On the Performance of an Algebraic Multigrid Solver on Multicore Clusters

A. H. Baker, M. Schulz, and U. M. Yang
{*abaker,schulzm,umyang*}*@llnl.gov*

Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
PO Box 808, L-560, Livermore, CA 94551, USA

Abstract. Algebraic multigrid (AMG) solvers have proven to be extremely efficient on distributed-memory architectures. However, when executed on modern multicore cluster architectures, we face new challenges that can significantly harm AMG's performance. We discuss our experiences on such an architecture and present a set of techniques that help users to overcome the associated problems, including thread and process pinning and correct memory associations. We have implemented most of the techniques in a MultiCore SUPport library (MCSup), which helps to map OpenMP applications to multicore machines. We present results using both an MPI-only and a hybrid MPI/OpenMP model.

1 Motivation

Solving large sparse systems of linear equations is required by many scientific applications, and the AMG solver in *hypre* [14], called BoomerAMG [13], is an essential component of simulation codes at Livermore National Laboratory (LLNL) and elsewhere. The implementation of BoomerAMG focuses primarily on distributed memory issues, such as effective coarse grain parallelism and minimal inter-processor communication, and, as a result, BoomerAMG demonstrates good weak scalability on distributed memory machines, as demonstrated for weak scaling on BG/L using 125,000 processors [11].

Multicore clusters, however, present new challenges for libraries such as *hypre*, caused by the new node architectures: multiple processors each with multiple cores, sharing caches at different levels, multiple memory controllers with affinities to a subset of the cores, as well as non-uniform main memory access times. In order to overcome these new challenges, the OS and runtime system must map the application to the available cores in a way that reduces scheduling conflicts, avoids resource contention, and minimizes memory access times. Additionally, algorithms need to have good data locality at the micro and macro level, few synchronization conflicts, and increased fine-grain parallelism [4]. Unfortunately, sparse linear solvers for structured, semi-structured and unstructured grids do

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DE-AC52-07NA27344 (LLNL-CONF-429864).

not naturally exhibit these desired properties. Krylov solvers, such as GMRES and conjugate gradient (CG), comprise basic linear algebra kernels: sparse matrix-vector products, inner products, and basic vector operations. Multigrid methods additionally include more complicated kernels: smoothers, coarsening algorithms, and the generation of interpolation, restriction and coarse grid operators. Various recent efforts have addressed performance issues of some of these kernels for multicore architectures. While good results have been achieved for dense matrix kernels [1, 21, 5], obtaining good performance for sparse matrix kernels is a much bigger challenge [20, 19]. In addition, efforts have been made to develop cache-aware implementations of multigrid smoothers [9, 15], which, while not originally aimed at multicore computers, have inspired further research for such architectures [18, 12].

Little attention has been paid to effective core utilization and to the use of OpenMP in AMG in general, and in BoomerAMG in particular. However, with rising numbers of cores per node, the traditional MPI-only model is expected to be insufficient, both due to limited off-node bandwidth that cannot support ever-increasing numbers of endpoints, and due to the decreasing memory per core ratio, which limits the amount of work that can be accomplished in each coarse grain MPI task. Consequently, hybrid programming models, in which a subset of or all cores on a node will have to operate through a shared memory programming model (like OpenMP), will become commonplace.

In this paper we present a comprehensive performance study of AMG on a large multicore cluster at LLNL and present solutions to overcome the observed performance bottlenecks. In particular, we make the following contributions:

- A performance study of AMG on a large multicore cluster with 4-socket, 16-core nodes using MPI, OpenMP, and hybrid programming;
- Scheduling strategies for highly asynchronous codes on multicore platforms;
- A MultiCore SUPport (MCSup) library that provides efficient support for mapping an OpenMP program onto the underlying architecture;
- A demonstration that the performance of AMG on the coarsest grid levels can have a significant effect on scalability.

Our results show that both the MPI and the OpenMP version suffer from severe performance penalties when executed on our multicore target architecture without optimizations. To avoid the observed bottlenecks we must pin MPI tasks to processors and provide a correct association of memory to cores in OpenMP applications. Further, a hybrid approach shows promising results, since it is capable of exploiting the scaling sweet spots of both programming models.

2 The Algebraic Multigrid (AMG) Solver

Multigrid methods are popular for large-scale scientific computing because of their algorithmically scalability: they solve a sparse linear system with n unknowns with $O(n)$ computations. Multigrid methods obtain the $O(n)$ optimality by utilizing a sequence of smaller linear systems, which are less expensive to

compute on, and by capitalizing on the ability of inexpensive smoothers (e.g., Gauss-Seidel) to resolve high-frequency errors on each grid level. In particular, because multigrid is an iterative method, it begins with an estimate to the solution on the fine grid. Then at each level of the grid, a smoother is applied, and the improved guess is transferred to a smaller, or coarser, grid. On the coarser grid, the smoother is applied again, and the process continues. On the coarsest level, a small linear system is solved, and then the solution is transferred back up to the fine grid via interpolation operators. Good convergence relies on the smoothers and the coarse-grid correction process working together in a complimentary manner.

AMG is a particular multigrid method that does not require an explicit grid geometry. Instead, coarsening and interpolation processes are determined entirely based on matrix entries. This attribute makes the method flexible, as often actual grid information may not be available or may be highly unstructured. However, the flexibility comes at a cost: AMG is a rather complex algorithm.

We use subscripts to indicate the AMG level numbers for the matrices and superscripts for the vectors, where 1 denotes the finest level, so that $A_1 = A$ is the matrix of the original linear system to be solved, and m denotes the coarsest level. AMG requires the following components: grid operators $A_1, \ldots, A_m$, interpolation operators P_k, restriction operators R_k (here we use $R_k = (P_k)^T$), and smoothers S_k, where $k = 1, 2, \ldots m-1$. These components of AMG are determined in a first step, known as the *setup phase*. During the setup phase, on each level k, the variables to be kept for the next coarser level are determined using a coarsening algorithm, P_k and R_k are defined, and the coarse grid operator is computed: $A_{k+1} = R_k A_k P_k$.

Once the setup phase is completed, the *solve phase*, a recursively defined cycle, can be performed as follows, where $f^{(1)} = f$ is the right-hand side of the linear system to be solved and $u^{(1)}$ is an initial guess for u:

Algorithm: $MGV(A_k, R_k, P_k, S_k, u^{(k)}, f^{(k)})$.
If $k = m$, solve $A_m u^{(m)} = f^{(m)}$.
Otherwise:
Apply smoother S_k μ_1 times to $A_k u^{(k)} = f^{(k)}$.
Perform coarse grid correction:
Set $r^{(k)} = f^{(k)} - A_k u^{(k)}$.
Set $r^{(k+1)} = R_k r^{(k)}$.
Set $e^{(k+1)} = 0$.
Apply $MGV(A_{k+1}, R_{k+1}, P_{k+1}, S_{k+1}, e^{(k+1)}, r^{(k+1)})$.
Interpolate $e^{(k)} = P_k e^{(k+1)}$.
Correct the solution by $u^{(k)} \leftarrow u^{(k)} + e^{(k)}$.
Apply smoother S_k μ_2 times to $A_k u^{(k)} = f^{(k)}$.

The algorithm above describes a V(μ_1, μ_2)-cycle; other more complex cycles such as W-cycles are described in [3].

Determining appropriate coarse grids is non-trivial, particularly in parallel, where processor boundaries require careful treatment (see, e.g.,[6]). In addition,

interpolation operators often require a fair amount of communication to determine processor neighbors (and neighbors of neighbors) [7]. The setup phase time is non-trivial and may cost as much as multiple iterations in the solve phase. The solve phase performs the multilevel iterations (often referred to as cycles). These iterations consist primarily of applying the smoother, restricting the error to the coarse-grid, and interpolating the error to the fine grid. These operations are all matrix-vector multiplications (MatVecs) or MatVec-like, in the case of the smoother. An overview of AMG can be found in [11, 17, 3].

For the results in this paper, we used a modification of the BoomerAMG code in the *hypre* software library. We chose one of our best performing options: HMIS coarsening [8], one level of aggressive coarsening with multipass interpolation [17], and extended+i(4) interpolation [7] on the remaining levels. Since AMG is generally used as a preconditioner, we investigate it as a preconditioner for GMRES(10).

The results in this paper focus on the solve phase (since this can be completely threaded), though we will also present some total times (setup + solve times). Note that because AMG is a fairly complex algorithm, each individual component (e.g., coarsening, interpolation, and smoothing) affects the convergence rate. In particular, the parallel coarsening algorithms and the hybrid Gauss-Seidel parallel smoother, which uses sequential Gauss-Seidel within each task and delayed updates across cores, are dependent on the number of tasks, and the partitioning of the domain. Since the number of iterations can vary based on the experimental setup, we rely on average cycle times (instead of the total solve time) to ensure a fair comparison.

BoomerAMG uses a parallel matrix data structure. Matrices are distributed across cores in contiguous block of rows. On each core, the matrix block is split into two parts, each of which are stored in compressed sparse row (CSR) format. The first part contains the coefficients local to the core, whereas the second part contains the remaining coefficients. The data structure also contains a mapping that maps the local indices of the off-core part to global indices as well as information needed for communication. A complete description of the data structure can be found in [10].

Our test problem is a 3D Laplace problem with a seven-point stencil generated by finite differences, on the unit cube, with $100 \times 100 \times 100$ grid points per node. Note that the focus of this paper is a performance study of AMG on a multicore cluster, and not a convergence study, which would require a variety of more difficult test problems. This test problem, albeit simple from a mathematical point of view, is sufficient for its intended purpose. While the matrix on the finest level has only a seven-point stencil, stencil sizes as well as the overall density of the matrix increase on the coarser levels. We therefore encounter various scenarios that can reveal performance issues, which would also be present in more complex test problems.

3 The Hera Multicore Cluster

We conduct our experiments on Hera, a multicore cluster installed at LLNL with 864 nodes interconnected by Infiniband. Each node consists of four AMD Quad-core (8356) 2.3 GHz processors. Each core has its own L1 and L2 cache, but four cores share a 2 MB L3 cache. Each processor provides its own memory controller and is attached to a fourth of the 32 GB memory per node. Despite this separation, a core can access any memory location: 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. This leads to non-uniform memory access (NUMA) times depending on the location of the memory.

Each node runs CHAOS 4, a high-performance computing Linux variant based on Redhat Enterprise Linux. All codes are compiled using Intel's C and OpenMP/C compiler (Version 11.1). We rely on MVAPICH over IB as our MPI implementation and use SLURM [16] as the underlying resource manager. Further, we use SLURM in combination with an optional affinity plugin, which uses Linux's NUMA control capabilities to control the location of processes on sets of cores. The impact of these settings are discussed in Section 4.

4 Using an MPI-only Model with AMG

As mentioned in Section 1, the BoomerAMG solver is highly scalable on the Blue Gene class of machines using an MPI-only programming model. However, running the AMG solver on the Hera cluster using one MPI task for each of the 16 cores per node yields dramatically different results (Figure 1). Here the problem size is increased in proportion to the number of cores (using 50×50×25 grid points per core), and BG/L shows nearly perfect weak scalability with almost constant execution times for any number of nodes for both total times and cycle times. On Hera, despite having significantly faster cores, overall scalability is severely degraded, and execution times are drastically longer for large jobs.

To investigate this observation further we first study the impact of affinity settings on the AMG performance, which we influence using the before mentioned affinity plugin loaded as part of the SLURM resource manager. The black line in Figure 2 shows the performance of the AMG solve phase for a single cycle on 1, 64, and 216 nodes with varying numbers of MPI tasks per node without affinity optimizations (Aff=16/16 meaning that each of the 16 tasks has equal access to all 16 cores). The problem uses 100×100 ×100 grid points per node. Within a node we partition the domain into cuboids so that communication between cores is minimized, e.g., for 10 MPI tasks the subdomain per core consists of 100×50×20 grid points, whereas for 11 MPI tasks the subdomains are of size 100×100×10 or 100×100×9, leading to decreased performance for the larger prime numbers. From these graphs we can make two observations: the performance generally increases for up to six MPI tasks per node; adding more tasks is counterproductive. Second, this effect is growing with the number of

nodes. While for a single node, the performance only stagnates, the solve time increases for large node counts. These effects are caused by a combination of local memory pressure and increased pressure on the internode communication network.

Additionally, the performance of AMG is impacted by affinity settings: while the setting discussed so far (Aff=16/16) provides the OS with the largest flexibility for scheduling the tasks, it also means that a process can migrate between cores and with that also between processors. Since the node architecture based on the AMD Opteron chip uses separate memory controllers for each processor, this means that a process, after it has been migrated to a different processor, must satisfy all its memory requests by issuing remote memory accesses. The consequence is a drastic loss in performance. However, if the set of cores that an MPI task can be executed on is fixed to only those within a processor, then we leave the OS with the flexibility to schedule among multiple cores, yet eliminate cross-processor migrations. This choice results in significantly improved performance (gray, solid line marked Aff=4/16). Additional experiments have further shown that restricting the affinity further to a fixed core for each MPI task is ineffective and leads to poor performance similar to Aff=16/16.

It should be noted that SLURM is already capable of applying this optimization for selected numbers of tasks, as indicated by the black dashed line in Figure 2, but a solution across all configurations still requires manual intervention. Note that for the remaining results in this paper optimal affinity settings were applied (either manually using command line arguments for SLURM's affinity plugin or automatically by SLURM itself).

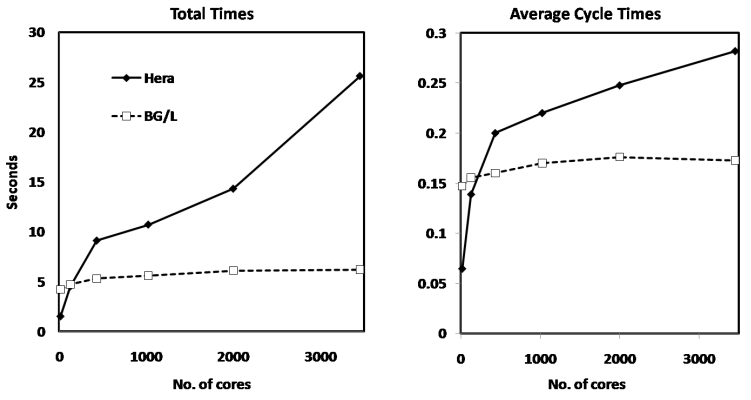

Fig. 1. Total times, including setup and solve times, (left) and average times per iteration (right) for AMG-GMRES(10) using MPI only on BG/L and Hera. Note that the setup phase scales much worse on Hera than the solve phase.

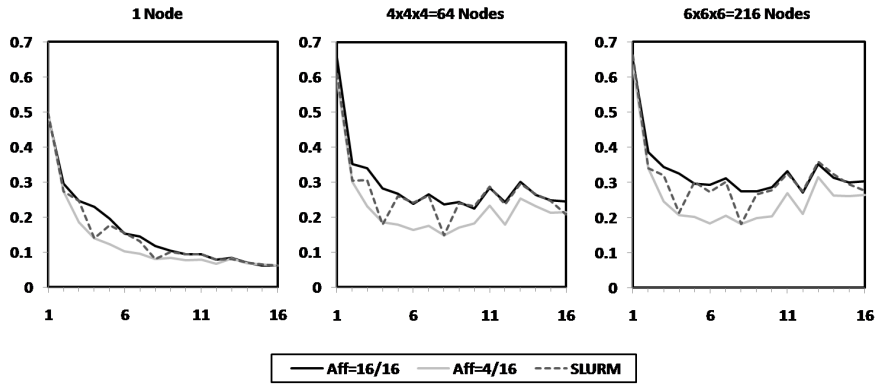

Fig. 2. Average times in seconds per AMG-GMRES(10) cycle for varying numbers of MPI tasks per node.

5 Replacing on-node MPI with OpenMP

The above observations clearly show that an MPI-only programming model is not sufficient for machines with wide multicore nodes, such as our experimental platform. Further, the observed trends indicate that this problem will likely get more severe with increasing numbers of cores. With machines on the horizon for the next few years that offer even more cores per node as well as more nodes, solving the observed problems is becoming critical. Therefore, we study the performance of BoomerAMG on the Hera cluster using OpenMP and MPI.

5.1 The OpenMP Implementation

Here we describe in more detail the OpenMP implementation within BoomerAMG. OpenMP is generally employed at the loop level. In particular for m OpenMP threads, each loop is divided into m parts of approximately equal size. For most of the basic matrix and vector operations, such as the MatVec or dot product, the OpenMP implementation is straight-forward. However, the use of OpenMP within the highly sequential Gauss-Seidel smoother requires an algorithm change. Here we use the same technique as in the MPI implementation, i.e., we use sequential Gauss-Seidel within each OpenMP thread and delayed updates for those points belonging to other OpenMP threads. In addition, because the parallel matrix data structure essentially consists of two matrices in CSR storage format, the OpenMP implementation of the multiplication of the transpose of the matrix with a vector is less efficient than the corresponding MPI implementation; it requires a temporary vector to store the partial matrix-vector product within each OpenMP thread and the subsequent summation of these vectors.

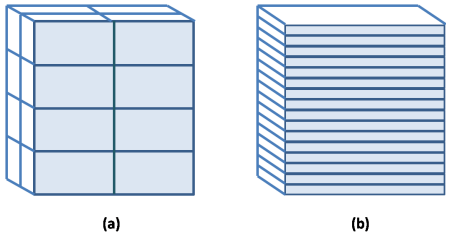

Fig. 3. Two partitionings of a cube into 16 subdomains on a single node of Hera. The partitioning on the left is optimal, and the partitioning on the right is the partitioning used for OpenMP.

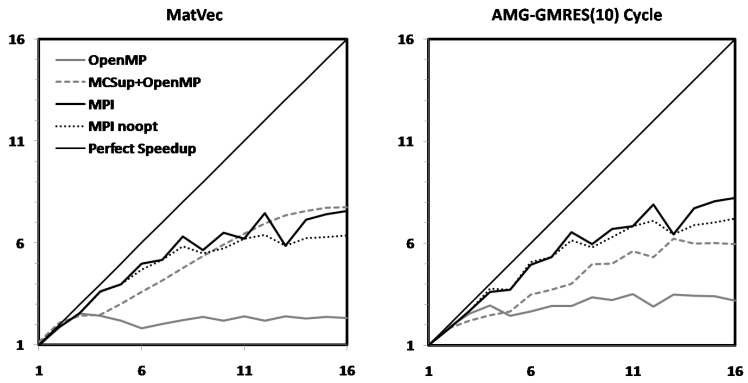

Fig. 4. Speedup for the MatVec kernel and a cycle of AMG-GMRES(10) on a single node of Hera.

Overall, the AMG solve phase, including GMRES, is completely threaded, whereas in the setup phase, only the generation of the coarse grid operator (a triple matrix product) has been threaded. Both coarsening and interpolation do not contain any OpenMP statements.

Note that, in general, the partitioning used for the MPI implementation is not identical to that of the OpenMP implementation. Whereas we attempt to optimize the MPI implementation to minimize communication (see Figure 3(a)), for OpenMP the domain of the MPI task is sliced into m parts due to the loop-level parallelism, leading to a less optimal partitioning (see Figure 3(b)). Therefore, Figure 4 (discussed in Section 5.2) also contains timings for MPI using the less-optimal partitioning (Figure 3(b)), denoted 'MPI noopt', which allows a comparison of MPI and OpenMP with the same partitioning.

5.2 Optimizing Memory Behavior with MCSup

The most time intensive kernels, the sparse MatVec and the smoother, account for 60% and 30%, respectively, of the solve time. Since these two kernels are similar in terms of implementation and performance behavior, we focus our investigation on the MatVec kernel. The behavior of the MatVec kernel closely matches the performance of the full AMG cycle on a single node. Figure 4 shows the initial performance of the OpenMP version compared to MPI in terms of speedup for the MatVec kernel and the AMG-GMRES(10) cycle on a single node of Hera (16 cores). The main reason for this poor performance lies in the code's memory behavior and its interaction with the underlying system architecture.

On NUMA systems, such as the one used here, Linux's default policy is to allocate new memory to the memory controller closest to the executing thread. In the case of the MPI application, each rank is a separate process and hence allocates its own memory to the same processor. In the OpenMP case, though, all memory gets allocated and initialized by the master thread and hence is pushed onto a single processor. Consequently, this setup leads to long memory access times, since most accesses will be remote, as well as memory contention on the memory controller responsible for all pages. Additionally, the fine-grain nature of threads make it more likely for the OS to migrate them, leading to unpredictable access times.

Note that in this situation even a first-touch policy, implemented by some NUMA-aware OS and OS extensions, would be insufficient. Under such a policy, a memory page would be allocated on a memory close to the core that first uses (typically writes) to it, rather than to the core that is used to allocate it. However, in our case, memory is often also initialized by the master thread, which still leads to the same locality problems. Further, AMG's underlying library *hypre* frequently allocates and deallocates memory to avoid memory leakage across library routine invocations. This causes the heap manager to reuse previously allocated memory for subsequent allocations. Since this memory has already been used/touched before, its location is now fixed and a first touch policy is no longer effective.

To overcome these issues, we developed MCSup (MultiCore SUPport), an OpenMP add-on library capable of automatically co-locating threads with the memory they are using. It performs this in three steps: first MCSup probes the memory and core structure of the node and determines the number of cores and memory controllers. Additionally, it determines the maximal concurrency used by the OpenMP environment and identifies all available threads. In the second step, it pins each thread to a processor to avoid later migrations of threads between processors, which would cause unpredictable remote memory accesses.

For the third and final step, it provides the user with new memory allocation routines that they can use to indicate which memory regions will be accessed globally and in what pattern. MCSup then ensures that the memory is distributed across the node in a way that memory is located locally to the threads most using it. This is implemented using Linux's NUMAlib, a set of low-level routines that provide fine-grain control over page and thread placements.

5.3 Optimized OpenMP Performance

Using the new memory and thread scheme implemented by MCSup greatly improves the performance of the OpenMP version of our code, as shown in Figure 4. The performance of the 16 OpenMP thread MatVec kernel improved by a factor of 3.5, resulting in comparable single node performance for OpenMP and MPI. Note that when using the same partitioning the OpenMP+MCSup version of the MatVec kernel shows superior performance than the MPI version for 8 or more threads. Also the performance of the AMG-GMRES(10) cycle improves significantly. However, in this case using MPI tasks instead of threads still results in better performance on a single node. The slower performance is primarily caused by the less efficient OpenMP version of the multiplication of the transpose of the matrix with a vector.

6 Mixed Programming Model

Due to the apparent shortcomings of both MPI- and OpenMP-only programming approaches, we next investigate the use of a hybrid approach allowing us to utilize the scaling sweet spots for both programming paradigms and present early results. Since we want to use all cores, we explore all combinations with m MPI processes and n OpenMP threads per process with $m * n = 16$ within a node. MPI is used across nodes. Figure 5 shows total times and average cycle times for various combinations of MPI with OpenMP. Note, that since the setup phase of AMG is only partially threaded, total times for combinations with large number of OpenMP threads such as OpenMP or MCSup are expected to be worse, but they outperform the MPI-only version for 125 and 216 nodes. While MCSup outperforms native OpenMP, its total times are generally worse than the hybrid tests. However when looking at the cycle times, its overall performance is comparable to using 8 MPI tasks with 2 OpenMP threads (Mix 8×2) or 2 MPI tasks with 8 OpenMP threads (Mix 2×8) on 27 or more nodes. Mix 2×8 does not use MCSup, since this mode is not yet supported, and therefore shows a similar, albeit much reduced, memory contention than OpenMP. In general, the best performance is obtained for Mix 4×4, which indicates that using a single MPI task per socket with 4 OpenMP threads is the best strategy.

7 Investigating the MPI-only Performance Degradation

Conventional wisdom for multigrid is that the largest amount of work and, consequently, the most time is spent on the finest level. This also coincides with our previous experience on closely coupled large-scale machines such as Blue Gene/L, and hence we expected that the performance and scalability of a version of the AMG preconditioner restricted to just two levels is similar to that of the multilevel version. However, our experiments on the Hera cluster show a different result.

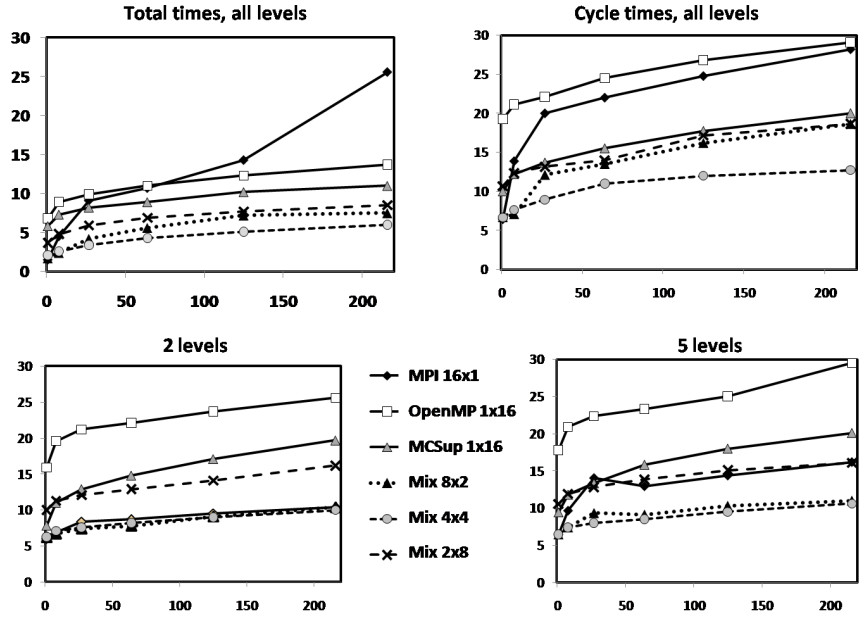

Fig. 5. Total times (setup + solve phase) in seconds of AMG-GMRES(10) (top left) and times in seconds for 100 AMG-GMRES(10) cycles (top right) using all levels (7 to 9) of AMG. Times for 100 cycles using two (bottom left) or five (bottom right) levels only. '$m \times n$' denotes m MPI tasks and n OpenMP threads per node.

The left plot on the bottom of Figure 5 illustrates that on two levels the MPI-only version performs as well as Mix 8×2 and Mix 4×4, which indicates that the performance degradation within AMG for the MPI-only model occurs on one or more of the lower levels. The right plots in Figure 5 confirm that, while the MPI-only version shows good scalable performance on two levels, its overall time is increasing much more rapidly than the other versions with increasing numbers of levels. While both OpenMP and MCSup do not appear to be significantly affected by varying the number of levels, performance for the variants that use more than one MPI task per node decreases (the Mix 4×4 case is least affected). We note that while we have only shown the degradation in MPI-only performance with increasing numbers of levels for the solve phase, the effect is even more pronounced in the setup phase.

To understand the performance degradation for the MPI-only version on coarser levels, we must first consider the difference in the work done at the finer and coarser levels. In general, on the fine grid the matrix stencils are smaller (our test problem is a seven-point stencil on the finest grid), and the matrices are sparser. Neighbor processors, with which communication is necessary, are

generally fewer and "closer" in terms of process ranks and messages passed between processors are larger in size. As the grid is coarsened, processors own fewer rows in the coarse grid matrices, eventually owning as little as a single row or even no rows at all on the coarsest grids.[1] On the coarsest levels there is very little computational work to be done, and the messages that are sent are generally small. However, because there are few processes left, the neighbor processes may be farther away in terms of process ranks. The mid-range levels are a mix of all effects and are difficult to categorize. All processors will remain at the mid-levels, but the stencil is likely bigger, which increases the number of neighbors. Figure 6 shows the total communication volume (setup and solve phase) collected with TAU/ParaProf [2] in terms of number of messages sent between pairs of processes on 128 cores (8 nodes) of Hera using the MPI-only version of AMG. From left to right in the figure, the number of AMG levels is restricted to 4, 6, and 8 (all) levels, respectively. Note that the data in these plots is cumulative, e.g., the middle 6-level plot contains the data from the left 4-level plot, plus the communication totals from levels 5 and 6. The fine grid size for this problem is 8,000,000 unknowns. The coarsest grid size with the 4, 6, and 8 levels is 13643, 212, and 3 unknowns, respectively.

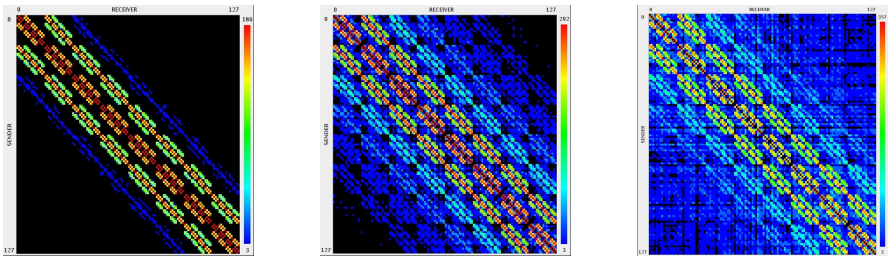

Fig. 6. Communication matrices indicating the total number of communications between pairs of 128 cores on 8 nodes. The x-axis indicates the id of the receiving MPI task, the the y-axis indicates the id of the sender. Areas of black indicate zero messages between cores. From left to right, results are shown for restricting AMG to 4, 6, and 8 (all) levels, respectively.

These figures show a clear difference in the communication structure in different refinement levels. For 4 levels we see a very regular neighborhood communication pattern with very little additional communication off the diagonal (black areas on the top/right and bottom/left). However, on the coarser levels, the communication added by the additional levels becomes more arbitrary and long-distance, and on the right-most plot with 8 levels of refinement, the communication has degraded to almost random communication. Since our resource manager SLURM generally assigns process ranks that are close together to be

[1] When all levels are generated, the AMG algorithm coarsens such that the coarsest matrix has fewer than nine rows.

physically closer on the machine (i.e., processes 0-15 are one a node, processes 16-31 are on the next node, etc.), we benefit from regular communication patterns like we see in the finer levels. The more random communication in coarser levels, however, will cause physically more distant communication as well as the use of significantly more connection pairs, which need to be initialized. The underlying Infiniband network used on Hera is not well suited for this kind of communication due to its fat tree topology and higher cost to establish connection pairs. The latter is of particular concern in this case since these connections are short lived and only used to exchange very little communication and hence the setup overhead can no longer be fully amortized.

When comparing the setup and the solve phase, we notice that the solve phase is less impacted by the performance degradation. While the setup phase touches each level only once, the solve phase visits each level at least twice (except the coarsest) in each iteration. This enables some reuse of communication pairs and helps to amortize the associated overhead.

We note that on more closely coupled machines, such as Blue Gene/L with a more even network topology and faster communication setup mechanisms, we don't see this degradation. Further the degradation is less in the hybrid OpenMP case, since fewer MPI tasks, and with that communication end points as well as communication pairs, are involved.

8 Summary

Although the *hypre* AMG solver scales well on distributed-memory architectures, obtaining comparable performance on multicore clusters is challenging. Here we described some of the issues we encountered in adapting our code for multicore architectures and make several suggestions for improving performance. In particular, we greatly improved OpenMP performance by pinning threads to specific cores and allocating memory that the thread will access on that same core. We also demonstrated that a mixed model of OpenMP threads and MPI tasks on each node results in superior performance. However, many open questions remain, particularly those specific to the AMG algorithm. In the future, we plan to more closely examine kernels specific to the setup phase and include OpenMP threads in those that have not been threaded yet. We will also explore the use of new data structures.

References

1. K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, and K. Yelick. The landscape of parallel computing research: A view from Berkeley, 2006.
2. R. Bell, A. Malony, and S. Shende. ParaProf: A Portable, Extensible, and Scalable Tool for Parallel Performance Profile Analysis. In *Proceedings of the International Conference on Parallel and Distributed Computing (Euro-Par 2003)*, pages 17–26, Aug. 2003.

3. W. Briggs, V. Henson, and S. McCormick. *A Multigrid Tutorial, 2nd ed.* SIAM, Philadelpia, PA, 2000.
4. A. Buttari, J. Dongarra, J. Kurzak, and P. Luszczek. The impact of multicore on math software, LNCS vol. 4699. In M. Heroux, P. Raghavan, and H. Simon, editors, *Applied Parallel Computing. State of the Art in Scientific Computing*, 2007.
5. A. Buttari, P. Lusczek, J. Kurzak, J. Dongarra, and G. Bosilca. A rough guide to scientific computing on the PlayStation 3, 2007.
6. E. Chow, R. Falgout, J. Hu, R. Tuminaro, and U. Yang. A survey of parallelization techniques for multigrid solvers. In M. Heroux, P. Raghavan, and H. Simon, editors, *Parallel Processing for Scientific Computing*. SIAM Series on Software, Environments, and Tools, 2006.
7. H. De Sterck, R. D. Falgout, J. Nolting, and U. M. Yang. Distance-two interpolation for parallel algebraic multigrid. *Num. Lin. Alg. Appl.*, 15:115–139, 2008.
8. H. De Sterck, U. M. Yang, and J. Heys. Reducing complexity in algebraic multigrid preconditioners. *SIMAX*, 27:1019–1039, 2006.
9. C. Douglas, J. Hu, M. Kowarschik, U. Ruede, and C. Weiss. Cache optimization for structures and unstructured grid multigrid. *Electronic Transactions on Numerical Analysis*, 10:21–40, 2000.
10. R. Falgout, J. Jones, and U. M. Yang. Pursuing scalability for hypre's conceptual interfaces. *ACM ToMS*, 31:326–350, 2005.
11. R. D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Eng.*, 8(6):24–33, 2006.
12. C. Garcia, M. Prieto, J. Setoain, and F. Tirado. Enhancing the performance of multigrid smoothers in simultaneaous multithreading architectures. In *VECPAR 2006, LNCS 4395*, pages 439–451. Springer, 2007.
13. V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.
14. *hypre*. High performance preconditioners. http://www.llnl.gov/CASC/linear_solvers/.
15. M. Kowarschik, I. Christadler, and U. Ruede. Towards cache-optimized multigrid using patch-adaptive relaxation. In Dongarra, Madsen, and Wasniewski, editors, *PARA 2004, LNCS 3732*, pages 901–910. Springer, 2006.
16. Lawrence Livermore National Laboratory. SLURM: Simple Linux Utility for Resource Management. http://www.llnl.gov/linux/slurm/, June 2005.
17. K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, London, 2001.
18. D. Wallin, H. Loef, E. Hagersten, and S. Holmgren. Multigrid and Gauss-Seidel smoothers revisited: Parallelization on chip multiprocessors. In *ICS 2006, Proceedings*, pages 145–155, 2006.
19. S. Williams, L. Oliker, R. Vuduc, J. Shalf, and K. Yelick. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35:178–194, 2009.
20. S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proceedings of IEEE/ACM Supercomputing '07*, 2007.
21. S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. *International Journal of Parallel Programming*, 2007.