LLNL-PROC-463941

Scaling Algebraic Multigrid Solvers: On the Road to Exascale

A. H. Baker, R. D. Falgout, T. Gamblin, T. Kolev, M. Schulz, U. M. Yang

December 13, 2010

Competence in High Performance Computing - CiHPC 2010
Schwetzingen, Germany
June 22, 2010 through June 24, 2010

Disclaimer

This document was prepared as an account of work sponsored by an agency of the United States government. Neither the United States government nor Lawrence Livermore National Security, LLC, nor any of their employees makes any warranty, expressed or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States government or Lawrence Livermore National Security, LLC. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States government or Lawrence Livermore National Security, LLC, and shall not be used for advertising or product endorsement purposes.

Scaling Algebraic Multigrid Solvers: On the Road to Exascale

Allison H. Baker, Robert D. Falgout, Todd Gamblin, Tzanio V. Kolev, Martin Schulz and Ulrike Meier Yang

Abstract Algebraic Multigrid (AMG) solvers are an essential component of many large-scale scientific simulation codes. Their continued numerical scalability and efficient implementation is critical for preparing these codes for exascale. Our experiences on modern multi-core machines show that significant challenges must be addressed for AMG to perform well on such machines. We discuss our experiences and describe the techniques we have used to overcome scalability challenges for AMG on hybrid architectures in preparation for exascale.

1 Introduction

Sparse iterative linear solvers are critical for large-scale scientific simulations, many of which spend the majority of their run time in solvers. Algebraic Multigrid (AMG) is a popular solver because of its linear run-time complexity and its proven scalability in distributed-memory environments. However, changing supercomputer architectures present challenges to AMG's continued scalability.

Multi-core processors are now standard on commodity clusters and high-end supercomputers alike, and core counts are increasing rapidly. However, distributed-memory message passing implementations, such as MPI, are not expected to work efficiently with more than hundreds of thousands of tasks. With exascale machines expected to have hundreds of millions or billions of tasks and hundreds of tasks per node, programming models will necessarily be hierarchical, with local shared-memory nodes in a larger distributed-memory message-passing environment.

With exascale in mind, we have begun to focus on a hybrid programming model for BoomerAMG [11], the AMG solver in the *hypre* [12] library. BoomerAMG has demonstrated good weak scalability in distributed-memory environments, such

Allison Baker, Rob Falgout, Todd Gamblin, Tzanio Kolev, Martin Schulz and Ulrike Yang
Lawrence Livermore National Laboratory, Center for Applied Scientific Computing e-mail: {abaker,rfalgout,tgamblin,tzanio,schulzm,umyang}@llnl.gov

as on 125,000 processors of BG/L [8], but our preliminary study [4] has shown that non-uniform memory access (NUMA) latency between sockets, deep cache hierarchies, multiple memory controllers, and reduced on-node bandwidth can be detrimental to AMG's performance.

To achieve high performance on exascale machines, we will need to ensure numerical scalability and an efficient implementation as core counts increase, memory capacity per core decreases, and on-node cache architectures become more complex. Some components of AMG that lead to very good convergence do not parallelize well or depend on the number of processors. We examine the effect of high level parallelism involving large numbers of cores on one of AMG's most important components, smoothers. We also investigate an OpenMP/MPI implementation of AMG, and its performance on three supercomputers with different node architectures: a cluster with four quad-core AMD Opteron processors, a Cray XT5 machine with two hex-core AMD Opteron processors, and a BlueGene/P system with a single quad-core PowerPC processor per node. The techniques used in these environments have broad applicability beyond AMG and will enable other solvers and simulation codes to prepare for exascale.

The remainder of this paper is organized as follows. In Section 2, we give an overview of the AMG method and detail our strategy to run efficiently at exascale. Section 3 describes mathematical and computational challenges of exascale for AMG smoothers. We describe the necessary steps to achieve good OpenMP performance on a multi-core node in Section 4. Finally, we demonstrate the performance of our hybrid BoomerAMG implementation on the three multi-core architectures in Section 5. Section 6 presents our conclusions.

2 The Algebraic Multigrid Solver

Multigrid (MG) linear solvers are particularly well-suited to parallel computing because their computational cost is linearly dependent on the problem size. This optimal property, also referred to as algorithmic scalability, means that proportionally increasing both the problem size and the number of processor (i.e., weak scaling), results in a roughly constant number of iterations to solution. Therefore, unsurprisingly, multigrid methods are currently quite popular for large-scale scientific computing and will play a critical role in enabling simulation codes to perform well at exascale.

2.1 Overview

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

MG has two phases: *setup* and *solve*. The primary computational kernels in the setup phase are the selection of the coarse grids, creation of the interpolation operators, and the representation of the fine grid matrix operator on each coarse grid. The primary computational kernels in the solve phase are a matrix-vector multiply (MatVec) and the smoothing operator, which may closely resemble a MatVec.

AMG is a flexible and unique type of MG method because it does not require geometric grid information. In AMG, coarse “grids” are simply subsets of the fine grid variables, and the coarsening and interpolation algorithms make use of the matrix entries to select variables and determine weights. These algorithms can be quite complex, particularly in parallel. More detailed information on AMG may be found in either [8] or [14].

2.2 Scaling Strategy for AMG

A well-designed AMG method is algorithmically scalable in that the number of iterations should stay fixed with increasing problem size. However, an effective AMG code must also be computationally scalable: the run times should stay constant with weak scaling. Therefore, both the algorithmic details related to the underlying mathematics (which impact the convergence rate) and the implementation details of the algorithm are important. To prepare our code for exascale computing, we have begun to examine the primary components of AMG, beginning with the solve phase, to determine what issues will need to be addressed.

The BoomerAMG code was originally developed with MPI in mind (as OpenMP was not competitive at the time). As a first step towards exascale, we have focused on incorporating OpenMP more fully into the code because we found performance on multi-core clusters using only MPI to be poor [4]. As discussed in Section 4, the initial OpenMP results were disappointing and required careful management of memory and threads to achieve good performance. Because of its simplicity and importance, the MatVec kernel was a natural focus for these initial efforts into investigating hybrid MPI + OpenMP performance. MatVec dominates the solve phase time (approximately 60%) as it is used for both restricting and interpolating the error, computing the residual, and, if AMG is used as a preconditioner, for the Conjugate Gradient (CG) or GMRES iteration step. From an implementation perspective, the matrix is stored in a compressed sparse row (CSR) parallel data structure [7]. MatVec is threaded at the loop level such that each thread operates on a subset of its process’ rows.

The other key component of the solve phase is the application of the smoother, which typically constitutes about 30% of the solve time. The smoother is critical because its effectiveness in reducing the error impacts the overall MG convergence rate, directly impacting the solution time. The challenge in designing a good parallel smoother is that Gauss-Seidel, the most popular MG smoother, is inherently sequential. Therefore in parallel AMG, a so-called hybrid Gauss-Seidel, which performs Gauss-Seidel within each task and delays updates across tasks, is typically employed. The convergence of hybrid Gauss-Seidel may be impacted by exascale computing due to the use of millions of concurrent tasks, as well as memory limitations requiring smaller problem sizes per processor. In addition, the use of OpenMP could affect convergence as loop-level threading in the smoother further partitions each process' domains. We discuss these concerns in detail in Section 3.

The AMG setup phase time is non-negligible and problem dependent; in some cases, depending on the number of iterations required for convergence, it can rival the time of the solve phase. Our preparation of the AMG setup phase for exascale computing is a work in progress. In particular, the coarsening and interpolation algorithms may be quite complicated in parallel [5], and the long-distance variety [6] require a sizable amount of point-to-point communications. At this point, the interpolation routines in BoomerAMG are only partially threaded due to complexity, and none of the coarsening routines use any threading at all. The third setup phase component, determining coarse grid operators via a triple matrix product, is completely threaded. However these coarse grid operators become far less-sparse than the fine grid matrix. The increasing operator density increases the number of communication pairs, which significantly impacts the MatVec time on the coarsest levels [4]. Future work for the setup phase will include the development of more thread-friendly data structures for the interpolation and coarsening routines and the investigation into ways to reduce communication on coarser grid levels, including the storage of redundant data when coarsening or the use of coarse grid operators that are not the result of a triple matrix product (i.e., non-Galerkin).

3 Smoothers

The smoothing process is at the heart of the AMG algorithm, and the quality of the smoother directly affects the design and the scalability of the multigrid solver. For a linear system with a symmetric and positive definite (SPD) matrix A, a smoother is another matrix M such that the iteration

$$e^0 = e\,, \qquad e^{n+1} = (I - M^{-1}A)e^n \quad \text{for } n = 1,2,\ldots$$

reduces the high-frequency components of an initial error vector e. This makes the remaining error smooth, so it can be handled by the coarse grid corrections. The smoother should also be convergent, so that the above iteration does not increase the low-frequency components of the error. When M is not symmetric, one can consider the symmetrized smoother $\widetilde{M} = M^T(M^T + M - A)^{-1}M$, which corresponds to a smoothing iteration with M, followed by a pass with M^T. The symmetrized smoother is often used when preconditioning CG with AMG, since CG requires a symmetric preconditioner.

One classical example of a convergent smoother is the Gauss-Seidel (GS) method, which is obtained by setting $M_{GS} = L + D$, where L is the lower triangular part and D is the diagonal part of A. Note that M_{GS} is not symmetric, so GS is frequently symmetrized in practice. Another class of general smoothers are the polynomial methods, where M is defined implicitly from $I - M^{-1}A = p(A)$ where p is a polynomial satisfying $p(0) = 1$. Both of these smoothing approaches have been essential in the development of serial algebraic multigrid and have excellent smoothing properties.

Parallel architectures, however, present serious challenges for these algorithms; GS is sequential in nature, while polynomial smoothers need knowledge of the spectrum of the matrix. Thus, a major concern for extending AMG for massively parallel machines has been the development of parallel smoothers that can maintain both scalability and good smoothing properties. In this section we summarize the theoretical and numerical results from [2] for several promising smoothers in the BoomerAMG code. Previous research in parallel smoothers can be found in [1, 15].

3.1 Hybrid Gauss-Seidel

The default smoother in BoomerAMG is a parallel version of GS known as hybrid Gauss-Seidel (hybrid-GS), which can be viewed as an inexact block-diagonal (Jacobi) smoother with GS sweeps inside each process. In other words, hybrid-GS corresponds to the block-diagonal matrix M_{HGS}, each block of which equals the process-owned $L + D$ part of A (BoomerAMG matrix storage is row-wise parallel).

Even though hybrid-GS has been successful in many applications, its scalability is not guaranteed, since it approaches Jacobi when the number of processors is large, or when the problem size per processor is small. Our strategy for addressing this issue is to investigate different variants of hybrid-GS through a qualitative smoother analysis based on the two-grid theory from [9, 10]. In particular, we proposed [2] the following criterion for smoother evaluation: if there is a constant C, independent of the problem size and the parallelization, such that $\langle \widetilde{M}x, x\rangle \leq C\langle \widetilde{M}_{GS}x, x\rangle$ for any vector x, then the smoother given by M will have multigrid smoothing properties comparable to (sequential) GS. This theoretical tool is important, because it allows us to predict the scalability of various approximations of M_{GS} on current and future parallel architectures.

As one application of the theory, we showed [2] that hybrid-GS will be a good smoother when the off-processor part of the matrix rows is smaller than the diagonal in each processor. This is the case, for example, when A is diagonally dominant and each diagonal processor block has at least two non-zero entries per row. However, there are practical cases where the off-processor part of the matrix is significant, due to the problem being solved (e.g., definite Maxwell discretizations) or due to the parallel partitioning of A (e.g., due to the use of threading). In these cases, hybrid-

GS will behave much worse than GS and can be divergent, even for large problem sizes per processor.

To improve the robustness of hybrid-GS, we proposed [13] and analyzed [2] the ℓ_1 Gauss-Seidel smoother (ℓ_1-GS), which corresponds to $M_{\ell_1 GS} = M_{HGS} + D^{\ell_1}$, where D^{ℓ_1} is a diagonal matrix containing the ℓ_1 norms of the off-processor part of each matrix row. This smoother has the nice property that it is convergent for any SPD matrix A. It was also shown [2] that $M_{\ell_1 GS}$ is comparable to M_{GS} with a constant $C = (1+4/\theta)^2$ where θ satisfies $D \geq \theta D^{\ell_1}$. Since θ depends only on the discretization stencil, for many practical problems it will be bounded away from zero, and thus $M_{\ell_1 GS}$ will result in a scalable AMG solver. To improve the performance, it is advantageous to consider parallel partitioning that lower the constant C, e.g., by using knowledge from the application in order to reduce the number and magnitude of the off-diagonal entries.

3.2 Chebyshev

As a second part of our strategy for scalable multigrid smoothers, we also explore polynomial methods where the high end of the spectrum of A is approximated with several CG iterations and a fixed scaling for the lower bound.

The theory from [9] can be applied also in this case to conclude that the best polynomial smoothers are given by shifted and scaled Chebyshev polynomials. To balance cost and performance, in practice we usually use the second order Chebyshev polynomial for $D^{-1/2}AD^{-1/2}$ (Cheby(2)). The cost of this method is comparable with the symmetrized hybrid-GS smoother.

Polynomial smoothers have the major advantage that their iterations are independent of the ordering of the unknowns or the parallel partitioning of the matrix. They also need only a matrix-vector multiply routine, which is typically finely tuned on parallel machines. These advantages, however, need to be balanced with the cost of estimating the high end of the spectrum of A. In our experience so far, this cost has not affected the scalability of Cheby(2).

3.3 Numerical Results

We illustrate the numerical performance of Cheby(2) and the symmetrized version of hybrid-GS and ℓ_1-GS with several results from [2]. The test problem describes a variable coefficient diffusion which is posed on the unit square and discretized with unstructured linear triangular finite elements, see Figure 1. We report the iteration counts for BoomerAMG used as a two-grid solver (AMG) or a preconditioner in CG (AMG-CG) with a relative convergence tolerance of 10^{-6}.

We first investigate the impact of threading through several weak scaling runs which alternate between the use of MPI and OpenMP on compute nodes with four

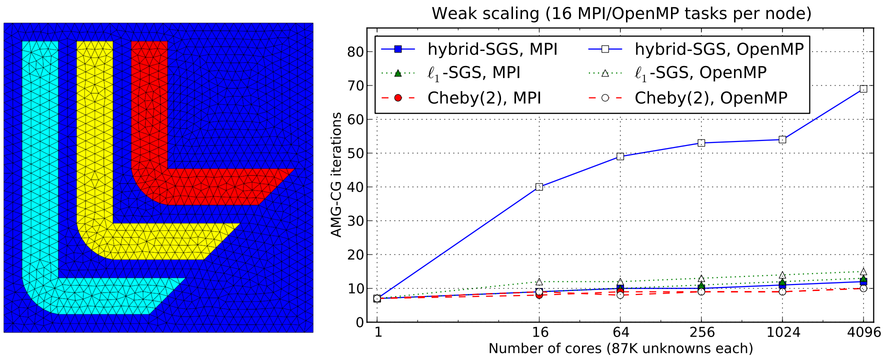

Fig. 1 Coarse mesh for the model problem with indicated material subdomains (left). The diffusion coefficient has 3 orders of magnitude jumps between the interior/exterior materials. Comparison of the scalability of AMG-CG with the different smoothing options when using threading (right).

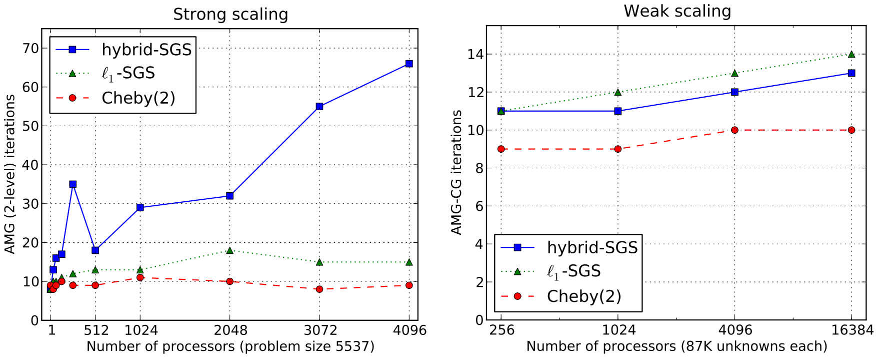

Fig. 2 Strong scaling of a two-level AMG solver with a very small problem sizes per processor (left), and weak scalability of AMG-CG with sufficiently large problem sizes per processor (right).

quad core processors each for a total of 16 cores per node. In this particular application, the numbering of the unknowns inside each processor is not guaranteed to have good locality, so the straightforward (not application-assisted) use of OpenMP introduces a bad partitioning onto the cores. As suggested by the theory, the performance of hybrid-SGS deteriorates significantly in this case, while ℓ_1-SGS and Cheby(2) remain robust. In contrast, in the MPI case the application provides parallel partitioning with a good constant θ, so all methods scale well. Note that the MPI weak scaling results on the right in Figure 2 indicate that with application-assisted parallel partitioning, all smoothers can lead to good weak scalability on very large number of processors.

Finally, we demonstrate the impact of small problem sizes per processor through the strong scaling runs presented on the left in Figure 2. This is an important test case, given the expected memory availability on future architectures. Since small problem sizes per processor are correlated with a large off-processor part of the matrix rows, the hybrid-SGS method deteriorates, as the smoothing analysis indicates is possible. In contrast, the effect on both ℓ_1-SGS and Cheby(2) is minimal, which is reassuring for their use on multi-core machines with small amounts of node memory.

4 On-node Threading and Memory Management

Modern High Performance Computing (HPC) systems feature increasingly complex node architectures with a rising number of compute cores per node, while the total amount of memory per node remains constant. Under such scenarios, flat programming models such as pure MPI will fail. We must provide programmers with multiple levels of concurrency, and the most common approach is a combination of MPI for cross-node communication with OpenMP for intra-node threading.

We study this setup on the Hera cluster at LLNL, a Linux cluster with 864 nodes connected by Infiniband. Each node has 16 cores distributed among four processors or sockets. Each socket features its own 8GB memory (2 GB per core), for a total node memory of 32GB. Any processor can access any memory location, but accesses to locations that belong to another processor's memory system incur an additional penalty. Systems such as Hera with Non-Uniform Memory Access latencies are called NUMA systems.

Figure 3 shows the speedup for two central kernels in AMG. The black line represents the MPI only version executed on a single node with varying numbers of cores after some minor scheduling and affinity optimizations (dotted line). Compared to this performance, the OpenMP-only version of the code (the gray solid line) performs significantly worse for all but the smallest number of cores.

On closer examination we found that the observed performance penalty is caused by the memory layout of the matrix structures in combination with the NUMA properties of the Hera nodes. Memory is allocated by the master thread and later used by all threads in the program. Since the underlying system aims to allocate all memory close to the core from which the memory is requested, all data was allocated on the memory of processor 0 leading to a large number of costly remote memory accesses from all threads running on cores in different processors as well as memory contention on processor 0.

To compensate for these problems, we developed the *MCSup* Multi-Core Support library. It provides the user with an API to allocate memory in a distributed manner across all processors in a way that matches the implicit thread distribution of OpenMP. Combined with a proactive per processor thread pinning, which is implicitly provided by *MCSup*, this technique can eliminate most remote memory accesses and reduce contention. This helps reduce the execution significantly and, as the gray dashed line in the figure shows, leads to execution times and overheads comparable to the pure MPI version.

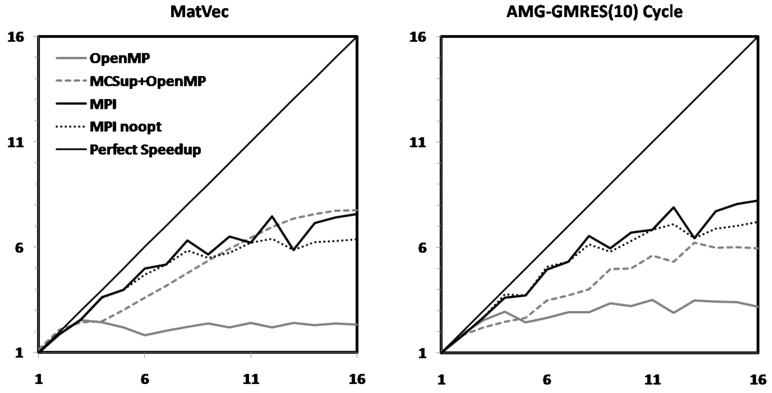

Fig. 3 Speedup for the MatVec kernel and a cycle of AMG-GMRES(10) on a single node of Hera.

5 Scaling of AMG

In this section we present scaling results for BoomerAMG on three different multicore architectures: the quad-core/quad-socket Opteron cluster Hera at Lawrence Livermore National Laboratory (up to 11,664 cores), the dual hex-core Cray XT-5 Jaguar at Oak Ridge National Laboratory (up to 196,608 cores), and the quad-core Blue Gene/P system at Argonne National Laboratory (up to 128,000 cores). On each machine, we investigate an MPI-only version of AMG, as well as hybrid 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 investigate the performance of AMG-GMRES(10) applied to a Laplace problem on a domain of size $N \times N \times \alpha N$, where $\alpha = 1$ on Hera and Intrepid, and $\alpha = 0.9$ on Jaguar. The domain is decomposed in such a way that each processor has $50 \times 50 \times 25$ unknowns on Hera and Intrepid and $50 \times 50 \times 30$ on Jaguar. We consider both MPI-only and hybrid MPI/OpenMP runs and use the notation described in Figure 4. In addition, for Hera, we include a version, labeled "H*m*x*n*MC", that uses the MCSup library described in Section 4.

We use hybrid-GS as a smoother. The number of iterations to convergence varies across experimental setups from 17 to 44. Note that, since both the coarsening algorithm and the smoother are dependent on the number of tasks and the domain partitioning among MPI tasks and OpenMP threads, the number of iterations can vary for different combination of MPI tasks and OpenMP threads, even when using the same problem size and number of cores. We present total times in Figure 4. Separate setup and cycle times for this problem on the three architectures are described in the original paper [3].

It is immediately apparent that on the two NUMA architectures, Hera and Jaguar, the MPI-only versions as well as H12x1 on Jaguar, perform significantly worse than

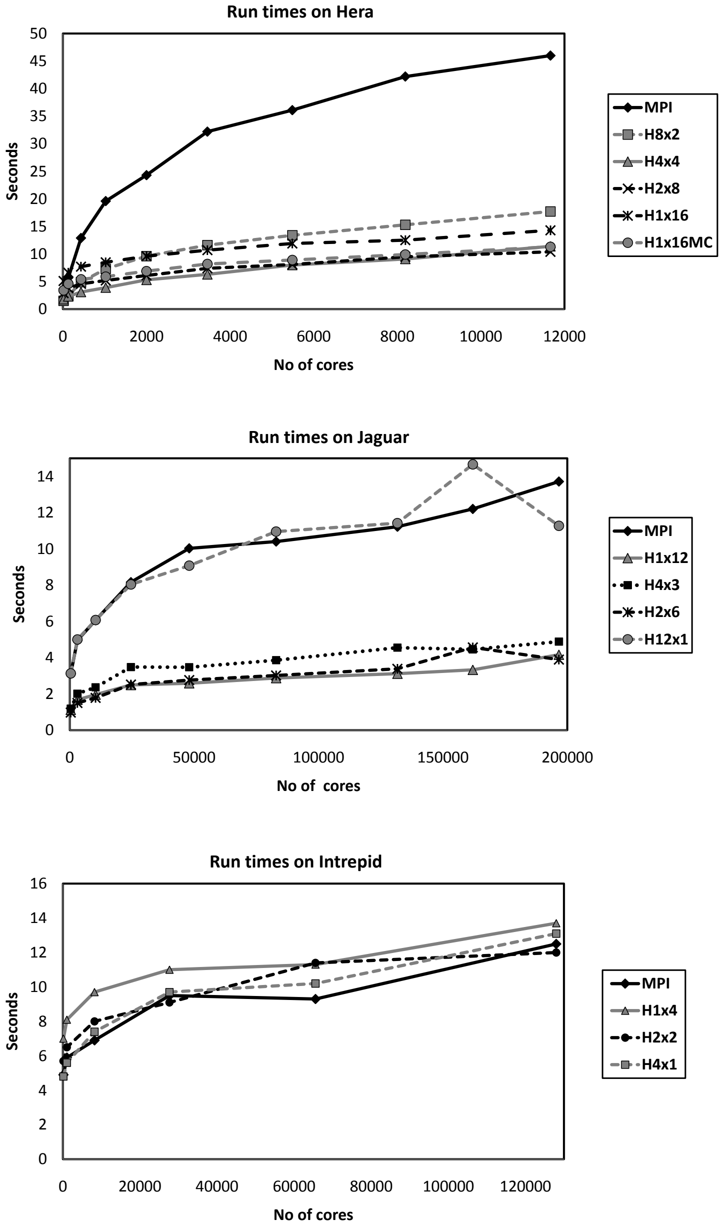

Fig. 4 Total times for AMG-GMRES(10) applied to a 7-point 3D Laplace problem on three different multi-core architectures; H*m*x*n* denotes runs performed with *m* MPI tasks per cluster and *n* OpenMP threads per MPI task, “MPI” denotes runs performed with the MPI-only version, H*m*x*n*MC denotes the use of MCSup

the other versions, whereas on Intrepid the MPI-only version generally performs best, with the exception of the less optimal processor geometries of 27,648 and 128,000 cores, where H2x2 is somewhat better. The worst performance on Intrepid is observed when using 4 threads per MPI task. Note that not all of the setup phase is threaded, leading to less parallelism when OpenMP is used, causing the lower performance of H1x4 on Intrepid, which has significantly slower cores than Jaguar or Hera. Interestingly enough this effect is not notable on Hera and Jaguar, which are however severely effected by the fact that the algorithms in the setup phase are complex and contain a large amount of non-collective communication leading to a large communication overhead and network contention. This effect is still visible, but less pronounced in the solve phase, which has a smaller amount of communication. On Hera, the worst performance for the solve phase is obtained for H1x16 (see [3, 4]), and is caused by the NUMA architecture. Using the *MCSup* library, see H1x16MC, performance is significantly improved. In the setup phase there is no NUMA effect for H1x16, since it mainly uses temporary storage, which is allocated within an OpenMP thread and therefore placed into the right memory module. For Hera and Jaguar initially the best times are obtained for the version that maps best to the architecture (H4x4 for Hera and H2x6 for Jaguar), to be then surpassed by H2x8 for Hera and H1x12 for Jaguar, versions with smaller network contention. Note that for the largest run on Jaguar, H1x12 takes more iterations than H2x6, causing H2x6 to be faster.

6 Conclusions

We investigated algebraic multigrid for exascale machines and considered both mathematical as well as computer science aspects to achieving scalability. Our investigation of smoothers showed that hybrid-GS promises to work well for certain problems even when we are dealing with millions or billions of cores. For more complicated problems ℓ_1-GS and polynomial smoothers are a viable fully parallel alternative to hybrid-GS because their convergence is not affected by the high level of parallelism needed for efficient implementations on exascale machines. Our tests showed that the performance of AMG varied across the three different multicore architectures considered. A general solution is not possible without taking into account the specific target architecture. With the right settings we can achieve a performance level using a hybrid OpenMP/MPI programming model that is at least equivalent to the existing MPI model, yet has the promise to scale to core counts that prohibit the use of MPI-only applications.

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. Adams, M. Brezina, J. Hu, and R. Tuminaro. Parallel multigrid smoothing: Polynomial versus Gauss-Seidel. *J. Comput. Phys.*, 188:593–610, 2003.
2. A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang. Multigrid smoothers for ultra-parallel computing. 2010. (submitted). Also available as a Lawrence Livermore National Laboratory technical report LLNL-JRNL-435315.
3. A. H. Baker, T. Gamblin, M. Schulz, and U. M. Yang. Challenges of scaling algebraic multigrid across modern multicore architectures. In *Proceedings of the 25th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2011)*, 2011. To appear. Also available as LLNL Tech. Report LLNL-CONF-458074.
4. A. H. Baker, M. Schulz, and U. M. Yang. On the performance of an algebraic multigrid solver on multicore clusters. In *Proceedings of the Ninth International Meeting on High Performance Computing for Computer Science (VECPAR 2010)*, 2010. Berkeley, CA, June 2010. http://vecpar.fe.up.pt/2010/papers/24.php.
5. 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.
6. 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.
7. R. Falgout, J. Jones, and U. M. Yang. Pursuing scalability for hypre's conceptual interfaces. *ACM ToMS*, 31:326–350, 2005.
8. R. D. Falgout. An introduction to algebraic multigrid. *Computing in Science and Eng.*, 8(6):24–33, 2006.
9. R. D. Falgout and P. S. Vassilevski. On generalizing the algebraic multigrid framework. *SIAM J. Numer. Anal.*, 42(4):1669–1693, 2004. UCRL-JC-150807.
10. R. D. Falgout, P. S. Vassilevski, and L. T. Zikatanov. On two-grid convergence estimates. *Numer. Linear Algebra Appl.*, 12(5–6):471–494, 2005. UCRL-JRNL-203843.
11. V. E. Henson and U. M. Yang. BoomerAMG: a parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.
12. *hypre*. High performance preconditioners. http://www.llnl.gov/CASC/linear_solvers/.
13. T. Kolev and P. Vassilevski. Parallel auxiliary space AMG for H(curl) problems. *J. Comput. Math.*, 27:604–623, 2009.
14. K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, London, 2001.
15. U. M. Yang. On the use of relaxation parameters in hybrid smoothers. *Numer. Linear Algebra Appl.*, 11:155–172, 2004. UCRL-JC-151575.