Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP

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

*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801
†Computational Science Center, IBM TJ Watson Research Center, Cambridge, MA 02142
‡Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551
{*gahvari,wgropp*}*@illinois.edu, kjordan@us.ibm.com,* {*schulzm,umyang*}*@llnl.gov*

***Abstract*—The rise of multicore cluster architectures has led to intense interest in using a combination of MPI and OpenMP to more effectively program these machines. We present a performance model for hybrid implementation of the solve cycle of algebraic multigrid (AMG), a popular iterative solver for large sparse linear systems and a key component of many scientific simulations. We validate the model on two leading parallel platforms, and discuss implications for applications programmed in a hybrid model on future machines.**

I. Introduction

Recent trends in computer architecture have led to the rise of multi- and many-core architectures. Due to thermal and power constraints, single-core speeds have stopped rising, and improvements in performance are mainly achieved through increased parallelism by integrating more and more processing cores on a single chip. Consequently, massively parallel supercomputers will become even more massively parallel, with future exascale machines projected to have core counts in the hundreds of millions to billions.

Traditionally, the dominant model for programming such massively parallel machines has been some form of message-passing, typically MPI, since it is best suited for distributed memory architectures. The rise of machines with multicore nodes, however, has led to an increasing interest on the part of application developers in incorporating shared memory programming into previously distributed memory codes in order to better match the underlying architecture. One example is the application we are focusing on, algebraic multigrid (AMG). AMG is a popular linear solver for large, sparse linear systems of linear equations that performs work linear in the number of unknowns when it works well, making it highly suitable for solving large problems on massively parallel machines. AMG has scaled very well on the IBM Blue Gene/L [1] and Blue Gene/P [2] platforms, but has run into problems on multicore clusters [3], [4].

In this paper, we explore the performance difficulties reported in these studies using a novel performance model for the hybrid MPI/OpenMP execution of AMG on multicore architectures. Our model builds upon a performance model we previously developed for the AMG solve cycle when programmed entirely in a distributed memory model [5]. We first make an adjustment to this model to improve its overall fit to the observed performance, and then add threading aspects to the model to cover the case of hybrid MPI/OpenMP. We present results on two multicore cluster architectures, and discuss implications based on the model for future platforms.

The remainder of the paper proceeds as follows. Section II summarizes AMG and its performance when programmed using hybrid MPI/OpenMP. Section III gives an overview of our existing performance model, then details the additions we make to cover hybrid execution models. Section IV presents the results of our validation experiments, and Section V discusses the implications of our results. Section VI discusses related work, and Section VII presents our concluding remarks.

II. Algebraic Multigrid

Multigrid methods are popular means of solving large, sparse linear systems of equations. Their popularity arises from ideal computational complexity — when they work well, multigrid methods solve a system with n unknowns in $\mathcal{O}(n)$ oprations. They work by solving a sequence of much smaller "coarse grid" problems to accelerate the solution of the original "fine grid" problem, which is usually quite large. Multigrid was originally developed for solving problems on structured grids. Algebraic multigrid (AMG) extends multigrid so that it can work on problems not defined on an explicit grid. AMG forms its grids and operators to transfer between them (restriction to transfer from fine to coarse and interpolation to transfer from coarse to fine) entirely from the coefficients of the input matrix. This involves a setup phase to select grids and to create operators before the problem is actually solved in the solve phase. A more detailed description is available in [6].

During the solve phase, a smoother is applied on each level k. Levels are numbered from finest to coarsest, starting at 0. The residual is formed and then transferred to the next coarsest grid $(k+1)$ through multiplication by the restriction operator. An error correction is determined on the coarse grid, where it is interpolated back to grid k. One more smoothing step is performed, and the result is interpolated

up to grid $k-1$. Interpolation is accomplished through multiplication by the appropriate interpolation operator. We assume a V-cycle, in which there is one progression from the finest grid to the coarsest grid coupled with a progression back to the finest grid, but note that our methodology can be straightforwardly extended to more complicated multigrid cycling strategies. The classical smoother used for AMG is the highly sequential Gauss-Seidel iteration. For better parallel performance, we use a parallel variant called hybrid Gauss-Seidel, which involves performing Gauss-Seidel within a process and performing Jabobi iteration across process boundaries.

Our experiments use BoomerAMG [7], the parallel AMG code in the hypre [8] software library. We use HMIS coarsening [9] with extended+i interpolation [10] truncated to at most 4 coefficients per row and aggressive coarsening with multipass interpolation [11] on the finest level.

A. Parallellization

In BoomerAMG, all matrices are stored in a parallel CSR (compressed sparse row) format, in which the matrix A is partitioned by rows into matrices A_k, $k = 1 \dots p$, where p is the number of MPI processes, and the portion belonging to each process is stored on that process as two matrices in sequential CSR format: $A_k = D_k + O_k$. D_k contains all entries in A_k whose column indices point to rows stored on process k. O_k contains the remaining entries. Matrix-vector multiplication Ax requires evaluation of $A_k x = D_k x^D + O_k x^O$ on each process, where x^D is the portion of the vector x stored locally and x^O is the portion that needs to be sent by other processes. Further detail is available in [12]. OpenMP parallelization is done within MPI tasks, and accomplished at the loop level using `parallel for` constructs, which spawn a number of threads that each execute a portion of the loop being parallelized (in this case `for` loops). The parallelized loops are the ones that perform smoother application and matrix-vector multiplication.

B. Hybrid Performance

The main challenge to achieving good scalability for AMG on multicore clusters is performance degradation on coarse grids. Processes have more communication partners that are farther away, with much less computation to balance out the communication as well. This was shown for a 3D 7-point Laplace model problem with $50 \times 50 \times 25$ points per core on Hera, a multicore Opteron cluster with four quad-core processors per node, in [5].

Using hybrid MPI/OpenMP alleviates these issues to some degree, but not entirely. Figure 1 plots the time spent at each level of an average (taken by measuring 10 cycles) V-cycle when solving the 7-point model problem on Hera on 1024 cores, using different combinations of MPI and OpenMP on each node. In the majority of cases, there still is performance degradation on coarse grids, for the reasons

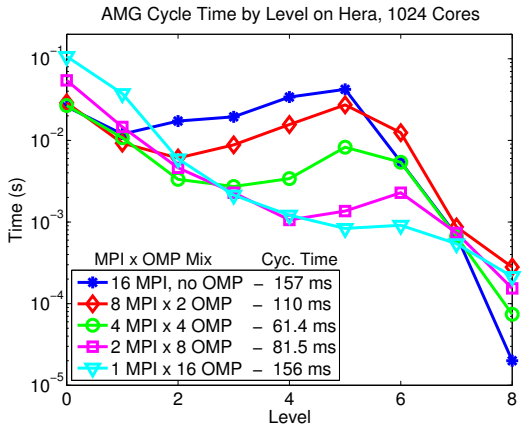

Figure 1. Level-by-level timings for an AMG V-cycle when solving the 3D 7-point Laplace model problem. Total cycle times, which are the sums of the times at each level, are listed in the legend.

given above. The best performance is obtained when using 4 MPI tasks and 4 OpenMP threads per node, i.e., 1 MPI task per multicore processor/socket. This is the case for other core counts as well. What is happening is that, while using OpenMP reduces the MPI traffic, which improves performance on coarse grids, it also degrades performance on fine grids. This was found to occur because of inefficient memory allocation and thread migration between cores when using OpenMP [3]. Explicitly pinning threads and processes to cores and sockets combined with a special support library for memory allocation was able to drastically reduce this problem, but the mix of 4 MPI tasks and 4 OpenMP threads per node was still found to yield the best overall performance. We will discuss OpenMP performance in more detail later on, but the pinning of threads and processes and the use of special routines are beyond the scope of this paper.

III. Performance Model

We previously developed and validated a performance model for the AMG solve cycle when programmed using a distributed memory programming model [5]. At its core, the model is based on a simple latency-bandwidth model for communication, but then adds communication distance terms and penalties to take into account multicore issues and limited bandwidth. The penalties can be "on" or "off" depending on the target architecture, and in fact, the best fit to a particular machine will have some penalties in effect and others not. After giving an overview of this model, we will build upon it in two significant ways. First, we update the limited bandwidth penalty to take link contention into account. This is something the previous model did not do, but is necessary to characterize the communication tradeoffs for different process/thread configurations. We then further augment the model to take threading in the form of hybrid MPI/OpenMP constructs into account.

A. Distributed Memory Model

The underlying distributed memory performance model uses a combination of problem-specific information and machine parameters to ensure its applicability to the widest possible space of architectures. In the baseline model, communication is modeled as follows: the time to send an n-element message is $T_{\text{send}} = \alpha + n\beta$, where α is the communication start-up time and β is the per-element send cost. The α term covers both software overhead and latency involved in message passing, and the β term is connected to the available bandwidth. The remaining parameters are:

- P – total number of cores
- C_i – number of unknowns on grid level i
- $s_i, \hat{s}_i$ – average number of nonzeros 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
- t_i – time per flop on level i

We do not mention the restriction operator separately here, as in our experiments the restriction operator is the transpose of the interpolation operator. We also assume one smoothing step before restricting and one smoothing step after interpolation, which is the default in BoomerAMG.

To model the overall solve cycle, we break it down into a series of individual steps. If T^i_{solve} is the time spent at level i of the cycle, we have

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

T^i_{smooth} is the time spent smoothing on level i, T^i_{restrict} is the time spent restricting from level i to level $i+1$, and T^i_{interp} is the time spent interpolating from level i to level $i-1$. If there are L levels, the cycle time is given by

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

We now consider each of these individual steps. At level i in the cycle, we have to first run a smoother sweep, form the residual, and restrict it to level $i+1$ if i is not the coarsest level. Eventually, the computation will return to that level, requiring another smoother sweep followed by interpolation of the correction to level $i-1$ if i is not the finest level. All of these operations are either matrix-vector multiplication (residual formation) or a very similar operation. We therefore model each operation as a matrix-vector multiply using the appropriate operator, with two flops per matrix entry. The smoothing time at level i is

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

The time spent restricting from level i to $i+1$ 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 $i-1$ 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 model, we make the following additions to reflect issues observed on real machines. To reflect messages traveling long distances, we add a γ term that represents the delay per hop. In the baseline model, this change is reflected 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. We assume h is the diameter of the network within the job's partition to account for routing delays. h_m depends on the network topology and is 1 in torus or mesh networks, and 2 in fat-tree networks where a message has to travel through 1 switch or over 2 links.

There are also bandwidth limitations. Peak hardware bandwidth is rarely achieved in message passing under ideal conditions. What is typically achievable is itself rarely achieved under non-ideal conditions. To take this into account, we multiply β by $\frac{B_{\max}}{B}$, where $B_{\max}$ is the peak aggregate per-node bandwidth, and B is the measured bandwidth corresponding to β ($B = \frac{8}{\beta}$ with β the time to send one double-precision floating point value).

An additional issue on multicore nodes is increased contention to get onto the interconnect, and, if the network itself is not built to handle the resulting traffic, also 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 above by $\lceil c\frac{P_i}{P} \rceil$, where c is the number of cores per node, and P_i is the number of active processes on level i. By active processes, we mean ones that have not "dropped out" due to lack of work, which happens in AMG when coarsening has proceeded to a point where some processes have no unknowns left in their respective domains.

B. Updated Bandwidth Penalty

Our baseline distributed memory model produces an accurate fit to the communication performance on an IBM Blue Gene/P with only the γ term and the β penalty in effect. However, on other machines, there was a tendency for the best-fit model to underpredict the runtime. To remedy this, we take into account another situation that the previous model ignored, but that is of increasing importance as the number of cores per node increases, namely contention from messages sharing links. If m is the total number of messages in the network, l is the number of links, and $\hat{\beta}_{\text{pen}}$ is the β penalty described in the previous section, then the updated β penalty is $\beta_{\text{pen}} = \hat{\beta}_{\text{pen}} + \frac{m}{l}$. This adds the reduced

bandwidth from link contention to the reduced hardware bandwidth represented by the prior β penalty. We found this to overpenalize on Blue Gene/P, which is an indication of how well its interconnect handles traffic as well as its moderate core count, but as we will see in the validation section, it does well on the machines we evaluate in this paper.

C. Extension to Hybrid MPI/OpenMP

The model in its current state does not include the ability to express conditions arising from hybrid programming with MPI and OpenMP. Given that past work has found this to be either beneficial for or detrimental to AMG on multicore clusters, depending on the architecture and the mix of MPI tasks and OpenMP threads being used, we want to better understand the reasons for this performance behavior, with the goal to optimize AMG on current platforms and to make projections of its efficiency on future machines.

Our first step is to modify the communication parameters of the distributed memory model to take into account that not every core will be running an MPI task. The network parameters α, β, and γ do not change. The communication counts and numbers of active processes are assumed to change based on the reduced number of processes, so we do not explicitly modify them. We do not explicitly modify the limited bandwidth penalty either, as it depends on the number of messages. We are then left with updating the multicore penalties to α and γ according to the number of MPI processes per node instead of the number of cores, as there will be fewer processes contending for resources in the interconnect. Instead of the multiplier in the distributed memory model, we use $\left\lceil t\frac{P_i}{p}\right\rceil$, where t is the number of MPI tasks per node and p is the total number of MPI processes. This reduces to the multicore penalty in the distributed memory model in the MPI-only case.

We now consider issues specific to threading. One is limited memory bandwidth. Unlike the message passing case, there is no definite partitioning of the memory. Threads can contend with each other when accessing memory that is shared by multiple cores, which reduces the available memory bandwidth. To take this into account, we penalize t_i as follows. Let b_j be the memory bandwidth per thread for j threads. We define $p_{\text{mem}} = \frac{b_1}{b_j}$ to be the memory bandwidth penalty for j threads, and multiply t_i by p_{mem}.

Furthermore, the operating system can migrate threads across cores. If these cores are on different sockets or connected to separate memory controllers, then this can cause a significant decline in on-node performance if there are enough threads. This was observed in [3] and [4] when there was less than one MPI task per socket. To take this into account, we penalize with the worst case in mind – if p_{node} is the number of processors on a node, and j is the number of threads, then we define the processor penalty p_{proc} to be $\max\left\{1, \frac{j}{p_{\text{node}}}\right\}$. We multiply t_i by this penalty as well. The overall penalized value of t_i is then obtained by multiplying by both penalties: $t_i \leftarrow p_{\text{mem}} p_{\text{proc}} t_i$.

There is also overhead involved in spawning threads, but we do not consider this. Runs of the EPCC OpenMP benchmark [13] found this to be in the microsecond range, so it is dominated by the time spent in MPI calls. Table I summarizes the changes made to the model to incorporate OpenMP.

Table I
SUMMARY OF CHANGES TO PERFORMANCE MODEL TO INCORPORATE OPENMP.

	Without OpenMP	With OpenMP
α penalty	$\alpha(h_m) \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \alpha(h_m)$	$\alpha(h_m) \leftarrow \left\lceil t\frac{P_i}{p}\right\rceil \alpha(h_m)$
γ penalty	$\gamma \leftarrow \left\lceil c\frac{P_i}{P}\right\rceil \gamma$	$\gamma \leftarrow \left\lceil t\frac{P_i}{p}\right\rceil \gamma$
t_i penalty	none	$t_i \leftarrow p_{\text{mem}} p_{\text{proc}} t_i$

IV. MODEL VALIDATION

A. Machine Descriptions

Hera is a Linux cluster at Lawrence Livermore National Laboratory consisting of 800 compute nodes, with four quad-core 2.3 GHz AMD Opteron processors per node. The batch scheduler allows for a maximum of 256 nodes to be allocated to a single job. The nodes are connected by an Infiniband interconnect organized as a two-level fat-tree topology. The 72 first-level switches are each connected to 12 nodes, and have 12 more ports that connect to the second-level switches. There are four second-level switches, so three ports per first-level switch are connected to each second-level switch. The hardware bandwidth between nodes is 2.5 GB/s. The specific version of Linux being run is CHAOS, a specialized version of RHEL5 adapted for HPC. All experiments use gcc 4.1.2 as the compiler, and the MPI implementation is MVAPICH v0.99.

Jaguar is a Cray XK6 system at Oak Ridge National Laboratory. It was previously a Cray XT5, but was upgraded at the beginning of the year, and will have its name changed to Titan at the end of the year. There are 18,688 compute nodes, with one 16-core 2.2 GHz AMD Opteron 6200 Series processor per node. There are also 960 NVIDIA Tesla X2090 GPUs, but we do not consider them here. The nodes are connected by a 3D torus interconnect, with a hardware bandwidth of 20.8 GB/s between nodes. All experiments use the PGI compiler, version 12.1, and the MPI implementation is Cray's native MPI.

B. Experimental Setup

On both of the architectures tested, we ran 10 AMG solve cycles and measured the amount of time spent in each level, dividing by 10 to get a measure of the time spent in each level for an average solve cycle. As a test problem, we used a 3D 7-point Laplace problem on a cube, with a problem size

Table II
MACHINE PARAMETERS α, β, AND t_i.

	Hera	Jaguar
α	1.31 μs	1.68 μs
β	6.08 ns	1.59 ns
γ	2.68 μs	57.5 ns
t_0	5.12 ns	3.38 ns
t_1	1.39 ns	1.30 ns
t_2	1.09 ns	0.928 ns

Table III
MEMORY BANDWIDTH PER THREAD REPORTED BY STREAM TRIAD FOR VARYING NUMBERS OF OPENMP THREADS.

	Hera	Jaguar
1 OpenMP thread	3.05 GB/s	6.91 GB/s
2 OpenMP threads	2.95 GB/s	3.87 GB/s
4 OpenMP threads	2.83 GB/s	2.79 GB/s
8 OpenMP threads	1.37 GB/s	1.41 GB/s
16 OpenMP threads	1.23 GB/s	1.40 GB/s

of $50 \times 50 \times 25$ points per core. The mapping of MPI tasks to nodes used were the defaults on each machine, which in both cases was a block mapping, where each node is filled with MPI ranks before moving to the next one.

C. Machine Parameters

1) Distributed Memory Model: We determine α and β from best-case latency and bandwidth measurements taken by the latency-bandwidth benchmark in the HPC Challenge suite [14]. To determine γ, we start with the formulation of α as a function of the number of hops h in the models that take distance into account:

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

As $\alpha(h_m)$ is the latency for the shortest possible message distance, we set this to be the minimum latency reported in the benchmark results, which is our value for machine α. The maximum latency possible is

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

where D is the diameter of the network. We set $\alpha(D)$ to be the maximum latency reported in the benchmark results. Then

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

We measure the computation rates t_i using a serial sparse matrix-vector multiply benchmark [15] run on one node, simultaneously on all the cores to properly stress the memory system. We obtain specific values for the first three levels (t_0, t_1, and t_2), and use the value obtained for t_2 to approximate t_i on all coarser levels. The obtained values are determined from the observed computation rate for matrix-vector multiplications matching the dimension and number of nonzero entries per row of the solve operators for the respective levels. The values for α, β, γ, and t_i appear in Table II.

2) Updated Bandwidth Penalty: Computing the updated penalty to β requires computing the number of links in use. On Jaguar, the number of available links is simply three times the number of nodes in use due to the 3D torus interconnect. On Hera, the number of available links depends on which nodes on the fat-tree are given by the scheduler. The number of first-level links in use will always be N, where N is the number of nodes in the job's partition. The number of second-level links in use, however, can vary. The minimum possible is $4\left\lceil \frac{N}{12} \right\rceil$, which assumes all the nodes connected to a first-level switch are allocated before moving on to the next switch. The maximum possible is $4\min\{N, 72\}$, which assumes a cyclic allocation of nodes. We assume that the median number of second-level links are in use, and multiply this number by 3 to take into account the increased link bandwidth on the second level. The total number of links we assume to be in use is then $N + 6\left(\left\lceil \frac{N}{12} \right\rceil + \min\{N, 72\}\right)$.

3) Hybrid Model: We use the STREAM Triad benchmark [16] to compute the memory bandwidth per thread when using OpenMP, averaging the reported result over 10 trials. The results, for each machine, are in Table III. The number of multi-processors or sockets per node is 4 on Hera, but on Jaguar, it is only nominally 1. The AMD Opteron 6200 series processor actually consists of two dies with eight cores per die [17], so we treat it as a pair of eight-core processors.

D. Validation Results

The baseline model, with and without taking communication distance into account, and the possible combinations of penalties to α, β, and γ give us a total of nine possible performance scenarios. We show results for the most relevant six, which are the baseline model, the model with the γ term, and all models with a penalty to β, labeled in plots as follows:

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

These six are the most relevant because they include the two baseline scenarios with no extra penalties, and all of the best fits to the data in the distributed memory case had at least a penalty to β [5].

We apply these models to AMG and compare them with the actual measured performance, which is shown as a solid

black line in the following figures. Of the models, which are plotted in colored lines with markers, the best fit is shown as a solid line, while the others are shown as dotted lines. As the penalties all deal with message-passing issues, the best fit is not allowed to add penalties when moving to fewer MPI tasks. The coarsest grid, which is solved using Gaussian Elimination instead of smoothing, is not shown.

1) Updated Bandwidth Penalty: Figure 2 shows the results of the model with the updated β penalty. Like before, the best fit on Hera comes from the application of every penalty. Average cycle time prediction accuracy for the core counts tested increases from 89% to 95%.

On Jaguar, the best fit model is the α-β-γ model with the bandwidth penalty on β. This is a departure from the previous Cray XT5 machine, on which all penalties applied [5]. This reflects a dramatic improvement in the interconnect, which was upgraded moving from XT5 to XK6. The average cycle time prediction accuracy is 78%. It would be higher but for the measured t_0 and t_1 terms underpredicting the computation time on the fine levels.

2) Hybrid Model: The hybrid model results are in Figure 3 for Hera, and Figure 4 for Jaguar. The titles show the total number of MPI tasks and the number of OpenMP threads per node, the product of which is the number of cores.

On Hera, the best fit model applies all penalties except for two cases. For 128 MPI $\times$ 8 OpenMP, the best fit came from the α-β-γ model with the bandwidth penalty on β. In the case of 16 OpenMP threads per node, there were no multicore penalties. Consequently, the best fit model is again the α-β-γ model with the bandwidth penalty on β. The overall fit was very good, with the cycle time prediction accuracy at worst 85%, and usually well above 90%.

On Jaguar, the best fit model is the α-β-γ model with the bandwidth penalty on β. Cycle time prediction accuracies were very good, again at least 85% and usually above 90%, except for the case of 8 OpenMP threads per node. In that case, the memory bandwidth penalty was excessive, predicting more of a slowdown than what the actual results indicated. The fits are not as good as they were on Hera, though. The results tend to combine an overprediction of runtime on the finest level with an underprediction of runtime on the majority of the coarse levels. Cycle time prediction accuracies for each possible mix of MPI tasks and OpenMP threads are given in Table IV.

Table IV
CYCLE TIME PREDICTION ACCURACIES OF THE PERFORMANCE MODEL FOR VARYING ON-NODE MIXES OF MPI TASKS AND OPENMP THREADS.

Hera	1024 Cores	3456 Cores
16 MPI, no OpenMP	97.79%	91.91%
8 MPI $\times$ 2 OpenMP	85.31%	95.57%
4 MPI $\times$ 4 OpenMP	90.72%	95.89%
2 MPI $\times$ 8 OpenMP	99.74%	92.42%
1 MPI $\times$ 16 OpenMP	95.95%	85.06%

Jaguar	1024 Cores	8192 Cores	65536 Cores
16 MPI, no OpenMP	73.14%	79.27%	78.72%
8 MPI $\times$ 2 OpenMP	93.72%	91.14%	85.48%
4 MPI $\times$ 4 OpenMP	96.18%	98.59%	98.76%
2 MPI $\times$ 8 OpenMP	52.84%	63.16%	78.56%
1 MPI $\times$ 16 OpenMP	85.32%	94.42%	98.35%

V. IMPLICATIONS

Our results have important implications for AMG and other applications on future supercomputing platforms that will have even more cores overall, per-chip, and per-node. Per-chip node counts were envisioned by one well-known study to reach the thousands [18], and a later study specific to future exascale systems [19] discussed designs with over 700 cores per chip. Hera had an interconnect unable to effectively handle message passing traffic from nodes with just 16 cores, which was illustrated by the applicability of the model with all network penalties applied. This was not the case for Jaguar, but even so, it is unlikely to expect interconnects to scale so that they can handle message passing traffic from nodes with hundreds or thousands of cores.

This makes the addition of a shared memory programming model like OpenMP very attractive as a means of performance improvement through reducing network traffic. It would ideally serve as an improvement over on-node message passing as well, but this was not the case for the two machines we ran on. The single-node performance of AMG on Hera is best without using OpenMP [3], and in our runs of the AMG solve cycle on Jaguar, the MPI-only performance was better than that of any of the runs that used OpenMP. The threading penalties in our performance model give two reasons for this: limited memory bandwidth and thread migration. The former is an issue inherent to the programming model arising from threads contending for shared resources. The latter can be controlled, but two studies that explored this still found that the best performance came from using some mix of MPI tasks and OpenMP threads other than entirely OpenMP within nodes [3], [4]. Both caused slowdowns on the computation-heavy fine grids that in all cases on Jaguar, and when using more than 4 OpenMP threads per node on Hera, more than offset any gains from reduced network traffic.

Our findings suggest that on future machines, the optimum will be some mix of message passing and shared memory programming that does not use only threading within nodes. Performance models can play a large role in determining the mix that results in the best performance for a particular problem on a particular machine. On machines like Hera, however, the entire set of network penalties applied for most mixes that used message passing within nodes. Thus, the contention issues we reported earlier that impede the scalability of AMG [5] will still remain. Ultimately, algorithmic changes that reduce the amount of communication

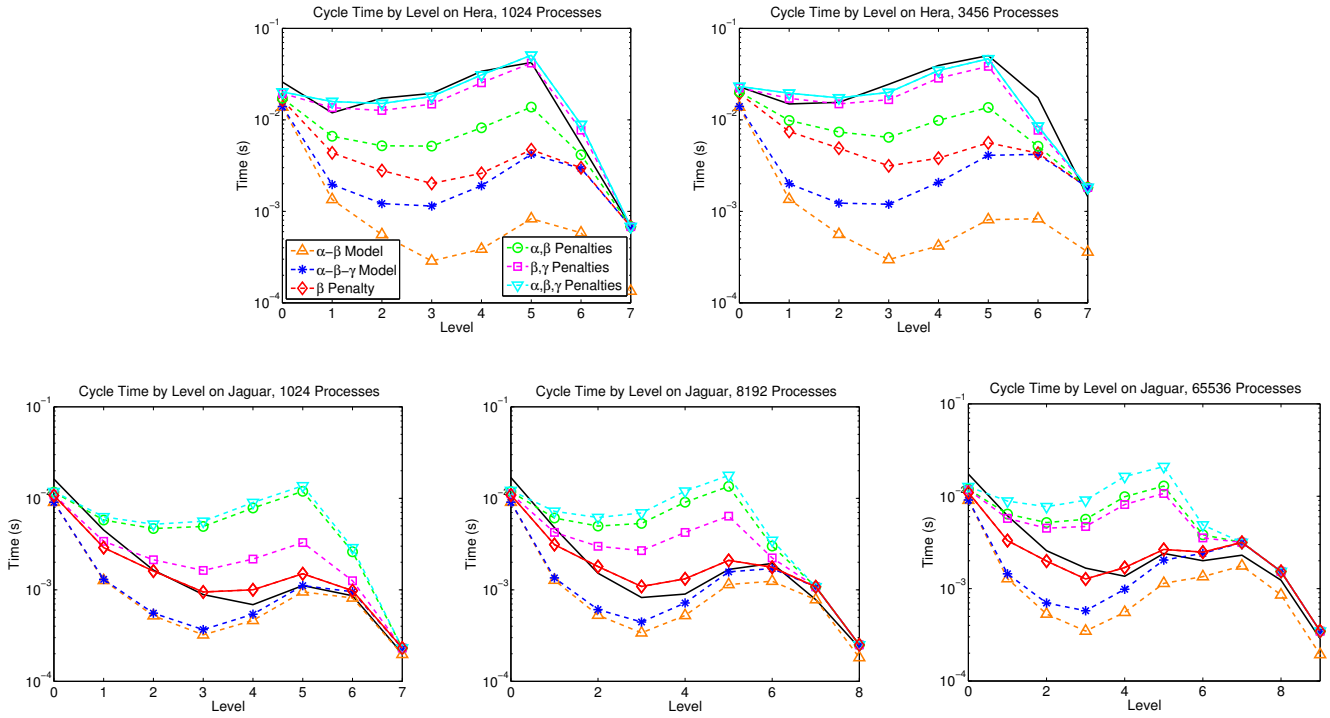

Figure 2. Performance model results using the new β penalty on Hera (top row) and Jaguar (bottom row).

will be necessary to ensure the scalability of AMG to future platforms. Using a hybrid programming model alone, even though it is a better fit to the architecture, will not be enough. The same issues will apply to other applications. An appropriate choice of programming model will be important, but it cannot be expected to solve all scalability issues.

VI. Related Work

There has been substantial interest in hybrid programming over the years, dating back to the emergence of clusters of shared memory multiprocessors. [20] provides a good overview of the issues, and proposes a set of four parameters to assess and model the performance of a hybrid code versus an MPI code. We went in a different direction with our approach for two reasons. The main reason was that we were building on an existing performance model, on which we wanted to base our hybrid model. Another reason was the goal to keep the number of introduced parameters as small as possible for simplicity. Our model also adds two parameters for hybrid modeling instead of four. [21] developed a performance model for hybrid applications based on the parameters in [20]. Their model relies on static analysis of the application being modeled, tying it to codes rather than algorithms.

Two other studies also developed performance models for hybrid applications [22], [23]. The former presents a large framework that involves obtaining relevant traces of a code and then instrumenting and measuring that code on a machine, enabling prediction of the effects of adding OpenMP to an all-MPI code and the effects of running a hybrid code on a larger version of the machine on which measurements were taken. The latter takes the approach of combining models of the OpenMP and MPI aspects of a hybrid application. It relies on measurements of MPI operations on the target architecture for modeling the MPI component rather than communication counts and machine parameters. Validation results are reported on runs using at most 512 cores.

VII. Conclusions

To aid our understanding of hybrid programming and its usefulness in scaling AMG and other applications, we developed a performance model for the AMG solve cycle when programmed in hybrid MPI/OpenMP. Our model is based on our prior performance model of AMG for distributed memory programming models. We updated one of its architectural penalty terms, which resulted in an improved overall fit to observed performance, and included the ability to model multiple threads per MPI process. Our results highlight major factors behind hybrid application performance, and underscore the need for algorithmic change to ensure that AMG scales to future supercomputing platforms.

Future work will be along two main thrusts. The first is extending the model to cover another emerging feature on future supercomputers, namely simultaneous multithreading, which targets cores hosting multiple hardware threads.

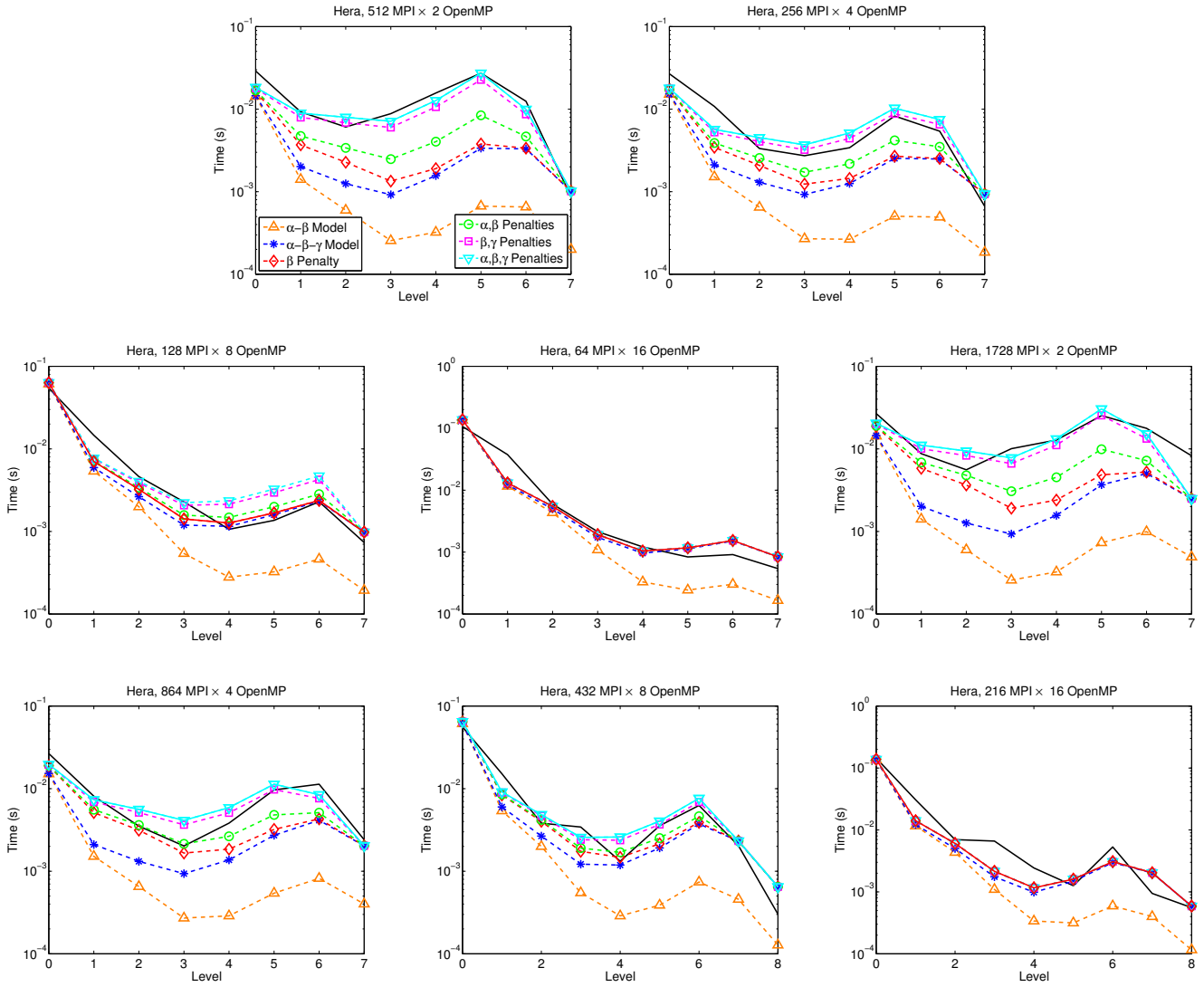

Figure 3. Hybrid model results on Hera.

This is, for example, a key component of the IBM Blue Gene/Q architecture [24] and its first installation, the 20 petaflop Sequoia system [25]. Modeling and analyzing this 16 core/64 hardware thread per node system will be key to help application developers exploit this machine to its fullest. The second is using the model to predict the effectiveness on future machines of hybrid programming and its combination with algorithmic changes currently underway to address the scalability issues we have seen with AMG on multicore clusters. Our ultimate goal is for AMG to scale effectively to future exascale computers.

ACKNOWLEDGEMENTS

We thank the reviewers for their many helpful comments and suggestions, and Luke Olson for a useful conversation while this paper was in development. This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-SC0004131, and performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-533431). It also used resources of the National Center for Computational Sciences at 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. These resources were made available via the Performance Evaluation and Analysis Consortium End Station, a Department of Energy INCITE project. Neither Contractor, DOE, or the U.S. Government, nor any person acting on their behalf: (a) makes any war-

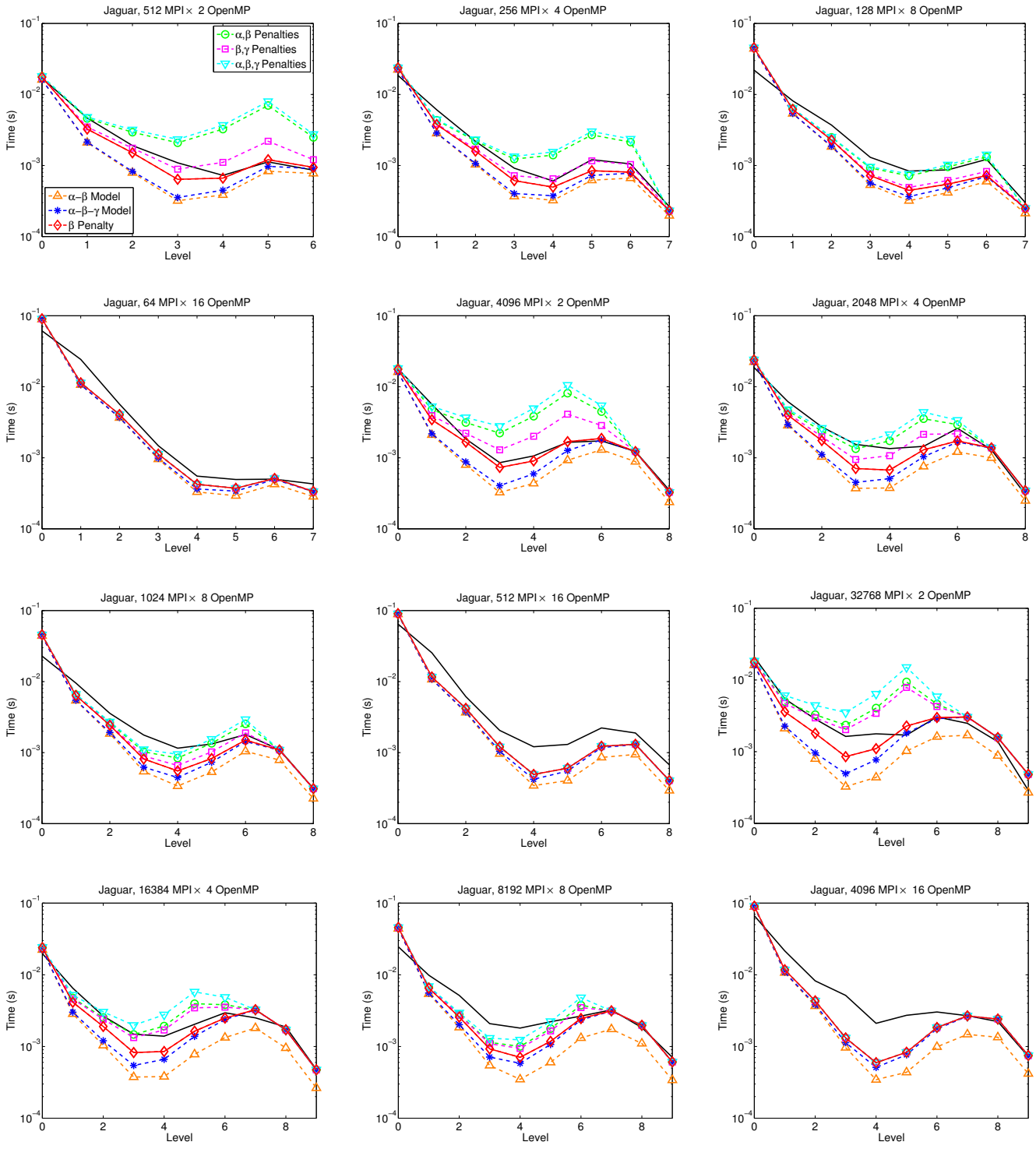

Figure 4. Hybrid model results on Jaguar.

ranty or representation, express or implied, with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in this document.

REFERENCES

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

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

[3] A. H. Baker, M. Schulz, and U. M. Yang, "On the Performance of an Algebraic Multigrid Solver on Multicore Clusters," in *VECPAR'10: 9th International Meeting on High Performance Computing for Computational Science*, Berkeley, CA, June 2010.

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

[5] H. Gahvari, A. H. Baker, M. Schulz, U. M. Yang, K. E. Jordan, and W. Gropp, "Modeling the Performance of an Algebraic Multigrid Cycle on HPC Platforms," in *25th ACM International Conference on Supercomputing*, Tucson, AZ, June 2011.

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

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

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

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

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

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

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

[13] J. M. Bull, "Measuring Synchronisation and Scheduling Overheads in OpenMP," in *First European Workshop on OpenMP*, Lund, Sweden, October 1999.

[14] J. Dongarra and P. Luszczek, "Introduction to the HPCChallenge Benchmark Suite," University of Tennessee, Knoxville, Tech. Rep. ICL-UT-05-01, March 2005.

[15] H. Gahvari, M. Hoemmen, J. Demmel, and K. Yelick, "Benchmarking Sparse Matrix-Vector Multiply in Five Minutes," in *SPEC Benchmark Workshop 2007*, Austin, TX, January 2007.

[16] J. D. McCalpin, "Sustainable Memory Bandwidth in Current High Performance Computers," Advanced Systems Division, Silicon Graphics, Inc., Tech. Rep., 1995.

[17] "AMD Opteron 6200 Series Processors Linux Tuning Guide," http://developer.amd.com/Assets/51803A_OpteronLinuxTuningGuide_SCREEN.pdf.

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

[19] P. Kogge, Ed., *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. DARPA, 2008.

[20] E. Chow and D. Hysom, "Assessing Performance of Hybrid MPI/OpenMP Programs on SMP Clusters," Lawrence Livermore National Laboratory, Tech. Rep. UCRL-JC-143957, May 2001.

[21] L. Adhianto and B. Chapman, "Performance modeling of communication and computation in hybrid MPI and OpenMP applications," *Simulation Modelling Practice and Theory*, vol. 15, pp. 481–491, 2007.

[22] R. Aversa, B. D. Martino, M. Rak, S. Venticinque, and U. Villano, "Performance prediction through simulation of a hybrid MPI/OpenMP application," *Parallel Computing*, vol. 31, pp. 1013–1033, 2005.

[23] X. Wu and V. Taylor, "Performance Modeling of Hybrid MPI/OpenMP Scientific Applications on Large-scale Multicore Cluster Systems," in *14th IEEE International Conference on Computational Science and Engineering*, Dalian, Liaoning, China, August 2011, pp. 181–190.

[24] R. A. Haring, M. Ohmacht, T. W. Fox, M. K. Gschwind, P. A. Boyle, N. H. Christ, C. Kim, D. L. Satterfield, K. Sugavanam, P. W. Coteus, P. Heidelberger, M. A. Blumrich, R. W. Wisniewski, A. Gara, and G. L. Chiu, "The IBM Blue Gene/Q Compute Chip," *IEEE Micro*, vol. 32, pp. 48–60, 2012.

[25] "IBM unveils BlueGene/Q at SC11," https://www.llnl.gov/news/aroundthelab/2011/Nov/ATL-111711_sc.html.