Thesis for the Degree of Doctor

KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Framework

KoHPCG - 이기종 환경에서 성능 이식성을 보장하는 Kokkos 프레임워크 기반의 HPCG 벤치마크 프로그램

June 2025

Department of Computer Science and Engineering

Graduate School of Soongsil University

MUHAMMAD RIZWAN

Thesis for the Degree of Doctor

KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Framework

KoHPCG - 이기종 환경에서 성능 이식성을 보장하는 Kokkos 프레임워크 기반의 HPCG 벤치마크 프로그램

June 2025

Department of Computer Science and Engineering

Graduate School of Soongsil University

MUHAMMAD RIZWAN

Thesis for the Degree of Doctor

KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Framework

Thesis Supervisor: Prof. Dr. Jaeyoung Choi

Thesis submitted in partial fulfillment of the requirements for the Degree of Doctor

June 2025

Department of Computer Science and Engineering

Graduate School of Soongsil University

MUHAMMAD RIZWAN

To approve the submitted thesis for the

Degree of Doctor by MUHAMMAD RIZWAN

Thesis Committee

Chair	Chae-Woo Yoo	(signature)
Member	Jongsun Choi	(signature)
Member	Myungho Lee	(signature)
Member	Imbum Kim	(signature)
Member	Jaeyoung Choi	(signature)

June 2025

Department of Computer Science and Engineering

Graduate School of Soongsil University

ACKNOWLEDGEMENT

First and foremost, all praise is due to **Almighty Allah**, the Most Merciful and the Most Compassionate, who granted me the strength, patience, and perseverance to complete this endeavor. Without His divine guidance and mercy, this milestone would not have been possible.

This dissertation stands as a testament not only to individual effort, but to the unwavering support and inspiration of many remarkable individuals around me. I am deeply grateful to **my supervisor, Dr. Jaeyoung Choi**, for his invaluable guidance, insightful feedback, and continuous encouragement throughout my PhD journey. His mentorship, marked by both professionalism and kindness, has been a cornerstone of my academic growth.

I would like to express my heartfelt thanks to **my committee members** for their constructive suggestions and support. My appreciation also goes to all **lab mates and alumni**, whose contributions created a stimulating and collaborative research environment.

Special thanks go to **all my teachers** who laid the foundation for my academic journey. I am equally grateful to all individuals—named or unnamed—who supported me directly or indirectly with their encouragement, assistance, and prayers.

I extend my sincerest gratitude to **my parents and family members**, whose love, patience, and unwavering belief in me have been the cornerstone of my motivation.

I also want to acknowledge the friendship and solidarity of my **fellow countrymen** as well as **friends from around the world**, who brought joy and perspective into this journey with their diverse experiences.

Finally, to all those who helped me along the way, even if your name is not listed here, please accept my deepest thanks. Your support was valuable and will always be remembered.

Date: June 2025 **MUHAMMAD RIZWAN**

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

ABMC	Algebraic Block Multi-Coloring
ADB	Assignable Data Buffer
ALP	ALP/GraphBLAS programming framework
AMD	Advanced Micro Devices
AVL	Architectural Vector Length
BMC	Block Multi-Coloring
BFS	Breadth-First Search
CG	Conjugate Gradient
COO	Coordinate
CSC	Compressed Sparse Column
CSCS	Swiss National Supercomputing Centre
CPE	Core Processing Elements
CPU	Central Processing Unit
CSR	Compressed Sparse Row
CU	Compute Unit
CUDA	Compute Unified Device Architecture
cuSPARSE	CUDA Sparse Matrix library
DDOT	Double-precision DOT product
DDR	Double Data Rate (synchronous DRAM memory)
DIA	Diagonal format
DOACROSS	A type of loop optimization technique
DSCR	Data Stream Control Register
ELL	ELLPACK
FPGA	Field-Programmable Gate Array

GPU	Graphics Processing Unit
GS	Gauss-Seidel
HBM	High Bandwidth memory
HG	Hierarchical Grid
HIP	Heterogeneous-Compute Interface Library for Portability
HPC	High-Performance Computing
HPL	High-Performance Linpack
HPCG	High-Performance Conjugate Gradient
HPGMG	High Performance Geometric Multigrid
IA	Intel Architecture
IBM	International Business Machines
JAD	Jagged Diagonal format Machines
KHPCG	Kokkos based HPCG Benchmark
KNL	Xeon Phi Intel Knights Landing
KOKKOS	A performance portable programming library
LDM	Local Device Memory
LS	Level Scheduling
MC	Multi-Coloring
MIC	Many Integrated Core
MGM	Multigrid Method
MGPCG	Multi-Grid Preconditioned Conjugate Gradient
MKL	Math Kernel Liberary
MPE	Management Processing Elements
MPI	Message Passing Interface
NDP	Near-Data Processors
NEC VE	NEC Vector Engine, A high-performance vector processor architecture developed by NEC Corporation
NVIDIA	A GPU manufacturer and designer company
NUMA	Non-uniform memory access
OpenMP	Open Multi-Processing

ORNL	Oak Ridge National Laboratory
PCG	Preconditioned Conjugate Gradient
PDE	Partial Differential Equation
PE	Processing Element
RB	Red-Black Coloring
RBGS	Red-Black Gauss-Seidel
RISC-V	Reduced Instruction Set Computing – V (fifth open-ISA generation)
Rmax	Maximum Measured Performance
ROCm	Radeon Open Compute
Rpeak	Peak Performance
SELLPACK	A slieced variant of ELLPACK data format
SIMD	Single Instruction Multiple Data
SpMV	Sparse Matrix-Vector Multiplication
SVE	Scalable Vector Extension
SymGS	Symmetric Gauss-Seidel
SX-ACE	A vector parallel processor
SYCL	Standard C++ for Heterogeneous Computing
SKL	Intel Skylake processors
TOP500	A project ranks most powerful supercomputers
UMBC	University of Maryland, Baltimore County
VE	Vector Engine
VPU	Vector Processing Unit
WAXPBY	Weighted A times X plus B times Y vector operation.
Xilinx	Xilinx is the company name (now owned by AMD), pioneer in FPGA technology

ABSTRACT

KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Framework

MUHAMMAD RIZWAN

Department of Computer Science and Engineering

Graduate School of Soongsil University

The High-Performance Conjugate Gradient (HPCG) benchmark complements the HPL benchmark for supercomputing system evaluation. HPL emphasizes dense linear algebra operations with high floating-point performance, while HPCG emphasizes memory access patterns and sparse linear algebra operations that are common of many scientific applications to better assess modern supercomputing architectures. The memory-bound kernels of the reference HPCG implementation, especially the sequential Symmetric Gauss-Seidel (SymGS) routine and the bandwidth-limited Sparse Matrix-Vector Multiplication (SpMV) operation, limits performance.

This thesis introduces **KoHPCG**, a performance-portable HPCG benchmark using Kokkos programming model to address portability issues. The work begins with a thorough survey of HPCG optimizations to identify bottlenecks and improvements. Multiple algorithmic variants of the Symmetric Gauss-Seidel method are developed

and evaluated to improve parallel scalability without affecting the numerical accuracy.

The implementation translates all core HPCG kernels into Kokkos, including DDOT, WAXPBY, SpMV, SymGS, and MG operations. This uses Kokkos::Views for memory management, parallel execution, and execution and memory space abstractions to achieve performance portability across architectures. In multi-node configurations scaling up to 16 nodes, evaluated on Intel Xeon Phi (KNL) and Xeon Skylake (SKL) with MPI+OpenMP configurations. The results show significant improvements over the reference HPCG implementation and previous Kokkos-based variants like KHPCG.

The contributions of this work include: (1) a thorough analysis of HPCG optimization techniques and bottlenecks, (2) novel algorithmic variants of SymGS that improve parallel scalability, (3) a complete performance-portable implementation of HPCG using Kokkos, and (4) comprehensive performance evaluation on intel architectures. This research gives the HPC community a robust and extensible benchmarking framework for realistic performance evaluation on different systems, laying the groundwork for performance-portable HPCG improvements.

Keywords: *High-Performance Conjugate Gradient* (*HPCG*), *Symmetric Gauss–Seidel* (*SymGS*), *Iterative Methods, Parallel Computing, Performance Optimization, Performance Portability, Kokkos, Multigrid* (*MG*), *High-Performance Computing* (*HPC*).

국문초록

KoHPCG - 이기종 환경에서 성능 이식성을 보장하는 Kokkos 프레임워크 기반의 HPCG 벤치마크 프로그램

리즈완 무하마드

컴퓨터학과

숭실대학교 대학원

HPCG (High-Performance Conjugate Gradient) 벤치마크는 슈퍼컴퓨팅 시스템 평가를 위한 HPL 벤치마크를 보완합니다. HPL 은 높은 부동소수점 연산 성능을 갖춘 조밀한 선형대수 연산을 강조하는 반면, HPCG 는 과학적 응용에서 흔히 나타나는 메모리 접근 패턴 및 희소 선형대수 연산을 강조하여 현대 슈퍼컴퓨팅 아키텍처를 보다 현실적으로 평가할 수 있도록 합니다. 특히, 참조 HPCG 구현의 메모리 병목 커널인 SymGS (Symmetric

Gauss-Seidel) 루틴과 대역폭 제한을 받는 SpMV (Sparse Matrix-Vector Multiplication) 연산은 성능을 제한하는 요인입니다.

본 논문은 이식성 문제를 해결하기 위해 Kokkos 프로그래밍 모델을 사용한 성능 이식 가능한 HPCG 벤치마크인 KoHPCG 를 소개합니다. 본 연구는 HPCG 최적화 기법에 대한 철저한 조사를 바탕으로 병목 현상과 개선 사항을 파악하는 것에서 출발합니다. 수치 정확도에 영향을 주지 않으면서 병렬 확장성을 향상시키기 위한 여러 가지 SymGS 알고리즘 변형을 개발하고 평가합니다.

KoHPCG 구현은 DDOT, WAXPBY, SpMV, SymGS, MG 연산을 포함한 모든 핵심 HPCG 커널을 Kokkos 로 변환하며, 메모리 관리, 병렬 실행, 실행/메모리 공간 추상화를 위해 Kokkos::Views 를 사용하여 아키텍처에 구애받지 않는 성능 이식성을 달성합니다. Intel Xeon Phi (KNL) and Xeon Skylake (SKL) 시스템에서 MPI+OpenMP 환경으로 최대 16 개의 노드를 사용하는 다중

노드 환경에서의 확장성 실험을 통해 기존 참조 HPCG 구현 및 KHPCG 와 같은 이전 Kokkos 기반 구현보다 유의미한 성능 향상을 보여줍니다.

본 연구의 주요 기여는 다음과 같습니다: (1) HPCG 최적화 기술 및 병목 현상에 대한 철저한 분석, (2) 병렬 확장성을 개선하는 새로운 SymGS 알고리즘 변형, (3) Kokkos 를 활용한 완전한 성능 이식 가능한 HPCG 구현, (4) 인텔 아키텍처를 대상으로 한 포괄적인 성능 평가. 이 연구는 다양한 시스템에서 현실적인 성능 평가를 가능하게 하는 견고하고 확장 가능한 벤치마크 프레임워크를 HPC 커뮤니티에 제공함으로써, 성능 이식 가능한 HPCG 개선의 기반을 마련합니다.

키워드: *고성능 컨주게이트 그래디언트(HPCG), 대칭 가우스–자이델(SymGS), 반복 해법, 병렬 컴퓨팅, 성능 최적화, 성능 이식성, Kokkos, 멀티그리드(MG), 고성능 컴퓨팅(HPC).*

CHAPTER 1. Introduction

The High-Performance LINPACK benchmark (HPL) [1] has been utilized to measure supercomputer performance since the 1990s. Primarily employed to solve the dense linear algebraic equations. Despite HPL's longstanding reliability as a standard, modern supercomputers and emerging applications have exposed its bounds. HPL measures only peak performance, however it fails to accurately represent the performance of modern applications. These applications are sophisticated and require optimal coordination of different components within the computer system.

High-performance computing (HPC) utilizes the High-Performance Conjugate Gradient (HPCG) [2–5] as an entirely different benchmark for evaluating and comparing the performance of modern supercomputers. The HPCG benchmark utilizes the conjugate gradient method, which function on sparse matrices. A sparse matrix is a matrix characterized by a significantly large number of zero elements in comparison with non-zero elements. HPCG specifically accounts for deficiencies of HPL and presents a complement to the well-established HPL benchmark. HPCG more accurately measures the performance of modern applications by evaluating their capacity to deal with complex problems. HPCG evaluates the coordination of all components of the computing system, including memory bandwidth, computational

capability, interconnect network efficiency, and overall system synchronization. HPCG is a benchmark that more accurately represents the performance of modern, practical, and real-world applications.

HPL solves the linear equations in dense matrices using Gaussian elimination with partial pivoting, whereas HPCG concentrates on partial differential equations (PDEs) and solves linear systems of equations in sparse matrices, discretized using 27-point stencils for three-dimensional elliptical PDEs. The HPCG reference implementation uses the preconditioned conjugate gradient (PCG) algorithm in conjunction with the multigrid method (MGM). The ability of HPCG to evaluate different components of the system provides it an advantage over HPL. The recent development of Exaflops (10^{18} flops) supercomputers makes the use of HPL to measure system performance less attractive for practical applications.

Kokkos [6] is a C++ library designed to enhance performance portability across diverse hardware architectures. These architectures encompass central processing units (CPUs), graphics processing units (GPUs), and other novel platforms. Besides offering abstractions for parallel execution and data management, it allows developers to produce code that is efficient and portable, without having to deal with the complexities that are specific to the hardware architectures. Kokkos is compatible with various backends, including CUDA, HIP, SYCL, OpenMP, and C++ threads, facilitates seamless transitions between different execution environments. Kokkos Kernels offers a compilation of performance-optimized routines. These routines comprise of sparse and dense linear algebra, batched operations, and graph algorithms. All of these routines are formulated according to the Kokkos programming model. The interplay of these two factors enables developers to construct high-performance applications that align with the progressively evolving

paradigm of HPC.

1.1 Motivation

The foundation of HPCG is the Preconditioned Conjugate Gradient (PCG) algorithm, which depends on essential computational kernels: Sparse Matrix-Vector Multiplication (SpMV) and Symmetric Gauss-Seidel (SymGS). These kernels are memory bound, with SymGS identified as the main performance bottleneck owing to its inherently sequential nature and data dependencies. This bottleneck significantly restricts scalability and impacts the effective use of modern hardware, where computational throughput outweighs the available memory bandwidth.

Over the past decade, numerous researchers have proposed methodologies to improve the performance of SymGS across various hardware platforms, including CPUs, GPUs, MICs, and FPGAs. However, these efforts have predominantly resulted in architecture-specific optimizations that require significant optimization and are not easily portable across diverse systems. The increasing diversity of HPC systems renders the absence of performance portability a major limitation.

A notable attempt to address this issue was the development of KHPCG [7, 8], a Kokkos-based variant of HPCG aimed at delivering performance portability through hardware abstraction.The Kokkos programming model [6, 9, 10] accommodates multiple backends, including CUDA, HIP, SYCL, and OpenMP, allowing applications to operate on diverse platforms without requiring the modification of architecture-specific code. Although KHPCG an important step forward but it was suffered from several limitations, including limited parallelism (support for a single MPI rank), suboptimal performance, and numerical instability, especially in the multicolor implementation of SymGS. The report [11] emphasized the main

problems in the practical implementation of KHPCG.

These challenges highlight a clear need for an effective solution. The motivation behind this thesis arises with the goals in mind, as follows:

- Addressing the performance bottleneck caused by the SymGS routine in the HPCG benchmark through the development of parallel variants of SymGS.
- Porting the core HPCG computational kernels to the Kokkos programming model to facilitate portability.
- Providing the HPC community with a scalable, efficient, and performance-portable implementation of the HPCG benchmark capable of functioning across diverse architectures.

1.2 Problem Statement

Although HPCG is valuable for simulating realistic application behavior, its performance is considerably constrained by memory-bound kernels, especially SymGS and SpMV operations. The existing reference implementation of SymGS in HPCG is sequential, presenting two significant issues:

1. It fails to properly exploit the parallel processing capabilities of modern CPUs, GPUs, and other accelerators.
2. It does not scale efficiently with increasing core counts or across different architectures.

Thus, there is a need to redesign the SymGS routine for parallel execution without compromising its convergence properties and to integrate this within a performance-portable framework to support future computing environments.

1.3 Research Objectives and Contributions

The primary objective of this thesis is to develop a high-performance and performance-portable implementation of the HPCG benchmark, focusing particularly on optimizing the SymGS kernel. The specific goals are as follows:

- Analyze the limitations of the current SymGS implementation in the HPCG benchmark.
- Investigate existing parallelization strategies for SymGS based on graph theory.
- Design and implement a parallel and scalable SymGS algorithm suitable for various hardware architectures.
- Integrate the new SymGS implementation into the HPCG benchmark using the Kokkos framework [6] for performance portability.
- Evaluate the performance of the proposed solution across multiple architectures including CPUs and GPUs.

Key Contributions:

- A thorough study and classification of SymGS optimization strategies for HPCG.
- Development of new parallelized variants of the SymGS without compromising the numerical stability while improving performance.
- The optimized SymGS method is integrated to a Kokkos-based HPCG benchmark implementation so that it can be used on multiple platforms.
- Empirical evaluation of the proposed implementation on Intel Xeon Phi (KNL) and Xeon Skylake (SKL) systems to demonstrate performance improvements

and scalability.

1.4 Thesis Organization

The remainder of this thesis is organized as follows:

- **Chapter 2. Background**

  This chapter introduces the foundational concepts and tools you need to know in order to understand the rest of the thesis. It includes:

  - **HPCG:** Overview of the High Performance Conjugate Gradient benchmark.
  - **Kokkos:** Introduction to the Kokkos programming model and its role in performance portability.
  - **SymGS:** Explanation of the Symmetric Gauss-Seidel routine and its significance in HPCG.

- **Chapter 3. Literature Review**

  Examines prior research efforts and developments, concentrating on:

  - **HPCG:** Implementation details, its evolution and adoption. Its optimization constraints as some changes are allowed, but the authors who developed HPCG imposed some restrictions and insisted that those parts of the benchmark should not be modified.
  - **SymGS:** Optimizations and limitations that have been reported in previous work.
  - **Kokkos-Based Implementation:** Review of Kokkos integration into HPCG and related performance-portable efforts.

- **Chapter 4. Techniques and Trends in HPCG**

  Explores broader patterns and practices used in optimizing HPCG, such as:

  - **Parallelization Strategies:** Different parallelization strategies adopted by other researchers to improve the architecture-specific performance.
  - **Data Layout and Storage Formats:** Different data storage formats and their impact.
  - **Open Challenges:** Technical issues in the pursuit of scalable and portable HPCG.

- **Chapter 5. SymGS Variants**

  Details the development and parameters selection of:

  - **Our Developed SymGS Variants:** Present the concept and algorithmic details of our effort to parallelize the SymGS and conduct a comparative analysis of these variants.

- **Chapter 6. Performance-Portable KoHPCG**

  Presents the design and build details of a Kokkos-based portable HPCG implementation that includes our optimized SymGS variant.

- **Chapter 7. Conclusion and Future Work**

  Summarizes the thesis contributions and outlines directions for future research.

CHAPTER 2. Background

2.1 HPCG

HPCG is a new benchmark more relevant to real application for HPC systems than other benchmarks like HPL. Its primary objectives are to achieve the ability to estimate the system performance for the target application by mirroring the computational behaviors in actual environments and contribute to enhancing computer systems that address practical use cases, to complement the measurements that show the theoretical potential of the system. The HPCG benchmark measures supercomputer performance, providing a more realistic measure than the HPL benchmark [12].

2.1.1 Preconditioned Conjugate Gradient Method

Conjugate Gradient (CG) method is a numerical iterative solver used to solve linear systems, and the convergence rate of the CG method is measured to evaluate systems performance. The HPCG benchmark depends upon the Preconditioned Conjugate Gradient (PCG) algorithm [12], which is an iterative computational technique helpful when solving large size sparse linear system of equations [2].

Algorithm 1 **Preconditioned Conjugate Gradient (PCG)** begins by initializing

Algorithm 1 Preconditioned Conjugate Gradient (PCG)

Input: Matrix A, vectors b, initial guess x, tolerance (ϵ), max iterations k_{max}
Output: Approximate solution vector x
Set: x_0 ▷ Set initial guess: x_0
$r_0 = b - A \cdot x_0$ ▷ Compute initial residual
$p_0 = r_0$ ▷ Set initial search direction
$\text{normr}_0 = \|r_0\|_2$ ▷ Compute initial residual norm
for $k = 1$ to k_{max} **do**
    $z_k = \text{MG}(A, r_k)$ ▷ Compute preconditioned residual
    $\text{rtz}_k = r_k \cdot z_k$ ▷ Compute dot product
    $Ap_k = A \cdot p_k$ ▷ Compute
    $\alpha_k = \frac{\text{rtz}_k}{p_k \cdot Ap_k}$ ▷ Compute step size
    $x_{k+1} = x_k + \alpha_k p_k$ ▷ Update solution
    $r_{k+1} = r_k - \alpha_k Ap_k$ ▷ Update residual
    if $\|r_{k+1}\|_2 / \text{normr}_0 < \epsilon$ **then** ▷ Check for convergence
        break
    end if
    $\beta_k = \frac{\text{rtz}_{k+1}}{\text{rtz}_k}$ ▷ Compute new direction
    $p_{k+1} = z_{k+1} + \beta_k p_k$ ▷ Update search direction
end for

an approximate solution x_0 and compute the initial residual $r_0 = b - Ax_0$, which estimates the error in the initial guess, and the direction of search is initially set as the residual, $p_0 = r_0$, and the algorithm iterates to update residual and search direction to reach for the solution. In each iteration a multigrid preconditioner is used to improve the convergence, then a dot product and a matrix-vector multiplication are performed to compute the step size α_k, and it determines to move along the search direction. The solution and then the residual are updated to recalculate the remaining error. If the error is sufficiently small, the iteration process stops algorithm to perform, when it check for convergence by comparing the current residual against a given ϵ. If not, the search direction is updated, to ensure it remains conjugate to the previous directions, and the process repeats. With appropriate preconditioning, this iterative approach allows the PCG method to efficiently solve large sparse systems of equations. HPCG

is primarily relies on the performance of the SpMV and SymGS. HPCG solves a sparse linear equation with a simple additive Schwarz using the PCG algorithm [2].

2.1.2 HPCG Execution Flow Process

The HPCG benchmark is designed to simulate real-world computation patterns usually found in scientific/engineering applications. Its execution flow as shown in Figure 2.1 begins with allocating for the local sub-domain of each MPI process and the geometry setup, which divides the problem domain for parallel computing. Then, initializes sparse matrices and prepares main data structures such as the matrix A, solution vector x, and right-hand side vector b. Key computational operations are **Compute SymGS**, which performs SymGS iterations to approximate the solution of the sparse system, while **Compute SpMV** called to accomplish the sparse matrix-vector multiply, **Compute Dot-Product** computes the vector-vector dot products, and **Compute WAXPBY** does the weighted vector adds. Finally, users may implement their optimization routines using the **OptimizeProblem** function provided in the reference implementation. Once the iterative process is completed, results produces a report consisting of timing information, flops, memory bandwidth, and validation information.

The core computational kernels participating in the HPCG benchmark are Dot Prodeuct (DDOT), WAXPBY, SymGS, SpMV, Restriction and Prolongation operations, which are explained in detail in Section 2.1.6.

HPCG forms the problem setup then a symmetric positive definite matrix is created from Compressed Sparse Row (CSR) format [2]. Such an approach makes use of memory and computations in the most efficient manner possible, and the benchmark really tests the capabilities of a machine. The amount of data in the matrix is

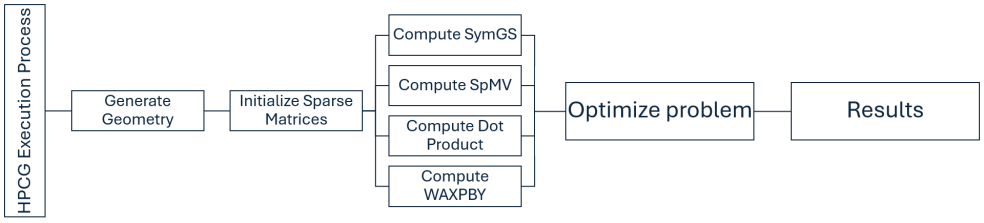


Figure 2.1 HPCG Execution Process Flow

designed to optimally fit the machine' s capacity to conduct an exhaustive assessment of its performance. HPCG benchmark implies the local SymGS preconditioner. Whereas this preconditioner helps in the reduction of the matrix and so helps in faster convergence in PCG algorithm. The matrix is divided into lower and upper triangular matrices; this allows the preconditioner to gradually improve the solution making it functionally efficient. Also, the benchmark requires verification and/or validation processes, computation of pre/post conditions, and invariants. Convergence tests and comparison with the reference kernels were employed to check the accuracy of the computation to assure that the results obtained are consistent. HPCG benchmark replicates the actual application workloads and uses multiple iterations. Numerical results obtained at each iteration are checked with expected answers for verification and cache is cleared before each iteration. It eliminates cases of false popularity from cache usage as well as ensures an impartial evaluation of the system. Finally, HPCG produces a report consisting of timing information, flops, and validation computations. System configuration is documented throughout this report and may be critical in understanding benchmark performance. This makes HPCG benchmark comprehensive and fair at the same time for HPC systems evaluation.

2.1.3 Problem Setup in HPCG

Figure 2.2 shows a 27-point stencil 3D in HPCG benchmark designed to solve the linear system of equation:

$$A \cdot x = b.$$

where A is a sparse matrix of size $n \times n$, x is an unknown vector of size n, b is a known vector of size n.

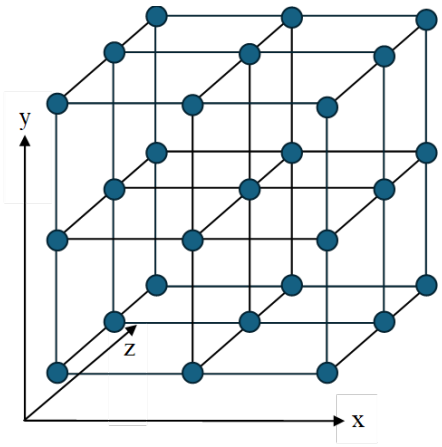


Figure 2.2 27 point stencil in HPCG

It approximate the values of the solution vector x using the PCG algorithm, which is an iterative computational technique helpful when require to solve the system of large and sparse linear equations. The benchmark is using the Poisson equation discretized on a 3D cubic domain with homogeneous Dirichlet boundary conditions. This benchmark performs domain decomposition, using an additive Schwarz method. Each subdomain is further preconditioned with a SymGS, one of the core numerical kernels of HPCG.

2.1.3.1 Sparse Matrix Representation

The matrix A is sparse and most of its entries are zero. In the context of the three-dimensional grid, every cell of the grid is connected with its neighbor, which gives the nonzero entry of the matrix A. The 27-point stencil, in a 3D discretization connects it with 26 immediate neighbours, including the 6 face neighbours, 12 edge neighbours, and 8 corner neighbours. This leads to a sparse matrix format where there is at most 27 non-zero value in each row represents each grid point with the current point and its neighbors.

2.1.3.2 Domain Decomposition and Process Layout

A 3D grid is the problem domain, which is sub-divided into smaller subdomains. These subdomains are distributed among multiple MPI processes to parallelize the computation. N_x, N_y and N_z are the dimensions of the local sub-domain and in the process layout NR_x, NR_y, and NR_z are number of MPI processes in x, y and z directions, respectively. The global domain size is thus given by $(NR_x \times N_x) \times (NR_y \times N_y) \times (NR_z \times N_z)$, and the total computation is divided among the $NR_x \times NR_y \times NR_z$ MPI processes, each handling a subgrid of the overall domain.

2.1.3.3 Stencil Operator

The 27-point stencil operator, illustrated in the Figure 2.2, demonstrates how a central grid point and its immediate neighbors are coupled in a 3D grid. Each dot represents a grid point, and edges indicate how each point interacts is coupling with its neighbors. This stencil is one of the most important factors in the HPCG benchmark because the sparse matrix operations rely directly on this stencil due to the definition of the structure of matrix A.

The HPCG benchmark is designed to emulate the behavior of real-world applications that solve large, sparse linear systems on 3D grids. At the center of defining the sparsity pattern of matrix A is the 27-point stencil operator, and efficient computation is divided among several MPI processes to leverage parallelism in high performance computing environments. This leads to a sparse matrix format where there is at most 27 non-zero value in each row represents each grid point with the current point and its neighbors.

2.1.4 Properties

The HPCG benchmark constructs a 3D partial differential equation model problem and uses preconditioned conjugate gradient iterations on the sparse linear system [4]. The characteristics are defined by input parameters, but some constraints are applied at the setup stage. The benchmark builds up a sparse linear system that is distributed with 27-point stencil for each of the grid points. This results in a matrix with specific properties [4, 5]:

1. **Nonzero Entries Per Row:**

$$\text{Nonzero entries per row} = \begin{cases} 27, & \text{for interior points} \\ 7 \text{ to } 18, & \text{for boundary points} \end{cases}$$

2. **Matrix Properties:**

The matrix is **positive definite**, **symmetric**, and **non-singular**.

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

where $A^T = A$ (symmetric), $\forall x \neq 0$ when $x^T Ax > 0$ (positive definite), and $\det(A) \neq 0$ (non-singular).

3. **Exact Solution Vector:**

   The exact solution vector x is known, with all elements equal to 1.0.

4. **Matching Right-Hand-Side Vector:**

   The right-hand-side vector b is constructed to match the exact solution.

$$A \cdot x = b.$$

5. **Initial Guess:**

   The initial guess vector x_0 is selected with all zeros.

2.1.5 Optimization Constraints

In the reference implementation of HPCG, some optimizations are allowed, but the authors who have developed HPCG, imposed some limitations and insisted for not modifying those aspects of the benchmark [4].

2.1.5.1 Allowed Optimizations

Mesh Partitioning/Reordering optimization is in fact the reorganization of the mesh points to help in the minimization of overhead that may arise in cases of data distribution between different processors and also help in the optimal use of cache in a hierarchical memory system.

User defined data structures eliminating different levels of abstraction and creating custom data structures tailored to the computational kernels can enhance memory access patterns and data locality and enhance the computational performance. When data structures are well defined then there is improvement in the efficient usage of memory.

System-Specific Communication Infrastructure Optimization can lead to the overall performance improvement. Utilization of specific network hierarchies and topologies tailored for hardware characteristics helps in improving the communication strategies.

Advance MPI communication if different types of MPI features are used, for example synchronization using neighborhood collective is an efficient communication pattern for specific application need [13]. Advance MPI features of course decreases the communication overhead and able to share data in the more efficient way which leads to the scalability on the large core counts.

Data storage format changes are permitted to sparse matrix data structure to improve the memory access but these changes for SpMV and SymGS kernels must not eliminate the indirect addressing of the input vector.

Computational Kernel Optimizations must have the same mathematical preconditioner, which must be capable of stressing the different component of

the system. Therefore the computational kernels coding which required special consideration for optimizing are:

- Compute DOT Product
- Compute WAXPBY
- Compute SpMV
- Compute SymGS
- Compute MG

2.1.5.2 Not Allowed Optimizations

Basic Conjugate Gradient (CG) Algorithm optimization by means of different variants of the CG method avoiding some challenging aspects of the classical algorithm. Some examples of the prohibited variants are Reordered conjugate-gradient methods [14–18] and Pipelined conjugate-gradient methods [19, 20]

Matrix Data Properties with prior knowledge concerning the pattern of sparsity, or the structure, or exploiting the discretization symmetry of matrix, and domain dimensionality information.

Spectral Properties information exploitation for the utilization of the optimal preconditioners or acceleration of its iterations unrealistically based on the known spectral properties of the matrix.

Data Representation Simplifications by applying the infrequency of the matrix pattern as regular or using near-regularity to the pattern. Reduction in the storage requirement by changing the defined precision or exploiting the symmetry in the data

is also prohibited.

Other modifications such as the optimizations which bypass the objectives of the benchmark are generally not permitted.

2.1.6 Core Kernels in HPCG

In HPCG, for solving $A \cdot x = b$, a PCG algorithm is used as described in Algorithm 1. These algorithms of the core kernels are presented in the simplest way to understand the task performed by these kernels, but the optimization of these kernels is not as simple as they seems simple in these algorithms because the data dependencies significantly complicate the optimization, when they use within the PCG. Section 3.1 summarize the efforts put in by researchers towards addressing the challenges in optimizations.

2.1.6.1 Dot Product (DDOT)

$$\alpha = x \cdot y \tag{2.1}$$

where α is scalar and the $x,\ y$ are vectors.

DDOT calculates the scalar result of two input vectors x and y, each of length n.

2.1.6.2 WAXPBY

$$w = \alpha \cdot x + \beta \cdot y \tag{2.2}$$

where α and β are the scalar values, x and y are input vectors, and w is the resultant vector.

WAXPBY is the weighted addition of two vectors x and y (scaled vector αx plus a scaled vector βy), also known as vector vector coefficient multiplication [21]. An

abbreviation come from the operation it performs as mentioned in equation 2.2. The operation takes the vector x, scales it by α, and the vector y, scales it by β and then add these two scaled vectors together and get the resultant vector w.

2.1.6.3 Sparse Matrix-Vector Multiplication (SpMV)

$$y = A \cdot x. \tag{2.3}$$

SpMV computes the product of a sparse matrix A with a vector x to produce a vector y. The reference implementation employs the Compressed Sparse Row (CSR) data format. However, there are various data formats discussed extensively in literature by many researchers. Some of the most common sparse matrix data formats are also mentioned in Section 4.1.

2.1.6.4 Symmetric Gauss-Seidel (SymGS)

The Symmetric Gauss-Seidel (SymGS) method is an effective iterative solver for sparse linear systems of the form:

$$A \cdot x = r, \tag{2.4}$$

where A is an $n \times n$ sparse symmetric positive definite (SPD) matrix, r is the residual vector, and x is the solution vector. The matrix A is decomposed into three components:

- L: the strictly lower triangular part of A,
- U: the strictly upper triangular part of A,
- D: the diagonal matrix containing the diagonal entries of A.

The SymGS algorithm consists of two main phases forward and backward sweeps, applied sequentially during each iteration. These steps aim to update the solution vector by attenuating high-frequency errors.

Forward Sweep: In the forward sweep, the updated solution is obtained using the lower triangular and diagonal parts of A:

$$(L + D) \cdot x^{(k+1)} = r - U \cdot x^{(k)}, \tag{2.5}$$

where $x^{(k)}$ is the solution at the k-th iteration.

Backward Sweep: The backward sweep then refines this update using the upper triangular and diagonal components:

$$(U + D) \cdot x^{(k+1)} = r - L \cdot x^{(k)}. \tag{2.6}$$

Residual Computation: The residual vector r is computed as:

$$r = b - A \cdot x^{(k)}. \tag{2.7}$$

Role in HPCG: In the HPCG benchmark, SymGS serves as a smoother within the multigrid preconditioner. Its primary objective is to suppress high-frequency error components that arise during iterative solution of the linear system, in the PCG method.

However, due to the inherent data dependencies in both the forward and backward sweeps, parallelization of SymGS is challenging. These dependencies limit concurrent updates to the solution vector, making SymGS less scalable on modern

multi-core and many-core architectures.

2.1.6.5 Multigrid V-cycle (MG)

Multigrid V-cycle is among the most effective iterative methods in solving large systems of linear equations. The method take advantage of mulltiple levels of grid resolutions by recursively moving between finer and coarser grids for better convergence by addressing high-frequency and low-frequency errors. This multigrid method (MGM) accelerates the convergence of coarser grids by reducing the errors efficiently, compared to conventional iterative methods.

MG takes a matrix A and a vector r as input and initializes a solution vector x to zero. If multigrid data available for the matrix A, the algorithm applies pre-smoothing using iterative solver SymGS. This helps to reduce the high-frequency errors that are present in the initial guess provided for x. The smoothing step is next followed by the computation of the residual. This residual is taken as the error in the current approximation. The residual is then restricted to a coarser grid, and on the coarser grid the low-frequency errors could be resolved. The restriction reduces the size which makes the solution faster to solve. The algorithm then solves the problem on the coarser grid recursively. This recursion allows the algorithm to repeat the same multigrid steps at each coarser level where the problem becomes small enough to solve easily. Once solved at the coarsest level, the solution is prolongated back to the finer grid. This prolongation step interpolates the coarser grid solution back to the finer grid, to enable it for further refinement of the solution.

Then post-smoothing is done on the finer grid by using SymGS again. This post-smoothing step will ensure that any high-frequency errors, that may be generated by the prolongation can reduced to get an accurate solution. If multigrid data is not

exist, then algorithm falls back to applying the SymGS smoother directly for solving the system without multigrid coarsening or prolongation. This fallback ensures that, even in the absence of multigrid the algorithm could still solve. The preconditioner utilized in HPCG is a V-cycle from the geometric multigrid methods. The total number of V-cycle levels are hard coded to 4 in the reference implementation of the HPCG as shown in Figure 2.3.

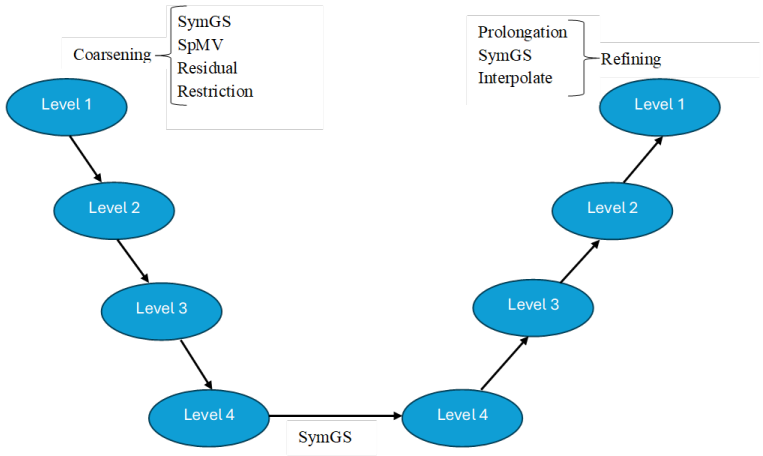


Figure 2.3 Geometric multigrid V-cycle preconditioner in HPCG

2.2 Kokkos EcoSystem

Kokkos [6, 9, 10] is a C++ template library and programming model that makes it possible to run programs on different architectures by abstracting parallel execution and data management and mapping high-level constructs onto backends like CUDA, HIP, SYCL, OpenMP, and threads. Kokkos is engineered to address complex node architectures featuring N-level memory hierarchies and various execution resource types. Its main abstractions are execution and memory spaces, execution patterns and policies, memory layouts, and these traits make it possible to specialize at compile

time for the target hardware. Kokkos provides a complete ecosystem as shown in Figure 2.4 for performance portability which includes math kernels, tools for debugging, profiling, and tuning, and community support through documentation, and tutorials.

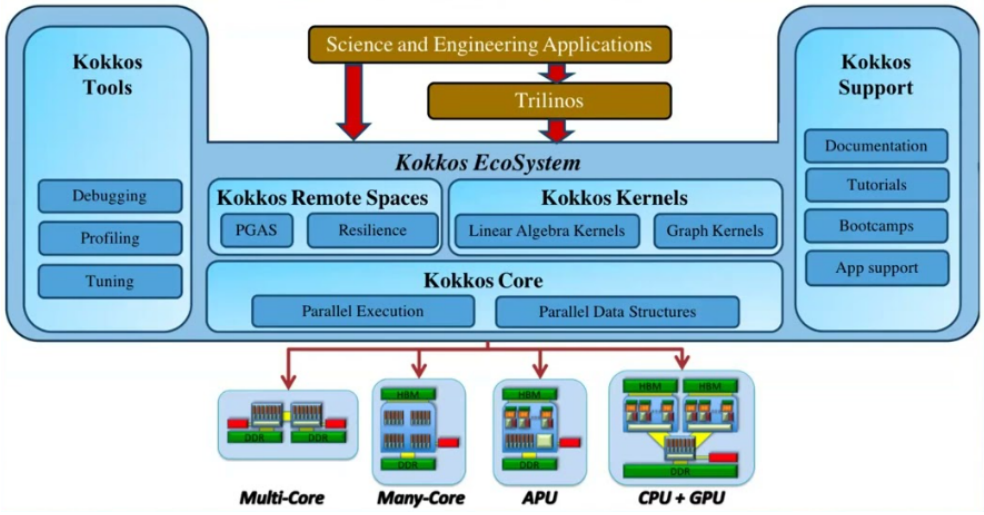


Figure 2.4 Overview of the Kokkos ecosystem including tools, core components, remote spaces, and kernel libraries. Adapted from [22].

Kokkos was initially created at Sandia National Laboratories as part of the Exascale Computing Project to fulfill the requirement for a unified codebase that can target CPUs, GPUs, and other accelerators without necessitating algorithmic rewrites. It decouples algorithmic intent from hardware specifics by providing compiletime abstractions for parallel execution and memory management.

2.2.1 Programming Model

- **Core Abstractions:** The programming model is based on abstractions: execution spaces, execution patterns, execution policies, memory spaces, memory layouts, and memory traits.

- **Execution Spaces:** Execution spaces (like CUDA, HIP, OpenMP, and Threads) tell where parallel kernels run. This lets code change according to target architecture just by changing template parameters.
- **Memory Spaces:** Specify where the data is stored (like CudaSpace or HostSpace).
- **Execution Patterns:** Parallel constructs like parallel_for, parallel_reduce, and parallel_scan.
- **Execution Policies:** Control execution behavior, such as range and team policies.
- **Memory Layouts:** Define the data arrangement in memory (e.g., LayoutLeft, LayoutRight).
- **Memory Traits:** Define properties like unmanaged or atomic access.

2.2.1.1 Sample Code Example

This simple 1D vector addition implementation show Kokkos programing structure for portable execution across backends for example with OpenMP for CPUs, CUDA for GPUs.

Kokkos Example (Portable)

```
Kokkos::parallel_for("VecAdd",
  Kokkos::RangePolicy<>(0,N),
  KOKKOS_LAMBDA(const int i) {
    C(i) = A(i) + B(i);
});
```

The Kokkos example hides the details of managing threads at a low level, so the same code can run on both CPUs and GPUs by choosing the right execution space at compile time.

2.2.2 Packages/Repositories

The Kokkos EcoSystem as shown in Figure 2.4 has a number of packages/ repositories, which are listed below.

- **Kokkos Core:** The Core library implements the above mentioned abstractions, assigning tasks to backends and controlling data lifecycle without causing runtime overhead.
- **Kokkos Kernels:** Kokkos Kernels is a library that works with Kokkos and provides sparse and dense linear algebra routines, batched BLAS/LAPACK, and graph algorithms. It works well on both CPUs and GPUs.
- **Kokkos Remote Spaces:** Remote Spaces adds a Partitioned Global Address Space (PGAS) model to Kokkos, which lets you use distributed-memory data structures with resilience capabilities.
- **Kokkos Tools:** The Tools interface offers mechanisms for debugging, profiling, and autotuning frameworks, presenting comprehensive runtime

information through a standardized callback APIs.

- **Kokkos Support:** Comprehensive documentation, tutorials, and bootcamps guide users in integrating Kokkos into their projects via CMake and GitHub repositories.

CHAPTER 3. Literature Review

This chapter provides an overview of the research that has already been done on optimization strategies for the HPCG benchmark. This chapter discussed architecture-specific optimizations, which cover CPUs, GPUs, hybrid systems, FPGAs, GraphBLAS and performance-portable frameworks like Kokkos. Chapter 4 contains details of sparse matrix data formats and parallelization optimization strategies discussed in the literature review.

3.1 HPCG Optimization Techniques

In HPCG, the most time-consuming kernel is SymGS, followed by SpMV because of its memory-bound nature. The optimization of HPCG primarily focuses on the optimization of these two main kernels. For optimization of SpMV, effective strategies are domain decomposition, parallelization, and the utilization of the efficient storage format. For SymGS optimization, parallelization techniques and hardware-specific tuning applied by the researchers are summarized in the following subsection. In both kernels, the hybrid parallelization approaches combine MPI (distributed memory) and OpenMP (shared memory) parallelization schemes. Hardware-based parallelization, such as vectorization through SIMD instructions and task offloading on GPUs or FPGAs, also significantly enhances the

performance of these kernels. Other than these, task load balancing, communication computation improvement, and cache-friendly data reordering techniques are also discussed literature. The multigrid method offers another avenue for optimization by subdividing the problems and solving them at multiple levels. The effectiveness of the optimization using these techniques necessitates the careful tuning of the problem based on the specific hardware architecture characteristics. Successful optimization of the HPCG benchmark required a careful combination of these techniques tailored to the target system and the properties of the sparse matrix used.

In this section, we group the optimization techniques discussed in the reviewed papers by architecture and year-wise. Since the introduction of HPCG in 2013, several optimizations have been done regarding different architectures.

3.1.1 CPU-Based Systems

3.1.1.1 Intel Architecture - IA

In 2015 [23] presented the updates on their intel-based work [24], which further tested on single and multi-node with the enhancements introduced in HPCGv3.0. They also focused on software upgradation such as optimized sparse matrix operations in the Intel Math Kernel Library (MKL) [25] and the open source SpMP library [26] for sparse matrix pre-processing and optimization of multi-grid implementation to enhance the performance of HPCG. The upgradation for HPCGv3.0 with these optimizations in the GenerateProblem and SetupHalo routines reduced the overhead to less than 3% and 4% on Haswell (HSW) and Knights Corner (KNC) architectures, respectively.

In 2016 [27] expanded upon the previous work [24] of the authors, optimized HPCG for Intel Xeon and Xeon Phi processors using techniques like point-to-point

synchronization, loop fusion, and hybrid parallelization schemes. The novelty of this work lay in the fusion of GS and SpMV, and these optimization techniques significantly reduced the amount of data transfer from memory by enabling data to be reused from the cache. [28] optimized HPCG for a CPU+MIC platform, with techniques like offload programming, vectorization, and parameter tunings. The author applied different optimization techniques, including the integration of an offload programming mode to use the Intel Xeon Phi coprocessors for computationally intensive kernels, vectorization using the Intel compiler auto-vectorization options, and parameter optimizations to find the best matrix size for MIC architecture. They were parallelized and optimized the four main kernels of the HPCG. Their approach allowed the CPU to handle less parallelizable tasks, and the MIC co-processors managed to execute heavily parallelizable routines. Further, the authors highlight the impact of heat dissipation from the devices on performance stability and develop an intelligent dynamic cooling solution for the MIC coprocessors to keep them at optimal temperatures and maintain their performance stability. They demonstrated that heat management is also critical for achieving stable performance on MIC coprocessors. They achieved significant speedup over the reference implementation.

In 2024 [29] optimized implementation primarily developed for ARM-based systems but also tested on an Intel Xeon system. They also tested their techniques on both single nodes and distributed cluster environments, scaling up to 256 nodes and 16,384 cores with 2048 MPI processes. The results showed nearly linear scalability and improved performance, particularly when incorporating asynchronous communication strategies.

3.1.1.2 Arm-Based Architectures

In 2019 [30, 31] presented the profiling of HPCG on an Arm-based platform Cavium ThunderX2. Demonstrated an optimized implementation of the HPCG benchmark with an emphasis on shared memory parallelization using OpenMP. In addition to a dynamic slicing technique for adaptive block geometry, the authors employed two primary optimization strategies: multi-color reordering and block multi-color reordering of the SymGS preconditioner. The optimizations presented in this paper [31] are identical to those that have already been discussed in the technical report [30]. The strategies outlined in this paper were specifically developed for the ARMv8.1 Cavium ThunderX2 processor, which features a shared memory architecture.

In 2023 [32] implemented a new HPCG based on the ALP/GraphBLAS [33] and tested it on ARM Kunpeng (920-4826) up to 7 nodes ARM cluster. It had scalability issues in distributed settings. The objective of this work was to leverage the algebraic abstraction and optimization capabilities of ALP/GraphBLAS [34], which is a C++ variant of GraphBLAS called ALP. The authors proposed a new implementation of HPCG based on ALP/GraphBLAS [35] and conducted the evaluation of its performance on both shared and distributed memory systems. The SymGS smoother was replaced with a Red-Black Gauss-Seidel (RBGS) to facilitate parallelism, and restriction and refinement operations were implemented as matrix-vector multiplications as the key design changes. Their implementation [35] outperformed in shared memory experiments on x86 and ARM architectures and encountered significant scalability limitations in distributed systems, compared to the reference implementation, mainly due to the communication overheads and the inability of GraphBLAS to efficiently manage data distribution.

In 2024 [36] focused on the theoretical development of the optimization techniques, and another paper [29] expanded on the practical implementation and provided the performance evaluation across different applications in detail. These two papers are closely related, and the research presented focused on optimizing the Multi-Grid Preconditioned Conjugate Gradient (MGPCG) method [37], which was subsequently applied to the HPCG benchmark. In this research, the authors developed novel techniques to optimize the SymGS and a block multi-color (BMC) scheduling method with point-to-point synchronization to improve the parallelism and the load balance. [29] tested their optimizations on Arm-based platforms, evaluated their optimized HPCG implementation, and compared them with vendor-tuned HPCG implementations on three different systems: Phytium 2000+, Kunpeng 920, and Thunder X2 ARMv8.

3.1.1.3 K Computer

In 2016 [38] presented the optimization on the K computer, focused particularly on single-node performance optimization of the HPCG benchmark, with several optimization techniques tailored to take advantage of the K computer's architecture. The authors utilized memory layout reorganization to achieve sequential memory access, which reduced the cache misses and improved its throughput. They also introduced data access improvements by aligning loop directions to optimize data locality and improve cache misses. Parallelization was done using multithreading and coloring methods to eliminate data dependency. Additionally, a blocked coloring technique was employed to preserve data locality within blocks for efficient multithreading of the SYMGS kernel, resulting in a substantial improvement in cache efficiency. These optimizations resulted in a significant performance improvement on the K Computers, and secured second position in November 2014 HPCG results

with 4.4% of ratio to HPL performance.

3.1.1.4 Sunway TaihuLight Supercomputer

In 2017 [39] presented the optimizations of HPCG on the Sunway many-core processor. The authors introduced a technique called the Hierarchical Grid (HG) algorithm, which they designed specifically for the Sunway architecture with an aim to enhance the performance of HPCG on the Sunway TaihuLight supercomputer. First, they used trivial methods to optimize the key kernels in MG V-cycle and SpMV, such as Level-Scheduling (LS) and Multi-Coloring (MC) methods used for the parallelism of the SymGS smoother. The authors proposed a new technique, HG, after realizing the limitations, such as poor locality and limited parallelism of the LS and MC. HG divided the domain into grids and subgrids mapped to the Computing Processing Elements (CPEs) cluster. They also implemented an efficient data prefetch mechanism and transfer scheme using DMA operations to manage data exchange between Management Processing Elements (MPEs) and CPEs. They used team collaborative computing to assign SpMV inner elements to CPEs and the border elements to MPE. The paper also demonstrated the scalability of their approach and comprehensive analysis of their parallel model and optimization strategies. They claimed that their approach is not only for HPCG but also for other HPC applications on the Sunway processor.

In 2018 [40] presented comprehensive optimizations, including block multi-coloring on Sunway TaihuLight to exploit hardware characteristics. The authors developed a series of optimization techniques for the HPCG benchmark on the heterogeneous many-core architecture of the Sunway TaihuLight supercomputer. Due to the high bandwidth requirements of HPCG, the main challenge was to improve the

performance of memory-bound kernels by leveraging the specific architecture of Sunway TaihuLight, which has limited memory bandwidth as compared to its computational power. The key optimizations employed in this study included a block multi-coloring approach for parallelizing the SymGS kernel, which balanced parallelism and convergence rate. Parallelism increased by dividing the computations into blocks that can fit into the LDM of CPEs while maintaining the data locality. They also implemented locality-aware layout transformations to improve data access patterns by transforming the sparse matrix storage format into ELLPACK and vectors access reordered to align with the parallelism scheme and improved the access efficiency by grouping blocks with the same color. They also developed a requirement-based data access method that mapped only necessary data for the limited local memory of each core, which reduced the data movement overhead. The required data for computations was accessed through DMA transfers, while the on-chip register communications were used to exchange data between CPEs to enhance efficiency. The researchers further decomposed operations into smaller tasks to enable fine-grain overlapping of computation and data access. Additional optimizations included code transformation, SIMD vectorization, index compression, register message combination and local data management. This work has shown that with careful optimizations used in this study, memory-bound applications like HPCG, even on architectures with challenging memory bandwidth constraints, can efficiently scale on large systems.

3.1.1.5 OceanLight Sunway Supercomputer

In 2021 [41] presented a series of optimization techniques intended to enable and scale the HPCG benchmark on the new generation of the Sunway supercomputer, which was equipped with over 42 million heterogeneous cores. Instead of using

multi-coloring or block multi-coloring techniques for parallelism, the authors introduced a novel two-level blocking technique to exploit parallelism in the SymGS kernel and maintain the convergence rate. This is the first paper that uses this technique to optimize the HPCG on the Sunway supercomputer. Further, they also proposed a fine-grained kernel fusion scheme that improves the data locality to alleviate the bandwidth load on local storage and another notable work was a low overhead thread coordination mechanism that transfers data between the cores. They used a simplified ELLPACK sparse matrix format for better memory alignment. These optimizations enabled to scale up to 653,760 MPI processes with 95.5% efficiency, and the optimized implementation scaled to over 42 million cores and maintained a performance of 5.91 Pflops, utilizing 73.0% of the theoretical memory bandwidth. The performance was further enhanced to 27.6 Pflops by relaxing the constraints of the HPCG benchmark. This work [42] presented effective optimization strategies for sparse linear solvers on modern heterogeneous supercomputer architectures.

3.1.1.6 NEC SX-ACE Vector Supercomputer

In 2015 [43] explored various optimization techniques for HPCG benchmark on the SX-ACE supercomputer [44]. To take advantage of the architectural features of NEC SX-ACE as the vector parallel processor with a high-bandwidth memory system, the authors employed different data packing formats, including CSR, JAD [45], and ELLPACK [46] for efficient packing of sparse matrix data and found ELLPACK as most effective for its vector calculations and memory access efficiency. To eliminate data dependencies during parallelization, eight-color multi-coloring [47] and hyperplane [48, 49] techniques are also employed. They gained performance improvements using a combination of JAD+coloring, ELLPACK+coloring,

Hyperplane with selective caching in the available on-chip Assignable Data Buffer (ADB), and problem size tuning. [43] presented a series of optimizations on the SX-ACE vector supercomputer, and their optimized implementation achieved 11.4% efficiency in the case of using 512 nodes and over 30 Gflops on a single node.

In 2023 [50] presented an optimized HPCG implementation for long-vector architectures in order to achieve a performance on high-end RISC-V accelerators, mainly through kernel optimizations on enhancing memory hierarchy usage. This work applied several optimizations to the HPCG benchmark and was of great importance for the domain of HPC because it optimized the HPCG benchmark for long vector architectures, targeting specifically the NEC VE and the RISC-V vector extension (RISC-VV) platforms. The paper presented a portable and highly optimized implementation of HPCG as open source [51] to long-vector architectures.

3.1.1.7 Near-Data Processing (NDP) Architecture

In 2017 [52] discussed the use of IBM Power8 near-data processors (NDPs) [53] in the optimization of HPCG and Graph500 [54] benchmark. The Graph500 benchmark focuses on breadth-first searches (BFS) in large graphs and stresses memory access and global communication. The optimizations of the Graph500 benchmark are not discussed as they are not part of the scope of this paper. For detailed information on it, please refer to the original paper [52], and the distributed Graph500 details can be found in [55]. The researchers employed a series of optimizations for the HPCG using NDP architecture. They designed a system of 8 NDPs, which contain multiple small and slow cores positioned close to the memory. The architecture also utilized a shared memory approach with coherent access across the NDPs. They replace the traditional MPI + OpenMP approach with a nested OpenMP model for parallelization,

spawning threads for each NDP and its cores. They also restructured the code to create a single parallel region encompassing the kernels instead of employing thread teams, which significantly reduced the threading overhead. They determined that a 4 KB data cache per NDP core was optimal to optimize the data locality and cache. They tested different memory access granularities and found that a 64B access granularity, combined with software prefetching, provided the best results when using DDR3200 memory. They implemented the software prefetching for Dot Product and WAXPBY kernels due to their data access patterns. However, the prefetching was more challenging due to data dependency in SpMV and SymGS kernels and the blocked multi-coloring approach for parallelism in these kernels. This study highlighted the importance of inter-NDP bandwidth, which became equally important as local memory bandwidth for the optimization of the applications. They optimized inter-NDP communication for high bandwidth and low latency, utilizing an NDP Access Point (NDP-AP) for efficient remote data access.

3.1.2 GPU-Based Systems

3.1.2.1 GPU-Accelerated Systems

In 2014 [56,57] presents an optimized HPCG benchmark using CUDA for NVIDIA GPU-accelerated supercomputers, namely Titan and Piz Daint. The key optimization technique of this paper is graph coloring to enhance the parallelism of the SymGS smoother. They employed the parallel coloring technique with the local maxima [58, 59] and incorporated improvements proposed by [60]. They primarily focused on parallelizing the SymGS. They used cuSPARSE library [61] and customized CUDA-based kernels, and switched matrix data format from CSR to ELLPACK so that the memory access coalesced, which was essential for optimizing GPU

efficiency. This paper contributed to optimizing memory-bound workloads on GPUs. They benchmarked their optimized HPCG variant on both the Cray XK7 system at Oak Ridge National Laboratory (ORNL) [62] and the Cray XC30 system at the Swiss National Supercomputing Centre (CSCS) [63]. The Cray XK7 (Titan) has an AMD Opteron processor and a Gemini interconnect that has a 3D torus topology [64]. The Cray XC30 (Piz Daint) features an Intel Xeon processor and an Aries interconnect that has a dragonfly topology [65]. This was the first CUDA implementation of HPCG for GPUs, focused on parallelizing the SymGS smoother using graph coloring techniques, and their implementation achieved the fastest per-processor performance reported at that time when tested at full scale on large GPU-accelerated supercomputers like the Cray XK7 at ORNL and the Cray XC30 at CSCS.

In 2016 [66] expanded on the aforementioned research with a more comprehensive examination of the HPCG benchmark covering a broader range of GPU architectures, incorporating improved optimization techniques, and providing a more detailed analysis. Furthermore, it highlights the differences in the efficiency and prospective of GPU and CPU executions, which have not been thoroughly examined in their previous research.

3.1.3 Hybrid Architectures

3.1.3.1 Tianhe-2 Supercomputer

In 2014 [24] is one of the first papers to have focused on the optimization of the HPCG benchmark for the multi and many-core architecture and has achieved the performance of 580 Tflops on the Tianhe-2 supercomputer. They achieved this by utilizing an approach that combines both multi-core and many-core Intel Xeon and Xeon Phi co-processors. SpMV and SymGS are the core kernels of

many solvers [67, 68] including HPCG. Achieving high performance of symGS smoother is particularly challenging due to its limitation in fine-grain parallelism [69] as it is inherently sequential in reference to the implementation of the HPCG. Focusing on the essential SymGS smoother, they uncover and evaluate significant limitations of the parallelism and introduced a novel hybrid approach combining the point-to-point synchronization in sparsification and block multi-color reordering technique Algebraic Block Multi-Coloring (ABMC). To optimize the data locality and memory access patterns, they also used the SELLPACK sparse matrix data format. [70] focused on improving only the CPU-based system on 6,144 nodes by utilizing techniques such as red-black relaxation, SIMDization, loop unrolling, forward-backward sweep fusion, and OpenMP parallelization, including a reformulation of the mathematical equivalent CG algorithm to minimize collective communication costs. They also replaced the default CSR format with a simplified SELLPACK format, a variant of ELLPACK [71], which improved data locality and access patterns in SpMV and SymGS. They also fused the residual computation in SpMV and restriction operation into a single subroutine in geometric multigrid v cycle inspired by the work [72]. Compared to prior work [24] on 12-core Intel Xeon processors, this optimized HPCG achieved both higher single-CPU performance and superior large-scale performance on Tianhe-2. [73] was an extension of the previous study [70] for hybrid CPU-MIC based architecture on the Tianhe-2 platform. Their previous work focused only on CPU-based optimization, while in this study, they leveraged both Intel Xeon CPUs and Intel Xeon Phi coprocessors (many integrated cores) MIC resources. Key optimizations include inner-outer subdomain partitioning, asynchronous data transfer, red-black relaxation parallelization, and optimized workload distribution across both CPU and MIC cores. [74] highlighted the importance of multi-coloring techniques for Gauss-Seidel with SIMD-friendly sparse

matrix formats. [75] were especially concerned with improving the communication in computer systems, and their solution involved pipelined CG variants, and that kind of optimization was not allowed, as mentioned in Section 2.1.5.2. Although all these papers achieved significant improvements in performance compared to the reference implementation, they varied in their emphasis: [24] and [73] selected the use of heterogeneous structures, [70] and [74] optimized for CPU-only systems and [75] focused on communication efficiency. These works laid a solid groundwork for optimizing HPCG and other applications on the world' s most powerful supercomputers, including Tianhe-2.

In 2016 [76] has optimized HPCG on Tianhe-2 using CPU + MIC heterogeneous architecture. Based on the previous work [70, 73], the paper [76] presented a comprehensive hybrid CPU-MIC algorithm for optimizing HPCG on the Tianhe-2 supercomputer. Key innovations included an improved inner-outer subdomain partitioning strategy that better optimized the workload between the CPU and MIC while reducing the amount of data transfer and a fused scheduling technique that overlapped the computation and communication. The researchers employed forward and backward fused algorithms and block multicolor parallelization techniques for the SymGS kernel.

3.1.4 Other Architectures and Environments

3.1.4.1 FPGA-Based System

In 2021, the first known approach for reconfigurable hardware implementation of HPCG was presented by [21]. They use different optimization techniques for FPGA platforms, such as the Xilinx Alveo U280 FPGA. These optimizations include memory access optimization using CSR format, with a modification from the standard

HPCG implementation by packing data into 512-bit for efficient memory reads. Their implementation also supports multiple numerical precisions (double, single, half) where some acceleration offers nearly linear performance. The Berkeley Roofline model was used by the authors to validate their optimizations and show near-optimal performance for Xilinx Alveo U280. They also emphasized that the performance is mainly constrained by the memory bandwidth. The main contribution of this work lies in its demonstration of the applicability of FPGA for HPCG, originally designed to be run on CPUs and GPUs. This is an efficient, scalable implementation of the HPCG benchmark that challenges CPU and GPU architectures. Their FPGA design exhibited better power efficiency than GPUs and CPUs. They utilized HBM memory and customized data patterns for FPGA architecture.

In 2022, a bachelor student, Rahul Steiger's thesis [77] presented an implementation of the HPCG benchmark by optimizing FPGA-specific libraries of key kernels in Python and then integrating them in an optimized version [21] of HPCG using the DaCe framework, but the implementation was significantly slower than existing version [21]. However, it laid the groundwork for future HPCG implementations in Python by demonstrating how specialized kernel implementations could be incorporated without modifying the high-level Python code and provided valuable insights into the efficiency of the approach, potentially facilitating broader adoption of FPGAs in HPC applications. He also conducted a performance comparison between the FPGA implementation and other FPGA and CPU-based versions of HPCG.

3.1.4.2 Kokkos-Based

In 2016 [7] was a thesis work of a student from the College of Saint Benedict and Saint John's University. It presented the development of KHPCG [8], which

was the first try for creating a performance-portable version of the HPCG benchmark using KOKKOS library [6,9,10], which is a hardware abstraction library. The author replaced custom data types and parallel loops with Kokko's multidimensional arrays and also used KOKKOS parallel dispatch to replace the existing parallel loops. This work focused on optimizing the HPCG benchmark to enhance its performance on CPU and GPU architectures, with particular attention paid to hybrid systems that include both types of processors. Two parallelization strategies for SymGS were implemented, and the performance of these different preconditioning parallelization approaches using levels and coloring techniques across OpenMP and CUDA was compared, which showed that the coloring algorithm generally outperformed the leveling algorithm. This work provided a basis for future work on creating a more performance-portable reference implementation of HPCG that can be easily optimized for different architectures. Its performance can presumably be stable across different hardware platforms. However, the report [11] highlights the two primary challenges in the practical implementation of the KHPCG. The utilization of the coloring approach used for parallelism in KHPCG has raised concerns regarding the validity of the results. Furthermore, due to compatibility issues with CUDA 11, the sparse matrix-vector routine in the cuSPARSE module of Kokkos was deprecated. These challenges highlight the necessity of a thorough restructuring of the KHPCG in order to ensure that KHPCG functions optimally as a benchmark tool across different platforms.

3.1.5 HPCG Benchmark Implementation Variants

The reference HPCG benchmark implementation can be found in the project git repository [78] and on the official web [88]. Its latest release is 3.1, and we call it native HPCG implementation. Open source and some Vendor-optimized variants modified based on the

Table 3.1 HPCG benchmark variants and implementation references

Sr#	Variants	References	Reference HPCG Version Used
1	Native	[78]	N/A
CPU-Based			
2	IBM	[79]	HPCGv2.4
3	Intel CPU	[80]	HPCGv3.0
4	ARM	[81, 82]	HPCGv3.0
5	Sunway	[42]	HPCGv3.1
6	ALP/GraphBLAS	[35]	HPCGv3.1
7	VE native	[51]	HPCGv3.1
GPU-Based			
8	Intel GPU	[83]	HPCGv3.1
9	NVIDIA	[84, 85]	HPCGv3.1
10	AMD ROCm	[86]	HPCGv3.1
Others			
11	FPGA Xilinx	[87]	HPCGv3.1
12	KHPCG	[8]	HPCGv2.4

older versions of the reference/native HPCG code listed in Table 3.1.

The optimizations are aimed at taking advantage of customization for the vendor-specific platform and providing performance improvements to the specific architectures. Out of these variants, Intel and NVIDIA implementations are not fully open source because they used their architecture's specific optimized kernels to enhance the overall performance of the HPCG benchmark. Those kernels are proprietary and are designed for their respective platforms. ARM, FPGA Xilinx, IBM, AMD ROCm, and others offer open-source variants and are available for modifications. The details regarding the optimization of different HPCG benchmark variants have been provided in Section 3.1 in much detail. To delve deeper into the specifics of each variant, it is recommended to refer to the respective research works.

- ARM Optimized Variant: [30, 31, 89]
- Sunway Optimized Variant: [41]
- ALP/GraphBlas Optimized Variant: [32]
- FPGA Xilinx Optimized Variant: [21]

- VE Native Optimized Variant: [50]
- KHPCG Variant: The Kokkos-based HPCG variant [7]

However, for the IBM-optimized variant, no particular research paper has been written. Some details are presented in [79], according to which the IBM research group fine-tuned the CPU-only variant of HPCG for target processors such as IBM BGQ and POWER9 systems. Using the reference HPCGv2.4, they analyzed that there are some issues with the coloring method that decrease convergence speed and cache efficiency in the presented work [90]. To resolve these problems, they employed a stencil discretization technique, which led them to achieve performance improvement by rearranging the data into a uniform diagonal matrix structure. They also work on architecture-specific fine-tuning and in the enhancement of the backward prefetching of the SymGS smoother. Similarly, some details are found in [91,92] for HPCG optimization work performed on the AMD ROCm platform for AMD GPUs, which aims at enhancing compute intensity and scaling performance. This includes the usage of HIP (Heterogeneous-Compute Interface) for GPU offloading, managing memory access in a more efficient way, and device-specific tuning to improve performance. Moreover, the optimization of HPCG for AMD processors focused on memory bandwidth optimal usage, parallelism using OpenMP, as well as on optimizing the data locality using strategies such as task scheduling and eliminating synchronization overhead to increase the performance of key kernels SpMV and SymGS.

3.1.6 Summary

The high-level key differences in architecture-specific implementations are as follows: GPU implementations focused on massive parallelism and graph coloring techniques. CPU versions emphasized loop/kernel fusion, improved data layouts, vectorization, OpenMP, and multi-coloring techniques for parallelism. FPGA designs

customized memory access and compute paths. VE architecture aimed at long-vector processing was supported by efficient utilization of memory hierarchy and vectorized computation to increase the efficiency of vector machine architectures such as NEC VE. GPUs achieved the highest raw performance, while FPGAs showed the best power efficiency. The diverse optimization techniques have different strengths and challenges tailored for different architectures on various supercomputers, including Tianhe-2, K computer, Titan, Mira, Piz Daint, and Sunway TaihuLight, etc.

3.1.7 Supplementary Influential Works

The original authors of the HPCG benchmark, Jack Dongarra, Michael Heroux, and Piotr Luszczek, have published several updates regarding the development and enhancements of HPCG. These updates documented in publications such as [3–5], highlighted the evolution of HPCG to accurately represent the computational characteristics of modern scientific applications. In [3], they examined the influence of the HPCG on the HPC community after one year. Subsequently, the studies conducted in 2016, [4, 5] discussed the enhancements and improvements in HPCG. Besides the aforementioned literature on optimizing HPCG, some workshops and conference presentations have also been contributed, but most of their full content is not easily accessible. For example, [93–95] are cited by various research papers, marking them as influential in HPCG optimization. While the full content of these is not publicly accessible, their repeated citation in the literature underlined their importance in shaping the research and optimization direction of HPCG. Also, the paper [96] optimized the conjugate gradient using a pipelined algorithm of conjugate gradient on CPU+GPU architecture. As this kind of optimization is not allowed for the optimization of HPCG benchmarks, we skipped the details as they are not aligned

with the scope of the study.

CHAPTER 4. Technique and Trends in HPCG

This chapter discuss the different data formats. It also goes into more detail about the significant parallelization approaches employed by the other researcher, as mentioned in the Chapter 3.

4.1 Data Formats and Storage Strategies

4.1.1 Common Sparse Matrix Formats

Data format for sparse matrix representation is very important in the optimization of the SpMV. Some of the most common and basic data formats are listed in the Table 4.1 and illustrated in Figure 4.1

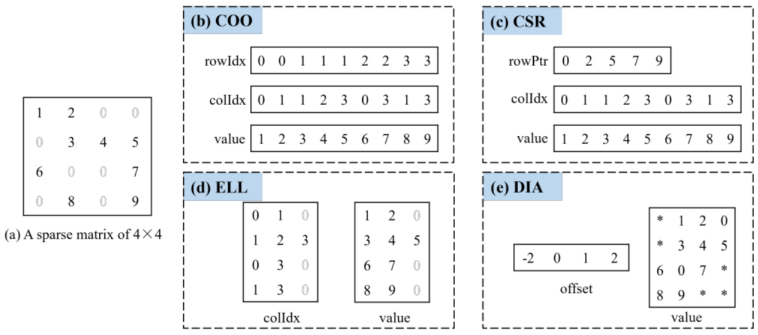


Figure 4.1 Illustration of the basic data formats. Reproduced from [98]

Table 4.1 Common sparse matrix data formats

Name	Abbreviation	Description
Coordinate	COO	COO format stores three arrays of row indices, column indices, and the values of non-zero entries of the matrix A. It is the simplest and most flexible format but is not memory efficient for large matrices.
Compressed Sparse Row	CSR	CSR also uses three arrays which include the values array, column indices array, and the row pointer array, and it relatively offers good performance because of memory efficiency.
Compressed Sparse Column	CSC	CSC is like CSR but column oriented. It is the transpose of CSR and is more useful for column-wise operations.
ELL	ELL	ELL format stores two arrays of the same size, values array, and the column indices array. It organizes the non-zero elements in each row into a fixed-length array and pads shorter rows with zeroes if necessary. More efficient for matrices with a bounded number of non-zeros per row and enables efficient parallel processing and vectorization but can lead to significant memory overhead for matrices with varying row lengths.
Diagonal	DIA	DIA stores diagonals in a separate dense matrix, suitable for matrices with a banded structure, reduces memory footprints, and allows fast access to diagonals but is less efficient for general sparse matrices.
References: [97, 98]		

There are also different variants of these basic data formats, such as sliced, blocked, and hybrid variants. Sliced variants usually improve the load balancing and parallelization of the operation, while blocked versions enhance the cache utilization and vectorization. Hybrid versions leverage the benefits of multiple formats. Other than these, some variants with bitmask and compression techniques are used to

reduce the memory footprints. Some specialized format variations are also used to handle sparse matrix irregularities for specific optimizations based on the types of sparse matrices. These formats help balance storage, computation, and matrix pattern or architectural compatibility. [98] conducted a comprehensive survey on sparse matrix-vector multiplications. For more detailed insights and a thorough understanding of these data formats, please refer to their work [98].

Sparse matrix operations in other applications and in HPCG are affected by these formats. These formats help in solving sparse matrix processing problems like computational performance, memory optimization, load balancing, and hardware-specific tunings. The use of these specific formats depends upon the characteristics of the matrix and the target platform.

4.1.2 Novel Data Structures for HPCG

The reference implementation of the HPCG benchmark uses the CSR data format. Upon reviewing the researchers' work on HPCG optimization, we found that most of them identified ELLPACK or its variant, Sliced ELLPACK (SELLPACK), as the primary data format for enhancing the performance of benchmark. Some of them restructured the format into other formats like JAD, DIA, etc., but they also confirmed that the ELLPACK is most suitable, especially for the vector processing architectures. The Table 4.2 summarize the papers reporting about the use of these data formats and provides a list of papers which investigate the application of these data formats for the optimization of HPCG.

The choice of data format has a great impact on both performance and memory consumption in HPCG. ELLPACK though introduced memory overhead due to fixed length arrays, but enhancing parallelization and reducing memory access latencies.

Table 4.2 Data formats reported in literature of the HPCG optimization

Data Formats	**Reference**	**Architecture**
ELLPACK	[40, 41, 43, 50, 56, 66, 73–76]	Tianhe-2 Supercomputer, SX-ACE supercomputer, Hybrid CPU-MIC, Intel Xeon multi-core processors and Xeon Phi many-core coprocessors, GPU-focused (NVIDIA CUDA), Tesla K20X, K40 GPUs, Cray XK7, XC30, Sunway supercomputer, Sunway TaihuLight Supercomputer, NEC VE and RISC-VV
SELLPACK	[24, 27, 70, 73, 76]	Titan (Cray XK7) and Piz Daint (Cray XC30) supercomputers, Hybrid CPU-MIC
JAD	[43]	SX-ACE supercomputer
DIA	[74]	SX-ACE supercomputer, Tianhe-2 Supercomputer
CSR/ Modified CSR	[7,21,28,29,31,32,36,38, 39, 43, 52, 66, 77]	SX-ACE supercomputer, K Computer, GPU-focused (NVIDIA CUDA), Tesla K20X, K40 GPUs, Cray XK7, XC30, Hybrid CPU-MIC, Near-data processing (NDP) architecture, Sunway many-core processor, ARMv8 (Cavium ThunderX2), FPGA Xilinx Alveo U280, IA x86

Overall, the use of these formats, together with parallelization techniques, brings significant memory efficiency and improve performance of HPCG.

4.2 Parallelization Optimization Techniques

The parallelization approaches are of most importance, which is generally difficult to optimize because of sparse matrix and dependencies. MPI, OpenMP, CUDA, pipelining and vectorization indeed serve as fundamental parallelization solutions which already employed in HPCG. MPI for distributed memory parallelism, OpenMP for the shared memory parallelism, CUDA for GPU based parallelism and Vectorization for vector-based architecture parallelism are already in use and most of these require additional optimization strategies. These parallelization techniques are effective in HPCG depending with the type of hardware and problem size to be solved. Hence, it is often the case that a combination of techniques like coloring for intra-node parallelism, MPI for inter-node communications, and pipelining to have multiple operations overlap, are employed on different supercomputing architectures. Coloring appears to be most effective in respect of parallelizing the main kernels of HPCG, SpMV and SymGS smoother. The coloring technique assist to achieve a good trade-off between the number of independent tasks parallelism and the rate of convergence for the solution. Scheduling pattern of these tasks also influence on the HPCG parallelization. Performance of the parallel computations can be improved providing they manage to correctly assess the dependencies of different computational tasks and properly schedule them. This is particularly important in the multigrid component of HPCG as the operations in each of the levels of the grid have certain degree of dependencies.

4.2.1 Coloring

The basic concept of the coloring technique is the distance-1 or distance-2 coloring discussed in the literature [99, 100]. The distance-1 coloring technique, also known

as vertex coloring, ensures that no two adjacent vertices have the same color, which helps to identify sets of vertices for parallel processing. In contrast, the distance-2 coloring technique is an extension of distance-1 coloring, ensuring that two vertices within a distance of 2 do not have the same color.

Multi-Coloring (MC):

In multi-coloring, the vertices are divided into multiple color classes as shown in Figure 4.2. There are different variants of multi-coloring techniques, and the data associated with the same color can be processed in parallel. The colors are assigned in such a way that they help in balancing the parallelism and convergence in the iterative solvers.

Red-Black (RB) Coloring:

In multi-coloring, red-black coloring is the basic and most well-known coloring technique used in the optimization of the SymGS smoother in multigrid preconditioner. In red-black coloring, the grid is divided into two colors, red and black, alternatively, where adjacent nodes follow a chessboard pattern as shown in Figure 4.3. Parallel processing is performed such that the red colors

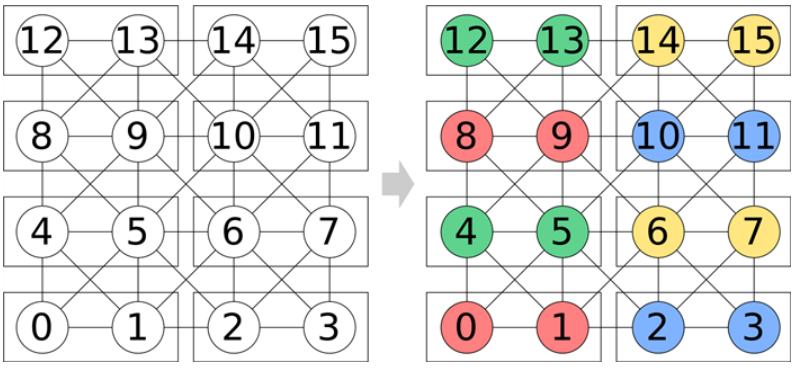


Figure 4.2 Multi-coloring, reproduced from [89]

have no dependency on the black color data sets and can be processed in parallel independently. It is a very common technique in structured grid-like iterative problems because it is simple to implement and allows for efficient parallelization by preserving the convergence rate in iterative methods like Gauss-Seidel. This technique is mostly suitable for multi-core architectures and is effective for small-scale parallelism with a slow convergence rate.

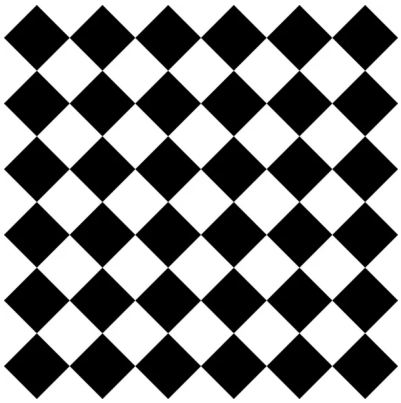


Figure 4.3 Chess board pattern, red black coloring

Multi-Coloring with 4 or 8 Colors:

The 4 or 8 colors multi-coloring approach is suitable for many-core architectures, providing relatively better parallelism than RB with improved convergence. This technique is more complex than RB due to potential cache locality issues. The performance and efficiency of the 4-color and 8-color multi-coloring techniques are different. The 4-color outperforms the 8-color in HPCG [74] as it takes fewer iterations for convergence, which means faster computation with less overhead, hence being efficient and scalable for large problems. In contrast, the 8-color technique takes more iterations, which increasing the overhead, and impacts on the convergence, as the problem size increases.

Block Multi-Coloring (BMC)

In the block multi-coloring technique as shown Figure 4.4, groups of vertices are blocked such that data arrangement within the blocks remains sequential to improve data locality. Coloring is applied to the blocks instead of individual vertices. This technique reduces coloring overhead and enhances cache performance in hierarchical memory systems. It is more complex to implement and requires careful selection of block sizes.

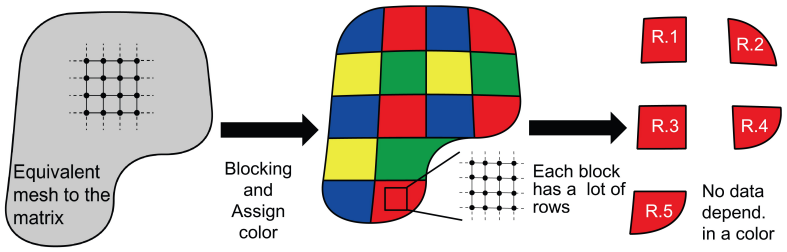


Figure 4.4 Block Multi-Coloring, reproduced from [38]

Algebraic Block Multi-Coloring (ABMC)

The Algebraic Block Multi-Color (ABMC) [47] is a parallel processing technique while solving large sparse linear systems of equations. ABMC as shown in Figure 4.5 involves two main steps, blocking and coloring algorithmically proposed in [101]. Blocking divide the matrix into sub matrices which helps in placing the data in sequence that is complementary to the memory caches for fast access and processing of data. Coloring applied to these blocks with different colors in order to reflect their relative dependence upon one another. The blocks of the same color do not have dependency on each other, so one color block can be handled at a time through parallel threads efficiently. It enhances computational effectiveness because numerous blocks can be tackled at once which shortens the time taken to solve the problem. This is especially helpful in situations where there is the use of iterative solvers with large

sparse matrices as encountered in scientific and engineering computations [24].

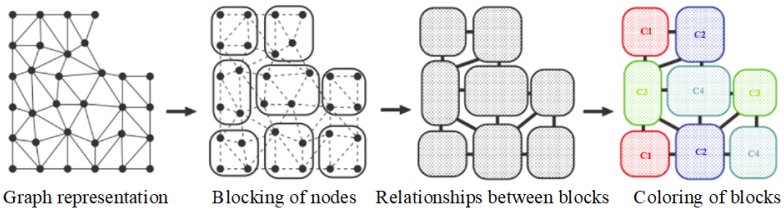


Figure 4.5 Algebraic block multi-coloring, reproduced from [47]

Hybrid Coloring

In the hybrid coloring technique, different coloring techniques are combined to balance data load and parallelism according to the computational power of different nodes in heterogeneous systems. These techniques are merged based on the target architecture.

In HPCG, these coloring techniques are used to create independent sets of matrix rows/columns that can be processed in parallel. This improves the performance of sparse matrix operations and multigrid preconditioners. In HPCG, coloring techniques help improve parallelization, load balancing, and convergence rate by modifying memory access patterns, which aids in better cache access in complex memory hierarchies. As a result, this improves system scalability. The choice of coloring techniques in HPCG can vary depending on the target architecture, problem structure, data sparsity pattern of the matrix, and the required optimization trade-off between parallelism and convergence rate. It is crucial to note that the choice of coloring technique in HPCG optimization is coupled with other optimization techniques and data formats.

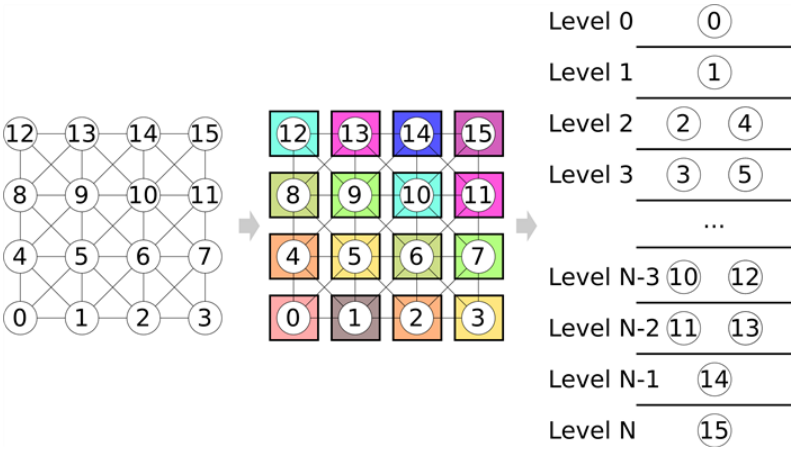


Figure 4.6 A Multi-level task dependency graph, reproduced from [89]

4.2.2 Multi Level Task Dependency Graph

Instead of relying on coloring, this technique creates a multi-level task dependency graph (TDG) whose nodes correspond to the elements of the grid, and the edges correspond to the dependencies among these elements. A graph is divided into levels as shown in Figure 4.6, which are processed one after the other, so that the computation order requirement is satisfied. The levels range from 0 to N. Because elements of the same level do not depend on each other, and all can be computed simultaneously. Unlike coloring techniques this scheme respects the data dependency order and therefore does not require any number of additional iterations to achieve the same residual. This approach, however, changes the order in which the data required to process. Unlike the Gauss-Seidel method, which strictly follows a row-by-row sequence, this one calculates any row whose dependencies are met; hence, several rows can be processed at one time. It speeds up some computations, but it affects spatial and temporal locality negatively, thereby slowing down the overall performance. Another issue is of the variation in the parallelism of the different levels, with little or no parallelism in the beginning and towards the end, but more in the

middle.

4.2.3 Hyperplane

The hyperplane parallelization technique is accessing the elements in a diagonal fashion so that elements in a 'hyperplane' can be proces in parallel, by avoiding the data dependencies. Unlike the multi-coloring techniques, in which all the elements with the same color are processed in parallel, in the hyperplane parallelism mechanism the calculations are done diagonally across an array. This helps remove data dependencies that would cause bottlenecks on parallel processing and, therefore, facilitates higher parallelism. By using this technique, performance improve due to the faster convergence, which results in an overall improvement in computational efficiency. This technique was applied in the optimization of HPCG for SX-ACE supercomputer by [43].

Figure 4.7 Hyperplane (2D), reproduced from [43]

4.2.4 Hierarchical Grid (HG)

The Hierarchical Grid (HG) technique as shown in Figure 4.7 is a novel approach which discretizes computational domain into a cascaded structure of sub-grids that

would mimic the physical layout of the Core Processing Elements (CPEs). The technique involves several key steps: sub-domain partitioning where the complete grid is split into 64 divided sub-grids or CPE mesh cluster pattern having 8 CPEs each; a special pattern of execution which maintains parallelism and data dependency on each other; and, the management of data to use the little available LDM in each CPE to the most efficient it can be. The HG technique enhances the data transfer and pre-fetching mechanism to reduce the overhead of DMA operation and also syncs CPEs data for data consistency. This is the way how partitioning strategy helps to achieve parallelism and get benefits connected to locality. This technique introduced by [39] to enhance the parallel execution of computations on the structure of the Sunway processors.

4.2.5 Two-level Blocking Scheme

The two-level blocking scheme was introduced in [41] as a novel method for rearranging the sparse matrix for the SymGS kernel in HPCG. The objective of this approach was to demonstrate sufficient parallelism while simultaneously maintaining a fast convergence rate. In the first level, the sparse matrix was divided into m large blocks, which were referred to as layers, in accordance with the original order. In order to preserve their interdependence, these layers were processed sequentially. At the second level of each specific layer, the directed graph derived from this layer was subjected to a graph partitioning method, which was subsequently followed by a coloring process on the generated blocks as shown in Figure 4.8. This ensured that, in parallel processing, the smaller blocks of the same color can be processed independently of the other differently colored blocks. To prevent performance degradation, these blocks were designed to be as large as possible, due to the limited capacity of the Local Device Memory (LDM) . Data

exchange and synchronization were required prior to transitioning to the subsequent color. This method outperformed other methods, such as multi-coloring and block multi-coloring, by maintaining a balance between parallelism and convergence rate. Furthermore, the two-level blocking scheme facilitates vector reuse across successive layers and the fusion of the SymGS backward sweep with the subsequent SpMV operation. Due to the above mentioned reasons, two-level blocking scheme is suitable for the architecture of the Sunway supercomputer in terms of strong parallelism and fast convergence.

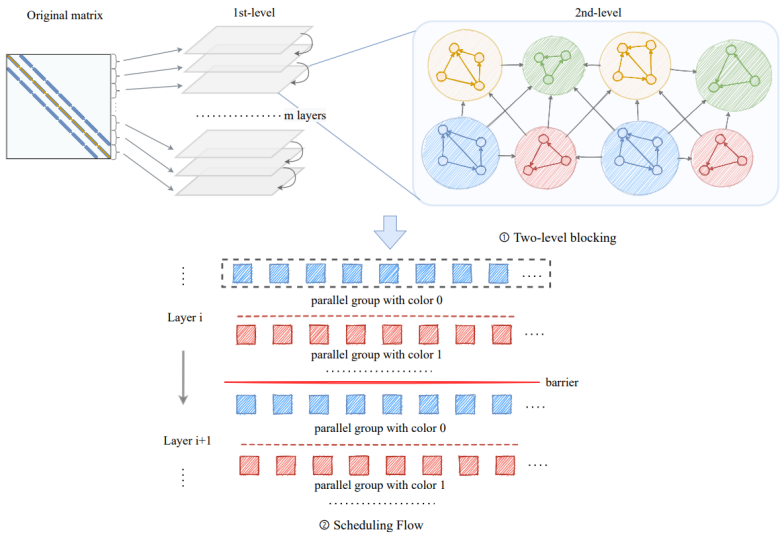


Figure 4.8 A two-level blocking scheme adapted from [41]

4.2.6 Block Multi-Color Scheduling (BMC) Scheduling

Block Multi Color Scheduling (BMC) Scheduling as shown in Figure 4.9 is an efficient parallelization technique designed to optimize algorithms such as the SymGS. The grid points are colored by traditional multi-coloring approach in such a manner that enables the processing of the same colored data points in parallel. Nevertheless, these methods may result in adverse consequences, including a decrease

in convergence rate and a loss of data locality. The original approach is expanded by the necessity of grouping grid points into neighboring blocks and coloring these blocks. This enhances data localities and contributes to the acceleration of parallel execution rates. The BMC Scheduling is a more advance approach used by [29, 36] and the overview of this approach is as follows:

- Block Partitioning: The grid is partitioned into blocks, where the neighboring grid points are grouped into a block. This enhances data locality in comparison to conventional point-based coloring techniques.
- Coloring Block: Blocks are assigned colors such that blocks of the same color have no dependencies and can be processed in parallel. This usually involves by using multiple colors across different dimensions of the grid.
- Asynchronous Execution: Instead of synchronizing threads after processing of each color, blocks are processed asynchronously. Threads process new blocks asynchronously as soon as their dependencies resolved, and this approach increased core utilization and reduced idle time.
- Dependency Management: A mechanism keeps track of dependencies between blocks of different colors. As a block complete its processing, it updates the status of other dependent blocks on it.
- Synchronization Sparsification: Synchronization barriers are minimized and overall performance is improved by employing different techniques to reduce unnecessary dependencies between blocks.
- Dynamic Scheduling: A mechanism that ensures the availability of a block for processing when all of its dependencies are met, and enable load balancing by threads to select newly accessible blocks without the need for global

synchronization.

- Adaptive Block Sizing: Selection of block size determined based on the available threads and on the grid dimensions. This helps in balance the load and parallelism maintaining the convergence rate
- Bi-directional Sweeps: The algorithm consists of forward and backward sweeps, follow the dependency hierarchy among the blocks, established by the coloring scheme.

This approach improves parallelism, optimizes memory access patterns, and scales performance for multi-core setup, mostly effective in unstructured matrices.

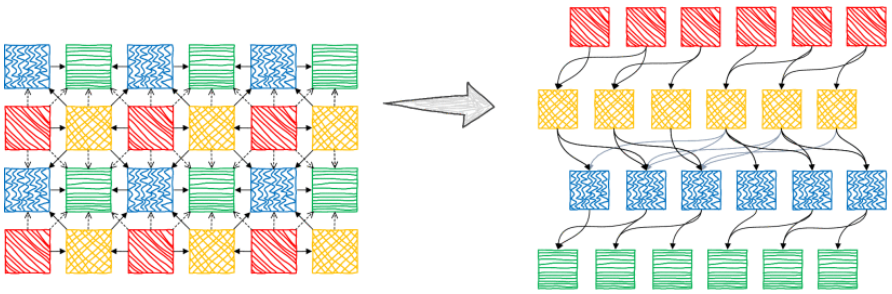


Figure 4.9 Block multicoloring with synchronization sparsification involves (a) the block multicoloring process and the dependencies between the blocks, where a dashed arrow represents a redundant dependency. (b) After synchronization sparsification, the scheduling of tasks is adjusted to avoid unnecessary synchronization, reproduced from [29]

CHAPTER 5. Parallel Implementation of Symmetric Gauss–Seidel (SymGS) Variants

5.1 Reference SymGS and its parallel variants

Algorithm 2 is a standard symmetric Gauss-Seidel method, is referred to as the Reference SymGS as implemented in the HPCG, which is inherently sequential. 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 2 Symmetric Gauss-Seidel - Reference SymGS

Require: Matrix $A \in \mathbb{R}^{n \times n}$, right-hand side vector $r \in \mathbb{R}^n$, initial guess $x \in \mathbb{R}^n$
Ensure: Updated solution vector x
Forward Sweep:
for $i = 1, 2, \ldots, n$ **do**
  for $j \in$ nonzeros in i^{th} row **do**
    $x_i = \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
  end for
end for
Backward Sweep:
for $i = n, n-1, \ldots, 1$ **do**
  for $j \in$ nonzeros in i^{th} row **do**
    $x_i = \frac{r_i - \sum_j A_{ij} x_j + A_{ii} x_i}{A_{ii}}$
  end for
end for

Previous research on stencil computations in multi-core systems by [102–105]

resulted in improved ways to access memory and schedule tasks for structured grid-based solvers like SymGS. One of the most popular ways to improve SymGS performance is the multi-coloring method [24, 30, 106–108], which uses colors to group independent tasks for parallel processing, and also level scheduling [50, 69, 89] by restructuring of computations according to dependency levels to enhance parallelism.

In Multi-Color SymGS routine we 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, the number of colors needed depends on the structure of the matrix.

In Level-Scheduled SymGS we 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 of a node on other nodes, that node will belong to the initial level L_0, 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 forward and backward sweeps subsequently, to modify the solution vector x.

Hybrid Jacobi-Gauss-Seidel methods [109–112] combine parallel friendly Jacobi with improved GS convergence, suggesting potential for scalable SymGS approach. For Hybrid Jacobi-Gauss-Seidel variant we integrates the parallelism of the Jacobi method with the accelerated convergence of GS by partitioning the rows of a sparse

matrix A into two groups according to the number of non-diagonal nonzeros. Rows with more connections are designated to the Jacobi set and updated concurrently. In contrast, the other rows constitute the GS set and are updated sequentially. The algorithm initially executes a parallel Jacobi forward sweep, subsequently performs sequential GS forward and backward sweeps, and concludes with a final parallel Jacobi update. The Jacobi ratio (jr) governs the division between the two methods.

5.2 Our Designed Variants

5.2.1 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 GS 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 of the sweeps to ensure that in each block, the solution vector x is updated before moving to the next block. We then use the final solution vector for a residual check. This method improves the convergence over reference SymGS methods.

5.2.2 Over Relaxation SymGS

Algorithm 4 Over-Relaxation SymGS concept inherited from the JOR approach which uses Jacobi updates by over-relaxation to improve the convergence rate of the

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

Let n be the number of rows in A
Compute the number of spatial blocks: $m = \lceil n/b \rceil$
Initialize block starts: block_starts$[i] = i \cdot b, \quad \forall i \in \{0, 1, \ldots, m-1\}$
Initialize block sizes: block_sizes$[i] = \min(b, n - \text{block_starts}[i])$
Determine dependencies for each block based on non-zero elements in A
Let l be the index representing non-zero elements in each row of A
for each block k from 0 to $m - 1$ **do** ▷ Forward Sweep
    for each temporal step t from 1 to s **do**
        for each dependent block j in depBlocks$[k]$ **do**
            Update rows in block j parallely:

$$x_j^{\text{new}} = \frac{r_j - \sum_l A_{jl} x_l^{\text{old}} + A_{jj} x_j^{\text{old}}}{A_{jj}}$$

        end for
        Update rows in block k parallely:

$$x_i^{\text{new}} = \frac{r_i - \sum_j A_{ij} x_j^{\text{old}} + A_{ii} x_i^{\text{old}}}{A_{ii}}, \quad \forall i \in \text{block}[k]$$

        Copy x_i^{new} to x_i^{old} for the next iteration
    end for
end for
for each block k from $m - 1$ to 0 **do** ▷ Backward Sweep
    for each temporal step t from 1 to s **do**
        for each dependent block j in depBlocks$[k]$ **do**
            Update rows in block j in reverse order parallely:

$$x_j^{\text{new}} = \frac{r_j - \sum_l A_{jl} x_l^{\text{old}} + A_{jj} x_j^{\text{old}}}{A_{jj}}$$

        end for
        Update rows in block k in reverse order parallely:

$$x_i^{\text{new}} = \frac{r_i - \sum_j A_{ij} x_j^{\text{old}} + A_{ii} x_i^{\text{old}}}{A_{ii}}, \quad \forall i \in \text{block}[k]$$

        Copy x_i^{new} to x_i^{old} for the next iteration
    end for
end for
Copy x^{new} to x for residual check
return Updated vector x

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

Algorithm 4 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**
    for $j \in$ nonzeros in i^{th} row **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
end for
$x \leftarrow x^{\text{temp}}$
Backward Sweep:
for $i = n$ to 1 in parallel **do**
    for $j \in$ nonzeros in i^{th} row **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
end for
$x \leftarrow x^{\text{temp}}$
return x

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 $x_i^{\text{temp}} = x_i + \omega(x_i^{\text{temp}} - x_i)$, which seeks to expedite error reduction by utilizing both temporary and current values.

A variety of relaxation-based methods had been explored in the past [113–115] for GS:

- Successive Over-Relaxation (SOR) [116] is a point-wise relaxation technique that employs a single forward sweep with a relaxation factor ω. Despite its sequential nature and absence of symmetry limiting parallel execution.

- Symmetric SOR (SSOR) extends SOR with a backward sweep, creating symmetry suitable for PCG preconditioning of SPD systems. It remains inherently sequential despite its mathematical advantages.
- Jacobi Over-Relaxation (JOR) employs a temporary vector approach in the Jacobi method. Eliminating coupling achieves total parallelism; however, it results in significantly slower convergence rates.
- Jacobi-SOR (JSOR) employs domain-partitioned sub-grids to integrate one-line SOR updates within a Jacobi method. This provides moderate parallelism but experiences convergence degradation proportional to partition count.
- Multi-Color SOR employs graph coloring to facilitate concurrent updates of same colored points. Although additional colors compromise spectral efficiency for increased parallelism, two-color schemes predominantly preserve SOR convergence characteristics.
- Parallel SOR (PSOR) [113] reorganizes point-wise SOR through domain partitioning and strategic communication. Exhibits convergence rates comparable to sequential SOR.
- Block Parallel SOR (BPSOR) [114] minimizes communication overhead by performing block updates that converge at a rate comparable to sequential block SOR, thereby extending PSOR to block partitions through multi-type ordering.
- Parallel Symmetric SOR (PSSOR) symmetrizes BPSOR [115] under multi-type ordering to yield an SPD preconditioner similar to SSOR. Even for highly anisotropic problems with significant scalability, it preserves nearly sequential SSOR convergence.

- Our Overrelaxation-SymGS variant employs the JOR like buffer technique within the forward-backward SymGS framework. This method is mathematically valid as a PCG smoother, exhibiting symmetry and compatibility with parallel processing.

5.2.3 Wavefront SymGS

Algorithm 5 Wavefront SymGS utilizes the wavefronts to organize and schedule the updates in an iteration. The wavefront technique uses each coordinate point (x, y, z) in a three-dimensional grid $nx \times ny \times nz$ has a 'wave index' derived from the sum $x + y + z$.

Algorithm 5 Wavefront SymGS Method

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
Initialize $x_{\text{old}} \leftarrow x$, $x_{\text{new}} \leftarrow x$
Define wavefronts W_k for $k \in [0, nx + ny + nz - 2]$
Forward Wavefront Update:
for $k = 0$ to num_waves $- 1$ **do**
    for each row $i \in W_k$ (in parallel) **do**
        for $j \in$ nonzeros in i^{th} row **do**
            $s \leftarrow r_i - \sum_j A_{ij} x_{\text{old},j}$
            $s \leftarrow s + A_{ii} x_{\text{old},i}$
            $x_{\text{new},i} \leftarrow \frac{s}{A_{ii}}$
        end for
    end for
    $x_{\text{old}} \leftarrow x_{\text{new}}$
end for
Backward Wavefront Update:
for $k =$ num_waves $- 1$ to 0 **do**
    for each row $i \in W_k$ (in parallel) **do**
        for $j \in$ nonzeros in i^{th} row **do**
            $s \leftarrow r_i - \sum_j A_{ij} x_{\text{old},j}$
            $s \leftarrow s + A_{ii} x_{\text{old},i}$
            $x_{\text{new},i} \leftarrow \frac{s}{A_{ii}}$
        end for
    end for
    $x_{\text{old}} \leftarrow x_{\text{new}}$
end for
Copy Updated Solution:
for $i = 1$ to n_{row} (in parallel) **do**
    $x_i \leftarrow x_{\text{new},i}$
    $x_{\text{old},i} \leftarrow x_{\text{new},i}$
end for
Return: Updated vector x

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 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. We loop over every valid point (x, y, z), compute the row index and place this index into a local buffer corresponding to that wave. 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, using double-buffering to ensure numerical accuracy. For each wave W_k, rows are updated in parallel by computing $s = r_i - \sum_j A_{ij} x_{\text{old},j}$, adding back the diagonal part $A_{ii} x_{\text{old},i}$, and dividing by A_{ii} to get the new $x_{\text{new},i}$ To refresh subsequent waves, updated values are copied into x_{old} 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.

5.3 Experiments and Results

5.3.1 Methodology

The Figure 5.1 shows the high-level execution flow of HPCG benchmark and its relationship with SymGS. It begins with a standard CG function that initializes necessary parameters and performs the operations like computing SpMV, WAXPBY, and dot products. An iterative loop runs for a maximum number of iterations or until convergence is reached. Preconditioning triggers the multigrid preconditioner function (ComputeMG), which at different grid levels uses SymGS as pre- and post-smoother. The right side shows the standalone SymGS variants implementation,

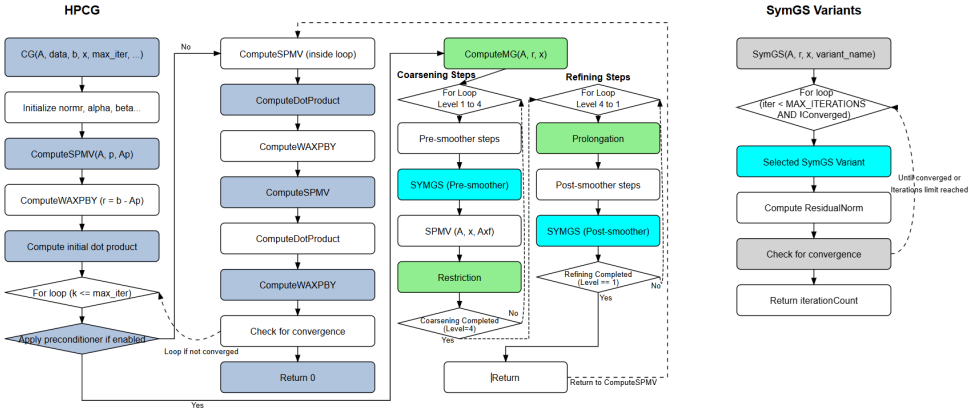


Figure 5.1 Diagram of execution flows between our SymGS variants implementation and the HPCG iterative solver, emphasizing distinctions in preconditioning and iterative methodologies.

which follows its own iterative process with convergence checks and residual norm calculations. The key difference between the standalone SymGS and HPCG is that, while it uses a linear iterative framework, and HPCG employs a recursive multigrid method with SymGS serving as a critical smoother component to enhance convergence at different grid levels. We evaluated different variants based on various parameters and subsequently selected the most effective one for use in HPCG. In HPCG, we substituted the pre- and post- smoother SymGS with our implemented SymGS (the step highlighted in cyan color) and assessed the performance of HPCG.

5.3.2 Settings

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.

A MPI+OpenMP parallelization strategy is utilized in all experiments. The total number of OpenMP threads allocated per MPI process is established according to the following relation:

$$\text{OpenMP Threads per MPI Process} = \frac{\text{Total Number of Cores}}{\text{Number of MPI Processes}}.$$

KNL featuring 68 cores and SKL with 40 cores. For instance, when only 2 MPI processes are specified, each each process is assigned 34 and 20 OpenMP threads on KNL and SKL, respectively. This mapping strategy is consistently applied across all experiments unless otherwise specified.

5.3.3 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:

$$\text{Total Time} = \text{Setup} + \text{Compute}.$$

- **Iterations:** Total number of iterations performed until convergence or meeting the stopping criteria.

- **Avg/Iter (s):** Average time per iteration, computed as:

$$\text{Avg/Iter (s)} = \frac{\text{Compute Time}}{\text{Iterations}}.$$

- **Error Metrics:** Relative Error (Rel. Error), Root Mean Square Error (RMSE), and Mean Absolute Error (MAE) are used 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 of reference SymGS compared to the improved method:

$$\text{Speedup} = \frac{\text{Total Time}_{\text{(reference)}}}{\text{Total Time}_{\text{(method)}}}.$$

- **Improvement:** Increment in Gflops compared to the baseline reference SymGS.

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

- **MultiColor(MC)-c:**

$$c \in \{2, 4, 6, 8\},$$

  where c is the number of colors used.

- **TemporalBlocking(TB)-a-b:**

$$a, b \in \{2, 4, 8, \ldots, 64\},$$

where 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.

- **Hybrid_Jacobi_GS:**

$$jr \in \{0.5, 0.6, \ldots, 1.0\},$$

Jacobi ratios (jr) in increments of 0.1, ranging from 0.5 to 1.0.

- **OverRelaxation-ω:**

$$\omega \in \{0.2, 0.4, \ldots, 1.4\},$$

where overrelaxation parameter omega (ω) was adjusted from 0.2 to 1.4 in increments of 0.2

5.3.4 Results on Knights Landing (KNL)

5.3.4.1 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 10^{-8}. For larger problem sizes ($> 64^3$), the tolerance was adjusted to 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 5.1, 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 some problem sizes, the computation time is reduced with 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 costs less than that of 8 colors; however, the total time, inclusive of setup time, increases when the problem size gets bigger. 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. 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. Temporal Block variants significantly reduce the compute time, with the total time decreasing as the number of temporal steps increases. For 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 2.6× 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.

Table 5.1 Performance comparison of SymGS variants for different problem sizes on KNL.

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	-	-	-	0.1206	1.0000	1.0000
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-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-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**
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	-	-	-	0.0583	0.4838	0.4838
Problem Size: 32^3, Total number of rows: 32,768											
Reference	0.0000	13.8172	13.8172	0.0284	487	-	-	-	0.1183	1.0000	1.0000
MultiColor-4	1.7956	6.5644	8.3600	0.0098	672	1.80E-07	3.95E-09	3.00E-09	0.2697	1.6528	2.2806
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	1,926	5.47E-08	1.20E-09	9.12E-10	0.2849	0.6091	**2.4089**
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**
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	-	-	-	0.1670	1.4120	1.4120
Problem Size: 64^3, Total number of rows: 262,144											
Reference	0.0000	440.4997	440.4997	0.2352	1,873	-	-	-	0.1178	1.0000	1.0000
MultiColor-4	114.4299	189.5306	303.9606	0.0729	2,599	6.23E-07	1.85E-08	1.38E-08	0.2368	1.4492	2.0109
MultiColor-8	0.1034	201.6544	201.7578	0.0695	2,903	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**
TemporalBlocking-2-16	0.1967	20.4646	20.6613	0.1749	117	1.79E-07	5.32E-09	3.97E-09	0.1568	21.3200	1.3318
TemporalBlocking-2-64	0.1974	16.7922	16.9896	0.5597	30	2.09E-06	6.21E-08	4.63E-08	0.0489	25.9276	0.4153
Hybrid_Jacobi_GS-0.9	0.1195	633.7727	633.8922	0.0848	7,470	2.53E-08	7.86E-10	5.71E-10	0.3264	0.6949	**2.7715**
OverRelaxation-1.0	0.0000	139.3856	139.3856	0.0682	2,045	2.20E-07	6.54E-09	4.74E-09	0.3961	3.1603	**3.3637**
Wavefront	0.0134	205.6821	205.6956	0.0893	2,303	3.20E-09	6.47E-11	4.88E-11	0.3101	2.1415	**2.6332**
LevelScheduled	0.2711	172.6534	172.9245	0.0922	1,873	-	-	-	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	3,106	-	-	-	0.1140	1.0000	1.0000
MultiColor-4	1339.0005	1079.5646	2418.5651	0.2479	4,355	1.15E-04	4.12E-06	3.05E-06	0.1701	1.0645	1.4926
MultiColor-8	0.3492	1168.2227	1168.5719	0.2415	4,837	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	1,551	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
TemporalBlocking-2-64	0.6118	83.7539	84.3657	1.7093	49	1.33E-04	4.76E-06	3.53E-06	0.0549	30.5167	0.4814
Hybrid_Jacobi_GS-0.9	0.4621	3605.4660	3605.9281	0.2908	12,397	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	6,204	2.60E-06	9.19E-08	6.83E-08	0.3033	1.3363	**2.6691**
Wavefront	0.1024	4241.8401	4241.9425	1.1104	3,820	3.72E-06	1.32E-07	9.82E-08	0.0851	0.6069	0.7464
LevelScheduled	0.9873	2220.6792	2221.6666	0.7150	3,106	-	-	-	0.1321	1.1588	1.1588

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

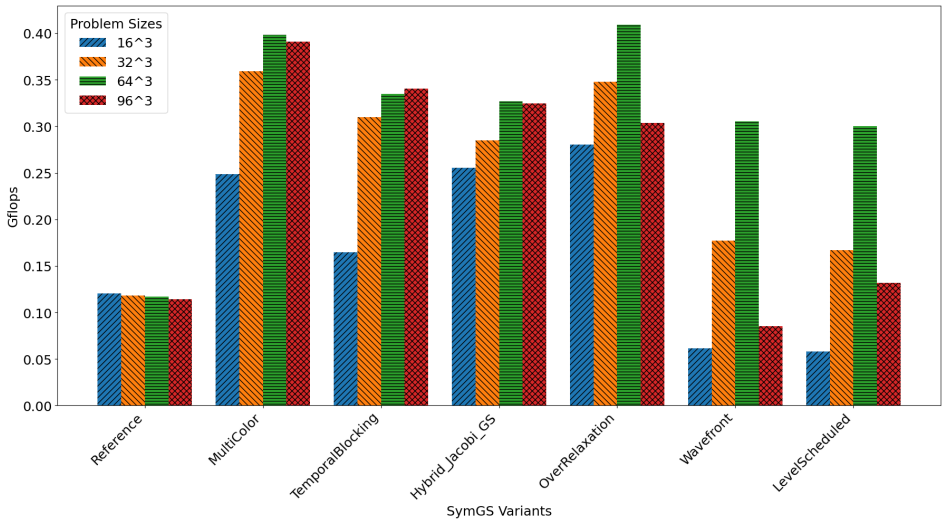


Figure 5.2 Performance comparison of SymGS variants in terms of Gflops across problem sizes (16^3, 32^3, 64^3, and 96^3) on 1 MPI process with OpenMP threads=68.

surpasses the reference SymGS. The Hybrid Jacobi GS SymGS consistently performs better with $jr = 0.9$ 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. Overall MultiColor, Temporal Blocking, and OverRelaxation SymGS variants demonstrate superior performance in Gflops as compared to other variants, as shown in Figure 5.2.

5.3.4.2 HPCG

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 $jr = 0.9$.
  - OverRelaxation with $\omega = 1.0$.

  Performance results for these variants are shown in Figure 5.3.

Figure 5.3 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

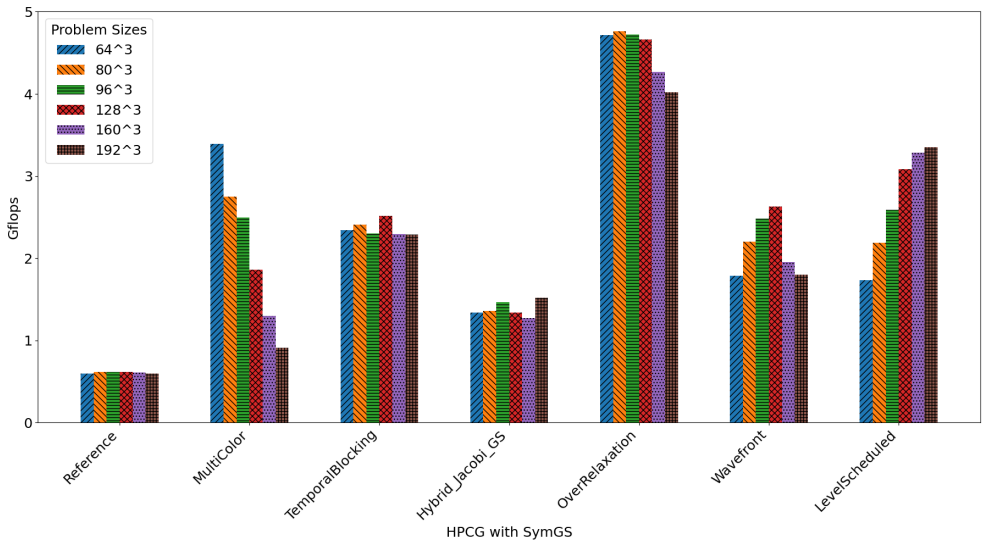


Figure 5.3 Performance of HPCG using the different SymGS variants across different problem sizes (64^3 to 192^3) on 1 MPI process with OpenMP threads=68.

the native HPCG without any optimization. We examined the different variants of SymGS in HPCG and assessed their performance compared to the reference using 1 MPI process and 68 threads on single node and observed the Overrelaxation SymGS is relatively the most efficient in comparison to other variants.

The HPCG with the Overrelaxation SymGS variant is regarded as the most effective, achieving an average of $7\times$ 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 $2.92\times$, while for 192^3, it attains $5.6\times$, 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 $3.5\times$, While Wavefront SymGS also exhibit the moderate performance improvement and relatively performing better on the mid-sized problems with average performance improvement of $3.8\times$ with some drop in performance on small and large problem sizes.

MultiColor begins with a substantial improvement of $5.7\times$ for 64^3, but its efficacy diminishes progressively as the problem size increases, yielding an improvement of merely $1.52\times$ for 192^3. The Hybrid_Jacobi_GS are comparable, exhibiting a modest average improvement of $2.2\times$, accompanied by minimal oscillations that suggest performance stability across the different problem sizes. The OverRelaxation performs the best consistently, and the Level Scheduled performs well, especially with larger problem sizes. As has been seen before, all of the SymGS variants including Temporal Blocking and Wavefront provide a stable performance.

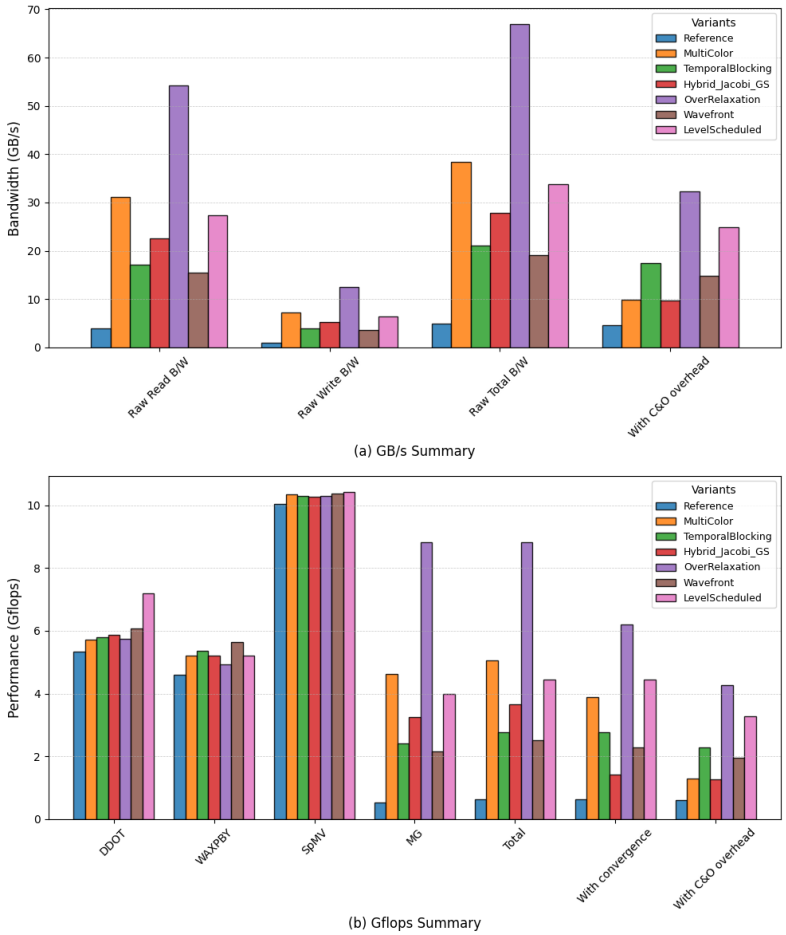


Figure 5.4 Performance and Bandwidth Summary of HPCG at a problem size of 160^3 using different SymGS variants

We observed that the native implementation of HPCG performs better with the MPI-only configuration, each MPI process only has one thread, so our optimized routine has no room for parallelism. As a result, the routine operates almost sequentially. Additionally, the use of an extra temporary vector within each thread further impacts performance, making it slightly inferior to the reference implementation, when tested 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.

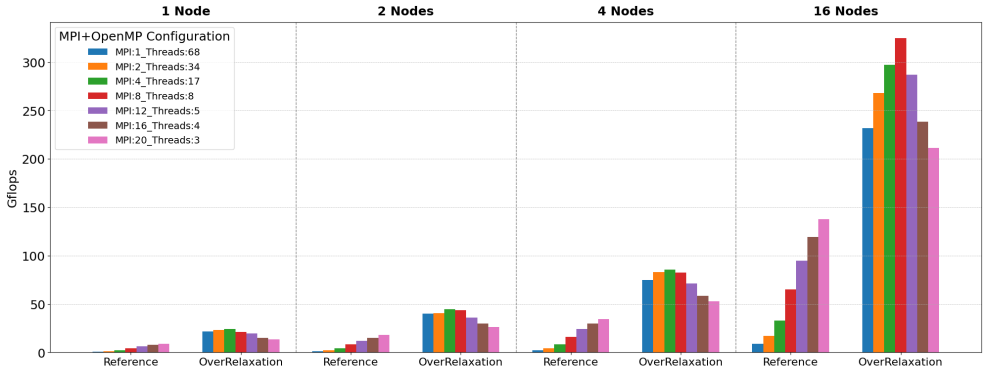
Figure 5.4 (a) illustrates the bandwidth summary extracted from the HPCG summary reports. OverRelaxation SymGS enhanced bandwidth utilization. The total bandwidth with convergence and optimization (C and O) is improved as compared to the reference implementation, thereby improving overall performance.

Figure 5.4 (b) illustrates 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. It indicates that the performance improvement of MG within HPCG is a result of the performance improvement of SymGS.

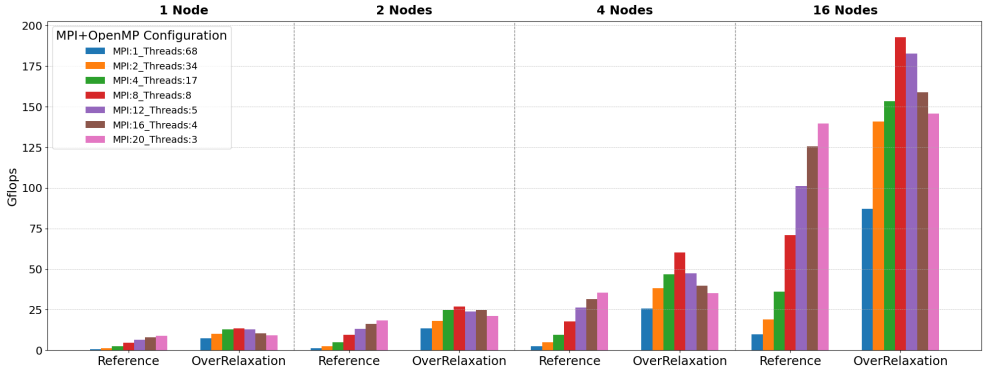
5.3.4.3 Scalability Analysis

We tested the MPI+OpenMP version of HPCG on a problem size of 160^3, ensuring that it used enough memory as required by the HPCG specification, which requires that at least quarter 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, the memory usage rises significantly, and with MPI processes exceeding 20, the job fails due to memory limitations.

The performance trends as shown in Figure 5.5 indicate that the HPCG with reference SymGS scales well with an increasing number of MPI processes. However,



(a) MG (Multigrid)



(b) HPCG

Figure 5.5 Performance comparison of (a) MG (Multigrid) and (b) HPCG on KNL for Reference and OverRelaxation using different MPI+OpenMP configurations on multiple nodes (1, 2, 4, and 16) for a problem size of 160^3.

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 parallel execution. thus the overall performance drops. We have evaluated and recorded the performance of MG, extracted from the HPCG summary reports also presented, as shown in Figure 5.5 (a). However, there exists a notable difference in the performance of HPCG and MG, as illustrated in Figure 5.5, which indicates that the MG performance is relatively good compared to HPCG. This highlights the need to improve the other kernels and reduce

communication and optimization overhead to further improve the overall performance of HPCG.

The best performance is achieved with 8 MPI processes per node and 8 threads. The performance of HPCG using OverRelaxation SymGS reaches 192.82 Gflops on 16 Nodes. However, on 20 MPI processes per node with only three threads, performance degradation exhibits at higher node counts, demonstrating that balancing MPI processes and OpenMP threads is crucial for optimal parallel efficiency.

5.3.5 Results on 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.

5.3.5.1 SymGS variants and HPCG

The performance of SymGS variants on SKL resembles the trends observed on KNL, OverRelaxation and MultiColor SymGS remain the top two performer, respectively. Table 5.2 presents a comparative analysis of SymGS variants for a problem size of 32^3 (32,768 rows) and Figure 5.6 (on left) illustrates this comparison in terms of Gflops.

OverRelaxation SymGS with $\omega = 1.4$ outperformed the other variants, reducing the execution time to 0.9627 seconds and achieving a 1.88$\times$ acceleration compared to the reference SymGS. This improves performance by 2.65$\times$, 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 1.66$\times$ speedup over the reference, and a 2.55$\times$

Table 5.2 Performance comparison of 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	-	-	-	0.9005	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	-	-	-	0.4076	0.4527	0.4527
Hybrid Jacobi-GS 0.5	0.0030	4.2873	4.2904	0.0026	1,629	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	1,926	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**

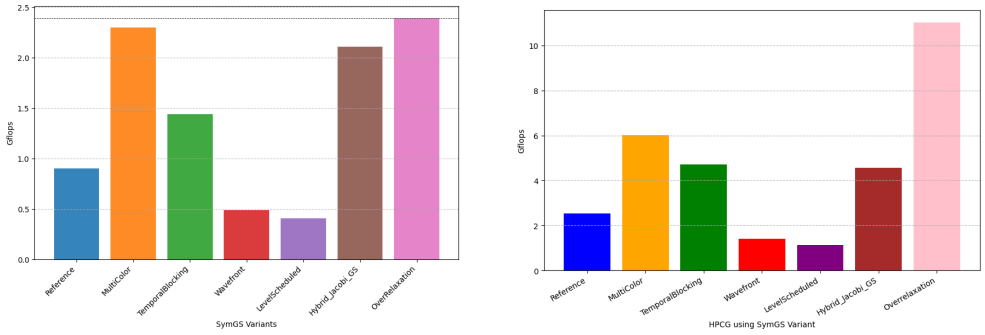


Figure 5.6 Performance comparison of SymGS variants (left) and their impact in HPCG (right) on the problem size of 32^3 on 1 MPI porcess with OpenMP threads=40.

increase in Gflops.

In Temporal Blocking SymGS variants, specifically TB-2-2, outperform its other combinations. Hybrid Jacobi-GS with $jr = 0.9$ results in a 2.34× increase in Gflops, indicating that combining Jacobi parallelism and GS updates improves performance. The Wavefront and Level Scheduled SymGS variants do not perfrom good on Skylake for this problem size. In fact, they both perform worse than the reference implementation.

In the HPCG benchmark, the OverRelaxation SymGS variant was used with $\omega = 1.0$, 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 $(a, b) = (2, 2)$, Hybrid Jacobi-GS with $jr = 0.9$, 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.

Figure 5.6 (on right) 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 good performance as in KNL. MultiColor SymGS and Temporal Blocking SymGS also show notable improvement compared to the reference implementation, although these are lag behind OverRelaxation.

5.3.5.2 Scalability Analysis

Since the problem size of 32^3 is insufficient to challenge the memory subsystem or satisfy the memory requirements specified by HPCG, the larger problem size of 288^3 is assessed in multi-node configurations. We have evaluated the HPCG using the Reference and OverRelaxation SymGS. We have optimized only the SymGS component of the MG in HPCG. We recorded the performance results of HPCG using the Reference and OverRelaxation SymGS variants, and additionally showcased the performance of MG, extracted from the HPCG summary reports as shown in Figure 5.7. The MG performance using the OverRelaxation SymGS

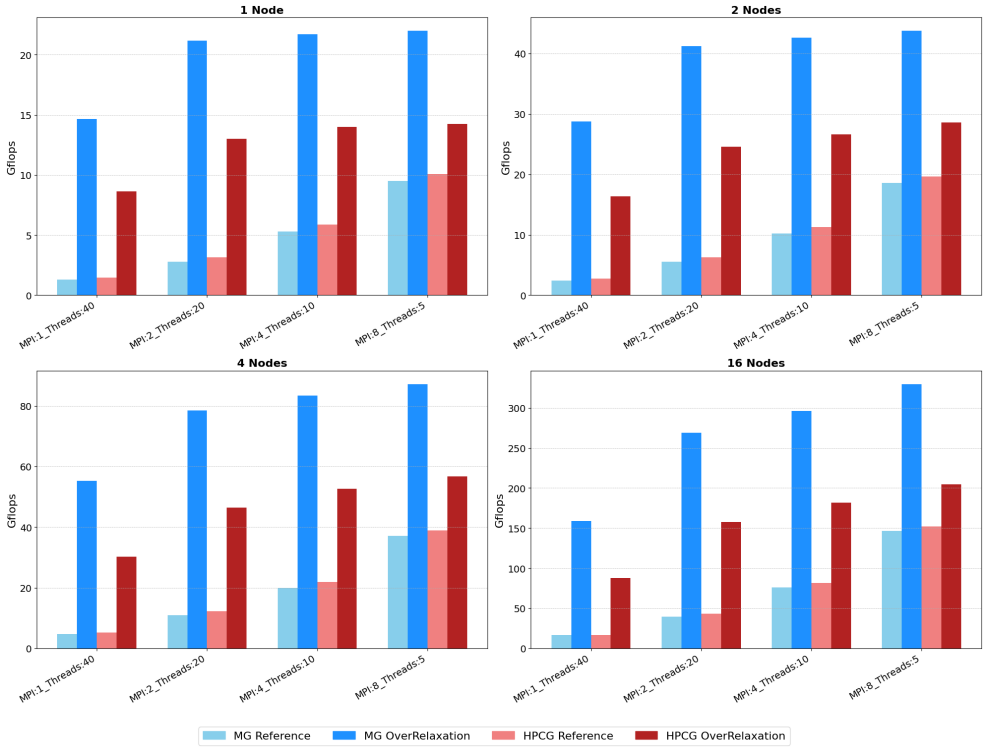


Figure 5.7 Performance of MG and HPCG for a problem size of 288^3 using different MPI+OpenMP configurations across multiple SKL nodes.

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 the Reference. On 16 nodes, OverRelaxation achieves 159.1 Gflops at one MPI process, indicating an approximate 10× 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× surpassing the 1.48 Gflops recorded by the Reference SymGS. As the number of node increases, OverRelaxation maintains a significant improvement, especially in settings with a fewer 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× improvement. On increasing MPI counts, specifically 8 MPI processes across 16 nodes, OverRelaxation achieves 204.86 Gflops, in contrast to 151.84 Gflops for the Reference, illustrating sustained performance improvements despite rising communication overhead. The results demonstrate that OverRelaxation SymGS offers significant local 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 MG. However, there exists a notable difference in the performance of HPCG and MG, as illustrated in Figure 5.7, which indicates that the MG 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.

5.4 Observations and Discussion

Many researchers have explored different methods to enhance HPCG performance on different architectures, with prominent techniques including the conversion of data formats from CSR to ELLPACK and the application of multi–coloring strategies, such as Blocked Multi–Coloring.

Multi-coloring and level scheduling-based approaches have previously been explored by other researchers, particularly for 27-point stencil problems. However, our study introduces OpenMP-based parallel SymGS techniques, including Temporal Blocking, Wavefront, Hybrid Jacobi-GS, and Overrelaxation variants, which significantly differ from existing implementations in the literature. Notably, prior research employing similar techniques predominantly addressed stencil problems with fewer points, whereas our study specifically targets the more complex 27-point stencil configuration used in the HPCG benchmark. Among the introduced techniques, Temporal Blocking is particularly advantageous for scientific problems that require SymGS methods to converge within fewer iterations. A 4-color approach on some problem sizes gives relatively better computational performance than the 8-color approach and its convergence rate is also better than using the 8-color, but its setup time is higher. Further, within the HPCG, it properly works on the coarse grids; however, on the fine grid with 4 colors the overall HPCG result becomes invalid due to a failure in the symmetry test. To solve this, additional dependency handling is required, which diminishes the overall performance gain, so we have used 8 colors in our HPCG results comparisons.

After investigating several SymGS variants, our findings indicate that the Overrelaxation approach using a relaxation factor of $\omega = 1$ mirrors the reference implementation in both the forward and backward sweeps by employing a temporary

buffer vector for partial updates. This design enables parallel execution and proves to be simpler and more efficient than multi–coloring based dependency resolution. While previous research has emphasized that multi-coloring techniques are optimal for enhancing SymGS performance, our study demonstrates that the Overrelaxation method produces good results.

5.4.1 Parallelism

When we run P MPI ranks per node while keeping the problem size per node constant at N^3, each MPI rank handles a smaller subproblem of size $(N')^3$ with

$$N' = \frac{N}{\sqrt[3]{P}},$$

and the available parallelism per node for the SymGS smoother is reduced accordingly. In this setting, the overall number of tasks that can be executed in parallel (each row's work is roughly proportional to the number of nonzeros in that row) is given by

$$\text{Parallelism} = P \left(\frac{N}{\sqrt[3]{P}} \right)^2 \frac{1}{7} = \sqrt[3]{P} \frac{N^2}{7}.$$

In a 3D 27-point stencil, used in HPCG, each grid point depends on itself and its six immediate neighbors, one in each direction (left, right, front, back, up, and down), resulting in a total of seven dependencies per point or matrix row. This expression shows that, with P MPI ranks per node, the maximum parallel performance (in terms of the number of concurrent tasks or operations that can be exposed) scales as $\sqrt[3]{P}$ [24].

This mathematical form represents the possible room for performance optimization in multi-MPI configuration. On 1 MPI process we observed the performance

improvement of around $7 - 10\times$, whereas the overall performance on multi-MPI processes is only $2\times$ for MG and overall $1.4\times$ of HPCG. This indicate that there is still significant room for further optimization in SymGS, the overall performance improvement also required the optimization of other computational kernels within HPCG, particularly the SpMV.

This study presented and assessed different parallel variants of SymGS, enhancing their performance for SPD matrices in the HPCG benchmark. The implementation of these variants demonstrated enhanced parallel performance without compromising on numerical stability, resulting in an 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 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 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.

Our investigation of various SymGS variants reveals that the overrelaxation method with $\omega = 1$ replicates the reference implementation in both forward and backward sweeps by utilizing just a temporary buffer vector for partial updates. This design facilitates parallel execution and demonstrates simplicity and efficiency compared to

multi-color based dependency resolution approach.

Based on this study, the promising research directions for HPC researchers are to apply hybrid techniques such as multi-coloring with temporal blocking to enhance the convergence of multi-coloring approach.

Future work includes Kokkos-based implementations. We also plan to adopt ELLPACK data layout and optimize other HPCG kernels such as SpMV for improved overall performance.

CHAPTER 6. KoHPCG – High-Performance Conjugate Gradient Benchmark Program on Kokkos Performance Portability Framework

In KoHPCG implementation, we rewrite all computational kernels using Kokkos constructs, preserving the original algorithmic structure of HPCG.

- Used Kokkos::View for managing multi-dimensional arrays across execution spaces.
- Used Kokkos::parallel_for and Kokkos:: parallel_reduce to enable thread-level and data-level parallelism.
- Used Kokkos::fence and memory traits to ensure consistency across host and device execution.
- For now, we used default execution and memory space using Kokkos:: DefaultExecutionSpace and ExecSpace::memory_space.

6.1 Kokkos-Based Implementation

DDOT

```
Kokkos::parallel_reduce("DDOT", n,
  KOKKOS_LAMBDA(const local_int_t i, double andupdate) {
    update += x.values(i) * y.values(i);
}, local_result);
```

Kokkos kernels also offer this operation as kokkosBlas::dot(...). In our current results, we utilized KokkosBlas operation because Kokkos kernels internally employ the architecture-tuned implementation for enhanced portability.

WAXPBY

```
Kokkos::parallel_for("WAXPBY", Kokkos::RangePolicy<>(0, n),
  KOKKOS_LAMBDA(const int i) {
    w(i) = alpha * x(i) + beta * y(i);
});
```

SpMV

```
Kokkos::parallel_for("SpMV", Kokkos::RangePolicy<>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t i) {
    double sum = 0.0;
    int nnz = (int)(nonzerosInRow_d(i));
    for (int j = 0; j < nnz; j++) {
      sum += matrixValues_d(i, j) * xv(mtxIndL_d(i, j));
    }
    yv(i) = sum;
});
```

Kokkos kernels also provide this operation as KokkosSparse::spmv(...). In our current implementation, we utilized the Compressed Sparse Row (CSR) based SpMV offered by KokkosSparse, as it provides us the relatively good HPCG performance when combined with our optimized SymGS kernel. We noted that the Block Sparse Row (BSR) format exhibited superior SpMV performance compared to CSR when evaluated only on SpMV.

SymGS

```
Kokkos::View<double*, Layout, MemorySpace> x_temp("x_temp", x.localLength);
deep_copy(x_temp, xv);

Kokkos::parallel_for("Forward_Sweep",
  Kokkos::RangePolicy<ExecSpace>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t i) {
    double sum = rv(i);
    int cur_nnz = nnzInRow(i);
    double diag_val = diag(i);
    double x_i = xv(i);
    for (int j = 0; j < cur_nnz; j++) {
      const local_int_t col = indices(i, j);
      sum -= values(i, j) * xv(col);
    }
    sum += x_i * diag_val;
    x_temp(i) = sum / diag_val;
});
Kokkos::fence("Forward␣Sweep");

Kokkos::parallel_for("Backward_Sweep",
```

```
  Kokkos::RangePolicy<ExecSpace>(0, nrow),
  KOKKOS_LAMBDA(const local_int_t idx) {
    const local_int_t i = nrow - 1 - idx;
    double sum = rv(i);
    int cur_nnz = nnzInRow(i);
    double diag_val = diag(i);
    for (int j = 0; j < cur_nnz; j++) {
      const local_int_t col = indices(i, j);
      sum -= values(i, j) * x_temp(col);
    }
    sum += x_temp(i) * diag_val;
    xv(i) = sum / diag_val;
});
Kokkos::fence("Backward Sweep");
```

Given the inherently sequential nature of the SymGS operation, we optimized and parallelized it by employing a temporary vector to eliminate the dependency. This straightforward modification requires reading and writing to different memory locations to avoid conflicts arising from partial updates, thereby facilitating the parallelization of the SymGS operation. This approach permits concurrent forward and backward execution, data dependencies are not violated, and improves the performance of the SymGS operation, which overall contributes in the performance improvement of HPCG.

MG

Multigrid (MG) is composed of SymGS, SpMV, Prolongation, and Restriction operations. In our implementation, we modified SpMV and SymGS to the Kokkos

programming model, so we only update the Prolongation and Restriction operations with required modifications using Kokkos::parallel_for.

6.2 Experiments and Results

6.2.1 Experimental Setup

By using Kokkos and Kokkos Kernels, we developed the Kokkos-based variant of HPCG, which we named KoHPCG. We configured the Kokkos and Kokkos Kernels packages using Trilinos with CMake 3.26.2 and C++17 with Intel MKL, MPI, and OpenMP support.

We conducted experiments on two different hardware platforms on the KISTI Nurion system [117], Intel Xeon Phi 7250 (KNL) and Intel Xeon Gold 6148 (SKL) processors. All experiments employed an MPI+OpenMP parallelization strategy. The number of OpenMP threads assigned per MPI process was determined using the relation:

$$\text{OpenMP Threads per MPI Process} = \frac{\text{Total Number of Cores}}{\text{Number of MPI Processes}}.$$

The KNL processor features 68 cores per node, along with 96 GB DDR4 and 16 GB high-bandwidth memory (HBM). The SKL processor consists of 40 cores per node with 192 GB of memory. For example, when two MPI processes are launched on KNL and SKL, each process is assigned 34 and 20 OpenMP threads, respectively. This threads per MPI process mapping scheme was consistently applied across all test cases unless stated otherwise.

We conducted experiments on 1, 2, 4, 8, and 16 nodes. In the results, the notation

TP:40_ N:2_ MN:20_ T:3 signifies that the result pertains to 20 MPI processes with 3 OpenMP threads per process across 2 nodes, totaling 40 MPI processes. TP denotes the total MPI processes, N signifies the number of nodes, MN represents the MPI processes per node, and T refers to the OpenMP threads utilized per process. The results are organized in ascending order based on the number of nodes and the MPI processes per node.

6.2.2 Results on Knights Landing (KNL)

We evaluated the performance of HPCG variants as shown in Figure 6.1, Reference, KHPCG, and KoHPCG (OUR), focusing on kernels performance in Gflops, total memory bandwidth, and overall HPCG performance, on problem sizes 64^3 and 160^3 on KNL single node and 1 MPI process with 68 threads. Evaluated on just 1 MPI process because of the limitation of KHPCG for fair performance comparison. The Reference version, representing the original HPCG v3.1, shows very bad MG performance, under 0.6 Gflops, because of the SymGS performance, which gives a low total HPCG performance of 0.62 Gflops. KHPCG shows a significant performance drop, especially in SpMV and MG, where it falls below 0.08 Gflops. The bandwidth utilization is very low in that case, and also the low HPCG performance of 0.08 Gflops, indicating it as ill-suited for the evaluated architecture.

Our Kokkos-based implementation, KoHPCG, outperforms other variants, especially in MG performance and the performance drop recorded in WAXPBY. MG peaks at 19.7 Gflops, with 5.49 Gflops on 64^3 and 4.97 Gflops on 160^3 in HPCG overall performance, KoHPCG performs better in overall HPCG and achieves memory bandwidth utilization over 129 GB/s. These results highlight the efficacy of the Kokkos programming model for performance portability and the major impact of

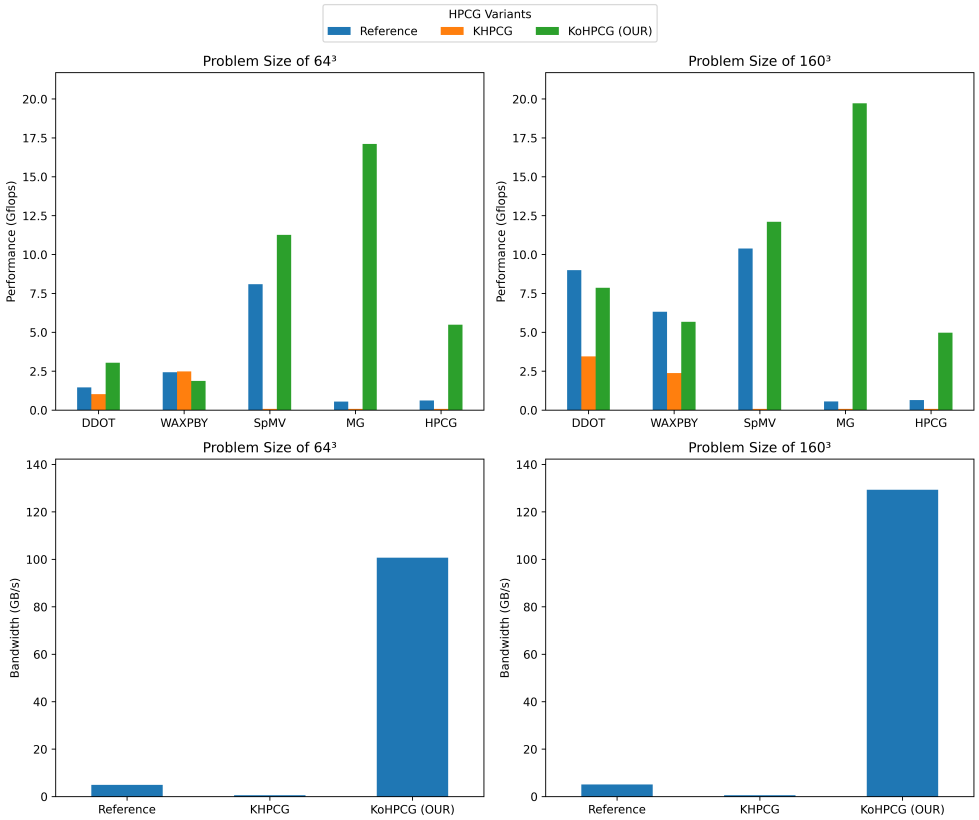


Figure 6.1 Comparison of performance (Gflops) and memory bandwidth (GB/s) across different HPCG variants for problem sizes 64^3 and 160^3, on single node and 1 MPI process with 68 threads. The Reference version corresponds to the original HPCG v3.1 implementation. KHPCG [8] is an early Kokkos-based but it is limited in scalability and stability. KoHPCG (OUR) is our Kokkos-based optimized version of HPCG with the kernels (DDOT, WAXPBY, SpMV, SymGS, and MG) ported to the Kokkos programming model, designed for performance portability.

the SymGS kernel on HPCG performance.

The Figure 6.2 shows a comparative performance analysis of Reference HPCG and KoHPCG for SpMV, MG, and the overall HPCG benchmark, quantified in Gflops across multi-node different configurations. HPCG required that at least one fourth memory of the system should be used, so on KNL, we evaluated the results on 160^3

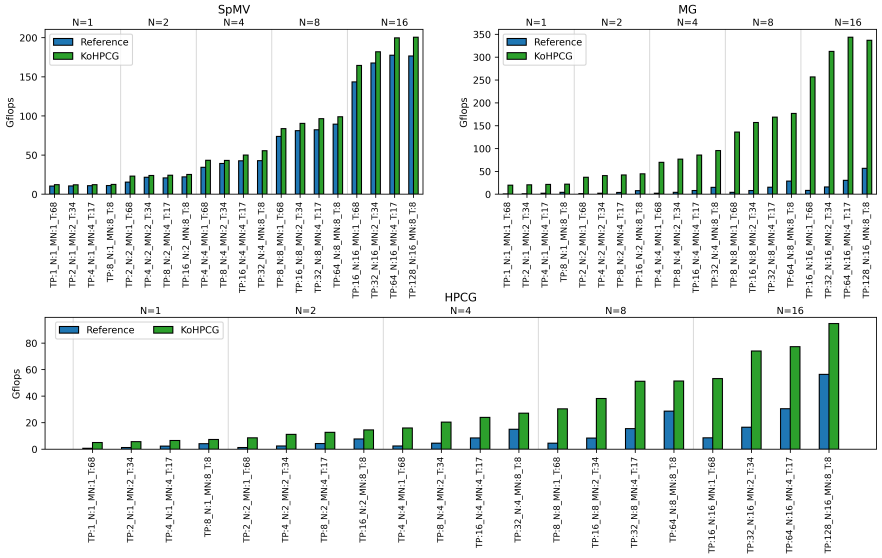


Figure 6.2 Comparison of Gflops performance for SpMV, MG, and overall HPCG on Intel KNL for a problem size of 160^3 across different multi-node configurations. The KoHPCG implementation exhibits superior performance relative to the Reference HPCG. Each configuration is represented in the format TP:X_N:Y_MN:Z_T:W, where TP signifies the total number of MPI processes, N indicates the number of nodes, MN represents the number of MPI processes per node, and T denotes the number of OpenMP threads per process. The notation TP:40_N:2_MN:20_T:3 signifies 40 MPI processes allocated over 2 nodes, with 20 MPI processes per node and 3 OpenMP threads per process. Results are sorted in ascending order based on node count and MPI processes per node from left to right.

problem size on multi-node environment for 1, 2, 4, 8 and 16 nodes for different MPI + OpenMP settings. The graph illustrates the significant performance improvements of KoHPCG over the Reference implementation.

The performance improvement of KoHPCG relative to the Reference HPCG for a problem size of 160^3 on Intel KNL demonstrate the improvement across the computational kernels. These performace results are extracted from the HPCG report. The MG exhibits a performance improvement of $17.3\times$. In DDOT, it demonstrates a significant performance improvement of $4.65\times$ across the evaluated configurations. SpMV, although a critical and generally memory-bound kernel,

demonstrates a relatively modest improvement of 1.17×. Similarly, the WAXPBY kernel exhibits ignorable improvement in some configurations, but on 8 and 16 nodes, its performance dropped, indicating that it is further required to optimize. The MG consistently surpasses others kernels in improvement, with DDOT and overall HPCG demonstrating significant improvement. The little and no performance improvement of SpMV and WAXPBY suggests that further optimization may be required. KoHPCG provides a substantial and scalable performance improvement compared to the Reference implementation of HPCG.

6.2.3 Results on Skylake Scalable Processor (SKL)

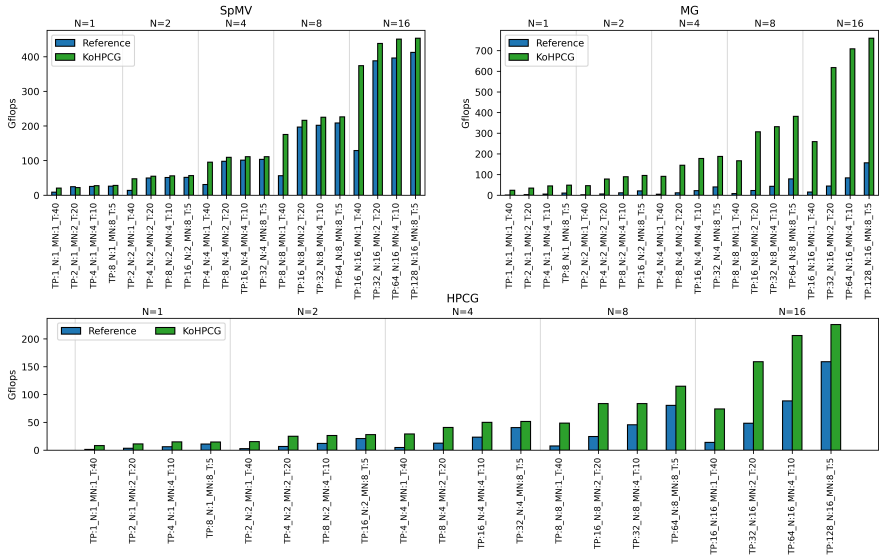


Figure 6.3 Comparison of Gflops performance for SpMV, MG, and overall HPCG on Intel SKL for a problem size of 192^3 across different multi-node configurations.

Figure 6.3 shows the performance on problem size of 192^3, For the SpMV kernel, KoHPCG consistently outperforms the Reference with improvement in performance. Similarly in the MG kernel, where KoHPCG demonstrates improvement, up to 5× particularly in higher nodes like TP:128_N:16, where performance improves

from 156.5 Gflops to 760.1 Gflops. For the overall HPCG performance, KoHPCG continues to deliver substantial improvements across different configurations, often doubling the performance of the Reference implementation.

The performance improvement trends of KoHPCG over Reference for HPCG benchmark kernels DDOT, WAXPBY, SpMV, MG and overall HPCG performance on the SKL system, evaluated across multiple problem sizes of 64^3, 192^3, and 320^3 on 1, 2, 4, 8, and 16 nodes. Which highlights the MG kernel as having the substantial performance improvement upto 11.75×, followed by DDOT 7.71×, and moderate improvements in SpMV 1.72× and HPCG 3.41×. WAXPBY demonstrates a performance decline and is of 0.84×.

The parallelization strategy for SymGS significantly reduces data dependency bottlenecks, but it experiences additional overhead from increased use of memory. The current implementation of KoHPCG requires additional memory to transform the original HPCG data structures into Kokkos-compatible formats. The original data structures must remain unaltered to ensure appropriate benchmarking of the reference implementations of the core computational kernels. Duplicate data structures are required. This limitation affects our capacity to execute KoHPCG with increased MPI process counts on larger problem sizes due to memory constraints of the system.

This is an ongoing work, and future efforts will focus on improving memory efficiency and extending evaluation to GPU-based systems. Our evaluation revealed that systems with just 96 GB of DDR memory performing better than the system which using an extra 16 GB of High Bandwidth Memory (HBM) in HPCG performance. Although the higher bandwidth of HBM, it did not yield improved performance for memory-bound operations such as SpMV and SymGS. However, the additional memory facilitated execution on a slightly higher number of MPI

processes.

The performance of the WAXPBY kernel is suboptimal in some configurations, indicating a necessity for additional tuning. So far, KoHPCG has been tested on Intel architectures, comprehensive validation on diverse platforms, including GPUs, is part of our planned future work to evaluate its full portability and performance potential.

6.2.4 Results on GPU Based System

The GPU-based results were obtained on a system equipped with an Intel Core i9-10920X CPU (24 cores) and NVIDIA GeForce RTX 2080 Ti GPU. The KoHPCG benchmark was evaluated for problem sizes of 64^3, 96^3, 128^3, and 160^3, achieving Gflops performances of 8.96, 9.46, 9.24, and 9.15, respectively. These results were generated using a single MPI process and a single GPU. While the current setup is limited by the absence of multi-GPU support, likely due to some Kokkos configurational setup issues or missing device-aware adaptations in the code, these results however demonstrate successful GPU-based execution and highlight the portability of KoHPCG across heterogeneous architectures. We will address the existing limitations by refining the Kokkos configuration and restructuring the code to support scalable multi-GPU execution.

CHAPTER 7. Conclusion

This thesis introduced and implemented **KoHPCG**, a performance-portable and optimized variant of the HPCG benchmark. To address the critical scalability limitations of the original reference implementation, parallelizing the inherently sequential Symmetric Gauss-Seidel (SymGS) kernel through the design and evaluation of several algorithmic variants, including temporal blocking, wavefront-style dependency scheduling, over-relaxation etc.

Through experimentation on KNL and Skylake (SKL) systems, KoHPCG demonstrated improved scalability, better utilization of memory bandwidth, and enhanced computational throughput. In particular, the over-relaxation variant proved to be a compelling choice, offering simplicity in implementation and robust performance. The developed SymGS variants preserved the numerical correctness, while enabling effective use of shared-memory parallelism via OpenMP and Kokkos execution abstractions.

A key contribution of this work is to develop parallel variant of SymGS and its integration into a Kokkos-based framework including transformation of the core computational kernel of HPCG into Kokkos, for performance portability across heterogeneous architectures. Unlike prior efforts such as KHPCG, KoHPCG supports multi-node, multi-threaded execution and resolves the limitations related

to parallelism and numerical stability.

The broader implication of this research lies in providing the HPC community with a modernized, scalable, and portable benchmarking tool. It serves as a foundation for further research in memory-bound kernel optimization and performance-portable programming.

7.1 Future Research Directions

Looking ahead, several promising directions can further advance this research:

- **GPU and Heterogeneous Platform Support:** Extend KoHPCG to exploit GPU architectures using Kokkos' CUDA and HIP backends, and evaluate its performance on GPU systems.
- **Hybrid SymGS Strategies:** Investigate combinations of temporal blocking with multicolor or algebraic coloring techniques to further reduce synchronization overhead and improve data locality.
- **Advanced Scheduling Techniques:** Explore task-based parallelism and asynchronous execution models using Kokkos to optimize communication-computation overlap.
- **Benchmark Generalization:** Broaden the benchmark' s capabilities to support more realistic partial differential equation (PDE) systems by developing better preconditioner and multigrid methods. Geometric multigrid methods can be significantly improved with algebraic multigrid methods. These will bring even better convergence rates for grids and complex problem domains.

REFERENCES

[1] J. J. Dongarra, "The linpack benchmark: An explanation," in *International Conference on Supercomputing*. Springer, 1987, pp. 456–474.

[2] J. Dongarra, P. Luszczek, and M. Heroux, "HPCG technical specification," *Sandia National Laboratories, Sandia Report SAND2013-8752*, 2013.

[3] ——, "HPCG: one year later," *ISC14 Top500 BoF*, vol. 818, 2014.

[4] J. Dongarra, M. A. Heroux, and P. Luszczek, "High-performance conjugate-gradient benchmark: A new metric for ranking high-performance computing systems," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 3–10, 2016.

[5] ——, "A new metric for ranking high-performance computing systems," *National Science Review*, vol. 3, no. 1, pp. 30–35, 2016.

[6] Kokkos Team, "Kokkos: The manycore performance portability programming model," https://kokkos.org/, 2020, accessed: 2024-08-22.

[7] Z. Bookey, "Performance portable high performance conjugate gradients benchmark," All College Thesis, College of Saint Benedict/Saint John's

University, 2016, accessed: 2024-08-14. [Online]. Available: https://digitalcommons.csbsju.edu/honors_thesis/12

[8] ——, "KHPCG 3.0," https://github.com/zabookey/KHPCG3.0, 2016, accessed: 2024-02-19.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of parallel and distributed computing*, vol. 74, no. 12, pp. 3202–3216, 2014.

[10] C. R. Trott, D. Lebrun-Grandié, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 805–817, 2021.

[11] C. Ulmer, "Sand2021-1220: Benchmarking the nvidia a100 graphics processing unit for high-performance computing and data analytics workloads," Sandia National Laboratories, Tech. Rep., 2021, accessed: 2024-08-14. [Online]. Available: https://www.craigulmer.com/data/2021/SAND2021-1220_uur.pdf

[12] M. A. Heroux and J. Dongarra, "Toward a new metric for ranking high performance computing systems." Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); University of Tennessee, Tech. Rep., 2013.

[13] T. Hoefler, P. Gottschling, A. Lumsdaine, and W. Rehm, "Optimizing a conjugate gradient solver with non-blocking collective operations," *Parallel Computing*, vol. 33, no. 9, pp. 624–633, 2007.

[14] G. Meurant, “Multitasking the conjugate gradient method on the cray x-mp/48,” *Parallel Computing*, vol. 5, no. 3, pp. 267–280, 1987.

[15] A. T. Chronopoulos and C. W. Gear, “S-step iterative methods for symmetric linear systems,” *Journal of Computational and Applied Mathematics*, vol. 25, no. 2, pp. 153–168, 1989.

[16] V. Eijkhout, *Lapack working note 51: Qualitative properties of the conjugate gradient and lanczos methods in a matrix framework*. Citeseer, 1992.

[17] E. D’Azevedo, V. Eijkhout, and C. Romine, “Lapack working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors,” University of Tennessee, Knoxville, USA, Technical Report CS-93-185, 1993.

[18] J. Dongarra and V. Eijkhout, “Finite-choice algorithm optimization in conjugate gradients,” *Computer Science Technical Report UT-CS-03-502, University of Tennessee, Knoxville*, 2003.

[19] P. Ghysels and W. Vanroose, “Hiding global synchronization latency in the preconditioned conjugate gradient algorithm,” *Parallel Computing*, vol. 40, no. 7, pp. 224–238, 2014.

[20] M. Tiwari and S. Vadhiyar, “Efficient executions of pipelined conjugate gradient method on heterogeneous architectures,” *arXiv preprint arXiv:2105.06176*, 2021.

[21] A. Zeni, K. O’ Brien, M. Blott, and M. D. Santambrogio, “Optimized implementation of the HPCG benchmark on reconfigurable hardware,” in *European Conference on Parallel Processing*. Springer, 2021, pp. 616–630.

[22] C. Edwards, C. Trott *et al.*, "Kokkos tutorial: Kokkos kernels," https://indico.math.cnrs.fr/event/12037/attachments/5040/8157/KokkosTutorial_08_KokkosKernels.pdf, 2024, accessed May 2025.

[23] J. Park, A. Kleymenov, M. Smelyanskiy, and V. Pirogov, "HPCG on intel architecture update nov 2015," https://www.hpcg-benchmark.org/downloads/sc15/sc15-hpcg-bof-intel.pdf, 2015, [Accessed 23-08-2024].

[24] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, "Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 945–955.

[25] Intel Corporation, "Intel math kernel library release notes and new features," https://www.intel.com/content/www/us/en/developer/articles/release-notes/intel-math-kernel-library-release-notes-and-new-features.html, 2015, accessed: 2024-08-26.

[26] I. Labs, "Spmp: A high-performance sparse matrix library," https://github.com/IntelLabs/SpMP, 2016, accessed: 2024-08-26.

[27] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, M. M. A. Patwary, V. Pirogov, P. Dubey, X. Liu, C. Rosales *et al.*, "Optimizations in a high-performance conjugate gradient benchmark for ia-based multi-and many-core processors," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 11–27, 2016.

[28] Q. Pan and X. Wang, “Performance evaluation and optimization of HPCG benchmark on cpu+ mic platform,” *International Journal of Hybrid Information Technology*, vol. 9, no. 11, pp. 239–254, 2016.

[29] F. Yuan, X. Yang, S. Li, D. Dong, C. Huang, and Z. Wang, “Optimizing multi-grid preconditioned conjugate gradient method on multi-cores,” *IEEE Transactions on Parallel and Distributed Systems*, 2024.

[30] 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, technical Report.

[31] D. Ruiz, F. Spiga, M. Casas, M. Garcia-Gasulla, and F. Mantovani, “Open-source shared memory implementation of the HPCG benchmark: Analysis, improvements and evaluation on cavium thunderx2,” in *2019 International Conference on High Performance Computing and Simulation* (*HPCS*). IEEE, 2019, pp. 225–232.

[32] A. Scolari and A.-J. Yzelman, “Effective implementation of the high performance conjugate gradient benchmark on graphblas,” in *2023 IEEE International Parallel and Distributed Processing Symposium Workshops* (*IPDPSW*). IEEE, 2023, pp. 216–225.

[33] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke *et al.*, “Mathematical foundations of the graphblas,” in *2016 IEEE High Performance Extreme Computing Conference* (*HPEC*). IEEE, 2016, pp. 1–9.

[34] A. Yzelman, D. Di Nardo, J. Nash, and W. Suijlen, “A c++ graphblas: specification, implementation, parallelisation, and evaluation,” *Preprint*,

vol. 58, 2020.

[35] A. Scolari and A.-J. Yzelman, "Alp/graphblas," https://github.com/Algebraic-Programming/ALP, 2023, accessed: 2024-08-30.

[36] X. Yang, S. Li, F. Yuan, D. Dong, C. Huang, and Z. Wang, "Optimizing multi-grid computation and parallelization on multi-cores," in *Proceedings of the 37th International Conference on Supercomputing*, 2023, pp. 227–239.

[37] A. McAdams, E. Sifakis, and J. Teran, "A parallel multigrid poisson solver for fluids simulation on large grids." in *Symposium on Computer Animation*, vol. 65, 2010, p. 74.

[38] K. Kumahata, K. Minami, and N. Maruyama, "High-performance conjugate gradient performance improvement on the k computer," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 55–70, 2016.

[39] C. Liao, J. Chen, W. Han, H. Cao, Z. Su, W. Yin, and H. An, "A hierarchical grid algorithm for accelerating high-performance conjugate gradient benchmark on sunway many-core processor," in *Proceedings of the 3rd International Conference on Communication and Information Processing*, 2017, pp. 361–368.

[40] 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 Transactions on Architecture and Code Optimization* (*TACO*), vol. 15, no. 1, pp. 1–20, 2018.

[41] 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 *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–13.

[42] Q. Zhu, H. Luo, and C. Yang, "Reference and optimized versions of HPCG Benchmark," Aug. 2021. [Online]. Available: https://doi.org/10.5281/zenodo.5286616

[43] K. Komatsu, R. Egawa, Y. Isobe, R. Ogata, H. Takizawa, and H. Kobayashi, "An approach to the highest efficiency of the HPCG benchmark on the sx-ace supercomputer," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis* (*SC15*), *Poster*, 2015, pp. 1–2.

[44] S. Momose, T. Hagiwara, Y. Isobe, and H. Takahara, "The brand-new vector supercomputer, sx-ace," in *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 2014, pp. 199–214.

[45] E. F. D' Azevedo, M. R. Fahey, and R. T. Mills, "Vectorized sparse matrix multiply for compressed row storage format," in *Computational Science–ICCS 2005: 5th International Conference, Atlanta, GA, USA, May 22-25, 2005. Proceedings, Part I 5*. Springer, 2005, pp. 99–106.

[46] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on cuda," Nvidia Technical Report NVR-2008-004, Nvidia Corporation, Tech. Rep., 2008.

[47] T. Iwashita, H. Nakashima, and Y. Takahashi, "Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg

method," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*. IEEE, 2012, pp. 474–483.

[48] Y. OYANAGI, "Hyperplane vs. multicolor vectorization of incomplete lu preconditioning," *Journal of Information Processing*, vol. 2, no. 1, 1987.

[49] S. Fujino, M. Mori, and T. Takeuchi, "Performance of hyperplane ordering on vector computers," *Journal of Computational and Applied Mathematics*, vol. 38, no. 1-3, pp. 125–136, 1991.

[50] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "HPCG on long-vector architectures: Evaluation and optimization on nec sx-aurora and risc-v," *Future Generation Computer Systems*, vol. 143, pp. 152–162, 2023.

[51] C. Gómez, F. Mantovani, E. Focht, and M. Casas, "Ve-native port of the HPCG benchmark," https://github.com/efocht/hpcg-ve-open, 2021.

[52] E. Vermij, L. Fiorin, C. Hagleitner, and K. Bertels, "Boosting the efficiency of HPCG and graph500 with near-data processing," in *2017 46th International Conference on Parallel Processing* (*ICPP*). IEEE, 2017, pp. 31–40.

[53] W. J. Starke, J. Stuecheli, D. Daly, J. Dodson, F. Auernhammer, P. Sagmeister, G. L. Guthrie, C. F. Marino, M. Siegel, and B. Blaner, "The cache and memory subsystems of the ibm power8 processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 3–1, 2015.

[54] Graph 500, "Graph 500 benchmark," https://graph500.org/, 2017, accessed: 2024-08-13.

[55] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proceedings of 2011 International Conference for High*

Performance Computing, Networking, Storage and Analysis, 2011, pp. 1–12.

[56] E. Phillips and M. Fatica, "A cuda implementation of the high performance conjugate gradient benchmark," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*. Springer, 2014, pp. 68–84.

[57] ——, "Optimizing the high performance conjugate gradient benchmark on gpus," 2014. [Online]. Available: https://developer.nvidia.com/blog/optimizing-high-performance-conjugate-gradient-benchmark-gpus/

[58] M. Luby, "A simple parallel algorithm for the maximal independent set problem," in *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, 1985, pp. 1–10.

[59] M. T. Jones and P. E. Plassmann, "A parallel graph coloring heuristic," *SIAM Journal on Scientific Computing*, vol. 14, no. 3, pp. 654–669, 1993.

[60] J. Cohen and P. Castonguay, "Efficient graph matching and coloring on the gpu," in *GPU Technology Conference*, 2012, pp. 1–10.

[61] NVIDIA, "cusparse documentation," https://docs.nvidia.com/cuda/cusparse/, n.a, accessed: 2024-08-13.

[62] O. R. N. Laboratory, "Cray xk7 system at oak ridge national laboratory," 2024, accessed: 2024-08-13. [Online]. Available: https://www.ornl.gov

[63] S. N. S. Centre, "Cray xc30 system at the swiss national supercomputing centre," 2024, accessed: 2024-08-13. [Online]. Available: https://www.cscs.ch

[64] A. Bland, W. Joubert, D. Maxwell, N. Podhorszki, J. Rogers, G. Shipman, and A. Tharrington, "Titan: 20-petaflop cray xk7 at oak ridge national laboratory," in *Contemporary High Performance Computing*. Chapman and Hall/CRC, 2017, pp. 399–420.

[65] S. R. Alam, L. Gilly, C. J. McMurtrie, and T. C. Schulthess, "Cscs and the piz daint system," in *Contemporary High Performance Computing*. CRC Press, 2019, pp. 149–173.

[66] E. Phillips and M. Fatica, "Performance analysis of the high-performance conjugate gradient benchmark on gpus," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 28–38, 2016.

[67] E. Rothberg and A. Gupta, "Parallel iccg on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck," *Parallel Computing*, vol. 18, no. 7, pp. 719–741, 1992.

[68] U. M. Yang *et al.*, "Boomeramg: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, no. 1, pp. 155–177, 2002.

[69] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, "Sparsifying synchronization for high-performance shared-memory sparse triangular solver," in *Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29*. Springer, 2014, pp. 124–140.

[70] X. Zhang, C. Yang, F. Liu, Y. Liu, and Y. Lu, "Optimizing and scaling HPCG on tianhe-2: early experience," in *Algorithms and Architectures for Parallel*

Processing: 14th International Conference, ICA3PP 2014, Dalian, China, August 24-27, 2014. Proceedings, Part I 14. Springer, 2014, pp. 28–41.

[71] A. Monakov, A. Lokhmotov, and A. Avetisyan, "Automatically tuning sparse matrix-vector multiplication for gpu architectures," in *High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5*. Springer, 2010, pp. 111–125.

[72] S. Williams, D. D. Kalamkar, A. Singh, A. M. Deshpande, B. Van Straalen, M. Smelyanskiy, A. Almgren, P. Dubey, J. Shalf, and L. Oliker, "Optimization of geometric multigrid for emerging multi-and manycore processors," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[73] Y. Liu, X. Zhang, C. Yang, F. Liu, and Y. Lu, "Accelerating HPCG on tianhe-2: a hybrid cpu-mic algorithm," in *2014 20th IEEE International Conference on Parallel and Distributed Systems* (*ICPADS*). IEEE, 2014, pp. 542–551.

[74] C. Chen, Y. Du, H. Jiang, K. Zuo, and C. Yang, "Hpcg: preliminary evaluation and optimization on tianhe-2 cpu-only nodes," in *2014 IEEE 26th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 2014, pp. 41–48.

[75] F. Liu, C. Yang, Y. Liu, X. Zhang, and Y. Lu, "Reducing communication overhead in the high performance conjugate gradient benchmark on tianhe-2," in *2014 13th International Symposium on Distributed Computing and Applications to Business, Engineering and Science*. IEEE, 2014, pp. 13–18.

[76] Y. Liu, C. Yang, F. Liu, X. Zhang, Y. Lu, Y. Du, C. Yang, M. Xie, and X. Liao, "623 tflop/s HPCG run on tianhe-2: Leveraging millions of hybrid cores," *The International Journal of High Performance Computing Applications*, vol. 30, no. 1, pp. 39–54, 2016.

[77] R. Steiger, "HPCG for fpgas: A data-centric approach," B.S. thesis, ETH Zurich, 2022.

[78] J. Dongarra, M. Heroux, and P. Luszczek, "HPCG benchmark," 2019, accessed: 2024-01-21. [Online]. Available: https://github.com/hpcg-benchmark/hpcg

[79] C. B. Cristiano Malossi, Panagiotis Chatzidoukas, "Ibm hpcg," https://github.com/IBM/HPCG, IBM Research, 2018, accessed: 2024-04-11.

[80] Intel, "Getting started with intel cpu optimized hpcg," https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2024-1/getting-started-with-intel-cpu-optimized-hpcg.html, 2015, accessed: 2024-03-22.

[81] N. John C. Linford, "Arm HPCG benchmarks," https://gitlab.com/arm-hpc/benchmarks/hpcg, 2017, accessed: 2024-06-17.

[82] D. Ruiz, "HPCG for arm," https://github.com/ARM-software/HPCG_for_Arm, 2020, accessed: 2024-08-01.

[83] Intel, "Versions of the intel gpu optimized hpcg," https://www.intel.com/content/www/us/en/docs/onemkl/developer-guide-linux/2024-1/versions-of-the-intel-gpu-optimized-hpcg.html, 2020, accessed: 2024-01-09.

[84] N. S. Mohammad Almasri, “Nvidia hpc benchmarks,” https://github.com/NVIDIA/nvidia-hpcg, NVIDIA, 2023, accessed: 2024-07-05.

[85] NVIDIA, “Nvidia hpc benchmarks container,” 2024, accessed: 2024-09-02. [Online]. Available: https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks

[86] AMD, “rochpcg: HPCG benchmark based on rocm platform,” https://github.com/ROCm/rocHPCG, 2020, accessed: 2024-08-02.

[87] A. Zeni, K. O’Brien, M. Blott, and M. D. Santambrogio, “HPCG fpga,” https://github.com/Xilinx/HPCG_FPGA, 2019, accessed: 2024-05-28.

[88] “HPCG benchmark,” https://www.hpcg-benchmark.org/, 2013, accessed: 2024-02-15.

[89] D. Ruiz, “Parallelizing hpcg’s main kernels,” 2018, accessed: 2024-09-02. [Online]. Available: https://community.arm.com/arm-community-blogs/b/high-performance-computing-blog/posts/parallelizing-hpcg

[90] I. P. S. P. Group, “High performance conjugate gradient benchmark (hpcg) for ibm power9 systems,” November 2018, sC18, Dallas, Texas. [Online]. Available: https://www.hpcg-benchmark.org/downloads/sc18/HPCG_IBM_P9_v05.pdf

[91] X. Lulu and E. Strohmaier, “Optimizing HPCG for amd processors,” 2019, accessed: 2024-09-02. [Online]. Available: https://www.hpcg-benchmark.org/downloads/sc19/HPCG-AMD-Lulu.pdf

[92] AMD, “Optimizing HPCG with openmp on amd processors,” 2020, accessed: 2024-09-02. [Online]. Available: https://www.iwomp.org/

wp-content/uploads/iwomp-2020-Sponsor-AMD.pdf

[93] J. Park and M. Smelyanskiy, “Optimizing gauss–seidel smoother in hpcg,” in *ASCR HPCG Workshop*, 2014.

[94] K. Kumahata, K. Minami, and N. Maruyama, “HPCG on the k computer,” in *ASCR HPCG Workshop*, 2014.

[95] K. Kumahata and K. Minami, “HPCG performance improvement on the k computer,” in *Presentation at Supercomputers Conference* (*SC’14*), 2014.

[96] S. Kopysov, N. Nedozhogin, and L. Tonkov, “Parallel pipelined conjugate gradient algorithm on heterogeneous platforms,” *International Journal of Computer and Information Engineering*, vol. 16, no. 10, pp. 423–430, 2022.

[97] N. Bell and M. Garland, “Implementing sparse matrix-vector multiplication on throughput-oriented processors,” in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[98] J. Gao, B. Liu, W. Ji, and H. Huang, “A systematic literature survey of sparse matrix-vector multiplication,” *arXiv preprint arXiv:2404.06047*, 2024.

[99] D. Bozdağ, U. Catalyurek, A. H. Gebremedhin, F. Manne, E. G. Boman, and F. Özgüner, “A parallel distance-2 graph coloring algorithm for distributed memory computers,” in *International conference on high performance computing and communications*. Springer, 2005, pp. 796–806.

[100] J. R. Blair and F. Manne, “An efficient self-stabilizing distance-2 coloring algorithm,” *Theoretical Computer Science*, vol. 444, pp. 28–39, 2012.

[101] T. Iwashita and M. Shimasaki, “Algebraic multicolor ordering for parallelized iccg solver in finite-element analyses,” *IEEE transactions on magnetics*, vol. 38, no. 2, pp. 429–432, 2002.

[102] T. Mo and R. Li, “Accelerating stencil computation on gpgpu by novel mapping method between the global memory and the shared memory,” *Computing and Informatics*, vol. 37, no. 3, pp. 533–552, 2018.

[103] G. Sornet, F. Dupros, and S. Jubertie, “A multi-level optimization strategy to improve the performance of stencil computation,” *Procedia Computer Science*, vol. 108, pp. 1083–1092, 2017.

[104] S. M. F. Rahman, Q. Yi, and A. Qasem, “Understanding stencil code performance on multicore architectures,” in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011, pp. 1–10.

[105] K. Datta, S. Williams, V. Volkov, J. Carter, L. Oliker, J. Shalf, and K. Yelick, “Auto-tuning the 27-point stencil for multicore,” in *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, vol. 70, 2009.

[106] J. Brown, “Scalable reconfigurable dataflow architectures for the high performance conjugate gradient benchmark,” Master’s thesis, University of California, Davis, 2024.

[107] L. Sinjorgo, “Multicoloring of highly symmetric graphs,” *Master’s thesis, Tilburg University, The Netherlands*, 2021.

[108] M. M. Halldórsson and G. Kortsarz, “Multicoloring: Problems and techniques,” in *International Symposium on Mathematical Foundations of*

Computer Science. Springer, 2004, pp. 25–41.

[109] M. da Silva Menezes, P. H. L. Silva, J. P. C. de Oliveira, R. L. Marques, and I. Mezzomo, "A parallel iterative hybrid gauss-jacobi-seidel method," *Journal of Computational and Applied Mathematics*, p. 116629, 2025.

[110] A. Ahmadi, F. Manganiello, A. Khademi, and M. C. Smith, "A parallel jacobi-embedded gauss-seidel method," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 6, pp. 1452–1464, 2021.

[111] X. Yang, N. Wang, and L. Xu, "A parallel gauss-seidel method for convex problems with separable structure," *Numerical Algebra, Control and Optimization*, vol. 10, no. 4, pp. 557–570, 2020.

[112] V. A. Ulmeanu and A. P. Ulmeanu, "Parallelised hybrid heuristics in numerical linear algebra for data science," *Available at SSRN 4437899*.

[113] D. Xie and L. Adams, "New parallel sor method by domain partitioning," *SIAM Journal on Scientific Computing*, vol. 20, no. 6, pp. 2261–2281, 1999.

[114] D. Xie, "A new block parallel sor method and its analysis," *SIAM Journal on Scientific Computing*, vol. 27, no. 5, pp. 1513–1533, 2006.

[115] ——, "New parallel symmetric sor preconditioners by multi-type partitioning," *International Journal of Computer Mathematics*, vol. 86, no. 2, pp. 287–300, 2009.

[116] D. M. Young, *Iterative solution of large linear systems*. Elsevier, 2014.

[117] KISTI, "National supercomputing center - nurion," https://www.ksc.re.kr/eng/resources/nurion, 2018, accessed: 2024-08-05.