[2025-3]

KEASE 기술서

한국형 엑사스케일 응용 SW 개발 환경 프레임워크

Exploring Variants of Symmetric Gauss-Seidel (SymGS) Algorithms in HPCG

2025. 3. 10

숭 실 대 학 교

[개 정 이 력]

개정 일자	버전	설명	개정페이지	작성자	확인자
2025.3.10	1.0	INITIAL DRAFT	전체	Muhammad Rizwan	최재영

목 차

0. Abstract

This study explored the algorithmic variants of the Symmetric Gauss-Seidel (SymGS) method for Symmetric Positive Definite (SPD) matrices in HPCG. SymGS is a smoother in iterative techniques for large sparse linear systems, especially in multigrid methods and the High-Performance Conjugate Gradient (HPCG) benchmark. Nonetheless, the reference implementation of SymGS in HPCG exhibits limited parallel scalability due to data dependencies. We present several variants of SymGS. Utilizing comprehensive insights from sparse matrix graph structure analysis, we evaluated methods and tested the improved computational efficiency through parallelism, memory optimization, and enhanced convergence rate. The adjacency matrix visualization illustrates the structure and connectivity of the sparse matrix employed in our experiments. We also demonstrate the efficacy of our optimized SymGS in HPCG by solving large-scale sparse problems n = 1603 on KNL and 2883 on SKL using MPI+OpenMP configuration on multi-node environment. Our optimized SymGS demonstrates significant performance gains over the native implementation, achieving an average 1.4X improvement in HPCG on both KNL and SKL for 1, 2, 4, and 16 nodes. These results validate the efficiency of our approach in large-scale parallel computing, reinforcing its effectiveness in optimizing HPCG performance.

1. Introduction

The efficient solution of extensive sparse linear systems remains a critical challenge in scientific computing; as matrix size increases and architecture evolves, the complexity increases. In large-scale scientific computing, these systems are utilized in computational fluid dynamics, structural mechanics, reservoir simulation, and solvers for partial differential equations (PDEs) [1]. Among the iterative methods for solving $A \cdot x = b$, Symmetric Gauss-Seidel (SymGS) [2] has been a preferred option due to its simplicity and better convergence characteristics relative to the standard Jacobi method [3].

The High-Performance Conjugate Gradient (HPCG) benchmark [4, 5] was introduced to evaluate the performance of modern supercomputers on problems that better represent real-world high-performance computing (HPC) workloads than traditional dense benchmarks such as LINPACK [6, 7]. Within HPCG, the SymGS method plays a critical role as a smoother in the multigrid preconditioner, being invoked at each iteration to process the residual. Despite its advantages, the standard implementation of SymGS in HPCG, referred to as the 'reference SymGS' encounters significant performance bottlenecks in contemporary multi- and many-core architectures. This limitation stems from the inherent sequential nature of reference SymGS, which involves a forward sweep followed by a backward sweep through the matrix rows. Such sequential operations impede parallelism exploitation, thereby constraining performance scalability on modern processors where the disparity between compute capabilities and memory bandwidth widens. Addressing this issue necessitates the development of alternative parallelizable variants of SymGS that can optimize the utilization of modern and future computing platforms. In recent years, various novel technologies have been examined to enhance the parallel performance of SymGS. These methodologies, grounded in graph theory, data block decomposition, and spectral analysis, aim to incorporate parallelism into the iterative scheme while preserving the positive attributes of the convergence characteristics. Achieving equilibrium among these factors may necessitate periodic interaction between memory access patterns, computational demands, and numerical stability.

This work presents a comprehensive analysis of enhanced SymGS variants specifically tailored to overcome the limitations of reference implementation in multigrid preconditioners. Using insights from sparse matrix graph structures and analyzing the features of sparse matrix structures, we discover and incorporate efficiency improvements that play to exploit parallelism simultaneously as they reinforce the stability of the convergence properties. These enhancements are not simply arising incrementally; even a small-scale enhancement of the efficiency of the solver would significantly reduce the total computation time for large-scale scientific applications.

1.1 Motivation

HPCG, used to benchmark modern supercomputers, spends much time in SymGS due to its sequential nature. Parallel updates while maintaining Gauss-Seidel data dependencies are the biggest challenge. Each iteration requires partial results of neighboring unknowns. Optimizing SymGS can improve HPCG performance.

The principal strategies for enhancing the SymGS smoother:

- Enhancing Parallelism: Partitioning and graph coloring segment the matrix into autonomous subsets that can be processed simultaneously to reduce update conflicts.
- Enhancing Convergence: Spectral acceleration techniques can reduce convergence iterations, improving computational efficiency.

Despite numerous efforts to address these issues, balancing parallel performance and numerical stability remains challenging, particularly when implementing SymGS in large-scale, real-world applications. To tackle this, we simulate HPCG matrix and geometry structures and implement standalone SymGS algorithms to evaluate their efficiency.

1.2 Contribution

This section describes our contributions to improving the parallelism of SymGS in the HPCG benchmark:

- *Theoretical Framework and Pseudo-Code*: We provide a theoretical framework and pseudo-code algorithms for each variant to demonstrate implementation.
- *Geometry and Structural Analysis*: We extensively examine matrix geometry and structure, emphasizing domain partitioning and graph coloring for concurrency prospects.
- *SymGS variants*: We presented the Temporal Block, Multi-Color, Wavefront, Level Scheduling, Over-Relaxation, and Hybrid Jacobi-GS SymGS variants. Moreover, we also evaluated the Chebyshev-accelerated SymGS method, however, due to its consistent inferior performance, we have excluded it from this report to save space.
- *Experimental Evaluation*: We test these variants on a sparse matrix used in HPCG in the multi- node cluster environment of the Intel Xeon Phi Processor 7250, also known as Knights Landing (KNL), and the Intel Xeon Gold 6148 Scalable Skylake Processor (SKL) to assess their performance.
- *Parameter Optimization*: Based on problem attributes and hardware setups, we advise on block size, temporal steps, color count, Jacobi ratio, and over-relaxation factors.

These contributions address the performance bottlenecks of the reference SymGS implementation while maintaining robust convergence properties and improving HPCG scalability and efficiency on modern supercomputing platforms.

This report is organized as follows: Section 2 comprises background about HPCG and the SymGS method, establishing a foundation for our subsequent discussions. Section 3 offers a quantitative examination of geometric and matrix structures, which is crucial for comprehending optimization methodologies. Section 4 encapsulates our contributions, explaining the development and mathematical intricacies of SymGS variants and describing the specific SymGS variants we presented, whereas Section 5 outlines our methodology and Section 6 describes experimental results, findings, and analysis, providing profound insights into the influence of our work on HPC.

2. Background

2.1 High-Performance Conjugate Gradient (HPCG)

HPCG is one of the benchmarks designed within the last decade to measure the ability of supercomputers to work on computations and data access patterns, typically for scientific applications. However, compared to other benchmarks such as High-Performance LINPACK (HPL), which are based on a pure computational performance measure, HPCG measures the performance of the system using sparse matrices, memory access patterns, and communication patterns typical of real-life applications.

HPCG solving a symmetric positive definite (SPD) sparse linear system through a preconditioned conjugate gradient (PCG) approach. The benchmark comprises essential computational kernels, including sparse matrix-vector SpMV multiplication, vector updates, dot products, and SymGS. Many scientific computing applications proved HPCG to be a better and a relevant benchmark for real world scientific computing applications. This benchmark is designed to evaluate a memory subsystem and interconnects of different components of the supercomputer. By focusing on data access patterns and communication, HPCG offers insights into performance across a wide range of applications, especially those that are memory-bound rather than compute-bound. HPCG enhances conventional benchmarks such as HPL by offering a more thorough system performance assessment on modern scientific tasks. For more detailed information on the HPCG benchmark, please visit the official website [8].

2.2 Multigrid V-Cycle and 3D Grid Stencil in HPCG

HPCG utilizes a multigrid (MG) to precondition the conjugate gradient (CG) solver. The multigrid V-cycle is a systematic approach that shifts the problem to coarser grids to reduce errors across various scales and subsequently enhances the solution on finer grids. In the multigrid V-cycle, as shown in Fig 1, the grid is partitioned into smaller subdomains, allocated across multiple processes to facilitate parallel computation. Each subdomain is additionally preconditioned with SymGS to refine the solution before transitioning between grid levels. In conjunction with SymGS smoothing, the multigrid method enables HPCG to efficiently manage extensive sparse linear systems while optimizing computation and communication within contemporary HPC architectures.

The sparse matrix A utilized in HPCG is derived from a 27-point stencil in a three-dimensional grid discretization as shown in Fig 2. Every grid point is linked to 26 adjacent neighbors, comprising 6 face neighbors, 12 edge neighbors, and 8 corner neighbors.

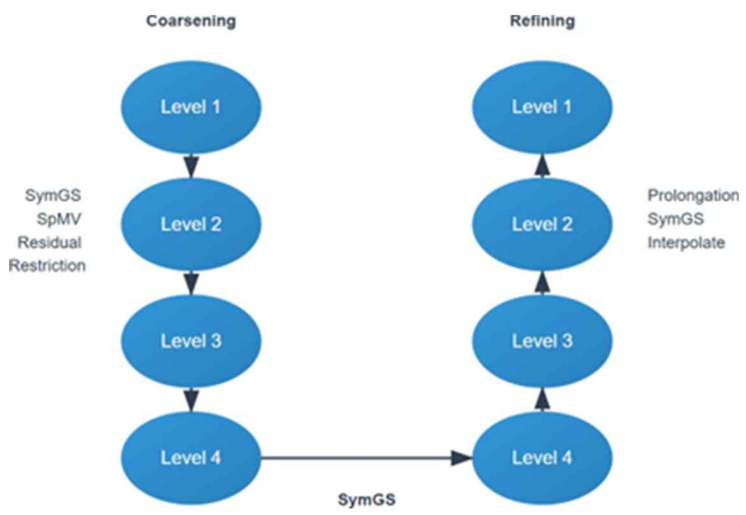


Figure 1: Multigrid V-cycle preconditioner in HPCG, illustrating the levels of coarsening and refining.

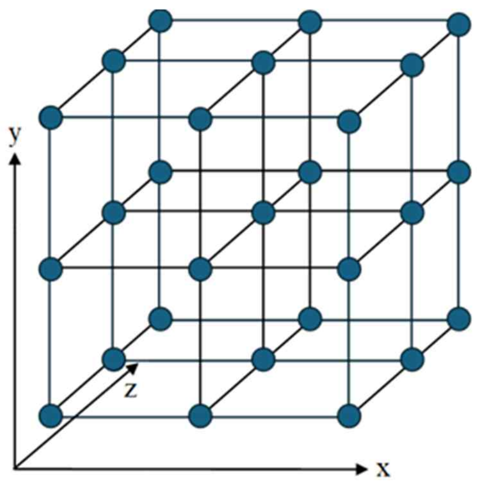


Figure 2: 27-point stencil in HPCG showing the grid points connected to neighbors.

2.3 Symmetric Gauss-Seidel Method in HPCG

The Symmetric Gauss-Seidel (SymGS) method is an effective iterative method for solving sparse linear systems of the form $A \cdot x = r$, where A is an $n \times n$ sparse symmetric positive definite (SPD) matrix, r is a residual vector, and x is the solution vector. Matrix A can be partitioned into three parts:

- L: The lower triangular section of A, comprising elements situated beneath the diagonal.
- U: The upper triangular section, comprising elements above the diagonal.
- D: The diagonal matrix comprising the diagonal elements of A.

The SymGS algorithm involves two sweeps:

- Forward Sweep: The forward sweep solves the system using the lower triangular part and diagonal, and can be expressed as: $(L + D)\, x(k+1) = r - U\, x(k)$
- Backward Sweep: The backward sweep refines the solution using the upper triangular part and diagonal, and can be expressed as: $(U + D)\, x(k+1) = r - Lx(k),$

where

- U is the upper triangular part of A,
- D is the diagonal vector of A,
- L is the lower triangular part of A,
- $x(k+1)$ is the updated solution vector at iteration $k + 1$,
- $x(k)$ is the solution vector from the previous iteration,
- r is the residual vector, calculated as $r = b - A \cdot x(k)$.

Within the HPCG benchmark, SymGS has the role of being smoother in the multigrid preconditioner. The first purpose is to attenuate the high-frequency error component in each iteration when applying the PCG method. Despite that, the forward-backward procedure of SymGS faces some data dependencies, hence a challenge to parallelization on modern multi and many-core systems. Typical implementations of SymGS face limitations to concurrency as a result of these dependencies, thereby rendering scalability issues. This method noticeably enhances convergence.

2.4 Challenges in Parallelizing SymGS

Forward-backward sweep in SymGS introduces inherent data dependencies, making it difficult to parallelize effectively. Due to these dependencies, the reference implementations of SymGS suffer from limited concurrency.

3. Geometry and Structure Analysis

The sparsity of the matrix significantly influences the computational efficiency of iterative solvers. For our experiments, we replicated a matrix structure similar to the HPCG benchmark. The matrix data are stored in the Compressed Sparse Row (CSR) format. The smallest problem size that HPCG supports is 163. The total number of nodes reaches 4,096 at this smallest size, resulting in a matrix with 4,096 rows. The matrix has a sparsity of 99.3%, as each row contains up to 27 non-zero elements. The banded diagonal characteristic of the sparse matrix is visible in the adjacency matrix, as shown in Fig 3. Near the diagonal entries, the matrix exhibits a densely interconnected structure. According to the graph analysis data, the node count is 4,096, the edge count is 46,620, the maximum degree is 26, and the average degree is approximately 22.76. During the graph analysis, we identified 1,352 dense subgraphs within the matrix. Such subgraphs also briefly denote local dependency, characterized by intertwined nodes and a high connection density. These dense regions can influence the performance of the iterative solvers because strongly coupled areas require additional time for sweeps because of dependencies. The complexity of the graph structure results from the high density of subgraphs, which suggests that specific components of the matrix necessitate additional computational effort or time. The computational overhead is augmented by the increased communication required by nodes of higher degree in parallel solver implementations. Dense subgraph dependencies can impede convergence by acting as bottlenecks in the iterative process.

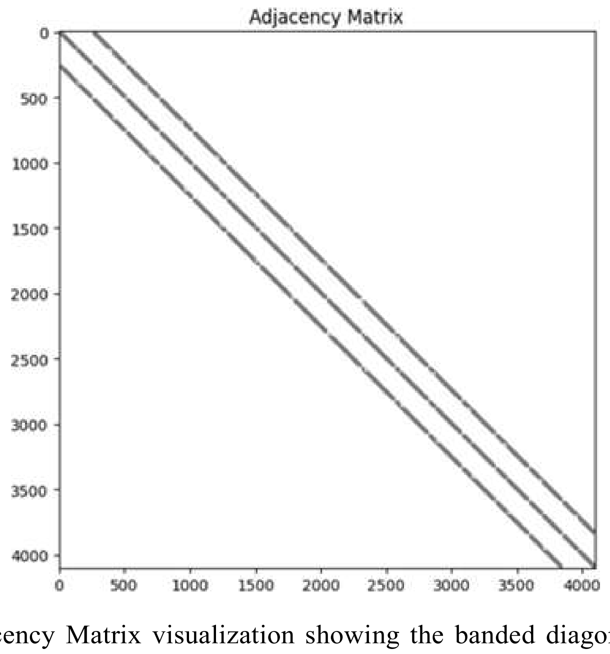


Figure 3: Adjacency Matrix visualization showing the banded diagonal pattern.

4. Algorithmic Variants of SymGS

In this section, we briefly explain each SymGS variant to highlight its key implementation idea and theoretical background.

4.1 Reference SymGS

Algorithm 1 is a standard symmetric Gauss-Seidel method, is referred to as the Reference SymGS as implemented in the HPCG, which sequentially solves during forward and backward sweeps. During the forward sweep, each row employs the updated values from the preceding rows and the initial values for the subsequent rows conversely, the backward sweep processes them in the reverse manner. This approach does not include data reordering and parallelization, serving as the baseline for comparison with other methods.

Algorithm 1 Symmetric Gauss-Seidel - Reference SymGS

```
Require: Matrix $A \in \mathbb{R}^{n \times n}$, right-hand side vector $\mathbf{r} \in \mathbb{R}^n$, initial guess $\mathbf{x} \in \mathbb{R}^n$
Ensure: Updated solution vector $\mathbf{x}$
 1: Forward Sweep:
 2: for $i = 1, 2, \ldots, n$ do
 3:     $x_i = \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
 4: end for
 5: Backward Sweep:
 6: for $i = n, n-1, \ldots, 1$ do
 7:     $x_i = \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
 8: end for
```

4.2 Multi-Color SymGS

Algorithm 2 Multi-Color SymGS

```
 1: Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, residual vector $r$, initial guess vector $x$
 2: Let $\mathcal{C} = \{C_1, C_2, \ldots, C_k\}$ be the set of color groups
 3: for $c = 1$ to $k$ do
 4:     Forward color sweep
 5:     for each $i \in C_c$ in parallel do
 6:         $x_i \leftarrow \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
 7:     end for
 8: end for
 9: for $c = k$ to 1 do
10:     Backward color sweep
11:     for each $i \in C_c$ in parallel do
12:         $x_i \leftarrow \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
13:     end for
14: end for
15: Output: Updated vector $x$
```

Algorithm 2 Multi-Color SymGS routine employs coloring to divide rows into independent sets, with the ability to process rows concurrently within a color. This helps assign colors to rows so that the adjacent rows have different colors, facilitating parallelism in forward and backward sweep. However, it depends on the number of colors needed and the structure of the matrix. This algorithm is highly effective when a multitude of colors is required, and the matrix structure is sparse.

4.3 Temporal Block SymGS

Algorithm 3 Temporal Block SymGS divides the matrix using a spatial block of size b, and then iteratively, updates are performed over the selected number of temporal steps s. These steps begin with the forward sweep from the first block through the last. In each block, dependent blocks are first updated, then updated by the rows of a current block using the Gauss-Seidel update, based on preceding rows updates and diagonal elements of the matrix. This ensures that each row is optimized using the values that have been recently obtained. After completing the forward sweep, the blocks are scanned from the last to the first block, and similar changes are made in reverse order. Each block performed several temporal iterations in both sweeps to ensure that in each block, the solution vector x is updated before moving to the next block. Using temporal blocks has allowed the algorithm to take advantage of parallelism and avoid unnecessary work with the matrix by focusing on changing the non-zero elements. The final solution vector is then used for residual check. This method improves convergence over standard Gauss-Seidel methods.

Algorithm 3 Temporal Block SymGS

```
Require: Sparse matrix $A \in \mathbb{R}^{n \times n}$, residual vector $r$, initial solution vector $x$, desired block size $b$, and temporal steps $s$
 1: Let $n$ be the number of rows in $A$
 2: Compute the number of spatial blocks: $m = \lceil n/b \rceil$
 3: Initialize block starts: block_starts$[i] = i \cdot b, \quad \forall i \in \{0, 1, \dots, m-1\}$
 4: Initialize block sizes: block_sizes$[i] = \min(b, n - \text{block\_starts}[i])$
 5: Determine dependencies for each block based on non-zero elements in $A$
 6: Let $l$ be the index representing non-zero elements in each row of $A$
 7: for each block $k$ from 0 to $m-1$ do                                  ▷ Forward Sweep
 8:     for each temporal step $t$ from 1 to $s$ do
 9:         for each dependent block $j$ in depBlocks$[k]$ do
10:             Update rows in block $j$ parallely:
                    $$x_j^{\text{new}} = \frac{r_j - \sum_l A_{jl} x_l^{\text{old}} + A_{jj} x_j^{\text{old}}}{A_{jj}}$$
11:         end for
12:         Update rows in block $k$ parallely:
                $$x_i^{\text{new}} = \frac{r_i - \sum_j A_{ij} x_j^{\text{old}} + A_{ii} x_i^{\text{old}}}{A_{ii}}, \quad \forall i \in \text{block}[k]$$
13:         Copy $x_i^{\text{new}}$ to $x_i^{\text{old}}$ for the next iteration
14:     end for
15: end for
16: for each block $k$ from $m-1$ to 0 do                                 ▷ Backward Sweep
17:     for each temporal step $t$ from 1 to $s$ do
18:         for each dependent block $j$ in depBlocks$[k]$ do
19:             Update rows in block $j$ in reverse order parallely:
                    $$x_j^{\text{new}} = \frac{r_j - \sum_l A_{jl} x_l^{\text{old}} + A_{jj} x_j^{\text{old}}}{A_{jj}}$$
20:         end for
21:         Update rows in block $k$ in reverse order parallely:
                $$x_i^{\text{new}} = \frac{r_i - \sum_j A_{ij} x_j^{\text{old}} + A_{ii} x_i^{\text{old}}}{A_{ii}}, \quad \forall i \in \text{block}[k]$$
22:         Copy $x_i^{\text{new}}$ to $x_i^{\text{old}}$ for the next iteration
23:     end for
24: end for
25: Copy $x^{\text{new}}$ to $x$ for residual check
26: return Updated vector $x$
```

4.4 Hybrid Jacobi-GS SymGS

Algorithm 4 Hybrid Jacobi-Gauss-Seidel integrates the parallelism of the Jacobi method with the accelerated convergence of Gauss-Seidel by partitioning the rows of a sparse matrix A into two groups according to the number of non-diagonal non-zeros. Rows with more connections are designated to the Jacobi set (J) and updated concurrently using a temporary vector. In contrast, the other rows constitute the Gauss-Seidel set (G) and are updated sequentially to leverage the most recent values for enhanced convergence. The algorithm initially executes a parallel Jacobi forward sweep on (J), subsequently con- ducts sequential Gauss-Seidel forward and backward sweeps on (G) and concludes with a parallel final Jacobi update.

The Jacobi ratio (α) governs the division between the two methods. The value of α induces parallelism by augmenting the number of Jacobi rows, thereby enhancing performance on parallel architectures while at the expense of the convergence rate. The value of α within the optimal range improves convergence with additional Gauss-Seidel iterations but reduces the parallelism due to sequential updates. The algorithm assigns weights to these methods to leverage contemporary concurrent computing capabilities, while maintaining favorable convergence properties and being optimized for large sparse systems.

Algorithm 4 Hybrid Jacobi-Gauss-Seidel

```
1: Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, residual vector $r \in \mathbb{R}^n$, initial guess $x \in \mathbb{R}^n$, Jacobi ratio $\alpha$
2: Output: Updated solution vector $x$
3: $n \leftarrow$ number of rows in $A$
4: $c_i \leftarrow$ number of non-diagonal nonzeros in row $i$ of $A$ for $i = 1, \dots, n$
5: Sort rows by $c_i$ in descending order to obtain permutation $\pi$
6: $m \leftarrow \lfloor \alpha \times n \rfloor$
7: $\mathcal{J} \leftarrow \{\pi(1), \pi(2), \dots, \pi(m)\}$                ▷ Indices for Jacobi updates
8: $\mathcal{G} \leftarrow \{\pi(m+1), \pi(m+2), \dots, \pi(n)\}$        ▷ Indices for Gauss-Seidel updates
9: $x_{\text{temp}} \leftarrow x$
10: Parallel execution across Jacobi indices
11: for all $i \in \mathcal{J}$ do                                       ▷ Jacobi forward sweep
12:     $s \leftarrow r_i - \sum_{j \neq i} A_{ij} x_j$
13:     $x_{\text{temp},i} \leftarrow \frac{s}{A_{ii}}$
14: end for
15: for all $i \in \mathcal{G}$ do                                       ▷ Gauss-Seidel forward sweep
16:     $s \leftarrow r_i - \sum_{j<i} A_{ij} x_{\text{temp},j} - \sum_{j>i} A_{ij} x_j$
17:     $x_i \leftarrow \frac{s}{A_{ii}}$
18: end for
19: for all $i \in \mathcal{G}$ in reverse order do                      ▷ Gauss-Seidel backward sweep
20:     $s \leftarrow r_i - \sum_{j<i} A_{ij} x_j - \sum_{j>i} A_{ij} x_{\text{temp},j}$
21:     $x_i \leftarrow \frac{s}{A_{ii}}$
22: end for
23: Parallel execution across Jacobi indices
24: for all $i \in \mathcal{J}$ do                                       ▷ Final Jacobi update
25:     $x_i \leftarrow x_{\text{temp},i}$
26: end for
```

4.5 Over Relaxation SymGS

Algorithm 5 Over Relaxation SymGS

```
Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, right-hand side vector $r \in \mathbb{R}^n$, initial guess $x \in \mathbb{R}^n$
Output: Updated solution vector $x$
Ensure that the length of vector $x$ matches the number of columns in matrix $A$
$n \leftarrow$ number of rows in $A$
$\omega \leftarrow$ over-relaxation factor
Initialize a temporary vector $x^{\text{temp}} \leftarrow x$
Forward Sweep:
for $i = 1$ to $n$ in parallel do
    $s \leftarrow r_i - \sum_j A_{ij} x_j$
    $x_i^{\text{temp}} \leftarrow \frac{s + x_i A_{ii}}{A_{ii}}$
    $x_i^{\text{temp}} \leftarrow x_i + \omega \left( x_i^{\text{temp}} - x_i \right)$
end for
$x \leftarrow x^{\text{temp}}$
Backward Sweep:
for $i = n$ to $1$ in parallel do
    $s \leftarrow r_i - \sum_j A_{ij} x_j$
    $x_i^{\text{temp}} \leftarrow \frac{s + x_i A_{ii}}{A_{ii}}$
    $x_i^{\text{temp}} \leftarrow x_i + \omega \left( x_i^{\text{temp}} - x_i \right)$
end for
$x \leftarrow x^{\text{temp}}$
return $x$
```

Algorithm 5 Over-Relaxation SymGS concept inherited from the Jacobi Over Relaxation approach which uses Jacobi updates by over-relaxation to improve the convergence rate of the Jacobi method. However, we employed a forward and backward sweep to enhance the solution vector *x* by utilizing previously known values alongside newly computed values. The algorithm computes in each row and stores in a temporary vector instead of updating the resultant vector *x*. The temporary update is subsequently over-relaxed with the assistance of the factor ω, which facilitates dependency resolution. The forward sweep executes row processing in parallel from 1 to *n*, whereas the backward sweep processes rows in reverse order from *n* to 1, also in parallel.

Excessive relaxation modifies the update as $x_i^{temp} = x_i + \omega\, (x^{temp} - x_i)$, which seeks to expedite error reduction by utilizing both temporary and current values. This method enhances parallel execution, rendering it appropriate for parallel computing architectures.

4.6 Wavefront SymGS

Algorithm 6 Wavefront Symmetric Gauss-Seidel (SymGS) Method

```
1: Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, right-hand side vector $r \in \mathbb{R}^n$, initial guess $x \in \mathbb{R}^n$
2: Output: Updated solution vector $x$
3: Initialize $x_{\text{old}} \leftarrow x$, $x_{\text{new}} \leftarrow x$
4: Define wavefronts $W_k$ for $k \in [0, nx + ny + nz - 2]$
5: Forward Wavefront Update:
6: for $k = 0$ to num_waves $- 1$ do
7:     for each row $i \in W_k$ (in parallel) do
8:         $s \leftarrow r_i - \sum_j A_{ij} x_{\text{old},j}$
9:         $s \leftarrow s + A_{ii} x_{\text{old},i}$
10:        $x_{\text{new},i} \leftarrow \frac{s}{A_{ii}}$
11:    end for
12:    $x_{\text{old}} \leftarrow x_{\text{new}}$
13: end for
14: Backward Wavefront Update:
15: for $k =$ num_waves $- 1$ to $0$ do
16:     for each row $i \in W_k$ (in parallel) do
17:         $s \leftarrow r_i - \sum_j A_{ij} x_{\text{old},j}$
18:         $s \leftarrow s + A_{ii} x_{\text{old},i}$
19:         $x_{\text{new},i} \leftarrow \frac{s}{A_{ii}}$
20:     end for
21:     $x_{\text{old}} \leftarrow x_{\text{new}}$
22: end for
23: Copy Updated Solution:
24: for $i = 1$ to $n_{\text{row}}$ (in parallel) do
25:     $x_i \leftarrow x_{\text{new},i}$
26:     $x_{\text{old},i} \leftarrow x_{\text{new},i}$
27: end for
28: Return: Updated vector $x$
```

Algorithm 6 Wavefront SymGS utilizes the wavefronts to organize and schedule the updates in a Gauss-Seidel iteration. The wavefront technique uses each coordinate point *(x, y, z)* in a three-dimensional grid *nx* × *ny* × *nz* has a 'wave index' derived from the sum x + y + z. The wavefront consists of all points (or rows) on the same diagonal plane *(x + y + z = constant).* This means that all points (or rows) that lie on the same plane diagonal *(x + y + z = constant)* form a wavefront. We group points with the same sum into the same wave so that all neighbors of a point are in earlier or later waves, allowing concurrent updates within each wave. We loop over every valid (x, y, z), compute the row index *idx = z · (nx · ny) + y · nx + x,* and place this index into a local buffer corresponding to *wave = x + y + z*. These local buffers are then merged in a thread-safe manner to form the final waves array, where waves[k] contain all row indices whose coordinates sum to *k*. Once these wavefronts are defined, the Wavefront SymGS algorithm updates the solution vector x in two sweeps, forward (from 0 to the highest wave) and backward (from the highest wave to 0), using double buffering to ensure numerical accuracy. For each wave Wk, rows are updated in parallel by computing s = ri − Σ Aijxold,j, adding back the diagonal part Aiixold,i, and dividing by Aii to get the new xnew,i To refresh subsequent waves, updated values are copied into xold at the end of each wave. The backward sweep reverses the wave order but uses the same update strategy to respect the latest updated neighbors using the Gauss-Seidel operation in both directions for improved convergence while taking advantage of wavefront parallelism. This arrangement ensures that neighboring information has been computed in previous wavefronts when a row in one wavefront is updated. These wavefronts can be navigated sequentially to execute the forward Gauss- Seidel method concurrently within each diagonal segment, which results in a scalable SymGS approach that exploits parallelism yet remains true to Gauss-Seidel's sequential dependency requirements, making it an effective iterative solver for large, and sparse 3D problems.

4.7 Level-Scheduled SymGS

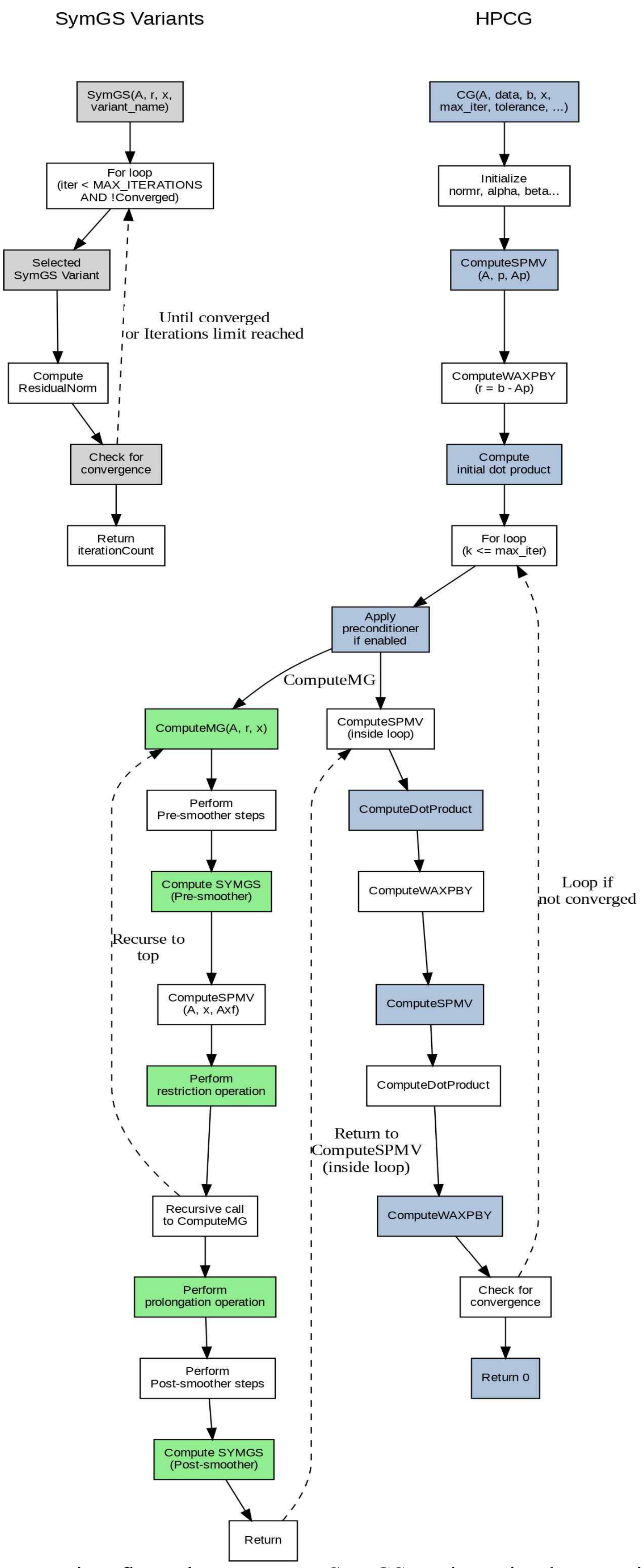
Algorithm 7 Level-Schedule SymGS Algorithm

```
Input: Sparse matrix $A \in \mathbb{R}^{n \times n}$, residual vector $\mathbf{r} \in \mathbb{R}^n$, initial guess vector $\mathbf{x} \in \mathbb{R}^n$
Output: Updated solution vector $\mathbf{x}$
Initialize: $\mathbf{L} \leftarrow \emptyset$, num_levels $\leftarrow 0$
Compute in-degrees for each node:
    $d(i) = \sum_{j=1}^{n} \mathbb{I}(A_{ij} \neq 0 \land j > i), \quad \forall i \in [1, n]$
Identify the initial level (no dependencies):
    $L_0 = \{i \mid d(i) = 0\}$
    $\mathbf{L} \leftarrow \mathbf{L} \cup \{L_0\}$, num_levels $\leftarrow 1$
while $L_k \neq \emptyset$ do
    $L_{k+1} \leftarrow \emptyset$
    for all $i \in L_k$ do
        for all $j$ such that $A_{ij} \neq 0 \land j > i$ do
            $d(j) \leftarrow d(j) - 1$
            if $d(j) = 0$ then
                $L_{k+1} \leftarrow L_{k+1} \cup \{j\}$
            end if
        end for
    end for
    $\mathbf{L} \leftarrow \mathbf{L} \cup \{L_{k+1}\}$
    num_levels $\leftarrow$ num_levels + 1
end while
Forward Sweep
for $k = 0$ to num_levels - 1 do
    for all $i \in L_k$ in parallel do
        $x_i \leftarrow \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
    end for
end for
Backward Sweep
for $k =$ num_levels $- 1$ to 0 do
    for all $i \in L_k$ in parallel do
        $x_i \leftarrow \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
    end for
end forreturn $\mathbf{x}$
```

Algorithm 7 Level-Scheduled SymGS starts by creating a dependency graph of the matrix *A*, where the row number is the node and the dependencies are given as non-zero entries in the matrix A. It then computes for the in-degrees, and these are actually the number of other rows that it depends on for calculation. When there is no dependency on its other nodes on a node, that node will belong to the initial level L0, and it can be processed in parallel with others. Subsequent levels are created by considering all the dependent nodes and lowering their in-degrees throughout progressive levels until all nodes are placed in a level.

Once the level schedule is developed, the algorithm executes a forward sweep, and a backward sweep is done subsequently to modify solution vector x. During the forward sweep, it processes the level of the node by level, updating the value of each variable xi using the formula: $x_i \leftarrow r_i - \Sigma_j A_{ij} x_j + A_{ii} x_i / A_{ii}$.

5. Methodology



SymGS Variants
HPCG
SymGS(A, r, x, variant_name)
For loop (iter < MAX_ITERATIONS AND !Converged)
Selected SymGS Variant
Until converged or Iterations limit reached
Compute ResidualNorm
Check for convergence
Return iterationCount
CG(A, data, b, x, max_iter, tolerance, ...)
Initialize normr, alpha, beta...
ComputeSPMV (A, p, Ap)
ComputeWAXPBY (r = b - Ap)
Compute initial dot product
For loop (k <= max_iter)
Apply preconditioner if enabled
ComputeMG
ComputeMG(A, r, x)
ComputeSPMV (inside loop)
Perform Pre-smoother steps
ComputeDotProduct
Compute SYMGS (Pre-smoother)
ComputeWAXPBY
Loop if not converged
Recurse to top
ComputeSPMV (A, x, Axf)
ComputeSPMV
Perform restriction operation
ComputeDotProduct
Return to ComputeSPMV (inside loop)
Recursive call to ComputeMG
ComputeWAXPBY
Perform prolongation operation
Check for convergence
Return 0
Perform Post-smoother steps
Compute SYMGS (Post-smoother)
Return

Figure 4: Diagram of execution flows between our SymGS variants implementation and the HPCG iterative solver, emphasizing distinctions in preconditioning and iterative methodologies.

In this study, we simulated the matrix and vector structures from the HPCG benchmark and developed a standalone implementation to assess different variants of the SymGS. The distinction in execution flow between the iterative solver in our standalone implementation and HPCG is illustrated in Fig. 4, with our standalone implementation referred to as SymGS variants, alongside the HPCG iterative solver, which is a component of the HPCG benchmark. Both methods employ iterative processes. However, their convergence criteria, preconditioning techniques, and operations vary in the context mentioned.

The execution of the SymGS variants stated in Section 4 commences by invoking a function that accepts *A* as the matrix, *r* as the residual vector, *x* as the solution vector, and variant name as the specific variant in use. To achieve this, the algorithm initiates a for loop that continues until a predetermined maximum number of iterations is reached or convergence is attained. Each iteration entailed selecting a variant of SymGS that modifies the solution vector, and the norm of a residual was calculated to assess convergence. Upon convergence, the function yields the total number of iterations executed. Alternatively, it persists indefinitely and ceases upon the fulfillment of the termination conditions.

In HPCG, it initially invokes CG to initiate the process, as illustrated on the right side of the Fig 4. In HPCG, *A* denotes the matrix, data refers to the CG data specific to HPCG, *r* represents the right-hand side vector, *x* signifies the solution vector, max iter indicates the maximum iterations for convergence, and tolerance is the criterion for convergence. The algorithm begins by defining α, β, and the norm of the residual vector, denoted as *normr*. This is followed by a SpMV and dot product to compute the initial estimate of the residual. The HPCG flow involves a for loop that executes as a calculation loop until convergence is achieved or the maximum number of iterations is reached. The solver preconditions within the loop. The ComputeMG routine, which employs a multigrid (MG) method for preconditioning convergence, is distinctive. HPCG employs a four-level multigrid preconditioner, and this procedure is multi-step. SymGS pre-smoothers are employed to mitigate finer grid errors. A sparse matrix-vector product update is subsequently executed. The residual is confined to a coarser grid and recursively invoked, continuing this process on progressively coarser grids until the coarsest level is attained. Following coarse-grid correction, the solution is extended to finer grids, and SymGS post-smoothing steps are implemented to mitigate errors. Subsequently, CG invoked additional kernels to revise the solution vector and residual. The SymGS variants and the HPCG solver primarily differ in their iterative process architecture and preconditioning methods. HPCG employs a recursive multigrid method incorporating pre- and post-smoothing steps to enhance convergence. Our independent implementation employs a linear iterative framework. During the evaluation of our implemented SymGS within HPCG, we substituted the pre-smoother and post-smoother SymGS with our optimized version of SymGS.

Subsequently, we assess the performance of each standalone SymGS variant by evaluating the setup time, computation time, computation time per iteration in seconds, and iterations required for convergence. We evaluated the performance of these variants compared to the reference SymGS implementation and examined the performance of the computed solution using three different metrics: Relative Error, Root Mean Square Error (RMSE), and Mean Absolute Error (MAE), to measure the difference between the resultant vector after computation and the reference SymGS resultant vector.

We evaluated these variants based on various parameters and subsequently selected the most effective for use in HPCG. In HPCG, we substituted the pre- and post-smoother SymGS with our implemented SymGS and assessed the performance of HPCG. The experiments and a detailed discussion of the results are presented in Section 6.

6. Experiments and Results

In this section, we presented the performance results. We conducted experiments on Intel Xeon Phi 7250 (KNL) and Intel Xeon Gold 6148 (SKL) processors. On KNL we used *-xMIC-AVX512*, whereas SKL utilizes *-xCORE-AVX512*, to optimize for the AVX-512 instruction set tailored for the Many Integrated Core (MIC) architecture. Additional flags comprise *-O3, -qopt-prefetch=0, -qopenmp*, and *-std=c++17*. The mpiicpc compiler is utilized for MPI-based execution.

6.1 Performance metrics

SymGS variants performance was evaluated using various metrics. The following metrics were used to evaluate performance and compare the variants with the reference SymGS implementation:

- Setup (s): Time spent on pre-processing or setup before computational iterations.
- Compute (s): Time spent in the main routine to perform computations.
- Total (s): Total execution time, calculated as: Total Time = Setup + Compute
- Avg/Iter (s): Average time per iteration, computed as:
- Avg/Iter (s) = Compute Time / Iterations
- Iterations: Total number of iterations performed until convergence or meeting the stopping criteria.
- Rel. Error, RMSE, MAE: Error metrics, for measuring the difference between the resultant vector after computation and the reference SymGS resultant vector.
- Gflops: Performance in gigaflop operations per second.
- Speedup: Ratio of runtime improvement compared to the reference SymGS:
- Speedup = Total Time$_{\text{(reference)}}$ / Total Time$_{\text{(method)}}$
- Improvement: Increase in Gflops compared to the baseline reference SymGS.

Our study analyzed various SymGS variants, each utilizing different optimization parameters. The tested parameter ranges were as follows:

- MultiColor(MC)-*c*: $c \in \{2, 4, 6, 8\}$
- TemporalBlocking(TB)-*a*-*b*: $a, b \in \{2, 4, 8, \ldots, 64\}$
- Hybrid Jacobi GS: $j_r \in \{0.5, 0.6, \ldots, 1.0\}$
- OverRelaxation-*ω*: $\omega \in \{0.2, 0.4, \ldots, 1.4\}$

The parameter ranges varied based on the variance that differentiates one variant from another. The MultiColor-*c* values corresponded to 2, 4, 6 and 8, started from 2 and increased by a factor of 2, where *c* is the number of colors used for the multi-color SymGS. For TemporalBlocking-*a*-*b*, in our experiments, *a* is the number of spatial blocks, which divides the total number of rows into blocks, and *b* is the number of temporal steps used within each iteration/block. *a* and *b* were starting from 2 and considered as powers of two within the range *a, b* in [2, 64]. The Hybrid Jacobi GS evaluated Jacobi ratios (j_r) in increments of 0.1, ranging from 0.5 to 1.0. In OverRelaxation-*ω*, the overrelaxation parameter omega (*ω*) was adjusted from 0.2 to 1.4 in increments of 0.2.

6.2 Results on Intel Knights Landing (KNL)

6.2.1 SymGS variants

We assessed the performance of Symmetric Gauss–Seidel (SymGS) variants on an Intel Xeon Phi multi-core processor featuring 68 cores, concentrating on various problem sizes associated with matrix dimensions 16^3, 32^3, 64^3, and 96^3. We utilized a single MPI process, leveraging all 68 threads, and for problem sizes up to 64^3, established the SymGS tolerance at 1×10^{-8}. For larger problem sizes ($> 64^3$), the tolerance was adjusted to 1×10^{-6} to decrease computation time, as more strict tolerance criteria would necessitate additional iterations for convergence. Only the most promising parameter combinations that yielded some significant performance considered for discussion were included in Table 1, while less effective configuration results were omitted for the sake of clarity and conciseness.

The Multi Color SymGS variant performance results indicate that the setup time is high for 2 and 6 colors and relatively reduced when utilizing 8 colors. A similar pattern was observed in larger problem sizes, where 2, 4, and 6 colors exhibit greater setup times compared to 8 colors. Nonetheless, for small problem sizes, the computation time is reduced by using 4 colors in comparison to 8 colors. The number of iterations is concurrently increasing across 8 colors for all problem sizes. Though, the computational time for 4 colors is lower than that for 8 colors; however, the total time, inclusive of setup time, increases with the increase in problem size. Furthermore, the overall performance in Gflops improves with 8 colors for larger problem sizes, as the setup time also increases with increasing problem sizes when utilizing fewer than 8 colors. The reason for this is that the eight colors are evenly allocated across all rows in disjoint subsets; however, with fewer colors, some inter-dependent rows share identical colors. Despite attempts to recolor neighboring rows, certain dependent rows still retain the same color, ultimately impairing performance on larger problem sizes. Gflops performance increases with larger problem sizes, especially for MultiColor-8. In comparison to the reference SymGS, Multi Color SymGS exhibits an increase in the number of iterations and setup time; however, the average time per iteration is significantly reduced, resulting in enhanced overall performance relative to the reference SymGS.

The Temporal Block SymGS approach was evaluated using different configurations compared to the reference SymGS. Temporal Block variants significantly reduce the compute time, with the total time decreasing as the number of temporal steps increases. For example, TB-2-64 attains the minimal total time and converges in the fewest iterations. The Temporal Blocking approach significantly decreases the iterations needed for convergence. At the problem size of 32^3, the reference SymGS necessitates 487 iterations, whereas TB-2-64 requires merely 8 iterations, demonstrating its efficacy in enhancing computational efficiency. Although the average time per iteration increases slightly with more temporal blocks. In terms of performance, Temporal Blocking variants achieved remarkable gains in Gflops, with TB-2-2 delivering a 2.6X improvement over the reference SymGS. Additionally, the speedup achieved increases with the number of blocks, reaching up to 29 for TB-2-64, making it the most effective variant among those tested in terms of total computational time.

Wavefront SymGS performance remains good in mid-sized problems but is inferior to the reference SymGS in both small and large problem sizes. The Level Scheduled SymGS demonstrates improvement in performance at a mid-sized problem and surpasses the reference SymGS. The Hybrid Jacobi GS SymGS consistently performs better with $j_r = 0.9$ across all problem sizes. OverRelaxation SymGS performance varies but relatively remains good at at $\omega = 0.8$, 1.0, and 1.4; however, it fails to satisfy HPCG convergence criteria when the diagonal entries of the matrix are exaggerated and scaled to 10^6 not achieving convergence within two iterations, except for $\omega = 1.0$. Consequently, OverRelaxation is confined to $\omega = 1.0$ to adhere to HPCG constraints. Among the evaluated routines, Temporal Block SymGS exhibits the greatest speedup and optimal computation times across all problem sizes, with satisfactory performance improvement in Gflops. MultiColor variants demonstrate satisfactory performance; however, they are typically surpassed by Temporal Block SymGS in larger problems. Overall MultiColor, Temporal Blocking, and OverRelaxation SymGS variants demonstrate superior performance in Gflops as compared to other variants, as shown in Fig. 5.

Table 1: Performance comparison across SymGS variants for different problem sizes.

Variant	Setup(s)	Compute(s)	Total(s)	Avg/Iter(s)	Iterations	Rel. Error	RMSE	MAE	Gflops	Speedup	Improvement
Problem Size: 16^3, Total number of rows: 4,096											
Reference	0.0000	0.4341	0.4341	0.0033	133	0.00E+00	0.00E+00	0.00E+00	0.1206	1.0000	1.0000
MultiColor-2	0.0387	0.3934	0.4322	0.0020	193	5.02E-08	8.60E-10	6.59E-10	0.1757	1.0044	1.4575
MultiColor-4	0.0299	0.2550	0.2850	0.0014	180	2.47E-08	4.20E-10	3.03E-10	0.2485	1.5231	2.0613
MultiColor-6	0.0235	0.2757	0.2992	0.0015	183	2.23E-08	3.77E-10	2.65E-10	0.2406	1.4506	1.9959
MultiColor-8	0.0016	0.3156	0.3172	0.0016	199	3.39E-08	5.82E-10	4.34E-10	0.2468	1.3684	2.0474
TemporalBlocking-2-2	0.0031	0.1497	0.1528	0.0023	64	6.37E-08	1.05E-09	8.36E-10	0.1648	2.8416	1.3674
TemporalBlocking-2-16	0.0030	0.0649	0.0679	0.0081	8	6.37E-08	1.05E-09	8.36E-10	0.0463	6.3898	0.3843
TemporalBlocking-2-64	0.0031	0.0555	0.0586	0.0278	2	6.37E-08	1.05E-09	8.36E-10	0.0134	7.4075	0.1114
Hybrid_Jacobi_GS-0.9	0.0015	0.7852	0.7867	0.0015	511	7.14E-08	1.19E-09	9.46E-10	0.2556	0.5518	2.1199
OverRelaxation-0.2	0.0000	1.8179	1.8179	0.0014	1296	8.49E-08	1.40E-09	1.12E-09	0.2805	0.2388	2.3266
OverRelaxation-0.8	0.0000	0.4584	0.4584	0.0014	321	6.92E-08	1.14E-09	9.09E-10	0.2755	0.9470	2.2855
OverRelaxation-1.0	0.0000	0.3607	0.3607	0.0014	256	6.37E-08	1.05E-09	8.36E-10	0.2792	1.2034	2.3162
OverRelaxation-1.4	0.0000	0.2595	0.2595	0.0014	182	3.53E-08	5.65E-10	4.53E-10	0.2760	1.6728	2.2892
Wavefront	0.0080	1.0243	1.0323	0.0064	161	4.04E-08	6.66E-10	5.31E-10	0.0614	0.4205	0.5090
LevelScheduled	0.0029	0.8941	0.8971	0.0067	133	0.00E+00	0.00E+00	0.00E+00	0.0583	0.4838	0.4838
Problem Size: 32^3, Total number of rows: 32,768											
Reference	0.0000	13.8172	13.8172	0.0284	487	0.00E+00	0.00E+00	0.00E+00	0.1183	1.0000	1.0000
MultiColor-2	2.3610	6.3941	8.7551	0.0088	727	2.73E-07	5.97E-09	4.55E-09	0.2786	1.5782	2.3559
MultiColor-8	0.0125	6.9822	6.9947	0.0093	749	2.59E-07	5.68E-09	4.32E-09	0.3593	1.9754	3.0381
TemporalBlocking-2-2	0.0245	2.5850	2.6095	0.0107	241	3.90E-08	8.31E-10	6.38E-10	0.3099	5.2949	2.6203
TemporalBlocking-2-16	0.0242	0.6527	0.6770	0.0211	31	7.22E-07	1.58E-08	1.20E-08	0.1536	20.4108	1.2992
TemporalBlocking-2-64	0.0242	0.4458	0.4700	0.0557	8	1.22E-06	2.66E-08	2.03E-08	0.0571	29.3996	0.4830
Hybrid_Jacobi_GS-0.9	0.0116	22.6729	22.6845	0.0118	1926	5.47E-08	1.20E-09	9.12E-10	0.2849	0.6091	2.4089
OverRelaxation-0.8	0.0000	11.6420	11.6420	0.0097	1206	4.29E-08	9.15E-10	7.02E-10	0.3476	1.1868	2.9391
OverRelaxation-1.0	0.0000	9.4386	9.4386	0.0098	964	3.90E-08	8.31E-10	6.38E-10	0.3427	1.4639	2.8977
OverRelaxation-1.4	0.0000	6.7409	6.7409	0.0098	687	5.22E-08	1.12E-09	8.58E-10	0.3419	2.0498	2.8916
Wavefront	0.0023	11.2886	11.2909	0.0189	597	1.04E-08	2.07E-10	1.58E-10	0.1774	1.2237	1.5002
LevelScheduled	0.0255	9.7603	9.7858	0.0200	487	0.00E+00	0.00E+00	0.00E+00	0.1670	1.4120	1.4120
Problem Size: 64^3, Total number of rows: 262,144											
Reference	0.0000	440.4997	440.4997	0.2352	1873	0.00E+00	0.00E+00	0.00E+00	0.1178	1.0000	1.0000
MultiColor-8	0.1034	201.6544	201.7578	0.0695	2903	8.65E-07	2.56E-08	1.91E-08	0.3985	2.1833	3.3840
TemporalBlocking-2-2	0.1964	77.1227	77.3191	0.0826	934	2.47E-08	7.16E-10	5.37E-10	0.3346	5.6972	2.8410
Hybrid_Jacobi_GS-0.9	0.1195	633.7727	633.8922	0.0848	7470	2.53E-08	7.86E-10	5.71E-10	0.3264	0.6949	2.7715
OverRelaxation-0.8	0.0000	318.7694	318.7694	0.0682	4671	2.75E-08	7.98E-10	5.98E-10	0.4059	1.3917	3.4706
OverRelaxation-1.4	0.0000	180.6020	180.6020	0.0677	2667	3.47E-08	1.01E-09	7.58E-10	0.4090	2.4391	3.4730
Wavefront	0.0134	205.6821	205.6956	0.0893	2303	3.20E-09	6.47E-11	4.88E-11	0.3101	2.1415	2.6332
LevelScheduled	0.2711	172.6534	172.9245	0.0922	1873	0.00E+00	0.00E+00	0.00E+00	0.3000	2.5474	2.5474
Problem Size: 96^3, Total number of rows: 884,736											
Reference	0.0000	2574.5603	2574.5603	0.8289	3106	0.00E+00	0.00E+00	0.00E+00	0.1140	1.0000	1.0000
MultiColor-4	1339.0005	1079.5646	2418.5651	0.2479	4355	1.15E-04	4.12E-06	3.05E-06	0.1701	1.0645	1.4926
MultiColor-6	977.2436	1100.5016	2077.7452	0.2466	4463	1.20E-04	4.33E-06	3.19E-06	0.2029	1.2391	1.7805
MultiColor-8	0.3492	1168.2227	1168.5719	0.2415	4837	1.62E-04	5.80E-06	4.29E-06	0.3910	2.2032	3.4310
TemporalBlocking-2-2	0.6584	429.4003	430.0587	0.2769	1551	2.60E-06	9.19E-08	6.83E-08	0.3407	5.9865	2.9894
TemporalBlocking-2-16	0.6520	140.8085	141.4605	0.7258	194	5.97E-06	2.14E-07	1.58E-07	0.1295	18.1999	1.1368
Hybrid_Jacobi_GS-0.9	0.4621	3605.4660	3605.9281	0.2908	12397	5.21E-06	1.46E-07	1.05E-07	0.3247	0.7095	2.8320
OverRelaxation-1.0	0.0000	1931.9599	1931.9599	0.3114	6204	2.60E-06	9.19E-08	6.83E-08	0.3033	1.3363	2.6691
Wavefront	0.1024	4241.8401	4241.9425	1.1104	3820	3.72E-06	1.32E-07	9.82E-08	0.0851	0.6069	0.7464
LevelScheduled	0.9873	2220.6792	2221.6666	0.7150	3106	0.00E+00	0.00E+00	0.00E+00	0.1321	1.1588	1.1588

We used the above-discussed SymGS variants in HPCG and then evaluated the performance. These SymGS variants are configured within HPCG as follows:

SymGS variants used in HPCG:

- MultiColor with c = 8.
- Temporal Blocking with a = 2, b = 2.
- Hybrid Jacobi GS with j_r = 0.9.
- OverRelaxation with ω = 1.0.

Performance results for these variants are shown in Fig 6.

Fig 6 illustrates a comparison of HPCG results based on the SymGS variants across different problem sizes, ranging from 64^3 to 192^3. The reference executes the native HPCG without any optimization. We examined the different variants of SymGS in HPCG and assessed their performance compared to the reference.

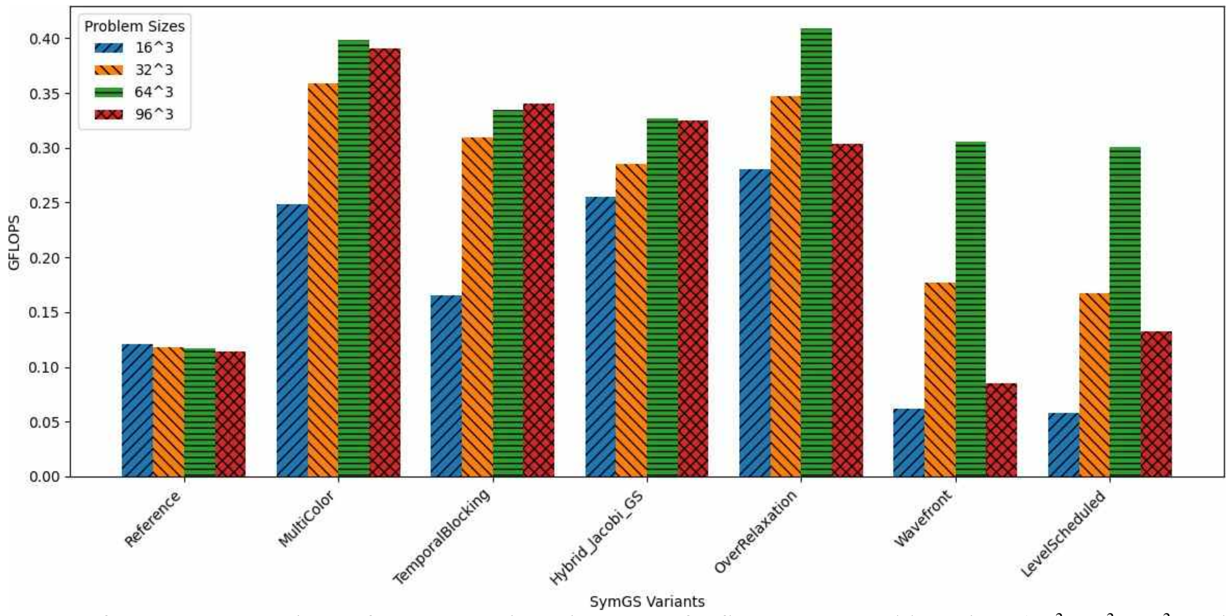


Figure 5: Performance comparison of SymGS variants in terms of Gflops across problem sizes (16^3, 32^3, 64^3, and 96^3).

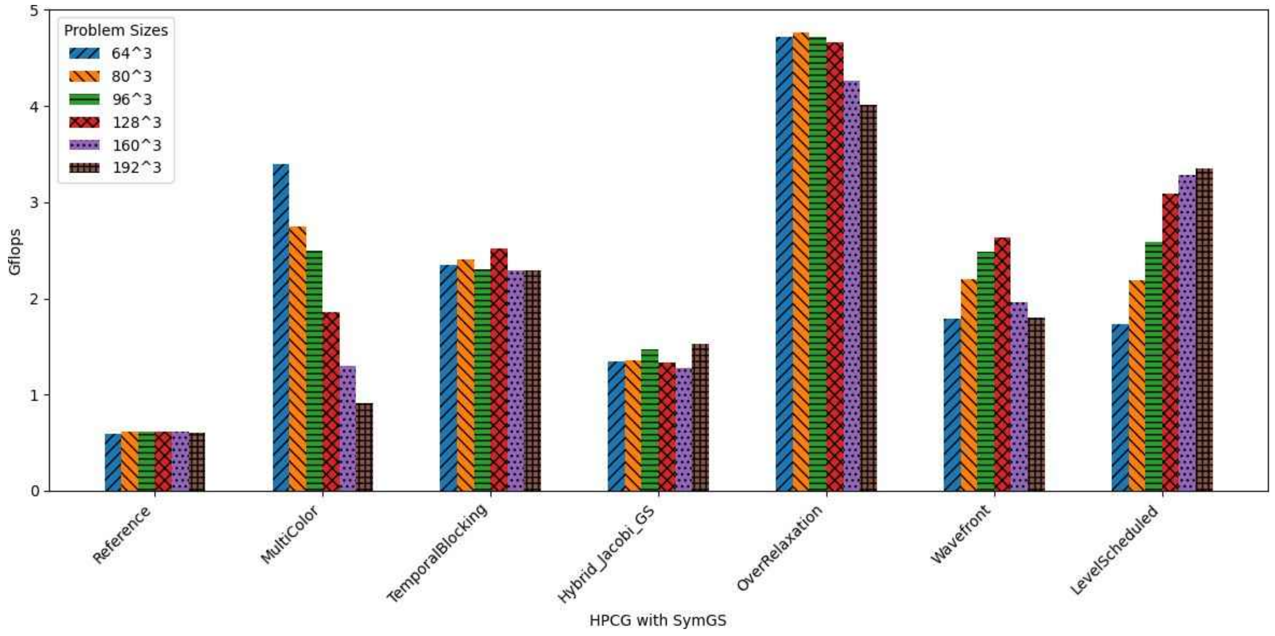


Figure 6: Performance of HPCG using the different SymGS variants across different problem sizes (64^3 to 192^3).

6.2.2 HPCG Results

The HPCG with the Overrelaxation SymGS variant is regarded as the most effective, achieving an average of 7X performance improvement across all tested problem sizes. However, its performance reduced slightly with larger problem sizes. Conversely, Level Scheduled exhibits entirely different trends, with its performance enhancing as the problem size increases. For 64^3, it achieves a performance of 2.92X, while for 192^3, it attains 5.60X, demonstrating the suitability of this method for larger problem sizes. Temporal Blocking exhibits a moderate and relatively linear improvement in performance as the problem size increases, with an average improvement ratio of 3.5X, While Wavefront SymGS also exhibits the moderate performance improvement and relatively high on the mid-sized problem sizes with average performance improvement of 3.8X with some drop-in performance on small and large problem sizes. MultiColor begins with a substantial improvement of 5.7X for 64^3, but its efficacy diminishes progressively as the problem size increases, yielding an improvement of merely 1.52X for 192^3. The Hybrid Jacobi GS are comparable, exhibiting a modest average improvement of 2.2X, accompanied by minimal oscillations that suggest performance stability across the problem sizes.

The OverRelaxation performs the best consistently, and Level Scheduled performs well, especially with larger problem sizes. As has been seen before, all the SymGS variants including Temporal Blocking and Wavefront provide stable performance.

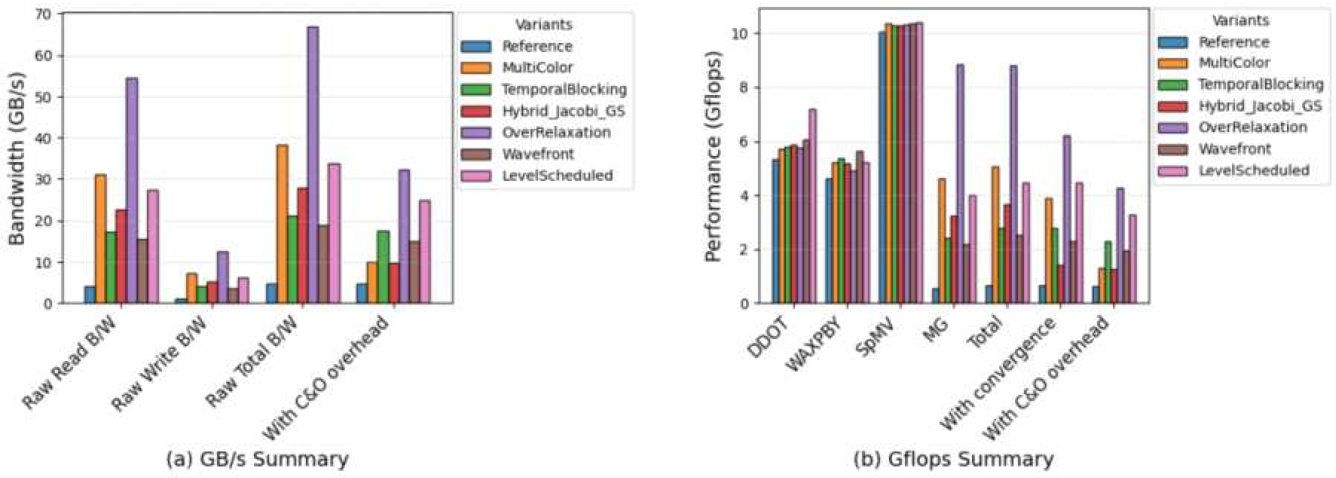


Figure 7: Performance and Bandwidth Summary of HPCG at a problem size of 160^3 using different SymGS variants

We observed that the native implementation of HPCG performs better with multiple MPI processes on the problem size of 160^3. The total number of 4,096,000 equations and 109,215,352 non-zero terms are computed at this problem size.

Fig 7 (a) illustrates that the bandwidth summary is extracted from the HPCG summary reports. OverRelaxation SymGS enhanced bandwidth utilization. The total bandwidth with convergence and optimization (C&O) is improved as compared to the reference implementation, thereby improving overall performance. Fig 7 (b) illustrates that the performance of DDOT, WAXPBY, and SpMV is nearly identical across all SymGS variants, except for the enhanced performance of DDOT when employing the Level Scheduled SymGS. This improvement may contribute to the overall performance improvement of HPCG performance using Level Scheduled in large problem sizes. However, we observe performance variability in MG as we only optimize SymGS which functions as a pre- and post-smoother within MG.

6.2.3 Scalability Analysis

We tested the MPI+OpenMP version of HPCG on a problem size of 1603, ensuring that it used enough memory as required by the HPCG specification, which requires that at least 25% of the system memory be utilized. The system under test is configured with KNL nodes, each with 96 GB of memory per node, where only 86 GB is available for use. As the number of MPI processes increases, memory usage rises significantly, and with MPI processes exceeding 20, the job fails due to memory limitations.

The performance trends indicate that the Reference method scales well with an increasing number of MPI processes. However, in OverRelaxation, the SymGS phase is parallelized using OpenMP threads, which introduces a trade-off. When the number of MPI processes increases, the room for OpenMP-based parallelization reduces, limiting overall parallel execution. As a result, performance drops when using a large number of MPI processes per node.

Scalability analysis shows strong scaling up to Node 4, where performance improves significantly with increasing MPI processes. Beyond Node 4, weak scaling becomes evident, as performance improvements slow due to communication overhead and memory constraints. The best performance is achieved with 8 MPI processes per node and 8 threads. The performance of HPCG using OverRelaxation SymGS reaching 192.82 Gflops on 16 Nodes. However, on 20 MPI processes per node with only 3 threads, exhibit performance degradation at higher node counts, demonstrating that balancing MPI processes and OpenMP threads is crucial for optimal parallel efficiency.

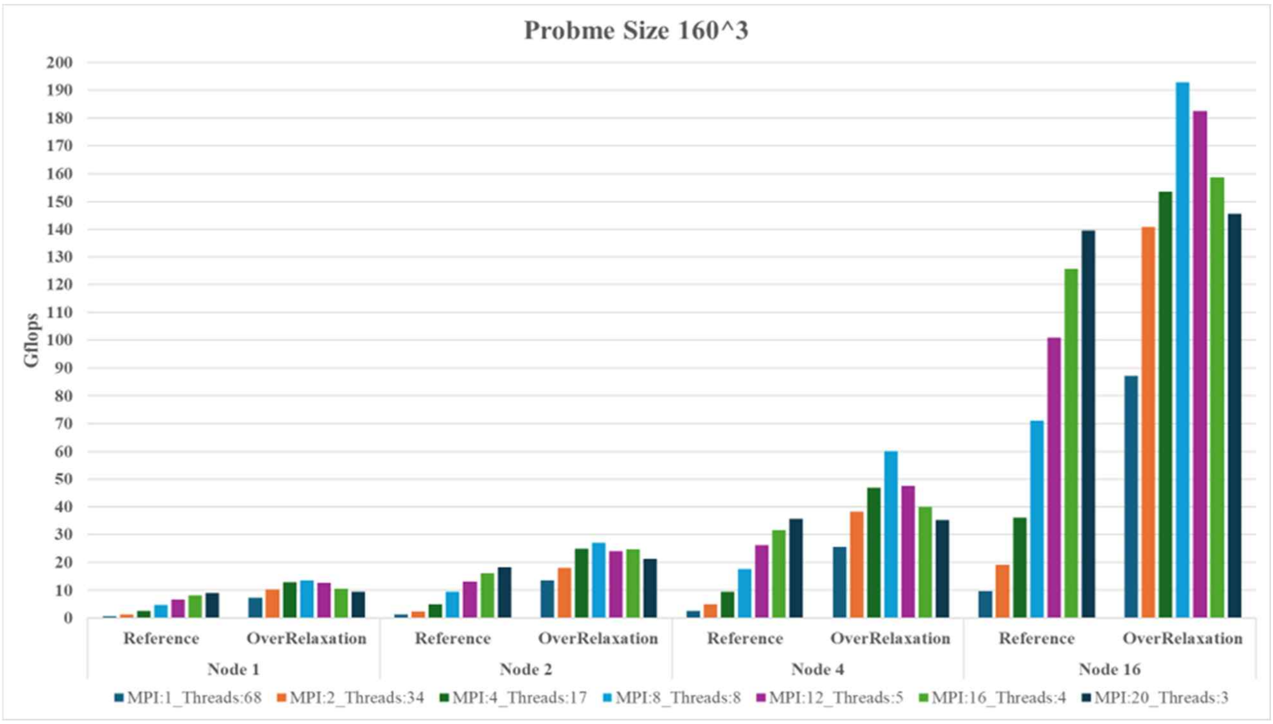


Figure 8: Performance comparison for Reference and OverRelaxation HPCG across various MPI+OpenMP configurations on multiple nodes (1, 2, 4, and 16) for a problem size of 160^3

6.3 Results on Intel Skylake (SKL)

In this section we presented the performance evaluation of the SymGS variants and HPCG on the Intel Xeon Gold 6148 (Skylake) processor, which features two sockets, each containing 20 cores, resulting in a total of 40 cores, each core operates at a frequency of 2.40 GHz. Each node is equipped with 192 GB of available memory, providing a robust platform for evaluating large-scale computations.

6.3.1 SymGS variants

Table 2: Performance comparison of different SymGS variants for a problem size of 32^3.

Variant	Setup(s)	Compute(s)	Total(s)	Avg/Iter(s)	Iterations	Rel. Error	RMSE	MAE	Gflops	Speedup	Improvement
Problem Size: 32^3, Total number of rows: 32,768											
Reference	0.0000	1.8145	1.8145	0.0037	487	0.00E+00	0.00E+00	0.00E+00	0.9005	1.0000	1.0000
MultiColor-2	0.4710	1.0472	1.5182	0.0014	731	2.49E-07	5.47E-09	4.16E-09	1.6154	1.1952	1.7940
MultiColor-4	0.2755	1.0750	1.3505	0.0016	675	1.92E-07	4.21E-09	3.21E-09	1.6770	1.3436	1.8623
MultiColor-6	0.2839	1.1479	1.4318	0.0017	672	1.74E-07	3.83E-09	2.90E-09	1.5747	1.2673	1.7487
MultiColor-8	0.0023	1.0901	1.0924	0.0015	749	2.59E-07	5.68E-09	4.32E-09	2.3003	1.6610	2.5546
TemporalBlocking-2-2	0.0047	0.5568	0.5615	0.0023	241	3.90E-08	8.31E-10	6.38E-10	1.4401	3.2318	1.5993
TemporalBlocking-2-8	0.0036	0.3902	0.3938	0.0064	61	3.37E-07	7.37E-09	5.61E-09	0.5197	4.6073	0.5771
TemporalBlocking-2-32	0.0039	0.2991	0.3030	0.0187	16	1.22E-06	2.66E-08	2.03E-08	0.1771	5.9877	0.1967
TemporalBlocking-4-2	0.0040	0.6022	0.6061	0.0033	181	1.40E-07	2.42E-09	1.84E-09	1.0019	2.9936	1.1126
TemporalBlocking-4-8	0.0041	0.4765	0.4806	0.0099	48	4.16E-07	9.76E-09	6.96E-09	0.3351	3.7755	0.3721
TemporalBlocking-4-32	0.0041	0.5674	0.5715	0.0405	14	1.34E-06	2.93E-08	2.23E-08	0.0822	3.1748	0.0913
Wavefront	0.0007	4.0938	4.0945	0.0069	597	1.04E-08	2.07E-10	1.58E-10	0.4892	0.4432	0.5433
LevelScheduled	0.0039	4.0046	4.0084	0.0082	487	0.00E+00	0.00E+00	0.00E+00	0.4076	0.4527	0.4527
Hybrid Jacobi-GS 0.5	0.0030	4.2873	4.2904	0.0026	1629	5.31E-07	9.89E-09	6.96E-09	1.2739	0.4229	1.4147
Hybrid Jacobi-GS 0.9	0.0029	3.0606	3.0635	0.0016	1926	5.47E-08	1.20E-09	9.12E-10	2.1093	0.5923	2.3424
OverRelaxation 1.0	0.0000	1.3511	1.3511	0.0014	964	3.90E-08	8.31E-10	6.38E-10	2.3939	1.3430	2.6585
OverRelaxation 1.4	0.0000	0.9627	0.9627	0.0014	687	5.22E-08	1.12E-09	8.58E-10	2.3943	1.8849	2.6590

The performance of SymGS variants on SKL resembles the trends observed on KNL, OverRelaxation and MultiColor SymGS remain the top two performers, respectively. Table 2 presents a comparative analysis of SymGS variants for a problem size of 323 (32,768 rows) and Fig. 9 illustrates this comparison in terms of Gflops. OverRelaxation SymGS with ω = 1.4 outperformed the other variants, reducing the execution time to 0.9627 seconds and achieving a 1.88X acceleration compared to the reference SymGS. This improves performance by 2.659X, resulting in 2.39 Gflops. The MultiColor SymGS variant with eight colors is the second-best performer, with a total time of 1.0924 seconds, a 1.66X speedup over the reference, and a 2.5546X increase in Gflops.

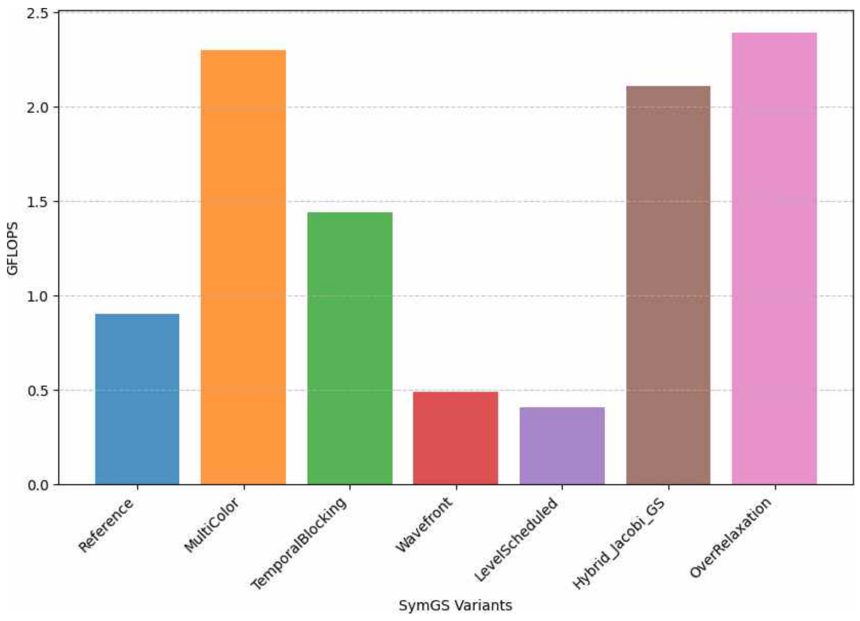


Figure 9: Performance comparison of SymGS variants in terms of Gflops across problem size 32^3.

In Temporal Blocking SymGS variants, specifically TB-2-2, outperform its other combinations. Hybrid Jacobi-GS with jr = 0.9 results in a 2.3424X increase in Gflops, indicating that combining Jacobi parallelism and Gauss-Seidel updates improves performance. The Wavefront and Level Scheduled SymGS variants do not perform good on Skylake for this problem size. In fact, they both perform worse than the reference implementation.

6.3.2 HPCG Results

In the HPCG benchmark, the OverRelaxation SymGS variant was used with ω = 1.0, however its performance is slightly lower compared to $\omega = 1.4$, as it converges within just two iterations on the exaggerated diagonal, a design feature of HPCG, which exaggerates diagonal dominance to assess spectral convergence properties. These SymGS variants were used in HPCG, including MultiColor SymGS with $c = 8$, Temporal Blocking SymGS with $a = 2$ and $b = 2$, Hybrid Jacobi-GS with $j_r = 0.9$, and OverRelaxation with $\omega = 1.0$.

First, we evaluated the reference implementation of HPCG on SKL without modifications to identify the optimal problem size for performance. We have observed that HPCG performs better on the problem size of 32^3 on a single node for possible combinations of MPI and OpenMP settings.

Fig. 10 shows the performance evaluation of HPCG using these variants for the problem size of 32^3 with a single MPI process using all 40 threads on Skylake. The HPCG with OverRelaxation SymGS variant achieves the highest performance, highlighting its improved parallelism when integrated into the multigrid preconditioner. MultiColor SymGS and Temporal Blocking SymGS also show notable improvement compared to the reference implementation, although these are behind Overrelaxation.

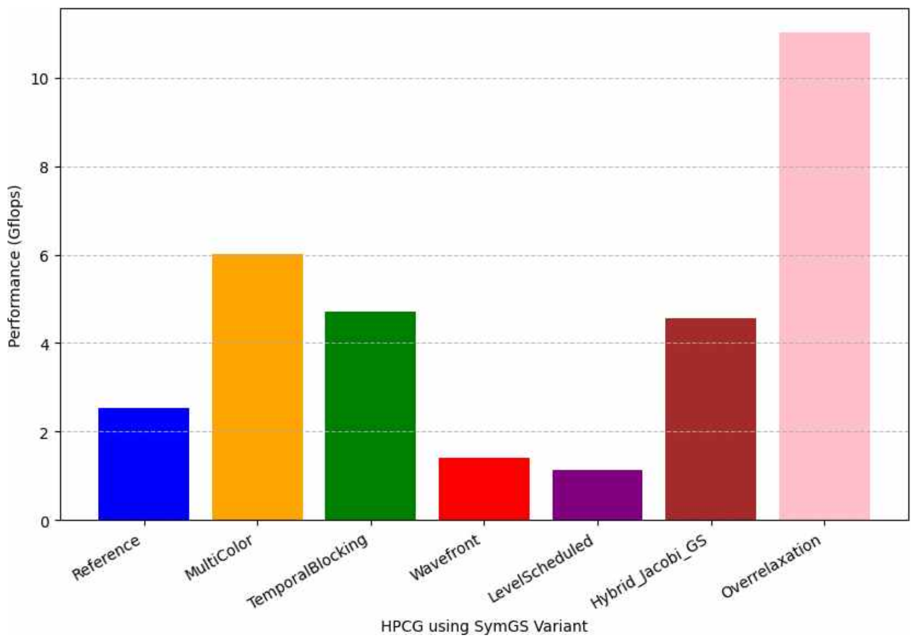
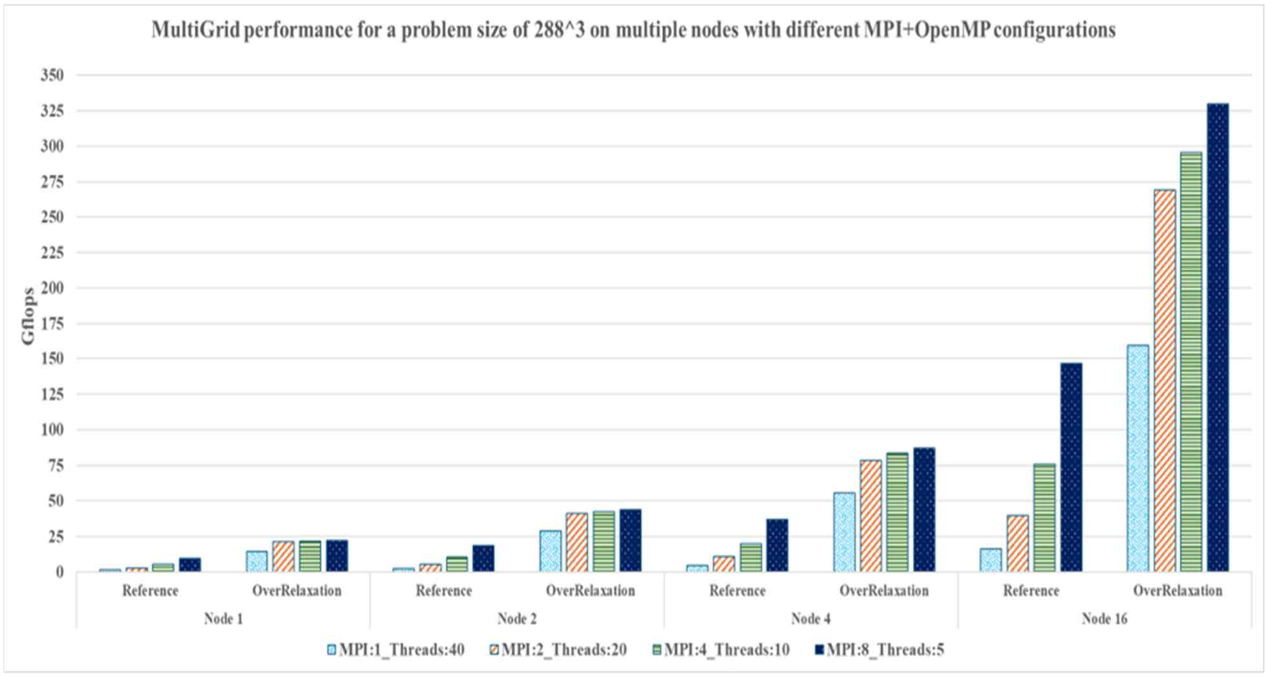


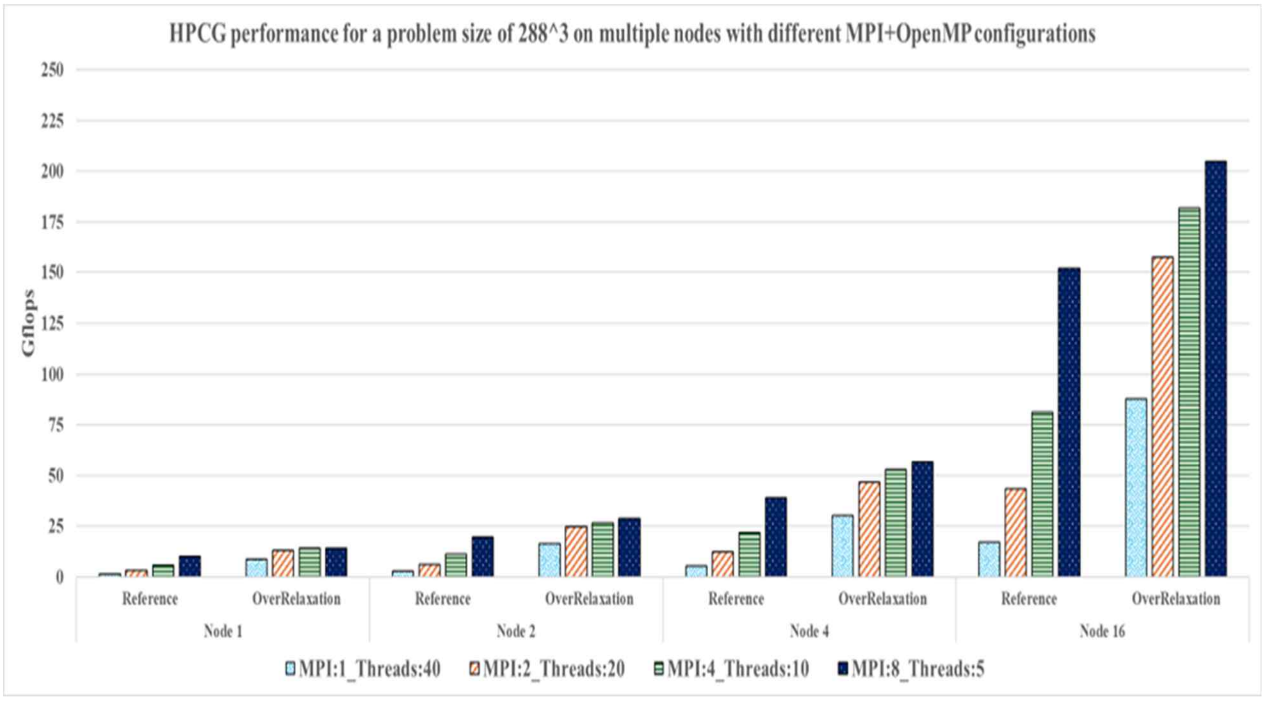
Figure 10: Performance Evaluation of HPCG with SymGS variant on the problem size of 32^3

6.3.3 Scalability Analysis

The problem size of 32^3 is insufficient to fully stress the memory subsystem or meet the memory requirements outlined by HPCG. Therefore, a larger problem size of 288^3 was evaluated in multi-node configurations. We evaluated HPCG performance using both the Reference and the OverRelaxation SymGS. We have optimized only the SymGS component of the Multigrid in HPCG. We recorded the performance results of HPCG using the Reference and OverRelaxation SymGS variants, and additionally showcased the performance of Multigrid, extracted from the HPCG summary reports as shown in Fig. 11. As shown in the Fig. 11 (a), the Multigrid performance using the OverRelaxation SymGS surpasses the Reference SymGS across all node counts, ranging from single-node to sixteen-node configurations. On a single node utilizing 1 MPI process and 40 threads, OverRelaxation attains 14.65 Gflops. With an increase in the number of MPI processes, the performance of OverRelaxation significantly surpasses that of the Reference, attaining 21.99 Gflops at 8 MPI processes, in contrast to only 9.49 Gflops for Reference. On 16 nodes, OverRelaxation achieves 159.1 Gflops at one MPI process, indicating an approximate 10_X improvement over the Reference, and consistently surpasses the Reference across all MPI configurations, ultimately attaining 329.7 Gflops at 8 MPI processes per node, in contrast to 146.5 Gflops for the Reference on 16 Node configuration. The job terminated due to memory limitations when increasing the MPI processes per node for this large problem size. However, we observed that with small problem sizes, increasing the number of MPI processes per node results in a reduction of threads per MPI process for parallelism in the Overrelaxation SymGS variant. When there are only one or two threads per MPI process, the potential for parallelism is minimal, resulting in performance that is slightly inferior to the Reference implementation. A comparable trend is noted for HPCG, wherein OverRelaxation consistently improves performance across all configurations. On a single node utilizing one MPI process, OverRelaxation attains 8.64 Gflops, nearly 6_X surpassing the 1.48 Gflops recorded by the Reference SymGS. As the node count increases, OverRelaxation maintains a significant improvement, especially in settings with a reduced MPI count, where the SymGS solver can exploit enhanced OpenMP-based parallelism. On two nodes with one MPI process, OverRelaxation attains 16.38 Gflops, whereas the Reference achieves 2.76 Gflops, indicating an approximate 6_X improvement. On increasing MPI counts, specifically 8 MPI processes across 16 nodes, OverRelaxation achieves 204.86 Gflops, in contrast to 151.84 Gflops for Reference, illustrating performance improvements despite rising communication overhead. The results demonstrate that OverRelaxation SymGS offers significant computational acceleration and enhanced scalability in multi-node setups, rendering it a viable alternative to the Reference SymGS, particularly for memory-constrained, sparse matrix challenges such as those encountered in HPCG and Multigrid. However, there exists a notable difference in the performance of HPCG and Multigrid, as illustrated in Fig 11, which indicates that the Multigrid performance is relatively good as compared to HPCG this highlights the need to improve the other kernels and reduce the communication and optimization overhead to further improve the overall performance of HPCG.



(a) MultiGrid



(b) HPCG

Figure 11: Performance of MultiGrid and HPCG for a problem size of 288^3 using different MPI+OpenMP configurations across multiple nodes.

7. Conclusion and Future Work

This study presented and assessed different variants of Symmetric Gauss-Seidel (SymGS), enhancing their performance for symmetric positive definite (SPD) matrices in the HPCG benchmark. The Temporal Blocking, Multi Color, OverRelaxation, and Level-Scheduled variants have proven to be the most effective for tackling significant challenges related to parallelism in SymGS. The implementation of these variants demonstrated enhanced parallel performance without compromising on numerical stability, resulting in a substantial increase in computational efficiency for solving large sparse linear systems. The implementations of Temporal Blocking, Over Relaxation, and Multi Color variants of SymGS surpass the Reference SymGS. The Over Relaxation SymGS variant demonstrated an increase in performance when integrated into the HPCG benchmark compared to the native HPCG. These results underscore the efficacy of the optimized SymGS variants for integration into HPCG, yielding significant computational improvement.

Experimental results validated on Intel Xeon Phi (KNL) and Intel Xeon Gold (SKL) platforms utilizing MPI+OpenMP configurations demonstrated performance improvement compared to the native implementation, with our proposed SymGS variants. The study provides a theoretical framework, pseudo-code algorithms, and insights into parameter optimization that enhance the robustness and scalability of SymGS in contemporary high-performance computing settings.

For future work, these variants can be combined with better data layouts, cache blocking, and communication-avoidance strategies to further improve HPCG performance. Additional optimization of other HPCG kernels (e.g., Sparse Matrix-Vector multiply) and further synergy between the smoother and coarser levels of the multigrid cycle could amplify the overall benefits on modern architectures.

8. References

[1] Zdravko Virag, Ivo Dˇzijan, and Severino Krizmani´c. Improved symmetric gauss-seidel method for solving sparse linear systems appear in cfd. In 5th International Congress of Croatian Society of Mechanics, 2006.

[2] Nirupma Bhatti et al. Comparative study of symmetric gauss-seidel methods and preconditioned symmetric gauss-seidel methods for linear system. International Journal of Science and Research Archive, 8(1):940–947, 2023.

[3] Richard Barrett, Michael Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk Van der Vorst. Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, chapter The Jacobi Method. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1994. Accessed: 2025-01-09.

[4] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. A new metric for ranking high-performance computing systems. National Science Review, 3(1):30–35, 2016.

[5] Jack Dongarra, Michael A Heroux, and Piotr Luszczek. High-performance conjugate-gradient bench- mark: A new metric for ranking high-performance computing systems. The International Journal of High-Performance Computing Applications, 30(1):3–10, 2016.

[6] Jack J Dongarra. The linpack benchmark: An explanation. In International Conference on Super- computing, pages 456–474. Springer, 1987.

[7] Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. Concurrency and Computation: practice and experience, 15(9):803–820, 2003.

[8] HPCG benchmark. https://www.hpcg-benchmark.org/, 2015. Accessed: 2024-02-15.