Title

Comparative Performance Analysis of Coarse Solvers for Algebraic Multigrid on Multicore and Manycore Architectures

Permalink

https://escholarship.org/uc/item/6d9275tz

ISBN

978-3-319-32148-6

Authors

Druinsky, Alex
Ghysels, Pieter
Li, Xiaoye S
et al.

Publication Date

2016

DOI

10.1007/978-3-319-32149-3_12

Peer reviewed

Comparative Performance Analysis of Coarse Solvers for Algebraic Multigrid on Multicore and Manycore Architectures*

Alex Druinsky[1], Pieter Ghysels[1], Xiaoye S. Li[1], Osni Marques[1], Samuel Williams[1], Andrew Barker[2], Delyan Kalchev[2], and Panayot Vassilevski[2]

[1] Lawrence Berkeley National Laboratory
[2] Lawrence Livermore National Laboratory

Abstract. We study the performance of a two-level algebraic-multigrid algorithm, with a focus on the impact of the coarse-grid solver on performance. We consider two algorithms for solving the coarse-space systems: the preconditioned conjugate gradient method and a new robust HSS-embedded low-rank sparse-factorization algorithm. Our test data comes from the SPE Comparative Solution Project for oil-reservoir simulations. We contrast the performance of our code on one 12-core socket of a Cray XC30 machine with performance on a 60-core Intel Xeon Phi coprocessor. To obtain top performance, we optimized the code to take full advantage of fine-grained parallelism and made it thread-friendly for high thread count. We also developed a bounds-and-bottlenecks performance model of the solver which we used to guide us through the optimization effort, and also carried out performance tuning in the solver's large parameter space. As a result, significant speedups were obtained on both machines.

Keywords: algebraic multigrid, HSS matrices, manycore machines

1 Introduction

We study the performance of a novel algebraic multigrid algorithm that was recently introduced by Brezina and Vassilevski [4] for solving difficult elliptic PDEs with a variable coefficient that can be resolved only using a fine-grained discretization. We use the two-level variant of the method, implemented in the serial C++ code `SAAMGe` [11, 12], and our focus is on the impact of the coarse-grid solver on the performance of the algorithm. The coarse grid represents the parallel bottleneck in the code, and therefore solving the corresponding systems efficiently is crucial for the solver's performance. Outside of the coarse grid,

* This material is based upon work supported by the US Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research (ASCR), Applied Mathematics program under contract number DE-AC02-05CH11231. This work was performed under the auspices of the DOE under Contract DE-AC52-07NA27344, and used resources of the National Energy Research Scientific Computing Center, which is supported by ASCR under contract DE-AC02-05CH11231.

most of the work in SAAMGe is formulated in terms of sparse matrix-vector multiplications (SpMVs) that involve large matrices with regular sparsity structure. Optimizing such SpMVs is a well-studied problem and optimized implementations are available for many architectures.

We consider two coarse-grid solvers. One is the preconditioned conjugate gradient method (PCG), which we precondition using a single step of the Jacobi iteration. Although careful optimization of PCG can make it a powerful coarse-grid solver, using it when convergence is slow can be expensive due to its low arithmetic intensity. As an alternative, we use STRUMPACK, a new HSS-embedded low-rank sparse-factorization algorithm [10]. It has a higher arithmetic intensity, and it is robust, meaning that it can be used as a direct solver.[3] The use of low-rank structure in the factorization reduces the amount of work, memory space, and memory bandwidth that we expend, making the solver potentially more efficient than earlier sparse-factorization algorithms.

There exist thorough studies of parallel-performance optimization and analysis of algebraic multigrid (AMG) [2,3,8,9]. Our paper differs from these studies by focusing on the architecture level and performing detailed performance-bound modeling, taking into account the arithmetic intensity of the different algorithmic components. We are the first to apply this methodology to AMG, and this allows us to bridge the gap in the understanding of the limits of AMG performance on individual multicore nodes with large core counts.

Our main contributions are the following. We make a comprehensive study of the impact of the coarse-grid solver on the performance of a two-level AMG solver. We perform extensive optimization on two multicore architectures, one of which is the challenging Xeon Phi, characterized by having a large number of relatively slow cores and a high memory latency. We incorporate a novel randomized HSS-embedded sparse-factorization algorithm as the coarse-grid solver. Finally, we develop and validate a bounds-and-bottlenecks Roofline model which helps us to identify performance optimization opportunities on our target architectures and to characterize the limitations of our code.

2 Background

2.1 Test Problems and Machines

We use the oil-reservoir simulation benchmark SPE10 (model 2) from the SPE Comparative Solution Project [6]. Here, fluid flow in porous media is described by the Darcy equation (in primal form):

$$-\nabla \cdot (\kappa(x) \nabla p(x)) = f(x) \tag{1}$$

[3] The STRUMPACK library can use the factorization either to solve the system directly, or to precondition the flexible GMRES iteration [16]. In our study, we found that performance is best if we tune STRUMPACK's parameters so that GMRES is not required. We expect this effect to be problem dependent. For details, see Sect. 4.

where $p(x)$ is the pressure field and $\kappa(x)$ is the permeability of the medium. The model is described on a regular Cartesian grid of $60 \times 220 \times 85$ cells. The coefficient $\kappa(x)$ admits a wide range of variation between distinct horizontal layers of the medium, which makes this a challenging problem to solve. Finite-element discretization of the problem produces a fine-grid matrix of order 1.2 million with a regular sparsity structure and an average of 26.4 nonzero elements in each row. From this matrix, the `SAAMGe` algorithm produces a coarse-grid matrix whose dimensions and sparsity pattern depend on the algorithm's parameters, as we explain below.

We carried out our study on two machines at the National Energy Research Scientific Computing Center in Oakland, CA. One machine is Edison, a Cray XC30 machine of 5,600 Ivy Bridge EP nodes. The other is Babbage, a commodity Intel cluster in which each node contains two Xeon Phi Knights Corner coprocessors. In the following, we refer to these machines as IVB-EP and KNC, respectively. In this paper, we focus on one CPU socket of an IVB-EP node and one KNC coprocessor. Each IVB-EP socket consists of 12 cores with a theoretical 230.4 peak gflop/s rate. A KNC coprocessor has 60 cores with a theoretical 1,010.9 peak gflop/s rate. Although KNC has more computational power and memory bandwidth, it has slower cores, a wider SIMD architecture and more primitive hardware stream prefetchers. As a result, SpMV-like operations are much more difficult to optimize on KNC. The cache hierarchies of both machines are coherent and have the same net capacity—30 MB. However, whereas IVB-EP provides a 30 MB unified L3 cache, KNC maintains 60 caches of 512 KB each. This results in superfluous data movement and coherency transactions that can impede the effective bandwidth on KNC. Furthermore, ineffective software prefetching can highlight the fact that KNC's memory latency can be an order of magnitude higher than that of IVB-EP [13]. As a result, although KNC has $5\times$ the nominal bandwidth of IVB-EP, it can often be underutilized or squandered given complex memory access patterns endemic in sparse methods.

2.2 Algebraic Multigrid

The SA-ρAMGe method that we study [4] works by forming a coarse-grid matrix A_c and solving the problem iteratively by repeating the following steps:

1. Pre-smoothing: $y_k \leftarrow x_k + M^{-1}(b - Ax_k)$
2. Coarse-grid correction: $z_k \leftarrow y_k + PA_c^{-1}P^T(b - Ay_k)$
3. Post-smoothing: $x_{k+1} \leftarrow z_k + M^{-1}(b - Az_k)$

Here, M^{-1} is a polynomial smoother and P is the so-called prolongation operator. The operator P is formed by representing A as the sum of local stiffness matrices (which requires knowledge of the finite-element discretization) and computing the eigenvectors that correspond to the smallest eigenvalues of each such matrix. We form the *tentative prolongator* $\bar{P}$, which is a rectangular block-diagonal matrix whose diagonal blocks correspond to the local stiffness matrices and consist of the eigenvectors that we computed. The ultimate prolongator has the form $P = S\bar{P}$, where S is a matrix-polynomial smoother. The coarse-grid matrix is obtained by forming the sparse-matrix product $A_c = P^T AP$.

2.3 HSS Sparse Solver

PCG is relatively easy to implement and parallelize, but its convergence can be slow for numerically difficult problems. Sparse-factorization methods can serve as powerful alternatives. Here, we consider an HSS-embedded sparse-factorization method that has asymptotically lower complexity than traditional factorizations. The algorithm has a shared-memory parallel implementation in the package `STRUMPACK` [10], which uses *nested-dissection* ordering and *multifrontal* factorization. The sparse solver consists of the following steps: preprocessing (e.g., sparsity-preserving ordering by a graph-partitioning algorithm such as METIS, and symbolic factorization), numerical factorization and solution.

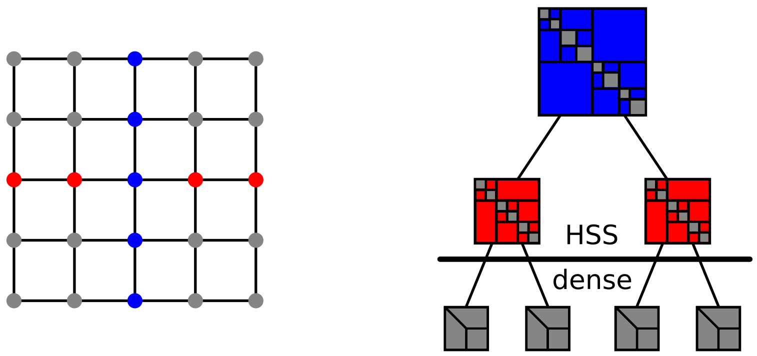


Fig. 1. A regular two-dimensional grid, partitioned using nested dissection (left) and the corresponding separator tree (right). The nodes of the separator tree represent frontal matrices. Only the nodes of the top levels are compressed using HSS.

The novelty is to represent the dense frontal matrix corresponding to a node of the separator tree as a hierarchically-semiseparable (HSS) matrix [10, 14, 18]. The HSS structure exploits the *data-sparsity* of the dense matrix using low-rank compression. Furthermore, the *hierarchical partitioning* of the matrix blocks and the use of *nested bases* lead to factorization and solve algorithms that are asymptotically faster than the classical ones and use less memory.

Figure 1 illustrates the nested-dissection procedure on a small regular mesh. The blue points denote the root separator, splitting the domain in two unconnected parts. The nested-dissection procedure is then repeated on both parts recursively. Each of the separators corresponds to one frontal matrix, placed in a data structure called the separator or elimination tree, illustrated in Figure 1 (right). The largest frontal matrices, corresponding to the top ℓ_s levels of the elimination tree, are approximated as HSS matrices. The other smaller frontal matrices are stored as full-rank dense matrices. The factorization is computed by a bottom-up traversal of the elimination tree, computing a partial factorization of each front and passing the Schur complement from child to parent. The structure of an HSS matrix is also illustrated in Figure 1 (right). In an HSS matrix, diagonal blocks are recursively partitioned until at the finest level diagonal blocks are stored as full-rank dense matrices. Off-diagonal blocks are represented as low-rank products $A_{ij} = U_i B_{ij} V_j^* + \mathcal{O}(\epsilon)$.

The `STRUMPACK` solver can be used as a preconditioner, where the quality of the preconditioner is controlled by the accuracy of the low-rank approximations in the HSS structure and by the number of HSS levels ℓ_s. The HSS approximation accuracy can be controlled by a user specified tolerance ϵ. A single solve with the `STRUMPACK` preconditioner is more expensive than a single PCG iteration, but it is more effective for numerically challenging PDEs.

3 Code Optimizations

Obtaining top performance on a traditional multicore architecture such as the IVB-EP is a well-studied problem, and therefore we focus in this section on the performance optimizations that we conducted on KNC. The challenges on this platform are due to a large core count, wide SIMD FPU and distributed coherent L2 caches.

3.1 PCG Thread-Friendly Optimizations

In contrast with IVB-EP, a large proportion of the time on KNC is spent on solving coarse-grid systems, and this proportion is increasing as we use more threads. Ultimately, using 180 threads, coarse-grid PCG accounts for more than 50% of the total time on KNC. Furthermore, our study in Sect. 4 showed that AMG performs best when the coarse-grid system is small, and so the optimizations that we require are different from those that are typically done in large-scale implementations.

The initial version of PCG in our code was implemented as a serial code that launched parallel OpenMP kernels such as dot product, vector linear combination and SpMV. Arranging the computation in this way incurs an overhead of entering and exiting an OpenMP `parallel` region for each computational kernel. For this reason, we introduced an alternative implementation, **omp-for-all**, in which the whole iteration is nested inside a single OpenMP `parallel` region. This yielded a speedup of 1.55 (using 180 threads).

We accomplished a further 1.69 speedup by introducing the **omp-for-spmv** variant, in which all kernels are serial, except for SpMV, which is parallel. Our explanation for the speedup in this case is that the coarse grid is represented by a small matrix and therefore the overhead of parallelizing the dot-product and vector-linear-combination kernels outweighs the benefits of such parallelization.

Finally, we also considered the **omp-parallel-spmv** variant, which we obtained from the original code by replacing all parallel kernels with serial ones, except for SpMV. Similarly to **omp-for-spmv**, in this version all kernels except SpMV are serial and there is only one `parallel` region. However, in contrast with **omp-for-spmv**, the parallel region here is inside the main loop and therefore incurs an overhead in each iteration. This version is slightly slower than **omp-for-spmv**, reaching 88% of **omp-for-spmv**'s performance using 180 threads. Nevertheless, **omp-parallel-spmv** is competitive with **omp-for-spmv** and it allows us to use an external library that implements the SpMV kernel, such as the one proposed in [13].

3.2 HSS Thread-Friendly Optimizations

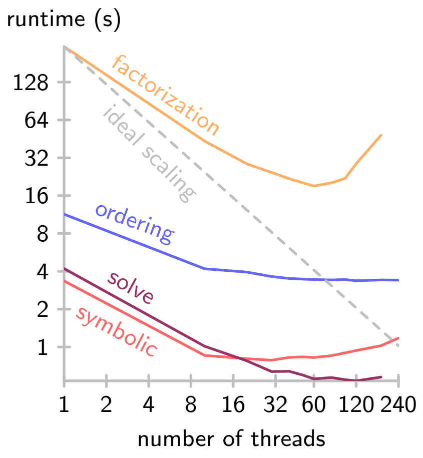
We now consider the use of `STRUMPACK` for solving coarse-grid systems. Table 1 and Fig. 2 show the running time of the algorithm.

Table 1. Runtime (seconds) of the HSS solver, broken down into the time dedicated to ordering, symbolic factorization, numeric factorization and solve. The coarse-grid matrix is of order 53,709 and has 24.8 million nonzeros, and was generated from a fine-grid matrix of order 2.4 million. The HSS compression level is 1 and tolerance is 10^{-4}, which corresponds to an HSS rank of 217.

Machine	Factorization (s)			Solve (s)	Threads
	ordering	symbolic	numeric		
IVB-EP	0.44	0.34	5.6	0.23	12
KNC	3.4	0.83	19.1	0.56	60

The solve time on both machines is more than an order of magnitude faster than factorization, which is for the better, because factorization is required only once, whereas solve is required in each AMG iteration. On both machines, the computation scales well, with the exception of ordering and symbolic factorization on KNC, where performance stagnates early, at about 10 threads. These steps involve purely combinatorial algorithms which are hard to parallelize on a machine architecture optimized for a high flop rate. Parallel scaling on IVB-EP is better than on KNC, with a speedup of 12 threads over one thread of 6.5× and 3.3× for factorization and solve, respectively, compared to speedups of 12.8× and 7.6× using 60 threads on KNC.

The solver is implemented using OpenMP task parallelism, so that the numeric-factorization phase is represented by a single parallel region, and therefore the only barriers in the code correspond to dependencies between the tasks. We took the following additional steps to improve performance on KNC. We replaced the default memory allocator by the more scalable TBB [1] allocator. Tasks are generated by recursive functions. We tuned the total number of tasks, i.e., the task granularity, specifically for each machine. We replaced the SCOTCH graph partitioner, which we were using for ordering the matrix, with METIS, and



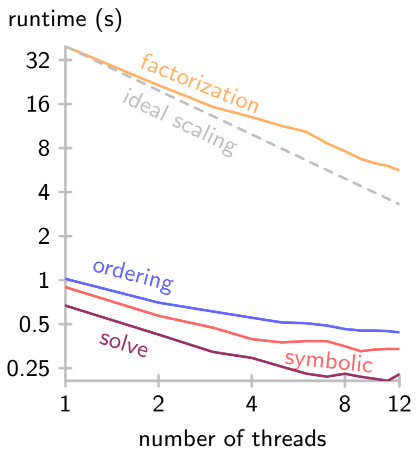


Fig. 2. Wall-clock time for solving the coarse-grid system on the Xeon Phi (left) and the Ivy Bridge EP (right). We used the optimized version of `STRUMPACK` as described in Sect. 3.2, and solved using three iterations of HSS-preconditioned GMRES.

thereby gained a 10× time savings in the ordering step on KNC.[4,5] We disabled a preprocessing step that performs permutation and scaling using the MC64 code [7]. This step is not required because our matrix is symmetric positive-definite. Finally, we disabled code for counting the number of executed flops, which we found was causing a 3× slowdown in the solve phase on KNC.

4 Performance Comparison

To complement our optimization effort, we conducted a comprehensive performance assessment of the solvers from the architectural viewpoint (two machines) as well as the algorithmic one (varying the algorithm's parameters). In particular, we considered five parameters of `SAAMGe`, and explored the effect of changing these parameters within a five-dimensional space. For each parameter we consider a range of values, as described in Table 2.[6] There are altogether 216 configurations explored on each machine. We used 60 cores on KNC and 12 cores on IVB-EP.

The following summarizes our findings. First, choosing the proper algorithm parameters is crucial to achieve good performance. The difference in runtime

[4] Based on this experience, we changed the `STRUMPACK` default to METIS.

[5] The ordering phase consists of running METIS, applying the computed permutation to the matrix and sorting the column indices within each row of the permuted matrix. METIS runs serially, but the rest of the work is done in parallel.

[6] We also used the following parameters to control the accuracy of the computed solution. For the HSS algorithm, we used four levels of compression with compression tolerance 10^{-4} and zero GMRES iterations. For PCG, we used relative tolerance 10^{-4}. These were chosen so as to maximize performance without sacrificing accuracy.

Table 2. The parameters of the algorithm and the corresponding values that we explored. The number of elements per agglomerate determines the size of the local stiffness matrices; ν_P and $\nu_{M^{-1}}$ are respectively the polynomial degrees of the interpolator smoother S and the relaxation smoother M^{-1}; and θ is the spectral tolerance, which determines how many eigenvectors of each local stiffness matrix represent that matrix in the tentative prolongator.

Parameter	Values
coarse solver	HSS, PCG
elements-per-agglomerate	64, 128, 256, 512
ν_P	0, 1, 2
$\nu_{M^{-1}}$	1, 3, 5
θ	0.001, $0.001 \times 10^{0.5}$, 0.01

can be as large as an order of magnitude among the 216 configurations. Taking PCG on IVB-EP as an example, the fastest configuration took 9.6 seconds, while the slowest took 168.1 seconds — more than a 17× difference. Second, on the same machine, the HSS coarse-grid solver always won over PCG. For the best configurations, HSS is 1.54× faster than PCG on IVB-EP and 1.34× on KNC. Finally, with the best configurations, IVB-EP is 1.7× and 1.49× faster than KNC using HSS and PCG, respectively.

5 Roofline Performance Model

We developed a bounds-and-bottlenecks Roofline model to drive the performance optimization of our OpenMP code [17].[7] The goal is to gain insight about the machine's performance bottlenecks and terminating performance optimization. Here, we focus on the AMG solution cycle; modeling AMG setup is future work.

The model consists of formulas, one for each component of the algorithm, expressing the number of bytes that we move between the levels of the memory hierarchy and the number of flops that we carry out. To obtain runtime estimates from this model, we divide the total memory traffic by the machine bandwidth, and also divide the total number of flops by the machine flops rate. This yields two lower bounds on the runtime: one that corresponds to memory bandwidth being the bottleneck, and the other to the floating-point units.

Concurrent with traditional Roofline analysis, the inputs to our model are: 1) The machine peak flop rate and its sustainable memory bandwidth, measured using a modified STREAM benchmark [15]; 2) The dimensions of A and A_c, denoted by n and n_c, respectively; 3) The number of nonzeros in A, A_c and P, denoted by `nza`, `nzc` and `nzp`, respectively; 4) The number of AMG cycles; and 5) Parameters that are specific to the coarse solver: the average number of PCG iterations per AMG cycle when we use PCG, and the memory size of the HSS factors when we use HSS.

[7] See also [5] for earlier work on such models.

5.1 The Model for the Combination of AMG with PCG

The model that we obtain for the version of the solver in which we use PCG to solve coarse-grid systems is shown in Table 3. We used the following combination of parameters: elements-per-agglomerate is set to 400, $\nu_{M^{-1}} = 3$ and $\theta = 0.001$. The corresponding runtime bounds on IVB-EP are shown in Fig. 3.[8]

Table 3. The costs associated with each AMG cycle.

Stage	Bytes	Flops
pre- and post-smooth	$(3\nu + 1)(12\,\mathtt{nza} + 3 \cdot 8n)$	$2(3\nu + 1)(\mathtt{nza} + 2n)$
restriction	$12\,\mathtt{nza} + 12\,\mathtt{nzp} + 3 \cdot 8n$	$2(\mathtt{nza} + \mathtt{nzp})$
one coarse solve (PCG/J)		
multiply by A_c	$12\,\mathtt{nzc}$	$2\,\mathtt{nzc}$
preconditioner	$2 \cdot 8n_c$	n_c
vector operations	$5 \cdot 8n_c$	$2 \cdot 5n_c$
interpolation	$12\,\mathtt{nzp} + 8n$	$2\,\mathtt{nzp}$
stopping criterion	$12\,\mathtt{nza} + 4 \cdot 8n$	$2(\mathtt{nza} + n)$

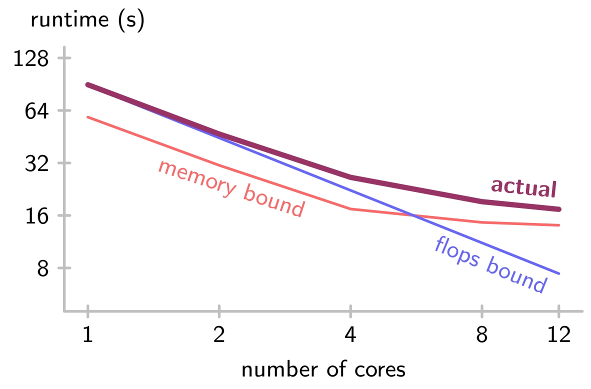


Fig. 3. Runtime bounds and actual runtime (seconds) of the AMG iteration on the Ivy Bridge EP.

[8] Roofline models often use a corrected machine gflop/s rate that accounts for an imbalanced mix of multiply and add operations in the computation. We do not do this here, because in our computation, multiplies and adds are almost perfectly balanced. The only exception is multiplications by a diagonal matrix in the polynomial smoother and the Jacobi preconditioner, but these multiplications correspond to a small fraction of the work.

When 1 or 2 cores are used, our flops-based bound is within 1% and 7% of actual runtime respectively. As the number of cores is increased, memory bandwidth becomes the bottleneck. For 4, 8 and 12 cores, our memory-bandwidth-based bound is within 23% of the actual runtime. We attribute the difference to the extremely different memory access patterns in AMG compared to the STREAM benchmark.

5.2 The Model for the Combination of AMG with HSS

Following the same practice as in Sect. 5.1, we conduct a performance-bound analysis when HSS is used as the coarse solver. Comparing to Table 3, the costs are the same for smoothing, restriction, interpolation, and termination. The difference is in the coarse solve, where the code needs to stream through the factored matrix. For our test cases, the factors are larger than the largest cache of the machines. Therefore, we assume that the factors are read from DRAM. Table 4 shows the performance upper bound based on the DRAM sustained bandwidth. On IVB-EP, the best configuration is: elements-per-agglomerate = 256, $\nu_{M^{-1}} = 1$ and $\theta = 0.01$. On KNC, the best configuration is: elements-per-agglomerate = 256, $\nu_{M^{-1}} = 3$ and $\theta = 0.01$. The bandwidth-based performance bound is quite accurate on IVB-EP, yielding an estimate within a gap of 31% of the actual time. Among all the stages, the best match between model and reality is the smoothing step—about an 18% gap. The worst gap corresponds to the coarse solve—about 55%.

On the other hand, the estimated time on KNC is far less than the actual time, implying that the memory bandwidth is severely underutilized. Attributing this significant performance difference to either architecture or model is an area of continued investigation.

Table 4. Runtime bounds and actual runtime (seconds) of the AMG iteration. HSS is used as coarse-grid solver. The column "R/I" represents the combined restriction and interpolation steps. "Stopping" refers to the evaluation of the stopping criterion.

		Memory bandwidth model				
machine		smoothing	R/I	HSS	stopping	total
IVB-EP	model	3.2	1.5	0.9	0.41	6.0
(12-core)	actual	3.8	2.0	1.4	0.88	7.9
KNC	model	1.8	0.22	0.13	0.09	2.25
(60-core)	actual	7.4	0.86	1.9	1.0	10.7

6 Conclusion

We proposed a series of optimizations to improve the performance of the coarse-grid solver. The optimizations aim to expose fine-grained parallelism, exploit high memory bandwidth, and reduce OpenMP overheads. These led to a 2.6× reduction of the AMG solve time on a 60-core Xeon Phi KNC machine. We expect these optimizations to be effective on other manycore architectures as well. We also compared the performance of PCG with `STRUMPACK` when the two algorithms are used as coarse-grid solvers. We found that PCG is at a disadvantage because of its slow convergence. HSS usually leads to a faster AMG cycle, up to 2× faster than PCG. We expect the relative performance of PCG and HSS to depend on the problem. If the problem is such that the AMG parameters can be tuned so as to produce a well-conditioned coarse-grid matrix, then PCG could outperform HSS. Additionally, we explored the parameter space of our AMG algorithm and found high variation in performance. This makes the algorithm a good candidate for an autotuning approach. Our roofline model yields a bound that is within 23% (for PCG) and 31% (for HSS) of the actual performance on the Ivy Bridge EP. The gap is much more significant on the Xeon Phi KNC, which indicates that the bottleneck on that machine is not the memory bandwidth or the FPU but rather the high memory latency. More effective prefetching could hide this latency, but achieving this is challenging because of the relatively primitive hardware prefetchers on that machine, and because of the irregular memory access pattern in our coarse-grid solvers. We are exploring this optimization.

Finally, the only aspect of performance that we considered in our study was time to solution. This is the objective that users care most about. Nevertheless, in future work it would also be valuable to consider other parameters, such as the financial cost of the hardware and its energy efficiency.

Acknowledgments

We thank the anonymous referees for their many comments that greatly helped to improve the paper.

References

1. Intel threading building blocks, `https://www.threadingbuildingblocks.org`
2. Baker, A.H., Schulz, M., Yang, U.M.: On the performance of an algebraic multigrid solver on multicore clusters. In: Proc. of VECPAR '10. pp. 102–115 (2011)
3. Bolz, J., Farmer, I., Grinspun, E., Schröoder, P.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Trans. Graph. 22(3), 917–924 (Jul 2003)
4. Brezina, M., Vassilevski, P.S.: Smoothed aggregation spectral element agglomeration AMG: SA-ρAMGe. In: Lirkov, I., Margenov, S., Waśniewski, J. (eds.) Large-Scale Scientific Computing, LNCS, vol. 7116, pp. 3–15. Springer (2012)
5. Callahan, D., Cocke, J., Kennedy, K.: Estimating interlock and improving balance for pipelined architectures. J. Parallel Distrib. Comput. 5(4), 334–358 (1988)
6. Christie, M.A., Blunt, M.J.: Tenth SPE comparative solution project: Comparison of upscaling techniques. SPE Reserv. Eval. Eng. 4(4), 308–317 (2001)
7. Duff, I.S., Koster, J.: The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J. Matrix Anal. Appl. 20, 889–901 (1999)

8. Gahvari, H., Baker, A.H., Schulz, M., Yang, U.M., Jordan, K.E., Gropp, W.: Modeling the performance of an algebraic multigrid cycle on HPC platforms. In: Proc. of ICS '11. pp. 172–181 (2011)
9. Gahvari, H., Gropp, W., Jordan, K.E., Schulz, M., Yang, U.M.: Modeling the performance of an algebraic multigrid cycle using hybrid MPI/OpenMP. In: Proc. of ICPP '12. pp. 128–137 (2012)
10. Ghysels, P., Li, X.S., Rouet, F.H., Williams, S., Napov, A.: An efficient multi-core implementation of a novel HSS-structured multifrontal solver using randomized sampling. SIAM J. Sci Comput. (2014), preprint
11. Kalchev, D., Ketelsen, C., Vassilevski, P.S.: Two-level adaptive algebraic multigrid for a sequence of problems with slowly varying random coefficients. SIAM J. Sci Comput. 35(6), B1215–B1234 (2013)
12. Kalchev, D.: Adaptive Algebraic Multigrid for Finite Element Elliptic Equations with Random Coefficients. Master's thesis, Sofia University, Bulgaria (2012)
13. Liu, X., Smelyanskiy, M., Chow, E., Dubey, P.: Efficient sparse matrix-vector multiplication on x86-based many-core processors. In: Proc. of ICS '13. pp. 273–282 (2013)
14. Martinsson, P.: A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. SIAM J. Matrix Anal. Appl. 32(4), 1251–1274 (2011)
15. McCalpin, J.D.: Memory bandwidth and machine balance in current high performance computers. IEEE TCCA Newsletter pp. 19–25 (1995)
16. Saad, Y.: A flexible inner-outer preconditioned GMRES algorithm. SIAM J. Sci Comput. 14(2), 461–469 (1993)
17. Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for multicore architectures. Commun. ACM 52(4), 65–76 (2009)
18. Xia, J., Chandrasekaran, S., Gu, M., Li, X.S.: Fast algorithms for hierarchically semiseparable matrices. Numer. Linear Algebra Appl. 17(6), 953–976 (2010)