[2025-2]

KEASE 기술서

한국형 엑사스케일 응용 SW 개발 환경 프레임워크

High-Performance Conjugate Gradient Benchmark
- A Comprehensive Survey -

2024. 12. 31

숭 실 대 학 교

[개 정 이 력]

개정 일자	버전	설명	개정페이지	작성자	확인자
2024.12.31	1.0	INITIAL DRAFT	전체	Muhammad Rizwan	최재영

목 차

1. Abstract

This survey paper focuses on examining the optimization techniques and trends for the High-Performance Conjugate Gradient (HPCG) benchmark employed in the last ten years. The HPCG benchmark was introduced to eliminate the limitations of the High-Performance Linpack (HPL) benchmark and reflect the realistic performance measure of modern supercomputer architectures. Our study evaluates HPCG optimizations performed by High-Performance Computing (HPC) researchers on diverse hardware architectures such as CPU, GPU, MIC, and FPGA, etc., with a focus on optimizing the reference HPCG benchmark code for data formats, parallelization strategies, and architecture-specific tuning. We reviewed the optimizations performed by the researchers and presented a comprehensive analysis of these optimizations. This work offers the first comprehensive review of HPCG optimizations, aiming to discuss the previous findings and provide a systematic analysis for further optimizations in the future. Our study aims to guide researchers in identifying the most suitable directions to expand their knowledge and develop further optimization strategies in the HPCG benchmark.

2. Introduction

2.1 Background

The High-Performance Linpack (HPL) benchmark [1], since the 1990s, has been used to measure the performance of supercomputers and mostly used to solve dense linear algebraic equations. Though HPL has been a reliable standard for many years, modern supercomputers and new applications have revealed their limitations. HPL measures peak performance, but it does not accurately reflect how modern applications perform. These applications are more complex and require optimal coordination of various functions of the computer system. High-Performance Computing (HPC) community uses the High-Performance Conjugate Gradient (HPCG) [2–5] as a new benchmark to evaluate and compare the performance of modern supercomputers. The HPCG benchmark is based on the conjugate gradient method, which works on symmetric positive definite (SPD) matrices. HPCG specifically addresses the limitations of HPL and provides an alternative to the HPL benchmark, which has been in use for decades. HPCG better evaluates the performance of modern applications by measuring their ability to solve more complex problems. HPCG tests the coordination of all parts of the computing machine, such as memory bandwidth, computation power, interconnect network performance, and overall system synergy. HPCG is a benchmark that better reflects the performance of modern, practical, and real-life applications.

2.2 Motivation

The HPCG benchmark, introduced by Dongarra and Heroux in 2013 [2, 6], is used to evaluate the performance of modern computing systems. This benchmark not only evaluates the computational capacity of the system but also covers other aspects, especially memory access patterns and communication behavior. HPCG has gained significant attention from researchers because it represents the real-world application benchmark compared to its classical counterpart, the HPL benchmark. Researchers focused on optimizing the performance of HPCG on various architectures, including CPUs, GPUs, MICs, FPGAs, etc. Many researchers contributed and performed optimizations using different techniques and methodologies and presented their valuable work. Our motivation for writing this survey paper stems from the need for a comprehensive overview of a variety of techniques and strategies used in the optimization of HPCG. To the

best of our knowledge, this is the first work, and there is no such comprehensive survey that summarizes the work progressed on HPCG optimization over the past decade. Our study aims to fill this gap by providing valuable information to researchers by consolidating previous works and identifying future research directions in this field. The primary objective of this study is to summarize the challenges in optimizing the HPCG benchmark and the progress made by the researchers in the last decades and provide a clear and systematic overview.

2.3 Research Questions

This study will answer the following research questions:

- RQ01: How have benchmark optimization techniques for HPCG evolved over the last decade, and what is the best strategy for performance improvement across hardware architectures: CPUs, GPUs, FPGAs, etc.?
- RQ02: Which data formats are used by existing implementations of the HPCG benchmark, and how does the format affect the effectiveness of operations over sparse matrices in the benchmark?
- RQ03: What parallelization strategies and hardware-specific optimizations, e.g., vectorization, task offioading, etc., are being applied for the optimization of HPCG on modern architectures?
- RQ04: What are the current issues or constraints in enhancing the HPCG benchmark on various architectures, and what must be done to minimize the bottlenecks so as to make it highly scalable?
- RQ05: For the development and optimization of the HPCG benchmark, what trends are expected in the future, and what research directions can be explored to overcome current challenges?

In relation to RQ01, we have studied previous papers listed in Table 4 and analyzed the evolution of optimization techniques across various hardware architectures. This reveals how different optimizations have adapted to different hardware architectures. Regarding RQ02 and 03, we summarize the information in Sections 4 and 5 regarding all possible parallelization strategies and data formats employed by the researchers to enhance the performance of the HPCG benchmark. RQ04 enables us to understand the problems in HPCG optimizations, and we discussed these in Section 7.1, considering the existing research and elaborating on how the researchers had tackled these and mentioned in Section 6. Finally, regarding RQ05, the emerging trends and future possible HPCG benchmark optimizations are discussed in Section 7.2.

2.4 Contributions

This survey paper will add value to the field of HPC with the following contributions:

- First comprehensive overview of the HPCG optimizations that researchers have developed in the last decade.
- Analyze the optimization approaches employed for HPCG across different hardware architectures.
- Summarize the influence of data formats and parallelization approaches on the performance of HPCG.
- Identify the issues in improving the performance of HPCG and provide a foundation for further investigations to impact optimization strategies in the future.

2.5 Need for HPCG benchmark (HPL VS HPCG)

Fugaku and Frontier are the top two supercomputers in terms of HPCG benchmark as of the most recent June 2024 TOP500 [7] rating. They were placed first and fourth, respectively, as shown in Table 1. Although Frontier was ranked first in the HPL benchmark, Fugaku outperformed it in the HPCG benchmark. Supercomputer Fugaku achieves an Rmax of 442.01 Pflops, which is 82.2% of its Rpeak of 537.21 Pflops. Frontier appears to be the most powerful, with a Rmax of 1,206.00 Pflops, almost tripling the Fugaku Rmax performance of 442.01 Pflops.

HPCG implementation has local and global communication, which impacts performance compared to HPL, which has a high computation to communication ratio.

Table 1. Comparison of HPCG and HPL benchmark for Fugaku and Frontier

HPCG Rank	HPL Rank	System	Cores	Rpeak (Pflops)	Rmax (Pflops)	HPCG (Tflops)	Rmax vs Rpeak	HPCG vs Rmax
1	4	Supercomputer Fugaku - Fujitsu	7,630,848	537.21	442.01	16,004.50	82.28%	3.62%
2	1	Frontier HPE Cray - AMD	8,699,904	1,714.81	1,206.00	14,054.00	70.33%	1.17%

Fugaku shines in HPCG ranking, which is more relevant for real-world application performance measure. It achieves 82.28% of Rpeak's performance, which is significantly higher than Frontier's performance, 70.33%. Fugaku outperforms Frontier with 16,004.5 Tflops compared to Frontier's 14,054 Tflops in performance measure using HPCG. Fugaku achieves 3.62% of its Rmax and 2.98% of its Rpeak in the HPCG benchmark performance, whereas Frontier lags at 1.17% of its Rmax and just 0.82% of its Rpeak. This indicates that Fugaku is better at turning its theoretical power into practical results, making it potentially more effective for real-world applications. Frontier leads in HPL ranking, which is for computational performance measure, and is top ranked, while Fugaku excels in the efficiency of real-world application performance. Fugaku proves to be a competitor in its theoretical capabilities for practical applications. This simple comparison highlights the importance of the HPCG benchmark and shows it as a candidate for future benchmarks.

HPL uses Gaussian Elimination with partial pivoting for the solution of a dense linear system, and the efficiency of this is directly proportional to dense matrix-matrix multiplication, whereas HPCG uses the PCG method for solving the sparse systems of linear equations. The simple comparison of HPL VS HPCG is presented in Table 2.

Table 2. Comparison of HPL and HPCG

Feature	HPL	HPCG
Type of Measurement	Computing Performance	Practical Performance
Complexity	Tests simple problems	Tests complex problems
Utilization	Depends on processor speed	Depends on processor, memory, and network coordination
Applications	Based on dense linear algebra	Based on PCG and sparse linear algebra
Primary Limitation	Compute Power	Memory bandwidth
Emphasis	Theoretical peak performance	Real application performance

3. High-Performance Conjugate Gradient

HPCG is a new benchmark that is more relevant to real applications for high-performance computing (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 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 [6].

3.1 Preconditioned Conjugate Gradient (PCG) Method

The Conjugate Gradient (CG) method is a numerical method used to solve linear systems of equations. The CG method is used in HPCG because it is closer to real-world problems and better suited for assessing the practical performance of supercomputers. The Preconditioned Conjugate Gradient (PCG) method is an enhanced variant of the CG method that utilizes a preconditioner to accelerate convergence. The benchmark utilizes the Preconditioned Conjugate Gradient (PCG) algorithm [6], which is an iterative solver effective for SPD matrices and makes the computational cost high enough to measure the peak performance of the system when solving a linear system of equations [2].

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
$r_0 = b - Ax_0$ ▷ 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 matrix-vector product
  $\alpha_k = \frac{\text{rtz}_k}{p_k \cdot Ap_k}$ ▷ 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

Algorithm 1 “Preconditioned Conjugate Gradient (PCG)” begins by initializing an approximate solution $x0$ and computes the initial residual $r0 = b - Ax0$, which estimates the error in the initial guess, and the direction of search is an initially set as the residual, $p0 = r0$, and the algorithm iterates to update residual and search direction to reach for the solution converges. In each iteration, a multigrid (MG) preconditioner is used to improve convergence, then a dot product and a matrix-vector multiplication are performed to compute the step size αk, 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 the algorithm from performing when it checks for convergence by comparing the current residual against a given tolerance (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, ill conditioned sparse systems of equations. HPCG primarily relies on the performance

of the Sparse Matrix-Vector Multiplication (SpMV) and Symmetric Gauss-Seidel (SymGS). HPCG solves sparse linear equations with a simple additive Schwarz method using the PCG algorithm, which executes SymGS heavily when it performs MG preconditioner at line 8 in the Algorithm 1.

3.2 HPCG Execution Flow Process

The HPCG benchmark is designed to simulate real-world computation patterns usually found in scientific/engineering applications. Its execution flow begins with allocating 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 symmetric Gauss-Seidel (GS) 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 produce a report consisting of timing information, flops, memory bandwidth, and validation information.

HPCG forms the problem setup, and then a normalized symmetric positive definite matrix is created from the random matrix and compressed in 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 designed to optimally fit the machine's capacity to conduct an exhaustive assessment of its performance. HPCG benchmark implies the local MG preconditioner with SymGS operation. This preconditioner reduces the condition number of the matrix and, therefore, helps in faster convergence in the CG 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. Numerical results obtained at each iteration are checked with expected answers for verification, and the cache is cleared before each iteration. It eliminates cases of false popularity from cache usage and 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 the HPCG benchmark comprehensive and fair at the same time for HPC systems evaluation.

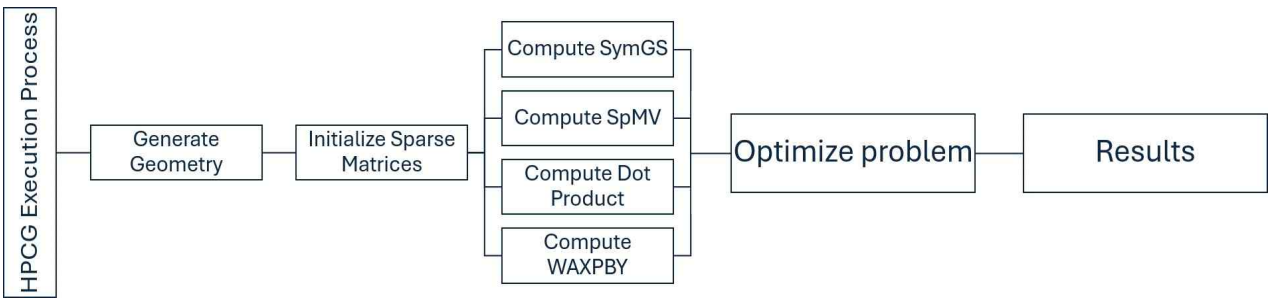

Fig. 1. HPCG Execution Process Flow

3.3 Problem Setup in HPCG

Fig. 2 shows a 27-point stencil 3D in HPCG benchmark designed to solve the linear system of equations:

$$A \cdot x = b,$$

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

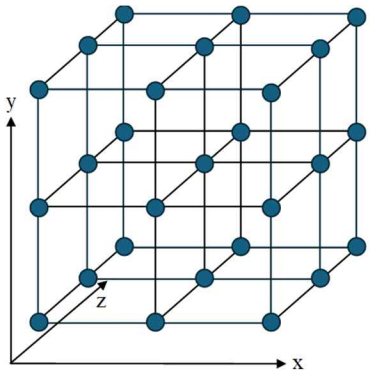

Fig. 2. 27 point stencil in HPCG

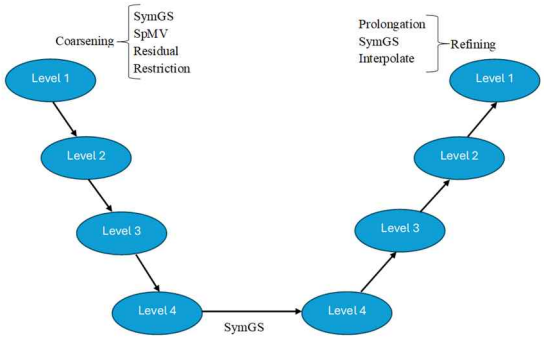

Fig. 3. Geometric multigrid V-cycle preconditioner in HPCG

The matrix A is sparse and in the context of the three-dimensional grid, every cell of the grid is connected to 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, 12 edge, and 8 corner neighbours. This leads to a sparse matrix format where there are at most 27 non-zero values in each row, representing each grid point with the current point and its neighbours. The matrix is weakly diagonally dominant for interior points and strongly diagonally dominant for boundary points, reflecting a conservation principle for interior points and zero Dirichlet boundary conditions at the edges. As a result, the matrix has the specific properties [4, 5], which is symmetric, positive definite, and non-singular. The matrix has up to 27 nonzero entries per row for the interior points and 7 to 18 nonzero entries for boundary points. The matrix A and the matching right-hand-side vector b are given, with all values 1.0 of an exact solution vector x.

The initial values for the solution vector is set to zeros ($xo = 0$). Then it approximate the values of the solution vector x using the PCG algorithm and each subdomain is further preconditioned with a SymGS, one of the core numerical kernels of HPCG.

The grid is subdivided into smaller subdomains. These subdomains are distributed among multiple MPI processes to parallelize the computation. Nx, Ny, and Nz are the dimensions of the local sub- domain, and the process layout, NRx, NRy, and NRz are a number of MPI processes in x, y, and z directions, respectively. The global domain size is thus given by $(NRx \times Nx) \times (NRy \times Ny) \times (NRz \times Nz)$, and the total computation is divided among the $NRx \times NRy \times NRz$ MPI processes, each handling a subgrid of the overall domain.

3.4 Optimization Constraints

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

3.4.1 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 that enhance the computational performance. When data structures are well-defined, there is improvement in the efficient usage of memory.

System-Specific Communication Infrastructure Optimization: can lead to overall performance im- provement. Utilization of specific network hierarchies and topologies tailored for hardware charac- teristics helps in improving communication strategies.

Advance MPI communication: if different types of MPI features are used, for example, synchroniza- tion using neighborhood collective is an efficient communication pattern for specific application need [8]. Advanced MPI features, of course, decrease the communication overhead and allow data to be shared in a more efficient way, which leads to scalability on the large core counts.

Data storage format: changes are permitted to the sparse matrix data structure to improve 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 components 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

3.4.2 Not Allowed Optimizations

Basic Conjugate Gradient Algorithm: optimization by means of different variants of the CG method avoiding some challenging aspects of the classical algorithm. Some examples of the prohibited vari- ants are Reordered conjugate-gradient methods [9–13] and Pipelined conjugate-gradient methods [14, 15].

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 that bypass the objectives of the benchmark, are generally not permitted.

3.5 Core Kernels in HPCG

The HPCG benchmark for solving $A \cdot x = b$ depends on several core computational kernels for solving sparse linear systems via the "Preconditioned Conjugate Gradient" (PCG) method. These kernels include: the "Dot Product", "WAXPBY" (weighted addition of two vectors), the "Sparse Matrix-Vector Multiplication" (SpMV), "Symmetric Gauss-Seidel" (SymGS) method, and "Multigrid V-cycle" (MG). The efficient computation of this benchmark is significantly de- pendent on the performance of SpMV and SymGS. Optimizing these kernels is challenging due to the inherent data dependencies that introduce complexity in parallelism. The multigrid V-cycle preconditioner, as shown in Fig. 3 is a key operation of HPCG, which accelerates convergence by addressing errors across different grid levels, using a combination of pre and post-smoothing steps to improve the efficiency of the solution process.

3.6 Evolution of HPCG Reference Implementations

Table 3 shows the version history and the major changes in the reference implementations, which lists the details of the updates from its initial release to sequence updates released over the years. The information about the different optimized HPCG Sofware releases is also available in [16].

4. Methodology for Paper Selection

Table 3. HPCG Versions History

Date	Version	Description
19-Nov-13	v1.0	• Initially released and early updates in versions from v0.1 - v1.0 • Added ref code to indicate optimization for dot-product, SPMV, SYMGS, and WAXPBY • Implemented a simple additive Schwarz pre-conditioner and symmetric Gauss-Seidel sweep.
29-Jan-2014	v2.0	• The preconditioner was replaced with a multi-grid method using three (3) levels of coarsening. • The approach allowed better cache usage and was a major enhancement over the previous version.
31-Jan-2014	v2.1	• First implementation of multigrid preconditioner for the conjugate gradient method. • Number of coarse grid levels set to 3 for production benchmark. • Injection is used as the grid transfer operator and symGS as pre and post smoother. • Not implemented a true multigrid preconditioner as the bottom solver for the coarsest grid was not added. • Introduced a Vector struct for easier implementation of kernels on GPUs.
27-May-2014	v2.2	• Reduced the penalty overhead cost by a factor of 10. • Fixed bugs of v2.1.
02-Jun-2014	v2.3	• Fixed the Flop count errors that appeared in v2.2.
03-Jun-2014	v2.4	• Based on a conjugate gradient solver having a three-level hierarchical multi-grid method with Gauss-Seidel relaxation. • Fixed bugs of the previous version.
11-Nov-2015	v3.0	• Added GenerateProblem as a timed portion of the benchmark. • Added memory usage and bandwidth reporting. • Introduced Quick Path option to easily get results. • Added a command-line option (–rt=) to specify runtime. • Added support to make build easy on Windows systems. • Change the way of computation of residual variance. • Modified the order of array allocation in the reference code to attain a better perfor- mance. • Set the minimum iteration count for the optimized run to match the reference run.
28-Mar-2019	v3.1	• Transformed the output format for results from YAML to a basic line-oriented key-value format with nested naming. • Implemented more efficient search algorithms for optimal 3D grid partitioning. • Resolved any unresolved bugs reported on the reference code's GitHub project and integrated fixes into the source code.

For our study, we selected those papers that are mainly focused on the optimization of the HPCG benchmark introduced by Dongarra and Heroux in 2013. Our selection prioritizes the papers published within the last decade (2013-2024). The HPCG benchmark encompasses complex computational kernels such as SpMV, SymGS, and the multigrid method, and we deliberately exclude the papers that solely discuss the optimization of these kernels because the extensive literature is available on these individual techniques.

This decision stems from limiting the number of papers and the emphasis of the discussion on the optimization of the HPCG benchmark, considering the constraint insisted by the original authors and already discussed in Section 2.4 of this paper. Our selection criteria prioritize papers that discuss data formats tailored for HPCG optimization, parallelization techniques within HPCG constraints, architecture-specific optimizations, and approaches to overcome HPCG optimization challenges. These papers consider the constraints set by the HPCG benchmark authors, which include the restriction on modifying the core algorithm, data structures, and communication patterns. Our paper discusses the complexity involved in the optimization, considering the architectural limitations and memory data access patterns. Furthermore, we also evaluate these papers and their proposed optimizations of HPCG based on the scalability and portability across different hardware and software. To the best of our knowledge, this is the first work, and there is no comprehensive survey that has summarized the work on HPCG optimization over the past decade. Our study aims to fill this gap by providing valuable information to researchers, consolidating the previous work information, and identifying future research directions in this field. The primary objective of this study is to summarize the challenges in the optimization of the HPCG benchmark and the progress made in the last decades by the researchers to provide a clear and systematic overview.

The list of papers that have been reviewed for this study is shown in Table 4. The table columns include the Paper reference, the number of Citations, the Dependency Order of the refereed papers, Related, Category, Target Architecture and Techniques/Methodology. This table enables the readers to understand the scope and focus of this survey.

Table 4. List of the selected papers for review

No.	Paper	No. of Citation	Dependency Order	Related	Category	Target-Architecture	Techniques/ Methodology
1	Dongarra & Heroux, 2013 [6]	65	0	A	Reference	General	-
2	Heroux et al., 2013 [2]	29	1	A	Reference	General	-
3	Park et al., 2014 [17]	67	1, 2	B	Optimization	Intel Architecture (IA)	SELLPACK, P2P, ABMC, Task Scheduling
4	Zhang et al., 2014 [18]	36	1, 2, 10, 11	C	Optimization	Tianhe-2 CPU-only nodes	SELLPACK, RB relaxation, forward-backward sweeps fusion, reformulation of the CG
5	Phillips & Fatica, 2014 [19]	33	1, 2, 10	D	Optimization	NVIDIA GPU	ELLPACK, customized CUDA based kernels, overlapping communication and computation in SpMV, multi-color reordering
6	Dahnken et al., 2014 [20]	30	1, 2	E	Optimization	IA	Identified the hotspot and performance issues using Intel VTune Amplifier & Inspector XE, and explored thread synchronization and locking issues
7	Y. Liu et al., 2014 [21]	20	1, 2, 4, 10, 11	C	Optimization	Tianhe-2 (hybrid CPU-MIC)	SELLPACK for MICs and ELLPACK for CPUs, inner-outer sub domain partitioning, asynchronous data transfer, RB relaxation, forward-backward sweeps fusion
8	Chen et al., 2014 [22]	12	1, 2	F	Optimization	Tianhe-2 CPU-only nodes	ELLPACK, 4 & 8-colors multi-coloring reordering
9	F. Liu et al., 2014 [23]	5	1, 2, 4, 9	C	Optimization	Tianhe-2 (hybrid CPU-MIC)	Intra-node communication, computation-communication overlapping, reformulated CG (Asynchronous & Pipelined), kernels fusion (WAXPBY & Dot-product), global & P2P communication optimization
10	Park and Smelyanskiy, 2014 [24]	5	1, 2	B	Optimization	IA	GS smoother optimization

No.	Paper	No. of Citation	Dependency Order	Related	Category	Target-Architecture	Techniques/ Methodology
11	Kumahata and Minami, 2014 [25]	4	1, 2	G	Optimization	K Computer	Coloring and blocking, loop direction reversing, data reorgani- zation for continuous memory allocation
12	Dongarra et al., 2014 [3]	7	1, 2	A	Discussion	General	-
13	Kumahata et al., 2014 [26]	3	1, 2	G	Optimization	K Computer	Continuous memory allocation, BMC, loop unrolling, parameters tuning
14	Marjanovic et al., 2015 [27]	55	1, 2, 12	H	Evaluation	Intel Sandy-Bridge, AMD Opteron, Intel Xeon X5560 chips (Nehalem)	Performance evaluation model
15	Komatsu et al., 2015 [28]	20	2	I	Optimization	SX-ACE super-computer	ELLPACK+coloring, MC, hyperplane with selective caching, problem size tuning
16	Slettebak, 2015 [29]	0	1, 2, 3	J	Evaluation	IA, Maya com-puting cluster	Performance analysis
17	Park et al., 2015 [30]	1	1, 2, 3	B	Optimization	IA	Updated their previous work [17] for HPCGv3.0
18	Kumahata et al., 2016 [31]	31	1, 2, 3, 4, 5, 12	G	Optimization	K computer	Memory layout reorganization, realigning loop directions, BMC
19	Dongarra et al., 2016b [5]	313	1	A	Reference	General	-
20	Dongarra et al., 2016a [4]	186	1, 2	A	Reference	General	-
21	Y. Liu et al., 2016 [32]	35	1, 2, 3, 4, 5, 7, 11	C	Optimization	Hybrid CPU-MIC	SELLPACK/ELLPACK, BMC, inner-outer domain partitioning
22	Park et al., 2016 [33]	18	1, 3, 5, 7, 8, 9, 11, 14	B	Optimization	IA	SELLPACK, P2P, BMC, fused backward GS sweep with SpMV, task scheduling
23	Marjanović et al., 2016 [34]	15	1, 14	K	Evaluation	Intel Haswell, IBM POWER8, ARM, NVIDIA GPU and Intel Xeon Phi, and NEC SX-ACE	Aggregate metric approach, in- terconnection network impact, problem size selection in bench-marking practices
24	Phillips et al., 2016 [35]	7	1, 2, 10	D	Optimization	GPU-focused: NVIDIA CUDA, Tesla K20X, K40 GPUs, Cray XK7, XC30	Graph coloring, matrix reordering, communication-computation overlap, hybrid CPU-GPU approach
25	Bookey, 2016 [36]	0	1, 2	L	Optimization	CPU and GPU	Use leveling and coloring techniques with Kokkos parallel dispatch and data abstraction
26	Pan & Wang, 2016 [37]		2, 12	M	Optimization	CPU+MIC het-erogeneous ar-chitecture	Vectorization, CPU-MIC task distribution, architectural specific optimizations, heat dissipation optimization to maintain the system stability
27	Vermij et al., 2017 [38]	12	3, 14, 17	N	Optimization	Near-data pro-cessing (NDP) based on IBM POWER8	BMC, data restructuring, threading overhead reduction, inter-NDP communication optimization, prefetching
28	Liao et al., 2017 [39]	0	3, 4, 7, 21, 22	O	Optimization	Sunway many-core processor	Hierarchical grid (HG), collaborative computing MPE-CPE, SIMD optimization, sub-grid distribution, task partitioning, DMA data prefetch, register-based communication, node-level parallelism
29	Ao et al., 2018 [40]	36	3, 5, 7, 18, 20, 21	O	Optimization	Sunway TaihuLight	ELLPACK, BMC, locality-aware layout transformation, concurrent gather collective, fine-grain task overlapping, task decomposition, on-chip register communication, communication overhead reduction
30	Ruiz et al., 2018 [41]	6	1, 2, 3, 4, 5, 14, 18, 26	P	Optimization	ARMv8 (Cavium ThunderX2)	Multi-color reordering, multi-block reordering, dynamic blocksizing, thread binding, arm specific optimizations
31	Chester et al., 2018 [42]	3	1, 2, 20	Q	Evaluation	IA	MPI communication evaluation using Intel Trace Analyzer and Collector
32	Ruiz et al., 2019 [43]	8	1, 2, 3, 4, 5, 14, 18, 27	P	Optimization	ARM (Cavium ThunderX2) and x86 (Intel Skylake Platinum 8176)	Parallelization strategies, multi-color and multi-block reordering, arm architecture optimization, dynamic slicing, arm vs x86 performance comparison

No.	Paper	No. of Citation	Dependency Order	Related	Category	Target-Architecture	Techniques/ Methodology
33	Jäger et al., 2019 [44]	1	1, 2, 20	S	Evaluation	Intel x86 processor (Intel Ivy Bridge)	Gnu vs intel compiler comparison for energy efficiency analysis, compiler optimization impact, energy-aware scheduling
34	Rho et al., 2019 [45]	0	1	T	Evaluation	Intel and AMD systems	Benchmark automation and evaluation
35	Alappat et al., 2020 [46]	32	14	U	Evaluation	Intel Broadwell EP and Cascade Lake SP	Use of roofline model for evaluation
36	Zeni et al., 2021 [47]	21	1, 2, 3, 4, 5, 14, 18, 27, 30	V	Optimization	FPGA Xilinx Alveo U280	Custom load balancer, multi-FPGA configuration, PCIe communication reduction, kernel-to-kernel streaming, FPGA-specific optimizations, inter-FPGA communication minimization, numerical precision tuning, Berkeley Roofline model
37	Zhu et al., 2021 [48]	27	1, 2, 5, 8, 21, 18, 22, 29	O	Optimization	Sunway supercomputer	Simplified ELLPACK, two-level blocking scheme, kernel fusion
38	Steiger, 2022 [49]	2	1, 20, 36	P	Optimization	Xilinx Alveo U280 FPGA With HBM memory	Modified CSR with diagonal indices, kernel-to-kernel streaming, DaCe framework
39	Scolari & Yzelman, 2023 [50]	2	2, 18, 19, 21, 24, 30	Q	Optimization	x86 and ARM systems	RB coloring, Graph BLAS operations, BSP model with all-to-all communication
40	Yang et al., 2023 [51]	2	1, 3, 4, 18, 21, 22, 37	R	Optimization	Multi-core systems (ARM and x86 architectures)	Kernel fusion, CSR format, BMC reordering, asynchronous scheduling, adaptive block size, dynamic task allocation, dependency sparsification, asynchronous communication
41	Gómez et al., 2023 [52]	2	2, 15, 21, 22, 24, 32	S	Optimization	NEC VE and RISC-VV	ELLPACK, matrix decomposition, level scheduling, vector architectures optimization, loop unrolling, low locality data management
42	Yuan et al., 2024 [53]	0	4, 18, 30, 37, 40	R	Optimization	ARM and x86 Architectures	Kernel fusion, CSR, asynchronous BMC scheduling, point-to-point synchronization, dependency sparsification, monolithic MG kernel

Paper: This column lists the publication by authors, year, and citation number to provide a quick reference of the paper.

Citation: This column lists the number of citations each paper received until August 2024. Citations serve as an indicator of the impact and influence of a paper within the scientific community. Papers with a high number of citations in the early years following their publication typically indicate highly influential research that has an impact on the related field. Papers published in 2014 that have received numerous citations highlight their importance in the optimization of HPCG optimization.

Dependency Order: This column shows the interconnectedness of the papers, which is based on the references cited within these selected papers. The dependency order helps to map out the scholarly network, indicating how follow-up research works and builds upon earlier ones.

Related: This column identifies closely related papers, such as those authored by the same research group or those that are closely related in content or methodology. Papers with the same letter in the related column are typically connected through similar research objectives or approaches.

Category: The categories are defined as follows:

- *Reference*: papers that relate to the base/reference implementation of HPCG.
- *Evaluation*: research papers that have evaluated the HPCG benchmark using different techniques and on different architectures.
- *Discussion*: paper related to discussing the impact or progress of HPCG in the HPC community.
- *Optimization*: papers describing optimization applied for HPCG.

Target Architecture: This column presents details regarding the hardware platforms or system architectures targeted by each paper.

Techniques/Methodology: The third column highlights what specific optimization techniques and methodologies are employed by each paper in its efforts to evaluate or improve the performance of the HPCG benchmark on the target architecture.

Our focus in this survey paper is on the optimization approaches described in these papers, which we summarized in Section 6 for the reader.

5. Data Formats and Storage Strategies

5.1 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 Coordinate (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), and ELLPACK (ELL) [54, 55]. HPCG reference implementation uses the CSR format. 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. [55] 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 [55].

Sparse matrix operations in other applications and in HPCG are affected by these formats. Using the suitable format helps 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.

5.2 Novel Data Structures for HPCG

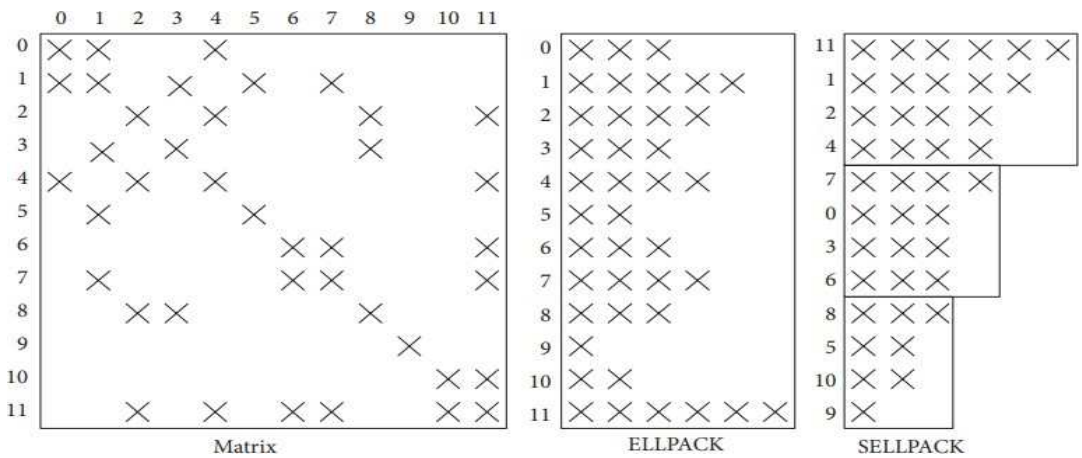

Fig. 4. ELLPACK and sliced version of ELLPACK (SELLPACK) data format representation, adapted from [56]

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 the 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. This format is highly vectorizable but results in considerable memory overhead as compared to CSR, especially for matrices with rows of varying lengths. SELLPACK extends this by splitting the matrix into slices with rows of similar lengths, minimizing padding and reducing memory usage, which boosts parallel performance. SELLPACK strikes a balance between memory efficiency and computational performance, making it well-suited for vector architectures, particularly in the HPCG benchmark. Fig. 4 illustrates the simple representation of ELLPACK and SELLPACK data format of the matrix, and Table 5 summarizes the papers reporting about the use of these data formats and provides a list of papers that investigate the application of these data formats for the optimization of HPCG.

Table 5. Data Formats reported in the literature of the HPCG optimization

Data Formats	Reference	Architecture
ELLPACK	[19, 21–23, 28, 32, 35, 40, 48, 52]	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	[17, 18, 21, 32, 33]	Titan (Cray XK7) and Piz Daint (Cray XC30) supercomputers, Hybrid CPU-MIC
JAD	[28]	SX-ACE supercomputer
DIA	[22]	SX-ACE supercomputer, Tianhe-2 Supercomputer
CSR/ Modified CSR	[28, 31, 35–39, 43, 47, 49–51, 53]	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

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 enhanced parallelization and reduced memory access latencies. Overall, the use of these formats, together with parallelization techniques, brings significant memory efficiency and improves performance of HPCG.

6. Parallelization Optimization

6.1 Parallelization Techniques

The parallelization approaches are of most importance, which are generally difficult to optimize because of sparse matrix and its dependencies. Coloring appears to be most effective in respect of parallelizing the main kernels of HPCG, SpMV, and SymGS smoother. The coloring technique assists 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.

6.1.1 Coloring

The basic concept of the coloring technique is the distance-1 or distance-2 coloring discussed in the literature [57, 58]. 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 Fig. 5. 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. Parallel processing is performed so that the red colors 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.

Multi-Coloring with 4 or 8 Colors: The 4 or 8-color 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 [22] 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 increases the overhead and impacts the convergence as the problem size increases.

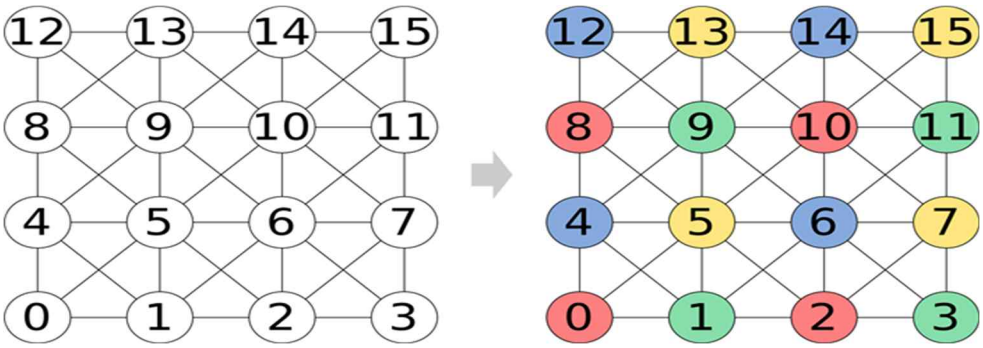

Fig. 5. Multi-coloring, reproduced from [59]

Block Multi-Coloring (BMC): In the block multi-coloring technique, as shown in Fig. 6, 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.

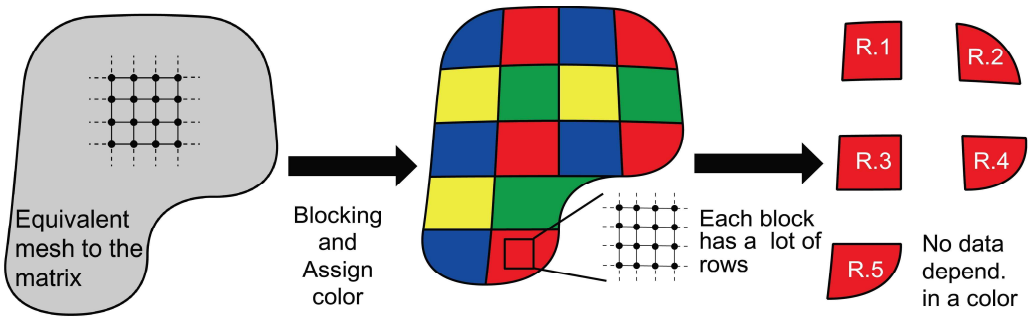

Fig. 6. Block Multi-Coloring, adapted from [31]

Algebraic Block Multi-Coloring (ABMC): ABMC [60] is a parallel processing technique for solving large sparse linear systems of equations. ABMC, as shown in Fig. 7 involves two main steps, blocking and coloring algorithmically proposed in [61]. 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 needed 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 [17].

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.

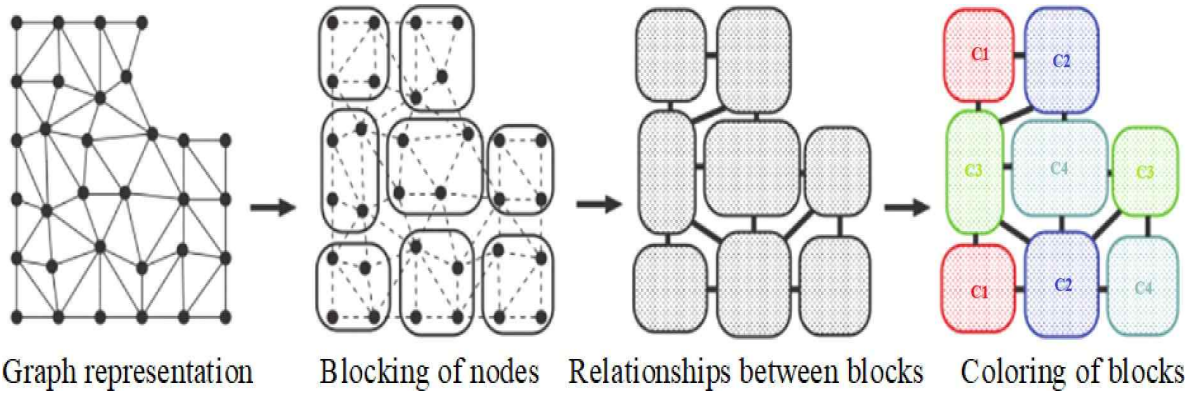

Fig. 7. Algebraic block multi-coloring, adapted from [60]

6.1.2 Advanced Parallelization and Scheduling Techniques

Instead of relying on coloring, researchers also use various other techniques, such as creating task dependencies and scheduling the tasks in a different manner. Some of these advanced techniques used are:

Multi-Level Task Dependency Graph (TDG): TDG creates a graph where nodes correspond to the elements, and the edges represent the dependencies of the elements. It is further organized into levels, where each level contains independent tasks that can be executed in parallel. This scheme respects data dependency and changes the order in which data is required to be processed. However, this negatively affects the temporal locality. For further details, refer to [59].

Hyperplane (2D) Parallelization: Hyperplane accesses elements in a diagonal manner, thus avoiding data dependencies and therefore achieving faster convergence at higher computational efficiency. This technique was used on architectures like the SX-ACE by [28] and reported that such parallelism leads to performance gains.

Hierarchical Grid (HG): The HG technique recursively divides the problem at different resolution levels in cascaded sub-grids. It uses a sub-domain partitioning strategy, and the local computations within a grid reduce memory access latency and improve cache usage, allowing better performance and parallel efficiency. Details on this method are found in [39].

Two-Level Blocking Scheme: In two-level blocking approach, the sparse matrix is first partitioned into m large layers/blocks that are processed sequentially to preserve the intrinsic inter-dependencies. At the second level, each layer is a directed graph, partitioned and colored such that smaller blocks of the same color can be processed independently. This scheme maintains the best parallelism and convergence. Layers are processed sequentially, but sub-blocks can be executed in parallel, optimizing memory access and reducing computation time. This approach was applied by [48] for the optimization of HPCG on a Sunway supercomputer.

Block Multi-Color (BMC) Scheduling: The BMC scheduling is a parallelization technique that aug- mented the block multi-color scheme to group the grid points into blocks, colored them to minimize dependencies, and also integrated block partitioning, asynchronous execution, dependency man- agement, sparsification synchronization, dynamic scheduling, and adaptive block sizing techniques. The BMC scheduling scheme is described for the first time in [51, 53].

7. 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 offioading 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 in the selected papers. 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 selected papers by architecture and year-wise and also mention the reported evaluation results. Since the introduction of HPCG in 2013, several optimizations have been done regarding different architectures.

7.1 CPU-Based Systems

7.1.1 Intel Architecture - IA

In 2015 [30] presented the updates on their intel-based work [17], 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) [62] and the open source SpMP library [63] 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 [33] expanded upon the previous work [17] 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, and achieved 21 Gflops on one Xeon Phi coprocessor. 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. [37] optimized HPCG for a CPU+MIC platform, with techniques like offioad programming, vector- ization, and parameter tunings. The author applied different optimization techniques, including the integration of an offioad programming mode to use the Intel Xeon Phi coprocessors for compu- tationally 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. Their effort achieved a speedup of 65% on

small matrix sizes and about 30% speedup on large matrix sizes. 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 [53] optimized implementation primarily developed for ARM-based systems but also tested on an Intel Xeon system, and demonstrated a 1.62x performance improvement in the intel optimized HPCG. 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.

7.1.2 Arm-Based Architectures

In 2019 [41, 43] presented the profiling of HPCG on an Arm-based platform Cavium ThunderX2. The optimized version improved the performance up to 9.5x compared to the referenced version of HPCG. 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 [43] are identical to those that have already been discussed in the technical report [41]. The strategies outlined in this paper were specifically developed for the ARMv8.1 Cavium ThunderX2 processor, which features a shared memory architecture.

In 2023 [50] implemented a new HPCG based on the ALP/GraphBLAS [64] 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 [65], which is a C++ variant of GraphBLAS called ALP. The authors proposed a new implementation of HPCG based on ALP/GraphBLAS [66] 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 [66] 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 [51] focused on the theoretical development of the optimization techniques, and another pa- per [53] 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 [67], 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. [53] 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. The optimized HPCG implementation achieved performance improvement of 2.12x to 2.54x over ARM-specific implementations.

7.1.3 K Computer

In 2016 [31] 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 19.6x times performance improvement compared to the original code, achieving 0.461 Pflops, a significant performance on the K Computers, and secured second position in November 2014 HPCG results with 4.4% of ratio to HPL performance.

7.1.4 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. These conventional approaches attained performance improvement of 1.54x and 5.52x, respectively, compared to the reference implementation of HPCG. 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. They achieved 3.54 Gflops on a single Sunway core group and scaled to 192 Tflops on 81,920 MPI processes, with 70% node-level parallel efficiency using these optimizations.

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. The optimized HPCG sustains 77% of theoretical memory bandwidth and scales over 10 million cores with an aggregated performance of 480.8 Tflops with a weak scaling of 87.3%.

7.1.5 OceanLight Sunway Supercomputer

In 2021 [48] 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 [68] presented effective optimization strategies for sparse linear solvers on modern heterogeneous supercomputer architectures.

7.1.6 NEC SX-ACE Vector Supercomputer

In 2015 [28] explored various optimization techniques for HPCG benchmark on the SX-ACE super-computer [69]. 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 [70], and ELLPACK [71] 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 [60] and hyperplane [72, 73] 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. [28] 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 [52] presented an optimized HPCG implementation for long-vector architectures in order to achieve a performance improvement of 7.9% on NEC VE and a 14.9% improvement on high-end RISC- V accelerators, mainly through kernel optimizations on enhancing memory hierarchy usage. They achieved 122.71 Gflops on the NEC VE platform, a 1.6% increase from previous implementations. 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 [74] to long-vector architectures.

7.1.7 Near-Data Processing (NDP) Architecture

In 2017 [38] discussed the use of IBM Power8 near-data processors (NDPs) [75] in the optimization of HPCG and Graph500 [76] 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 [38], and the distributed Graph500 details can be found in [77]. 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. Their approach led to a 42% performance increase when running a 643 problem on a single NDP, emphasizing the necessity for highly parallel code on the studied architecture. 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. Their strategy enhanced the performance by 8% for larger problem sizes. 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. With these optimizations, in both performance and power efficiency, they achieved 2x to 2.5x improvement, respectively, compared to CPU and GPU implementations for the HPCG benchmark. The optimized HPCG benchmark achieved the performance of 49 Gflops with eight NDPs, demonstrating strong scalability, and achieved 3.5x and 5x speedup for the HPCG benchmark on the IBM Power8 system compared to traditional CPU and GPU implementations.

7.2 GPU-Based Systems

7.2.1 GPU-Accelerated Systems

In 2014 [19, 78] 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 [79, 80] and incorporated improvements proposed by [81]. They primarily focused on parallelizing the SymGS. They used cuSPARSE library [82] 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) [83] and the Cray XC30 system at the Swiss National Supercomputing Centre

(CSCS) [84]. The Cray XK7 (Titan) has an AMD Opteron processor and a Gemini interconnect that has a 3D torus topology [85]. The Cray XC30 (Piz Daint) features an Intel Xeon processor and an Aries interconnect that has a dragonfly topology [86]. 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 [35] 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.

7.3 Hybrid Architectures

7.3.1 Tianhe-2 Supercomputer

In 2014 [17] 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 [87, 88] including HPCG. Achieving high performance of symGS smoother is particularly challenging due to its limitation in fine-grain parallelism [89] 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. [18] focused on improving only the CPU-based system and got a performance of 79.83 Tflops 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 [90], 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 [91]. Compared to prior work [17] on 12-core Intel Xeon processors, this optimized HPCG achieved both higher single-CPU performance and superior large-scale performance on Tianhe-2. [21] was an extension of the previous study [18] for hybrid CPU-MIC based architecture on the Tianhe-2 platform and achieved a performance of 50.2 Gflops on a single computing node. 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. [22] highlighted the importance of multi-coloring techniques for Gauss-Seidel with SIMD-friendly sparse matrix formats. Thus, they were able to achieve a computing performance of 80,151 Gflops on 8,192 CPU-only nodes. [23] 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 2.4.2. They achieved a performance of 10.51 Tflops on a system of 256 nodes, with a weak scaling efficiency of over 90%. Although all these papers achieved significant improvements in performance

compared to the reference implementation, they varied in their emphasis: [17] and [21] selected the use of heterogeneous structures, [18] and [22] optimized for CPU-only systems and [23] 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 [32] has optimized HPCG on Tianhe-2 using CPU + MIC heterogeneous architecture. Based on the previous work [18, 21], the paper [32] 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 & backward fused algorithms and block multicolor parallelization techniques for the SymGS kernel and, as a result, achieved the performance of 623 Tflops. Obtained 81.2% parallel efficiency on a full system with 3.12 million heterogeneous cores.

7.4 Other Architectures and Environments

7.4.1 FPGA-Based System

In 2021, the first known approach for reconfigurable hardware implementation of HPCG was presented by [47]. 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. They achieved a computational performance of 108.3 Gflops using a single Xilinx Alveo U280 FPGA and a performance of 346.5 Gflops using a configuration of multi-FPGA 4x Xilinx Alveo U280 FPGAs. 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 [49] 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 [47] of HPCG using the DaCe framework, but the implementation was significantly slower than existing version [47]. 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.

7.4.2 Kokkos-Based

In 2016 [36] was a thesis work of a student from the College of Saint Benedict and Saint John's University. It presented the development of KHPCG [92], which was the first try for creating a performance-portable version of the HPCG benchmark using KOKKOS library [93–95], 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 [96] 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.

7.5 HPCG Benchmark Implementation Variants

The reference HPCG benchmark implementation can be found in the project git repository [97] and on the official web [107]. Its latest release is 3.1, and we call it native HPCG implementation. Open source and some Vendor-optimized variants modified based on the older versions of the reference/native HPCG code listed in Table 6.

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 6 in much detail. To delve deeper into the specifics of each variant, it is recommended to refer to the respective research works.

Table 6. HPCG Benchmark Variants and References

Sr#	Variants	References	Reference HPCG Version Used
1	Native	[97]	N/A
CPU-Based			
2	IBM	[98]	HPCGv2.4
3	Intel CPU	[99]	HPCGv3.0
4	ARM	[100, 101]	HPCGv3.0
5	Sunway	[68]	HPCGv3.1
6	ALP/GraphBLAS	[66]	HPCGv3.1
7	VE native	[74]	HPCGv3.1
GPU-Based			
8	Intel GPU	[102]	HPCGv3.1
9	NVIDIA	[103, 104]	HPCGv3.1
10	AMD ROCm	[105]	HPCGv3.1
Others			
11	FPGA Xilinx	[106]	HPCGv3.1
12	KHPCG	[92]	HPCGv2.4

- ARM Optimized Variant: [41, 43, 59]
- Sunway Optimized Variant: [48]
- ALP/GraphBlas Optimized Variant: [50]
- FPGA Xilinx Optimized Variant: [47]
- VE Native Optimized Variant: [52]
- KHPCG Variant: The Kokkos-based HPCG variant [36]

However, for the IBM-optimized variant, no particular research paper has been written. Some details are presented in [98], 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 [108]. 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 [109, 110] 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 offioading, 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.

7.6 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.

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 more accurately represent the computational characteristics of modern scientific applications. In [3], they examined the initial 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, [24–26] are cited by various research papers, again 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 [111] 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.

8. Discussion

With the advent of exascale computing, the significance of the HPCG benchmark is becoming more pertinent by addressing the limitations of its counterpart HPL. Exascale systems will be capable of performing 1018 operations per second and require a complete benchmarking tool that will test not only computation power, but the processors memory bandwidth and how efficiently the different parts of the system can communicate with each other. With a focus on memory-bound operations and real-world workloads, HPCG is in a good position to meet for the current HPC benchmark requirements. However, HPCG needs to evolve to further represent new emerging challenges, especially those that penetrate from hardware heterogeneity and from increasing complexity within HPC applications.

Recent efforts have been made by a wide range of groups of researchers for optimizing HPCG with a significant improvement on various architectures. One of the most promising modified variants of the data formats contributing to the optimization of the benchmark emerged as ELL- PACK/SELLPACK (Sliced ELLPACK), particularly for vectorization supportable architectures along with the block coloring techniques for parallelism, dynamic scheduling for load balancing and architectural tuning, etc. It enables better vectorization and efficient use of cache memory by ensuring that the data is stored contiguously in memory.

8.1 Challenges

SymGS, the core kernel of HPCG, is limited by memory bandwidth rather than computing power. SymGS is inherently sequential, which creates significant challenges for the optimization of HPCG. Its data dependencies make it difficult to parallelize effectively, which becomes a major bottleneck when trying to utilize the massive parallelism available in exascale systems. The communication between nodes also becomes a critical bottleneck when scaling across heterogeneous architectures with both CPUs and GPUs. This is particularly problematic for operations like SymGS, which require frequent global synchronization. Optimization efforts to reduce communication delays and improve data locality are indeed two of the most crucial areas for optimizing HPCG performance. It is important to note the optimizations must be balanced against maintaining the core characteristics of the benchmark because HPCG was designed to represent a broad set of applications, and any optimizations should not fundamentally alter its behavior or make it less representative of real-world problems.

8.2 Future Perspectives

Problem Size: Investigate the impact of problem size as the local sub-domain problem will significantly impact the performance. Due to changes in dimensions along the x, y, and z axes of the computational grid, different architectures respond differently. This greatly impacts memory access patterns and, thereby, the communication overhead. An understanding of these effects enables a more efficient optimization of performance on various platforms. Develop autotuning approaches that can automatically evaluate problem sizes and make necessary adjustments, particularly for grids with unequal dimensions. One can also develop machine learning (ML) model to predict optimal configurations for the target architecture, tuning memory access patterns and parallelization strategies to pick the best configuration dynamically. Investigation of AI-

augmented auto-tuning in benchmarks on various architectures could considerably enhance their performance using optimized parameters that have traditionally been hand-tuned by experts.

Communication Strategies: Investigate the sophisticated communication techniques, the communication avoiding algorithms and asynchronous communication are going to increase in importance as communication overheads continue to constrain performance in large-scale systems. These will help reduce idle time spent waiting for data transfers between nodes, which would otherwise be improved. Implement topology-aware communication patterns that match the underlying network structure or investigate methods that could potentially reduce global communication requirements.

Task based model: In addition to OpenMP, a conventional parallelism model, there is vast flexibility in investigating task-based parallelism models exemplified by OmpSs-2 [112]. Additional optimization can be utilized to augment advanced systems by enabling out-of-order execution, dynamic scheduling, and load balancing.

Energy Efficiency Aware optimization: Exascale systems are well-known for their high power consumption, and any future benchmarking efforts must ensure that computational performance is balanced with effective power usage. Energy consumption becomes the more important aspect in exascale systems. Indeed, supercomputers of the future must strike a balance between performance and power consumption. Energy efficiency aware optimizations that reduce computations and data transfers, could result in substantial energy savings. Furthermore, systems will clearly depend on a deep understanding of the energy consumption profile of HPCG, which necessitates the integration of energy-aware tuning, as well as the elaboration of strategies toward its reduction without affecting performance.

Emerging Hardware Trends: As we move into the era of exascale computing and heterogeneous architectures, the development of heterogeneous architectures which combines traditional CPUs with GPUs, FPGAs, and other specific accelerators, driving the future of HPC systems. Different architectures will mean that the HPCG benchmark will have to evolve as well in order to fully exploit these architectures. One primary focus is the optimization of multi-node systems with heterogeneous processors, where data transfer between CPUs and GPUs constitutes a performance bottleneck. Optimizing data placement to reduce transfers and utilizing shared and distributed memory architectures will be crucial. Furthermore, the systems will involve billions of cores with different memory pools. In such environments, when memory access and interconnect bandwidth become bottlenecks, hierarchical memory architectures like high-bandwidth memory (HBM) and non-volatile memory (NVM) demand optimized strategies for cache re-usage, data locality and efficient memory access patterns. This will direct the research work to the re-conceptualizing of data layouts and the possible exploitation of new sparse matrix formats, which handle large-scale data efficiently with reduced memory traffic and maximize cache utilization.

Load Balance or Task Schedule: Load balancing and dynamic task scheduling are also important to scale HPCG across large heterogeneous systems. Optimizing the workload between the different processors at runtime based on real-time performance feedback can help with task distribution and reduce idle time, hence improving overall system performance.

Code Portability: Benchmarks must possess a high degree of portability to remain valuable in light of the growing diversity in hardware architecture. Frameworks like Kokkos could enhance performance portability across heterogeneous systems, addressing the challenge of porting GPU- optimized code to other architectures. These performance portability frameworks are important for HPCG portability in a world where hardware is evolving rapidly.

Algorithmic Improvements: Developing better preconditioner and multigrid methods. Simple geo- metric multigrid methods can be significantly improved with algebraic multigrid methods. These will bring even better convergence rates for grids and complex problem domains.

Profiling and Tuning Tools: The development and utilization of such tools will provide insight into how the benchmark utilizes the underlying hardware components, including memory band- width, cache utilization, and interconnect performance. By obtaining insight into bottlenecks and inefficiencies, developers can more effectively optimize their systems for particular workloads.

9. Conclusion

The paper presents a survey of the optimization techniques for the HPCG benchmark, which has evolved as a supercomputer performance metric over the last decade. Numerous techniques have been applied to optimize HPCG for different architectures such as CPU, GPU, MIC, and FPGA, etc. This study discussed these techniques, such as the data formats, parallelization approaches, architecture-specific tunings, and also the challenges associated with the optimization of the benchmark. This paper summarizes the optimization work performed by the researcher over the past decade. Further, the paper recommends more areas of research to explore and encourages the HPC community to continue refining the benchmark for its suitability in benchmarking future exascale systems.

The study also highlights the limitations of the traditional HPL benchmark, and HPCG addresses these limitations by focusing on stressing the system for memory bandwidth, computational power, and the communication overhead using sparse matrix operations and reflecting as the suitable candidate for real-world application benchmarking as compared to its counterpart, which solely focuses on the measure of computational peak performance. This study lists down most of the optimized implementations of HPCG best known to us and also highlights the different optimization approaches across various architectures but does not cover the impact of communication and net- work topologies on performance in multi-node cluster environments. While focusing on data format and optimization of parallelism, we had not approached most of the inter-node communication strategies and acknowledged the need for further research in this area.

10. References

[1] Jack J Dongarra. The linpack benchmark: An explanation. In International Conference on Supercomputing, pages 456–474. Springer, 1987.

[2] Jack Dongarra, Piotr Luszczek, and M Heroux. HPCG technical specification. Sandia National Laboratories, Sandia Report SAND2013-8752, 2013.

[3] J Dongarra, Piotr Luszczek, and M Heroux. HPCG: one year later. ISC14 Top500 BoF, 818, 2014.

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

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

[6] Michael Allen Heroux and Jack Dongarra. Toward a new metric for ranking high performance computing systems. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States); University of Tennessee, 2013.

[7] TOP500. HPCG - june 2024 | top500, June 2024. Accessed: August 6, 2024.

[8] Torsten Hoefler, Peter Gottschling, Andrew Lumsdaine, and Wolfgang Rehm. Optimizing a conjugate gradient solver with non-blocking collective operations. Parallel Computing, 33(9):624–633, 2007.

[9] Gerard Meurant. Multitasking the conjugate gradient method on the cray x-mp/48. Parallel Computing, 5(3):267–280, 1987.

[10] Anthony T Chronopoulos and Charles William Gear. S-step iterative methods for symmetric linear systems. Journal of Computational and Applied Mathematics, 25(2):153–168, 1989.

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

[12] Eduardo D'Azevedo, Victor Eijkhout, and Charles Romine. Lapack working note 56: Reducing communication costs in the conjugate gradient algorithm on distributed memory multiprocessors. Technical Report CS-93-185, University of Tennessee, Knoxville, USA, 1993.

[13] Jack Dongarra and Victor Eijkhout. Finite-choice algorithm optimization in conjugate gradients. Computer Science Technical Report UT-CS-03-502, University of Tennessee, Knoxville, 2003.

[14] Pieter Ghysels and Wim Vanroose. Hiding global synchronization latency in the preconditioned conjugate gradient algorithm. Parallel Computing, 40(7):224–238, 2014.

[15] Manasi Tiwari and Sathish Vadhiyar. Efficient executions of pipelined conjugate gradient method on heterogeneous architectures. arXiv preprint arXiv:2105.06176, 2021.

[16] HPCG Software Releases — hpcg-benchmark.org. https://www.hpcg-benchmark.org/software/, 2019. [Accessed 14-08-2024].

[17] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Xing Liu, Md Mosotofa Ali Patwary, Yutong Lu, and Pradeep 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, pages 945–955. IEEE, 2014.

[18] Xianyi Zhang, Chao Yang, Fangfang Liu, Yiqun Liu, and Yutong 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, pages 28–41. Springer, 2014.

[19] Everett Phillips and Massimiliano 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, pages 68–84. Springer, 2014.

[20] Alexander Supalov, Andrey Semin, Michael Klemm, and Christopher Dahnken. Optimizing HPC applications with intel cluster tools. Springer Nature, 2014.

[21] Yiqun Liu, Xianyi Zhang, Chao Yang, Fangfang Liu, and Yutong Lu. Accelerating HPCG on tianhe-2: a hybrid cpu-mic algorithm. In 2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS), pages 542–551. IEEE, 2014.

[22] Cheng Chen, Yunfei Du, Hao Jiang, Ke Zuo, and Canqun 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, pages 41–48. IEEE, 2014.

[23] Fangfang Liu, Chao Yang, Yiqun Liu, Xianyi Zhang, and Yutong 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, pages 13–18. IEEE, 2014.

[24] Jongsoo Park and Mikhail Smelyanskiy. Optimizing gauss–seidel smoother in hpcg. In ASCR HPCG Workshop, 2014.

[25] Kiyoshi Kumahata and Kazuo Minami. HPCG performance improvement on the k computer. In Presentation at Supercomputers Conference (SC'14), 2014.

[26] K Kumahata, K Minami, and N Maruyama. HPCG on the k computer. In ASCR HPCG Workshop, 2014.

[27] Vladimir Marjanović, José Gracia, and Colin W Glass. Performance modeling of the HPCG benchmark. In High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation: 5th International Workshop, PMBS 2014, New Orleans, LA, USA, November 16, 2014. Revised Selected Papers 5, pages 172–192. Springer, 2015.

[28] Kazuhiko Komatsu, Ryusuke Egawa, Yoko Isobe, Ryusei Ogata, Hiroyuki Takizawa, and Hiroaki 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, pages 1–2, 2015.

[29] Jack Slettebak. The HPCG benchmark for cluster computing, 2015.

[30] Jongsoo Park, Alexander Kleymenov, Mikhail Smelyanskiy, and Vadim 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].

[31] Kiyoshi Kumahata, Kazuo Minami, and Naoya Maruyama. High-performance conjugate gradient performance improvement on the k computer. The International Journal of High Performance Computing Applications, 30(1):55–70, 2016.

[32] Yiqun Liu, Chao Yang, Fangfang Liu, Xianyi Zhang, Yutong Lu, Yunfei Du, Canqun Yang, Min Xie, and Xiangke Liao. 623 tflop/s HPCG run on tianhe-2: Leveraging millions of hybrid cores. The International Journal of High Performance Computing Applications, 30(1):39–54, 2016.

[33] Jongsoo Park, Mikhail Smelyanskiy, Karthikeyan Vaidyanathan, Alexander Heinecke, Dhiraj D Kalamkar, Md Mosotofa Ali Patwary, Vadim Pirogov, Pradeep Dubey, Xing Liu, Carlos Rosales, et al. Optimizations in a high-performance conjugate gradient benchmark for ia-based multi-and many-core processors. The International Journal of High Performance Computing Applications, 30(1):11–27, 2016.

[34] Vladimir Marjanović, José Gracia, and Colin W Glass. Hpc benchmarking: Problem size matters. In 2016 7th International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS), pages 1–10. IEEE, 2016.

[35] Everett Phillips and Massimiliano Fatica. Performance analysis of the high-performance conjugate gradient benchmark on gpus. The International Journal of High Performance Computing Applications, 30(1):28–38, 2016.

[36] Zachary Bookey. Performance portable high performance conjugate gradients benchmark. All college thesis, College of Saint Benedict/Saint John's University, 2016. Accessed: 2024-08-14.

[37] Qingyi Pan and Xiaoying Wang. Performance evaluation and optimization of HPCG benchmark on cpu+ mic platform.

International Journal of Hybrid Information Technology, 9(11):239–254, 2016.

[38] Erik Vermij, Leandro Fiorin, Christoph Hagleitner, and Koen Bertels. Boosting the efficiency of HPCG and graph500 with near-data processing. In 2017 46th International Conference on Parallel Processing (ICPP), pages 31–40. IEEE, 2017.

[39] Chenzhi Liao, Junshi Chen, Wenting Han, Huanqi Cao, Zhichao Su, Wanwang Yin, and Hong 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, pages 361–368, 2017.

[40] Yulong Ao, Chao Yang, Fangfang Liu, Wanwang Yin, Lijuan Jiang, and Qiao Sun. Performance optimization of the HPCG benchmark on the sunway taihulight supercomputer. ACM Transactions on Architecture and Code Optimization (TACO), 15(1):1–20, 2018.

[41] Daniel Ruiz, Filippo Mantovani, Marc Casas, Jesús José Labarta Mancho, and Filippo Spiga. The HPCG benchmark: analysis, shared memory preliminary improvements and evaluation on an arm-based platform. , 2018. Technical Report.

[42] Dean G Chester, Steven A Wright, and Stephen A Jarvis. Understanding communication patterns in hpcg. Electronic Notes in Theoretical Computer Science, 340:55–65, 2018.

[43] Daniel Ruiz, Filippo Spiga, Marc Casas, Marta Garcia-Gasulla, and Filippo 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 & Simulation (HPCS), pages 225–232. IEEE, 2019.

[44] Armin Jäger, Jan-Patrick Lehr, and Christian Bischof. The influence of two modern compiler infrastructures on the energy consumption of the HPCG benchmark. SICS Software-Intensive Cyber-Physical Systems, 34:53–60, 2019.

[45] Seungwoo Rho, Ji Eun Choi, Geunchul Park, and Chan-Yeol Park. Performance analysis of various multi-and many- core systems centered on memory. In 2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS* W), pages 194–199. IEEE, 2019.

[46] Christie L Alappat, Johannes Hofmann, Georg Hager, Holger Fehske, Alan R Bishop, and Gerhard Wellein. Under- standing hpc benchmark performance on intel broadwell and cascade lake processors. In High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings 35, pages 412–433. Springer, 2020.

[47] Alberto Zeni, Kenneth O'Brien, Michaela Blott, and Marco D Santambrogio. Optimized implementation of the HPCG benchmark on reconfigurable hardware. In European Conference on Parallel Processing, pages 616–630. Springer, 2021.

[48] Qianchao Zhu, Hao Luo, Chao Yang, Mingshuo Ding, Wanwang Yin, and Xinhui 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, pages 1–13, 2021.

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

[50] Alberto Scolari and Albert-Jan Yzelman. Effective implementation of the high performance conjugate gradient benchmark on graphblas. In 2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 216–225. IEEE, 2023.

[51] Xiaojian Yang, Shengguo Li, Fan Yuan, Dezun Dong, Chun Huang, and Zheng Wang. Optimizing multi-grid computa- tion and parallelization on multi-cores. In Proceedings of the 37th International Conference on Supercomputing, pages 227–239, 2023.

[52] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. HPCG on long-vector architectures: Evaluation and optimization on nec sx-aurora and risc-v. Future Generation Computer Systems, 143:152–162, 2023.

[53] Fan Yuan, Xiaojian Yang, Shengguo Li, Dezun Dong, Chun Huang, and Zheng Wang. Optimizing multi-grid precondi- tioned conjugate gradient method on multi-cores. IEEE Transactions on Parallel and Distributed Systems, 2024.

[54] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In Proceedings of the conference on high performance computing networking, storage and analysis, pages 1–11, 2009.

[55] Jianhua Gao, Bingjie Liu, Weixing Ji, and Hua Huang. A systematic literature survey of sparse matrix-vector multiplication. arXiv preprint arXiv:2404.06047, 2024.

[56] Jianfei Zhang and Lei Zhang. Efficient cuda polynomial preconditioned conjugate gradient solver for finite element computation of elasticity problems. Mathematical Problems in Engineering, 2013(1):398438, 2013.

[57] Doruk Bozdağ, Umit Catalyurek, Assefaw H Gebremedhin, Fredrik Manne, Erik G Boman, and Füsun Özgüner. A parallel distance-2 graph coloring algorithm for distributed memory computers. In International conference on high performance computing and communications, pages 796–806. Springer, 2005.

[58] Jean RS Blair and Fredrik Manne. An efficient self-stabilizing distance-2 coloring algorithm. Theoretical Computer Science, 444:28–39, 2012.

[59] Daniel Ruiz. Parallelizing hpcg's main kernels, 2018. Accessed: 2024-09-02.

[60] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito 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, pages 474–483. IEEE, 2012.

[61] Takeshi Iwashita and Masaaki Shimasaki. Algebraic multicolor ordering for parallelized iccg solver in finite-element analyses. IEEE transactions on magnetics, 38(2):429–432, 2002.

[62] 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.

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

[64] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the graphblas. In 2016 IEEE High Performance Extreme Computing Conference (HPEC), pages 1–9. IEEE, 2016.

[65] AN Yzelman, D Di Nardo, JM Nash, and WJ Suijlen. A c++ graphblas: specification, implementation, parallelisation, and evaluation. Preprint, 58, 2020.

[66] Alberto Scolari and Albert-Jan Yzelman. Alp/graphblas. https://github.com/Algebraic-Programming/ALP, 2023. Accessed: 2024-08-30.

[67] Aleka McAdams, Eftychios Sifakis, and Joseph Teran. A parallel multigrid poisson solver for fluids simulation on large grids. In Symposium on Computer Animation, volume 65, page 74, 2010.

[68] Qianchao Zhu, Hao Luo, and Chao Yang. Reference and optimized versions of HPCG Benchmark, August 2021.

[69] Shintaro Momose, Takashi Hagiwara, Yoko Isobe, and Hiroshi Takahara. The brand-new vector supercomputer, sx-ace. In Supercomputing: 29th International Conference, ISC 2014, Leipzig, Germany, June 22-26, 2014. Proceedings 29, pages 199–214. Springer, 2014.

[70] Eduardo F D'Azevedo, Mark R Fahey, and Richard 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, pages 99–106. Springer, 2005.

[71] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[72] Yosio OYANAGI. Hyperplane vs. multicolor vectorization of incomplete lu preconditioning. Journal of Information Processing, 2(1), 1987.

[73] Seiji Fujino, Masatake Mori, and Toshiki Takeuchi. Performance of hyperplane ordering on vector computers. Journal of Computational and Applied Mathematics, 38(1-3):125–136, 1991.

[74] Constantino Gómez, Filippo Mantovani, Erich Focht, and Marc Casas. Ve-native port of the HPCG benchmark. https://github.com/efocht/hpcg-ve-open, 2021.

[75] William J Starke, Jeffrey Stuecheli, DM Daly, JS Dodson, Florian Auernhammer, PM Sagmeister, Guy L Guthrie, Charles F Marino, M Siegel, and Bart Blaner. The cache and memory subsystems of the ibm power8 processor. IBM Journal of Research and Development, 59(1):3–1, 2015.

[76] Graph 500. Graph 500 benchmark. https://graph500.org/, 2017. Accessed: 2024-08-13.

[77] Aydin Buluç and Kamesh Madduri. Parallel breadth-first search on distributed memory systems. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, pages 1–12, 2011.

[78] Everett Phillips and Massimiliano Fatica. Optimizing the high performance conjugate gradient benchmark on gpus, 2014.

[79] Michael Luby. A simple parallel algorithm for the maximal independent set problem. In Proceedings of the seventeenth annual ACM symposium on Theory of computing, pages 1–10, 1985.

[80] Mark T Jones and Paul E Plassmann. A parallel graph coloring heuristic. SIAM Journal on Scientific Computing, 14(3):654–669, 1993.

[81] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the gpu. In GPU Technology Conference, pages 1–10, 2012.

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

[83] Oak Ridge National Laboratory. Cray xk7 system at oak ridge national laboratory, 2024. Accessed: 2024-08-13.

[84] Swiss National Supercomputing Centre. Cray xc30 system at the swiss national supercomputing centre, 2024. Accessed: 2024-08-13.

[85] Arthur Bland, Wayne Joubert, Don Maxwell, Norbert Podhorszki, Jim Rogers, Galen Shipman, and Arnold Tharrington. Titan: 20-petaflop cray xk7 at oak ridge national laboratory. In Contemporary High Performance Computing, pages 399–420. Chapman and Hall/CRC, 2017.

[86] Sadaf R Alam, Ladina Gilly, Colin J McMurtrie, and Thomas C Schulthess. Cscs and the piz daint system. In Contemporary High Performance Computing, pages 149–173. CRC Press, 2019.

[87] Edward Rothberg and Anoop Gupta. Parallel iccg on a hierarchical memory multiprocessor—addressing the triangular solve bottleneck. Parallel Computing, 18(7):719–741, 1992.

[88] Ulrike Meier Yang et al. Boomeramg: A parallel algebraic multigrid solver and preconditioner. Applied Numerical Mathematics, 41(1):155–177, 2002.

[89] Jongsoo Park, Mikhail Smelyanskiy, Narayanan Sundaram, and Pradeep 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, pages 124–140. Springer, 2014.

[90] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multipli- cation for gpu architectures. In High Performance Embedded Architectures and Compilers: 5th International Conference, HiPEAC 2010, Pisa, Italy, January 25-27, 2010. Proceedings 5, pages 111–125. Springer, 2010.

[91] Samuel Williams, Dhiraj D Kalamkar, Amik Singh, Anand M Deshpande, Brian Van Straalen, Mikhail Smelyanskiy, Ann Almgren, Pradeep Dubey, John Shalf, and Leonid 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, pages 1–11. IEEE, 2012.

[92] Zachary Bookey. KHPCG 3.0. https://github.com/zabookey/KHPCG3.0, 2016. Accessed: 2024-02-19.

[93] H Carter Edwards, Christian R Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. Journal of parallel and distributed computing, 74(12):3202–3216, 2014.

[94] Christian R Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S Hollman, Dan Ibanez, et al. Kokkos 3: Programming model extensions for the exascale era. IEEE Transactions on Parallel and Distributed Systems, 33(4):805–817, 2021.

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

[96] Craig Ulmer. Sand2021-1220: Benchmarking the nvidia a100 graphics processing unit for high-performance computing and data analytics workloads. Technical report, Sandia National Laboratories, 2021. Accessed: 2024-08-14.

[97] J. Dongarra, M. Heroux, and P. Luszczek. HPCG benchmark, 2019. Accessed: 2024-01-21.

[98] Costas Bekas Cristiano Malossi, Panagiotis Chatzidoukas. Ibm hpcg. https://github.com/IBM/HPCG, 2018. Accessed: 2024-04-11.

[99] 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.

[100] NVIDIA John C. Linford. Arm HPCG benchmarks. https://gitlab.com/arm-hpc/benchmarks/hpcg, 2017. Accessed: 2024-06-17.

[101] Daniel Ruiz. HPCG for arm. https://github.com/ARM-software/HPCG_for_Arm, 2020. Accessed: 2024-08-01.

[102] 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.

[103] Nikita Shustrov Mohammad Almasri. Nvidia hpc benchmarks. https://github.com/NVIDIA/nvidia-hpcg, 2023. Accessed: 2024-07-05.

[104] NVIDIA. Nvidia hpc benchmarks container, 2024. Accessed: 2024-09-02.

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

[106] Alberto Zeni, Kenneth O'Brien, Michaela Blott, and Marco D. Santambrogio. HPCG fpga. https://github.com/Xilinx/ HPCG_FPGA, 2019. Accessed: 2024-05-28.

[107] HPCG benchmark. https://www.hpcg-benchmark.org/, 2013. Accessed: 2024-02-15.

[108] IBM Power Systems Performance Group. High performance conjugate gradient benchmark (hpcg) for ibm power9 systems, November 2018. SC18, Dallas, Texas.

[109] Xie Lulu and Erich Strohmaier. Optimizing HPCG for amd processors, 2019. Accessed: 2024-09-02.

[110] AMD. Optimizing HPCG with openmp on amd processors, 2020. Accessed: 2024-09-02.

[111] Sergey Kopysov, Nikita Nedozhogin, and Leonid Tonkov. Parallel pipelined conjugate gradient algorithm on hetero- geneous platforms. International Journal of Computer and Information Engineering, 16(10):423–430, 2022.

[112] Barcelona Supercomputing Center (BSC). Ompss-2: A parallel programming model. https://pm.bsc.es/ompss-2, 2024. [Accessed 14-08-2024].