Efficient Shared-Memory Implementation of High-Performance Conjugate Gradient Benchmark and Its Application to Unstructured Matrices

Jongsoo Park*, Mikhail Smelyanskiy*, Karthikeyan Vaidyanathan*, Alexander Heinecke*,
Dhiraj D. Kalamkar*, Xing Liu†, Md. Mosotofa Ali Patwary*, Yutong Lu‡, Pradeep Dubey*

*Parllel Computing Lab, Intel Corporation
†Georgia Institute of Technology, USA
‡National University of Defense Technology, China

***Abstract*—A new sparse high performance conjugate gradient benchmark (HPCG) has been recently released to address challenges in the design of sparse linear solvers for the next generation extreme-scale computing systems. Key computation, data access, and communication pattern in HPCG represent building blocks commonly found in today's HPC applications. While it is a well-known challenge to efficiently parallelize Gauss-Seidel smoother, the most time-consuming kernel in HPCG, our algorithmic and architecture-aware optimizations deliver 95% and 68% of the achievable bandwidth on Xeon and Xeon Phi, respectively. Based on available parallelism, our Xeon Phi shared-memory implementation of Gauss-Seidel smoother selectively applies block multi-color reordering. Combined with MPI parallelization, our implementation balances parallelism, data access locality, CG convergence rate, and communication overhead. Our implementation achieved 580 TFLOPS (82% parallelization efficiency) on Tianhe-2 system, ranking first on the most recent HPCG list in July 2014. In addition, we demonstrate that our optimizations not only benefit HPCG original dataset, which is based on structured 3D grid, but also a wide range of unstructured matrices.**

I. Introduction

High-performance sparse linear solvers, the back-bone of modern HPC, face many challenges on upcoming extreme-scale architectures. The High Performance Linpack (HPL) [1], widely recognized benchmark for ranking such system, does not represent challenges inherent to these solvers. To address this shortcoming, a new sparse high performance conjugate gradient benchmark (HPCG) has been recently proposed [2]. This is the first paper which analyzes and optimizes HPCG on two modern multi- and many-core IA-based architectures.

Extreme-scale solvers are designed to target exa-scale systems [3], where a large portion of concurrency will occur on the chip [4]. Specifically, these systems will feature nodes with 1000-way parallelism and 100s of GB/s of memory bandwidth. Taking advantage of these resources requires highly tuned single-chip implementation of the key compute-kernels. The two key kernels in the HPCG benchmark, sparse matrix vector multiplication (SpMV) and symmetric Gauss-Seidel smoother (SymGS), are also the key building blocks of various solvers [5, 6]. Achieving high performance of symmetric Gauss-Seidel smoother is particularly challenging due to its limited and fine-grain parallelism [7], especially on many-core architectures with a high amount of concurrency.

We employ optimizations that efficiently exploit a large amount of shared-memory parallelism, while preserving solver convergence rate, outperforming the approach of exploiting parallelism entirely by MPI. Our optimizations do not rely on the underlying geometry of HPCG, and thus also benefit a wide range of unstructured matrices. This paper makes the following contributions:

Point-to-Point Synchronization with Sparsification Method Applied to Gauss-Seidel Smoother on Multi-Core Processors: Since Gauss-Seidel smoother has limited and fine-grain parallelism, it is critical to minimize its synchronization overhead. By applying our point-to-point synchronization with sparsification (denoted P2P hereafter) previously proposed for sparse triangular solvers [7], we achieve a 7.9 GFLOPS of HPCG performance for a 192^3 problem in a 12-core Intel® Xeon® E5-2697 v2 processor. This corresponds to 95% of the stream bandwidth. Our P2P method results in 83 TFLOPS on SuperMUC supercomputer.

A Hybrid of P2P and Multi-Color Reordering Method Applied on Many-Core Processors: Our P2P method further scales well to ∼60 cores in Intel® Xeon Phi™ coprocessors at the finest level of multigrid preconditioner in HPCG. This avoids multi-color reordering at the finest level, thereby minimizing decrease in convergence rate; we apply multi-color reordering only to the coarser levels where parallelism is insufficient and the impact of reordering on convergence rate is smaller. This hybrid implementation achieves a 18 GFLOPS of HPCG performance on a Xeon Phi coprocessor with 16 GB of memory. Running multiple MPI ranks in the coprocessor further improves the performance up to 20 GFLOPS with 12 MPI ranks, which corresponds to 68% of the stream bandwidth. It also leads to 580 TFLOPS on Tianhe-2 supercomputer, which ranked first in July 2014 HPCG list[1].

Block-Color Reordering Applied to Unstructured Matri-

[1]This result is with our implementation that uses block-color reordering and 1 MPI rank per coprocessor. We developed a new hybrid scheme after the acceptance of this paper. The readers are referred to November 2014 HPCG list. We expect the list will have the latest Tianhe-2 result with the new scheme.

SC14, November 16-21, 2014, New Orleans, Louisiana, USA
978-1-4799-5500-8/14/$31.00

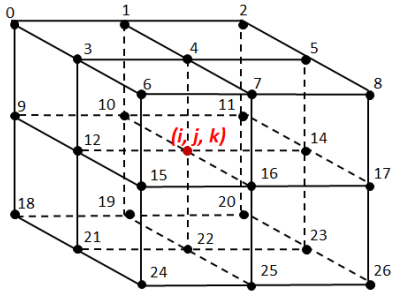


Fig. 1: 27-point stencil used by HPCG.

ces: We extend our work to unstructured matrices and show that block-color reordering method [8] is efficient at exploiting the parallelism in Xeon Phi coprocessors, while minimizing the side effects of reordering with respect to cache locality and convergence rate. We implement highly efficient parallel blocking and coloring methods.

The rest of this paper is organized as follows. Section II briefly introduces and characterizes the HPCG benchmark. Section III compares a few parallelization strategies of the most challenging part of the benchmark, SymGS. Section IV presents our hybrid scheme that is efficient at parallelizing SymGS for many-core Xeon Phi coprocessors. Section V quantifies the performance of our parallelization methods in multi- and many-core IA-based processors, both on the HPCG problem with regular grids and on unstructured matrices. Section VI reviews the related work, and section VII concludes and discusses future directions.

II. Characterization of HPCG

The high performance conjugate gradient benchmark (HPCG) is a recently announced benchmark [2], which uses a preconditioned conjugate gradient (PCG) algorithm to measure the performance of distributed-memory HPC platforms.

HPCG uses a regular 27-point stencil discretization in three dimensions of an elliptic partial differential equation, $\mathcal{L}\{u\} \equiv \vec{\nabla}^2 u = f$. As shown in Fig. 1, the equation associated with red point (i, j, k) at the center depends on the values at its own location and 26 surrounding neighbors. HPCG relies on this geometry to make problem generation scalable and simple to implement. However, its regular 3D structure is not directly exploited by the data structures and kernels. and they are designed to handle general sparse problems. For example, the regular grid is mapped into a matrix A in a general sparse matrix format. The resulting sparse linear system of equations, $Au = f$, has 27 non-zero entries per row for the interior and 8 to 18 non-zeros entries for boundary equations. Matrix A is also symmetric, positive definite.

As a result, HPCG uses the preconditioned conjugate gradient method to solve the linear system of equations. It employs multigrid as a preconditioner, which is used to refine the residual system of equations inside PCG. Multigrid methods provide a powerful technique to accelerate the convergence of iterative solvers for linear systems, when used as standalone solvers, or as preconditioners as in HPCG. Multigrid creates a hierarchy of grid levels and uses corrections of the solution

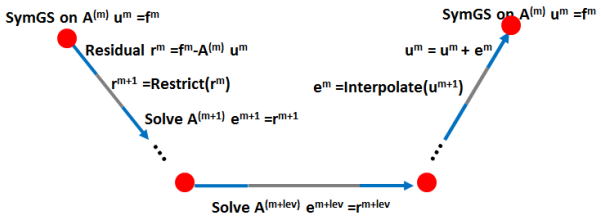


Fig. 2: The V-cycle of Algebraic Multi-grid Solver. Hierarchy of levels is denoted by the superscript over vectors and matrices.

from iterations on the coarser levels to improve the convergence rate of solution at the finer level.

Fig. 2 shows a level schematic of a multgrid V-cycle for solving $A^{(m)}u^m = f^m$. First, a smoother reduces the error while restrictions of the residual create progressively coarser grids. HPCG uses a symmetric Gauss-Seidel smoother (SymGS). The restriction of residual (r^m) is used to define the right-hand side at the next coarser level. Restriction is done by multiplying r^m with a coarsening matrix, defined in the setup phase of multigrid (not shown). In HPCG, the number of coarsening steps is 4. Finally, the coarsest correction is interpolated while going up the V-cycle to progressively finer grids, where it is smoothed. HPCG is composed of the following main kernels:

1) Sparse matrix-vector multiplication (SpMV).
2) Symmetric Gauss-Seidel smoother (SymGS) with forward and backward sweeps that compute the following equations:

$$\mathbf{y} = L_*^{-1}(\mathbf{b} - U\mathbf{x}) \qquad \text{(forward sweep)} \qquad (1)$$
$$\mathbf{z} = U_*^{-1}(\mathbf{b} - L\mathbf{y}) \qquad \text{(backward sweep)}, \qquad (2)$$

   where L_*/U_* are lower/upper triangular and L/U are strictly lower/upper triangular components of A.
3) Restriction and interpolation operators. This part of the benchmark, as an exception, exploits the underlying 3D structure to avoid a full algebraic multigrid implementation. The restriction operator coarsens the grid by a factor of two in all three dimensions, picking one of the points of the fine-grid and storing it into the coarse grid. The interpolation operator inserts values from the coarse grid into the fine grid points that correspond with the restriction operator.
4) Various types of BLAS1 routines, for example, to compute the norm, dot-product, or difference between two vectors.

Unlike HPL, which is bound by floating-point compute capability of a given architecture, HPCG performance is bound by the memory bandwidth, since nearly all HPCG kernels stream in large sparse matrices. These matrices are not typically fit in caches for problems of realistic sizes. The first two kernels, SpMV and SymGS, account for most of the floating-point operations, and the ratio between them is 1:2. While SpMV is embarrassingly parallel and its efficient implementation on Xeon Phi is presented by Liu et al. [9], forward and backward SymGS sweeps have potential loop-carried dependencies, which make parallelization and achieving a high level of bandwidth-bound performance a challenge.

III. Parallelization Challenges of Symmetric Gauss-Seidel Smoother

This section presents parallelization strategies of symmetric Gauss-Seidel smoother and discusses their limitations.

As can be seen from Eq. 1 and 2, performing SymGS translates into two sparse matrix vector multiplications (SpMV), as well as forward and backward triangular solves. Solving triangular system of equations is similar to SpMV, with respect to floating-point operations required (i.e., twice of the number of non-zero elements in the matrix), the amount of data accessed, and data access pattern. However, efficiently parallelizing a sparse triangular solver to achieve bandwidth-bound performance is considerably harder than SpMV, due to its *limited* and *fine-grain* parallelism.

This paper considers three intra-node parallelization approaches: (1) parallelizing SymGS "as is" using task scheduling, (2) increasing the parallelism of SymGS smoother by reordering the matrix, and (3) running multiple MPI ranks within a node. These methods can be combined: e.g., (3) can be combined with (1) or (2) to explore hybrid threaded-MPI parallelism. For each approach, we discuss its impact on the following four properties: the amount of parallelism, data access locality, convergence rate, and MPI communication overhead.

A. Task Scheduling with Point-to-Point Synchronization

This method constructs a *task dependency graph* (TDG) based on the non-zero pattern in the matrix. If there exists a non-zero element at (i, j), smoothing the ith variable depends on smoothing the jth. We construct the graph by adding an edge from j to i for each non-zero element at (i, j). In a TDG, each task represents smoothing one variable. We can quantify the parallelism by measuring the ratio between the total number of non-zeros to the number of non-zeros along the critical path of TDG, assuming smoothing a variable takes a time proportional to the number of non-zeros in the corresponding row. We define the *level* of a task as the length of the longest path from an entry vertex in TDG (distinguish this from levels in multigrid hierarchy). Tasks in a given level can be executed in parallel. For matrices with the same number of non-zeros per row, the parallelism equals the number of rows divided by the number of levels. Pseudo code of task scheduling of SymGS smoother is shown in Fig. 3.

For the 27-point stencil on an N^3 3D grid used in the HPCG benchmark, there are $\sim 7N$ levels[2] and thus the amount of parallelism is $\frac{N^2}{7}$. This is much more *limited* compared to sparse matrix vector multiplication with N^3 parallelism, where each row can be computed in parallel. In addition, smoothing each variable involves only a few floating-point operations, which makes each task *fine-grain*. Considering that each core in modern processors performs tens of operations per cycle, and that core-to-core communication incurs at least tens of cycles of latency [10], synchronization at the granularity of individual tasks can lead to prohibitive overheads.

```
1: for each row i in parallel with task scheduling do
2:     wait until all rows on which i depends are done
3:     sum ← b[i]
4:     for each non-diagonal non-zero j in ith row of A do
5:         sum ← sum − A.value[j] · x[A.colidx[j]]
6:     x[i] ← sum/A.diag[i]
```

Fig. 3: A parallel implementation of forward SymGS sweep with task scheduling with a direct acyclic dependency graph. The task corresponds to row i depends on the task corresponds to row j if there is a non-zero at (i, j) in the lower triangular matrix L or at (j, i) in the upper triangular matrix U.

To address this challenge, we developed a synchronization-efficient task scheduling method that coarsens tasks and eliminates redundant task dependency edges [7]. A conventional parallelization method synchronizes each level of TDG using barriers [11–13]. An N^3 HPCG problem has $7N$ levels, and thus requires $7N$ barriers in the conventional parallelization method. Our method instead uses point-to-point synchronization (denoted P2P hereafter), each of which involves checking a flag set by its parent task. For the HPCG problem, each thread requires checking an average of 2.5 flags per level after redundant dependencies eliminated. This is faster than a barrier synchronization whose latency is proportional to $log(p)$, where p is the number of threads. P2P achieves a performance that is close to the stream bandwidth bound limit for the HPCG problem on a Xeon processor as we will show in Section V. Even for matrices with tens of parallelism, our method achieves more than 80% of the stream bandwidth bound on the Xeon processor [7].

However, in many-core processors, addressing the limited and fine-grain parallelism is substantially more challenging, especially at coarser levels of the multigrid preconditioner in HPCG. For a 192^3 HPCG problem, which is a reasonable size that fits in the GDDR memory of Xeon Phi, the parallelism available in solving SymGS at the finest level of multigrid is ∼5000, or 300K floating point operations. A Xeon Phi coprocessor has ∼240 hardware threads and 8-wide double precision SIMD units with fused-multiply-add support. Therefore, on average per level, each thread is assigned a very small amount of work, ∼75 SIMD instructions. Even for such fine-grain parallelism, our P2P method is capable of saturating about 2/3 of the stream bandwidth limit. However, at the next coarser level, the problem size is halved for each dimension, leading to 1/4 parallelism and thus spending significantly more time at synchronization. Note that this paper also targets unstructured matrices, and many unstructured matrices have far less parallelism.

SpMV and triangular solver in each SymGS sweep can be fused into a single loop to improve the locality as shown in Fig. 3: the outputs of the SpMV part of computation are immediately used by the triangular solver part of computation. This fusion also allows us to reuse the input vector to SpMV ($\mathbf{x}$ in Eq. 1) as the output vector of the triangular solver ($\mathbf{y}$). When parallelizing, this reuse of the vector introduces anti-dependency a to b for each non-zero element at (a, b)

[2]More precisely $7(N-1)+1$ levels. In the HPCG problem, each level corresponds to a slanted plane, whose grid points within are independent with each other. Specifically, these planes are $x + 2y + 4z = k$ such that k is an integer and x, y, and z are non-zero integers smaller than N. We have $7(N-1)+1$ such planes. These slanted planes can be viewed as wave-fronts and thus level scheduling is sometimes called wave-front scheduling.

TABLE I: Comparison of SymGS Smoother Parallelization Methods. Equations in the parallelism column indicate the average number of tasks that can be computed in parallel at any given time. The numbers in parentheses in the convergence column are the number of iterations to reach the same residual as achieved by 50 iterations with the original matrix.

	Parallelism	Locality (L2 cache miss rate)	Convergence (# of iterations)
Task scheduling with the original matrix	$\frac{N^2}{7}$ (not enough for coarser levels)	Degrades from accessing slanted planes with distanced grid points (2.73%)	Not affected (50)
Multi-coloring with b^3-size blocks	$\frac{N^3}{8b^3}$ (enough with appropriate b)	Initially bad. Improves with larger b. (5.56% with $b = 1$, 1.56% with $b = 4$)	Degrades from ignoring the order between rows (66, 58, and 56 for block sizes 1, 2^3, 4^3)
P MPI ranks	$\sqrt[3]{P}\times$ parallelism when applied to the original matrix. No additional parallelism to multi-coloring	Improves from blocking effects (0.46% with sufficiently many ranks)	Degrades from ignoring inter-rank dependencies (53 with 12 ranks, 61 with 60 ranks)

in the upper triangular part of the matrix, in addition to the existing true dependency j to i for each non-zero element at (i, j) in the lower triangular part. But in matrices with mostly symmetric non-zero patterns, this introduces only few additional dependencies.

Scanning the grid level by level degrades the locality of accessing the vector $\mathbf{x}$ shown in Fig. 3. In the original order, a typical access pattern is $(x, y, z), (x+1, y, z), (x+2, y, z), ...$ When grids are accessed level by level, the pattern becomes $(x, y, z), (x-2, y+1, z), (x-4, y+2, z), ...$; we scan slanted planes that correspond to equation $x + 2y + 4z = k$, while the original order scans x-y planes. Since grid points are scanned with a longer stride, the data access locality degrades (but in a smaller amount than multi-color reordering that will be presented in the next section). The original order results in a 1.39% miss rate on a cache simulation for the L2 cache of Xeon Phi with 512 KB capacity, when accessing a vector that corresponds to a 192^3 grid. In the same setting, accessing level by level results in a 2.73% miss rate. The miss rates of both cases stay stable over various grid sizes until they abruptly drop to 0.46% once multiple planes fit in the cache. This miss rate can also be derived analytically as shown in Appendix.

B. Block Multi-Color Reordering

Alternatively, we can reorder the matrix to increase the parallelism. A popular reordering scheme for parallelism is multi-coloring, which groups rows that are not directly dependent [14]. This reordering scheme can be formulated as a graph coloring problem on an undirected adjacency graph induced from the sparse matrix. We run tasks a and c in parallel even with a transitive dependency $a \rightarrow b \rightarrow c$ as long as a and c are not directly connected, and thus multi-color reordering typically finds orders of magnitude more parallelism. For the 27-point stencil on a 3D grid in the HPCG benchmark, the adjacency graph can be colored with 8 colors[3]. Since rows in the same color can be processed in parallel, the amount of parallelism is increased to $\frac{N^3}{8}$.

However, multi-color reordering degrades the locality of accessing the output vector (vector $\mathbf{x}$ shown in Fig. 3) since vertices with the same color are not adjacent in the grid. Even though matrix elements are compactly stored in a sparse matrix format, the associated vector elements accessed together will be far apart, incurring more cache misses. With multi-color reordering, we access grids in a pattern of $(x, y, z), (x+2, y, z), ..., (x+N-2, y, z), (x+N-2, y+2, z), ...$ In addition, the reordering can decrease the convergence [15, 16]. We can increase the number of colors to mitigate the convergence rate drop, but this further degrades the locality of accessing the vectors by separating out grid points with the same color even more.

This locality and convergence rate drop can be addressed by block multi-color reordering [17]. Block multi-color reordering divides the grid into multiple blocks and applies multi-color reordering to the blocks. Since adjacent grid points in a block are processed successively, vector elements will exhibit a higher degree of reuse in the cache, improving the locality. Blocking of vertices also has similar effects as an increase in the number of colors, thus leading to an improvement in the convergence rate. This locality and convergence rate improvement comes at the expense of lower parallelism, but the trade-off can be controlled by the size of blocks. Using B-size blocks decreases the parallelism by B times, while larger blocks give better locality and convergence rate. For the 192^3 problem, multi-color reordering with 1, 8, and 64-size blocks result in 5.56%, 3.70%, and 1.56% miss rates, respectively. These miss rates can also be derived analytically as shown in Appendix. Reordering with 1, 8, and 64-size blocks requires, 66, 58, and 56 iterations, respectively, to reach the same residual as achieved by 50 iterations with the original matrix.

C. Running Multiple MPI Ranks per Node

Running P MPI ranks per node while preserving the problem size per node the same as N^3, each MPI rank works on smaller N'^3 problem with $N' = N/\sqrt[3]{P}$. Therefore, the overall parallelism is $P \cdot \frac{\left(N/\sqrt[3]{P}\right)^2}{7} = \frac{\sqrt[3]{P}N^2}{7}$. In other words, having P ranks within a node provides $\sqrt[3]{P}$ times more parallelism. This assumes that each rank uses task scheduling. Running multiple MPI ranks per node gives no additional parallelism to block multi-color reordering. Running multiple ranks also improves the data access locality. If the problem size per rank is small enough, vector elements corresponding to multiple levels of TDG can fit in the cache, reducing the miss rate to 0.46%.

However, increasing the number of MPI ranks per node has the following side effects. First, the convergence rate drops. By decomposing the matrix for multiple MPI ranks, inter-rank dependencies are ignored, and thus making the corresponding preconditioner less effective. For example, with 60 MPI ranks per Xeon Phi, 61 iterations are required to achieve the same residual obtained by 50 iterations with one MPI rank.

[3]In the 3D grid, a 2×2×2 cube structure repeats $(N/2)^3$ times. The left upper grid points of each small cube can be assigned to the same color because they are not adjacent with each other. The right upper grid points of each small cube can also be assigned to the same color. Therefore, 8 color suffices to cover all grid points.

Second, the MPI communication overhead increases with more MPI ranks per node, both in global reduction for vector inner product and in halo exchange operations. In addition, while the communication between the domains for regular grid (a.k.a. halo exchange) is nearest neighbor communication, it can easily become non-local for unstructured matrices.

Third, having more MPI ranks requires larger memory space as halos are duplicated. This amounts to about 13% increase in the memory allocated for an 192^3 matrix with 60 MPI ranks. The memory space is an important resource in today's coprocessors, which typically use GDDR with capacity between 4 and 16 GB, much smaller than the main memory on modern CPUs.

Table I summarizes the impact of the parallelization approaches presented in this section.

IV. Hybrid Parallelization of Gauss-Seidel Smoother for Many-Core Processors

We combine the three parallelization strategies presented in the previous section to strike the best balance of parallelism, locality, convergence rate, and MPI communication overhead.

For structured matrices such as the one in HPCG, we apply task scheduling with point-to-point synchronization (P2P) to the finest level of multigrid preconditioner, where we have sufficient parallelism. Then, we apply block-color reordering to the coarser levels to address their limited parallelism. Finally, we increase the number of MPI ranks per Xeon Phi coprocessor until the overall performance is maximized. To minimize MPI communication overhead as much as possible, it is important to have an efficient shared-memory implementation, and this section lists a few important optimizations. Since the finest level accounts for a major fraction of the total floating point operations, it is particularly important to highly optimize our P2P method for Xeon Phi.

For unstructured matrices, the amount of parallelism is typically much smaller, and our P2P method is in general not suitable in Xeon Phi. Running multiple MPI ranks also involves higher halo exchange overhead because unstructured matrices usually do not exhibit nearest communication pattern. Therefore, we use block multi-color reordering for unstructured matrices.

A. Optimized Xeon Phi Implementation of Task Scheduling with P2P Synchronization

1) Changing Storage Order Layout of Matrices for Locality: We store the matrix rows in the same order as they will be accessed in SymGS to improve the spatial locality and thus the efficiency of prefetches. We also segregate the matrix rows that will be accessed by different threads in SymGS to avoid false sharing. Specifically, we first store rows that will be accessed by the first thread, then store rows that will be accessed by the second thread, and so on. Within the rows assigned to a thread, we sort by the levels in TDG. Within each level, rows are sorted by their original order. We call this optimization *level relayout* and the order of rows used *level order*. Since the dependencies between smoothing rows are preserved in our task scheduling, level relayout changes nothing mathematically, preserving the convergence rate. Therefore, this should be distinguished from reordering methods such as multi-coloring, hence we call it relayout rather than reordering. We also permute matrix columns using the same level order, and vectors are also kept in the level order over the course of conjugate gradient loop. We permute the vectors using the level order before entering the loop, and permute them back using the inverse of level order after exiting the loop.

2) Saving Memory Space of Matrices: Recall that we add an anti-dependency per non-zero in the upper triangular part when we fuse the SpMV and triangular solver in the forward SymGS sweep as described in Section III-A. If we do a similar for the backward SymGS sweep, TDG for the backward sweep becomes the reverse graph of TDG for the forward sweep: i.e., adding anti-dependencies for the loop fusion makes the two TDGs symmetric. Then, the levels in TDG for the forward sweep are compatible with the TDG for the backward sweep: i.e., tasks in a given level of the former graph can also be executed in parallel in the latter. This allows us to *use the same matrix for both sweeps while preserving the spatial locality of accessing matrix data*. This greatly saves memory space, which is particularly important for Xeon Phi with limited memory. In a backward sweep, we execute tasks assigned to each thread in backward and wait for child tasks to finish (instead of waiting for parents as in the forward sweep). To optimize spatial locality and thus efficiency of prefetches, we traverse rows in a task in a backward order and also access columns in a row in a backward order. We can also reuse the same matrix for SpMV to further save memory space, but this involves trade-offs. Recall that the level order typically exhibits worse locality than the original order. It can be worth keeping the original matrix for SpMV, while permuting the input and output vectors before and after the SpMV, in particular when the locality gap is large and the matrix has many non-zeros per row (so that the cost of permuting the vectors can be amortized). In HPCG, the locality gap between the original and level order is insignificant, and, therefore, reusing the matrix reordered for SymGS in SpMV is faster.

3) Vectorization-Friendly Sparse Matrix Storage Format: We use sliced ELLPACK [18] (SELLPACK) to efficiently vectorize over multiple rows (8 in Xeon Phi)[4]. SELLPACK also improves the cache line utilizations and results in fewer cache lines fetched per gather. To match SELLPACK format, we use 8 rows as the minimum unit when we partition each level among threads.

B. Optimized Xeon Phi Implementation of Block Multi-Color Reordering

1) Parallel Blocking: We represent the matrix as an undirected graph, where we have an edge between i and j for every non-zero element (i, j) such that $i \neq j$. We would like to partition the graph into B-size blocks, while minimizing the number of inter-block edges to improve the locality and the convergence rate [8]. This objective of blocking is similar to that of METIS, a popular graph partitioner. However, we find that METIS incurs long pre-processing times.

Therefore, we implement a light-weight version of graph partitioning that only performs the first step of multi-level

[4]ELLPACK Sparse Block (ESB) introduced by Liu et al. [9] performs better for a wider range of matrices, but we use SELLPACK for fast prototyping. SELLPACK and ESB achieve a similar SpMV performance for HPCG matrices.

partitioning methods. Multi-level partitioning consists of three phases: coarsening, initial partitioning, and uncoarsening. We apply only the first step, coarsening, that is implemented with matching and contraction. Sec. V-D will show that our method achieves convergence rates similar to those of METIS and Iwashita et al. [8]. In contrast to the blocking method presented by Iwashita et al. [8], matching and contraction have well known parallel implementations such as the one in multi-threaded METIS [19]. Sec. V-D will also demonstrate that our matching and contraction implementation provides a high performance in Xeon Phi.

2) Parallel Graph Coloring: For a 27-point stencil on a 3D grid, we can easily color grid points with 8 colors by inspection. However, the HPCG benchmark rules forbid relying on the underlying grid structure; moreover, we target a general algorithm that works for unstructured matrices. We also want coloring to be high performance to minimize the pre-processing overhead. Coloring of a graph is an assignment of positive integers, called *colors*, to the vertices such that the adjacent vertices receive different colors. In block multi-coloring, we apply the coloring to the undirected graph representation of sparse matrix that is coarsened by blocking. Since the graph coloring problem is NP-hard [20], a greedy algorithm, which visits the vertices sequentially and assigns the smallest permissible color, has been shown quite effective in practice [21]. To parallelize the greedy coloring algorithm, we use the *speculation-and-iteration* technique, that has been shown to minimize the number of assigned colors and achieves scalable performance [22, 23]. The idea of the approach is to maximize concurrency by tentatively allowing coloring mistakes and then iteratively detect and resolve those mistakes later.

3) Vectorization of SymGS Smoother for Block-Colored Matrices: Similarly to HPCG matrices, we use SELLPACK format for efficient vectorization over multiple rows, a higher cache line utilization, and fewer cache lines fetched per gather. The SELLPACK format is however not directly applicable to SymGS smoother with block-colored matrices since there are potential sequential dependences between the rows in a given block. Therefore, we merge 8 blocks with the same color, interleaving their rows. In the merged larger block, the first 8 rows are from the first rows of each interleaved block, the second 8 rows are from the second rows of each interleaved block, and so on. Since blocks with the same color are independent, we can easily vectorize these rows (these 8 rows become a slice in the SELLPACK format). This can be viewed as an application of unroll-and-jam compiler transformation.

4) Locality and Memory Space Optimization: Similarly to HPCG matrices, we reorder the rows/columns of matrices and vectors to optimize spatial locality. The rows are first ordered by color, then by block within each color, and finally by the original order within each block. We call this *block-color order*. We use the same matrix for forward/backward SymGS sweeps and SpMV to save the memory space.

5) Adaptive Load Balancing: In unstructured matrices, the sparsity pattern often varies across different parts of the matrix, resulting in a significant load imbalance in SpMV up to 50% in Xeon Phi [9]. A similar load imbalance also occurs in SymGS smoother for multi-color reordered matrices. Since the SpMV and SymGS smoother are invoked multiple times in PCG, we can use performance data from earlier invocations to repartition the matrix to achieve better load balance [24]. We use the partitioning algorithm of Pinar and Aykanat [25] that was also applied to load balancing SpMV implementation in Xeon Phi [9]. In SymGS smoother, the adaptive redistribution is separately done for each color and each sweep direction because, for each case, the underlying data is partitioned differently among threads. We decompose the matrix up to 256 scheduling units per thread. Let C_i be the cost of ith scheduling unit. Suppose that i was one of N tasks assigned to thread p and that thread p took T_p. Then, we update with the rule $C_i = (C_i + T_p/N)/2$, and use C_is to adjust the partition. To be robust from noise, we update the costs every 32 invocations of SpMV or SymGS.

TABLE II: Processors evaluated for single-node performance

	Xeon E5-2697 v2 (IVT)	Xeon Phi 7120 (KNC)
Socket×core×SMT×SIMD	1 × 12 × 2 × 4	1× 61 × 4 × 8
Memory (GB)	64	16
Clock (GHz)	2.7	1.238
L1/L2/L3 Cache (KB)*	32/256/30,720	32/512/No-L3
Double-precision GFLOP/s	259	1,208
STREAM bandwidth [26] (GB/s)	50	177 [27]

*Private L1/L2, shared L3

TABLE III: Supercomputers evaluated for multi-node performance

	SuperMUC	Tianhe-2
The number of nodes used	9,216	15,360
Processors used for compute	Xeon E5-2680	Xeon Phi 31S1P
Processors×core per node	2×8	3×57
Memory per processor (GB)	16	8
Clock (GHz)	2.7	1.1
STREAM BW per processor (GB/s)	35	150
Network fabric	FDR Infiniband	TH Express-2
Network topology	fat-tree	fat-tree

V. EVALUATION

A. Setup

We performed our single-node experiments on the processors shown in Table II. The benchmark was compiled with Intel® C++ compiler version 14.0.1. `IVT` ran 1 thread per core, and `KNC` ran 4 threads per core. We experiment with the multigrid preconditioner with 4 levels, and 1 pre-smoothing and 1 post-smoothing per level.

We evaluated the multi-node performance of Xeon and Xeon Phi on two supercomputers, SuperMUC and Tianhe-2, shown in Table III. In Tianhe-2, the Xeon Phi coprocessors handled all computation, and the host Xeon processors only relayed MPI communication.

B. Single-Node Performance

Fig. 4 shows single-node performance with various SymGS parallelization methods. We run the conjugate gradient algorithm until it reaches the same residual as the one obtained by running the reference version with 50 iterations using the original, naturally ordered, matrix. On a Xeon E5-2697 v2 processor (`IVT`), our point-to-point synchronization method with dependency sparsification (P2P) achieves a 7.9 GFLOPS of HPCG performance. It is 1.8× faster than running the reference

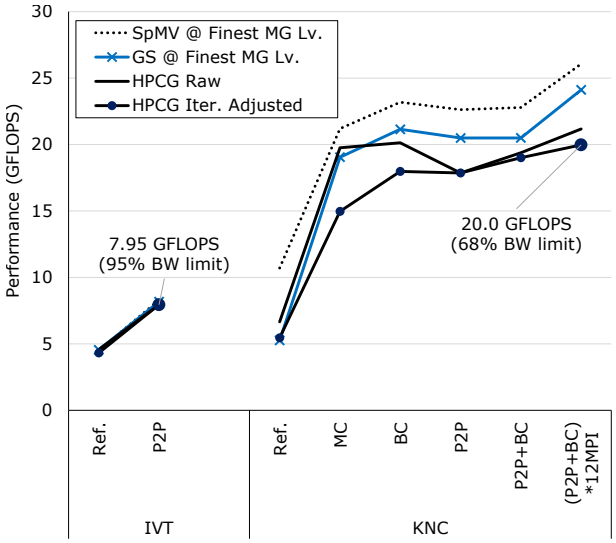


Fig. 4: The performance of HPCG in `IVT` and `KNC` with various parallelization methods.
- `Ref.`: the reference implementation ran with multiple MPI ranks
- `MC`: multi-color reordering
- `BC`: multi-color reordering with size-64 blocks
- `P2P`: low-synchronization scheduling [7] w/o reordering
- `P2P+BC`: `P2P` for finer levels, `BC` for the others
- `(P2P+BC)*12MPI`: 12 MPI ranks, each rank uses `P2P+BC`

"`HPCG Iter. Adjusted`" is adjusted by the number of iterations to reach the same residual: e.g., we multiply 50/66 for `MC` of `KNC` because the original and multi-color reordered matrices take 50 and 66 iterations to reach the same residual, respectively.

code with 1 MPI rank per core (`Ref.`), and corresponds to 95% of the stream bandwidth. As a result, there is no need for reordering or running multiple MPI ranks per processors that incurs convergence rate drop. These findings remain true for many unstructured matrices with limited parallelism, as will be shown in Section V-D.

In comparison, on Xeon Phi 7120 (`KNC`), SymGS at coarser levels of multigrid provides insufficient parallelism to `P2P`. Therefore, the overall HPCG performance with `P2P`, 17.9 GFLOPS, is quite lower than 20.5 GFLOPS SymGS performance at the finest level. Multi-color reordering scheme (`MC`) provides considerably higher parallelism at every level, but SymGS performance at the finest level is lower as 18.0 GFLOPS due to worse locality of accessing the output vector in SymGS. The number of iterations also increases to 66, incurring a 34% drop in the overall HPCG performance. Applying blocking to the multi-color reordering scheme (`BC`) improves the locality and convergence rate. We find that block size 64 strikes the balance of parallelism, locality, and convergence rate, achieving the best HPCG performance with `BC`. It improves SymGS performance at the finest level to 21.1 GFLOPS and decreases the number of iterations to 56, resulting in an 19.0 GFLOPS HPCG performance. `BC` also results in a slightly higher SpMV performance than `P2P` because of better locality (block-color order and level order incurs 1.56% and 2.73% L2 cache miss rates, respectively, as shown in Table I).

`P2P+BC` combines the best of two parallelization approaches. We apply `P2P` to the finest levels, where reordering

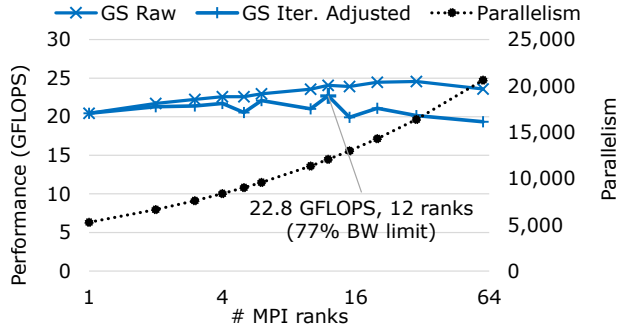


Fig. 5: The performance of SymGS at the finest multigrid level while running multiple MPI ranks in a Xeon Phi coprocessor with 16 GB memory. The problem size solved by the coprocessor is fixed as 192^3 (strong scaling). 60 MPI ranks provide only 4× parallelism and 5 GFLOPS higher performance. The performance adjusted by the number of iterations peaks at 10 MPI ranks.

affects the convergence the most. This reduces the number of iterations to 51. We apply `BC` to the coarser levels (with block size 64 to the second level and block size 1 to the others), addressing limited parallelism in SymGS at those levels. This results in a 20.5 GFLOPS of SymGS performance at the second finest level, significantly higher than 11.0 GFLOPS with `P2P`. As a result, `P2P+BC` has a higher raw HPCG performance than `P2P` (19.4 vs. 17.9 GFLOPS), while they have the same SymGS and SpMV performance at the finest level. Our optimized blocking and coloring result in small pre-processing overheads: they take 2.4× and 0.29× of one CG iteration time, respectively.

Finally, we can run multiple MPI ranks per Xeon Phi to provide further parallelism. Recall from Sec. III-C that running multiple ranks provides no additional parallelism when block multi-coloring is used for all multigrid levels. Fig. 5 plots the performance of SymGS at the finest multigrid level while varying the number of ranks. The raw SymGS performance increases up to 24.6 GFLOPS at 30 ranks. Further increasing the number of ranks drops the performance from MPI communication overhead. The number of iterations also increases up to 61 at 30 and 60 ranks. The SymGS performance adjusted with increase in iterations peaks at 22.8 GFLOPS with 12 ranks. Running 12 ranks also achieves the best overall HPCG performance to 20 GFLOPS on a Xeon Phi coprocessor, which is denoted as `(P2P+BC)*12MPI` in Fig. 4. Since we have fewer threads per rank here, we apply `P2P` to two finer levels, which helps keep the number of iterations as 53. Running multiple ranks helps improving data access locality, which also improves SpMV performance. For each problem size and multigrid level, we can mix and match `P2P` and `BC` with different block sizes combined with varying numbers of ranks per processor. We outperform the reference code running with 240 ranks (`Ref.`) by 3.7×, but the finale HPCG performance of 20 GFLOPS only achieves 68% of the stream bandwidth, despite our careful combination of multiple highly optimized parallelization schemes. This indicates that the HPCG benchmark calls for further innovations whether that can be new algorithms, optimized implementations, or architecture supports, in particular for many-core processors. This contrasts with the Xeon processor that easily achieves 95% of the bandwidth limit by simply applying `P2P` only.

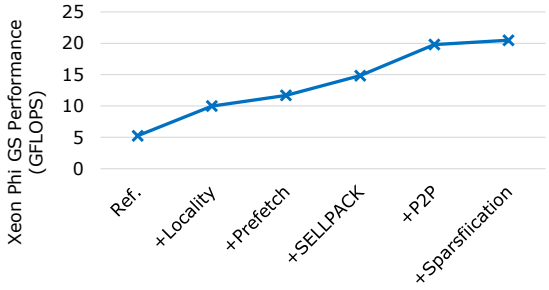


Fig. 6: The impact of optimizations on the Xeon Phi performance of SymGS parallelized with task scheduling.
- `Ref.`: the reference implementation ran with multiple MPI ranks
- `+Locality.`: storage layout optimization for locality (Sec. IV-A1)
- `+Prefetch.`: software prefetches
- `+SELLPACK`: vectorization-friendly matrix storage format [18]
- `+P2P`: point-to-point synchronization instead of barriers
- `+Sparsification`: eliminating unnecessary synchronization [7]

Fig. 6 shows how a series of optimizations progressively improve the performance of SymGS on Xeon Phi using `P2P` at the finest level of multigrid. This illustrates that we should systematically optimize a wide range of performance aspects such as locality, vectorization, and synchronization overhead to realize a high performance SymGS implementation. For example, the vectorization-friendly such as SELLPACK format plays a key role as also shown for SpMV by Liu et al. [9], providing a 27% speedup. Optimizing the storage order of matrices and vectors (`+Locality`) can be applied to other sparse matrix operations. Point-to-point synchronization with dependency sparsification (`+P2P` and `+Sparsification`) can also be applied to other parallel algorithms that are scheduled with directed acyclic graphs.

C. Scaling to Multiple Nodes

Fig. 7 shows the weak scaling performance of HPCG. We present the final HPCG performances that account the convergence and pre-processing overheads. On SuperMUC, the CG loop runs 50 iterations and pre-processing takes 0.6 seconds, accounting for 0% and 0.7% drops in the final HPCG performance, respectively. On Tianhe-2, the CG loop runs 57 iterations and pre-processing takes 3.9 seconds, accounting for 13% and 5% drops in the final HPCG performance. The SuperMUC result is particularly impressive, considering that it corresponds to 77% of the aggregated stream bandwidth. Note that Tianhe-2 results presented use `BC`. We expect a higher HPCG raw performance, a faster convergence, and a shorter pre-processing time with our hybrid scheme, `P2P+BC`. Both supercomputers achieve more than 80% parallelization efficiency. The gap from the ideal scaling stems from halo exchange and all-reduce times. SuperMUC spends 3% and 16% of the total time at halo exchange and all-reduce, respectively, and Tianhe-2 spends 10% and 9%. We can observe a noticeable decrease in parallelization efficiency on SuperMUC going from 512 to 1024 nodes. This is caused by its fat-tree layout that provides 4× reduced bandwidths between leaves consisting of 512 nodes.

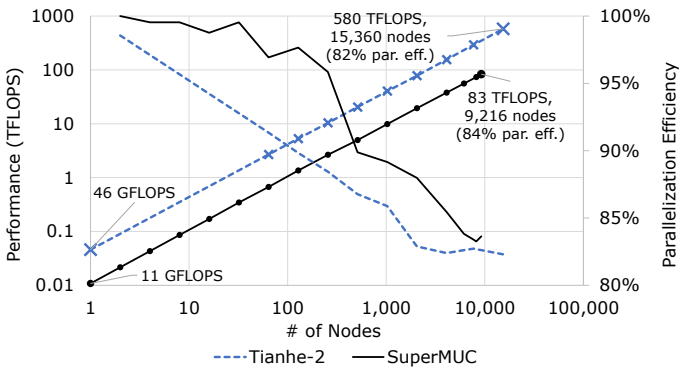


Fig. 7: Weak scaling in SuperMUC and Tianhe-2. SuperMUC runs 1 MPI rank and 128^3 sub-grid per Xeon processor socket. Tianhe-2 runs 1 MPI rank and 176^3 sub-grid per Xeon Phi coprocessor.

D. Application to Unstructured Matrices

Since our optimizations do not rely on the underlying structure of the HPCG problem, they can be applied to the unstructured matrices listed in Table IV. Fig. 8(a) shows the performance of our HPCG implementation for the evaluated unstructured matrices. Since the multigrid preconditioner within HPCG relies on the grid geometry for restriction and interpolation, we use a SymGS preconditioner, effectively setting the number of multigrid levels to 1. For each matrix, we compare `KNC` performance with various shared-memory parallelization schemes against the best result on `IVT`. For all matrices, the best performance in `IVT` can be obtained without reordering, similar to the HPCG problem. They achieve performance very close to the stream bandwidth limit, 8.3 GFLOPS = (50 GB/s)/(6 bytes per FLOP ratio in HPCG).

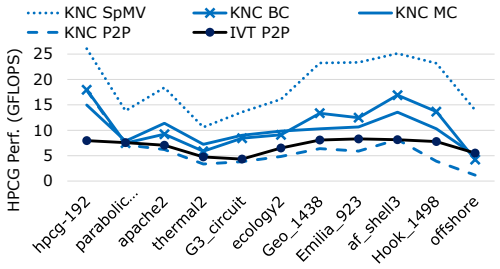
Fig. 8(a) demonstrates that block multi-color reordering is essential at exploiting parallel compute resources in `KNC` for the unstructured matrices. Fig. 8(b) plots available parallelism in SymGS with various shared-memory parallelization schemes. Since most of the evaluated unstructured matrices have considerably smaller parallelism than the regular HPCG problem with `P2P`, `KNC P2P` is slower than `IVT P2P`. The parallelism is substantially increased by multi-color reordering for all matrices. `BC` results in on average 9× lower parallelism than `MC` for the unstructured matrices. This is expected because we use block size 8 and typically the amount of parallelism is decreased proportionally with the block size. Still, `BC` provides a sufficient amount of parallelism to `KNC` in order to maintain the overhead from synchronization and load imbalance small. The adaptive load balancing scheme presented in Section IV-B5 further helps keep load imbalance to be lower than 20% (this provides an average of 5% overall speedup to `KNC BC`).

`BC` improves the convergence rate compared to `MC`, as shown in Fig. 8(c). While `MC` slows down the convergence rate by an average of 17%, `BC` reduces the slowdown to an average of 9%.

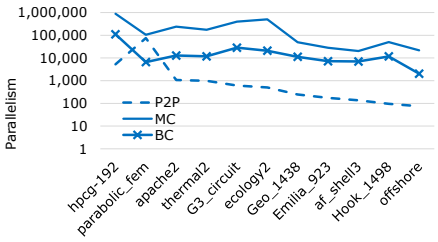
Fig. 8(d) demonstrates that `BC` also improves the data access locality compared to `MC`, reducing cache misses. However, this figure shows that `BC` still incurs considerably more misses than the original matrix order. This locality degradation is the biggest limiting factor in achieving a higher CG performance in `KNC BC`. Fig. 8(a) includes `KNC SpMV` that

TABLE IV: Unstructured matrices from the Univ. Florida collection [28].

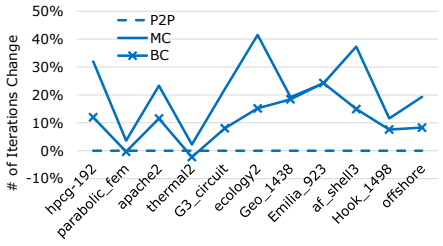
	rows	nnz/row
1. hpcg-192	7,078K	27
2. parabolic_fem	526K	7
3. apache2	715K	7
4. thermal2	1,228K	7
5. G3_circuit	1,585K	5
6. ecology2	1,465K	14
7. Geo_1438	1,438K	44
8. Emilia_923	923K	44
9. af_shell3	505K	35
10. Hook_1498	1,498K	41
11. offshore	260K	16



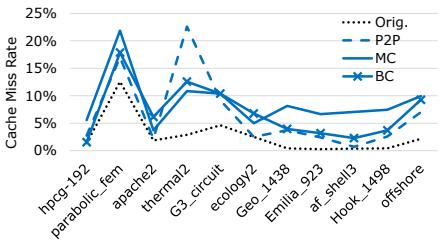
(a) *HPCG Performance*. The impact of reordering on the number of iterations is accounted. The performance of HPCG can approach `KNC SpMV` (SpMV with the original matrix) in an ideal situation with abundant parallelism, no locality degradation, and no convergence rate drop.



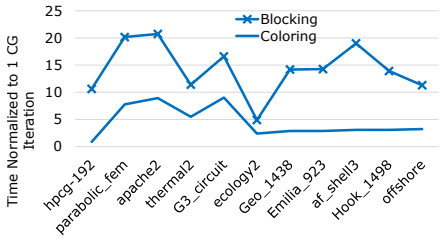
(b) *Parallelism*. Measured as (# of non-zeros)/(cumulative # of non-zeros in the rows corresponding to the longest dependency path).



(c) *Convergence Rate*. Measured as the number of iterations to reach 10^{-5} residual (for `hpcg-192`, the residual obtained by 50 iterations with `P2P` is used, following the HPCG benchmark rule).



(d) *Locality*. Measured as miss rates of accessing vectors. We simulate a `KNC` L2 cache with 512 KB capacity and 64-byte lines. `Orig.` denotes miss rates that is incurred by SpMV with the original matrix.



(e) *Pre-processing Overhead* in `BC`. Blocking uses a light-weight graph partitioning method based on graph matching and contraction.

Fig. 8: HPCG performance for unstructured matrices (an HPCG problem with a 192^3 grid is also shown for comparison), and analyses of parallelism in SymGS, convergence rate, data access locality, and pre-processing overhead. `BC` uses size-64 blocks for `hpcg-192` and size-8 blocks for the others. The number of multigrid levels is set to 1 for the unstructured matrices.

denotes the SpMV performance with the original matrix order, which is an upper bound of HPCG performance. The factors that affect the gap from this upper bound are parallelism, locality, and convergence rate, and we discussed above that `BC` provides sufficient parallelism with minimal convergence rate drop; therefore, locality remains as the biggest limiting factor. Further improving the data access locality remains as future work. On `IVT`, the vectors typically fit in the shared cache, and out-of-order cores can hide the latency of L2 misses that hit the shared cache. On `KNC` without a shared cache, most of the cache misses are satisfied by the caches of remote cores, and, therefore, improving the core-to-core latency can close the gap between `KNC BC` and `KNC SpMV`.

Fig. 8(e) presents the pre-processing time of `BC` normalized to the time for 1 CG iteration. Based on the expected number of iterations, this can guide whether the pre-processing cost can be amortized sufficiently. The pre-processing overhead is kept small by our parallel implementations of blocking and coloring with high performance. Our blocking implementation is based on graph matching and contraction, which corresponds to the first step of multi-level graph partitioning method [19] and is significantly faster than METIS that implements the full multi-level graph partitioning. In contrast to the blocking method presented by Iwashita et al. [8], matching and contraction have well known parallel implementations such as the one in multi-threaded METIS [19]. Our method, METIS, and Iwashita et al. [8] exhibit similar parallelism and convergence rates: 9×, 11×, and 8.9× lower parallelism than `MC`, and 9%, 7%, and 9% convergence rates drops, respectively.

VI. Related Work

Sparse solver benchmarks similar to HPCG have been proposed before. Specifically, the NAS parallel benchmarks (NPB) [29], widely used for HPC analysis, includes a CG benchmark. Even though it shares many attributes with HPCG, it has non-physical, random sparsity pattern, and uses no preconditioning. An iterative solver benchmark [30] includes preconditioned CG and GMRES as well as physically meaningful sparsity patterns together with several preconditioners, but it does not address scalable distributed memory parallelism.

A number of authors explored multi-coloring techniques within sparse preconditioned parallel solvers to improve their scalability in the context of both CPUs and GPUs [31, 32]. Iwashita et al. [8] also propose algebraic block-coloring, which we found critical for efficient implementation of symmetric Gauss-Seidel smoother on Xeon Phi. Our work however evaluates several alternative block coloring schemes, and develops its highly optimized parallel implementation on multi- and many-core architectures, for both HPCG-based as well as general matrices.

HPCG uses a conjugate gradient solver with a multigrid preconditioner. There is a large body of work on optimizing multigrid, both geometric and algebraic, for modern multi- and many-core architectures, such as CPUs [33], GPUs [34, 35] and Xeon Phi [36]. Our work expands previous efforts by

developing a unique set of algorithmic and architecture-aware optimizations in the context of both the new HPCG benchmark as well as unstructured problems in general.

A bulk of our optimizations are focused on the symmetric Gauss-Seidel smoother kernel within multigrid preconditioner. Optimizing the symmetric Gauss-Seidel smoother as well as sparse triangular solver kernels on multi- and many-core architecture has been done by a number of researchers in the past [13, 37–39]. Our work leverages recently proposed sparsification technique [7], originally evaluated on sparse triangular solver, to improve the performance of symmetric Gauss-Seidel smoother.

VII. Conclusion and Future Work

This paper demonstrates that we can achieve a high level of HPCG performance in modern multi- and many-core processors using shared-memory parallelization despite the challenges in the symmetric Gauss-Seidel smoother. This high performance is enabled by a careful combination of synchronization-efficient task scheduling, block multi-color reordering, and architecture-aware optimizations for locality and vectorization efficiency. The optimizations described in this paper are not limited to the HPCG benchmark and can also be applicable to other problems and sparse solvers as exemplified by our evaluation with unstructured matrices. We expect to see more innovations continued from our work on the HPCG benchmark, whose examples are listed as follows:

Multi-node Scalability: We can improve multi-node scalability by overlapping halo exchange with SpMV and SymGS. Since this involves fissioning loops, we should be careful about its negative impact on single-node performance from locality loss. It also remains to be shown what is the optimal number of ranks per Xeon Phi coprocessor when many coprocessors are used. Applying recent MPI features such as one-sided communication also appears to be a promising direction to pursue.

Further Locality Optimization: In `P2P`, accessing each level following the original matrix row order results in decomposing the corresponding slanted plane into threads in 1D: a thread accesses in a pattern of $(x, y, z), (x-2, y+1, z), (x-4, y+2, z), ...$, scanning a line. Instead, we can apply graph partitioning to each level to employ a 2D decomposition.

Auto Tuning: We can mix and match `P2P`, `BC` with different block sizes, and varying MPI ranks. We can develop an analytical model that predicts the best combination based on factors such as parallelism available in the matrix and the degree of concurrency in the target system.

Other Applications: Our pre-processing steps involve various graph algorithms such as coloring, matching, and transitive edge reduction. Scalable parallel implementations and fast approximate algorithms for these problems should benefit other data-intensive applications. Our `P2P` synchronization with dependency sparsification can also be applied to other directed acyclic graph scheduling problems.

Architecture Support for Bandwidth-Bound Applications with Limited Parallelism: If our `P2P` method relying on fine-grain parallelization can be faster than other parallelization schemes in most cases, it can greatly help programmability as in our simple Xeon implementation. Efficient vectorization of short loops with hardware support for horizontal add could improve the performance of CSR format, and thus can expose more parallelism than the formats that group multiple matrix rows. With faster inter core-to-core communication, this could allow exploiting even finer-grain parallelism. The ability to saturate the bandwidth with fewer threads can also greatly help `P2P` method as exemplified by our Xeon performance that is close to the bandwidth limit. It can also help other bandwidth-bound applications with limited parallelism and allow energy saving opportunities such as power gating idle cores.

Appendix
Analyses of cache miss rates of HPCG

We analyze the cache miss rates in accessing the input vectors of SymGS. Since SpMV involves a mostly similar data access pattern, the same analysis can be applied to SpMV. In contrast to SpMV, SymGS cannot however freely reorder the matrix rows due to the loop carried dependencies, and we consider the original order and the block-color order here. Although the analysis shown here assumes a 27-point stencil in a 3D grid, it can be extended to other structured problems.

The original order: Suppose that an x-y plane does not fit in the cache; i.e., we reuse along x and y directions, but not along z. Then, among 27 input grid points used for 1 output grid at (x, y, z), we miss 3 points at $(x+1, y+1, *)$, where $*$ denotes wildcard. This leads to a miss rate of $3/27/8 =$ **1.39%**. We divide by 8 assuming a perfect cache line utilization, which is realized by SELLPACK sparse matrix format [18]. When there is reuse along z, then only one misses among the 27 inputs, and the miss rate becomes **0.46%**.

The block-color order: Let us first look at multi-color reordering without blocking. Suppose again that we reuse along x and y directions, but not along z. Then, among 27 input grid points, we miss 12 points at $(x, y, *), (x+1, y, *), (x, y+1, *), (x+1, y+1, *)$. This leads to a miss rate of $12/27/8 =$ **5.56%**, where 8 is divided to consider the cache line size. For multi-color reordering with b^3 size blocks, we fetch $(b+2)^3$ grid points for each block. This results in $\left(\frac{b+2}{b}\right)^3/27/8$ miss rates, which is **3.70%** and **1.56%** for b equals 2 and 3, respectively.

Acknowledgements

First of all, the quick and competent help of the staff at LRZ and NSCC-GZ is highly appreciated. Although its results are not presented, Stampede was used as a test bed, and we thank Carlos Rosales-Fernandez and the other staff at TACC, Khaled Hamidouche and the other MVAPICH team members at OSU, Dmitry Dumov, and Ravi Narayanaswamy for supporting infrastructure at Stampede. The authors would like thank Michael A. Heroux for his insights related to the implementation and performance of the high-performance conjugate gradient benchmark, and David S. Scott and Alexander A. Kalinkin for discussion during the initial stage of our project. We thank Arif Khan, Nadathur Satish, and Narayanan Sundaram for discussion on parallel matching. We also thank Ludovic Sauge at BULL, Shane Story, and Vadim Pirogov for help communicate with various institutions.

REFERENCES

[1] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary, "HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers," http://www.netlib.org/benchmark/hpl/.

[2] J. Dongarra and M. A. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems," Sandia National Laboratories, Tech. Rep. 4744, 2013.

[3] "Workshop on Extreme-Scale Solvers: Transition to Future Architectures," in *American Geophysical Union*, Washington, DC, 2012.

[4] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, and K. Yelick, "ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008, `www.cse.nd.edu/Reports/2008/TR-2008-13.pdf`.

[5] E. Rothberg and A. Gupta, "Parallel ICCG on a Hierarchical Memory Multiprocessor - Addressing the Triangular Solve Bottleneck," *Parallel Computing*, vol. 18, no. 7, 1992.

[6] V. E. Henson and U. M. Yang, "BoomerAMG: a Parallel Algebraic Multigrid Solver and Preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2000.

[7] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying Synchronization for High-Performance Shared-Memory Sparse Triangular Solver," in *International Supercomputing Conference (ISC)*, 2014.

[8] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2012.

[9] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey, "Efficient Sparse Matrix-Vector Multiplication on x86-Based Many-Core Processors," in *International Conference on Supercomputing (ICS)*, 2013.

[10] R. S. Daniel Molka, Daniel Hackenberg and M. S. Müller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[11] E. Anderson and Y. Saad, "Solving Sparse Triangular Linear Systems on Parallel Computers," *International Journal of High Speed Computing*, vol. 1, no. 1, 1989.

[12] J. H. Saltz, "Aggregation Methods for Solving Sparse Triangular Systems on Multiprocessors," *SIAM Journal of Scientific and Statistical Computing*, vol. 11, no. 1, 1990.

[13] M. Naumov, "Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU," NVIDIA Corporation, Tech. Rep. 001, 2011.

[14] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2003.

[15] E. L. Poole and J. M. Ortega, "Multicolor ICCG Methods for Vector Computers," *SIAM Journal on Numerical Analysis*, vol. 24, no. 6, 1987.

[16] H. C. Elman and E. Agrón, "Ordering techniques for the preconditioned conjugate gradient method on parallel computers," *Computer Physics Communications*, vol. 53, no. 1, 1989.

[17] T. Iwashita and M. Shimasaki, "Block Red-Black Ordering: A New Ordering Strategy for Parallelization of ICCG Method," *International Journal of Parallel Programming*, vol. 31, no. 1, 2003.

[18] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures," in *International Conference on High Performance Embedded Architectures and Compilers (HiPEAC)*, 2010.

[19] D. LaSalle and G. Karypis, "Multi-Threaded Graph Partitioning," in *International Symposium on Parallel and Distributed Processing (IPDPS)*, 2013.

[20] A. H. Gebremedhin, F. Manne, and A. Pothen, "What Color Is Your Jacobian? Graph Coloring for Computing Derivatives," *SIAM review*, vol. 47, no. 4, 2005.

[21] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "ColPack: Software for Graph Coloring and Related Problems in Scientific Computing," *ACM Transactions on Mathematical Software (TOMS)*, vol. 40, no. 1, 2013.

[22] M. M. A. Patwary, A. H. Gebremedhin, and A. Pothen, "New Multithreaded Ordering and Coloring Algorithms for Multicore Architectures," in *Proceedings of 17th International European Conference on Parallel and Distributed Computing (Euro-Par 2011)*, 2011.

[23] M. M. A. Patwary, A. H. Gebremedhin, F. Manne, and A. Pothen, "Paradigms for Effective Parallelization of Inherently Sequential Graph Algorithms on Multi-core Architectures," 2013, ACM Transactions on Parallel Computing, under review.

[24] S. Lee and R. Eigenmann, "Adaptive Runtime Tuning of Parallel Sparse Matrix-Vector Multiplication on Distributed Memory Systems," in *International Conference on Supercomputing (ICS)*, 2008.

[25] A. Pinar and C. Aykanat, "Fast Optimal Load Balancing Algorithms for 1D Partitioning," *Journal of Parallel and Distributed Computing*, 2004.

[26] J. D. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," http://www.cs.virginia.edu/stream.

[27] K. Raman, "Optimizing Memory Bandwidth on Stream Triad," https://software.intel.com/en-us/articles/optimizing-memory-bandwidth-on-stream-triad, 2013.

[28] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Transactions on Mathematical Software*, vol. 15, no. 1, 2011, http://www.cise.ufl.edu/research/sparse/matrices.

[29] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS Parallel Benchmarks," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 1991.

[30] J. Dongarra, V. Eijkhout, and H. van der Vorst, "An iterative solver benchmark," *Sci. Program.*, vol. 9, no. 4, Dec. 2001.

[31] R. Li and Y. Saad, "GPU-Accelerated Preconditioned Iterative Linear Solvers," Technical report, University of Minnesota, Tech. Rep., 2010.

[32] V. Heuveline, D. Lukarski, and J.-P. Weiss, "Enhanced Parallel ILU(p)-based Preconditioners for Multi-core CPUs and GPUs – The Power(q)-pattern Method," *Preprint Series of the Engineering Mathematics and Computing Lab*, vol. 0, no. 08, 2011.

[33] A. H. Baker, R. D. Falgout, T. Gamblin, T. V. Koleg, M. Schulz, and U. M. Yang, "Scaling Algebraic Multigrid Solvers: On the Road to Exascale," *Competence in High Performance Computing 2010*, vol. 31, 2012.

[34] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, Jul. 2003.

[35] N. Bell, S. Dalton, and L. Olson, "Exposing Fine-Grained Parallelism in Algebraic Multigrid Methods," NVIDIA Corporation, NVIDIA Technical Report NVR-2011-002, Jun. 2011.

[36] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of Geometric Multigrid for Emerging Multi- and Manycore Processors," in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[37] E. Anderson and Y. Saad, "Solving Sparse Triangular Linear Systems on Parallel Computers," *Int. J. High Speed Comput.*, vol. 1, no. 1, Apr. 1989.

[38] D. Wallin, H. Löf, E. Hagersten, and S. Holmgren, "Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors," in *International Conference on Supercomputing (ICS)*, 2006.

[39] M. M. Wolf, M. A. Heroux, and E. G. Boman, "Factors Impacting Performance of Multithreaded Sparse Triangular Solve," in *Proceedings of the 9th International Conference on High Performance Computing for Computational Science*, 2011.