Optimizing Multi-grid Computation and Parallelization on Multi-cores

Xiaojian Yang*
National University of Defense Technology
China
yangxj@nudt.edu.cn

Shengguo Li*
National University of Defense Technology
China
nudtlsg@nudt.edu.cn

Fan Yuan
Xiangtan University
China
fyuan@smail.xtu.edu.cn

Dezun Dong†
National University of Defense Technology
China
dong@nudt.edu.cn

Chun Huang
National University of Defense Technology
China
chunhuang@nudt.edu.cn

Zheng Wang
University of Leeds
United Kingdom
z.wang5@leeds.ac.uk

ABSTRACT

Multigrid algorithms are widely used to solve large-scale sparse linear systems, which is essential for many high-performance workloads. The symmetric Gauss-Seidel (SYMGS) method is often responsible for the performance bottleneck of MG. This paper presents new methods to parallelize and enhance the computation and parallelization efficiency of the SYMGS and MG algorithms on multi-core CPUs. Our solution employs a matrix splitting strategy and a revised computation formula to decrease the computation operations and memory accesses in SYMGS. With this new SYMGS strategy, we can then merge the two most time-consuming components of MG. On top of these, we propose a new asynchronous parallelization scheme to reduce the synchronization overhead when parallelizing SYMGS. We demonstrate the benefit of our techniques by integrating them with the HPCG benchmark and two real-life applications. Evaluation conducted on four architectures, including three ARMv8 and one x86, shows that our techniques greatly surpass the performance of engineer- and vendor-tuned implementations across various workloads and platforms.

CCS CONCEPTS

• **Mathematics of computing** → **Solvers**; **Mathematical software performance**; • **Computing methodologies** → **Massively parallel algorithms**.

KEYWORDS

Multigrid, symmetric Gauss-Seidel, Asynchronous parallelization

*Both authors contributed equally to this research.
†Corresponding author

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '23, June 21–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0056-9/23/06…$15.00
https://doi.org/10.1145/3577193.3593726

ACM Reference Format:
Xiaojian Yang, Shengguo Li, Fan Yuan, Dezun Dong, Chun Huang, and Zheng Wang. 2023. Optimizing Multi-grid Computation and Parallelization on Multi-cores. In *2023 International Conference on Supercomputing (ICS '23), June 21–23, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3577193.3593726

1 INTRODUCTION

Sparse linear solvers underpin many high-performance scientific and industrial workloads for modeling our physical world. Examples of such workloads include a wide range of applications seen in computational flow dynamics [49] and environmental science [38, 62].

Sparse linear solvers can be implemented using direct or iterative methods. As iterative methods tend to be more efficient and easier to implement on parallel systems than the direct counterpart [52], most solvers choose to use iterative methods like the conjugate gradient (CG) method [22, 52] for solving large-scale linear systems. The convergence of an iterative solution is usually accelerated by a *multigrid* (MG) process. This is done by using an MG method to generate a sequence of grids (or meshes) through successive refinement, where grids are graded from fine to coarse. In this way, the iterative solution of the fine linear system is accelerated by solving the coarser systems. As such, the MG computation rate greatly influences the efficiency of an iterative method. However, the MG computation time can be expensive on large-scale problems that can have millions of equations and variables [10], making optimizing MG algorithms a non-trivial challenge.

The most time-consuming step of MG is to reduce high-frequency errors from coarser grids - a process known as *smoothing*. The smoothing operation is usually realized by using a symmetric Gauss-Seidel (SYMGS) method [56] to recursively compute on a collection of grids. Studies have shown that SYMGS generally converges much faster than alternative methods like the Jacobi method on multi-core CPUs [52], and hence it is the dominant approach for implementing MG. Unfortunately, the Gauss-Seidel smoother has a major drawback - it exhibits a low degree of parallelism due to extensive computation dependencies [43]. As we will show later in the paper, smoothing can account for 78% of the MG computation

time and should not be ignored when optimizing MG and sparse linear solvers.

Efforts have been made to parallelize the Gauss-Seidel method. Parallelization can be achieved by either using graph coloring techniques to group the independent variables of the linear system to solve partitions in parallel [2, 28, 46], or exploiting pipeline parallelism in the directed acyclic computation graph of a given grid setting through level scheduling [1, 40, 42]. These prior schemes represent an important step for optimizing the Gauss-Seidel method by focusing on computation parallelization. However, little work has attempted to optimize the memory access latency of SYMGS on multi-core CPUs. SYMGS is known to be memory-bounded because the algorithm needs to access large, sparse matrices that cannot fit into the last level cache and the kernel computation has a low arithmetic intensity [63]. Reducing the memory access latency is essential for gaining further performance improvement for SYMGS. The issue is increasingly important since the performance gap between the CPU and the memory subsystem becomes wider [7, 8].

This paper aims to push the boundary of memory and computation optimization of SYMGS on homogeneous multi-cores. Our approach revises the classical Gauss-Seidel method used by mainstreamed MG algorithms to reduce the number of computation and memory access operations. Our revision preserves the SYMGS semantics without changing the computation outcome, yet it can lead to noticeable performance improvement. We achieve this by first altering how sparse matrices are stored according to the computation characteristics of SYMGS. It allows us to merge sparse matrix-vector (SPMV) multiplication and SYMGS kernels, which dominate the execution time of MG, to reduce the number of memory accesses and computation operations. Although it may appear strikingly simple, our technique outperforms state-of-the-art MG optimization approaches [17, 32, 47, 54] by a large margin. Our computation optimization is orthogonal to existing SYMGS parallelization strategies, allowing a linear solver to leverage well-established parallelization strategies to exploit parallelism.

In addition to computation optimization on a single core, we propose a new way to improve parallelization efficiency using asynchronous pipeline parallelism across processor cores. This is based on the observation that the commonly used block multi-color (BMC) reordering strategy [27, 28] for parallelizing SYMGS can lead to frequent stalls of parallel threads, leading to sub-optimal performance for parallel execution. Our approach advances the standard BMC algorithm by using pipeline parallelization to exploit parallelism, allowing parallel threads to start computation asynchronously. Our strategy not only reduces the synchronization overhead but also improves the multi-core utilization by allowing parallel threads to start computation as soon as the data dependence is resolved.

Our techniques are generally applicable. We have integrated our techniques[1] to the high performance conjugate gradient (HPCG) benchmark [15, 43, 51] and two real-life application: YHAMG [17] and CitcomCU [53]. The HPCG benchmark provides a representative implementation of MG and is commonly used to evaluate the performance of high-performance systems. Furthermore, YHAMG is a highly optimized algebraic multigrid (AMG) library with multiple smoothers such as SYMGS, Jacobi, and Chebyshev polynomials. CitcomCU is a widely used finite element solver for earth simulations [5].

We evaluate our approach on both Intel x86 and ARMv8 multi-core processors, including an Intel Xeon and three ARM multi-core systems: Phytium 2000+ [45], Kunpeng (KP) 920 [26] and Thunder X2 [36]. We compared our approach against heavily-tuned MG implementations accelerated with industry-strength high-performance libraries as well as prior parallelization methods designed for MG algorithms [47, 51]. We evaluated our techniques on both a single computing node and in a distributed setup with 256 computing nodes.

Our approach outperforms prior works by a large margin. On the ARM platforms, it improves the ARM-specific HPCG implementation accelerated using the ARM Performance Library (ARMPL) [4] by 1.92x–2.33x. On the Intel platform, our approach improves the heavily-optimized HPCG version in the Intel Math Kernel Library (MKL) [47] by 1.40x. We show that such a performance advantage is transferable to real applications where our techniques give a 1.4x and 3.1x speedup over the engineer-tuned implementation of YHAMG and CitcomCU applications, respectively.

This paper makes the following contributions:

(1) It proposes a new SYMGS algorithm to reduce the computation and memory access costs on multi-cores (Section 3.1);
(2) It provides an empirical study on how to determine the best parameters of the commonly used block multi-color reordering parallelization method (Section 4.2);
(3) It presents an asynchronous method to reduce the synchronization overhead when parallelizing SYMGS (Section 4.3).

2 BACKGROUND

2.1 Multigrid Algorithms

Iterative methods have proven effective in solving linear systems arising in physical modeling [14, 52]. MG algorithms are often employed to implement an iterative solution for solving a linear system, $Ax = b$, where A, b, and x are a sparse matrix, a dense vector and a result vector respectively.

Although a diverse range of MG algorithms and implementations is available [6, 11, 13, 25, 41, 60], they typically operate on a hierarchy of discretizations (grids). There are different choices of MG methods with varying trade-offs between the speed of solving a single iteration and the rate of convergence across iterations, with the V-Cycle being the most commonly used implementation [52, 55]. We use V-Cycle as a working example, but our techniques can be equally applied to other mainstream MG implementations that use SYMGS, including F-Cycle and W-Cycle.

Algorithm 1 outlines how V-Cycle solves an h-level linear system, $A^h x = b$. An MG *cycle* has five operators: *bottom solver*, *pre-smoothing*, *post-smoothing*, *restriction* and *prolongation*. The cycle is implemented using *recursion* by repeatedly applying the MG function to *coarser* grids (line 8 in Algorithm 1). This process is repeated until the coarsest grid is reached where the cost of the direct solution is negligible so that a *bottom solver* (line 3 in Algorithm 1) is used to solve this collection of grids directly. Before and after operating on a finer grid, *pre-smoothing* (line 5 in Algorithm 1) and *post-smoothing* (line 10 in Algorithm 1) are respectively applied to cancel out errors from the coarser grids. Smoothing is followed by computing the *residual* error after the smoothing operation.

[1] Code and data available at https://github.com/YXJ-123/MGopt-APP.

Algorithm 1: A standard V-Cycle MG algorithm.

```
1  function MG(A^h, b^h, x_0^h)
2      if on the coarsest level then
3          x^h = BottomSolver(A^h, b^h, x_0^h)          ▷ Bottom solver
4      else
5          repeat m_1: x^h = SYMGS(A^h, b^h, x_0^h)     ▷ Presmoothing
6          r^h = b^h − A^h x^h                          ▷ Residual
7          r^2h = I_h^2h r^h, x_0^2h = 0                ▷ Restriction
8          x^2h = MG(A^2h, r^2h, x_0^2h)                ▷ Recursion
9          x^h = x^h + I_2h^h x^2h                      ▷ Prolongation
10         repeat m_2: x^h = SYMGS(A^h, b^h, x^h)       ▷ Postsmoothing
11     end
12     return x^h
13 end
```

Then, a *restriction* operation down samples the residual error to a coarser grid before the *prolongation* operator interpolates a correction computed on a coarser grid into a finer grid. In essence, the cycle repeatedly projects the current residual from a finer grid onto the next coarser grid and interpolates the solution from the coarser grid onto the finer one.

2.2 Symmetric Gauss-Seidel

SYMGS can be used as a *smoother* and a *solver* in a MG implementation like Algorithm 1. For example, in HPCG, SYMGS is used as the pre-smoother, post-smoother and bottom solver, where pre-smoothing and post-smoothing are applied once on each level of the grid (i.e., m_1 and m_2 are set to one in Algorithm 1). In other implementations, smoothing operations can be performed multiple times [5].

A SYMGS iteration consists of a forward and a backward *sweep* [52], which can be respectively defined as:

$$forward: \quad x_1 = (D+L)^{-1}(b - Ux_0), \tag{1}$$

$$backward: \quad x_2 = (D+U)^{-1}(b - Lx_1), \tag{2}$$

where the backward sweep takes as input the intermediate vector (x_1) produced by the forward sweep. Here, D is the diagonal entries of A, and L and U are the lower and upper triangular parts of A, respectively. Note that equations (1) and (2) contain a sparse triangular matrix-vector multiplication (**SPTRMV**) operation Ux_0 or Lx_1, and a sparse triangular linear solving (**SPTRSV**) operation, $(D+L)^{-1}\hat{b}$ or $(D+U)^{-1}\hat{b}$, where $\hat{b}$ is a vector. When substituting the output of the forward sweep, x_1, into the backward sweep, we have

$$x_2 = x_0 + G(b - Ax_0), \tag{3}$$

where $G = (D+U)^{-1}D(D+L)^{-1}$ is symmetric; and hence the name of symmetric GS.

2.3 Overhead of SYMGS

In an attempt to quantify the computation overhead of SYMGS in MG, we profile the sequential version of HPCG on the ARMv8-based Phytium 2000+ CPU. This benchmark solves a regular 27-point stencil discretization in three dimensions of an elliptic partial differential equation. HPCG uses a preconditioned conjugate gradient (PCG) algorithm [22, 24] implemented through a four-level V-Cycle MG method.

Our profiling results show that SYMGS accounts for 78% of the whole program execution time of HPCG, while SPMV has the second-largest overhead, contributing to 20% of the HPCG execution time. Note that we also observe a similar execution time distribution on other hardware platforms used in this paper. Moreover, independent studies have shown that SYMGS and SPMV operations are memory-bounded in HPCG, each has a low computation-to-memory ratio of 0.152 and 0.156 flops/byte, respectively [63]. Such a low arithmetic intensity further highlights the need of memory-aware optimization for MG.

3 COMPUTATION OPTIMIZATION

Our approach has two strands. The first is to devise the SYMGS and SPMV computation patterns in MG on a single core. Our optimization stores matrix A as sub-matrices to enable a new computation formula and kernel fusion to reduce computation overhead. The second is to exploit pipeline parallelism to reduce parallel thread starving running on multiple processor cores. We describe our computation optimization in this section and our parallelization optimization in Section 4.

3.1 SYMGS Kernel Optimization

Recall that the forward and backward sweep of SYMGS performs an SPTRMV and an SPTRSV operation (Section 2.2) on D, L, and U of matrix A. We split matrix A into submatrices D, L and U and use the CSR format to store the submatrices. Since D is diagonal, it can be stored as one vector. Our current implementation uses CSR, but other sparse matrix formats can be used too. For example, the matrices L and U in [63] are stored in SELL format, while the whole matrix A is stored in CSR format in [32]. We note that the matrix format conversation only needs to perform once, and hence the overhead of conversion is negligible. This storage format allows the SPTRMV and SPTRSV to be performed by accessing only one submatrix L or U instead of the whole matrix A. Therefore, the memory bandwidth cost is reduced and the data locality is also improved.

Since SPTRMV and SPTRSV can now be decoupled, we first execute the SPTRMV, $p_0 = -Ux_0$, in the forward sweep, which is then followed by SPTRSV. As a result, the forward sweep can be formulated as:

$$x_1 = (L+D)^{-1}(b + p_0). \tag{4}$$

As the backward sweep takes as input the output (x_1) of the forward sweep, we can rewrite the backward step of equation (2) as:

$$\begin{aligned} x_2 &= (D+U)^{-1}(b - Lx_1) \\ &= (D+U)^{-1}[b + Dx_1 - (D+L)x_1] \\ &= (D+U)^{-1}(Dx_1 - p_0). \end{aligned} \tag{5}$$

Now let $p_1 = Dx_1 - p_0$, the backward sweep in equation (5) becomes:

$$x_2 = (D+U)^{-1}p_1. \tag{6}$$

Comparing the original equation (2) of the backward sweep with our new formula in equation (6), we can see that our formula saves one SPTRMV operation because we do not compute Lx_1 in equation

Algorithm 2: Fusion of kernels SYMGS and SPMV

```
1 function Fusion(A, b, x_0)
2     Compute p_0 := -U x_0
3     Compute x_1 := (D + L)^{-1}(b + p_0)        ▷ forward step
4     Compute p_1 := D x_1 - p_0
5     Compute x_2 := (D + U)^{-1} p_1             ▷ backward step
6                                                 ▷ end of SYMGS-opt
7     Compute y := p_1 + L x_2                    ▷ end of SPMV
8     return x_2 and y
9 end
```

Algorithm 3: Pesudocodes for computing x_1 and p_1.

```
   Input: w := p_0                                ▷ do p_0 = -U x_0 in parallel.
   Output: x := x_1, w := p_1
1  for i = 0 to n do
2      double temp = b[i]
3      for j = L_p[i] to L_p[i + 1] - 1 do
4          temp -= L_v[j] * x[L_c[j]]
5      end
6      x[i] = (temp + w[i])/D[i]
7      w[i] = temp
8  end
```

(6). This formula permits us to combine equations (4) and (6) to build an optimized SYMGS, for which we call `SYMGS-opt`.

3.2 Computational Fusion

In our implementation, the computations of x_1 and p_1 at lines 3 and 4 of Algorithm 2 are fused to further reduce the memory access operations. Consider $x_1 = (L + D)^{-1}(b + p_0)$. Then for the element $x_1[i]$ in row i, there is

$$x_1[i] = \frac{1}{a_i[i]}(b[i] + p_0[i] - \sum_{j=0}^{i-1} a_i[j]x_1[j]), \tag{7}$$

and $p_1 = Dx_1 - p_0$, so p_1 can be reduced to

$$\begin{aligned} p_1[i] &= a_i[i]x_1[i] - p_0[i] \\ &= b[i] - \sum_{j=0}^{i-1} a_i[j]x_1[j]. \end{aligned} \tag{8}$$

Since p_1 is stored in p_0 inplace, only one auxiliary vector is needed, so the memory footprint overhead is small.

The pseudocode in Algorithm 3 describes how to compute the forward sweep, x_1 and p_1 used in Algorithm 2, assuming CSR is used. Here, L_p is the row pointer of L in the CSR form, L_v stores the nonzero values and L_c stores the column indexes of L in the CSR form.

3.3 Kernel Fusion

Kernel fusion is shown to be effective in optimizing MG [34, 44, 63]. Our computation formula also supports kernel fusion to reduce computation and memory access latency. In line with the standard implementation of MG, `SYMGS-opt` is followed by one SPMV of computing Ax_2 (line 6 of Algorithm 1). From the equation (6) we have:

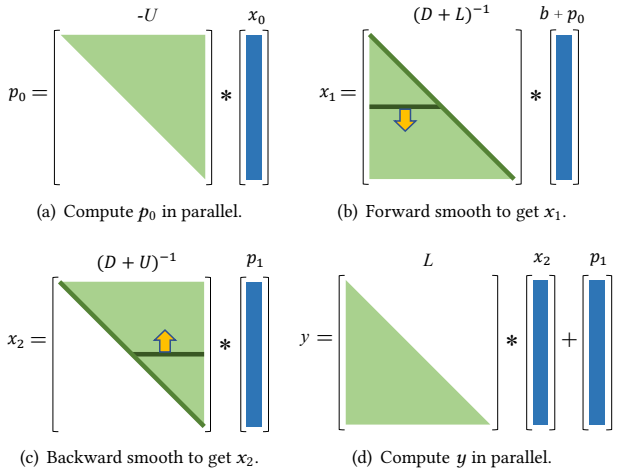
$$p_1 = (D + U)x_2. \tag{9}$$

Considering $A = L + D + U$, we can obtain Ax_2 by

$$Ax_2 = p_1 + Lx_2. \tag{10}$$

The main idea is that the intermediate vector $p_1 = (D + U)x_2$ has already been computed in `SYMGS-opt` during the backward step. This fusion merges `SYMGS-opt` with SpMV, and saves the computation and memory bandwidth costs.

Algorithm 2 outlines the computation process of our new SYMGS plus SpMV implementation (denoted as `Fusion`). Thus, by reusing the result of p_1 for computing Ax_2 in line 7 of Algorithm 2, our scheme saves computation and memory accesses for matrix A.



(a) Compute p_0 in parallel. (b) Forward smooth to get x_1.

(c) Backward smooth to get x_2. (d) Compute y in parallel.

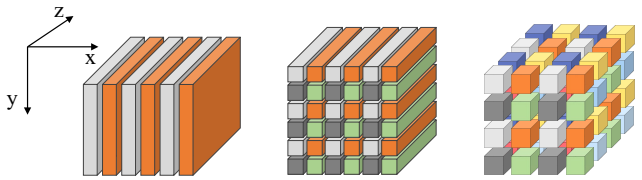
Figure 1: The computation flow of our `Fusion` implementation.

3.4 Efficiency Analysis

Figure 1 shows the workflow of our new `Fusion` implementation. We compare our optimizaiton with the classical SYMGS plus SPMV (one SYMGS followed by one SPMV) implementation where SYMGS is described in Section 2.2. The classical SYMGS implementation reads matrix A twice and costs $O(4nnz)$ flops, where nnz is the number of non-zeros of A, but it requires an additional SPMV operation to read matrix A once and costs $O(2nnz)$ flops, which leads to *three* reads to matrix A and $O(6nnz)$ flops. By contrast, our `Fusion` implementation (described in Algorithm 2) only needs to load matrices L and U from the main memory to the cache twice. This leads to *two* reads of matrix A. The computational complexity of our scheme is $O(4nnz)$ flops. As a result, our new implementation incurs only two-thirds of the memory accesses of the classical SYMGS plus SPMV implementation. Furthermore, if the initial vector x_0 is zero, the computation of $-Ux_0$ is not needed, and the computation cost of Algorithm 2 is reduced to $O(3nnz)$ flops.

4 PARALLELIZATION OF SYMGS

Our computation optimization can be integrated with existing parallelization schemes for SYMGS. This can be achieved by executing



(a) Cx=2, Cy=1, Cz=1 (b) Cx=2, Cy=2, Cz=1 (c) Cx=2, Cy=2, Cz=2

Figure 2: Block division with a different number of colors in each direction.

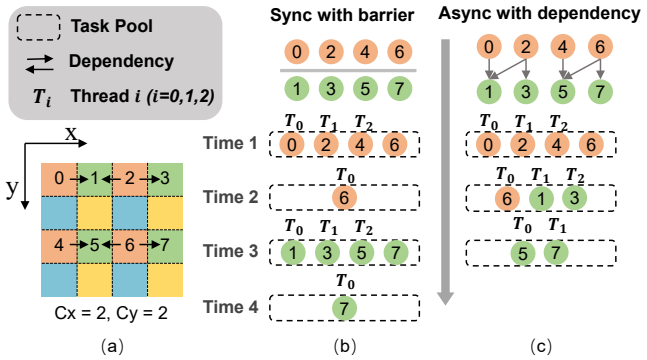


Figure 3: Using BMC to parallelize SYMGS algorithm. Figure(a) represents the dependencies between different color blocks. Figure(b) shows the commonly used synchronous method, all threads are synchronized before executing blocks of a different color. Figure(c) shows our proposed asynchronous scheme, the execution time is reduced.

the SPTRSV operation (see Section 2.2) across concurrently running threads, using the block multi-color reordering (BMC) [27, 28] parallelization scheme.

4.1 Block Multi-color Reordering

The idea of BMC is to group computation tasks into color blocks, where tasks with the same color have no dependence and can be executed in parallel. Although it breaks some of the original dependence relations and leads to slow convergence [16, 46], it introduces more parallelism in SYMGS. There are three different ways of applying BMC to a 3-dimensional geometric grid: flake block (1D), strip block (2D) and cube block (3D). These block partitioning schemes can be described by using three parameters, C_x, C_y and C_z, which respectively represent the number of colors in the x, y and z directions, as shown in Figure 2. For example, $C_x = 2, C_y = 1$ and $C_z = 1$ represents the 1D partition in Figure 2(a), where the y and z directions have no parallelism and all the flake blocks are along the x-dimension. We want to choose the best partition scheme to strike a balance between parallelism and convergence rate.

As can be seen from Figure 2, using a small number of colors (i.e., c is small) is unlikely to expose high parallelism due to the limited grid side length and unitary partition of blocks. On the other hand, using too many colors can lead to poor data locality and frequent thread synchronization. Figure 6 shows that there is no "one-size-fits-all" number of colors. To facilitate the subsequent analysis, we empirically set c as 8 (i.e., $c_x = c_y = c_z = 2$) on all platforms, which gives an overall good performance.

4.2 Adaptive Block Size Selection

BMC employs a block size parameter, denoted by $b = b_x b_y b_z$, which determines the amount of work assigned to a given number of threads that can simultaneously perform computation tasks of the same color. As the block size increases, the parallelism decreases, but the data locality and convergence rate improve. Conversely, if the block size decreases, parallelism increases, but data locality and convergence rate suffer. Here, our goal is to establish a metric that will enable us to select the appropriate block size b_i, ensuring a good convergence rate without compromising load balancing. For a typical MG implementation like the geometric multigrid (GMG) algorithm, the block size can be determined based on the geometric information of grids by considering computation balance. Specifically, for a 3-dimensional problem domain with x, y, and z directions, the *average* number of blocks per color (denoted by n_{per_color}) is

$$n_{per_color} = \frac{N_x}{c_x b_x} \times \frac{N_y}{c_y b_y} \times \frac{N_z}{c_z b_z}, \tag{11}$$

where $c = c_x \cdot c_y \cdot c_z$ is the total number of colors across three dimensions, and b_i and N_i represent the block size and grid side length of dimension i respectively. To ensure load balance, we use the following formula to choose the block size ($b = b_x b_y b_z$) for BMC:

$$n_t \cdot \beta = n_{per_color}, \tag{12}$$

where n_t is the number of parallel threads, and β is the *average* number of blocks to be processed by each thread, and $\beta \geq 1$. Our key insight is to let b_i as large as possible to enhance convergence rate without compromising load balancing, i.e., we hope that β is near one. In Section 6.2, we empirically show that our adaptive strategy for choosing the block size gives a good performance.

To improve data locality, we reorder the blocks based on their colors and store blocks of the same color consecutively in the memory space. Furthermore, the mapping of SYMGS to the 3D grid is straightforward, as each row of the matrix corresponds to a point on the grid. Specifically, SYMGS accesses rows in the same order as we scan over the 3D grid after parallelization.

We stress that the block size can dynamically change when moving to a coarser grid across MG iterations. The grid size of coarser level is almost an eighth of the finer level in MG. The block size decreases proportionally as the grid scale decreases, that is, the number of blocks at each level is kept the same.

4.3 Asynchronous Parallelization

In equation (12), β equals one under perfect load balance. However, n_{per_color} may not be divisible by n_t, and b_i is determined by letting β as near one as possible in this case. Since threads work on a whole block, load imbalance occurs when β is not an integer, which results in performance degradation. To solve this problem, we propose an asynchronous mode for BMC.

The vast majority of BMC implementations compute blocks of the same color in parallel by introducing a barrier among parallel threads before switching to computation on blocks of a different color. Figure 3(a) depicts the dependencies of the two color blocks, and in Figure 3(b), this synchronous scheme introduces a barrier for the parallel computations on blocks 0, 2, 4, and 6 before moving to blocks 1, 3, 5, 7. A problem with this synchronous parallelization scheme is that depending on the amount of work within a block, there is no guarantee that all the parallel threads can finish at the same time; as a result, some of the parallel threads may need to wait for the slowest peer before moving to compute the blocks with a different color. Furthermore, it is possible that the number of remaining blocks is smaller than the number of parallel threads. In this scenario, some processor cores and threads have to be idle in waiting for the computation of some blocks. One such example is given in Figure 3(b), where threads T_1 and T_2 have to wait T_0 at time 2. These issues lead to the under-utilization of computation resources.

To improve the parallelism for coloring-based parallelization, we propose to execute blocks of different colors asynchronously. As shown in Figures 3(a) and 3(b), BMC-based (synchronous) SYMGS executes blocks of different colors according to the direction order of x, y, and z, and each block is dependent on the blocks of the previously executed colors with a distance of one in each of the three directions. As can be seen from Figure 3(c), our asynchronous scheme allows a parallel thread to execute as long as its dependent blocks get executed, rather than waiting for the completion of all blocks of the previous color. Our asynchronous scheme reduces the stalling time between parallel execution and improve the utilization of processor cores, leading to faster execution time.

This asynchronous mode of BMC can effectively alleviate the load imbalance caused when β is not a whole number in equation (12). This is particularly important when the block partition method is used to solve linear equations with unstructured matrices where the number of blocks with different colors may vary greatly and the problem of unbalanced load is more prominent.

We use the tasking mode introduced by OpenMP 3.0 to implement our asynchronous parallelization scheme. Later in Sections 6.3 and 7.3, we show that our asynchronous approach can greatly improve the performance of BMC-based parallel SYMGS.

5 EXPERIMENTAL SETUP

5.1 Evaluation Platforms

To demonstrate the portability of our optimization strategies, we test our approach on both ARM and x86 platforms. Table 1 lists the hardware platforms used in the paper, including three ARMv8 architectures [18, 58] and an Intel Xeon Gold-5117 CPU. We stress that our work focuses on performance optimization of a single computing node, and we also test our approach on a Phytium 2000+ cluster using 256 computing nodes.

Our evaluation platforms run Linux kernel version 4.19.46, and we use GCC version 11.2.0 with the "-O3" compiler option and MPICH version 3.4.3.

Table 1: Hardware platforms used in evaluation.

	Phytium 2000+	KP 920	Thunder X2	Xeon
#Cores	64	64	32	14
Sockets	1	1	1	2
#NUMAs	8	2	1	2
CPU Freq.	2.2GHz	2.6GHz	2.5GHz	2.0GHz
L1 cache	32KB	64KB	32KB	896KB
L2 cache	2MB	512KB	256KB	28MB
L3 cache	None	64MB	32MB	38.5MB

5.2 Application Workloads

We ported our techniques to the official release of HPCG (version 3.1). This version does not use architecture-dependent optimizations like our competing baselines (Section 5.3). HPCG runs using multiple MPI processes where each process is further parallelized using OpenMP threads on a shared-memory machine[2]. We also port our techniques to realistic linear solver applications: YHAMG [17], CitcomCU [53], linear solver of unstructured matrices. All the programs are written in C++.

We use representative problem sizes in our evaluation. HPCG implements a four-level MG, and the scale of each level decreases by a factor of eight. The test grid size of HPCG is $192 \times 192 \times 192$. We apply CitcomCU to solve a real application problem with a local grid scale of $256 \times 256 \times 128$ per MPI process, and use YHAMG to solve the Poisson problem discretized by a 7-point and a 27-point stencil, respectively.

5.3 Competing Baselines

HPCG for ARM. We compared our HPCG implementation against the ARM-architecture-tuned HPCG developed by ARM [51, 54]. We denote this version as *HPCG_for_ARM*. This heavily-tuned implementation uses the ARM Neon single instruction multiple data (SIMD) instruction for matrix multiplications. It exploits level scheduling parallelization [43] at the finest grid level and uses BMC to parallelize the remaining coarser levels of grids. We also use the ARM Performance Libraries (ARMPL) (version 21.0) [4] to accelerate the SPMV kernel of this implementation for a fair comparison.

HPCG for x86. On the Intel platform, we compared our HPCG implementation against the optimized HPCG version (close source) provided by the Intel MKL library (release date January 2022) [47]. We denote this version as *HPCG_for_MKL*.

YHAMG and CitcomCU. Both applications have already been heavily optimized by their developers. Specifically, CitcomCU uses a optimized F-Cycle MG, and YHAMG uses hand-optimized SYMGS and SPMV kernels. We show that our techniques can be applied to the two applications to further improve their performance.

Synchronous BMC. We compare our asynchronous parallelization scheme to the synchronous BMC parallelization described in Section 4.3 on the parallel version of of the SYMGS kernel in HPCG, which runs on multi-core CPUs using OpenMP.

[2]OpenMP is used to parallelized SPMV and other matrix multiplication operations but SYGMS is implemented as a sequential kernel in the official release of HPCG.

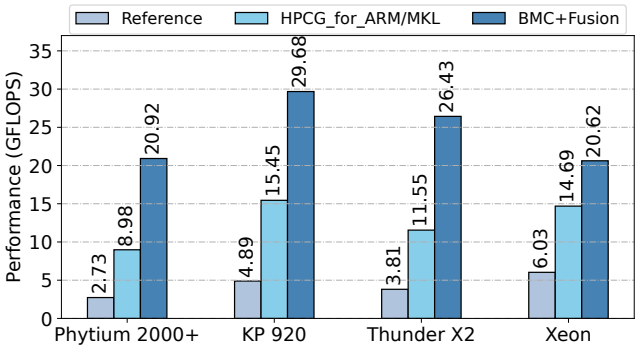


Figure 4: Comparison of our optimized HPCG (`BMC+Fusion`) and the hardware-specific implementations on a single node.

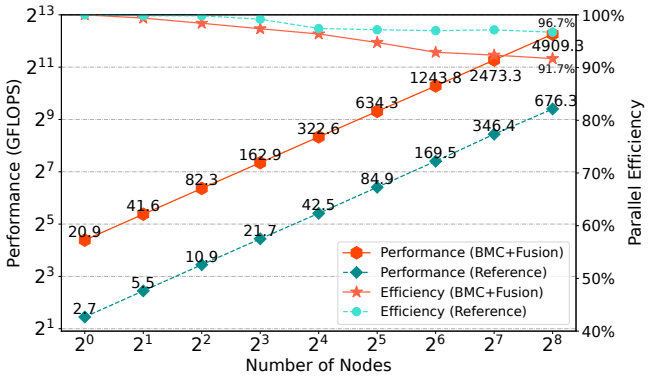


Figure 5: Performance and parallel efficiency of HPCG (with `BMC+Fusion` and Reference) in a Phytium 2000+ computing cluster with 256 nodes.

5.4 Evaluation Methodology

In our experiments, we evaluate the performance of our SYMGS kernel optimization (`SYMGS-opt`) described in Section 3.1 and our kernel fusion techniques (`Fusion`) described in Section 3.3, as well as their integration with BMC described in Section 4. We also evaluate our adaptive blocking strategy described in Section 4.2.

Unless stated otherwise, we report the *end-to-end* benchmark performance. We run each test case 10 times on unloaded machines and report the geometric mean of the runtime. The variance across different runs is small, less than 5%. We run the benchmarks with 8 MPI processes on ARM and 4 MPI processes on x86 (to match the 28 CPU cores), with 7 OpenMP threads per MPI process. For further analysis of other strategies, we use 8 and 32 OpenMP threads for each MPI process on ARM systems (28 on x86).

6 EXPERIMENTAL RESULTS

6.1 HPCG Performance

Single-node performance. Figure 4 reports the HPCG performance (measured as GFLOPS) of our optimizations (`BMC+Fusion`) and the competing baselines across our evaluation platforms. Here, we use the official release of HPCG as the `Reference`. The experiments were performed on a single computing node using all the physical processor cores. The vendor-tuned HPCG implementations, *HPCG_for_ARM* and *HPCG_for_MKL*, take advantage of the high-performance back-end libraries and hardware-specific optimization, delivering significant performance benefits over the reference implementation. Although our implementations are based on the platform-agnostic reference implementation, it outperforms the vendor-tuned versions on their targeting platforms by a large margin. Specifically, our techniques improve *HPCG_for_ARM* by 1.92x–2.33x and *HPCG_for_MKL* on x86 by 1.40x. The *HPCG_for_ARM* implementation uses BMC for parallelization together with ARM-specific SIMD instructions to accelerate SPMV computation. Our BMC implementation builds upon the official release and does not use ARM-specific instructions. Yet, our implementation gives significantly better performance.

Table 2: Breakdown of individual optimizations on HPCG.

Method	Phytium2000+	KP920	ThunderX2	Xeon
Reference	1.00x	1.00x	1.00x	1.00x
`SYMGS-opt`	2.73x	2.06x	1.82x	1.56x
`Fusion`	3.08x	2.21x	2.03x	1.80x
BMC	3.98x	3.29x	3.84x	1.98x
BMC+`SYMGS-opt`	5.71x	4.62x	4.88x	2.56x
HPCG_for_ARM/MKL	3.29x	3.16x	3.03x	2.44x
`BMC+Fusion`	**7.66x**	**6.07x**	**6.94x**	**3.42x**

Multi-node performance scalability. Figure 5 shows the performance when running our HPCG implementation with BMC parallelization (`BMC+Fusion`) in a 256-node Phytium 2000+ cluster using a total of 16,384 cores. As can be seen from the diagram, our implementation exhibits a good scalability with nearly linear improvement on GFLOPS as the number of computing nodes used increases. Specifically, when using 256 nodes, our implement delivers 4909.3 GFLOPS. The performance of our implementation translates to a 91.7% parallel efficiency[3]. Note that as we reduce the computation time in the optimized version, the communication overhead further limits parallel efficiency, which leads to a lower parallel efficiency for the optimized version. However, our approach still gives a higher throughput measured by GFLOPS compared to the reference implementation.

Performance breakdown. Table 2 gives a breakdown of individual optimizations on HPCG, using the official implementation of HPCG as the reference baseline (`Reference`) where SYGMS is implemented as a sequential kernel. Even without paralleling SYMGS, our kernel level optimizations, `SYGMS-opt` and `Fusion`, are able to give 1.56x to 3.08x improvement over the reference implementation. Our full optimization, `BMC+Fusion` gives the highest speedup across evaluation platforms, showing that our optimization can be useful in optimizing MG.

[3]The parallel efficiency is computed as T_n/nT_1. Here, T_1 is the measured throughput (GFLOPS) on a single node, and T_n is the measured throughput when using n computing nodes. Essentially, this higher-is-better metric quantifies how much of the theoretically perfect improvement (projected linearly from a single node to n nodes) has been realized.

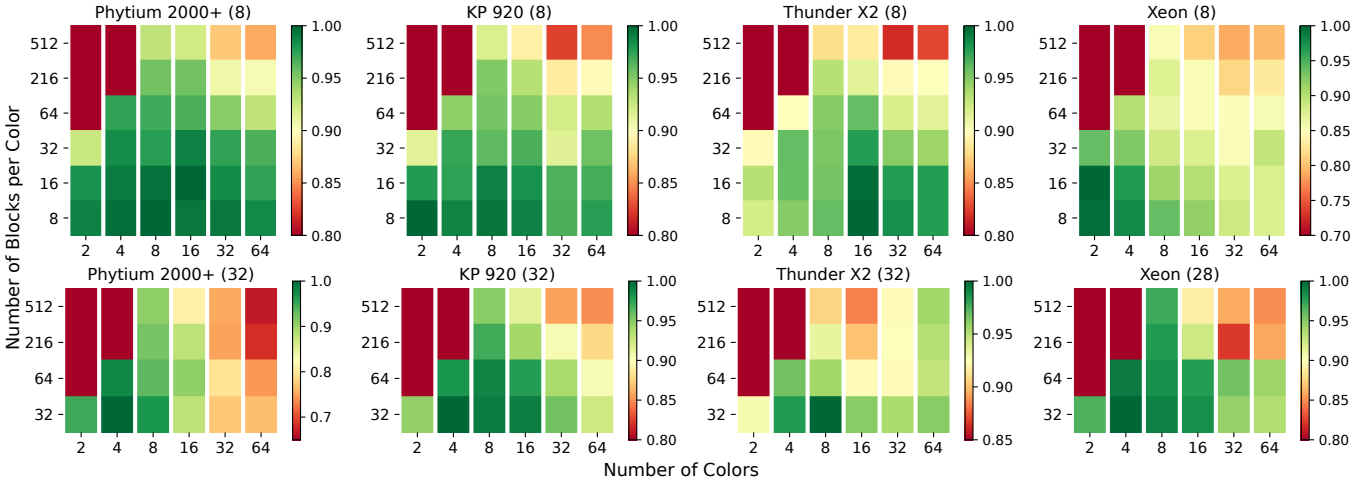


Figure 6: Impact of the BMC color and block counts on the performance of HPCG using 8 and 32 (26 threads on Xeon) parallel OpenMP threads. The numbers in the heat map are normalized to the optimal performance (1.0) on each platform.

6.2 Adaptive BMC Parallelization

We have described the strategy for adaptively selecting BMC block sizes in Section 4.2. Our strategy is based on an observation that the number of colors c and the block size b together determine the parallelism and work distribution across parallel threads. This experiment provides quantified data to justify our design choices. Since there are many possible choices for b in all three directions, we set b to $\{8, 16, 32, 64, 216, 512\}$ and c to $\{2, 4, 8, 16, 32, 64\}$, and perform a greedy search to sample the parameter space. To expand the selection of parameters, we apply BMC to the first two fine-grained grids. When the parameters are not evenly divided in all three directions, the y-axis direction is usually set with more colors or blocks because we find that it gives better performance.

Figure 6 shows the resulting HPCG performance when changing the number of blocks per color (i.e., the number of blocks can run in parallel). Here, we normalize the performance to the optimal performance found on each platform (i.e., 1.0 is the optimal performance). It can be observed that in almost all cases, the optimal performance can be achieved when the number of blocks is close to the number of threads, i.e., β approaches 1 in equation (12). The optimal numbers of colors c can vary across platforms, but using 4, 8 or 16 colors gives an overall good performance. Based on this observation, we set the number of colors as 8 and $c_i = 2$ in the three directions in subsequent experiments, and select the number of blocks according to equation (11).

We also compare our adaptive scheme with common fixed block size schemes for all grid levels of MG. For example, our scenario iterates 51 times when reaching the same residual as the reference, and 56 times when using the static block size of 64 [44]. HPCG achieved a throughput of 20.92 GFLOPS on Phytium 2000+ using the former and only 16.23 GFLOPS using the latter.

6.3 Asynchronous BMC

In this experiment, we compare our proposed asynchronous BMC with the standard synchronous implementation in the SYMGS phase of HPCG. Our experiment considers the scenarios where the number of blocks can not be evenly distributed across parallel threads.

Figure 7 shows the performance of our asynchronous mode and the standard synchronous mode on our evaluation platforms under load imbalance (OMP=7) and balancing (OMP=8) scenarios

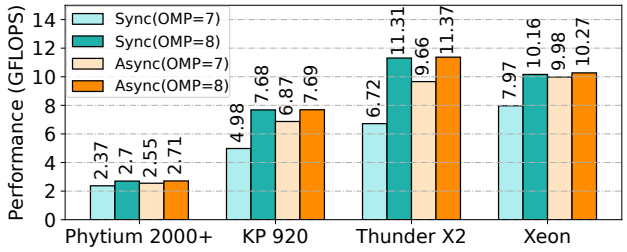
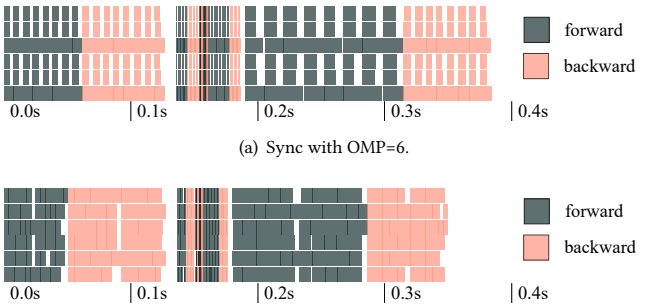


Figure 7: Performance comparison between the asynchronous and the synchronous mode in load imbalance (OMP=7) and load balancing (OMP=8) scenarios.



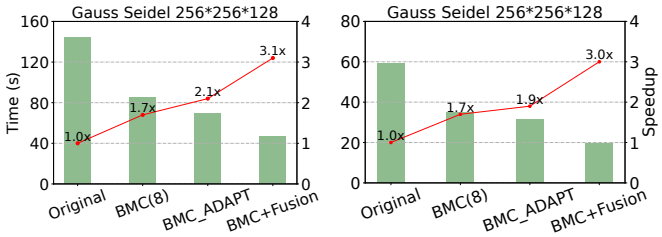
(a) Sync with OMP=6.

(b) Async with OMP=6.

Figure 8: Thread computation (solid blocks) and idle (blank gap) between thread invocations when computing the SYMGS kernel in the synchronous (a) and the asynchronous (b) mode using six parallel threads to process eight blocks of a color.

Table 3: Runtime measurement (in seconds) for different stages of YHAMG.

	Phytium 2000+				Intel Xeon			
YHAMG	7-point		27-point		7-point		27-point	
	Setup	**Solve**	**Setup**	**Solve**	**Setup**	**Solve**	**Setup**	**Solve**
Native	3.35	4.67	3.83	7.32	1.86	3.15	2.40	5.56
+SYMGS-opt	3.43	3.82	4.16	5.73	1.90	2.75	2.49	4.41
+Fusion	3.43	3.44	4.16	5.17	1.90	2.53	2.49	4.12



(a) *Phytium 2000+* (b) *Intel Xeon*

Figure 9: Speedup of CitcomCU on Phytium 2000+ and Intel Xeon platforms. Our leads to good speedup.

(with eight blocks per color) when using a different number of OpenMP threads. Our asynchronous parallelization scheme gives the best overall performance, whose advantage is more evident in the imbalanced mode.

To closely examine the benefit of our asynchronous scheme, consider now Figure 8. This diagram shows the computation and idle time of parallel threads measured on our Intel Xeon platform using six OpenMP threads for executing eight blocks of the same color. We can see that there are some gaps between thread invocations as the number of parallel threads used is smaller than the number of blocks that can run concurrently. However, we observe more frequent thread idle with longer idle time at the synchronous execution mode than in our asynchronous mode. This is because our asynchronous allows parallel threads to start working on a block of a different color as long as its dependence has been resolved, which in turn reduces the thread idle time. Since our approach improves thread utilization, it leads to a faster execution time compared to synchronous parallelization in this experiment.

7 APPLICATIONS

7.1 Algebraic Multigrid

In this evaluation, we integrate our `SYMGS-opt` and `Fusion` techniques to the algebraic multigrid (AMG) kernel [37, 50] in the YHAMG library [17]. We set the local grid scale to 128×128×128 for an MPI process. The experiments were performed on the Phytium 2000+ and Xeon systems using a single node. Note that the BMC parallelization technique is not used in this evaluation and we mainly measure the benefit of Algorithm 2.

Table 3 shows the performance results, where *Setup* and *Solve* denote the wall times (in seconds) for the setup and solve stage of AMG [55], respectively. On Phytium 2000+, we observe a modest processing overhead of 2.33%∼8.62% during the setup stage. This is due to the preprocessing overhead of converting the input matrix to our matrix storage format. However, this overhead is amortized at the later, most time-consuming solving stage (which often iterates many times in a typical MG-based solver), where our `Fusion` version improves the native YHAMG implementation by 35.8%∼41.6% per solver iteration. We also see a similar performance trend on Xeon. Like Phytium 2000+, although our approach introduces minor overhead with a 2.1%–3.7% increase in the setup processing time on Xeon, the cost is paid off by the 24.5%–34.9% performance improvement at the solver stage. For large-scale problems, where the solver dominates the execution time, our optimization techniques can give a clear boost to the application's performance.

7.2 CitcomCU

The native implementation of CitcomCU does not parallelize SYGMS. We upgrade it by using BMC to parallelize its SYMGS kernel, which accounts for 70% of the whole program execution time. We then evaluate two variants of our techniques, `BMC_ADAPT` that applies our block size selection scheme (Section 4.2) and `BMC+Fusion` that implements our full optimization.

Figure 9 shows the execution time (in seconds) and the speedup relative to the native implementation (`original`) of CitcomCU on Phytium 2000+ and Intel Xeon. Note that we also observed similar performance improvement on KP 920 and Thunder X2 as Phytium 2000+. Here, *BMC(8)* uses a block size $b = 8 \times 8 \times 8$ in all grid levels. Our block size choosing scheme, `BMC_ADAPT`, gives a 2x speedup on both the ARM and Xeon platforms, improving `BMC` by 30%. With our full optimization, `BMC+Fusion` gives the best performance, improving the native implementation and `BMC` by at least 3x and 1.87x respectively. The results show the effectiveness of our techniques on real programs.

7.3 Unstructured Matrices

Our computation and parallelization optimization methods can be applied to linear equation solvers for unstructured matrices. We apply our optimization to a set of representative unstructured matrices from SuiteSparse [12] given in Table 4. The main purpose of this evaluation is to show the benefits of asynchronous BMC strategy when comparing with synchronous mode.

In this experiment, we use ABMC [27] to parallelize SYMGS and measure the wall times when performing three times of SYMGS. We first use METIS (version 5.1.0) [29] to partition the matrices into blocks and then use Colpack [21] to map blocks into colors, where blocks of the same color can be executed in parallel. We note that most matrices show good performance using colors less than 10. So according to our block size choosing strategy in Section 4.2 (i.e., the number of blocks per color is close to the number of threads),

Table 4: Input matrices used in our evaluation.

ID	Input	Rows	#nnz	#nnz/rows
A	af_shell10	1,508K	52.67M	35
B	audikw_1	944K	77.65M	82
C	cage14*	1,506K	27.13M	18
D	cant	62K	4.01M	64
E	crankseg_1	53K	10.61M	201
F	ecology2	1,000K	5.00M	5
G	Flan_1565	1,565K	117.41M	75
H	G3_circuit	1,585K	7.66M	5
I	hollywood-2009	1,140K	113.89M	100
J	Hook_1498	1,498K	60.92M	41
K	inline_1	504K	36.82M	73
L	ldoor	952K	46.52M	49
M	ML_Geer*	1,504K	110.88M	74
N	pwtk	218K	11.63M	53
O	Serena	1,391K	64.53M	46
P	ship_003	122K	8.09M	66
Q	shipsec1	141K	7.81M	55
R	tmt_sym	727K	5.08M	7

*These two matrices are unsymmetric. Others are symmetric.

the number of blocks is set between 100 and 300 as it gives a good trade-off between parallelism and processing overhead.

Figure 10 shows the speedup when using the asynchronous mode versus the synchronous mode for each matrix across all platforms. We observe that our asynchronous parallelization scheme gives better performance over the standard synchronous mode on most of the matrices across platforms for a different number of blocks. Specifically, our approach improves the synchronous mode by 5% to 14% across evaluation platforms. On some small matrices like *cant*, our asynchronous scheduling approach does not give a clear advantage and can have a modest slowdown because there is not enough computation to amortize the overhead of dynamic scheduling. For some matrices, like *hollywood-2009*, it requires to use a large number of colors, and leads to low parallelism, which affects the performance benefit of asynchronous scheduling strategy. Nonetheless, our asynchronous scheduling approach leads to better performance over the synchronous mode for most of the test cases.

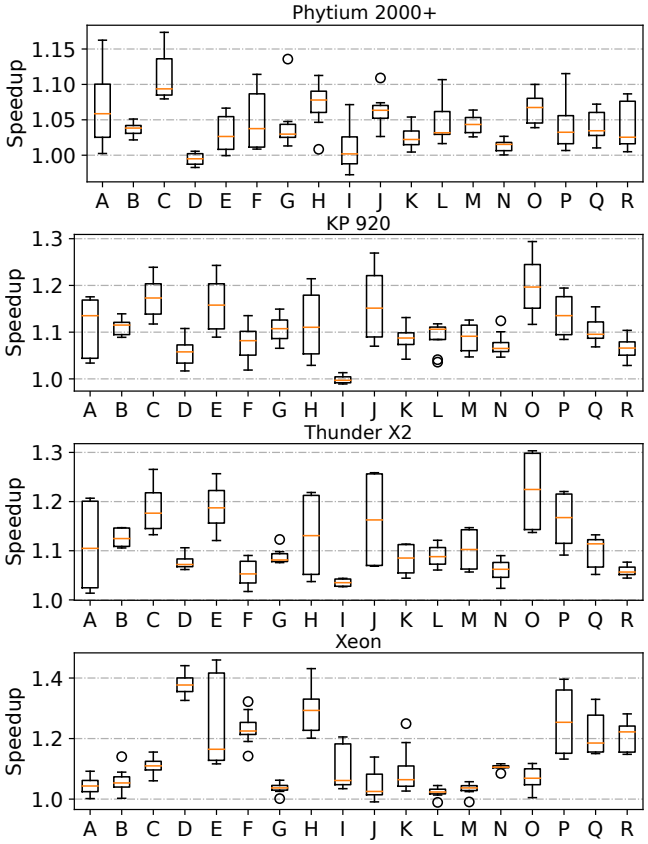


Figure 10: The speedup of asynchronous mode compared with synchronous mode when performing three times of SYMGS with different number of block partitions. A to R represent the serial numbers of matrices shown in Table 4.

8 DISCUSSIONS

Naturally, there is room for improvement and further work. We discuss a few points here.

Optimizing library. One of our ongoing works is to implement our techniques as an optimizing library to provide an API to optimize SYMGS and SPMV kernels in MG.

More fusions. We have shown that fusing the presmoother (SYMGS) with SPMV can greatly improve the performance of HPCG. We envision the same techniques can be applied to other MG operators too. For example, we can fuse SPMV with a restriction operator or merge the postsmoother (SYMGS) with the following SPMV outside MG to further improve the performance.

Sparse matrix storage formats. Our current implementation splits matrix A to submatrices L and U, which are then stored in the CSR format. While this strategy already gives significant improvement, we are aware that CSR may prevent vectorization. Our future work will exploit other storage formats like ELLPACK [30] and Sliced ELL [31], and investigate if a better, dedicated sparse matrix storage format can be designed for SYMGS and MG.

Distributed and NUMA optimization. This work focuses on performance optimization on shared-memory multi-cores. Since our approach does not introduce extra synchronization and communication overhead, a distributed technique [9] can directly benefit from the improved performance of a single CPU. Our approach currently does not model the NUMA impact. Therefore, NUMA-aware work distribution techniques [59] are complementary to our solution.

Heterogeneous devices. This work focuses on homogeneous multi-core CPUs because they remain the most widely used computing devices in HPC and MG. Our future work will look into extending our techniques to the GPU space. Given the expensive memory access overhead on GPUs, memory optimization would become even more important. It would also be interesting to see if our techniques can be integrated with optimization designed for GPU-based stencil computing [48] and linear solvers [41].

9 RELATED WORK

MG is often used as a preprocessor to accelerate the solution of large-scale linear problems. SYMGS is often used as the smoother of MG and many techniques have been proposed to parallelize it. Our work proposes a new way to optimize the SYMGS computation. We achieve this by partitioning the input matrix into submatrices around which we develop a new computation formula to reduce the computation cost. A closely related prior work for optimizing conjugate gradient computation [32] also breaks the forward and backward sweep of SYMGS to compute the upper and lower triangular parts, aiming to improve the locality of vector x. Unlike our approach that partitions and stores the lower and upper triangular parts of matrix A separately, [32] stores the entire matrix in the CSR format. As such, it does not reduce memory access to A nor save the computation. The work [63] stores matrices of L and U separately. However, it incurs more computation than our approach, as the SYMGS algorithm [63] requires one more SPTRMV than our `SYMGS-opt`. Our new SYMGS computation formula reduces the number of computation operations, which is more beneficial for computation-intensive kernels [19, 20, 39, 57].

The second contribution of our work is a new way to parallelize SYMGS to reduce the synchronization overhead. Generally speaking, there are two types of parallelization strategies: *level scheduling* [1, 43] and *color reordering* [3, 35, 63]. In this work, we exploit BMC to accelerate the SYMGS. The multi-color reordering method (MC) [46] has long been used to parallelize SYMGS, but its drawback is having poor data locality. Later BMC [28] alleviates the problem of poor locality by using blocks as shading units instead of points. There are three types of block partitioning methods from different dimensions: flake block [51], strip block [28] and cube block [3]. While most previous works consider increasing the number of colors in one or two fixed direction, we compare the effects of changing the number of colors in all three different directions. This block size is often set to a constant value like 64 [44], or a fixed very large block size (2625) is chosen in [32] via the autotuning technique, or it is determined as large as possible based on the number of physical processors available [3]. In fact, both the number of colors and the size of the blocks affect the rate of convergence and parallelism, which is embodied as iteration step. No one has yet come up with a strategy that will pick out these parameters and achieve the best performance in all cases. In this work, we further propose a strategy for choosing the block size adaptively, based on the grid scale and the number of OpenMP threads. Comparing with the previous methods which use a constant small block size for all grid levels, our adaptive strategy can improve the performance a lot. None of previous works combine level scheduling with color reordering. In this work, we apply the level scheduling technique to BMC, and an asynchronous BMC is proposed based on the dependencies of blocks of different colors. An algebraic block multicoloring (ABMC) method is proposed in [27], which is suitable for solving linear systems of equations with unstructured matrices. Our asynchronous BMC can also be applied to this block partitioning approach.

SPMV is another important kernel of MG. Kernel fusion is one of the commonly used techniques for optimizing MG. For example, a SYMGS-SPMV fusion technique is proposed in [44] for systems equipped with Intel Xeon Phi coprocessors. Different from it, we store submatrices L and U separately and one SPTRMV is saved in this work. SYMGS-SPMV is also used in the recent work [63]. Besides fusing SYMGS with SPMV, a red-black coloring along z-axis is proposed in [61] and a forward and backward SYMGS smooth fusion is tested on Tianhe-2 supercomputer. SPMV is naturally parallelizable and its optimizations mainly focus on choosing appropriate storage format [33], such as ELLPACK [30], Sliced ELL [31] or continuous CSR [23]. We use the CSR format and the other formats can be used together with our techniques.

10 CONCLUSION

We have presented a set of new techniques for optimizing multi-core multigrid (MG) algorithms, specifically focusing on improving the performance of the symmetric Gauss-Seidel (SYMGS) method, which is the most time-consuming kernel in MG. Our approach involves partitioning the sparse matrix into multiple storage units and using an optimized SYMGS to reduce computation and memory access overhead. To further reduce computational overhead, we merge SYMGS with the sparse matrix-vector (SPMV) multiplication kernels of MG. The resulting kernel is only two-thirds the computational cost of the classical SYMGS plus SPMV. For parallel executions, we utilize the block multi-color (BMC) method to parallelize SYMGS and propose an adaptive method for selecting the optimal block size. Moreover, we tackle the load imbalance issue among threads by combining BMC with level scheduling and propose an asynchronous BMC method.

Our techniques have been integrated into the HPCG benchmark that implements a representative MG algorithm and two real-life applications, YHAMG and CitcomCU. We also test our asynchronous BMC method on unstructured matrices from SuiteSparse [12]. Experimental results demonstrate that our approach outperforms the engineer-tuned implementation by a large margin on three ARMv8 and one x86 multi-core platforms.

ACKNOWLEDGMENTS

The authors would like to thank the referees for their valuable comments and Xiaoxiong Zhu for his helps in numerical experiments of CitcomCU. This work was supported in part by the National Key R&D Program of China under grant agreement 2021YFB0300101, the National Science Foundation of China (NSFC) under grant agreements 61902411, 62032023, 12002382, 11275269, and 42104078, and the Excellent Youth Foundation of Hunan Province under grant agreement 2021JJ10050. For the purpose of open access, the author has applied a Creative Commons Attribution (CC BY) licence to any Author Accepted Manuscript version arising from this submission. The data and code associated with this paper are openly available at https://github.com/YXJ-123/MGopt-APP.

REFERENCES

[1] Edward Anderson and Youcef Saad. 1989. Solving sparse triangular linear systems on parallel computers. *International Journal of High Speed Computing* 1, 01 (1989), 73–95.

[2] Hartwig Anzt, Edmond Chow, and Jack Dongarra. 2015. Iterative sparse triangular solves for preconditioning. In *Euro-Par 2015: Parallel Processing: 21st International Conference on Parallel and Distributed Computing, Vienna, Austria, August 24-28, 2015, Proceedings 21*. Springer, 650–661.

[3] Yulong Ao, Chao Yang, Fangfang Liu, Wanwang Yin, Lijuan Jiang, and Qiao Sun. 2018. Performance Optimization of the HPCG Benchmark on the Sunway

TaihuLight Supercomputer. *ACM Trans. Archit. Code Optim.* 15, 1, Article 11 (mar 2018), 20 pages.
[4] ARM. 2022. ARM performance libraries. https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries.
[5] Jamison Assunção and Victor Sacek. 2017. Benchmark comparison study for mantle thermal convection using the CitcomCU numerical code. In *15th International Congress of the Brazilian Geophysical Society & EXPOGEF, Rio de Janeiro, Brazil, 31 July-3 August 2017*. Brazilian Geophysical Society, 1630–1635.
[6] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Steven Benson, Jed Brown, Peter Brune, Kris Buschelman, Emil Constantinescu, Lisandro Dalcin, Alp Dener, Victor Eijkhout, William D. Gropp, Václav Hapla, Tobin Isaac, Pierre Jolivet, Dmitry Karpeev, Dinesh Kaushik, Matthew G. Knepley, Fande Kong, Scott Kruger, Dave A. May, Lois Curfman McInnes, Richard Tran Mills, Lawrence Mitchell, Todd Munson, Jose E. Roman, Karl Rupp, Patrick Sanan, Jason Sarich, Barry F. Smith, Stefano Zampini, Hong Zhang, Hong Zhang, and Junchao Zhang. 2022. *PETSc/TAO Users Manual*. Technical Report ANL-21/39 - Revision 3.17. Argonne National Laboratory.
[7] Grey Ballard, Erin Carson, James Demmel, Mark Hoemmen, Nicholas Knight, and Oded Schwartz. 2014. Communication lower bounds and optimal algorithms for numerical linear algebra. *Acta Numerica* 23 (2014), 1–155.
[8] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. 2011. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Anal. Appl.* 21, 2 (2011), 562–580.
[9] Amanda Bienz, William D Gropp, and Luke N Olson. 2020. Reducing communication in algebraic multigrid with multi-step node aware communication. *The International Journal of High Performance Computing Applications* 34, 5 (2020), 547–561.
[10] William L Briggs, Van Emden Henson, and Steve F McCormick. 2000. *A multigrid tutorial*. SIAM.
[11] Edmond Chow, Andrew J Cleary, and Robert D Falgout. 1999. Design of the hypre preconditioner library. *Object Oriented Methods for Inter-operable Scientific and Engineering Computing* (1999), 106–116.
[12] T. Davis and Y. Hu. 2011. The University of Florida Sparse Matrix Collection. *ACM Trans. Math. Softw.* 38, 1 (2011), 1:1–1:25.
[13] Denis Demidov. 2019. AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation. *Lobachevskii Journal of Mathematics* 40, 5 (2019), 535–546.
[14] J. Demmel. 1997. *Applied numerical linear algebra*. SIAM, Philadelphia.
[15] J. Dongarra and M. A. Heroux. 2013. *Toward a new metric for ranking high performance computing systems*. Technical Report SAND2013-4744. Sandia.
[16] Howard C Elman and Elvira Agrón. 1989. Ordering techniques for the preconditioned conjugate gradient method on parallel computers. *Computer Physics Communications* 53, 1-3 (1989), 253–269.
[17] Yuan Fan and Li Shengguo. 2022. YHAMG. https://gitee.com/e-level-parallel-algorithm/yhamg
[18] Jianbin Fang, Xiangke Liao, Chun Huang, and Dezun Dong. 2021. Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+. *Journal of Computer Science and Technology* 36, 1 (2021), 33–43.
[19] Andreas K Fidjeland, Etienne B Roesch, Murray P Shanahan, and Wayne Luk. 2009. NeMo: a platform for neural modelling of spiking neurons using GPUs. In *2009 20th IEEE international conference on application-specific systems, architectures and processors*. IEEE, 137–144.
[20] Andreas K Fidjeland and Murray P Shanahan. 2010. Accelerated simulation of spiking neural networks using GPUs. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.
[21] Assefaw H Gebremedhin, Duc Nguyen, Md Mostofa Ali Patwary, and Alex Pothen. 2013. Colpack: Software for graph coloring and related problems in scientific computing. *ACM Transactions on Mathematical Software (TOMS)* 40, 1 (2013), 1–31.
[22] G. H. Golub and C. F. Van Loan. 1996. *Matrix Computations* (3rd ed.). The Johns Hopkins University Press, Baltimore, MD.
[23] Joseph L Greathouse and Mayank Daga. 2014. Efficient sparse matrix-vector multiplication on GPUs using the CSR storage format. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 769–780.
[24] Magnus R Hestenes and Eduard Stiefel. 1952. Methods of conjugate gradients for solving. *Journal of research of the National Bureau of Standards* 49, 6 (1952), 409–436.
[25] Jonathan Joseph Hu and Andrey Prokopenko. 2015. *MueLu: Multigrid Framework for Advanced Architectures*. Technical Report. Sandia National Lab.(SNL-CA), Livermore, CA (United States).
[26] Huawei. 2019. Kunpeng 920. https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920
[27] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. 2012. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. 474–483. https://doi.org/10.1109/IPDPS.2012.51
[28] Takeshi Iwashita and Masaaki Shimasaki. 2003. Block red-black ordering: A new ordering strategy for parallelization of ICCG method. *International Journal of Parallel Programming* 31, 1 (2003), 55–75.
[29] George Karypis and Vipin Kumar. 1998. Multilevel k-way partitioning scheme for irregular graphs. *J. Parallel and Distrib. Comput.* 48, 1 (1998), 96—-129.
[30] David R Kincaid, John R Respess, David M Young, and Rober R Grimes. 1982. Algorithm 586: ITPACK 2C: A FORTRAN package for solving large sparse linear systems by adaptive accelerated iterative methods. *ACM Transactions on Mathematical Software (TOMS)* 8, 3 (1982), 302–322.
[31] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. Bishop. 2014. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units. *SIAM J. Sci. Comput.* 36, 5 (2014), C401–C423.
[32] Kiyoshi Kumahata, Kazuo Minami, and Naoya Maruyama. 2016. High-performance conjugate gradient performance improvement on the K computer. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 55–70.
[33] Jiajia Li, Guangming Tan, Mingyu Chen, and Ninghui Sun. 2013. SMAT: an input adaptive auto-tuner for sparse matrix-vector multiplication. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 117–126.
[34] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao. 2016. 623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores. *International Journal of High Performace Computing Applications* 30, 1 (2016), 39–54.
[35] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao. 2016. 623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores. *International Journal of High Performace Computing Applications* 30, 1 (2016), 39–54.
[36] Filippo Mantovani, Marta Garcia-Gasulla, José Gracia, Esteban Stafford, Fabio Banchelli, Marc Josep-Fabrego, Joel Criado-Ledesma, and Mathias Nachtmann. 2020. Performance and energy consumption of HPC workloads on a cluster based on Arm ThunderX2 CPU. *Future generation computer systems* 112 (2020), 800–818.
[37] SF McCormick and JW Ruge. 1982. Multigrid methods for variational problems. *SIAM J. Numer. Anal.* 19, 5 (1982), 924–929.
[38] P Morel. 1997. CESAM: A code for stellar evolution calculations. *Astronomy and Astrophysics Supplement Series* 124, 3 (1997), 597–614.
[39] Jayram Moorkanikara Nageswaran, Nikil Dutt, Jeffrey L Krichmar, Alex Nicolau, and Alex Veidenbaum. 2009. Efficient simulation of large-scale spiking neural networks using CUDA graphics processors. In *2009 International Joint Conference on Neural Networks*. IEEE, 2145–2152.
[40] Maxim Naumov. 2011. Parallel solution of sparse triangular linear systems in the preconditioned iterative methods on the GPU. *NVIDIA Corp., Westford, MA, USA, Tech. Rep. NVR-2011* 1 (2011).
[41] Maxim Naumov, M Arsaev, Patrice Castonguay, J Cohen, Julien Demouth, Joe Eaton, S Layton, N Markovskiy, István Reguly, Nikolai Sakharnykh, et al. 2015. AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods. *SIAM Journal on Scientific Computing* 37, 5 (2015), S602–S626.
[42] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep Dubey. 2014. Sparsifying synchronization for high-performance shared-memory sparse triangular solver. In *International Supercomputing Conference*. Springer, 124–140.
[43] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Xing Liu, Md Mosotofa Ali Patwary, Yutong Lu, and Pradeep Dubey. 2014. Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 945–955.
[44] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Md Mosotofa Ali Patwary, Vadim Pirogov, Pradeep Dubey, Xing Liu, Carlos Rosales, et al. 2016. Optimizations in a high-performance conjugate gradient benchmark for IA-based multi-and many-core processors. *The International Journal of High Performance Computing Applications* 30, 1 (2016), 11–27.
[45] Phytium. 2019. FT-2000+/64. https://en.wikichip.org/wiki/phytium/feiteng/ft-2000%2B-64
[46] Eugene L Poole and James M Ortega. 1987. Multicolor ICCG methods for vector computers. *SIAM J. Numer. Anal.* 24, 6 (1987), 1394–1418.
[47] Intel Product. 2022. Developer Reference for Intel oneAPI Math Kernel Library. https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
[48] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P Sadayappan. 2019. On optimizing complex stencils on GPUs. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 641–652.
[49] Oliver Rübel, Surendra Byna, Kesheng Wu, Fuyu Li, Michael Wehner, Wes Bethel, et al. 2012. TECA: A parallel toolkit for extreme climate analysis. *Procedia Computer Science* 9 (2012), 866–876.

[50] John W Ruge and Klaus Stüben. 1987. Algebraic multigrid. In *Multigrid methods*. SIAM, 73–130.
[51] Daniel Ruiz, Filippo Mantovani, Marc Casas, Jesús José Labarta Mancho, and Filippo Spiga. 2018. The HPCG benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform. https://upcommons.upc.edu/bitstream/handle/2117/116642/1HPCG_shared_mem_implementation_tech_report.pdf?sequence=8&isAllowed=y.
[52] Y. Saad. 1996. *Iterative methods for sparse linear systems*. PWS publishing Company, Boston, MA.
[53] Zhong Shijie. 1991. CitcomCU. https://github.com/geodynamics/citcomcu
[54] ARM Software. 2019. HPCG for ARM. https://github.com/ARM-software/HPCG_for_Arm
[55] Ulrich Trottenberg, Cornelius W Oosterlee, and Anton Schuller. 2000. *Multigrid*. Elsevier.
[56] Richard S Varga. 1962. *Iterative analysis*. Springer.
[57] Ioannis E Venetis and Astero Provata. 2022. Analysis of the Leaky Integrate-and-Fire neuron model for GPU implementation. *J. Parallel and Distrib. Comput.* 163 (2022), 1–19.
[58] Weiling Yang, Jianbin Fang, Dezun Dong, Xing Su, and Zheng Wang. 2021. LIBSHALOM: optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–14.
[59] Xiaosong Yu, Huihui Ma, Zhengyu Qu, Jianbin Fang, and Weifeng Liu. 2020. Numa-aware optimization of sparse matrix-vector multiplication on armv8-based many-core architectures. In *IFIP International Conference on Network and Parallel Computing*. Springer, 231–242.
[60] Thomas A Zang, Yau Shu Wong, and M Yousuff Hussaini. 1982. Spectral multigrid methods for elliptic equations. *J. Comput. Phys.* 48, 3 (1982), 485–501.
[61] X. Zhang, C. Yang, F. Liu, Y. Liu, and Y. Lu. 2014. Optimizing and scaling HPCG on Tianhe-2: early experience. In *Proc. 14th Int'l Conf. on Algorithms and Architectures of Parallel Processing(ICA3PP'14)*, X.-H. Sun et al (Ed.). Springer, Berlin Heidelberg, 28–41.
[62] Shijie Zhong, David A Yuen, Louis N Moresi, and G Schubert. 2007. Numerical methods for mantle convection. *Treatise on geophysics* 7 (2007), 227–252.
[63] Qianchao Zhu, Hao Luo, Chao Yang, Mingshuo Ding, Wanwang Yin, and Xinhui Yuan. 2021. Enabling and scaling the HPCG benchmark on the newest generation Sunway supercomputer with 42 million heterogeneous cores. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–13.