LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-420726

On the Performance of an Algebraic Multigrid Solver on Multicore Clusters

A. Baker, M. Schulz, U. M. Yang

November 25, 2009

VECPAR'10
Berkeley, CA, United States
June 22, 2010 through June 25, 2010

Disclaimer

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* [5], called BoomerAMG [4], 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 [3].

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, we need algorithms with good data locality at the micro and macro level, few synchronization conflicts, and increased fine-grain parallelism. Additionally, 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, little attention has been paid to effective core utilization and to the use of OpenMP in AMG in general, and in BoomerAMG in particular.

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

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.

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. They utilize a sequence of smaller linear systems and capitalize on the ability of inexpensive smoothers (e.g., Gauss-Seidel) to resolve low-frequency errors on coarser grids. At each level of the grid, the improved guess is then transferred to a smaller, or 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.

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. AMG has two distinct phases, the setup and the solve phase. In the setup phase, the coarse grids, interpolation operators, and coarse-grid operators are determined. The solve phase performs the multilevel iterations (often referred to as cycles). For AMG, the setup phase time is non-trivial and may cost as much as multiple iterations in the solve phase. An overview of AMG can be found in [3, 6].

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, using

HMIS coarsening [2], one level of aggressive coarsening with multipass interpolation [6], and extended+i(4) interpolation [1] on the remaining levels. While the main focus of the BoomerAMG implementation has been on an efficient MPI implementation, parts of the code have also been threaded with OpenMP. As such, the solve phase is completely threaded, whereas in the setup phase, only the generation of the coarse grid operator (essentially a triple matrix product) has been threaded. Both coarsening and interpolation do not contain any OpenMP statements. Since AMG is generally used as a preconditioner, we investigate it as a preconditioner for GMRES(10), which is completely threaded.

The results in this paper focus on the solve phase, though we will also present some total times (setup + solve times). Note that AMG is a fairly complex algorithm, and 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 MPI or OpenMP task and delayed updates across cores, are dependent on the number of tasks, the combination of MPI/OpenMP, 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. 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.

3 Experimental Setup

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, any 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). Further, we rely on SLURM as the underlying resource manager and MVAPICH over IB as our MPI implementation.

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, 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 task 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 reduced by the affinity setting: 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 is based on separate memory controllers for each processor, this means that a process, after it has been migrated to a differ-

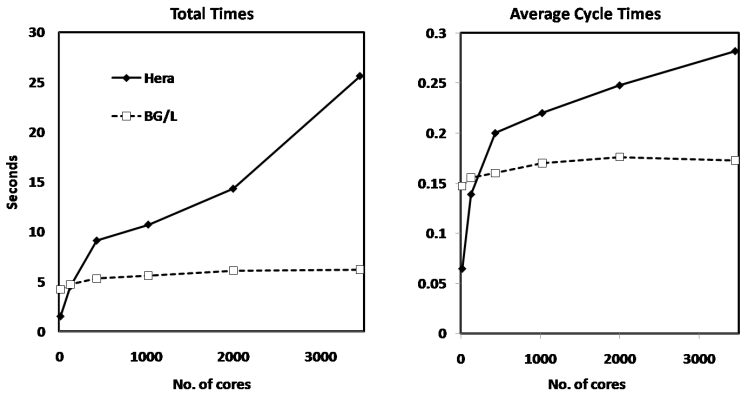

Fig. 1. Total times and average times per iteration for AMG-GMRES(10) using MPI only on BG/L and Hera.

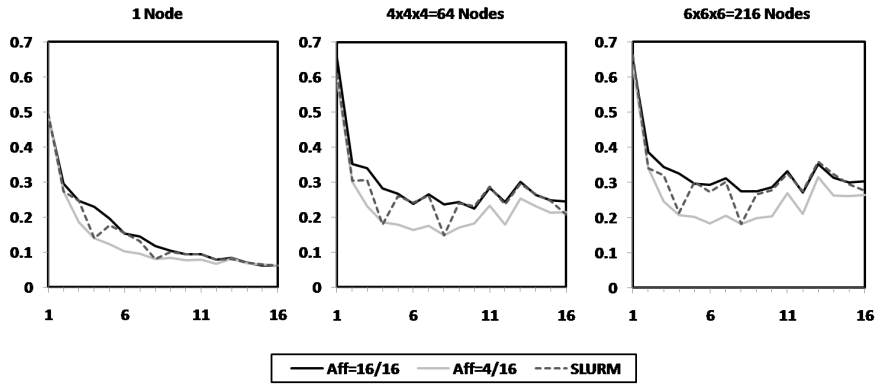

Fig. 2. Average times in seconds per AMG-GMRES(10) cycle for varying numbers of MPI tasks per node.

ent 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 proven 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, but a solution across all configurations still requires manual intervention.

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. The most time intensive kernels, the sparse matrix-vector product (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.

5.1 Optimizing Memory Behavior with MCSup

Figure 3 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 by the master thread and hence on a single processor. Consequently, this setup leads to large 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.

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 NUMAlib, a set of low-level routines that provide fine-grain control over page and thread placements.

5.2 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 3. 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.

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. Possible reasons for this could be other kernels, such as the multiplication of the transpose of the matrix with a vector, inefficiencies on lower levels of the grid refinement, or a larger fraction of code between OpenMP regions that are executed sequentially. In the final paper we will explore these issues further and discuss additional optimization strategies.

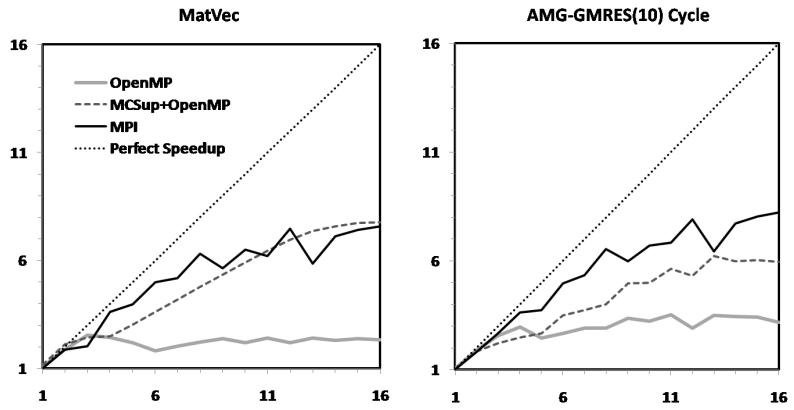

Fig. 3. Speedup for the MatVec kernel and a cycle of AMG-GMRES(10) on a single node of Hera.

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 4 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 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 suggestion for improving performance.

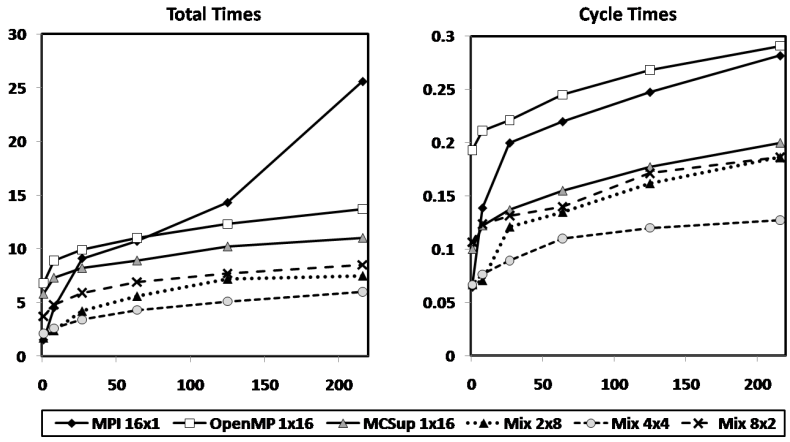

Fig. 4. Total times (setup + solve phase) in seconds of AMG-GMRES(10) (left) and average times in seconds for one AMG-GMRES(10) cycle (right). '$m \times n$' denotes m MPI tasks and n OpenMP threads per node.

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. 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 investigate performance degradation on the coarse levels and explore the use of new data structures.

References

1. 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.
2. H. De Sterck, U. M. Yang, and J. Heys. Reducing complexity in algebraic multigrid preconditioners. *SIMAX*, 27:1019–1039, 2006.
3. R. D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Eng.*, 8(6):24–33, 2006.
4. V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.
5. *hypre*. High performance preconditioners. http://www.llnl.gov/CASC/linear_solvers/.
6. K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, London, 2001.