Optimizing Multi-Grid Preconditioned Conjugate Gradient Method on Multi-Cores

Fan Yuan, Xiaojian Yang, Shengguo Li, Dezun Dong, Chun Huang, and Zheng Wang

***Abstract*—Multigrid preconditioned conjugate gradient (MGPCG) is commonly used in high-performance computing (HPC) workloads. However, MGPCG is notoriously challenging to optimize since most of its computation kernels are memory-bounded with low arithmetic intensity and non-trivial communication patterns among parallel processes. This article presents new techniques to improve the data locality and reduce the communication overhead of MGPCG by first merging the kernels of multigrid (MG). We then develop an asynchronous neighboring communication algorithm to reduce the data communications across parallel processes. We demonstrated the benefits of our approach by applying it to the high-performance conjugate gradient (HPCG) benchmark and integrating it with a real-life algebraic multigrid package. We test the resulting software implementations on three ARMv8 and one Intel Xeon system. Experimental results show that our approach leads to a 1.62x-2.54x speedup over the engineer- and vendor-tuned HPCG implementations across various workloads and platforms.**

***Index Terms*—Multigrid, HPCG, symmetric Gauss-Seidel, asynchronous parallelization.**

I. Introduction

SPARSE linear solvers play a pivotal role in contemporary high-performance computing (HPC) and industrial applications, including computational fluid dynamics [1], astronomy, and environmental science [2], [3]. In scientific computing, the preference leans towards iterative methods due to their efficiency and ease of implementation when addressing large, sparse linear systems, in contrast to direct methods.

Iterative methods are often coupled with appropriate preconditioners to expedite convergence. Among these preconditioners, the multigrid (MG) method is one of the most effective. Applying MG techniques as a preconditioner for the conjugate gradient method (CG) is known as the multigrid preconditioned conjugate gradient method (MGPCG). The core of the multigrid approach involves data-intensive operations such as *smoothing*, *restriction*, *prolongation*, and *coarse-grid correction* [4]. These operations require substantial data movement between various levels of the hardware memory hierarchy and are characterized by a relatively low arithmetic intensity.

To reduce the memory access latency of MGPCG, we adopt the kernel fusion technique to improve data locality and arithmetic intensity [5], [6], [7], [8], [9]. Specifically, we have revised the computation process of classical symmetric Gauss-Seidel (SYMGS) step to obtain a more efficient formula that reduces both memory and computation costs. This revised approach can be further integrated with subsequent sparse matrix-vector multiplication (SPMV) operations. The SYMGS-SPMV kernel is utilized twice in MGPCG: first, as a pre-smoothing step followed by the residual computation in MG, and second, as the last post-smoothing step followed by the SPMV operation in CG. Our revision reduces memory and computational costs while preserving SYMGS semantics and keeping computational results intact. In addition, we further propose a monolithic MG that integrates a large number of representative MG operations. This new MG kernel can save nearly half of memory movements compared to the reference implementation of HPCG.

In addition to optimizing the computation kernel, our approach proposes a new parallelization scheme to maximize the parallelism among parallel threads. Our scheme builds upon the widely used block multi-core reordering (BMC) scheme for parallelizing SYMGS. It extends the recently proposed asynchronous BMC scheme [10] to reduce the overhead of dynamic scheduling by using a dependency sparsification [11] to eliminate redundant thread dependencies and enhance its asynchronous advantage.

Communication among processors plays a crucial role in memory-bounded operations. Especially when computing massively (scalable) parallel problems, the communication overhead between clusters becomes a new bottleneck. To mitigate the communication overhead in a multi-process program, we employ an asynchronous neighboring communication strategy. In a distributed computing environment, we observe that the interdependence between Multigrid (MG) algorithms across different distributed regions is primarily confined to the boundary data. This strategy aims to achieve computation and communication overlap by utilizing non-blocking and asynchronous functions, as well as by separating local and external computing tasks.

Manuscript received 14 October 2023; revised 29 January 2024; accepted 14 February 2024. Date of publication 5 March 2024; date of current version 18 March 2024. This work was supported in part by the National Key Research and Development Program of China under Grant 2022YFB4501702, and in part by the National Natural Science Foundation of China under Grant 62102444 and under Grant 62073333. Recommended for acceptance by M. Si. *(Fan Yuan and Xiaojian Yang contributed equally to this work. (Corresponding authors: Shengguo Li; Dezun Dong.)*

Fan Yuan is with the Department of Mathematics, National University of Defense Technology, Changsha 410073, China, and also with the Department of Mathematics, Xiangtan University, Xiangtan 411105, China (e-mail: fyuan@smail.xtu.edu.cn).

Xiaojian Yang, Shengguo Li, Dezun Dong, and Chun Huang are with the College of Computer Science, National University of Defense Technology, Changsha 410073, China (e-mail: yangxj@nudt.edu.cn; nudtlsg@nudt.edu.cn; dong@nudt.edu.cn; chunhuang@nudt.edu.cn).

Zheng Wang is with the School of Information Sciences and Technology, Northwest University, Xi'an 710069, China (e-mail: zwang@nwu.edu.cn).

Digital Object Identifier 10.1109/TPDS.2024.3372473

1045-9219 © 2024 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

We observe that this approach can improve the parallelization efficiency from 92.06% to 93.28% on a Phytium 2000+ cluster consisting of 256 nodes.

We evaluate our approach on an Intel Xeon and three ARM multi-core systems: Phytium 2000+ [12], Kunpeng (KP) 920 [13] and Thunder X2 [14]. We compared our approach against heavily-tuned HPCG implementations accelerated with industry-strength high-performance libraries [15], [16]. On the ARM platforms, it improves the ARM-specific HPCG implementation accelerated using the ARM Performance Library (ARMPL) [17] by 2.12x–2.54x. On the Intel platform, our approach improves the heavily-optimized HPCG version in the Intel Math Kernel Library (MKL) [16] by 1.62x. We evaluated our techniques on both a single computing node and in a distributed setup with 2048 MPI processes.

Our techniques are generally applicable. We have integrated our techniques to YHAMG [18] and unstructured matrices. YHAMG is a highly optimized algebraic multigrid (AMG) library. The generalization of applying solutions to linear equations of unstructured matrices is more extensive. Our techniques give a 1.4x speedup over the engineer-tuned implementation of YHAMG applications, and the BMC scheduling strategy also brings a 14% to 18% improvement for unstructured matrices.

This paper makes the following innovations:

1) It simplifies the SYMGS algorithm and incorporates kernel fusion. On this basis, a monolithic MG is proposed to further reduce the calculation and memory access costs;
2) It proposes a BMC scheduling with point-to-point synchronization, which alleviates the load imbalance problem, and combines it with dependency sparsification approach to further reduce the dynamic scheduling overhead cost.
3) Based on the kernel fusion strategy, an asynchronous neighbor communication strategy is proposed to reduce the synchronization waiting time of HPCG in parallel environment.

II. Background

A. HPCG Benchmark

HPCG is a benchmark used to evaluate the performance of supercomputers in solving a system of linear equations using the conjugate gradient method, which is commonly used to solve sparse linear systems in various scientific and engineering applications. HPCG aims to provide a more realistic representation of modern HPC workloads, taking into account not only floating-point operations but also memory access patterns and communication overhead.

HPCG uses a regular 27-point stencil discretization of an elliptic partial differential equation $\Delta u = f$, defined in a three-dimensional regular cubic domain with homogeneous Dirichlet boundary condition. The coefficient matrix A is symmetric and positive definite. The linear system $Ax = b$ is solved using the PCG method described in Algorithm 1, which uses a four-level V-cycle multigrid (MG) method as a preconditioner to accelerate the convergence. A symmetric Gauss-Seidel (SYMGS) step is performed as the pre and post-smoother and coarsest-level solver, with a natural ordering of the grid points in the reference HPCG. Any user-defined reordering is allowed to reduce data dependency and improve data locality, but the performance penalty caused by the increased iterations will be counted to the final result.

Algorithm 1: PCG for Solving Ax = b.

```
1  function PCG(A, M, b, x_0)
2      Compute r_0 := b − Ax_0
3      for k = 0, 1, ... do
4          if ||r_k||_2/||r_0|| ≤ τ then                ▷ Dot
5              break
6          end
7          Solve z_k = M^{-1} r_k                       ▷ MG
8          if k = 0 then
9              p_k = z_k
10         else
11             β_k := (r_k, z_k)/(r_{k−1}, z_{k−1})     ▷ Dot
12             p_k = z_k + β_k p_{k−1}                  ▷ WAXPBY
13         end
14         α_k = (r_k, z_k)/(p_k, Ap_k)                 ▷ SPMV, Dot
15         x_{k+1} = x_k + α_k p_k                      ▷ WAXPBY
16         r_{k+1} = r_k − α_k Ap_k                     ▷ WAXPBY
17     end
18     return x_k
19 end
```

Note that MG serves as a preconditioner in PCG, and SYMGS functions as a smoother within MG. In HPCG, SYMGS is the most time consuming kernel and various optimizations have been designed for SYMGS [9], [19].

HPCG uses a 3D domain decomposition technique to partition the global domain into several small subdomains. Each subdomain is assigned to a MPI process. There are two types of MPI communications. One is the MPI allreduce communication required to compute vector inner products. Three inner products are requires in each CG iteration. The other is neighboring halo-exchange communication to exchange the values of unknowns on boundary before performing SPMV and SYMGS operations.

B. Multigrid Algorithms

Iterative methods have proven effective in solving linear systems arising in physical modeling [20], [21]. 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 [22], [23], [24], [25], [26], [27], they typically operate on a hierarchy of discretizations (grids). There are different cycle types 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 [20], [28]. We use V-cycle as a working example, but our techniques can be equally applied to other mainstream MG implementations that use F-cycle and W-cycle.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 2: Standard V-Cycle MG Algorithm.

```
1  function MG (A^h, b^h, x^h)
2      if on the coarsest level then
3          x^h ← BottomSolver(A^h, b^h, x^h)            ▷ Bottom solver
4      else
5          repeat m1: x^h ← SYMGS(A^h, b^h, x^h)        ▷ Presmoothing
6          r^h ← b^h − A^h x^h                          ▷ Residual
7          r^2h ← I_h^2h r^h, x^2h ← 0                  ▷ Restriction
8          x^2h ← MG(A^2h, r^2h, x^2h)                  ▷ Recursion
9          x^h ← x^h + I_2h^h x^2h                      ▷ Prolongation
10         repeat m2: x^h ← SYMGS(A^h, b^h, x^h)        ▷ Postsmoothing
11     end
12     return x^h.
13 end
```

Algorithm 2 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 2). This process is repeated until the coarsest grid is reached where the cost of solution is negligible so that a *bottom solver* (line 3 in Algorithm 2) is applied to this collection of grids directly. Before and after operating on a finer grid, *pre-smoothing* (line 5 in Algorithm 2) and *post-smoothing* (line 10 in Algorithm 2) are respectively applied to cancel out errors from the coarser grids. Smoothing is followed by computing the *residual* error after the smoothing operation. 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.

C. Symmetric Gauss-Seidel

SYMGS can be used as a *smoother* and a *solver* in a MG implementation like Algorithm 2. 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 2). In other implementations, smoothing operations can be performed multiple times [29].

A SYMGS iteration consists of a forward and a backward *sweep* [20], which can be respectively defined as:

$$forward: \quad x^* = (D+L)^{-1}(b - Ux), \tag{1}$$

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

where the backward sweep takes as input the intermediate vector (x^*) 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 (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 forward sweep, x_1, into the backward sweep, we have

$$x^{**} = x + G(b - Ax), \tag{3}$$

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

III. Computation Optimization

Our approach has two strands. The first is to devise the SYMGS and SPMV computation patterns in MG in the sequential case. 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 IV.

A. SYMGS Kernel Optimization

To minimize the global memory access of the matrix, efficient data reuse and operation scheduling are required. In particular, an SPMV is performed following the backward sweep of the SYMGS in the pre-smoothing of V-cycle. SPMV involves basically the same data access pattern as SYMGS, but the former has no data dependency. Therefore, an efficient SYMGS-MV fusion method can be designed to improve performance.

The Symmetric Gauss-Seidel (SYMGS) is commonly implemented through a recursive approach in numerical calculations. During the forward sweep, the product of the non-zeros in each line and their corresponding unknowns is calculated. Non-zeros before the diagonal are multiplied by the updated unknowns, while non-zeros after the diagonal are multiplied by those not yet updated: for $i = 1, 2, \ldots, n$,

$$x_i^* = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^* - \sum_{j=i+1}^{n} a_{ij}x_j\right). \tag{4}$$

The subsequent backward sweep has a similar pattern but in the reverse order: for $i = n, \ldots, 1$,

$$x_i^{**} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^* - \sum_{j=i+1}^{n} a_{ij}x_j^{**}\right). \tag{5}$$

As observed, the SYMGS procedure currently accesses the sparse matrix twice in total. However, partial redundant computations can be avoided by computing and storing the intermediate result $w_i = b_i - \sum_{j=1}^{i-1} a_{ij}x_j^*$ during the forward sweep, which can then be reused in the backward sweep. To be more specific, in the backward sweep, the stored value w_i can be used to compute x_i^{**} as

$$x_i^{**} = \frac{1}{a_{ii}}\left(w_i - \sum_{j=i+1}^{n} a_{ij}x_j^{**}\right), \tag{6}$$

for $i = n, \ldots, 1$. This approach reduces the memory accesses required for the matrix by half, which can result in significant performance improvements for large and sparse matrices.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 3: Multigrid Preconditioning Kernel.

function MG-mono(A, b)
  Perform the forward sweep: $x_1 := (D + L)^{-1}b$
  Perform the backward sweep: $x_2 := (D + U)^{-1}Dx_1$ and $w_2 := -Ux_2$
  if on the coarsest level **return** x_2
  Exchange the border of x_2 between adjacent processors
  Compute the residual and restriction $r_c := I_{c:}(b + w_2) - L_{c:}x_2 - E_{c:}x_2$
  Perform recursive V-cycle $e_c :=$ MG-mono(A_c, r_c)
  Prolongation the border of $x_3 = x_2 + I_{:c}e_c$ and exchange it between adjacent processors
  Perform the forward sweep: $x_4 := (D + L)^{-1}(b + w_2 - U_{:c}e_c - Ex_3)$ and $w_4 := b - Ex_3 - Lx_4$
  Perform the backward sweep: $x_5 := (D + U)^{-1}w_4$
  return x_5
end

Additionally, the result w_i can be reused again if an SPMV $y = Ax^{**}$ following immediately after the backward sweep is required. Since

$$\sum_{j=i}^{n} a_{ij}x_j^{**} = b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{*}, \tag{7}$$

where the left-hand side is the product of the non-zeros after the diagonal and their corresponding values in x^{**}. Then, the remaining calculation of SPMV becomes

$$y_i = \sum_{j=0}^{i-1} a_{ij}x_j^{**} + w_i. \tag{8}$$

This leads to an efficient SYMGS-MV fusion.

Based on the domain decomposition strategy, a local SYMGS approach is employed in HPCG. Similar to the computation of SPMV, a neighboring (halo exchange) communication is required to exchange the external values before the forward SYMGS sweep. The external values will not be updated in the complete SYMGS step. In the parallel implementation, the product of external part $\sum_{j=n+1}^{m} a_{ij}x_j$, where m is the number of local columns (including the external entries), is required to be added to the right-hand side of (4) and (5). An even more efficient approach is adding the product to vector w by replacing it with

$$\hat{w}_i = b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{*} - \sum_{j=n+1}^{m} a_{ij}x_j. \tag{9}$$

It should be noted that a halo exchange is required after the backward sweep and before the subsequent SPMV, and the external product is required to be recalculated.

To maximize data locality in the fusion, it is crucial to divide the matrix A into four parts: the diagonal part D, the lower part L for the non-zeros before the diagonal, the upper part U for those after, and the external part E for the non-zeros corresponding to the entries in other processes, as shown in Fig. 1. In our implementation, the diagonal part is stored in a dense vector, while the remaining parts are stored in CSR (Compressed sparse row) format for universality. The SYMGS-MV fusion significantly reduces computation and data movement by eliminating the lower product in the backward sweep (see (6)) and the upper product in SPMV (8). This optimization requires only additional vector space to store the results of intermediate computations. However, the performance gains obtained by fusing SYMGS and SPMV operations typically mask this additional storage requirement.

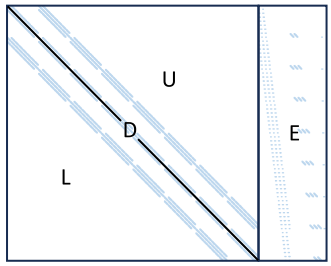


Fig. 1. Splitting the local distributed matrix A into four parts: $A = L + D + U + E$.

B. Multigrid Monolithic Kernel

The computation in multigrid is more complex than the SYMGS iteration. On the fine level of multigrid, SYMGS is applied twice in pre-smoothing and post-smoothing, respectively, and SPMV is applied once to compute the residual. The restriction operator only injects the corresponding points in the fine grid into the coarse grid, while the prolongation operator injects those from the coarse grid into fine grid. To futher enhance performance of HPCG, we design a monolithic kernel, which leverages the tight coupling of various kernels in multigrid and reduce the overhead associated with data movement and synchronization.

The multigrid monolithic kernel is described as Algorithm 3. Here $I_{c:}$ is the restriction operator, $I_{:c}$ is the prolongation operator and M_c is the preconditioning matrix for the coarse grid obtained by recursively applying multigrid V-cycle. $L_{c:} = I_{c:}L, E_{c:} = I_{c:}E$ are the corresponding coarse rows of L and E respectively, and $U_{:c} = UI_{:c}$ is the corresponding coarse columns of U. $L_{c:}, E_{c:}$ and $U_{:c}$ involve partially discontiguous memory accesses. To improve data locality, we store these matrices explicitly, which requires additional memory of size about $1/8$ of matrix A in total.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

Several optimization strategies are employed in Algorithm 3 to further reduce the overhead. First, since the initial guess is zero, the halo exchange communication and the computation of upper product for pre-smoothing can be safely removed (lines 2 and 3). Meanwhile, during the backward sweep of pre-smoothing, the uncorrected upper product $-Ux_2$ can be obtained directly since it has already been calculated (refer to (7)). Then the SYMGS and the subsequent SPMV can be fused to alleviate the bandwidth stress. Additionally, the SPMV is immediately used to compute the residual, which is then restricted to the coarse grid. In the restriction phase, only $1/8$ of the residual of fine grid points is required and injected into the coarse grid (line 6). So the SPMV computation on that other points can also be safely removed. Similarly, during the prolongation phase, only $1/8$ of fine grid points corresponding to the coarse grid points are corrected (line 8). The correction only occurs on the border of x for exchange between adjacent processors, as the local value of x_3 is not required in subsequent computations. Furthermore, it is possible to associate post-smoothing with pre-smoothing to achieve arithmetic reuse. Since the correction is only applied to the coarse points, and the overhead of the product is much smaller, the required corrected upper product for post-smoothing can be obtained by adding the correction of coarse points product to the uncorrected product, as $-Ux_3 = -U(x_2 + I_{:c}e_c) = w_2 - U_{:c}e_c$ (line 9). Similarly, w_4 can also be directly obtained during the forward sweep of post-smoothing, for use in the final backward sweep computations (line 10).

In the HPCG reference implementation, the MG preconditioning at the finest level necessitates complete memory access to the matrix A five times. In contrast, the MG monolithic kernel outlined in Algorithm 3 only requires accessing L and U approximately $2 + \frac{1}{8}$ times each. However, it's worth noting that Algorithm 3 does introduce some additional memory overhead for vector operations. Consequently, based on our assessments, the memory motion overhead of the MG monolithic kernel is roughly half that of the reference implementation.

Moreover, post-smoothing on the finest grid can be fused with the SPMV of Ap in PCG algorithm (line 14 of Algorithm 1). Specifically, the computation of Ap can be replaced with Az and a vector update (WAXPBY) operation, as follows:

$$Ap_k = Az_k + \beta_k Ap_{k-1}, \tag{10}$$

where z corresponds to the solution vector x in MG. Then, in Algorithm 3, we have:

$$Ax_5 = w_4 + (L + E)x_5. \tag{11}$$

This reduces one more access to the upper triangular matrix.

IV. Parallelization Optimization

A. BMC Scheduling With Point-to-Point Synchronization

The parallelism of symmetric Gauss-Seidel iteration with original ordering is very limited. To break the inherent dependency and expose more concurrency, multicoloring is commonly used, where points with the same color can be updated concurrently. However, multicoloring usually degrades the convergence rate, and also reduces the spatial locality of data access, because points with the same color are not adjacent in the original graph.

Block multicoloring (BMC) [30], [31] is proposed to improve the convergence rate and locality by grouping adjacent points into blocks. BMC takes advantage of the fact that the points within each block are typically adjacent to one another, resulting in the parallelism of $(N_xN_yN_z)/(bc)$, where b is the size of blocks, and c is the number of colors [10]. The values of b and c can be reasonably determined according to the geometric information of grids and the number of threads.

BMC faces similar problems. To enhance data locality and convergence speed, the common approach is to increase the block size. Classic BMC determines the execution order of blocks based on the priority of the color. However, this leads to a significant reduction in parallelism, particularly for the number of threads that cannot evenly distribute computational tasks, since a barrier synchronization occurs between different colors. For instance, we allocate four processes and seven threads on a 28-core Intel CPU. To improve synchronization latency in scenarios where the number of blocks with the same color is not a multiple of seven, reducing the block size is a viable approach. However, this sometimes results in performance degradation.

To address the load imbalance caused by the BMC approach, we propose an asynchronous block multicolor scheduling, which leverages the dependencies between blocks to schedule the tasks without barriers. We note that any block in the BMC graph depends only on the blocks that are adjacent to it and have a higher priority than it. Therefore, during scheduling, a block can be processed when all of its dependencies have been completed, without having to wait until all blocks with a higher priority than it have been processed. In other words, blocks of different colors can be processed in parallel as long as there is no dependency among them, which indirectly improves the parallelism of BMC. By using this point-to-point synchronization, we can avoid the additional overhead incurred by barrier synchronization, which is proportional to the logarithm of the number of cores. This method can be particularly useful in situations where the unbalanced tasks allocation makes synchronization prohibitively expensive.

In addition, we employ the synchronization sparsification technique [11] to eliminate redundant transitive dependencies. As shown in Fig. 2(a), the dependencies between different color blocks are too redundant in the classical BMC graph partitioning. In contrast, Fig. 2(b) demonstrates the scheduling flow after sparsification, which preserves critical dependencies while greatly reducing unnecessary edges. For example, in HPCG, about $3/4$ of the dependencies can be safely eliminated. This greatly reduces the overhead of dynamic scheduling. Generally, our scheduling scheme can operate efficiently, provided that the number of blocks with the same color is greater than the number of threads being used.

In our implementation, we use the tasking model introduced by OpenMP 3.0 to implement the point-to-point synchronization. Algorithm 4 describes the scheduling process for forward

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

Algorithm 4: Forward Sweep of SYMGS With BMC Scheduling.

```
function Forward(x, num_Colors, *blocks_Color, *dpPtr, *dpNum, *flag)
    #pragma omp parallel
    #pragma omp single
    for clr = 0 to num_Color − 1 do
        for blk = blocks_Color[clr] to blocks_Color[clr + 1] − 1 do
            #pragma omp task depend (iterator(it = dpPtr[blk] : dpPtr[blk + 1]), in:flag[dpNum[it]]) depend
              (out:flag[blk])
            for curRow = block_ptr[blk] to block_ptr[blk + 1] − 1 do
                Compute x*_curRow.
            end
        end
    end
    return x*.                          ▷ The Backward sweep process is similar and in reverse order.
end
```

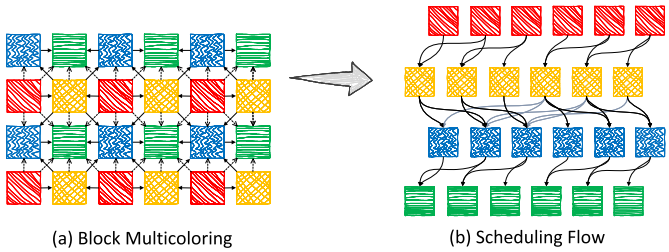


Fig. 2. Block Multicolor scheduling with synchronization sparsification. (a) The block multicoloring and the dependencies between the blocks, where the arrow with dashed line represents a redundant dependency. (b) The task scheduling after synchronization sparsification.

sweep of SYMGS, and the reverse for backward sweep. In this algorithm, the variables num_Colors and $*block_Color$ respectively denote the count of colors and the quantity of blocks of each color, $*dpPtr$ stores the number of edges each block depends on and their serial numbers are stored in the $*dpNum$, and $*flag$ is the carrier. This approach can be applied in the solution of unstructured matrices.

B. Asynchronous Neighboring Communication

Neighboring halo exchange communication is used to exchange data between adjacent subdomains or processors. This communication is necessary in parallel computing since each processor can only access local subdomain data and the boundaries of each processor's subdomain is also required for the computation. In the reference HPCG implementation, a neighboring halo exchange communication must be completed before the local computation of SPMV or SYMGS can commence. Although the actual time of the neighboring communication is quite short, the synchronous waiting caused by unbalanced task progress or variably core speed is still of concern. Thus, it is essential to use non-blocking and asynchronous functions to achieve the overlap of computation and communication.

The communication process involves three key steps: packing data and initiating halo communication, data transfer, and concluding the communication. Initially, the border data is collected and compacted into a buffer. Subsequently, non-blocking receiving requests are initiated from adjacent processors with MPI_IRECV routine, while non-blocking sending requests are initiated to adjacent processors with MPI_ISEND routine. As the data transfer proceeds in the background, the processor can continue executing computing tasks. Finally, the halo exchange is concluded by waiting for the completion of sending and receiving requests with the MPI_WAIT routine. Fig. 3(a) illustrates the computation and communication process in Algorithm 3. There are two neighboring communications at each level of the multigrid. It is important to highlight that the CPU remains idle during these communications. In the SPMV procedure, achieving the overlap of local computation and neighboring communication is relatively straightforward, as outlined in [5]. However, implementing asynchronous communication in SYMGS and integrating it with the reordering method remains a challenging task.

An improved kernel is presented in Fig. 3(b). The SPMV computation can be divided into two parts: a local component and an external component based on columns. The former only requires the local values of x, while the latter depends solely on border data. The local part is allowed to be computed concurrently with the transfer of border data. Subsequently, the external part can be calculated once the communication phase concludes. That is why we split the external part E from matrix A in Section III-A, which leads to the efficient communication methods. Building on this, the first SPTRSV in SYMGS can employ a similar method for asynchronous communication. The same principle applies to fusion. This strategy is also integrated into our MG kernel. In this improved configuration, the CPU can continue performing computational tasks during communication, significantly reducing CPU waiting times.

V. EXPERIMENTAL SETUP

A. Evaluation Platforms

To demonstrate the effectiveness of our optimization strategies, we critically evaluated our approach on both ARM and x86 platforms. Table I lists the hardware platforms used in this work, including three ARMv8 architectures [32], [33] and an Intel Xeon Gold-5117 CPU. We stress that our work focuses on performance optimization of individual CPUs, but we also extend our evaluation to an Phytium 2000+ cluster of 256 computing nodes.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

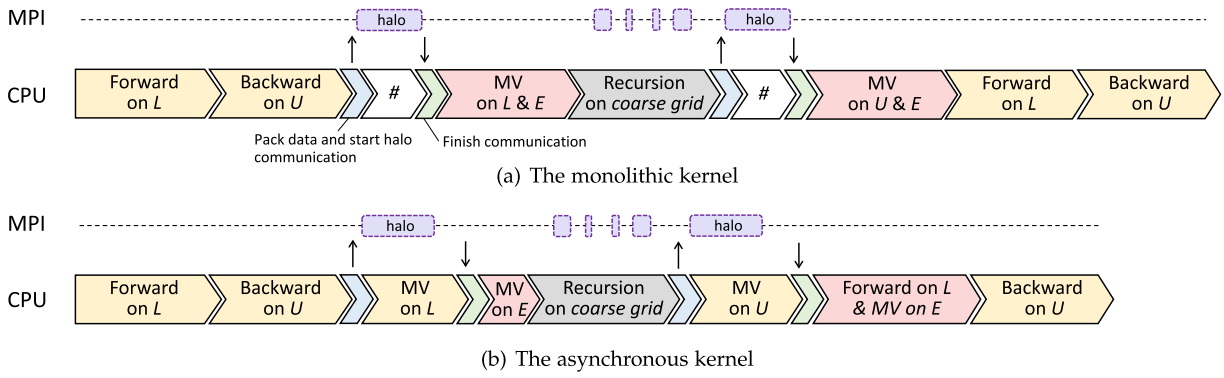


Fig. 3. Asynchronous neighboring communication reduces the data movement overhead in the multigrid monolithic kernel. The blocks marked with "#" on the CPU indicate idle time. The yellow blocks only involve local data, while the red blocks involve border data and need to wait for communication to end before starting.

TABLE I
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

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

B. 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 V-C). HPCG runs using multiple MPI processes where each process is further parallelized using OpenMP threads on a shared-memory machine. We also port our techniques to realistic linear solver applications: YHAMG [18] and the linear solver of unstructured matrices. All 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 YHAMG to solve the Poisson problem discretized by a 7-point and a 27-point stencil, respectively.

C. Competing Baselines

HPCG for ARM: We compared our HPCG implementation against the ARM-architecture-tuned HPCG developed by ARM [15], [34]. 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 [7] 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) [17] 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) [16]. We denote this version as *HPCG_for_MKL*.

YHAMG: This application has already been heavily optimized by its developers. YHAMG uses hand-optimized SYMGS and SPMV kernels. We show that our techniques can be applied to this application to further improve its performance.

Barrier Synchronization BMC: We compare our BMC scheduling scheme in Section IV-A with the barrier synchronization BMC scheme in numerous types of sparse linear solution scenarios, which runs on multi-core CPUs using OpenMP.

D. Evaluation Methodology

In our experiments, we evaluate our computational optimization strategies described in Section III and parallelization strategies described in Section IV respectively. These two types of approaches can be co-integrated to work together.

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).

VI. EXPERIMENTAL RESULTS

A. Overall Performance on Single-Node

Fig. 4 reports the HPCG performance (measured in GFLOPS) for various combinations of our optimizations and the competing baselines across our evaluation platforms. The experiments were performed on a single computing node using all the physical

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

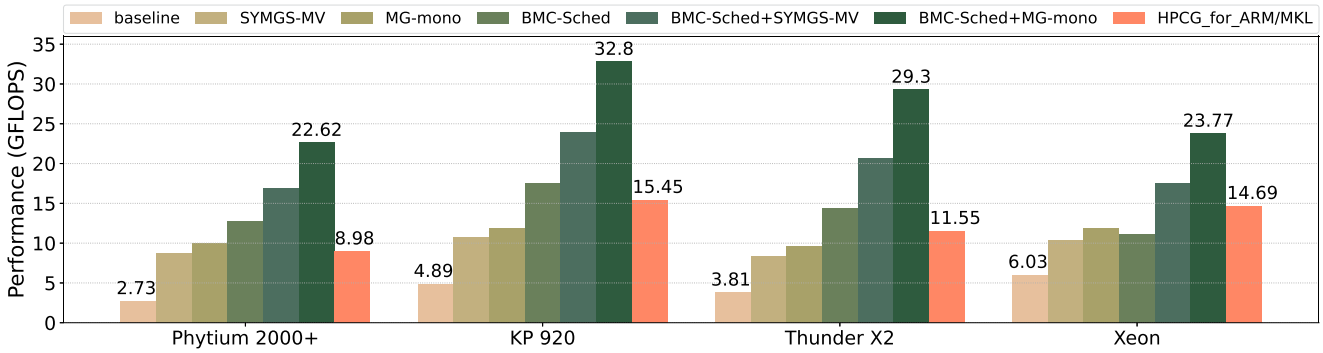


Fig. 4. Overall comparison of our optimized HPCG with the hardware-specific implementations on a single node. We have arranged combinations of various optimization strategies to show in more detail how each strategy works. The implementation of `BMC-Sched+MG-mono` has optimal performance.

processor cores. The meaning of these optimization versions are:

- `baseline`: Official release with no optimizations.
- `SYMGS-MV`: Fusion of optimized SYMGS and SPMV.
- `MG-mono`: MG monolithic kernel contains `SYMGS-MV`.
- `BMC-Sched`: A single BMC scheduling with point-to-point synchronization strategy.

The rest of the implementations are combinations of the above methods. The advantages of asynchronous neighboring communication strategy will be shown in Section VI-C. 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. From the results in Fig. 4, the implementation of `BMC-Sched+MG-mono` has optimal performance, which combines both computational optimizations and parallel optimizations. Specifically, our techniques improve *HPCG_for_ARM* by 2.12x–2.54x and *HPCG_for_MKL* on x86 by 1.62x.

B. Evaluation of Parallel Optimization

Our BMC scheduling policy maintains good performance even when the load is unbalanced. To show more clearly the mitigating effect of the BMC scheduling on load balancing, we compared the performance of BMC scheduling with the synchronization case on multiple platforms.

The performance variation of the two strategies in different scenarios is shown in Fig. 5. The experiments are performed at different numbers of parallel threads, the load imbalance with OpenMP = 6 and balance with OpenMP = 8, using eight blocks of each color. As can be seen in Fig. 5, the performance of `BMC-Sched` is similar to that of the synchronization policy when each thread is responsible for a similar number of tasks. However, when the tasks are not evenly distributed, the performance of the barrier synchronization policy drops drastically, while the BMC scheduling policy mitigates this degradation well.

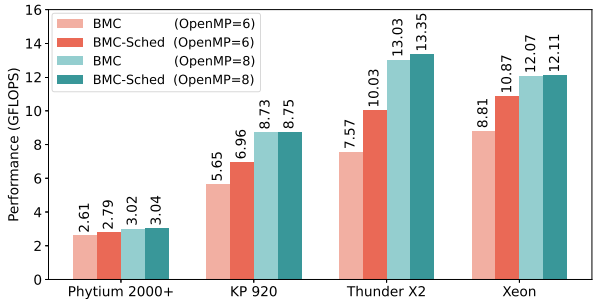


Fig. 5. Performance comparison between the BMC scheduling `BMC-Sched` and barrier synchronization `BMC` strategy for load imbalance (OpenMP = 6) and load balancing (OpenMP = 8) scenarios.

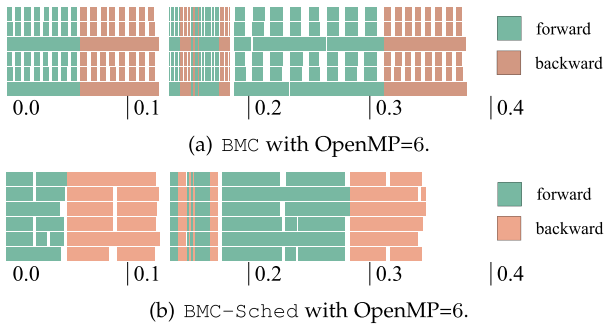


Fig. 6. Thread calls when computing SYMGS kernels in MG during BMC scheduling (a) and barrier synchronization (b) modes, where the threads are computed (solid blocks) and the threads are idle (blank gaps). The experiment used six parallel threads to process eight blocks of each color.

To closely examine the benefit of our scheduling scheme, please refer to Fig. 6. This diagram shows the computation and idle time of parallel threads measured on the 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 threads is smaller than the number of blocks that can be run concurrently. Furthermore, we observe more frequent and longer thread idle at the synchronization execution mode than in our scheduling mode. This is because our BMC scheduling policy allows parallel threads to

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

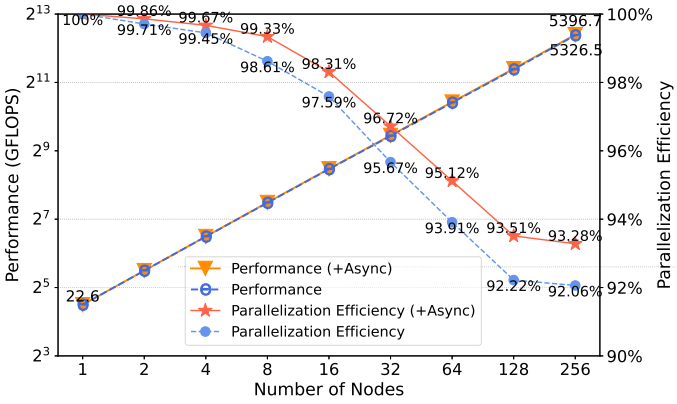


Fig. 7. Performance and parallelization efficiency of our HPCG implementation in a Phytium 2000+ computing cluster with 256 nodes.

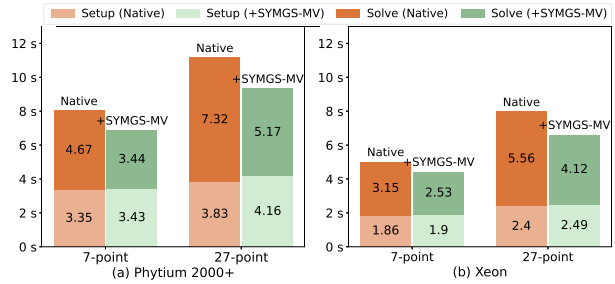


Fig. 8. Runtime measurement (in seconds) for different stages of YHAMG.

work on a block with 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 the synchronous mode in this experiment.

C. Multi-Node Performance Scalability

As the computation time is reduced by our computational optimizations, the communication overhead may start to limit the parallelization efficiency. Therefore, we test the parallel scalability of HPCG incorporated with our optimization strategies. The experiments are conducted on a 256-node Phytium 2000+ cluster, using a total of 16,384 in cores. We extended our experiments to 2048 MPI processes, each using eight cores.

We test the case with and without asynchronous neighboring communication strategy (`+Async`) based on the previous best optimization `BMC-Sched+MG-mono`. Their performance and parallelization efficiency are shown in Fig. 7, our implementations exhibit a good scalability with nearly linear improvement on GFLOPS as the number of computing nodes used increases. With 256 nodes, the other strategies can deliver a performance of 5326.5 GFLOPS with a parallelization efficiency of 92.06%. Our asynchronous neighbor communication strategy further improves the performance. Specifically, `+Async` further improves the GFLOPS to 5396.7, and the parallelization efficiency to 93.28%. Our communication optimization strategy is even more advantageous as the number of nodes increases. This implies an improvement in the efficiency of the communication process, thereby enhancing the overall performance of memory-bounded operations.

VII. Applications

A. Algebraic Multigrid

In this evaluation, we incorporate our `SYMGS-MV` technique into the algebraic multigrid (AMG) kernel [35], [36] in the YHAMG library [18]. We configure the local grid scale as $128 \times 128 \times 128$ and solve a classical Poisson problem discretized by a 7-point and a 27-point stencil, respectively. The

TABLE II
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	Hook_1498	1,498K	60.92M	41
J	inline_1	504K	36.82M	73
K	ldoor	952K	46.52M	49
L	ML_Geer*	1,504K	110.88M	74
M	pwtk	218K	11.63M	53
N	Serena	1,391K	64.53M	46
O	ship_003	122K	8.09M	66
P	shipsec1	141K	7.81M	55
Q	tmt_sym	727K	5.08M	7

*These two matrices are unsymmetric. Others are symmetric.

experimental trials were conducted on both the Phytium 2000+ and Xeon systems, each utilizing a single node configuration.

Fig. 8 shows the performance results, where *Setup* and *Solve* denote the wall times (in seconds) for the setup and solve stage of AMG [28], 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 `SYMGS-MV` 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.

B. Unstructured Matrices

Our computation and parallelization optimization strategies can be applied to linear solvers for unstructured matrices. We apply our optimizations to a set of representative unstructured matrices from SuiteSparse [37] given in Table II. The main

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

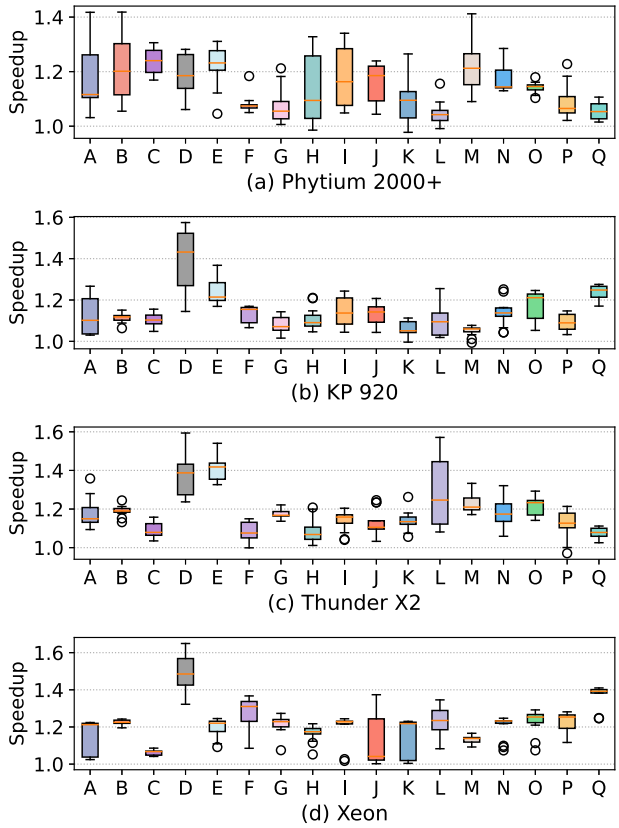


Fig. 9. Speedup of our BMC scheduling strategy when performing SYMGS compared to non-scheduled mode. A wide range of number of block partitions is chosen for each matrix. A to Q represent the serial numbers of matrices shown in Table II.

purpose of this evaluation is to demonstrate the benefits of BMC scheduling compared to the barrier synchronization mode.

In this experiment, we use an ABMC (Algebraic block multi-coloring) [30] to parallelize SYMGS and measure the wall times when performing three times of SYMGS. We first use METIS (version 5.1.0) [38] to partition the matrices into blocks and then use Colpack [39] to map blocks into colors, where blocks of the same color can be executed in parallel. Since the graphical partitioning scheme of the matrix can vary significantly when choosing different block numbers, this can lead to a significant performance degradation. Therefore, we chose a larger range of number of blocks, ranging from 100 to 1000, as this provides a good trade-off between parallelism and processing overhead.

Fig. 9 shows the speedup when using the scheduling mode versus the barrier synchronization mode for each matrix across all platforms. We observe that when the number of blocks is large, the dynamic scheduling overhead of the OpenMP task mechanism becomes large due to the complexity of inter-block dependencies, which leads to suboptimal performance. Therefore, we eliminate the overhead of dynamic scheduling by assigning task blocks to threads in advance during the preprocessing phase. The results show that our BMC scheduling strategy achieves good speedup ratios compared to non-scheduling even when there are very complex dependencies among blocks. Specifically, for the matrices in the experiments, our method improves the computational efficiency of BMC by 14% to 18% across all evaluation platforms.

VIII. 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. We proposes a new way to optimize SYMGS in [10]. We achieve this by partitioning the input matrix into submatrices around which we develop a new computation formula to reduce the computation cost. In comparison with [10], this work incorporates additional details within the context of distributed parallel computing, shown in Fig. 1. A closely related prior work [19] also optimizes the forward and backward sweep of SYMGS by accessing the upper and lower triangular parts separately, aiming to improve the locality of vector x. Unlike our approach that stores the lower and upper triangular parts of matrix A separately, [19] stores the entire matrix in the CSR format. As such, it does not reduce memory access to A nor save the computation. The work [9] stores matrices of L and U separately. However, its SYMGS algorithm requires one more SPTRMV operation than our optimization kernel.

Kernel fusion is one of the commonly used techniques for optimizing MG. For example, a SYMGS-SPMV fusion technique is proposed in [40] 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 [9]. Besides fusing SYMGS with SPMV, a red-black coloring along z-axis is proposed in [8] and a forward and backward SYMGS smooth fusion is tested on Tianhe-2 supercomputer.

Generally speaking, there are two types of parallelization strategies for SYMGS: *level scheduling* [7], [41] and *color reordering* [5], [9], [42]. The multi-color reordering method (MC) [43] has long been used to parallelize SYMGS, but its drawback is having poor data locality. Later BMC [31] alleviates the problem of poor locality by using blocks as shading units instead of points. Our previous work [10] combines level scheduling with color reordering and introduce an asynchronous BMC scheduling. In this work, we further extend it by combining with dependency sparsification approach to further reduce the dynamic scheduling overhead cost in Section IV-A. ABMC is proposed in [30], which is suitable for solving linear systems with unstructured matrices. Our scheduling strategy can also be applied to this block partitioning approach.

The analysis in [44] shows that it is crucial to improve network communication in multicore processors in large-scale multigrid computing. Various optimization techniques such as communication avoidance have also been studied in [44] for MG on multicore systems and Intel Xeon Phi coprocessors. A compiler technique that automatically weighs computational and communication workloads implements the communication-avoidance optimization of the smoothing operator in MG [45]. Every computational core in MG is subdivided into smaller, more fine-grained tasks, maximizing their utilization to overlap communications with other operations [42]. [5] implements a

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

heterogeneous scheduling approach on Tianhe-2 to minimize the data movement overhead of HPCG associated with MPI neighborhood communication and PCI-E data transmission. Our asynchronous neighbor communication strategy separates local and external computing tasks using non-blocking and asynchronous functions to overlap computation with communication. In comparison with [10], summarily, we introduce four additional contributions: 1) the details of optimized SYMGS in the context of distributed parallel computing is added, refer to Fig. 1; 2) a monolithic MG is proposed in Section III-B; 3) the asynchronous BMC scheme is further optimized; 4) an asynchronous neighboring communication strategy is introduced in Section IV-B.

IX. Conclusion

We have presented a set of new techniques to optimize multigrid (MG) algorithms and high-performance conjugate gradient (HPCG) on multi-cores. Our approach focuses on optimizing the symmetric Gauss-Seidel (SYMGS) method, the most time-consuming kernel in MG. Based on the split storage of sparse matrix, an optimized SYMGS is proposed, which reduces both the computation and memory access costs. We also propose a new multigrid monolithic kernel (`MG-mono`) that requires almost only half of the data access compared to the classic implementations. The optimized kernels (`SYMGS` and `MG-mono`) can be integrated with the sparse matrix-vector (SPMV) multiplication kernels to further reduce computation and memory access overhead. To parallelize SYMGS, we combine block multi-color (BMC) with level scheduling strategy and a BMC scheduling method is proposed to resolve the load imbalance problem among threads.

Furthermore, we present an asynchronous neighboring communication strategy based on our computational optimizations, which effectively improves the parallelization efficiency in large-scale distributed environments. We demonstrate the benefit of our approaches by porting them into the HPCG benchmark and one real-life application package, YHAMG [18]. Many unstructured matrices from SuiteSparse [37] are also used to test our BMC scheduling approach. Experimental results show that our approach improves 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.

References

[1] O. Rübel et al., "TECA: A parallel toolkit for extreme climate analysis," *Procedia Comput. Sci.*, vol. 9, pp. 866–876, 2012.
[2] P. Morel, "CESAM: A code for stellar evolution calculations," *Astron. Astrophys. Suppl. Ser.*, vol. 124, no. 3, pp. 597–614, 1997.
[3] S. Zhong, D. A. Yuen, L. N. Moresi, and G. Schubert, "Numerical methods for mantle convection," *Treatise Geophys.*, vol. 7, pp. 227–252, 2007.
[4] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A Multigrid Tutorial*. Philadelphia, PA, USA: SIAM, 2000.
[5] Y. Liu et al., "623 Tflop/s HPCG run on Tianhe-2: Leveraging millions of hybrid cores," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 39–54, 2016.
[6] G. Wang, Y. Lin, and W. Yi, "Kernel fusion: An effective method for better power efficiency on multithreaded GPU," in *Proc. IEEE/ACM Int. Conf. Green Comput. Commun. Int. Conf. Cyber Phys. Social Comput.*, 2010, pp. 344–350.
[7] J. Park et al., "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2014, pp. 945–955.
[8] X. Zhang, C. Yang, F. Liu, Y. Liu, and Y. Lu, "Optimizing and scaling HPCG on Tianhe-2: Early experience," in *Proc. 14th Int. Conf. Algorithms Architectures Parallel Process.*, Springer, 2014, pp. 28–41.
[9] Q. Zhu, H. Luo, C. Yang, M. Ding, W. Yin, and X. Yuan, "Enabling and scaling the HPCG benchmark on the newest generation sunway supercomputer with 42 million heterogeneous cores," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–13.
[10] X. Yang, S. Li, F. Yuan, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid computation and parallelization on multi-cores," in *Proc. 37th Int. Conf. Supercomputing*, New York, NY, USA, 2023, pp. 227–239.
[11] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Proc. Int. Supercomputing Conf.*, Springer, 2014, pp. 124–140.
[12] Phytium, "FT-2000+/64," 2019. [Online]. Available: https://en.wikichip.org/wiki/phytium/feiteng/ft-2000%2B-64
[13] Huawei, "Kunpeng 920," 2019. [Online]. Available: https://www.hisilicon.com/cn/products/Kunpeng/Huawei-Kunpeng/Huawei-Kunpeng-920
[14] F. Mantovani et al., "Performance and energy consumption of HPC workloads on a cluster based on arm ThunderX2 CPU," *Future Gener. Comput. Syst.*, vol. 112, pp. 800–818, 2020.
[15] D. Ruiz, F. Mantovani, M. Casas, J. J. Labarta Mancho, and F. Spiga, "The HPCG benchmark: Analysis, shared memory preliminary improvements and evaluation on an arm-based platform," 2018. [Online]. Available: https://upcommons.upc.edu/bitstream/handle/2117/116642/1HPCG_shared_mem_implementation_tech_report.pdf?sequence=8&isAllowed=y
[16] I. Product, "Developer Reference for Intel oneAPI Math Kernel Library," 2022. [Online]. Available: https://www.intel.com/content/www/us/en/develop/documentation/onemkl-developer-reference-c/top.html
[17] ARM, "Arm performance libraries," 2022. [Online]. Available: https://www.arm.com/products/development-tools/server-and-hpc/allinea-studio/performance-libraries
[18] Y. Fan and L. Shengguo, "YHAMG," 2022. [Online]. Available: https://gitee.com/e-level-parallel-algorithm/yhamg
[19] K. Kumahata, K. Minami, and N. Maruyama, "High-performance conjugate gradient performance improvement on the K computer," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 55–70, 2016.
[20] Y. Saad, *Iterative Methods for Sparse Linear Systems*. Boston, MA, USA: PWS-Kent, 1996.
[21] J. Demmel, *Applied Numerical Linear Algebra*. Philadelphia, PA, USA: SIAM, 1997.
[22] T. A. Zang, Y. S. Wong, and M. Y. Hussaini, "Spectral multigrid methods for elliptic equations," *J. Comput. Phys.*, vol. 48, no. 3, pp. 485–501, 1982.
[23] E. Chow, A. J. Cleary, and R. D. Falgout, "Design of the hypre preconditioner library," in *Proc. SIAM Workshop Object Oriented Methods Inter-Operable Sci. Eng. Comput.*, 1999, pp. 106–116.
[24] J. J. Hu and A. Prokopenko, "MueLu: Multigrid framework for advanced architectures," Sandia National Lab. (SNL-CA), Livermore, CA, USA, Tech. rep., 2015.
[25] D. Demidov, "AMGCL: An efficient, flexible, and extensible algebraic multigrid implementation," *Lobachevskii J. Math.*, vol. 40, no. 5, pp. 535–546, 2019.
[26] S. Balay et al., "PETSc/TAO users manual," Argonne National Laboratory, Lemont, IL, USA, Tech. Rep. ANL-21/39 - Revision 3.17, 2022.
[27] M. Naumov et al., "AmgX: A library for GPU accelerated algebraic multigrid and preconditioned iterative methods," *SIAM J. Sci. Comput.*, vol. 37, no. 5, pp. S602–S626, 2015.
[28] U. Trottenberg, C. W. Oosterlee, and A. Schuller, *Multigrid*. Amsterdam, The Netherlands: Elsevier, 2000.
[29] J. Assunção and V. Sacek, "Benchmark comparison study for mantle thermal convection using the CitcomCU numerical code," in *Proc. 15th Int. Congr. Braz. Geophysical Soc.*, 2017, pp. 1630–1635.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.

[30] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in ICCG method," in *Proc. IEEE 26th Int. Parallel Distrib. Process. Symp.*, 2012, pp. 474–483.
[31] T. Iwashita and M. Shimasaki, "Block red-black ordering: A new ordering strategy for parallelization of ICCG method," *Int. J. Parallel Program.*, vol. 31, no. 1, pp. 55–75, 2003.
[32] W. Yang, J. Fang, D. Dong, X. Su, and Z. Wang, "LIBSHALOM: Optimizing small and irregular-shaped matrix multiplications on ARMv8 multi-cores," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–14.
[33] J. Fang, X. Liao, C. Huang, and D. Dong, "Performance evaluation of memory-centric ARMv8 many-core architectures: A case study with Phytium 2000+," *J. Comput. Sci. Technol.*, vol. 36, no. 1, pp. 33–43, 2021.
[34] A. Software, "HPCG for ARM," 2019. [Online]. Available: https://github.com/ARM-software/HPCG_for_Arm
[35] J. W. Ruge and K. Stüben, "Algebraic multigrid," in *Multigrid Methods*. Philadelphia, PA, USA: SIAM, 1987, pp. 73–130.
[36] S. McCormick and J. Ruge, "Multigrid methods for variational problems," *SIAM J. Numer. Anal.*, vol. 19, no. 5, pp. 924–929, 1982.
[37] T. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, 2011.
[38] G. Karypis and V. Kumar, "Multilevel k-way partitioning scheme for irregular graphs," *J. Parallel Distrib. Comput.*, vol. 48, no. 1, pp. 96–129, 1998.
[39] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, "ColPack: Software for graph coloring and related problems in scientific computing," *ACM Trans. Math. Softw.*, vol. 40, no. 1, pp. 1–31, 2013.
[40] J. Park et al., "Optimizations in a high-performance conjugate gradient benchmark for IA-based multi-and many-core processors," *Int. J. High Perform. Comput. Appl.*, vol. 30, no. 1, pp. 11–27, 2016.
[41] E. Anderson and Y. Saad, "Solving sparse triangular linear systems on parallel computers," *Int. J. High Speed Comput.*, vol. 1, no. 01, pp. 73–95, 1989.
[42] Y. Ao, C. Yang, F. Liu, W. Yin, L. Jiang, and Q. Sun, "Performance optimization of the HPCG benchmark on the sunway TaihuLight supercomputer," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 1, Mar. 2018, Art. no. 11.
[43] E. L. Poole and J. M. Ortega, "Multicolor ICCG methods for vector computers," *SIAM J. Numer. Anal.*, vol. 24, no. 6, pp. 1394–1418, 1987.
[44] S. Williams et al., "Optimization of geometric multigrid for emerging multi- and manycore processors," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2012, pp. 1–11.
[45] P. Basu, A. Venkat, M. Hall, S. Williams, B. Van Straalen, and L. Oliker, "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid," in *Proc. 20th Annu. Int. Conf. High Perform. Comput.*, 2013, pp. 452–461.



Fan Yuan received the BS degree from the Dalian University of Technology, Dalian, China, in 2014, and the MS degree from the Guilin University of Electronic Technology, Guilin, China, in 2018. He is currently working toward the PhD degree in computational mathematics with Xiangtan University, Xiangtan, China. His research interests include numerical linear algebra, multigrid, and high-performance computing.



Xiaojian Yang received the BS degree from Xidian University, Xi'an, China, in 2021. He is currently working toward the MS degree with the National University of Defense Technology (NUDT), Changsha, China. His research interests include high-performance computing, performance optimization, and parallel programming.



Shengguo Li received the BS, MS, and PhD degrees in computational mathematics from the National University of Defense Technology, Changsha, China. He is currently an Assistant Professor with the College of Computer, NUDT. His research interests include numerical linear algebra, high-performance computing, and machine learning.



Dezun Dong received the BS, MS, and PhD degrees from the National University of Defense Technology (NUDT), Changsha, in 2002, 2004, and 2010, respectively. He is a professor with the College of Computer, NUDT, where he leads the Research Group on High-Performance Network and Architecture (HiNA) and serves as the deputy director of the Tianhe supercomputer. His research interests focus on high-performance networks and architecture for supercomputers, datacenter, and deep learning systems. He has published more than 70 peer-reviewed papers in reputed international journals and conferences.



Chun Huang received the MSc and PhD degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China. She is a full professor in computer science with NUDT, where she serves as the director of the Tianhe Supercomputer Compilation System. Her research interests include high-performance computing, system software, parallel compilers, parallel programming, performance optimization, and high-performance math libraries.



Zheng Wang is an adjunct faculty member with the School of Information Science and Technology, Northwest University. His research interests include high-performance computing and software engineering.

Authorized licensed use limited to: National Univ of Defense Tech. Downloaded on August 03,2026 at 06:25:21 UTC from IEEE Xplore. Restrictions apply.