LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-427763

Challenges of Algebraic Multigrid across Multicore Architectures

Allison Baker, Todd Gamblin, Martin Schulz, Ulrike Yang

April 13, 2010

Supercomputing 2010
New Orleans, LA, United States
November 13, 2010 through November 19, 2010

Disclaimer

Challenges of Algebraic Multigrid across Multicore Architectures

Allison H. Baker, Todd Gamblin, Martin Schulz, and Ulrike Meier Yang
Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 7000 East Avenue, Livermore CA 94550
{abaker, tgamblin, schulzm, umyang}@llnl.gov

***Abstract*—Algebraic multigrid (AMG) is a popular solver for large-scale scientific computing and an essential component of many simulation codes. AMG has shown to be extremely efficient on distributed-memory architectures. However, when executed on modern multicore architectures, we face new challenges that can significantly deteriorate AMG's performance. We examine its performance and scalability on three disparate multicore architectures: a cluster with four AMD Opteron Quad-core processors per node (Hera), a Cray XT5 with two AMD Opteron Hex-core processors per node (Jaguar), and an IBM BlueGene/P system with a single Quad-core processor (Intrepid). We discuss our experiences on these platforms and present results using both an MPI-only and a hybrid MPI/OpenMP model. We also discuss a set of techniques that helped to overcome the associated problems, including thread and process pinning and correct memory associations.**

I. Introduction

Sparse iterative linear solvers are a critical part of many simulation codes and often account for a significant fraction of their total run times. Therefore, the performance and scalability of linear solvers on modern multicore machines is of great importance for enabling large-scale simulations on these new high-performance architectures. Furthermore, of particular concern for multicore architectures is that for many applications, as the number of cores per node increases, the linear solver time becomes an increasingly larger portion of the total application time [1]. In other words, under strong scaling the linear solver scales more poorly than the remainder of the application code.

The AMG solver in *hypre* [2], called BoomerAMG, has effective coarse-grain parallelism and minimal inter-processor communication, and, therefore, demonstrates good weak scalability on distributed memory machines (as shown for weak scaling on BlueGene/L using 125,000 processors [3]). However, the emergence of multicore architectures in high-performance computing has forced a re-examination of the *hypre* library and the BoomerAMG code. In particular, BoomerAMG's performance can be harmed by the new node architectures due to multiple cores and sockets per node, different levels of cache sharing, multiple memory controllers, non-uniform memory access times, and reduced bandwidth. With the MPI-only model expected to be increasingly insufficient as the number of cores per node increases, we have turned our focus to a hybrid programming model for our AMG code, in which a subset of or all cores on a node operate through a shared memory programming model like OpenMP. In practice, few high-performance linear solver libraries have implemented a hybrid MPI/OpenMP approach, and for AMG in particular, obtaining effective multicore performance has not been sufficiently addressed.

In this paper we present a performance study of BoomerAMG on three radically different multicore architectures: a cluster with four AMD Opteron Quad-core processors per node (Hera), a Cray XT5 with two AMD Opteron Hex-core processors (Jaguar), and an IBM BlueGene/P system with a single Quad-core processor (Intrepid). We discuss the performance of BoomerAMG on each architecture (Section V) and detail the modifications that were necessary to improve performance (i.e., the "lessons learned", Section VI). In particular, we make the following contributions:

- A comprehensive study of the performance of AMG on the three leading classes of HPC platforms;
- An evaluation of the threading performance of AMG using a hybrid OpenMP/MPI programming model;
- Optimization techniques, including a multi-core support library, that significantly improves both performance and scalability of AMG;
- A set of lessons learned from our experience of running a hybrid OpenMP/MPI application that applies well beyond the AMG application.

The remainder of this paper is organized as follows: in Section II we present an overview of the AMG solver and its implementation. In Section III we introduce the two test problems we use for our study, followed by the three evaluation platforms in Section IV. In Section V we provide a detailed discussion of the performance of AMG on the three platforms, and in Section VI we discuss lessons learned as well as optimization steps. In Section VII we conclude with a few final remarks.

II. Algebraic multigrid (AMG)

Algebraic multigrid (AMG) methods [4], [5], [6] are popular in scientific computing due to their robustness when solving large unstructured sparse linear systems of equations. In particular, *hypre*'s BoomerAMG plays a critical role in a number of diverse simulation codes. For example, at Lawrence Livermore National Laboratory (LLNL), BoomerAMG is used in simulations of elastic and plastic deformations of explosive materials and in structural dynamics codes. Its scalability has enabled the simulation of problem resolutions that were previously unattainable. Elsewhere, BoomerAMG has been critical in the large-scale simulation of accidental fires and explosions, the modeling of fluid pressure in the eye, the speed-up of simulations for Maxillo-facial surgeries to correct deformations [7], sedimentary basin simulations [8], and the simulation of magnetohydrodynamics (MHD) formulations of fusion plasmas (e.g., the M3D code from the Princeton Plasma Physics Laboratory).

In this section, we first give a brief overview of multigrid methods, and AMG in particular, and then describe our implementation.

A. Algorithm overview

Multigrid is an iterative method, and, as such, a multigrid solver starts with an initial guess at the solution and repeatedly generates a new or improved guess until that guess is close in some sense to the true solution. Multigrid methods generate improved guesses by utilizing a sequence of smaller grids and relatively inexpensive smoothers. In particular, at each grid level, the smoother is applied to reduce the high-frequency error, then the improved guess is transferred to a smaller, or coarser, grid. The smoother is applied again on the coarser level, and the process continues until the coarsest level is reached where a very small linear system is solved. The goal is to have significantly eliminated error once the coarsest level has been reached. The improved guess is then transferred back up (interpolated) to the finest grid, resulting in a new guess on that original grid. See Figure 1. Effective interplay between the smoothers and the coarse-grid correction process is critical for good convergence.

The advantage of a multilevel solver is two-fold. By operating on a sequence of coarser grids, much of the computation takes place on smaller problems and is, therefore, computationally cheaper. Second, and perhaps most importantly, if the multilevel solver is designed well, the computational cost will only depend linearly on the problem size. In other words, a sparse linear system with N unknowns is solved with O(N) computations. This translates into a scalable solver algorithm, and, for this reason, multilevel methods are often called optimal. In contrast, many common iterative solvers (e.g., Conjugate Gradient) have the non-optimal property whereby the number of iterations required to converge to the solution increases with increasing problem size. An algorithmically scalable solution is particularly attractive for parallel computing because distributing the computation across a parallel machine enables the solution of increasing larger systems of equations. While multigrid methods can be used as standalone solvers, they are more frequently used in combination with a simpler iterative solver (e.g., Conjugate Gradient or GMRES), in which case they are referred to as preconditioners.

AMG is a particular multigrid method with the distinguishing feature that no problem geometry is needed; the "grid" is simply a set of variables. This flexibility is useful for situations when the grid is not known explicitly or is unstructured. As a result, coarsening and interpolation processes are determined entirely based on the entries of the matrix, and AMG is a fairly complex algorithm.

AMG has two separate phases, the *setup* and the *solve* phase. In the setup phase, the coarse grids, interpolation operators, and coarse-grid operators must all be determined for each of the coarse-grid levels. AMG coarsening is non-trivial, particularly in parallel where care must be taken at processor boundaries (e.g., see [9], [10]). Coarsening algorithms typically determine the relative strength of the connections between the unknowns based on the size of matrix entries and often employ an independent set algorithm. Once the coarse grid is chosen for a particular level, the interpolation operator is determined. Forming interpolation operators in parallel is also rather complex, particularly for the long-range variety that are required to keep memory requirements reasonable on large numbers of processors (e.g., see [11]). Finally the coarse grid operators (coarse-grid representations of the fine-grid matrix) must be determined in the setup phase. These are formed via a triple matrix product. While computation time for the AMG setup phase is problem dependent, it is certainly non-negligible and may have a significant effect on the total run time. In fact, if a problem converges rapidly, the setup time may even exceed the solve time.

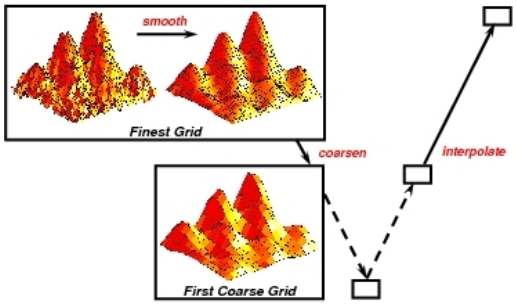


Fig. 1: Illustration of one multigrid cycle.

The AMG solve phase performs the multilevel iterations (often referred to as cycles). The primary components of the solve phase are applying smoother, which is similar to a matrix-vector multiply (MatVec), and restricting and interpolating the error (both MatVecs). AMG is commonly used as a preconditioner for Conjugate Gradient or GMRES, and in that case, the MatVec time dominates the solve phase run-time (roughly 60%), followed by the smoother (roughly 30%). Many reasonable algorithmic choices may be made for each AMG component (e.g., coarsening, interpolation, and smoothing), and the choice affects the convergence rate. For example, some coarsening algorithms coarsen "aggressively" [6], which results in lower memory usage but often a higher number of iterations. The total solve time is, of coarse, directly related to the number of iterations required for convergence.

B. Implementation

In this paper, we use a slightly modified version of the AMG code included in the *hypre* software library [12]. While AMG provides a wide range of input parameters, which can be used to fine tune the application, we chose the following options to generate the results in this paper. For coarsening, we use PMIS [13] and employ aggressive coarsening on the first level to achieve low complexities and improved scalability. For interpolation we use multipass interpolation [6], [14] on the first coarse level and extended+i(4) interpolation [11] on the remaining levels. The smoother is a hybrid symmetric Gauss-Seidel parallel smoother, and, because AMG is most commonly used as a preconditioner for both symmetric and nonsymmetric problems, our experiments use AMG as a preconditioner for GMRES(10).

Some parallel aspects of the AMG algorithm are dependent on the number of tasks and the domain partitioning among MPI tasks and OpenMP threads. The parallel coarsening algorithm and hybrid Gauss-Seidel parallel smoother are two examples of such components. Therefore, one cannot expect the number of iterations to necessarily be equivalent when, for example, comparing an experimental setup with 16 threads per node to one with 16 tasks per node. For this reason, we use average cycle times (instead of the total solve time) where appropriate to ensure a fair comparison.

While we tested both MPI and OpenMP in the early stages of BoomerAMG's development, we later on focused on MPI due to disappointing performance of OpenMP at that time. We use a parallel matrix data structure that was mainly developed

with MPI in mind. Matrices are assumed to be distributed across p processors in contiguous blocks of rows. On each processor, the matrix block is split into two parts, one of which contains the coefficients that are local to the processor. The second part, which is generally much smaller than the local part, contains the coefficients whose column indices point to rows stored on other processors. Each part is stored in compressed sparse row (CSR) format. The data structure also contains a mapping that maps the local indices of the off-processor part to global matrix indices as well as a information needed for communication. A complete description of the parallel matrix structure used can be found in [15]. The AMG algorithm requires various matrices besides the original matrix, such as the interpolation operator and the coarse grid operator. While the generation of the local parts of these operators generally can be performed as in the serial case, the generation of the off-processor part as well as the communication package and mapping of the original matrix is fairly complex and depends on the amount of ghostlayer points, i.e. those points that are immediate neighbors to a point i but are located on another processor. Therefore, a large number of ghostlayer points, which can be caused by a non-optimal partitioning, will not only affect communication but also increase computation. On the other hand, replacing pk MPI tasks by p MPI tasks with k OpenMP threads each that do not require ghostlayer points could lead to improved performance. We will evaluate the number of ghostlayer points on the first level for the problems considered here in the following section.

Our recent efforts with the AMG code have been aimed at increasing the use of OpenMP in the code. The solve phase, composed primarily of the MatVec and Smoother kernels, can be threaded in a straightforward manner (contiguous subsets of a processor's rows are operated on by each thread). The setup phase, however, is more complex and has not been completely threaded. The triple matrix product that forms the coarse-grid operator is threaded. Only a part of the interpolation operators is threaded, since the data structure is not particularly OpenMP friendly. This is due to the fact that the matrices are compressed and the total number of nonzeros for the operators that need to be generated is not known ahead of time. We currently have no coarsening routine that uses OpenMP, and therefore used the fastest coarsening algorithm available, PMIS, to decrease the time spent in the nonthreaded part of the setup. This algorithm leads to somewhat increased number of iterations and decreased scalability compared to HMIS.

III. Test problems

We use two test problems for our performance study.

A. Problem descriptions

The first is a 3D Laplace problem on a box with Dirichlet boundary conditions with a seven-point stencil generated by finite differences. We refer to this problem as "Laplace". When solving this problem on *Hera* and *Intrepid*, the domain is a unit cube, however on *Jaguar*, the domain size is a box of size $N \times N \times 0.9N$, to allow more optimal partitioning when using 6 or 12 threads per node.

We designed a second problem to represent a wider range of applications. This problem is a 3D diffusion problem on a more complicated grid. The 2D projection of this grid is shown in Figure 2 (the grid extends equally out of the page in the third dimension with 4 points). We refer to this problem as "MG" because of the shape of the grid. This problem also has jumps as well as anisotropies, which appear in many applications. The finite difference stencil for each of the eight parts of the grids are given in the figure as well. In particular, part 7 is anisotropic and there are jumps between parts 0-3 in the "M" part of the grid. Each part has 48 grid points. This grid can then be further refined by refinement factors in each direction to generate large problems for many processors.

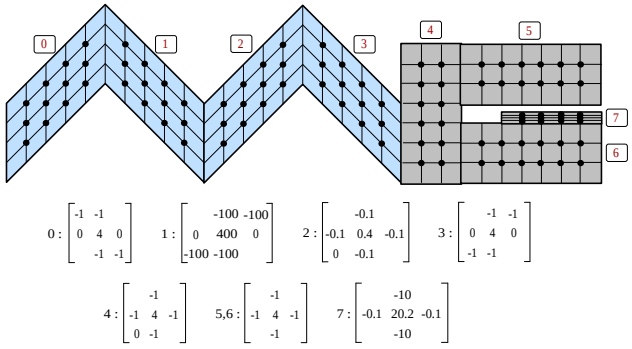


Fig. 2: The "MG" test problem grid and its associated finite difference stencils.

B. Ghostlayer Analysis

Our goal is to evaluate the number of ghostlayer points for each MPI task for both test problems and then analyze the effect on them, when we use threading within MPI tasks. We consider here only ghostlayer points in the first layer, i.e., immediate neighbor points of a point i that are located on a neighbor processor. Note that this analysis only applies to the first AMG level, since the matrices on coarser levels will have larger stencils, but fewer points within each MPI task. However since we aggressively coarsen the first grid, leading to significantly smaller coarse grids, the evaluation of the finest level generally requires the largest amount of computations and should take more than half of the time. Also, choosing a nonoptimal partitioning for the MPI tasks will generally lead to nonoptimal partitionings on the lower levels.

Let us first consider the Laplace problem. Assume that there are $p_x p_y p_z$ cores and a grid of size $N_x \times N_y \times N_z$ partitioned into $p_x \times p_y \times p_z$ subdomains. For simplicity assume that N_k is divisible by p_k for $k = x, y, z$. Since we want to consider both OpenMP threads and MPI tasks, we introduce parameters s_k, and $\tilde{p}_k$, $k = x, y, z$, with $\tilde{p}_k = \frac{p_k}{s_k}$. Now the number of MPI tasks is $\tilde{p}_x \tilde{p}_y \tilde{p}_z$ and the number of threads per MPI task is $s = s_x s_y s_z$. This leads to a subdomain size of $\frac{N_x}{\tilde{p}_x} \times \frac{N_y}{\tilde{p}_y} \times \frac{N_z}{\tilde{p}_z}$ for each MPI task, leading to $2(\frac{N_x}{\tilde{p}_x}\frac{N_y}{\tilde{p}_y} + \frac{N_x}{\tilde{p}_x}\frac{N_z}{\tilde{p}_z} + \frac{N_y}{\tilde{p}_y}\frac{N_z}{\tilde{p}_z})$ ghost layer points. In order to get the complete number of ghostlayer points one needs to multiply the number of ghost layer points for each MPI task by the number of total MPI tasks, $\tilde{p}_x \tilde{p}_y \tilde{p}_z$ and subtract the surface of the complete domain, which is $2(N_x N_y + N_x N_z + N_y N_z)$.

Now we can define the total number of ghostlayer points for the Laplace problem

$$\nu_L(s) = 2(\frac{p_z}{s_z} - 1)N_x N_y + 2(\frac{p_y}{s_y} - 1)N_x N_z + 2(\frac{p_x}{s_x} - 1)N_y N_z. \quad (1)$$

For the Laplace problem on Hera and Intrepid, we choose the following values for the parameters: $N = N_x = N_y = N_z$, $p_x = p_y = 2p$, and $p_z = 4p$. Inserting these values in (1), we obtain $\nu_L(s) = [(\frac{8}{s_z} + \frac{4}{s_y} + \frac{4}{s_y})p - 6]N^2$. We are interested in the

effect of threading on the ghostlayer points for large runs, since future architectures will have millions or even billions of cores. In particular, if p becomes large, the surface of the domain becomes negligible, and we get the following result that we refer to as the "ghostlayer ratio":

$$\rho_L(s) = \lim_{p\to\infty} \frac{\nu_L(1)}{\nu_L(s)} = 4/(\frac{2}{s_z} + \frac{1}{s_y} + \frac{1}{s_x}). \quad (2)$$

Note that for the Laplace problem, if we choose 16 threads, i.e. $s_x = s_y = 2$, and $s_z = 4$, $\rho_L(16) = 2.67$. On Jaguar, we chose somewhat different sizes to enable numbers that can be divided by 6 and 12. Here, $N_x = N_y = N$, $N_z = 0.9N$, $p_x = p_y = 4p$, and Inserting these numbers into Equation 1, one obtains the following ratio for Jaguar

$$\rho_L^J(s) = 17/(\frac{5}{s_z} + \frac{6}{s_y} + \frac{6}{s_x}). \quad (3)$$

For 12 threads, with $s_x = s_y = 2$ and $s_z = 3$, $\rho_L^J(12) = 2.22$.

The MG problem is generated starting with the grid given in Figure 2, and its uniform extension into the third dimension. It consists of eight parts with $n_x^i \times n_y^i \times n_z$ points each, $i = 0, ..., 7$, with $n_z = 4$ and the n_x^i and n_y^i as shown in Figure 2. It then is further refined using the refinement factors $R_x = p_x r_x$, $R_y = p_y r_y$ and $R_z = p_z r_z$, requiring $8p_x p_y p_z$ cores. The surface of the total grid consists of $192R_xR_y + 240R_xR_z + 64R_yR_z$ points. If we include threads and define $\tilde{p}_k$ and s_k for $k = x, y, z$, as above, we need to define new refinements $\tilde{r}_k$, $k = x, y, z$, with $\tilde{r}_k = s_k r_k$ to ensure that we solve the same problem. Note that $\tilde{r}_k \tilde{p}_k = R_k$. Combining this information, one can determine the total number of ghost layer points:

$$\begin{aligned}
\nu_{MG}(s) &= \sum_{i=0}^{7} 2(n_x^i n_y^i R_x R_y \frac{p_z}{s_z} \\
&\quad + n_x^i n_z^i R_x R_z \frac{p_y}{s_y} + n_y^i n_z^i R_y R_z \frac{p_x}{s_x}) \\
&\quad - 192R_xR_y - 240R_xR_z - 64R_yR_z \\
&= 192(\frac{p_z}{s_z} - 1)R_xR_y + (224\frac{p_y}{s_y} - 240)R_xR_z \\
&\quad + (200\frac{p_x}{s_x} - 64)R_yR_z.
\end{aligned}$$

On Hera and Intrepid, we will consider two versions of the MG-problem. For the first version, denoted "MG-1", we choose $p_x = p_y = 2p$, and $p_z = 4p$, as for the Laplace problem. For the second version, denoted "MG-2", $p_x = p_y = 4p$, and $p_z = 2p$. On Jaguar, we consider only one problem, denoted "MG-J" with $p_x = p_y = 4p$ and $p_z = 3p$. For all cases $r_x = r_y = r_z = 12$, leading to $R_x = r_y = 24p$ and $R_z = 48p$ for MG-1, $R_x = R_y = 48p$ and $R_z = 24p$ for MG-2 and $R_x = R_y = 48p$ and $R_z = 36p$ for MG-J. Using this information, we can compute the following ratios

$$\rho_{MG-1}(s) = \rho_{MG-2}(s) = \rho_{MG-J}(s) = \frac{77}{\frac{24}{s_z} + \frac{28}{s_y} + \frac{25}{s_x}}. \quad (4)$$

For 16 threads defined as in the Laplace case, $\rho_{MG} = 2.37$. So, we would expect threading to be not quite as effective for the MG problem than for the Laplace problem. For 12 threads, if defined as above, $\rho_{MG} = 2.23$, which is similar to the Laplace problem as defined for Jaguar.

IV. Multicore architectures

We study the performance of AMG on three widely different architectures: a traditional multi-core, multi-socket cluster connected by Infiniband (*Hera*), a Dual-Hex Core Cray XT-5 with a custom 3D torus/mesh network (*Jaguar*), and a Quad-core BlueGene/P architecture with a custom 3D torus network (*Intrepid*).

A. Quad-Core/Quad-Socket Opteron Cluster (Hera)

Our first test machine is a tradtional multi-core/multi-socket cluster solution at Lawrence Livermore National Laboratory named *Hera*. It consists of 864 diskless nodes interconnected by Quad Datarate (QDR) Infiniband (accessed through a PCI card). Each node consists of four sockets, each equipped with an AMD Quadcore (8356) 2.3 GHz processors. Each core has its own L1 and L2 cache, but the 2 MB L3 cache is shared by all four cores located on the same socket. Each processor provides its own memory controller and is attached to a fourth of the 32 GB memory per node. Accesses to memory locations served by the memory controller on the same processor are satisfied directly, while accesses through other memory controllers are forwarded through the Hypertransport links connecting the four processors.Therefore, depending on the location of the memory, this configuration results in non-uniform memory access (NUMA) times. depending on the location of the memory.

Each node runs CHAOS 4, a high-performance computing, yet full featured Linux variant based on Redhat Enterprise Linux. All codes are compiled using Intel's C and OpenMP/C compiler (Version 11.1) and use MVAPICH over IB as the MPI implementation.

B. Dual Hex-Core Cray XT-5 (Jaguar)

Our second test platform is the Cray XT-5 system *Jaguar* [1] installed at Oak Ridge National Laboratory. It consists of 18,688 nodes organized in 200 cabinets. Each node is equipped with two sockets holding an AMD Opteron Hex-core processor each, as well as 16 GB of main memory split between the two memory controllers of the two sockets, leading to a similar NUMA architecture as seen on *Hera*. The nodes of the XT-5 are connected with a custom network based on the SeaStar 2+ router. The network is constructed as 3D torus/mesh with wrap-around links (torus-like) in two dimensions and without such links (mesh-like) in the remaining dimension.

All applications on *Jaguar* use a restricted subset of Linux, called Compute Node Linux (CNL). While it provides a Linux like environment, it only offers a limited set of services. On the upside, it provides a lower noise ratio due to eliminated background processes. The scheduler on the Cray XT-5 aims at a compact allocation of the compute processes on the overall network, but if such a compact partition is not available, it will also combine distant nodes (w.r.t., to network hops) into one partition.

We used PGI's C and OpenMP/C compilers (v 9.04) and experimented with the two different OpenMP settings in the PGI compiler: `-mp=nonuma` and `-mp=numa` to disable and enable optimizations for NUMA architectures (which mainly consist of preemptive thread pinning as well as localized memory allocations). Further, we used Cray's native MPI implementation, which is optimized for the SeaStar network.

[1] Precisely, we are using Jaguar-PF, the newer XT-5 installation at ORNL.

C. Quad-Core Blue Gene/P Solution (Intrepid)

The final target machine is the tightly integrated Blue Gene/P system at Argonne National Laboratory named *Intrepid*. This system consists of 40 racks with 1024 compute nodes each and each node contains a quad-core 850 MHz PowerPC 450 Processor bringing the total number of cores to 163,840. In contrast to the other two systems, all four cores have a common and shared access to the complete main memory of 2 GB. This guarantees a uniform memory access (UMA) characteristics. All nodes are connected by a 3D torus network and application partitions are always guaranteed to map to an electrically isolated proper subset of the nodes organized as a complete torus with wrap-around in all three dimensions.

On the software side, BG/P systems use a custom compute node kernel that all applications have to link to. This micro kernel provides only the most basic support for the application runtime. In particular, it only supports at most one thread per core, does not implement preemption support, and does not enable the execution of concurrent tasks. The latter has the side effect that executions are virtually noise free. All major functionality, in particular network access and I/O, is function shipped, i.e., remotely executed on a set of dedicated I/O nodes associated with each partition. We compiled all codes using IBM's C and OpenMP/C compilers v9.0 and used IBM's MPICH2-based MPI implementation for communication.

V. Performance results

In this section we present the performance results for BoomerAMG on the three multicore architectures and discuss the notable differences in performance. On each machine, we investigate an MPI-only version of AMG, a version that uses OpenMP across all cores on a node and MPI for inter-node communication, as well as intermediate versions that use a mix of MPI and OpenMP on node. For each experiment, we utilize all available cores per node on the respective machine. We look at the AMG setup times and either the AMG solve time or AMG cycle time (the latter is used for the Laplace problem, where the number of iterations to convergence varies across experimental setups, from 17 for 128 cores to 37 for 128,000 cores). In addition, for *Hera* and *Jaguar*, which are NUMA systems, we include an optimized OpenMP version that we developed after careful analysis of the initial results. This optimized version, labeled 'MCSUP' in the figures, is described and discussed in detail in Section VI-B (and is, therefore, not discussed in this section).

A. Hera: AMD Opteron Quad-core

We investigate the performance of AMG-GMRES(10) on the Laplace and MG problems on the *Hera* cluster. For the Laplace problem, we obtained weak scaling results with $100 \times 100 \times 100$ grid points per node on up to 729 nodes (11664 cores). We scaled up the problem setting $p_x = p_y = 2p$ and $p_z = 4p$ leading to $16p^3$ cores with $p = 1, ..., 9$. The AMG-GMRES(10) setup and cycle times for the Laplace problem are given in Figures 3a and 3b, respectively. Recall that *Hera* has four quad-core processors per node, and we use all 16 cores on each node for all experiments. Therefore the "MPI" and "OMP" labels indicate the use of 16 MPI tasks or 16 OpenMP threads per node, respectively. The mixed on-node models are denoted with the label MIX'k' where k is the number of OpenMP threads per MPI process, i.e., "MIX8" translates to the use of 2 MPI tasks per node with 8 OpenMP threads each. For the MG-2 problem, we ran on up to 512 nodes (8192 cores) with 1,327,104 grid points per node (82,944 per core), with $r_x = r_y = r_z = 12$ and $p_x = p_y = 2p$ and $p_z = 4p$, for $p = 1, 2, 3$. AMG-GMRES(10) setup and solve times for the MG-2 problem are given in Figures 3c and 3d. For the MG-1 problem, we ran on up to 128 nodes (2048 cores), with $r_x = r_y = r_z = 12$ and $p_x = p_y = 4p$ and $p_z = 2p$, with $p = 1, 2$, and, because the results are similar to MG-2, we simply list them in Table I.

TABLE I: Timings (in seconds) for the MG-1 Problem on Hera.

	Procs	MPI	Mix8	Mix4	Mix2	OMP	MCSup
Setup	256	2.3	1.5	1.5	2.2	3.8	3.8
	2048	25.1	6.5	2.7	3.0	5.1	5.1
Solve	256	2.0	1.9	1.9	3.7	7.3	2.7
	2048	4.2	3.1	2.3	4.3	8.2	3.3

First we examine the setup phase for both problems (Figures 3a and 3c). The eye-catching trend in these figures is the extremely poor performance for the MPI-only programming model. The algorithms in the setup phase are complex and contain a large amount of non-collective communication. With 16 MPI tasks per node, the code creates a high aggregated messaging rate across each node leading to a high pressure on the Infiniband network card, which further increases with growing node count. The commodity Infiniband interconnect is simply not well-suited for this kind of traffic and hence turns into a bottleneck causing an overall slowdown.

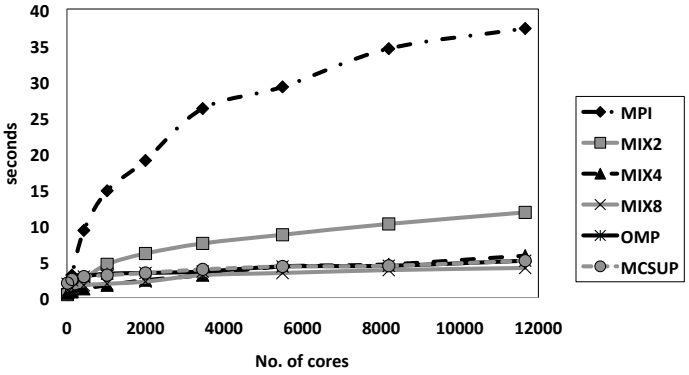
In the setup phase, MIX4, corresponding to a single MPI tasks per socket, initially obtains the best performance since it maps well to the machine architecture. However, for the largest node count it is surpassed by MIX8, which can again be attributed to the the network and the higher messaging rate needed to sustain the MIX4 configuration compared to MIX8.

The solve phase is less complex than the setup and mainly depends on the MatVec kernel. Its MPI performance (Figures 3b and 3d) is significantly better, and the OpenMP on-node version performs worst. This bad performance can be directly attributed to the underlying machine architecture and its NUMA properties, and we will discuss this as well as a solution to avoid this performance penalty in more detail in Secton VI-B.

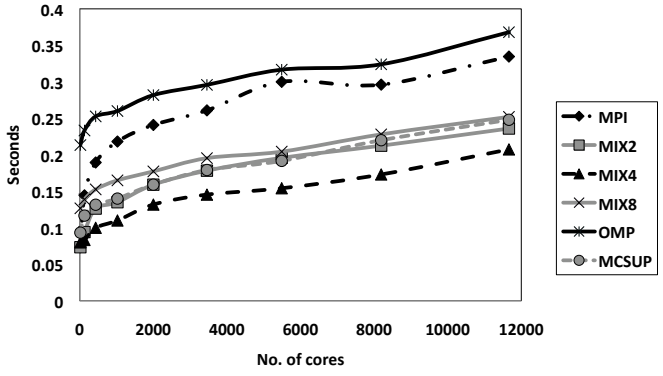
B. Jaguar: Cray XT5, AMD Opteron hex-core

On *Jaguar*, we had to slightly change the Laplace problem and the MG problem to enable the use of 6 and 12 threads. For the Laplace problem, we chose $N_x = N_y = 100p$, $N_z = 90p$, $p_x = p_y = 4p$, and $p_z = 3p$, leading to 18,750 grid points per core. We obtained results for this problem for $p = 2, 4, 8$ and show the resulting setup times in Figure 4a and the cycle times in Figure 4b. The number of iterations varied from 18 iterations for the smallest problem to 25 or 26 iterations for the largest problem. For the MG problem, we used $r_x = r_y = r_z = 12$, $p_x = p_y = 4p$, and $p_z = 3p$ with $p = 1, 2, 4$. Keep in mind that there are 8 parts, and therefore the total number of cores is $8 \times p_x \times p_y \times p_z$, leading to runs with the same number of cores as the Laplace problem, but larger numbers of grid points per core.

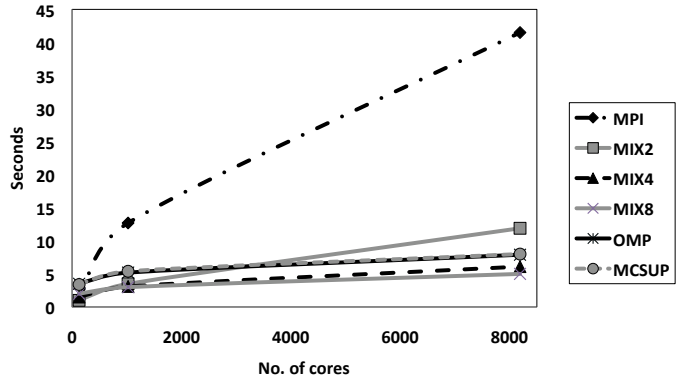
The results are comparable to those on *Hera*, which is not completely surprising, since both machines are NUMA architectures and are based on a similar multi-socket/multi-core node



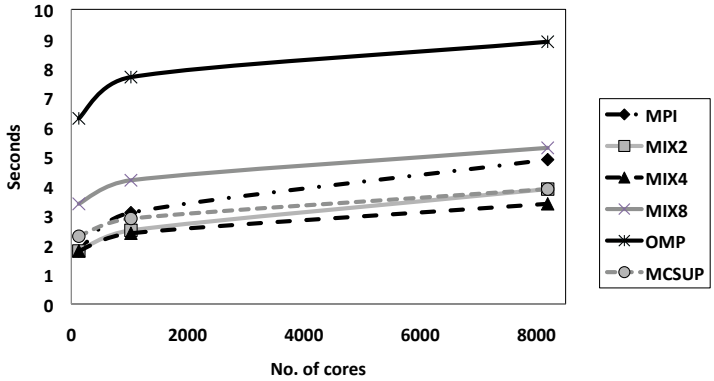
(a) Setup times for AMG-GMRES(10) on the Laplace problem.



(b) Cycle times for AMG-GMRES(10) on the Laplace problem.



(c) Setup times for AMG-GMRES(10) on the MG-2 problem.



(d) Solve times for AMG-GMRES(10) on the MG-2 problem.

Fig. 3: Timings on Hera

design. However, *Jaguar* features a custom interconnection network designed to withstand higher messaging rates than the Infiniband interconnect on *Hera*, allowing significantly better performance for one thread per process cases [2] For the solve phase, the MPI-only version shows better scalability, and behaves now similarly to numa and nonuma (OpenMP with the NUMA optimization enabled and disabled, respectively, c.f Section IV-B). The overall best performance is obtained through nonuma6 followed by numa6, which both use 6 threads per MPI task, i.e, use threading within one socket (analog to using MIX4 on *Hera*). Numa12 and nonuma12, which use 12 threads per node, show worse performance, which is again caused by the NUMA node architecture, similar to the OpenMP only case on *Hera*.

C. Intrepid: IBM BG/P Quad-core

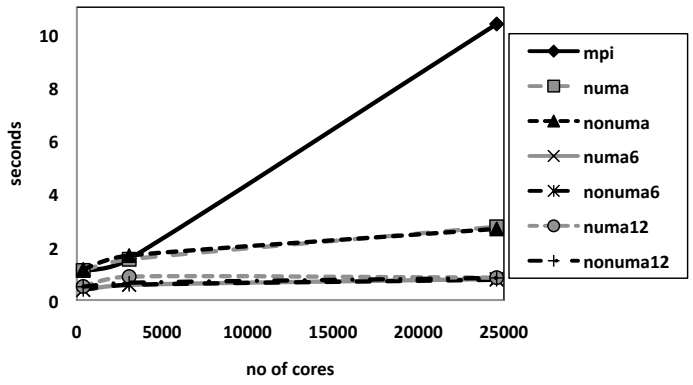
Blue Gene/P systems, like *Intrepid*, provide only a restricted operating system environment with a static task distribution. Codes can be run in one of three modes, Symmetric Multi-Processing (SMP), Dual (DUAL), and Virtual Node (VN), which determines the number of possible MPI tasks and threads. Our results are labeled accordingly. In SMP mode, we execute a single MPI process and use four threads per node; in DUAL mode we use two MPI processes with two threads each; and in VM mode, we use four MPI tasks per node with a single thread each. The latter configuration we run with two binaries, a plain MPI code without OpenMP (labeled "vn") and the OpenMP code executed with a single thread (labeled "vn-omp").

[2]We see a significantly worse performance for the MPI-only/no OpenMP case, which we believe to be an outlier caused by external conditions. We will verify this for the final paper.

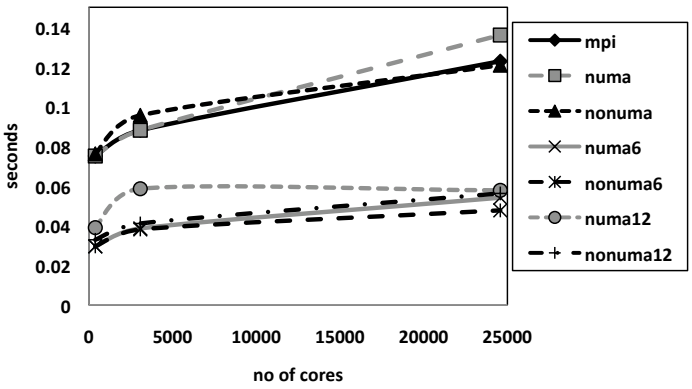
First we examine the weak scaling results for the Laplace problem on *Intrepid* with 250,000 grid points per node on up to 32,000 nodes (128,000 cores). The results for the AMG-GMRES(10) setup and cycle time (Figures 5a and 5b, respectively) show that the "vn" experiment, corresponding to the MPI-only model, is the generally the best-performing for both the setup and solve phase, although the solve cycle times are quite similar. This result is in stark contrast to the experiments on *Hera*, where the MPI-only model shows a significantly higher overhead. This is caused by the custom network on the BG/P system, which is designed to withstand higher messaging rates.

Overall, we see a good weak scaling behavior; only the times on 27,648 and 128,000 cores is slightly elevated due to the less optimal processor geometries compared to the 65,538 and 8192 core runs, which are both powers of two. For the setup phase, we see more variation in the results, and the "smp" case is the clearly worst performing, while the "dual" and "vn" experiments are quite similar. The problem partitioning likely does not play a role on this problem, with the "dual" case partition having the slight advantage of each MPI task subdomain being a perfect cube. The slower "smp" performance in the setup is likely due to the fact that the percentage of time spent in the nonthreaded portion of the setup phase increases with increasing number of threads as well as due to overhead associated with threading.

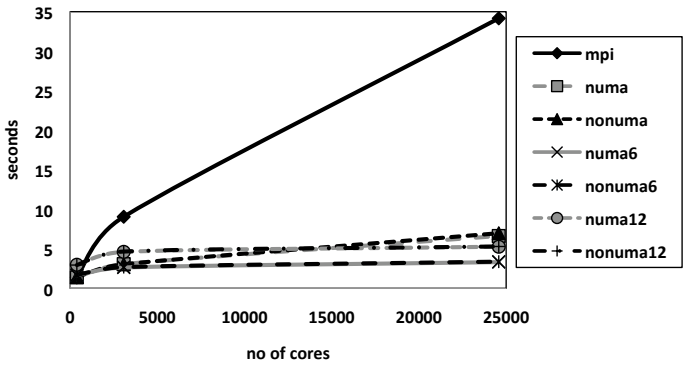
Since this effect becomes even more pronounced for the MG problem, we have listed timings for the MG-1 problem run on 32 nodes with at most 128 cores in Table II. Here we have also listed the times that one gets solving the same problem in MPI-only mode using 32, 64 and 128 MPI tasks. One can clearly see



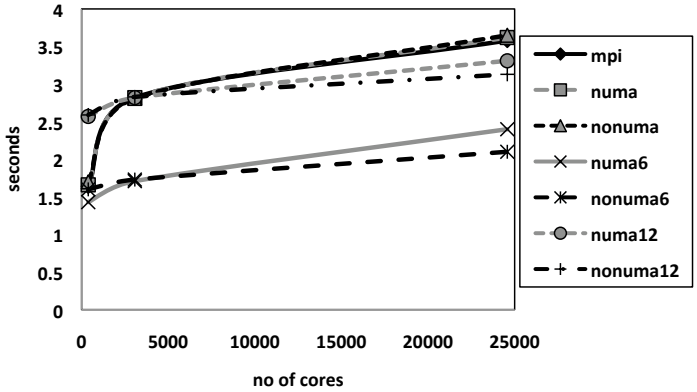
(a) Setup times for AMG-GMRES(10) on the Laplace problem.



(b) Cycle times for AMG-GMRES(10) on the Laplace problem.



(c) Setup times for AMG-GMRES(10) on the MG problem.



(d) Solve times for AMG-GMRES(10) on the MG problem.

Fig. 4: Timings on Jaguar

TABLE II: Timings (in seconds) for the MG-1 Problem with 10,616,832 grid points on Intrepid using at most 128 cores.

	Programming model	128 MPI tasks	64 MPI tasks	32 MPI tasks
Setup	MPI only	2.0	4.3	11.6
	MPI-OMP	2.5	4.0	7.1
Solve	MPI only	3.8	7.4	14.7
	MPI-OMP	3.8	4.1	4.3

TABLE III: Ghostlayer ratios for various numbers of threads

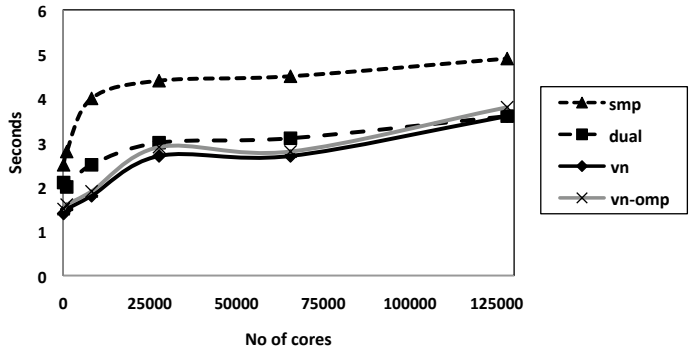
s	$\rho_L(s)$	$\rho_{MG-1}(s)$	$\rho_{MG-2}(s)$
2	1.33	1.18	1.18
4	1.60	1.31	1.51

the time improvement achieved by using two or 4 threads within each MPI task. In the setup phase the use of two threads leads to a speedup of 1.08, and 4 threads speed up the code by a factor of 1.72. However in the solve phase the speedup for two threads is 1.80, and for 4 threads 3.42. The use of OpenMP with one thread only shows a 25% overhead in the setup phase, whereas the solve phase performance is similar.

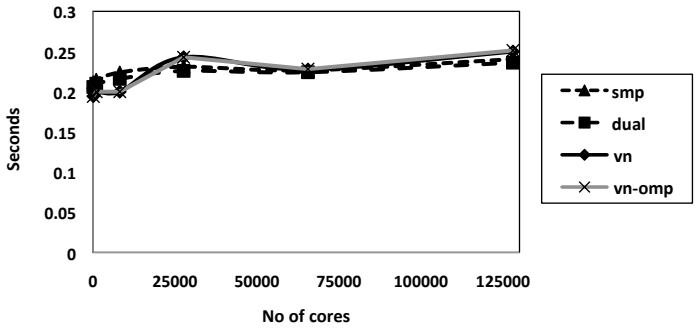
Now we take a look at the scaling results for the two variants of the MG problem. For the MG-1 problem, we ran on up to 16,384 nodes (65,536 cores) with 331,776 grid points per node, and results for the setup and solve phase are given in Figures 5c and 5d. For the MG-2 problem, we ran on up to 32,768 nodes (131,072 cores) with 331,776 grid points per node, and results for the setup and solve phase are given in Figures 5e and 5f. Like for the Laplace problem, we see that the MPI-only model ("vn") is the best-performing. Again, the differences in the solve phase are less pronounced than in the more-complex setup phase. Notice that for this more complicated grid, while the setup phase and solve phase take about the same amount of time in the "vn" case, the setup is considerably more expensive for the "smp" case. The "dual" mode is also more expensive than "vn" in the setup (unlike for the Laplace), indicating a limit to the parallelism that we can achieve with the threads on this more complex grid. Furthermore, on *Intrepid*, unlike *Hera*, we see that the performance of the "smp" case is not the same for the two different aspect ratios of the MG problem. Essentially, the "smp" performance is worse on MG-1, where the problem has been stretched in the z-direction. Now let us determine ρ_L, ρ_{MG-1} and ρ_{MG-2}. For two threads $s_x = s_y = 1$ and $s_z = 2$ for all three problems. For four threads $s_x = s_y 1$ and $s_z = 4$ for Laplace and MG-1, whereas $s_x = 1$ and $s_y = s_z = 2$ for MG-2. The actual values for ρ are listed in Table III. Overall the ratios for the Laplace problem are better than for MG-1 and MG-2, but while the ratios are equivalent for two threads, when using 4 threads the partitioning for MG-1 is worse than that for MG-2, explaining some of its worse performance.

VI. Lessons learned

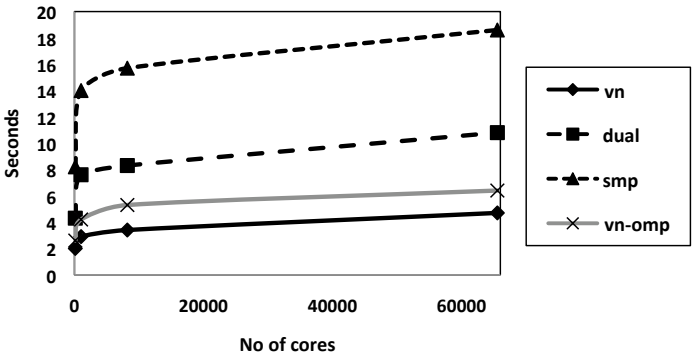
The measurements presented in the previous section reflect the performance of a production quality MPI code with carefully added OpenMP pragmas. It hence represents a typical scenario in which many application programmers find themselves when dealing with hybrid codes. The, in parts, poor performance shows



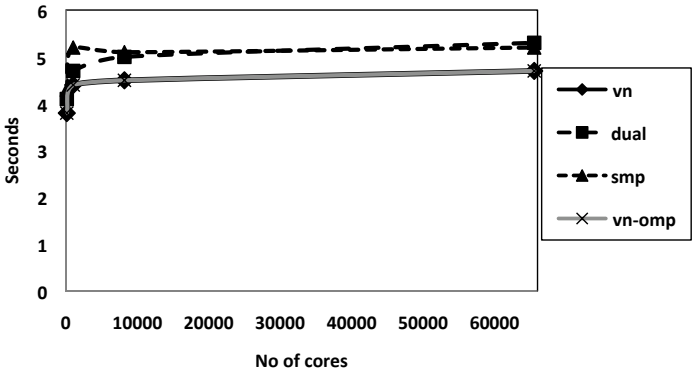
(a) Setup times for AMG-GMRES(10) on the Laplace problem.



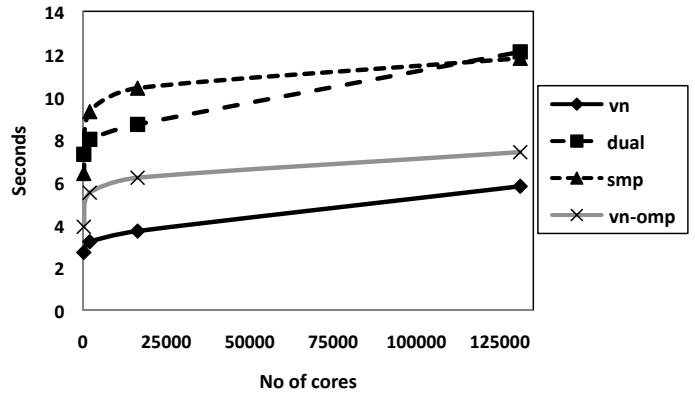
(b) Cycle times for AMG-GMRES(10) on the Laplace problem.



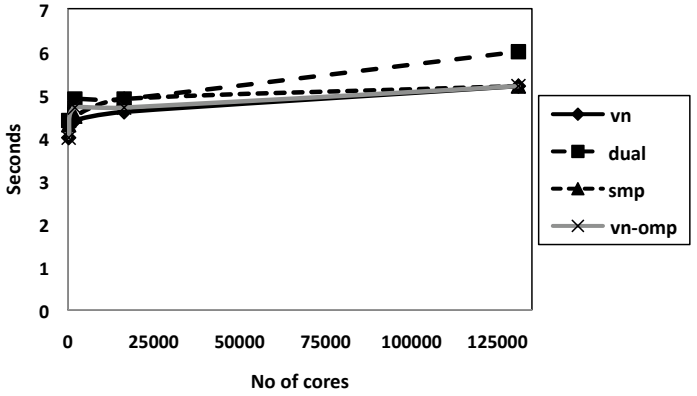
(c) Setup times for AMG-GMRES(10) on the MG-1 problem.



(d) Solve times for AMG-GMRES(10) on the MG-1 problem.



(e) Setup times for AMG-GMRES(10) on the MG-2 problem.



(f) Solve times for AMG-GMRES(10) on the MG-2 problem.

Fig. 5: Timings for problems on Intrepid.

how difficult it is to deal with hybrid codes on multicore platforms, but it also allows us to illustrate some lessons learned that help in optimizing the performance of AMG as well as other codes.

A. Network Performance vs. On-node Performance

AMG, as many of its related solvers in the *hypre* suite like SMG [16], are known to create a large amount of small messages. Consequently, it requires a network capable of sustaining a high messaging rate. This situation is worsened by running multiple MPI processes on a single multi-socket and/or multi-core node, since these processes have to share network access, often through a single network interface adapter.

We clearly see this effect on *Hera*, which provides both the largest core count per node and the weakest network: on this platform the MPI only version shows severe scaling limitations. Threading can help against this effect since it naturally combines the message sent from multiple processes into a single process. Consequently, the MIX4 and MIX8 version on this platform provides a better performance than the pure MPI version.

On the other two platforms, the smaller core count and the significantly stronger networks that are capable of supporting larger messaging rates, allow us to efficiently run the MPI only version at larger node counts. However, following the expectations for future architectures, this is a trend that we are likely unable to sustain: the number of cores will likely grow faster than the capabilities of the network interfaces and eventually a pure MPI model, in particular for message bound codes like MPI, will no longer be sustainable making threaded codes a necessity.

B. Correct Memory Associations and Locality

In our initial experiments on the multi-core/multi-socket cluster *Hera* we saw a large discrepancy in performance between running the same number of cores using only MPI and running with a hybrid OpenMP/MPI version with one process per node. This

behavior can be directly attributed to the NUMA nature of the memory system on each node: in the MPI only case, the OS automatically distributes the 16 MPI tasks per node to the 16 cores. Combined with Linux's default policy to satisfy all memory allocation requests in the memory that is local to the requesting core, all memory accesses are executed locally and without any NUMA latency penalties. In the OpenMP case, however, the master thread allocates the memory that is then shared among the threads. This leads to all memory being allocated on a single memory bank, the one associated with the master thread, and hence to remote memory access penalties as well as contention in the memory controller responsible for the allocations.

To fix this, we must either allocate all memory locally in the thread that is using it, which is infeasible since it requires a complete rewrite of most of the code to split any memory allocation to per thread allocations, or we must overwrite the default memory allocation policy and force a distribution of memory across all memory banks. The latter, however, requires an understanding of which memory is used by which thread and a custom memory allocator that distributes the memory based on this information.

For this purpose, we developed a multi-core support library, called *MCSup*, that provides a set of new NUMA-aware memory allocution routines, which allow programmers to explicitly specify which threads use which regions of the requested memory allocation. Each of these routines provides the programmer with a different pattern of how the memory will be used by threads in subsequent parallel regions. Most dominant in AMG is thereby a blocked distribution, in which each thread is associated with a contiguous chunk of memory of equal size. Consequently, MCSup creates a single contiguous memory region for each memory allocation request and then places the pages according to the described access pattern.

MCSup first detects the structure of the node it is executing on, i.e., determines the number of sockets and cores as well as their mapping to OpenMP threads, and then uses this information, combined with the programmer specified patterns, to correctly allocate the new memory. MCSup itself is implemented as a library that has to be linked the application source code. It uses Linux's `numalib` to get low level access to page and thread placement; the programmer only has to replace the memory allocations intended for cross thread usage with the appropriate MCSup routines.

The figures in Section V-A show the results obtained when using MCSup (labeled "MCSup"): with MCSup, the performance of OpenMP in the solve phase is substantially improved. We note that in the setup phase the addition of MCSup had little affect on the OpenMP performance because the setup phase algorithms are such that we are able to allocate the primary work arrays within the threads that use them (as opposed to allocation by the master thread as is required in the solve phase). Overall, though, we see that the performance with MCSup now rivals the performance of MPI only codes.

We see a similar trend on *Jaguar*: with its dual socket node architecture, it exhibits similar NUMA properties as *Hera*. However, due to limitations in the restricted Linux kernel (which we will discuss in more detail in Section VI-E) MCSup is not effective enough to enable an optimal execution of 12 threads per node. Running 6 threads per node, though, will match the number of MPI tasks with the number sockets and hence naturally keep memory allocations local.

C. Per Socket Thread and Process Pinning

The correct association of memory and threads ensures locality and the avoidance of NUMA effects, however, only as long as this association does not change throughout the runtime of the program. None of our two NUMA platforms actively migrates memory pages between sockets, but on the *Hera* system with its full featured Linux operating system, thread and processes can get scheduled across the entire multi-socket node, which can destroy the carefully constructed memory locality.

It is therefore necessary to pin threads and and processes to the appropriate cores and sockets, such that the memory/thread association determined by MCSup is maintained throughout the program execution. Note, that MCSup special memory allocations are only required if the threads of a single MPI process span multiple sockets; otherwise, pinning the threads of an MPI process to a socket is sufficient to guarantee locality, which we clearly see in the good performance of the MIX4 case on *Hera* and the 6 thread case on *Jaguar*.

D. Performance Impact of OpenMP Constructs

On *Intrepid* we see a large difference in performance of the setup phase between the MPI version ("vn") and the OpenMP version run with a single thread ("vn-omp"). Since the thread configuration is the same for those two versions, this overhead must stem from the OpenMP runtime system.

We traced the overhead back to a single routine in the setup phase — `hypre_BoomerAMGBuildCoarseOperator`. This routine contains four OpenMP `for` regions, each of them looping over the number of threads, with region three and four dominating the execution time. The timings for those regions along with the number of data loads and branch instructions are shown in Table IV.

The code of this function describes the (complex) setup of the matrix. It was manually threaded and uses a for loop to actually dispatch this explicit parallelism. However, we can replace the more common `for` loop with an explicit OpenMP parallel region, which implicitly executes the following block once on each available thread. While the semantics of these two constructs in this case is equivalent, the use of a parallel region slightly slows the execution, in particular in region 4.

Further, we find that both regions use a large number of private variables. While these are necessary for correct execution in the threaded case, we can omit them in the single thread case. This change has no impact on the code using an OpenMP for loop, but when removing private variables from the OpenMP parallel regions, the performance improves drastically from 47% to 16% overhead for region 3 and 31% to 10% for region 4.

We can see these performance trends also in the corresponding hardware performance counter data, also listed in Table IV, in particular the number of loads and the number of branches executed during the execution of these two regions. This can most likely be attributed to the outlining procedure and the associated changes in the code needed to privatize a large number of variables as well as additional book keeping requirements.

Based on these findings we are currently in the process of rewriting the code to reduce the number of private variables, such

Region 3	Time (s)	Rel. time	Loads	Branches
No OMP	20.09s	100%	100%	100%
OMP for loop	29.33s	146%	138%	150%
OMP for no priv.	29.35s	146%	137%	148%
OMP par. region	29.54s	147%	146%	152%
OMP reg. no priv.	23.32	116%	107%	103%

Region 4	Time (s)	Rel. time	Loads	Branches
No OMP	42.60s	100%	100%	100%
OMP for loop	52.29s	123%	119%	132%
OMP for no priv.	52.41s	123%	119%	133%
OMP par. region	55.80s	131%	121%	—
OMP reg. no priv.	46.92	110%	98%	98%

TABLE IV: Cumulative timings and hardware counter data for two OpenMP regions in `hypre_BoomerAMGBuildCoarseOperator` comparing code versions without OpenMP (baseline) to using an OpenMP for loop and an OpenMP parallel region each with and without private variables (128 MPI tasks, 1 thread per task, BG/P, problem MG-1)

that this performance improvement can be carried over in the threaded case. We will include those numbers in the final paper.

E. Understanding the Impact of Specialized Compute Kernels

Specialized compute kernels, such as Blue Gene's Compute Node Kernel (CNK) or Cray's Compute Node Linux (CNL), have been introduced to improve performance, since they reduce concurrent activities and thereby help to prevent noise, which is known to cause severe scalability problems.

However, since the kernels are restricted in their functionality, some services may longer be available or create a higher overhead than on full featured operating systems. We ran into this with MCSup on the *Jaguar* system: when executing an MCSup enabled code with a single thread per MPI task we saw enormous overheads of almost a factor 100. This overhead comes directly from MCSup's interaction with *Jaguar's* implementation of `libnuma`. It seems to create a massive contention in the OS, most likely during page table accesses and is unusable for this scenario.

On the other end of the spectrum, running with 12 threads per task performs worse then an execution with 6 threads and 2 MPI processes. In this case, the use of MCSup would be essential to guarantee a fair memory distribution. However, MCSup is not effective in this scenario either, since it fails to correctly allocate pages due to CNL's implementation of `libnuma` or since libnuma's overheads eliminates all savings. While we are still investigating the details of this performance anomaly, it is clear that neither variant leads to a production worthy environment and that, similar to the conclusions from above for *Hera*, it is best to run with one MPI process per socket.

VII. Conclusions

This paper presents a comprehensive study of a state-of-the-art Algebraic Multigrid (AMG) solver on three large scale multicore/multi-socket architectures. These kinds of systems already now cover a large part of the HPC space and will be dominating in the future. Good and (even more important) portable performance for key libraries, like AMG, on such systems will therefore be essential for their successful use. However, our study shows that we are still far from this goal, in particular with respect to performance portability.

The discussion in the previous section illustrates the many pitfalls waiting for developers of hybrid OpenMP/MPI codes. In order to achieve at least close to optimal performance, it is essential to guarantee memory locality, in particular in NUMA systems. Further, in order to maintain this locality, it is advised to turn off any kind of thread or process migration across sockets; threads of an MPI process should always be kept on the same socket to achieve both memory locality and minimize OS overhead. Further, as the example in Section VI-D shows, it is imperative to select the correct OpenMP primitive for a particular task, especially if multiple, equivalent pragmas are available and to reduce the number of private variables, since they infer additional bookkeeping. Finally, many systems provide only restricted operating systems, which may lead to severe performance anomalies when executed together with system-level libraries such as *MCSup*.

Overall, our results show that the performance and scalability of AMG on the three multicore architectures is quite varied and a general solution for obtaining good multicore performance is not possible without considering the specific target architecture, incl. node architecture, interconnect, and operating system capabilities. In many cases it is left to the programmer to find the right techniques to extract the optimal performance and the choice of techniques is not always straightforward. With the right settings, however, we can achieve a performance for hybrid OpenMP/MPI solutions that is at least equivalent to the existing MPI model, yet has the promise to scale to numbers of nodes that prohibit the use of MPI only applications.

VIII. Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344. It also used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357, as well as 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 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 the document.

References

[1] M. Heroux, "Scalable computing challenges: An overview," Talk at the 2009 SIAM Annual Meeting, Denver, CO, July 2009.

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

[3] R. D. Falgout, "An introduction to algebraic multigrid," *Computing in Science and Eng.*, vol. 8, no. 6, pp. 24–33, 2006.

[4] J. Ruge and K. Stüben, "Algebraic multigrid (AMG)," in *Multigrid Methods*, ser. Frontiers in Applied Mathematics, S. McCormick, Ed., vol. 3. SIAM, 1987.

[5] A. Brandt, S. McCormick, and J. Ruge, "Algebraic multigrid (AMG) for sparse matrix equations," in *Sparsity and its Applications*, D. J. Evans, Ed. Cambridge: Cambridge University Press, 1984, pp. 257–284.

[6] K. Stüben, "An introduction to algebraic multigrid," in *Multigrid*, U. Trottenberg, C. Oosterlee, and A. Schüller, Eds. Academic Press, London, 2001, pp. 413–532.
[7] J. Schmidt, G. Berti, J. Fingberg, J. Cao, and G. Wollny, "A finite element based tool chain for the planning and simulation of maxillo-facial surgery," in *ECCOMAS 2004*, P. Neittaanmaki, T. Rossi, K. Majava, and O. Pironneau, Eds., 2004.
[8] R. Masson, P. Quandalle, S. Requena, and R. Scheichl, "Parallel preconditioning for sedimentary basin simulations," in *LSSC 2003*, L. et al., Ed. Springer-Verlag, 2004, vol. 2907, pp. 93–102.
[9] E. Chow, R. Falgout, J. Hu, R. Tuminaro, and U. Yang, "A survey of parallelization techniques for multigrid solvers," in *Parallel Processing for Scientific Computing*, M. Heroux, P. Raghavan, and H. Simon, Eds. SIAM Series on Software, Environments, and Tools, 2006.
[10] U. Yang, "Parallel algebraic multigrid methods – high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. Bruaset and A. Tveito, Eds. Springer-Verlag, 2006, vol. 51, pp. 209–236.
[11] H. De Sterck, R. D. Falgout, J. Nolting, and U. M. Yang, "Distance-two interpolation for parallel algebraic multigrid," *Num. Lin. Alg. Appl.*, vol. 15, pp. 115–139, 2008.
[12] R. Falgout, J. Jones, and U. Yang, "The design and implementation of hypre, a library of parallel high performance preconditioners," in *Numerical Solution of Partial Differential Equations on Parallel Computers*, A. Bruaset and A. Tveito, Eds. Springer-Verlag, 2006, vol. 51, pp. 267–294.
[13] H. De Sterck, U. M. Yang, and J. Heys, "Reducing complexity in algebraic multigrid preconditioners," *SIMAX*, vol. 27, pp. 1019–1039, 2006.
[14] U. M. Yang, "On long distance interpolation operators for aggressive coarsening," *Num. Lin. Alg. Appl.*, vol. 17, pp. 353–472, 2010.
[15] R. Falgout, J. Jones, and U. M. Yang, "Pursuing scalability for hypre's conceptual interfaces," *ACM ToMS*, vol. 31, pp. 326–350, 2005.
[16] R. Falgout and U. Yang, "hypre: a Library of High Performance Preconditioners," in *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, Apr. 2002, pp. 632–641.