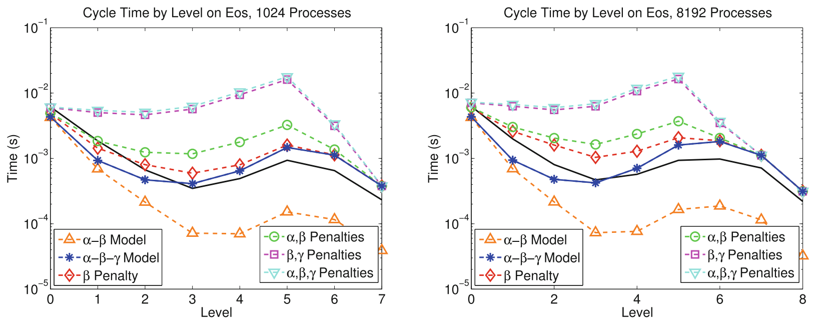


Fig. 5. Measured and modeled AMG cycle time by level on Eos using 1024 (left) and 8192 (right) cores, running all-MPI.

We did not allow the best fit to be a model with more penalties than the best fit for a configuration with more MPI tasks per node. We enforced this constraint because the penalties listed above deal specifically with issues resulting from there being many messages in the network, so it would not make sense for there to be a greater number of penalties when there are fewer MPI tasks per node. All levels are plotted except for the coarsest level. It is not shown because it was solved directly using Gaussian Elimination instead of smoothing.

In all cases, the best fit model was the baseline model plus only the distance penalty. We chose this over the model which also had the bandwidth penalty because the latter was overly pessimistic on 8192 cores in the all-MPI case but not so for 1024 cores. Given that using 8192 cores on Eos involves using 69 % of the machine while using 1024 cores on Eos involves using only 9 % of it, including limited bandwidth from link contention would, if it were a big factor, more accurately capture the performance when using more of the machine. Overall cycle time prediction accuracies are in Table 3. They are almost all at least 85 %, and in some cases above 90 %.

From these results, it is clear that the Aries interconnect does a good job avoiding contention, which is one of the goals of the dragonfly topology [18]. In fact, it is better at doing so in terms of penalty scenarios than any other interconnect on which the performance model has been tested [12–14]. There is also not much slowdown in cycle time when going from 1024 to 8192 cores. However, even with these key positives, there is still a lot of room for improvement. In spite of the lack of contention penalties, the baseline α-β model predicted much better performance than what was actually observed. The γ term was actually larger than the α term; the only other machines on which we observed this were a pair of fat-tree machines on which performance on coarse grids and scalability were very poor [12]. Hybrid MPI/OpenMP performance was also disappointing, highlighted by more rapid deterioration in the available memory bandwidth per thread than was seen in other machines on which the hybrid model was tested [11].

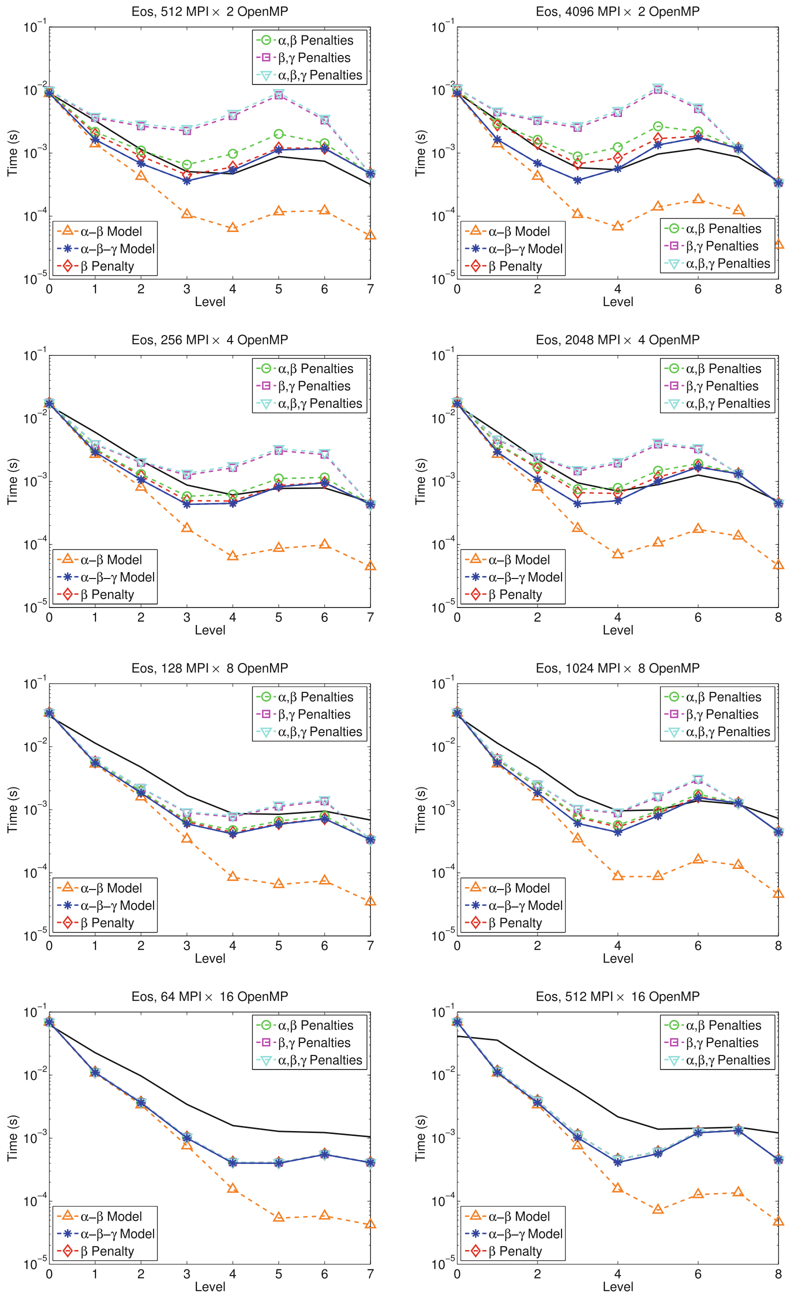


Fig. 6. Measured and modeled AMG cycle time by level on Eos using 1024 (left column) and 8192 (right column) cores, running hybrid MPI/OpenMP. The plot titles show the total number of MPI tasks and the number of OpenMP threads per MPI task.

Table 3. Measured and modeled AMG cycle times and cycle time prediction accuracies on Eos, organized by on-node MPI $\times$ OpenMP mix.

Mix	1024 Cores			8192 Cores		
	Modeled	Measured	Accuracy	Modeled	Measured	Accuracy
16×1	9.75 ms	11.3 ms	86.0 %	11.7 ms	13.0 ms	90.4 %
8×2	14.9 ms	16.3 ms	91.3 %	16.8 ms	18.1 ms	92.8 %
4×4	24.2 ms	27.6 ms	87.4 %	26.5 ms	29.7 ms	89.2 %
2×8	44.2 ms	51.8 ms	85.3 %	46.7 ms	53.9 ms	86.6 %
1×16	86.4 ms	104 ms	83.4 %	88.4 ms	104 ms	85.9 %

6 Model-Guided Performance Improvements

We have observed that, even with low network contention, there is still much room for improvement in the performance of AMG on Eos. We will now turn to a systematic means of improving the performance, driven by the performance model, that will also enable us to gain further insight into the machine.

6.1 Approach

We build on earlier work [15] that used a performance model to drive data redistribution in AMG. This work tested a method which reduced the number of messages sent between processes on coarse grids by having certain groups of processes combine their data and redundantly store it amongst themselves. The method was driven by applying the performance model we described in Sect. 4 during the setup phase before performing each coarsening step to make a decision on whether to redistribute or not. Once redistribution was performed, the remaining levels of the setup phase, and the corresponding level and all coarser ones in the solve phase, were performed using the redistributed operators. Processes would then only communicate with only a handful of other processes, rather than potentially hundreds of them, resulting in speedups often exceeding 2x on an Opteron cluster on which performance and scalability problems had been observed in the past. We use a similar approach with some differences; we will explain as we describe our approach and make note of the differences as they come up.

What we specifically need from the performance model are two quantities, which we call T^i_{switch} and T^i_{noswitch}. The former represents the time spent at level i in the AMG cycle if we perform redistribution, and the latter represents the time spent at that level if we do not. We compute these at each level $i > 0$, and perform redistribution on the first level for which $T^i_{\text{switch}} < T^i_{\text{noswitch}}$. We assume the network parameters α, β, and γ are available to us, along with the particular combination of penalties that is the best match to the overall performance on the machine. The other information we need is problem dependent. Much of it, however, is already available to us. The needed communication and computation counts for the solve operator can be obtained from the ParCSR data structure. The interpolation operator is not available; forming it would require actually performing coarsening, and we want to decide on redistribution before doing that, so we instead approximate both restriction and interpolation with matrix-vector multiplication using the solve operator. This enables us to write an expression for T^i_{noswitch} in terms of the baseline model:

$$T^i_{\text{noswitch}} = 10\frac{C_i}{P}s_i t_i + 5(p_i\alpha + n_i\beta)$$

We still need a value for t_i, which we measure on all active processes like we described in Sect. 5.1. However, instead of stopping after measuring t_2, we stop when the measured value for t_i is greater than the measured value for t_{i-1}. This happens when processes are close to running out of data. Then their t_i measurements are measuring primarily loop overhead instead of computation. t_i is expected to decrease as i increases because the time per floating-point operation has been observed to decrease with the trend of decreasing matrix dimension and increasing matrix density [10] that is seen when progressing from fine to coarse in AMG. Once we stop measuring, we set $t_i = t_{i-1}$ and $t_j = t_{i-1}$ for all levels $j > i$. A question arises of what to do in the hybrid MPI/OpenMP case, which was not covered in [15]. What we do here is use the same measurement scheme we just described, which measures t_i within MPI processes. The measured value will implicitly take the further division of labor into account.

We now turn to computing T^i_{switch}. An expression for this requires both an expression for collective communication used to perform the data redistribution itself and an expression for matrix-vector multiplication with the redistributed solve operator. Reference [15] used an all-gather operation to distribute data redundantly among processes that combined data. We instead use nonredundant data redistribution, where groups of processes combine their data but only one process stores the combined data. The reason for this is that the use of fully redundant redistribution creates many new MPI communicators, and at scale there would be enough to run into a memory-based or implementation-based upper limit on the number of new communicators [5]. Performing nonredundant redistribution in the solve cycle involves two gather operations to combine data from the solution vector and the right-hand side, and one scatter operation when it is time to transfer the result from the levels treated using the redistributed operators to the finer grids that do not use them.

Assuming that C groups of processes combine their data over a binary tree, we get a total of $\lceil \log_2 \frac{P_i}{C} \rceil$ sends for each collective operation. The gather operations involve sends of approximately size $\frac{C_i}{2C}, \frac{C_i}{4C}, \frac{C_i}{8C}, \ldots$ to combine the data, which we charge as the geometric sum $\frac{C_i}{C}\left(\frac{1}{1-\frac{1}{2}} - 1\right) = \frac{C_i}{C}$ units of data sent. The scatter operation is assumed to send approximately $\frac{C_i}{C}\lceil \log_2 \frac{P_i}{C} \rceil$ units of data per send. In terms of the baseline model, the time spent in collective operations is then

$$T^i_{\text{collective}} = 3\left\lceil \log_2 \frac{P_i}{C} \right\rceil \alpha + \frac{C_i}{C}\left(2 + \left\lceil \log_2 \frac{P_i}{C} \right\rceil\right)\beta.$$

The work in [15] sought to keep data movement on-node through a combination of a cyclic mapping of MPI tasks to nodes and having groups of $\frac{P}{C}$ adjacent MPI ranks combine their data. The machine it considered, however, exhibited much better on-node MPI performance than off-node MPI performance [4]. Running on a newer machine, and lacking an on-node performance model, we do not consider localizing data movement. We instead form an MPI communicator out of the processes that still have data and form groups consisting of $\frac{P_i}{C}$ adjacent MPI ranks. If C does not evenly divide P_i, then the first $P_i \mod C$ groups have $\lceil \frac{P_i}{C} \rceil$ processes, and the rest have $\lfloor \frac{P_i}{C} \rfloor$ processes.

We now derive an expression for the amount of time matrix-vector multiplication with the redistributed operator would take. We assume equal division of the gathered data, and equal division of the amount of data sent per message among the total number of sends in the nonredistributed operator. We also assume the number of groups of processes that combine data is less than the largest number of messages a process would send before redistribution, i.e., we are capping the number of communication partners a process could have at $C - 1 < p_i$, and we assume this number of communication partners for each process. The cost for matrix-vector multiplication using the redistributed operator then becomes, in terms of the baseline model,

$$T^i_{\text{new\_matvec}} = 2\frac{C_i}{C} s_i t_i + (C-1)\left(\alpha + \frac{n_i}{p_i}\beta\right).$$

Treating the operations at level i in the AMG solve cycle as five matrix-vector multiplications with the solve operator, as we did for the case with no redistribution, gives us the expression

$$T^i_{\text{switch}} = 5T^i_{\text{new\_matvec}} + T^i_{\text{collective}}$$

for the predicted time at level i when performing redistribution.

We note here that redistribution, by increasing the amount of data per process, will likely result in a different value for t_i that would ideally be used when computing $T^i_{\text{new\_matvec}}$. Measuring this value, however, could only be done after redistribution is performed. To avoid incurring this expense, we instead, as we search for the number of groups of processes C to form, restrict the lower end of the search space so that the locally stored data in the redistributed operator on each process participating in redistribution does not increase too much in size. Without this constraint, the minimum possible value for C is 1, which corresponds to all of the involved processes combining their data onto just one process. The size of the local data is determined to be one of three possibilities, which were used in [10] to classify sparse matrix-vector multiplication problems:

– 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.

We specifically exclude values of C that result in the problem category being at least halfway towards one of the larger ones. Crossing the boundaries from one size classification to another typically results in substantial changes in observed performance, and degradation when moving into a larger problem category sometimes occurs well before the boundary is crossed [10]. For categorization, the cache size is determined by dividing the size of the shared on-node cache by the number of MPI processes per node, as our t_i measurement occurs within MPI processes. The value of C resulting in the lowest value for T^i_{noswitch} is what is used when making a decision on whether or not to redistribute. When searching for this value, we searched over the powers of two less than p_i to save time in the setup phase; a more thorough search is an item for future work.

We employ one other safeguard against overeager redistribution. We do not redistribute if doing so is expected to have a big impact on the overall cycle time. To accomplish this, we keep track of a running sum of the time at each level in the solve cycle as predicted by the model, summing up T^i_{noswitch} for the current level and all finer ones. If there is a projected gain from switching, but that gain is projected to be less than 5 %, then we do not switch. This was not done in [15], but the experiments in that work were performed on an older machine on which coarse grid performance dominated overall runtime when no redistribution was performed. On a newer machine, we want to be more careful, and would rather miss a speedup than risk slowing the cycle down while chasing a small gain.

6.2 Redistribution Experiments

We tested model-guided data redistribution on Eos on two different problems, a 3D 7-point Laplacian and a linear elasticity problem on a 3D cantilever beam with an 8:1 aspect ratio. The 3D Laplacian was run with $30 \times 30 \times 30$ points per core on 512, 4096, and 8000 cores to match one of the test problems from [15]. The linear elasticity problem, which was generated by the MFEM software library [1], was run on 1024 and 8192 cores. Weak scaling in MFEM is accomplished by additional refinement of the base mesh, which resulted in a problem with 6350 points per core on 1024 cores and 6246 points per core on 8192 cores. The elasticity problem is governed by the equation

$$-\text{div}(\sigma(\mathbf{u})) = \mathbf{0},$$

where

$$\sigma(\mathbf{u}) = \lambda \text{div}(\mathbf{u}) I + \mu(\nabla \mathbf{u} + \mathbf{u}\nabla).$$

The beam has two material components discretized using linear tetrahedral finite elements. $\lambda = \mu = 50$ on the first component, and $\lambda = \mu = 1$ on the second. $\mathbf{u}$ is a vector-valued function $\mathbf{u}(x, y, z)$ with a component in each of the three dimensions. The boundary conditions are $\mathbf{u} = \mathbf{0}$ on the boundary of the first component, which is fixed to a wall, $\sigma(\mathbf{u}) \cdot \mathbf{n} = \mathbf{0}$ elsewhere on the boundary of the first component, and $\sigma(\mathbf{u}) \cdot \mathbf{n} = \mathbf{f}$ on the boundary of the second component. The force $\mathbf{f}$ is a vector pointing in the downward direction with magnitude 0.01. The beam is diagrammed in Fig. 7.

We ran 10 trials of solving each problem to a tolerance of 10^{-8} using conjugate gradient preconditioned by AMG, recording both the setup and solve phase

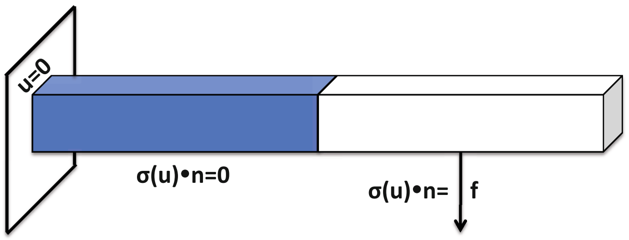


Fig. 7. 3D cantilever beam for the linear elasticity problem. The first component (left) is attached to the wall. The downward force **f** is pulling on the second component (right).

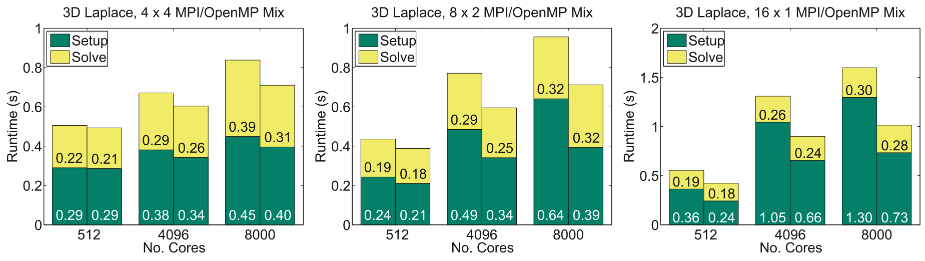
times. Like with the model validation experiments, we used the default block mapping of MPI tasks to nodes. We had to switch compilers to the PGI compiler, version 13.7-0, because the default Intel compiler failed to compile MFEM. When making the switching decision, we used the best fit performance model from Sect. 5.2, the baseline model plus the distance penalty term γ.

For the Laplace problem, we ran three different on-node mixes of MPI and OpenMP: 16×1, 8×2, and 4×4. We ran the elasticity problem using exclusively MPI, owing to difficulties compiling MFEM with OpenMP enabled, as hybrid MPI/OpenMP support in MFEM is currently experimental [1]. We did not use aggressive coarsening for the linear elasticity problem due to much poorer convergence when using it, following the default behavior of the linear elasticity solver in MFEM. Results for the Laplace problem are in Fig. 8, and results for the elasticity problem are in Table 4.

Table 4. Results on Eos for the Linear Elasticity problem.

	1024 Cores			8192 Cores		
	Setup	Solve	Total	Setup	Solve	Total
No redistribution	0.78 s	1.06 s	1.84 s	6.72 s	2.64 s	9.36 s
With redistribution	0.75 s	0.82 s	1.58 s	2.99 s	1.55 s	4.54 s
Speedup	1.04	1.29	1.16	2.25	1.93	2.06

The Laplace results reveal some interesting behavior. In the case of the solve phase, the best performance was using exclusively MPI, and there were mostly modest gains from data redistribution. This is not surprising when considering that the best fit from the performance modeling experiments was the model with no penalties to the baseline beyond the introduction of the distance term, a very favorable contention scenario. The setup phase, however, was a different story. Here, performance improved with the introduction of OpenMP, and the more MPI-rich configurations showed substantial speedups from redistribution



	4 × 4 Speedups			8 × 2 Speedups			16 × 1 Speedups		
Cores	512	4096	8000	512	4096	8000	512	4096	8000
Setup	1.00	1.12	1.12	1.14	1.44	1.64	1.50	1.59	1.78
Solve	1.05	1.12	1.26	1.06	1.16	1.00	1.06	1.08	1.07
Overall	1.02	1.12	1.18	1.10	1.32	1.35	1.31	1.46	1.58

Fig. 8. Results and corresponding speedups when using model-guided data redistribution for the 3D 7-point Laplace problem on Eos. The bars on the left in each graph show timings when doing no redistribution, while the bars on the right show timings when doing redistribution.

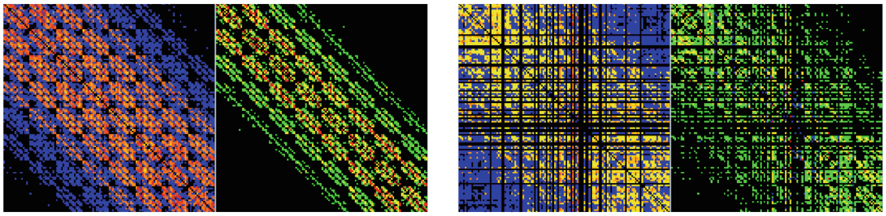


Fig. 9. Communication patterns on levels 4 (left) and 5 (right) for the 3D Laplace problem from the performance model validation experiments, with the setup phase on the left and the solve phase on the right.

at scale. This is a significant discrepancy in performance between the two phases; we will comment further in the next section.

Moving onto the linear elasticity problem, we see a modest speedup for the run on 1024 cores, but a large one for the run on 8192 cores. There was no big discrepancy between setup and solve phase speedup either. We should note that this problem had coarse grids with much larger stencils than the Laplace problem, with the largest coarse grid stencil for the elasticity problem averaging just under 500 nonzero entries per row compared to just under 100 for the Laplace problem. This means more messages are being sent over the interconnect, and we are seeing a big performance gain from reducing the number of messages even with an interconnect that was not showing much in the way of contention problems when we were validating the model. We will discuss this further in our concluding remarks.

7 Conclusions

To better understand the HPC potential of the dragonfly interconnect, we studied the performance of algebraic multigrid on a Cray XC30, developing a performance model and using it to analyze the performance of the AMG solve cycle. We made further use of the same performance model to guide data redistribution to improve performance. Substantial improvements in the setup phase for a 3D Laplace problem and in both phases for a linear elasticity problem showed that even an interconnect that rated very strongly in terms of penalties added on top of a basic α-β model does not automatically mean that there are no issues with interprocessor communication that could be improved upon.

One trait of note that was mentioned before is that the γ term in the performance model is larger than the α term, which was observed on two older fat-tree machines that suffered from poor coarse grid performance that hurt overall scalability. Though Eos features a much better interconnect, the presence of this property is still noteworthy, and suggests that communication between different router groups could suffer from substantial delays. That data redistribution has its biggest effect on runs using the majority of the machine hints at this.

What really stood out were the difference between the solve and setup phase speedups when using data redistribution for the 3D Laplace problem and the large speedup when solving the linear elasticity problem on 8192 cores. We mentioned earlier that the linear elasticity problem features much larger stencil sizes on coarse grids and thus dramatically increased interprocessor communication compared to the Laplace problem. The setup phase of AMG also features increased communication, substantially more than the solve phase. Figure 9 shows the communication patterns on the two most communication-intensive levels in the hierarchy from the 3D Laplace problem from the performance model validation experiments, levels 4 and 5, run in an all-MPI programming model on 128 cores on a multicore Opteron cluster that was analyzed in [12]. The plots were obtained using the performance analysis tool TAU [21]. On both levels, there was a lot more communication in the setup phase, with it being almost all-to-all on level 5.

So while the XC30 interconnect rated favorably in terms of contention penalties when we were testing our performance model, we saw that there were still large benefits to reducing the number of messages sent when that number was very large, whether it was through data redistribution, using a hybrid programming model, or a combination of both. In contrast, these benefits were found to be more modest for the same test problems on an IBM Blue Gene/Q, where reported overall speedups from data redistribution peaked at 17 % for the Laplace problem and 39 % for the linear elasticity problem, even though its interconnect did not rate as well in terms of the network contention penalties in our performance model [11]. Future work will involve examining the communication behavior and its effects on performance in more detail, including the construction of a performance model for the setup phase of AMG, to help pinpoint the major bottlenecks and see if there is a threshold at which network contention becomes a serious problem and if so, map it.

What we have seen so far on the Cray XC30, though, hints that the dragonfly topology will have problems with communication-heavy applications. Though the topology allows for wide variety in the specifics of the individual groups of routers that comprise the overall network, there is still the unifying feature of the all-to-all connections between the groups. Experiments in which we tasked the interconnect with handling a large number of messages led to performance degradation, especially when using the majority of the machine, that was readily improved when messages were traded for computation. These results point to a risk of slowdowns when communicating between groups of routers that will need to be addressed to make dragonfly interconnects effective at scale.

Acknowledgements. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy awards DE-SC0004131 and DE-FG02-13ER26138/DE-SC0010049, 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-659475). It used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725. Neither Contractor, DOE, or the U.S. Government, nor any person action 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. MFEM: Finite Element Discretization Library. https://code.google.com/p/mfem
2. Alverson, B., Froese, E., Kaplan, L., Roweth, D.: Cray XC® Series Network (2012). http://www.cray.com/Assets/PDF/products/xc/CrayXC30Networking.pdf
3. Baker, A.H., Gamblin, T., Schulz, M., Yang, U.M.: Challenges of scaling algebraic multigrid across modern multicore architectures. In: 25th IEEE Parallel and Distributed Processing Symposium, Anchorage, AK, May 2011
4. Baker, A.H., Schulz, M., Yang, U.M.: On the performance of an algebraic multigrid solver on multicore clusters. In: Palma, J.M.L.M., Daydé, M., Marques, O., Lopes, J.C. (eds.) VECPAR 2010. LNCS, vol. 6449, pp. 102–115. Springer, Heidelberg (2011)
5. Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Kumar, S., Lusk, E., Thakur, R., Träff, J.L.: MPI on a million processors. In: Ropo, M., Westerholm, J., Dongarra, J. (eds.) EuroPVM/MPI. LNCS, vol. 5759, pp. 20–30. Springer, Heidelberg (2009)
6. De Sterck, H., Falgout, R.D., Nolting, J.W., Yang, U.M.: Distance-two interpolation for parallel algebraic multigrid. Numer. Linear Algebra Appl. **15**, 115–139 (2008)
7. De Sterck, H., Yang, U.M., Heys, J.J.: Reducing complexity in parallel algebraic multigrid preconditioners. SIAM J. Matrix Anal. Appl. **27**, 1019–1039 (2006)
8. Dongarra, J., Luszczek, P.: Introduction to the HPCChallenge Benchmark Suite. Technical report ICL-UT-05-01, University of Tennessee, Knoxville, March 2005

9. Falgout, R.D., Jones, J.E., Yang, U.M.: Pursuing scalability for hypre's conceptual interfaces. ACM Trans. Math. Softw. **31**, 326–350 (2005)
10. Gahvari, H.: Benchmarking Sparse Matrix-Vector Multiply. Master's thesis, University of California, Berkeley, December 2006
11. Gahvari, H.: Improving the Performance and Scalability of Algebraic Multigrid Solvers through Applied Performance Modeling. Ph.D. thesis, University of Illinois at Urbana-Champaign (2014)
12. Gahvari, H., Baker, A.H., Schulz, M., Yang, U.M., Jordan, K.E., Gropp, W.: Modeling the performance of an algebraic multigrid cycle on HPC platforms. In: 25th ACM International Conference on Supercomputing, Tucson, AZ, June 2011
13. Gahvari, H., Gropp, W., Jordan, K.E., Schulz, M., Yang, U.M.: Modeling the performance of an algebraic multigrid cycle on HPC platforms using hybrid MPI/OpenMP. In: 41st International Conference on Parallel Processing, Pittsburgh, PA, September 2012
14. Gahvari, H., Gropp, W., Jordan, K.E., Schulz, M., Yang, U.M.: Performance modeling of algebraic multigrid on blue Gene/Q: lessons learned. In: 3rd In-ternational Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems, Salt Lake City, UT, November 2012
15. Gahvari, H., Gropp, W., Jordan, K.E., Schulz, M., Yang, U.M.: Systematic reduction of data movement in algebraic multigrid solvers. In: 5th Workshop on Large-Scale Parallel Processing, Cambridge, MA, May 2013
16. Henson, V.E., Yang, U.M.: BoomerAMG: a parallel algebraic multigrid solver and preconditioner. Appl. Numer. Math. **41**, 155–177 (2002)
17. hypre: High performance preconditioners. http://www.llnl.gov/CASC/hypre/
18. Kim, J., Dally, W.J., Scott, S., Abts, D.: Technology-driven, highly-scalable dragonfly topology. In: 35th International Symposium on Computer Architecture, Beijing, China, June 2008
19. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., Upton, M.: Hyper-threading technology architecture and microarchitecture. Intel Technol. J. **6**, 4–15 (2002)
20. McCalpin, J.D.: Sustainable Memory Bandwidth in Current High Performance Computers. Technical report, Advanced Systems Division, Silicon Graphics Inc. (1995)
21. Shende, S.S., Malony, A.D.: The TAU parallel performance system. Int. J. High Perform. Comput. Appl. **20**, 287–311 (2006)
22. Yang, U.M.: On long-range interpolation operators for aggressive coarsening. Numer. Linear Algebra Appl. **17**, 453–472 (2010)