LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-580692

Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned

H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, U. M. Yang

September 13, 2012

3rd International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems
Salt Lake City, UT, United States
November 12, 2012 through November 12, 2012

Disclaimer

Performance Modeling of Algebraic Multigrid on Blue Gene/Q: Lessons Learned

Hormozd Gahvari, William Gropp
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
{gahvari,wgropp}@illinois.edu

Kirk E. Jordan
IBM TJ Watson Research Center
Cambridge, MA 02142
kjordan@us.ibm.com

Martin Schulz, Ulrike Meier Yang
Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{schulzm,umyang}@llnl.gov

***Abstract*—The IBM Blue Gene/Q represents a large step in the evolution of massively parallel machines. It features 16-core compute nodes, with additional parallelism in the form of four simultaneous hardware threads per core, connected together by a five-dimensional torus network. Machines are being built with core counts in the hundreds of thousands, with the largest, Sequoia, featuring over 1.5 million cores. In this paper, we develop a performance model for the solve cycle of algebraic multigrid on Blue Gene/Q to help us understand the issues this popular linear solver for large, sparse linear systems faces on this architecture. We validate the model on a Blue Gene/Q at IBM, and conclude with a discussion of the implications of our results.**

I. Introduction

Since the end of the rise in single-core speeds, new machines have turned to increased parallelism for improvements in performance. The first big step of this trend was multicore processors, and increasing numbers of cores per chip and per node. The IBM Blue Gene/Q takes these multicore processors and adds another source of parallelism to the mix in the form of simultaneous multithreading (SMT), which allows each core to run multiple hardware threads at the same time.

Its new features and many installations make Blue Gene/Q an important machine. A number of places have turned to it to provide their supercomputing needs. In the June 2012 Top 500 supercomputer list [1], thirteen of the top 50 machines, four of the top 10 machines, and two of the top 3 machines are Blue Gene/Qs. The top-ranked machine, the Sequoia system at Lawrence Livermore National Laboratory, features over 1.5 million cores, and will be used to run many large-scale simulations [2].

A natural question to ask with the emergence of a new machine is how well applications will run on it, and what changes need to be made to them to ensure that they do run well on it. We are in particular interested in algebraic multigrid (AMG), a popular solver for large, sparse linear systems of equations that finds many uses in scientific and engineering simulations. AMG scaled very well on the IBM Blue Gene/L [3] and Blue Gene/P [4] platforms, but has since run into problems on emerging architectures with multicore nodes [5], [6]. To aid us in adapting AMG to Blue Gene/Q, we develop in this paper a performance model for the AMG solve cycle on Blue Gene/Q, and use it to analyze observed performance on a prototype system, highlighting issues that need to be addressed when adapting AMG to Blue Gene/Q. Our model builds upon our past work that featured performance models for AMG programmed entirely in a distributed memory model [7] and for AMG programmed in hybrid MPI/OpenMP [8], extending it to cover simultaneous multithreading.

The rest of the paper proceeds as follows. Section II talks about the Blue Gene/Q architecture in more detail and describes the machine on which we ran our experiments. Section III describes AMG and the implementation of it we used in our experiments. Section IV introduces our performance model, beginning with an overview of the model we developed in our past work and then detailing the extensions we make to cover simultaneous multithreading. Section V presents the results of our validation experiments, and is followed by our conclusions in Section VI.

II. Blue Gene/Q

Blue Gene/Q, the latest machine in IBM's highly successful Blue Gene line of supercomputers, represents a notable departure from its predecessors in many ways. The principle of connecting together cores slower than those of competing machines with a good interconnect to obtain efficient, scalable performance remains. However, while Blue Gene/L and Blue Gene/P had small per-node core counts (2 and 4, respectively), Blue Gene/Q features 16 cores per node, and each core can run four hardware threads simultaneously, allowing for up to 64-way per-node parallelism. These threads can be directly accessed by the programmer through common distributed-memory and shared-memory parallel programming models. The interconnect has also changed. Blue Gene/L and Blue Gene/P featured three-dimensional torus interconnects. This has been changed to a five-dimensional torus in Blue Gene/Q. The separate network for handling collective communication that existed in prior Blue Gene machines has been removed; collective functions are now also handled by the torus. More information about the Blue Gene/Q compute chip can be found in [9], and more information about the interconnect can be found in [10].

In this paper, we ran our experiments on Grotius, a Blue Gene/Q located at the IBM TJ Watson Research Center in

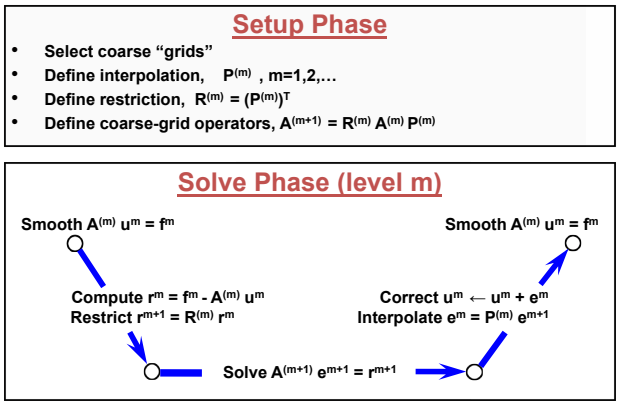

Fig. 1. AMG building blocks. Originally from [7], ©2011 ACM, Inc. Included here by permission.

Yorktown Heights, NY. At the time of our experiments, Grotius was a one rack system, but it has since been expanded to two racks. There are 1024 nodes in a rack, with one 16 core processor per node, for a total of 16,384 cores per rack. The SMT capabilities allow for up to 65,536 parallel tasks to run at the same time. Each processor has a clock rate of 1.6 GHz. The hardware bandwidth between nodes is 40 GB/s. All experiments use IBM's compiler, and the MPI implementation is an IBM-derived version of MPICH2.

III. Algebraic Multigrid

Multigrid (MG) linear solvers are particularly well-suited to parallel computing because their computational cost is linearly dependent on the problem size. This optimal property, also referred to as algorithmic scalability, means that proportionally increasing both the problem size and the number of processors (i.e., weak scaling) results in a roughly constant number of iterations to solution. Therefore, MG methods are popular for large-scale scientific computing and will play a critical role in enabling simulation codes to perform well on emerging machines such as Sequoia. Hence, our interest in the performance of AMG on Blue Gene/Q.

MG's low computational cost results from restricting the original linear system to increasingly coarser grids, which require fewer operations than the fine grid. An approximate solution is determined on the coarsest grid, typically with a direct solver, and is then interpolated back up to the finest grid. On each grid level an inexpensive smoother (e.g., a simple iterative method like Gauss-Seidel) is applied. The process of starting on the fine grid, restricting to the coarse grid, and interpolating back to fine grid again is called a V-cycle, which corresponds to a single MG iteration.

AMG is a special MG method, which does not require geometric grid information for the solution of a sparse linear system $A^{(0)}u = f^{(0)}$.

AMG consists of two phases: setup and solve, see Figure 1. The primary computational kernels in the setup phase are the selection of the variables for the coarser grids, the definition of the interpolation ($P^{(m)}$) and restriction ($R^{(m)}$) operators, and the creation of the coarse grid matrix operator $A^{(m+1)}$ for $m = 0, 1, ..., L-1$, where L is the number of levels. The variables for the $(m+1)$th level as well as the entries in $P^{(m)}$ and $R^{(m)}$ are determined by making use of the coefficients of $A^{(m)}$. These algorithms can be quite complicated. For our experiments, we use the parallel AMG code BoomerAMG in the hypre software library. We use HMIS coarsening [11] with extended+i interpolation [12] truncated to at most 4 coefficients per row and aggressive coarsening with multipass interpolation [13] on the finest level. The coarse grid operator is constructed via a triple matrix product, which, particularly for unstructured problems, leads to increasing matrix density and with it a larger number of neighbor processes and increased communication complexity. The primary computational kernels in the solve phase are the matrix-vector multiplication (MatVec) and the smoothing operator, which for our experiments is hybrid Gauss-Seidel. Hybrid Gauss-Seidel uses a sequential Gauss-Seidel algorithm locally within each process, with delayed updates across processes.

Sparse matrices in BoomerAMG are stored in the ParCSR matrix data structure, in which the matrix A is partitioned by rows into matrices A_k, $k = 0, \ldots, p-1$, where p is the number of MPI processes. A_k is stored locally as two matrices in sequential CSR (compressed sparse row) format, D_k and O_k. D_k contains all entries in A_k whose column indices point to rows stored on process k. O_k contains the remaining entries, which have column indices that point to rows stored on other processes. Matrix-vector multiplication Ax involves computing $A_k x = D_k x^D + O_k x^O$ on each process, where x^D is the portion of x stored locally and x^O is the portion that needs to be sent by other processes. More detail can be found in [14]. OpenMP parallelization is done within MPI tasks using `parallel for` constructs at the loop level. These spawn a number of threads that each execute a portion of the loop being parallelized. Static scheduling is used, which means that the work is divided equally among the threads before the loop starts. The parallelized loops are the ones that perform smoother application and matrix-vector multiplciation.

IV. Performance Model

Here, we present a performance model for the AMG solve cycle on Blue Gene/Q. The model is based on a model we previously developed [7], [8], with an addition to take into account the simultaneous multithreading on Blue Gene/Q. We first present the original model, followed by the modifications to take SMT into account.

A. Original Model

Our model is based on a simple latency-bandwidth model for 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 start-up time and β is the time to send one double-precision floating-point value. The α

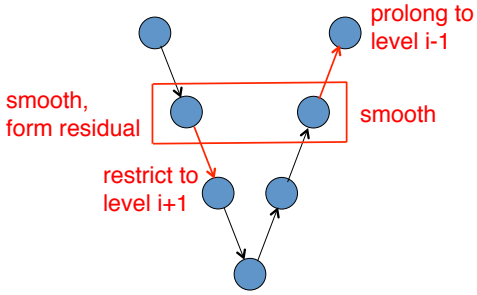

Fig. 2. Component operations in an AMG solve cycle.

term covers both the software overhead and latency involved in message passing, and the β term is tied to the available bandwidth. To handle computation, we multiply the number of floating-point operations by a computation rate, which we allow to vary by level, letting t_i be the time per floating-point operation at level i. The reason for this is that the operator sizes vary by level and the operations involved in an AMG cycle are either matrix-vector multiplication or a very similar operation (smoothing). A past study [15] found that the computation rate for this operation varies depending on the dimensions and number of nonzero entries in the matrix. The other parameters we use are:

- P – total number of cores
- 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

There is no separate mention of the restriction operator here because in our experiments, it is the transpose of the interpolation operator. We assume one smoothing step before restriction and one smoothing step after interpolation, which is the default in BoomerAMG.

We break the solve cycle into a series of individual steps. The total time spent in a cycle with L levels is given by

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

where T_{solve}^{i} is the time spent in the cycle at level i. This in turn splits into three operations, diagrammed in Figure 2:

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

T_{smooth}^{i} is the time spent smoothing on level i, T_{restrict}^{i} is the time spent restricting from level i to level $i+1$, and T_{interp}^{i} is the time spent interpolating from level i to level $i-1$.

We now consider the individual steps. At level i, we have to run a smoother sweep, form the residual, and restrict it to level $i+1$ if i is not the coarsest level. When the computation returns to that level, there will be another smoother sweep followed by interpolation to level $i-1$ if i is not the finest level. We model these operations as matrix-vector multiplication using the appropriate operator, with two floating-point operations per matrix entry. The smoothing time at level i is

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

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

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

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

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

To this model, we now add terms and penalties to cover communication distance, limited bandwidth, and network contention, issues we have observed on actual machines. To cover communication distance, we add a γ term that represents the delay per hop. The corresponding change to the model is replacement of 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. To account for routing delays and possible long hops across a machine room, we assume h is the diameter of the network partition allocated during runtime. h_m depends on the network topology; in the case of the torus network of Blue Gene/Q, $h_m = 1$.

Limited bandwidth is also an issue. Under ideal conditions, the peak hardware bandwidth is rarely achieved in message passing, and the achievable bandwidth is itself rarely reached under the non-ideal conditions under which applications usually run. We take this into account by multiplying β by $\frac{B_{\max}}{B}$, where $B_{\max}$ is the peak aggregate per-node bandwidth in hardware, and B is the measured bandwidth corresponding to β. For β reflecting the cost to send one double-precision floating-point value, $B = \frac{8}{\beta}$.

There is also network contention. The most basic manifestation of this is reduced bandwidth from messages sharing links. If m is the number of messages, and l is the number of links, we augment the previously defined penalty to β as follows. Instead of multiplying by $\frac{B_{\max}}{B}$ as before, we multiply β by the sum $\frac{B_{\max}}{B} + \frac{m}{l}$. For the 5D torus of Blue Gene/Q, $l = 5N$, where N is the number of nodes in the job's partition.

Additional contention penalties to the α and γ terms were derived in [7] to deal with issues resulting from delays in messages accessing the interconnect and traveling through switches. However, we do not use these in our model for Blue Gene/Q, as we do not expect there to be issues arising from this based on our experiences with other machines with strong interconnects, such as the Blue Gene/P [7] and Cray XK6 [8] machines we evaluated in our past work.

TABLE I
PENALTIES TO COMPUTATION RATE t_i TO TAKE INTO ACCOUNT HYBRID MPI/OPENMP AND SIMULTANEOUS MULTITHREADING. t_i IS MULTIPLIED BY THE PENALTY GIVEN IN THE TABLE.

n_{MPI}	1 SMT	2 SMT	3 SMT	4 SMT
1	$P_{\mathrm{OMP}}(16)$	$P_{\mathrm{OMP}}(32)$	$P_{\mathrm{OMP}}(48)$	$P_{\mathrm{OMP}}(64)$
2	$P_{\mathrm{OMP}}(8)$	$P_{\mathrm{OMP}}(16)P_{\mathrm{SMT}}(2)$	$P_{\mathrm{OMP}}(24)P_{\mathrm{SMT}}(2)$	$P_{\mathrm{OMP}}(32)P_{\mathrm{SMT}}(2)$
4	$P_{\mathrm{OMP}}(4)$	$P_{\mathrm{OMP}}(8)P_{\mathrm{SMT}}(2)$	$P_{\mathrm{OMP}}(12)P_{\mathrm{SMT}}(3)$	$P_{\mathrm{OMP}}(16)P_{\mathrm{SMT}}(4)$
8	$P_{\mathrm{OMP}}(2)$	$P_{\mathrm{OMP}}(4)P_{\mathrm{SMT}}(2)$	$P_{\mathrm{OMP}}(6)P_{\mathrm{SMT}}(3)$	$P_{\mathrm{OMP}}(8)P_{\mathrm{SMT}}(4)$
16	1	$P_{\mathrm{OMP}}(2)P_{\mathrm{SMT}}(2)$	$P_{\mathrm{OMP}}(3)P_{\mathrm{SMT}}(3)$	$P_{\mathrm{OMP}}(4)P_{\mathrm{SMT}}(4)$
32	–	$P_{\mathrm{SMT}}(2)$	–	$P_{\mathrm{OMP}}(2)P_{\mathrm{SMT}}(4)$
64	–	–	–	$P_{\mathrm{SMT}}(4)$

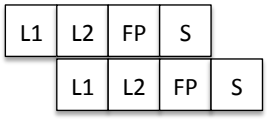

Fig. 3. Dual instruction issue for sparse matrix-vector multiply operations.

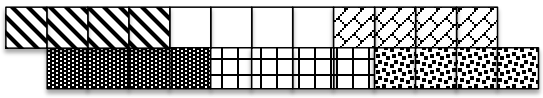

Fig. 4. Instruction issue for up to 6 MatVecs. Each MatVec is colored with a different pattern.

To cover hybrid MPI/OpenMP, we penalize t_i to take into account limited memory bandwidth resulting from threads contending when accessing memory shared by multiple cores. If b_j is the memory bandwidth per thread when running j OpenMP threads, then we multiply t_i by $\frac{b_1}{b_j}$. In [8], there was also a penalty to take into account threads migrating to cores on different sockets; we do not apply this penalty on Blue Gene/Q as each node has only one processor.

B. Additions to Cover Simultaneous Multithreading

The simultaneous multithreading feature on Blue Gene/Q allows for up to four threads to run at the same time. However, while four threads can perform floating-point operations at the same time, four instructions cannot be running at the same time. Instruction issue is only two-way [9], so this limits the achievable parallelism, especially for operations like sparse matrix-vector multiply that are dominated by fetching data from memory.

Figure 3 diagrams dual issue for sparse matrix-vector multiply, and we base our SMT model on this. We treat each MatVec as two loads, a single floating-point instruction (fused multiply-add), and a store. The instructions do not issue simultaneously, so instead of two MatVecs issuing in the same number of cycles as one, we have two MatVecs issuing in $\frac{5}{4}$ of the cycles. The speedup from using two SMT threads instead of one is thus capped at 1.6x. The penalty to t_i is multiplication by $\frac{2}{1.6} = 1.25$.

We use a similar analysis for a greater number of SMT threads, for which we find that the possible speedup is not that much greater. When using 3 SMT threads, 6 MatVecs can be issued in 13 cycles instead of 24, leading to a maximum speedup of about 1.85x. The corresponding penalty to t_i is multiplication by $\frac{3\cdot 13}{24} = 1.625$. When using 4 SMT threads, 4 MatVecs can be issued in 9 cycles instead of 16, leading to a maximum speedup of about 1.78x. The corresponding penalty to t_i is multiplication $\frac{4\cdot 9}{16} = 2.25$. The corresponding instruction issue is diagrammed in Figure 4.

In the MPI only case, we apply these penalties as derived. When adding OpenMP, we have to be careful not to double penalize. This would occur if we are using more than 16 OpenMP threads per node. Measured memory bandwidths when using this many threads would then implicitly contain a penalty for SMT. To avoid double penalizing, we apply the SMT penalty in the hybrid case as follows. Let $P_{\mathrm{SMT}}(t)$ be the SMT penalty given above when using t SMT threads, and let $P_{\mathrm{OMP}}(t)$ be the memory bandwidth penalty $\frac{b_1}{b_t}$ when using t OpenMP threads. If n_{SMT} is the number of SMT threads in use, n_{MPI} is the number of MPI tasks per node, and n_{OMP} is the number of OpenMP threads per node, then we multiply the measured t_i by $P_{\mathrm{SMT}}\left(\min\{n_{\mathrm{MPI}}, n_{\mathrm{SMT}}\}\right) P_{\mathrm{OMP}}(n_{\mathrm{OMP}})$, which takes SMT and OpenMP into account simultaneously. For clarity, a summary of the penalties to t_i for both OpenMP and SMT is given in Table I.

V. MODEL VALIDATION

A. Experimental Setup

We ran 10 AMG solve cycles and measured the amount of time spent in each level, dividing by 10 to get timings for an average solve cycle. Since AMG is used in iterative methods, involving several cycles at least, this measures the time we would expect to see spent in AMG per iteration. As a test problem, we used a 3D 7-point Laplace problem on a cube, with $50 \times 50 \times 25$ points per core. The mapping of MPI tasks per node was the default block mapping, in which each node is filled with MPI tasks before moving onto the next one.

B. Machine Parameters

1) Communication and Computation: Parameters for the communication and computation terms are given in Table II. We determine α and β from the best latency and bandwidth measurements taken by the latency-bandwidth benchmark in the HPC Challenge suite [16]. The benchmark used 8 byte messages to obtain its latency measurements and 2 MB messages to obtain its bandwidth measurements. We determine

TABLE II
MEASURED MACHINE PARAMETERS α, β, γ, AND t_i ON GROTIUS.

Parameter	Value
α	3.15 μs
β	2.19 ns
γ	336 ns
t_0	13.4 ns
t_1	11.4 ns
t_2	6.39 ns

TABLE III
MEMORY BANDWIDTH PER THREAD REPORTED BY STREAM TRIAD.

No. Threads	Bandwidth
1 thread	4117.8 MB/s
2 threads	4064.1 MB/s
3 threads	4037.7 MB/s
4 threads	4035.2 MB/s
6 threads	3921.1 MB/s
8 threads	3505.4 MB/s
12 threads	2267.0 MB/s
16 threads	1741.3 MB/s
24 threads	1109.5 MB/s
32 threads	874.24 MB/s
48 threads	661.06 MB/s
64 threads	512.39 MB/s

γ by formulating α as a function of the number of hops h in the performance model:

$$\alpha(h) = \alpha(1) + \gamma(h-1)$$

We assume $\alpha(1)$ is the best latency reported by the benchmark, and take the worst latency reported by the benchmark to be

$$\alpha(D) = \alpha(1) + \gamma(D-1),$$

where D is the diameter of the network. Then

$$\gamma = \frac{\alpha(D) - \alpha(1)}{D-1}.$$

With one exception, we measured the computation rates t_i using a serial sparse matrix-vector multiply benchmark [17] run simultaneously on all 16 cores of one node to properly stress the memory system, but without using any of the SMT capability so as not to duplicate the SMT penalty already in the model. The exception was t_0. The measured value reported by the benchmark was slower than the corresponding value for Blue Gene/P (reported in [7]), which has a much slower processor, so we instead ran our test problem on one node and measured t_0 from the measured time on level 0, assuming all of it was spent in computation.

2) OpenMP Parameters: We used the STREAM Triad benchmark [18] to compute the memory bandwidth per thread when using OpenMP, taking an average of the reported result over 10 trials. The reported bandwidths are in Table III.

C. Results

Modeled cycle times, measured cycle times, and the corresponding accuracies are given in Table V for 128 cores, Table VI for 1024 cores, and Table VII for 8192 cores. The results are organized with each row containing timings for a given number of MPI tasks per node, and each column containing timings for a given number of SMT threads per core. The number of OpenMP threads is not mentioned explictly, but can be determined by dividing 16x the number of SMT threads per core by the number of MPI tasks per node. If this is 1, then no OpenMP is used at all. More detail is given in Figure 5, which plots modeled and actual times by level. In these plots, the coarsest level is not shown because that level is solved using Gaussian elimination rather than a smoother application, which is what our model assumes. Operator parameters and communication counts used for the model are given in Table VIII for 8192 cores for the thread/task mixes of 1 MPI/64 OpenMP, 8 MPI/8 OpenMP, and 64 MPI/1 OpenMP per node.

TABLE IV
SYNCHRONIZATION OVERHEAD FOR AN OPENMP PARALLEL FOR LOOP FOR THE GIVEN NUMBER OF THREADS.

No. Threads	Overhead
2 threads	4.61 μs
3 threads	6.02 μs
4 threads	6.08 μs
6 threads	6.81 μs
8 threads	7.86 μs
12 threads	9.99 μs
16 threads	12.09 μs
24 threads	18.16 μs
32 threads	22.65 μs
48 threads	33.29 μs
64 threads	47.33 μs

The cycle time prediction accuracies are mostly good, either above 90% or not too far below, when the number of MPI tasks per node is at least one-eighth of the total number of parallel tasks per node possible for the given number of SMT threads in use. When the number of MPI taks per node falls below this is when the accuracy suffers. Then, the model overpredicts the runtime on fine grids and underpredicts it on coarse grids. The overall result is usually an overprediction of the runtime, except for the 4 SMT case, when it is a substantial underprediction.

The OpenMP performance itself merits a closer look. It is very poor, causing serious slowdowns when there are a lot of threads per node, and using OpenMP never beats configurations that use only MPI. Most striking is the performance degradation on coarse grids when there is only one MPI task per node. The conventional wisdom from our past work is that introducing OpenMP improves coarse grid performance at the cost of fine grid performance [8]. The predictions of the performance model reflect this. However, if coarse grid performance is also degraded, then there is something else going on that the model is not capturing. We have a likely culprit: synchronization. A measurement we took with an MPI trace library comparing the results with 64 MPI tasks per node and 1 MPI task and 64 OpenMP threads per node on 1024 cores revealed that in the former case, the slowest process spent 34 ms in MPI operations, with 31 ms in `MPI_Waitall`, but in the latter case, the slowest process spent 67 ms in MPI operations, with 66 ms in `MPI_Waitall`. Using MPI everywhere, for a total of 4096 MPI ranks on 1024 cores,

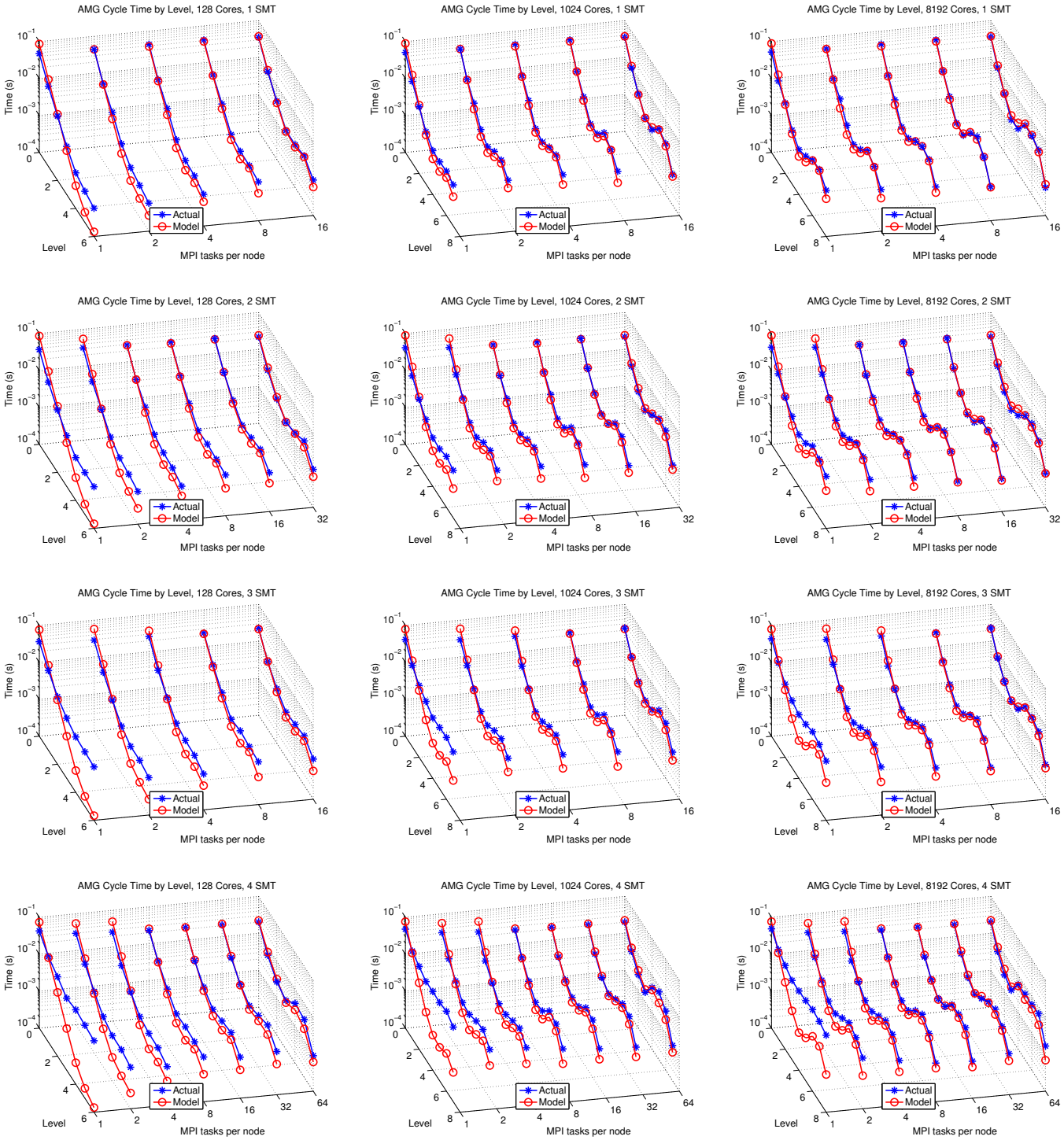

Fig. 5. Level-by-level model vs. actual cycle times.

TABLE V
MODELED CYCLE TIMES, ACTUAL CYCLE TIMES, AND CYCLE TIME PREDICTION ACCURACIES ON 128 CORES.

	Modeled Cycle Times (ms)				Actual Cycle Times (ms)				Accuracies			
	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT
1 MPI	117.5	117.1	103.8	100.6	72.2	60.0	71.5	117.8	37.2%	47.1%	54.8%	85.4%
2 MPI	61.7	76.0	79.3	75.7	65.4	49.3	47.9	52.0	94.2%	45.7%	34.4%	54.3%
4 MPI	55.0	41.4	53.2	69.6	64.5	44.9	42.3	43.1	85.2%	92.2%	74.1%	38.3%
8 MPI	55.8	38.0	34.8	39.1	62.5	43.5	40.5	41.5	89.3%	87.4%	85.8%	94.2%
16 MPI	56.6	39.3	35.5	36.6	54.8	42.6	39.4	39.9	96.6%	92.3%	90.2%	91.5%
32 MPI	–	40.5	–	37.9	–	38.1	–	39.5	–	93.6%	–	96.0%
64 MPI	–	–	–	40.2	–	–	–	40.0	–	–	–	99.3%

TABLE VI
MODELED CYCLE TIMES, ACTUAL CYCLE TIMES, AND CYCLE TIME PREDICTION ACCURACIES ON 1024 CORES.

	Modeled Cycle Times (ms)				Actual Cycle Times (ms)				Accuracies			
	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT
1 MPI	125.9	125.5	112.2	109.0	81.8	73.1	91.0	157.4	46.1%	28.3%	76.7%	69.3%
2 MPI	68.8	83.1	86.4	82.8	73.3	59.1	58.1	63.2	93.8%	59.3%	51.4%	69.0%
4 MPI	60.7	47.1	58.9	75.3	69.6	50.5	48.7	50.5	87.2%	93.3%	78.9%	50.8%
8 MPI	61.0	43.2	40.0	44.3	67.6	49.2	46.8	48.3	90.3%	87.8%	85.3%	91.8%
16 MPI	61.8	44.5	40.6	41.7	59.0	47.6	44.6	45.3	95.4%	93.3%	91.2%	92.0%
32 MPI	–	46.1	–	43.5	–	42.9	–	44.4	–	92.4%	–	98.0%
64 MPI	–	–	–	46.4 ms	–	–	–	47.4	–	–	–	98.0%

TABLE VII
MODELED CYCLE TIMES, ACTUAL CYCLE TIMES, AND CYCLE TIME PREDICTION ACCURACIES ON 8192 CORES.

	Modeled Cycle Times (ms)				Actual Cycle Times (ms)				Accuracies			
	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT	1 SMT	2 SMT	3 SMT	4 SMT
1 MPI	133.2	132.7	119.4	116.2	91.8	85.8	106.7	182.7	55.0%	45.4%	88.0%	63.6%
2 MPI	75.3	89.7	92.9	89.4	86.0	66.5	67.4	72.5	91.6%	65.1%	62.2%	76.7%
4 MPI	67.8	54.3	66.1	82.5	76.5	57.6	55.8	59.3	88.7%	94.1%	81.5%	60.9%
8 MPI	68.1	50.3	47.0	51.4	72.9	54.9	53.1	54.7	93.3%	91.6%	88.5%	93.8%
16 MPI	69.5	52.1	48.3	49.3	64.7	53.1	50.2	51.8	92.6%	98.1%	96.2%	95.2%
32 MPI	–	54.2	–	51.6	–	48.9	–	51.2	–	89.2%	–	99.3%
64 MPI	–	–	–	55.0	–	–	–	57.2	–	–	–	96.2%

should not result in more time spent in MPI operations than using MPI only between nodes, which only uses a total of 64 MPI ranks. However, the OpenMP synchronization costs rise noticeably with increasing thread counts on Blue Gene/Q. Table IV shows the results of a run of the EPCC OpenMP synchronization benchmark [19] for an OpenMP `parallel for` loop. Combining this with the limitations from the dual instruction issue and waiting for MPI tasks to finish communicating would lead to an unfavorable synchronization scenario for the case of 64 OpenMP threads per node. Additional data, plotted in Figure 6, shows level-by-level performance with just 1 MPI task per node and a varying number of OpenMP threads. Though fine grid performance mostly improves as the number of threads increases, coarse grid performance actually declines.

The best observed performance in all cases occurred when using 32 MPI tasks per node, with no OpenMP. This is a source of disappointment, as it meant BoomerAMG was not taking advantage of the full parallelism presented to us by the machine. According to the model, this was inherent to the application and the architecture – the limited ability of the dual instruction issue to issue enough simultaneous floating point operations for the MatVecs and smoother applications meant that the 3 SMT case resulted in the best predicted speedup. The results bore this out in that so long as there were at least 4 MPI tasks per node, the 3 SMT case resulted in the best performance compared to other numbers of SMT threads for the same number of MPI tasks. Unfortunately, it was not possible to test using 24 or 48 MPI tasks per node (paired with 2 OpenMP threads per task and no OpenMP threads, respectively), as the Blue Gene/Q scheduler requires the number of MPI tasks per node to be a power of two. Those cases might have been superior to 32 MPI tasks per node.

The other noticeable discrepancy between the modeled and measured results is that the model predicted the best performance to occur using 8 MPI tasks per node regardless of the number of SMT threads used. However, the all-MPI case was the actual best case except for the 4 SMT case. The explanation here is the unexpected coarse grid slowdown we saw when using OpenMP. If this is going to be a recurring problem when running BoomerAMG on Blue Gene/Q, something will need to be done to address it; we will discuss this further in our concluding remarks.

VI. CONCLUSIONS

To better understand the issues faced by algebraic multigrid on Blue Gene/Q, and how to best adapt it to this new and important architecture, we developed a performance model of the AMG solve cycle that took into account all-MPI performance, hybrid MPI/OpenMP performance, and its most

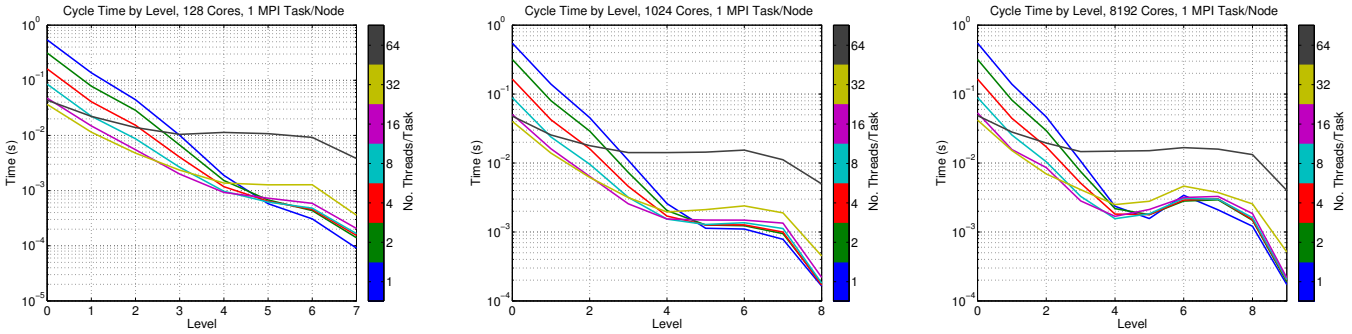

Fig. 6. Level-by-level times with 1 MPI task per node and an increasing number of OpenMP threads on 128, 1024, and 8192 cores.

important feature, simultaneous multithreading. The overall prediction accuracy was very good for the threading configurations which resulted in the best performance. However, there were significant discrepancies between the modeled and measured results for configurations that made significant use of OpenMP. Though we believe this is due to a synchronization phenomenon the model did not explain, the model nonetheless is telling us something important even in this case.

The first studies of AMG that involved running BoomerAMG in hybrid MPI/OpenMP configurations [5], [6] did not find the degradation on coarse grids when using OpenMP that we saw on Blue Gene/Q. There was also some unexpected and then-unexplained coarse grid degradation observed on a Cray XK6 when using OpenMP [8], so if this the same phenomenon, then it is something that will have to be dealt with on emerging and future machines. As mentioned earlier, future machines will have many more cores per node, and with simultaneous multithreading added to the mix, this means that the available on-node parallelism will soon become truly massive. Running all MPI tasks per node is not going to work well on these machines given all the message passing traffic this generates. We are already seeing this with AMG on Blue Gene/Q, with 64 MPI tasks per node worse than a number of other possible MPI/OpenMP mixes in spite of having the best fine grid performance. The predictions provided by the performance model hint that successfully incorporating a threaded programming model like OpenMP is going to be important in obtaining the best possible performance.

Simultaneous multithreading brings us to another important point when looking to the future. The way it works on Blue Gene/Q, with the dual instruction issue, limits the achievable parallelism when performing sparse matrix-vector multiply and a similar operation in relaxation. In their pursuit of increasing levels of parallelism, future machines are likely to run more simultaneous threads, but as we have seen, their effectiveness is very much dependent on the interplay between the architecture and the application. Applications will certainly have to adapt, though on the other end, the architecture needs to provide increased parallelism in such a way that the applications will be fundamentally able to take advantage of it. The dual instruction issue limitation on Blue Gene/Q and its adverse impact on sparse matrix-vector multiply highlights this issue. While applications heavy on this operation will need to adapt somehow to provide as many simultaneous floating point operations as they can, the fundamental memory-bound nature of the operation is such that there is only so much that can be done at the application level, and designers of subsequent machines might want to consider providing more parallelism for moving data from memory in order to allow for applications dominated by SpMV to better exploit the available parallelism for computation.

In the future, we will examine what we have observed in more detail and make use of this information as we adapt AMG to Blue Gene/Q. In particular, we will run additional experiments to determine the precise cause of the performance degradation we have seen when using OpenMP, and move from there to either adapting the runtime settings, the code, or both. We will also refine our performance model, and use it to predict the effectiveness of the adaptations to AMG that we will be making. We would also like to be able to use the model to determine the best mix of MPI tasks and OpenMP threads to run when given a particular problem on a particular number of cores. As machines get even larger and contain even more on-node parallelism, the optimal mix of threads and tasks is going to be more likely to vary with the problem being run, and being able to determine this automatically would relieve users of an otherwise time-consuming task.

ACKNOWLEDGEMENTS

This work was supported in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-SC0004131, and performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-580692). It also made use of the computing resources of the IBM TJ Watson Research Center. Neither Contractor, DOE, or the U.S. Government, nor any person acting on their behalf: (a) makes any warranty or representation, express or implied, with respect to the information contained in this document; or (b) assumes any liabilities with respect to the use of, or damages resulting from the use of any information contained in this document.

REFERENCES

[1] "TOP500 List - June 2012," http://www.top500.org/list/2012/06.

TABLE VIII

OPERATOR PARAMETERS AND COMMUNICATION COUNTS USED FOR THE MODEL FOR SELECTED MIXES OF MPI TASKS AND OPENMP THREADS PER NODE ON 8192 CORES. FROM LEFT TO RIGHT, FOR EACH OPERATOR: AVERAGE SENDS PER ACTIVE PROCESS, MAXIMUM SENDS, MAXIMUM ELEMENTS SENT, NUMBER OF UNKNOWNS, NUMBER OF NONZERO ENTRIES PER ROW, AND ACTIVE PROCESSES.

	Solve Operator, 1 MPI/64 OpenMP						Interpolation Operator, 1 MPI/64 OpenMP			
Lev.	Avg Sends	Max Sends	Elems. Sent	Unknowns	NNZ/row	Active Procs.	Avg Sends	Max Sends	Elems. Sent	NNZ/row
0	5.2	6	60000	512000000	7.0	512	14.5	19	7479	2.1
1	15.8	24	16806	40896779	18.0	512	15.0	22	2213	3.3
2	19.2	26	8784	8929521	51.6	512	15.5	24	607	3.6
3	19.7	26	3311	1203569	82.0	512	14.4	22	174	3.6
4	19.8	26	1470	163695	106.9	512	12.3	22	77	3.7
5	34.7	61	668	18132	102.9	512	9.0	28	43	3.6
6	58.5	138	426	2023	82.8	502	4.8	80	90	2.9
7	52.9	132	214	248	57.2	202	2.5	67	85	2.5
8	26.7	32	62	36	29.9	33	1.7	18	18	1.8
9	4.0	4	4	5	5.0	5	–	–	–	–
	Solve Operator, 8 MPI/8 OpenMP						Interpolation Operator, 8 MPI/8 OpenMP			
Lev.	Avg Sends	Max Sends	Elems. Sent	Unknowns	NNZ/row	Active Procs.	Avg Sends	Max Sends	Elems. Sent	NNZ/row
0	5.6	6	15000	512000000	7.0	4096	16.0	20	2053	2.1
1	17.8	24	4381	38687050	19.0	4096	16.6	22	616	3.4
2	22.2	26	2437	8005047	53.3	4096	16.5	25	188	3.6
3	22.6	26	1008	938688	83.1	4096	13.8	23	77	3.7
4	38.2	60	552	97778	96.7	4096	9.4	30	48	3.7
5	70.9	148	436	10081	91.0	3818	4.8	74	110	3.5
6	68.4	189	325	1253	73.0	1054	3.2	92	92	3.1
7	53.0	108	108	148	54.1	148	2.7	65	65	2.9
8	20.1	22	22	23	21.1	23	0.6	9	9	0.6
9	1.0	1	1	2	2.0	2	–	–	–	–
	Solve Operator, 64 MPI/1 OpenMP						Interpolation Operator, 64 MPI/1 OpenMP			
Lev.	Avg Sends	Max Sends	Elems. Sent	Unknowns	NNZ/row	Active Procs.	Avg Sends	Max Sends	Elems. Sent	NNZ/row
0	5.8	6	3750	512000000	7.0	32768	15.6	20	480	1.9
1	19.0	26	1135	38245922	19.8	32768	16.8	23	272	3.4
2	23.7	26	898	7528827	55.8	32768	16.1	25	101	3.7
3	29.1	43	493	798957	85.6	32768	11.7	26	57	3.7
4	69.4	126	430	77858	96.3	30952	5.1	64	94	3.7
5	85.8	158	326	8507	91.9	7542	3.5	101	122	3.6
6	74.3	180	300	1042	76.9	1009	2.7	128	128	2.8
7	49.2	124	124	145	50.2	145	2.0	62	62	2.1
8	12.8	16	16	17	13.8	17	1.0	10	10	1.2
9	2.0	2	2	3	3.0	3	–	–	–	–

[2] "NNSA's Sequoia supercomputer ranked as world's fastest," https://www.llnl.gov/news/newsreleases/2012/Jun/NR-12-06-07.html.

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

[4] 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.

[5] 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.

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

[7] 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.

[8] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, and U. M. Yang, "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.

[9] 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.

[10] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, "The IBM Blue Gene/Q Interconnection Network and Message Unit," in *Supercomputing 2011*, Seattle, WA, November 2011.

[11] 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.

[12] 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.

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

[14] 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.

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

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

[17] 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.

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

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