Portable Parallelism in C++

A Comparison of stdpar and stdexec with the HPCG Benchmark

By

AJINKYA BHALERAO

Department of Computer Science
UNIVERSITY OF BRISTOL

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of MASTER OF SCIENCE in the Faculty of Engineering.

SEPTEMBER 2025

Word count: 11000

Abstract

Parallelism over both CPU and GPU targets is becoming increasingly important in modern systems. While a variety of parallel programming models exist to target such hardware, many remain outside the C++ standard, limiting portability and consistency for developers. In recent years, however, the C++ standard has introduced parallel (`stdpar`) algorithms and is also moving towards incorporating a standard execution (`stdexec`) model.

Despite these developments, there remains limited understanding of how these models perform in practice. This study compares the `stdpar` and `stdexec` models using the HPCG benchmark, focusing chiefly on performance while also considering practical implications. To this end, NVIDIA's experimental implementation of `stdexec` is employed.

Two implementations of the HPCG benchmark (one for `stdpar`, one for `stdexec`) have been developed. To enable effective parallelism on both CPUs and GPUs, the serial symmetric Gauss Seidel (SYMGS) kernel has been parallelised. In addition, a modified version of the baseline HPCG implementation (which employs OpenMP) has been produced incorporating the parallel SYMGS algorithm for comparative purposes.

The `stdpar` and `stdexec` models have been compared along five dimensions: absolute runtime, scalability, heterogeneous programming support, portability, and the maturity of their programming environments. The contributions of this study are summarised below:

- While the `stdpar` and `stdexec` parallelism models achieve comparable runtimes, this study suggests a slight performance benefit when using `stdpar`.
- For certain kernels, the performance gap between `stdpar` and `stdexec` is more pronounced, such as those involving reduction operations.
- While `stdpar` and `stdexec` generally display comparable scaling behavior, for small problem sizes `stdpar` demonstrates superior CPU scaling relative to `stdexec`, with CPU performance exceeding GPU performance; this effect is not observed with `stdexec`.
- While `stdpar` code seems to consistently outperform `stdexec` code across different platforms, the performance difference is less on GPU platforms compared to CPU platforms.

Dedication and Acknowledgements

I would like to express my sincere gratitude to the University of Bristol, and to the professors and staff who make it an inspiring place to study and conduct research. I am especially thankful to my supervisor, Dr. Tom Deakin, for his invaluable guidance, support, and encouragement throughout the course of this project. I would also like to acknowledge the High Performance Computing Group at the University of Bristol for their advice and assistance whenever it was needed. Their expertise and willingness to help have been greatly appreciated. Finally, I would like to thank members of the NVIDIA HPC Compiler and Libraries Team, particularly those working on the `stdexec` Library, for their helpful discussions and support during the development of this work.

Author's Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Ajinkya Bhalerao, September 2025

Table of Contents

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

1.1 Motivation

Parallelism is becoming increasingly important as modern systems rely more heavily on multi-core processors, distributed architectures, and accelerators. In particular, the use of GPUs has become commonplace, not only in high-performance computing but also in general-purpose computing domains. While a variety of parallel programming models exist to target such hardware, many remain outside the C++ standard, limiting portability and consistency for developers. In recent years, however, the C++ standard has introduced parallel algorithms (also known as `stdpar` algorithms) and is moving towards incorporating a standard execution model (also known as `stdexec`) which incorporates sender and receiver abstractions.

Despite these developments, there remains limited understanding of how the emerging C++ parallelism models perform in practice. In high-performance computing, where efficiency and scalability are critical, it is important to assess whether these standard models can deliver competitive performance while also providing the portability and consistency that the C++ standard aims to ensure. To address this gap, this study compares the `stdpar` and `stdexec` models using the HPCG benchmark, focusing chiefly on performance while also considering practical implications. To this end, NVIDIA's experimental implementation of `stdexec` is employed, providing a basis for assessing how the model performs in practice relative to `stdpar`.

1.2 Aims and Objectives

This study aims to compare the performance of the `stdpar` and `stdexec` parallelism models. The comparison between the two models is performed along five dimensions:

- Absolute runtime
- Scalability
- Heterogeneous programming support

- Portability
- Maturity of their programming environments

1.3 Thesis Structure

The thesis is structured as follows:

- **Background and Context:** the HPCG benchmark is introduced. Parallelism models both within and outside of the C++ standard are discussed. A summary of related work is provided.
- **Theory of the HPCG Benchmark:** this section presents the algorithms and mathematical foundations required to understand the HPCG benchmark.
- **Development Process:** the implementation details of the `stdpar` and `stdexec` codes are briefly discussed. Particular attention is given to the `stdexec` dot product kernel, the parallelisation of the symmetric Gauss-Seidel algorithm, and the optimisation of memory layout.
- **Results and Discussion:** the performance characteristics of the `stdpar` and `stdexec` codes are examined across five dimensions: absolute runtime, scalability, heterogeneous programming support, portability, and programming environment maturity.
- **Conclusions and Further Work:** the main findings are summarised. Suggestions for further work are provided.

Chapter 2

Background and Context

2.1 The HPCG Benchmark

Benchmarks play a central role in high-performance computing by providing a standardised means of evaluating how well a system performs a given class of tasks. They serve as a *yardstick* for measuring and comparing performance, offering both hardware and software developers a reference point when designing and optimising new architectures and applications.

The High-Performance Conjugate Gradient (HPCG) benchmark was introduced in 2013 with the intention of being more representative of modern computer applications than traditional benchmarks [25]. The developers of the HPCG benchmark noted that the long-standing HPL benchmark [8] is dominated by dense matrix-matrix multiplication, with modest memory access demands and performance limited primarily by processor speed. In contrast, they noticed that many real-world scientific applications demanded large amounts of data access for relatively low computation. They observed that other benchmarks, such as the NAS Parallel Benchmarks [2] and one by Dongarra [7], were inadequate, lacking features like sparse matrices from physical systems and scalable distributed-memory parallelism.

The resulting benchmark consists of a series of kernels designed to capture a range of memory access patterns and computational behaviors (e.g. dot product, sparse matrix–vector multiplication, and Gauss–Seidel triangular solver) exhibiting relatively low arithmetic intensity. The conjugate gradient method (Section 3.1) is used to solve a sparse linear system generated from the 3D Poisson problem using a 27-point kernel. To accelerate convergence, multigrid preconditioning with symmetric Gauss-Seidel smoothing is performed. Multigrid preconditioning applies Gauss–Seidel smoothing across progressively coarser representations of the linear system. As a result, Gauss Seidel must operate on matrices of widely different different sizes, ranging from the large sparse system itself to much smaller coarse-grid approximations. Furthermore, Gauss–Seidel is inherently sequential due to data dependencies that require rows to be processed in order, making Gauss–Seidel particularly challenging to parallelise effectively.

Alongside HPL, the HPCG benchmark is included in the TOP500 rankings, which list the world's 500 most powerful supercomputers twice annually. However, systems typically attain far lower performance on HPCG than on HPL, reflecting the greater difficulty of sustaining high floating-point throughput on memory-bound workloads. Several hardware vendors have developed HPCG implementations optimized for their respective architectures. A notable example is the GPU-optimised implementation provided by NVIDIA [19].

2.2 Parallel Programming Outside the C++ Standard

There are numerous languages and frameworks outside of the C++ standard which facilitate the parallel execution of code across different execution targets, such as CPUs and GPUs[1].

Directive-based languages such as OpenMP and OpenACC allow the user to annotate sections of code for parallel execution or GPU offloading. These approaches offer a relatively simple means of achieving parallelisation. However, they provide limited fine-grained control, as parallelism can only be expressed in specific, predefined contexts.

Native GPU programming languages such as CUDA, OpenCL and HIP allow the user to write GPU kernels and invoke them from the host. While providing fine-grained control over GPU execution, these languages typically require separate source code for CPU and GPU components, which complicates development and maintenance. In addition, such languages are often tied to specific hardware architectures — for example, CUDA is restricted to NVIDIA GPUs.

Programming models such as SYCL, Kokkos and RAJA allow for increased portability of code. While SYCL allows kernels to be written as C++ lambdas and offloads them to a chosen backend (OpenCL, CUDA, CPU), Kokkos and RAJA allow specific patterns of code to parallelised e.g. `Kokkos::parallel_for`, `RAJA::forall`.

Other libraries deserving mention include Intel Thread Building Blocks (TBB) and High Performance ParallelX (HPX). TBB supports task-based parallelism on CPUs and offers features ranging from high-level parallel patterns (e.g. `tbb::parallel_for`, `tbb::parallel_sort`) to dependency-aware graph construction with nodes such as `function_node` and `join_node`. HPX provides a wide range of parallel and asynchronous programming facilities, such as asynchronous functions, futures, parallel algorithms (e.g. `hpx::reduce`), and the sender–receiver model. A notable capability of HPX is that these features extend beyond a single node, enabling distributed execution across clusters.

2.3 Portable Parallelism in Modern C++

Since C++11, the language has provided concurrency primitives such as threads, atomic variables, locks, condition variables, and futures[2]. However, these constructs are relatively low-level, and until recently C++ has lacked a unified framework for expressing asynchronous and parallel computation in a consistent manner.

Introduced in C++17, the Parallel Standard Library (PSTL) provides standard parallel (`stdpar`) algorithms —

[1]For an overview (including code examples) of many of the languages and frameworks mentioned in this section, refer to *Parallel and High Performance Computing* by Robey and Zamora [33].

[2]*C++ Concurrency in Action* by Williams [31] provides an excellent overview of such features.

ready-made functions for parallel execution that abstract away low-level concerns such as thread management and workload distribution. Each algorithm represents a generic computational pattern; for example, `for_each` applies a callable across a range of values, while `transform_reduce` combines element-wise transformation with a reduction operation. With NVIDIA's `nvc++` compiler, `stdpar` algorithms can be compiled for either CPU or GPU execution without requiring any changes to the source code, provided the code is GPU-compatible and satisfies certain constraints.

Despite their utility, the `stdpar` algorithms impose several important restrictions. First, they are synchronous: execution blocks until completion, preventing overlap between CPU and GPU work. Second, switching between CPU and GPU execution is inflexible; code must be compiled separately for each target, and workarounds are cumbersome. Third, the model is limited to fork–join parallelism, where iterations converge back to a single serial flow, offering little support for more complex execution patterns.

The standard execution framework (`stdexec`) shows considerable promise in addressing these limitations. It is formalized in proposal P2300 [6], has been drafted into the C++ standard, and is planned for inclusion in C++26[3]. NVIDIA has also developed an experimental implementation [18]. As the framework is still evolving, its terminology and concepts may change, but its current form can be summarised as follows.

At the core of the proposal are two abstractions: senders and receivers. A sender represents a step in a computation, while a receiver denotes the endpoint for one or more senders. They interact through `stdexec::connect` and `stdexec::start`: `connect` links a sender to a receiver and produces an operation state (capturing the information needed to perform the operation), and `start` initiates execution. An execution resource (or execution context) manages and performs the running of senders on specific hardware. Examples from the NVIDIA library include `static_thread_pool` (for CPU execution) and `stream_context` (for GPU execution). An execution resource yields a scheduler (using its `get_scheduler` method) which provides a means of allocating senders against the execution resource. Notably, a receiver must implement three methods: `set_value`, `set_stopped` and `set_error`, which define what the program does when the sender returns a valid value, the operation is cancelled, or the sender returns an error. Once the sender and receiver have been connected and `start` has been called, the execution resource will eventually call one of these three methods.

An important characteristic of senders is that they are composable, meaning they can be joined together into dependency-aware task graphs. Sender adaptors are integral in enabling this composabality. A sender adaptor is an algorithm that takes one or more senders and produces another sender. For instance, `then` composes a sender with an additional function node applied to its output values; `bulk` adds a node representing the parallel execution of a function across many iterations (similar to `for_each`); `when_all` produces a sender that completes once all its inputs have finished; and `continues_on` switches execution of a task graph from one execution resource to another. To simplify programming and avoid deeply nested callbacks, senders and sender adaptors can be composed using the Unix-style pipe operator (Listing 2.1). In this syntax, the left-hand operand is a sender, while subsequent operands are sender adaptors, enabling the construction of linear task graphs (pipelines).

[3]The sender/receiver abstractions are already incorporated into the current working draft of the C++ standard [13].

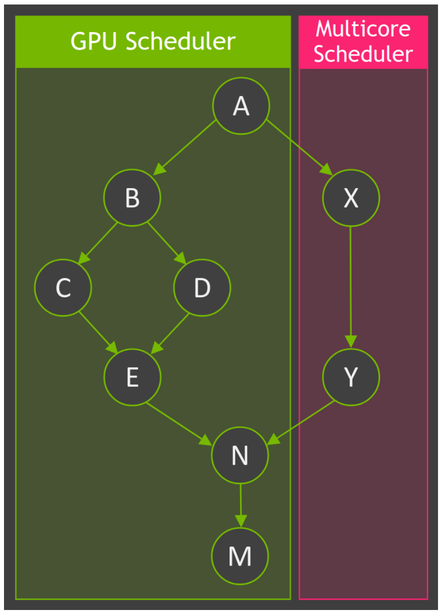


Figure 2.1: Illustration of dependency-aware task-graph (reproduced from NVIDIA [5]). Here, A is an initial task (likely a starting-point sender) executed on the GPU. Once A completes, B and X are executed on the GPU and CPU respectively; these tasks are likely sender adaptors. An arbitrary arrangement of tasks then follows before the final task M is performed.

```
my_sender = schedule(gpu_sched) | bulk(par_unseq, grid_sz, calc_energy) | then(calc_tot_energy);
```

Listing 2.1: Illustration of sender pipeline.

When senders are composed in this way, their associated work is asynchronous and may not yet have begun. To initiate execution and wait for its completion, the constructed task graph can be passed to `sync_wait`, which connects it to a blocking receiver and drives the entire graph to completion. In practice, receivers are not typically created by the user but are instantiated implicitly when `sync_wait` is invoked. Because `sync_wait` blocks the calling thread (effectively turning an asynchronous computation into a synchronous one) it should only be used when no other useful work remains on the calling thread.

In summary, the P2300 proposal introduces intuitive abstractions for specifying work across diverse execution resources without requiring the programmer to manage the low-level details of execution. Beyond NVIDIA's `stdexec` library, Intel has produced an implementation targeting bare-metal microcontrollers [12], albeit with only a subset of the proposed features. Meta's `libunifex` library [9] also implements P2300 and has reportedly been deployed in production software, though it does not support GPU offloading. In practice, a sender is defined as a C++ concept, and users may implement their own as long as the concept requirements are met. In this project, the senders, adaptors, execution resources, and schedulers provided by NVIDIA's `stdexec` library are employed.

2.4 Related Work

The survey of related work was guided by three central questions:

- Have studies comparing `stdpar` and `stdexec` been performed?
- To what extent has the HPCG benchmark been implemented using senders and receivers or the standard parallel C++ algorithms?
- More generally, have senders and receivers been employed in high-performance computing projects?

So far as we are aware, only one study directly compares the `stdpar` and `stdexec` libraries. Haseeb et al. [11] use PSTL, senders and receivers and CUDA for different problems (Smith-Waterman sequence alignment, heat equation, 1D/2D stencil operations) and note the three give similar performance. However, the study focuses on relatively small code fragments. What is still lacking is a demonstration of senders and receivers within a larger, more substantial application — one that more closely reflects the scope and complexity of real high-performance computing workloads.

To the best of our knowledge, HPCG has not been implemented using C++ standard parallel algorithms or senders and receivers. However, standard parallel algorithms have been used in other linear algebra applications. Hanawa et al. [10] port an existing benchmark (which solves the 3D Poisson equation with Conjugate Gradient method) to various implementations including PSTL. The NAS Parallel Benchmarks have also been implemented using PSTL [15, 17]. Malenza et al. [16] use the PSTL to solve an astrophysics-derived linear system using the LSQR algorithm. Breyer et al. [3] implement a kernel matrix assembly algorithm and a BLAS level 3 SYMM function in evaluating the performance of `stdpar` across different compilers. Asahi et al. [1] perform a plasma physics simulation with PSTL algorithms and obtain reasonable performance when comparing against Kokkos and OpenMP implementations. Scolari et al. [28] implement HPCG using GraphBLAS.

Apart from Haseeb et al. [11], only two more works document using senders and receivers within an HPC context. Solca et al. [30] use senders and receivers for an eigensolver using the Pika implementation of `stdexec` [23]. Compared to implementations from other libraries (e.g. ScaLAPACK, SLATE), it performs comparably and in some cases better. Yuan et al. [32] use senders and receivers for a fluids-based simulation using the Lattice Boltzmann method.

Among the few papers that have explored senders and receivers, some appear to have relied on early revisions of NVIDIA's implementation. As the implementation is still under active development, it remains important to conduct further studies and, as previous work has done, identify potential issues that could inform improvements by the NVIDIA team. Given that only three studies were found to implement senders and receivers, additional results would help build a broader consensus regarding their suitability for high-performance computing applications.

Chapter 3

Theory of the HPCG Benchmark

This chapter provides a concise overview of the theoretical background of the HPCG benchmark. In particular, it outlines how the benchmark employs the Conjugate Gradient method, together with multigrid preconditioning, to solve sparse linear systems.

3.1 The Conjugate Gradient Method

The Conjugate Gradient[1] (CG) method assumes a linear system of the form:

$$Ax = b \tag{3.1}$$

where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. A and b are known whereas x is unknown. CG also assumes A is symmetric and positive definite (SPD). A is positive definite given $v^T A v > 0$ for all $v \neq 0$. The quadratic form of Equation 3.1 is the following scalar function:

$$f(x) = \frac{1}{2} x^T A x - b^T x \tag{3.2}$$

For SPD A, $f(x)$ is convex (for $A \in \mathbb{R}^{2 \times 2}$ a plot of $f(x)$ would appear bowl-shaped) and the x value which minimises $f(x)$ is also the solution of Equation 3.1. CG starts with an initial guess x_0 and takes a series of steps refining the initial guess until it converges to the minimum of $f(x)$. The direction of the i^{th} step is the unit vector $d_{(i)}$ (sometimes referred to as $p_{(i)}$) and its magnitude is $\alpha_{(i)}$. The value $x_{(i+1)}$ after the i^{th} step is given by:

$$x_{(i+1)} = x_{(i)} + \alpha_{(i)} d_{(i)} \tag{3.3}$$

[1]For a detailed description of the Conjugate Gradient method, the reader is referred to Shewchuk [29].

All search directions are A-orthogonal (or conjugate) meaning $d_{(i)}^T Ad_{(j)} = 0$ for $i \neq j$. This minimises the number of steps to be taken, which is at most n. The $\alpha_{(i)}$ values are given by:

$$\alpha_{(i)} = \frac{d_{(i)}^T r_{(i)}}{d_{(i)}^T Ad_{(i)}} \tag{3.4}$$

where the residual r_i is defined as:

$$r_{(i)} = b - Ax_{(i)} \tag{3.5}$$

Gram-Schmidt conjugation is used to calculate the search directions. Given a set of n linearly independent vectors u_0, u_1, $u_2 \ldots u_{n-1}$, $d_0 = u_0$ and for $i > 0$:

$$d_{(i)} = u_i + \sum_{k=0}^{i-1} \beta_{ik} d_{(k)} \tag{3.6}$$

$$\beta_{ij} = -\frac{u_i^T Ad_{(j)}}{d_{(j)}^T Ad_{(j)}} \tag{3.7}$$

Equations 3.6 and 3.7 require old search directions are kept in memory to generate the new directions. CG eliminates this requirement by using the residuals to construct the search directions, setting $u_i = r_{(i)}$. It can be shown that the residual $r_{(i)}$ is orthogonal to all previous directions except $d_{(i-1)}$. As a result, Equations 3.6 and 3.7 become:

$$d_{(i)} = r_{(i)} + \beta(i) d_{(i-1)} \tag{3.8}$$

$$\beta_{(i)} = -\frac{r_{(i)}^T r_{(i)}}{r_{(i-1)}^T r_{(i-1)}} \tag{3.9}$$

Equation 3.4 becomes:

$$\alpha_{(i)} = \frac{r_{(i)}^T r_{(i)}}{d_{(i)}^T Ad_{(i)}} \tag{3.10}$$

Now, only $d_{(i-1)}$ needs to be stored in memory when calculating the new direction $d_{(i)}$. Using the residuals to construct the search directions also gives reduced time complexity of $O(n^{4/3})$ for three-dimensional problems. The CG algorithm is summarised in Listing 1 (ε is the desired convergence tolerance).

Algorithm 1 Conjugate Gradient method

```
1: $r_0 \leftarrow b - Ax_0$
2: $d_0 \leftarrow r_0$
3: $i \leftarrow 0$
4: while $\|r_i\| > \varepsilon$ do
5:     $\alpha_i \leftarrow \langle r_i, r_i \rangle / \langle Ad_i, d_i \rangle$
6:     $x_{i+1} \leftarrow x_i + \alpha_i d_i$
7:     $r_{i+1} \leftarrow r_i - \alpha_i A d_i$
8:     $\beta_{i+1} \leftarrow \langle r_{i+1}, r_{i+1} \rangle / \langle r_i, r_i \rangle$
9:     $d_{i+1} \leftarrow r_{i+1} + \beta_{i+1} d_i$
10:    $i \leftarrow i + 1$
11: end while
```

Iterative methods like CG often use preconditioning to accelerate convergence[2]. Preconditioners are generally applied to the residual $r_{(i)}$ producing the preconditioned residual $z_{(i)}$:

$$z_{(i)} = \mathcal{P}(A,\, r_{(i)}) \tag{3.11}$$

$\mathcal{P}$ is the preconditioner operator. For some preconditioners, Equation 3.11 can be expressed in the form:

$$z_{(i)} = M^{-1} r_{(i)} \tag{3.12}$$

M is typically an SPD matrix which approximates A and is computationally inexpensive to invert. Since the multigrid preconditioning used by CG cannot be expressed in this way, Equation 3.11 is used instead. A revised CG method (with changes in red) is given as Algorithm 2.

Algorithm 2 Conjugate Gradient method with preconditioning

```
1: $r_0 \leftarrow b - Ax_0$
2: $z_0 = \mathcal{P}(A,\, r_0)$
3: $d_0 \leftarrow z_0$
4: $i \leftarrow 0$
5: while $\|r_i\| > \varepsilon$ do
6:     $\alpha_i \leftarrow \langle r_i, z_i \rangle / \langle Ad_i, d_i \rangle$
7:     $x_{i+1} \leftarrow x_i + \alpha_i d_i$
8:     $r_{i+1} \leftarrow r_i - \alpha_i A d_i$
9:     $z_{i+1} = \mathcal{P}(A,\, r_{i+1})$
10:    $\beta_{i+1} \leftarrow \langle r_{i+1}, z_{i+1} \rangle / \langle r_i, z_i \rangle$
11:    $d_{i+1} \leftarrow z_{i+1} + \beta_{i+1} d_i$
12:    $i \leftarrow i + 1$
13: end while
```

[2]Saad [26] provides a discussion of preconditioned CG which the reader is referred to.

3.2 Symmetric Gauss-Seidel Algorithm

The Symmetric Gauss-Seidel (SYMGS) algorithm can be used as both an iterative solver and a preconditioner for linear systems of the form $Ax = b$. The HPCG benchmark uses SYMGS as an iterative solver within its multigrid preconditioning process. Given $x_{(i)}$ is the current approximate solution of the linear system, SYMGS calculates $x_{(i+1)}$. Let $\xi_i^{(k)}$ be the i^{th} component of $x_{(i)}$. $\xi_i^{(k+1)}$ is calculated by first performing a forwards sweep to calculate $\xi_i^{(f)}$:

$$\xi_i^{(f)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(f)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)}\right) \tag{3.13}$$

β_i is the i^{th} component of b and a_{ij} is the $(i,j)^{\text{th}}$ component of A. n is the dimension of A. The forwards sweep is performed in ascending $(i = 1,\, 2,\, 3 \ldots n)$ order. The forwards sweep is followed by a backwards sweep:

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(f)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k+1)}\right) \tag{3.14}$$

The backwards sweep is performed in descending $(i = n,\, n-1,\, n-2 \ldots 1)$ order.

3.3 Geometric Multigrid Preconditioning

The HPCG benchmark uses multigrid[3] (MG) preconditioning. Iterative solvers like Gauss-Seidel are effective at reducing high-frequency error components but struggle with low-frequency errors. The multigrid method transfers low-frequency errors to coarser grids where they appear relatively more high-frequency and can be more effectively reduced.

The MG process implemented by the HPCG benchmark operates on four levels $\ell = 0,\, 1,\, 2,\, 3$. The level $\ell = 0$ corresponds to the underlying problem domain. Each subsequent level in the multigrid hierarchy is twice as coarse than the level before and contains $\frac{1}{8}$ as many grid points. The domains of the levels are denoted by Ω^h, Ω^{2h}, Ω^{4h} and Ω^{8h} where h is the grid spacing of the finest level and Ω^h the finest-level domain $(l = 0)$.

The MG algorithm (3) follows the notation of *A Multigrid Tutorial* [4]. The algorithm is recursive with $()^h$ and $()^{2h}$ superscripts denoting entities in the current level and adjacent (coarser) level respectively. The algorithm takes two parameters: the matrix A^h and the vector $\mathbf{f}^h$. It returns a vector $\mathbf{v}^h$. From the perspective of CG, A^h, $\mathbf{f}^h$ and $\mathbf{v}^h$ correspond to A, r and z respectively (see Algorithm 2). $\mathbf{v}^h$ is first set to 0 (line 1) and then (assuming the current level is not the coarsest level) SYMGS iteratively solves the following system ν_1 times:

[3] A detailed discussion of the multigrid process can be found in *A Multigrid Tutorial* [4].

$$A^h \mathbf{u}^h = \mathbf{f}^h \tag{3.15}$$

The initial guess for the SYMGS iterations is $\mathbf{v}^h$. After each SYMGS iteration, the updated solution is stored in $\mathbf{v}^h$. Because of Equation 3.15, $\mathbf{f}^h$ and $\mathbf{u}^h$ are called the *right hand side vector* and *solution vector* respectively. SYMGS has therefore preconditioned the residual but only addressed the error frequencies which can be effectively reduced at this level. The problem is transferred to the next level by applying the restriction operator I_h^{2h} to $\mathbf{f}^h$ and producing $\mathbf{f}^{2h}$ (line 4). The algorithm is then applied recursively to the coarser problem (line 5). Upon reaching the coarsest level, SYMGS is applied (line 9) and the corrections at each level are accumulated by applying the prolongation operator I_{2h}^h to $\mathbf{v}^{2h}$ (line 6) which updates the value of $\mathbf{v}^h$. Post-prolongation, SYMGS is again applied (line 7) to refine $\mathbf{v}^h$ further.

Algorithm 3 Multigrid preconditioning $\mathbf{v}^h \leftarrow V^h(\mathbf{v}^h, \mathbf{f}^h)$

```
 1: $\mathbf{v}^h \leftarrow 0$
 2: if $l \neq 4$ then
 3:     Repeat $\nu_1$ times: $\mathbf{v^h} \leftarrow \text{SYMGS}(A^h, \mathbf{v}^h, \mathbf{f}^h)$
 4:     $\mathbf{f}^{2h} \leftarrow I_h^{2h}(\mathbf{f}^h)$
 5:     $\mathbf{v}^{2h} \leftarrow V^{2h}(\mathbf{v}^{2h}, \mathbf{f}^{2h})$
 6:     $\mathbf{v}^h \leftarrow \mathbf{v}^h + I_{2h}^h(\mathbf{v}^{2h})$
 7:     Repeat $\nu_2$ times: $\mathbf{v^h} \leftarrow \text{SYMGS}(A^h, \mathbf{v}^h, \mathbf{f}^h)$
 8: else
 9:     $\mathbf{v^h} \leftarrow \text{SYMGS}(A^h, \mathbf{v}^h, \mathbf{f}^h)$
10: end if
```

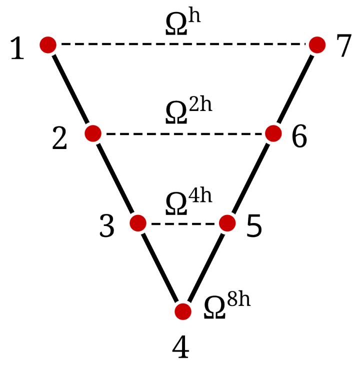
The matrices A^h used at each level do not change during the CG process and are computed once beforehand such that they never need to be computed by the MG algorithm. HPCG also sets $\nu_1 = \nu_2 = 1$. The restriction and prolongation operators transform the problem from fine-grid to coarse-grid and vice-versa. Various schemes may be employed to define I_h^{2h} and I_{2h}^h including injection, full-weighting and linear interpolation. The HPCG benchmark uses injection. For restriction, values from the fine grid are directly copied to the corresponding coarse grid points without modification or averaging. For prolongation, values from the coarse grid are directly added to the corresponding fine grid points. Equation 3.16 shows the element-wise restriction operation used to calculate $\mathbf{f}^{2h}$. Here, f_j^{2h} is the j^{th} element of $\mathbf{f}^{2h}$ and n_c the dimension of $\mathbf{f}^{2h}$. Equation 3.17 shows the element-wise prolongation operation used to correct $\mathbf{v}_j^h$. v_j^{2h} is the j^{th} element of $\mathbf{v}^{2h}$.

$$f_j^{2h} = f_{2j}^h \quad \forall j = 1, 2 \dots n_c \tag{3.16}$$

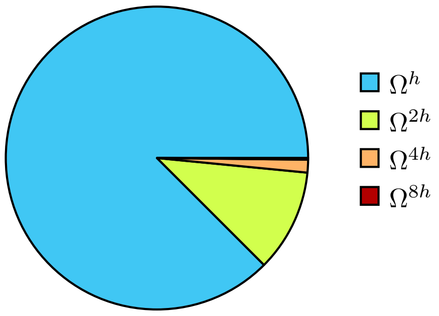
$$v_j^h = v_j^h + v_j^{2h} \quad \forall j = 1, 2 \dots n_c \tag{3.17}$$

The MG process may also be expressed as a V-cycle (see Figure 3.1a) consisting of seven stages. Stages 1, 2 and 3 constitute the coarsening phase, during which the problem is successively restricted to coarser grid levels. These stages involve applying SYMGS (line 3, Algorithm 3) and restriction (line 4) to progressively lower resolutions. Stage 4 represents the coarsest grid solve (line 9). Stages 5, 6 and 7 prolongate corrections back to finer grid levels through prolongation of the coarse preconditioned residual (line 6) and post-prolongation

SYMGS (line 7). A pie chart estimating the rough workload across levels (by comparing the sizes of A^h at each level) is also shown (Figure 3.1b).



(a)



(b)

Figure 3.1: (a) Multigrid (MG) process V-cycle consisting of seven stages across four levels of coarseness (Ω^h is the finest and Ω^{8h} the coarsest). (b) Estimated distribution of V-cycle work across grids. Percentages (from the relative sizes of the matrix A at each level) are 87.52%, 10.94%, 1.37% and 0.17% for Ω^h, Ω^{2h}, Ω^{4h} and Ω^{8h} respectively.

Chapter 4

Development Process

This chapter describes the development process for implementing the HPCG benchmark kernels with `stdpar` and `stdexec` algorithms. The chapter opens with a concise overview of how the implementations were designed. The discussion then turns to the dot product kernel, which proved challenging to implement with `stdexec`. The remainder of the chapter details the approach taken to parallelise the SYMGS algorithm. Specifically, it discusses the colouring strategy employed, the use of `stdpar` and `stdexec` to implement parallel SYMGS, and the memory layout optimisation introduced to improve spatial locality.

4.1 Code Design and Structure

4.1.1 `stdpar` Implementation

- For `stdpar`, the six kernels from the baseline code (`ComputeDotProduct_ref`, `ComputeRestriction_ref`, `ComputeProlongation_ref`, `ComputeSPMV_ref`, `ComputeSYMGS_ref` and `ComputeMG_ref`) were replaced with `_stdpar` equivalents[1]. These equivalents were like-for-like functions taking the same parameters and returning the same values as their baseline counterparts. The top-level CG algorithm (`CG.cpp`) remained unchanged (see Figure B.1). When the `-DSELECT_STDPAR` compiler flag was used, `CG.cpp` was compiled to use the `_stdpar` versions of the kernels rather than the baseline versions.

- OpenMP pragmas `#pragma omp parallel for` and `#pragma omp parallel for reduction` are used by the baseline code to parallelise relevant parts of the kernels. With `stdpar`, the `std::for_each` and `std::transform_reduce` functions respectively were used to provide equivalent functionality.

- For consistency with the baseline HPCG implementation, the `stdpar` WAXPBY kernel had three variants depending upon whether (i) $\alpha = 1$, (ii) $\beta = 1$ or (iii) $\alpha \neq 1$ and $\beta \neq 1$[2] which eliminated unnecessary multiplication operations.

[1]The code for the `stdpar` kernel implementations is provided in Appendix D. Also, designing for GPU-compatibility imposed constraints on both the `stdpar` and `stdexec` kernels which are detailed in Appendix I.

[2]WAXPBY performs the vector operation: $w = \alpha x + \beta y$.

- Although officially the HPCG benchmark code exposes only specific kernels for user optimisation, the `CG` algorithm also performs the `CopyVector` and `ZeroVector` operations. These two additional operations are serial algorithms executed on the CPU. During the development of GPU-based `stdpar` and `stdexec` HPCG implementations, the `CopyVector` and `ZeroVector` operations became significant performance bottlenecks and were replaced by equivalent WAXPBY operations for GPU offloading.

4.1.2 `stdexec` Implementation

- `CG_stdexec.cpp` was written as a standalone algorithm distinct from the `CG.cpp` file used by the baseline and `stdpar` code versions. This is because `CG_stdexec` used a fundamentally different structure compared to regular `CG`, with all invoked kernels being sender adaptors within a series of sender pipelines. When the `-DSELECT_STDEXEC` compiler flag was used, `CG` was compiled to call `CG_stdexec` instead of running the algorithm the baseline and `stdpar` implementations use (see Figure B.1).
- `stdexec` equivalents of five baseline kernels (`ComputeDotProduct_ref`, `ComputeRestriction_ref`, `ComputeSPMV_ref`, `ComputeProlongation_ref` and `ComputeSYMGS_ref`) were written and defined within `CG_stdexec.cpp` rather than having their own `.cpp` files[3].
- The baseline code's use of `#pragma omp parallel for` was generally mapped to `stdexec::bulk`. Non-kernel operations appearing between kernels were incorporated into the asynchronous sender pipeline with `stdexec::then`, ensuring the pipeline remained uninterrupted. Since an equivalent operation for `#pragma omp parallel for reduction` (used within the baseline dot product kernel) was unavailable in the `stdexec` library, the HPCG `stdexec` implementation performed a two-stage process consisting of a `stdexec::bulk` accumulation followed by a `stdexec::then` reduction (Section 4.2).
- Unlike the baseline and `stdpar` HPCG implementations, the `stdexec` WAXPBY kernel does not have three variants but only one. Implementing a three-way specialisation of WAXPBY would have increased the code complexity for negligible benefit, as this kernel is not performance-dominant and the optimisations merely remove a single multiplication.
- As mentioned before, `ZeroVector` and `CopyVector` operations were replaced with WAXPBY equivalents.
- The `stdexec` code incorporates several optimisations, primarily aimed at improving performance. These are described in detail in Appendix G.

The CG algorithm involves initial work followed by a repeating loop until convergence. Synchronisation points were therefore placed after the initial work and at the end of each loop iteration with `stdexec::sync_wait`. Since the first loop iteration performs slightly different operations compared to later iterations (due to initialisation steps) it was implemented separately from the main loop. Listing 4.1 shows the resulting code structure.

[3]The code for the `stdexec` kernel implementations is provided in Appendix E.

```
stdexec::sender auto pre_loop_work = ...;
stdexec::sync_wait(pre_loop_work);

stdexec::sender auto first_loop_work = ...;
stdexec::sync_wait(first_loop_work);

while(residual > tolerance){
  stdexec::sender auto loop_work = ...;
  stdexec::sync_wait(loop_work);
}
```

Listing 4.1: Placement of synchronisation points after pre-loop work, first-loop work and subsequent loop iterations.

Listing 4.2 shows a simplified form of the baseline `ComputeMG` algorithm. The recursion of the algorithm stops when it reaches the coarsest level of the V-cycle. The abstractions defined in proposal P2300 do not support a recursion mechanism. The approach taken was to unroll the recursion. An illustration of the resulting code is shown in Listing 4.3.

```
int ComputeMG(SparseMatrix &A, Vector &r, Vector &x){
  if(!lowestLevel){
    ComputeSYMGS(...);
    ComputeSPMV(...);
    ComputeRestriction(...);
    ComputeMG(A.Ac, A.mgData->rc, A.mgData->xc);
    ComputeProlongation(...);
    ComputeSYMGS(...);
  }
  else
    ComputeSYMGS(...);
  return 0;
}
```

Listing 4.2: Simplified structure of baseline `ComputeMG` algorithm. `Ac`, `rc` and `xc` are the coarse representations of `A`, `r` and `x` at the next level of the multigrid hierarchy.

```
//V-cycle stage 1
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeSYMGS(...)
| ComputeSPMV(...)
| ComputeRestriction(...));

//V-cycle stage 2
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeSYMGS(...)
| ComputeSPMV(...)
| ComputeRestriction(...));

//V-cycle stage 3
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeSYMGS(...)
| ComputeSPMV(...)
| ComputeRestriction(...));

//V-cycle stage 4
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeSYMGS(...));

//V-cycle stage 5
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeProlongation(...)
| ComputeSYMGS(...));

//V-cycle stage 6
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeProlongation(...)
| ComputeSYMGS(...));

//V-cycle stage 7
stdexec::sync_wait(stdexec::schedule(scheduler)
| ComputeProlongation(...)
| ComputeSYMGS(...));
```

Listing 4.3: Structure of code resulting from unrolling the `ComputeMG` algorithm and composing with sender adaptors. The code is schematic: in practice, the senders for each V-cycle stage are preconstructed and executed repeatedly (multi-shot) inside the CG loop. Listing E.7 shows the sender for the first stage.

4.2 `stdexec` Dot Product Kernel Implementation

One of the kernels in the HPCG benchmark is dot product computation. However there is currently no sender adaptor within the NVIDIA `stdexec` library which performs reduction operations. Listing 4.4 shows an implementation which used a `std::atomic<double>` variable which all threads would add to. This solution was much slower than the baseline kernel, likely resulting from thread contention for the atomic variable. Using `std::transform_reduce` from the `stdpar` library (see Listing 4.5) was also experimented. When generalising the kernel for CPU and GPU execution, a bin-based, two-stage reduction scheme was developed (Listing 4.6). While it may be subject to future refinement, this method functions reliably and is used in the current implementation.

```
std::atomic<double> dot_local_result(0.0);

#define COMPUTE_DOT_PRODUCT(VEC1, VEC2, RESULT) \
stdexec::then([&](){ dot_local_result = 0.0; }) \
| stdexec::bulk(par, nrow, [&](local_int_t i){ \
  dot_local_result.fetch_add((VEC1).values[i]*(VEC2).values[i], \
    std::memory_order_relaxed); }) \
| stdexec::then([&](){ (RESULT) = dot_local_result.load(); })
```

Listing 4.4: Initial dot product implementation using `std::atomic<double>` variable.
(code from earlier development stages implemented the kernel as a preprocessor macro)

```
#define COMPUTE_DOT_PRODUCT(VEC1VALS, VEC2VALS, RESULT) \
stdexec::then([&](){ \
  (RESULT) = std::transform_reduce(std::execution::par, \
    (VEC1VALS), (VEC1VALS) + nrow, (VEC2VALS), 0.0); \
})
```

Listing 4.5: Dot product kernel using `std::transform_reduce`.
(code from earlier development stages implemented the kernel as a preprocessor macro)

```
#define NUM_BINS 1000

auto dot_prod_stg1 = [=](local_int_t i, const double * const vec1_vals,
  const double * const vec2_vals){

  local_int_t minInd = i*(nrow/NUM_BINS);
  local_int_t maxInd = ((i + 1) == NUM_BINS) ? nrow : (i + 1)*(nrow/NUM_BINS);
  double bin_sum = 0.0;
  for(local_int_t j = minInd; j < maxInd; ++j){
    bin_sum = std::fma(vec1_vals[j], vec2_vals[j], bin_sum);
  }
  bin_vals[i] = bin_sum;
};

auto dot_prod_stg2 = [=](double *result){

  double result_cpy = 0.0;
  for(local_int_t i = 0; i < NUM_BINS; ++i) result_cpy += bin_vals[i];
  *result = result_cpy;
};
```

Listing 4.6: Dot product implementation using bin-based, two-stage reduction scheme.

4.3 Parallelisation of Symmetric Gauss Seidel Algorithm

4.3.1 Parallelisation Methodology

The symmetric Gauss-Seidel (SYMGS) kernel in the baseline HPCG implementation is serial. GPU offloading of the HPCG workload involved using a parallel SYMGS algorithm.

A multicolouring approach was used to parallelise the SYMGS algorithm. The SYMGS algorithm is inherently sequential resulting from data dependencies in its equations. For convenience, Equations 3.13 and 3.14 are reproduced below:

$$\xi_i^{(f)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(f)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k)}\right)$$

$$\xi_i^{(k+1)} = \frac{1}{a_{ii}}\left(\beta_i - \sum_{j=1}^{i-1} a_{ij}\xi_j^{(f)} - \sum_{j=i+1}^{n} a_{ij}\xi_j^{(k+1)}\right)$$

New values of the element $\xi_i^{(f)}$ depend upon all previous new values $\xi_j^{(f)}$ where $1 < j < i$. Likewise, new values of $\xi_i^{(k+1)}$ depend upon $\xi_j^{(k+1)}$ for $i < j \leq n$ when the backwards sweep is performed. These data dependencies prevent straightforward parallelisation of the SYMGS algorithm. Multicolouring partitions the computational domain into multiple sets called colours. The elements belonging to a given colour have no dependencies relative to each other and may therefore be performed in parallel.

An example of multicolouring is red-black colouring. When discretising a partial differential equation onto a 2D computational grid, a five-point stencil (see Figure 4.1) is commonly used and results in each grid point producing four numerical equations corresponding to its four neighbours within the stencil. Each point therefore has at most four data dependencies. An algorithm operating on all points in the computational domain and having data dependencies between points can be split into two sets as shown by the red and black colours of Figure 4.1. The black points have no relative dependencies and are performed in parallel. Likewise for the red points.

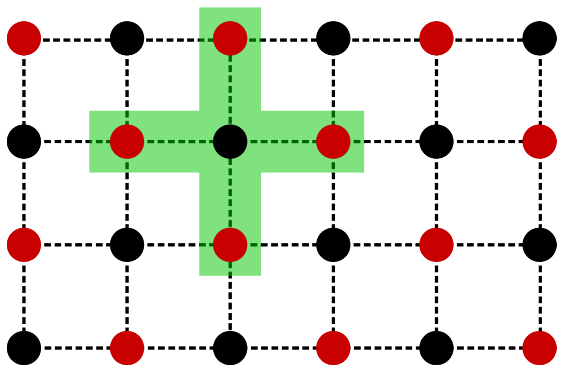


Figure 4.1: Illustration of red-black colouring applied to (6×4) 2D grid. The five-point stencil is shown in green.

HPCG uses a 27-point stencil (see Figure 4.2) and red-black colouring is therefore unsuitable. Instead, 8-colouring is used. For a grid point whose Cartesian coordinates are (i, j, k), the point is assigned a colour c according to:

$$c(i, j, k) = (i \bmod 2) + 2(j \bmod 2) + 4(k \bmod 2) \tag{4.1}$$

Equation 4.1 produces values $0 \leq c \leq 7$ and ensures no two neighbouring points in the 27-point stencil have the same colour. In Equations 3.13 and 3.14, all points of the same colour have $a_{ij} = 0$. When the points of a given colour are being updated using those equations, the contribution of updated values ($\xi_j^{(f)}$ in 3.13 and $\xi_j^{(k+1)}$ in 3.14) remains constant. The behaviour of the multicolored SYMGS therefore avoids data races and is deterministic.

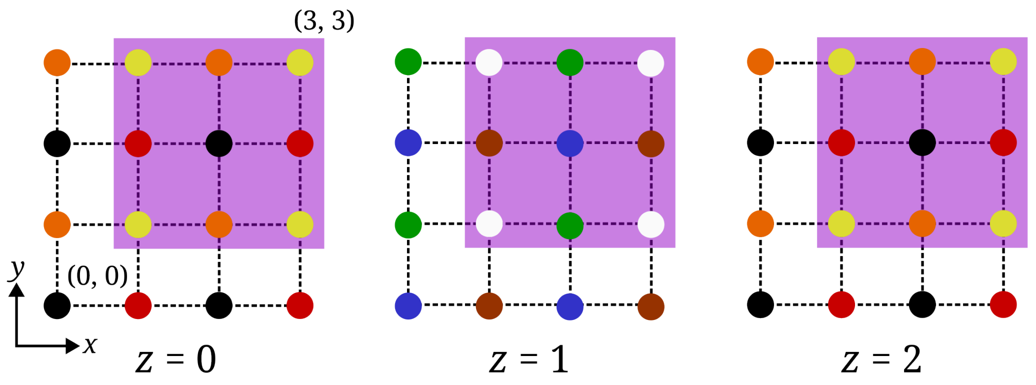


Figure 4.2: Illustration of 8-colouring applied to $(4 \times 4 \times 4)$ 3D grid. The 27-point stencil is shown in purple. Each point has a colour assigned by Equation 4.1.

4.3.2 Convergence Characteristics

Multicoloured SYMGS is only an approximation of sequential SYMGS because the usage of updated and un-updated values is different from the sequential algorithm. Because multicoloured SYMGS only approximates sequential SYMGS, more iterations are generally required to achieve convergence with CG compared to if sequential SYMGS was used.

In HPCG, a grid point of Cartesian dimensions (i, j, k) is placed in row n_{row} of matrix A where:

$$n_{\text{row}} = i + jN_x + k(N_x N_y) \tag{4.2}$$

N_x and N_y are the grid dimensions in the x and y directions. Since sequential SYMGS goes through all rows in order, when updating a particular grid point P, all points of the 27-point stencil before P in the row ordering will already have been updated. This results in 13 updated points and 13 un-updated points in the stencil. In contrast, how many updated and un-updated neighbouring points a grid point has with multicolouring depends on how many colours have been updated before. For example, points of the very first colour to be updated will use only un-updated values, whereas points of the very last colour to be updated will use only updated values.

The parallel SYMGS kernel made the CG algorithm perform 60 iterations to solve the sparse linear system within the HPCG benchmark[4]. In comparison the baseline (serial) SYMGS kernel generally needed 50 iterations to solve the same problem. Figure 4.3 shows the serial and parallel convergence behaviour for a single CG solve.

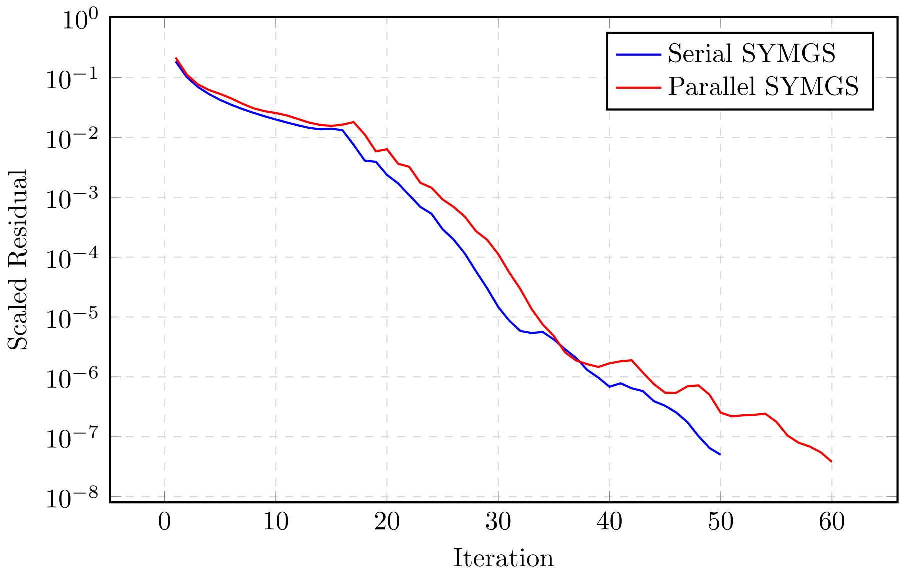


Figure 4.3: Convergence of CG algorithm with serial and parallel SYMGS for a typical pair of runs using default problem size of 104 × 104 × 104. CG with serial SYMGS generally took 50 iterations whereas with parallel SYMGS, 60 iterations were typically required. Since the convergence behaviour illustrated is independent of the particular hardware or algorithm implementation used, those details are not included here.

[4]When solving a problem of dimensions 104 × 104 × 104, which is the default HPCG problem size. Larger dimensions require more iterations to solve.

4.3.3 Implementation Details

The baseline (serial) SYMGS kernel has a forwards and backwards sweep. Each sweep has n iterations where n is the number of rows in the sparse matrix A. The `stdpar` parallel SYMGS algorithm decomposed each sweep into eight parts. Each part calculated new values for the rows belonging to a given colour in parallel. This was implemented by repeating each sweep eight times and only selecting the rows of the selected colour each time (see Listing 4.7) similar to the approach taken by Kumahata et al. [14]. The forwards and backwards sweeps of the serial kernel are performed in ascending ($i = 0, 1, 2 \dots n_{\text{row}} - 1$) and descending ($i = n_{\text{row}} - 1, n_{\text{row}} - 2 \dots 0$) order respectively. The forwards and backwards sweeps of the parallel SYMGS kernel both traversed the colours in the same (ascending) order ($color = 0, 1 \dots 7$) since the order in which colours were handled did not matter. This meant the forwards and backwards sweeps of the parallel kernel had identical code. To minimise code repetition, the forwards and backwards sweeps were combined into one sweep repeated twice.

```
for(int sweep = 1; sweep <= 2; sweep++){
  for(int color = 0; color < NUM_COLORS; color++){
    std::for_each(std::execution::par_unseq, rows.begin(), rows.end(), [](int i){
      if(colors[i] == color){
        //do the SYMGS update for row i
      }
    });
  }
}
```

Listing 4.7: Simplified code illustrating the structure of `stdpar` parallel SYMGS. Eight passes are performed (corresponding to the eight colours) across all rows of `A`, using iterators `rows.begin()` and `rows.end()`. When a given row has the desired colour its value is updated. The overall process is repeated twice resembling a forwards and backwards sweep.

Following reasoning similar to the `stdpar` parallel SYMGS implementation, the `stdexec` implementation had eight forward sweeps (one for each colour) and eight backward sweeps. Since the order in which colours were updated did not matter, the forward and backward sweeps traversed the colours in the same (ascending) order. The resulting arrangement was composed of 16 sender adaptors piped together (Listing 4.8). Using a `for` loop to avoid code repetition and give more concise code was avoided because this would have introduced a synchronisation point (through `sync_wait`) once per loop and may have worsened performance. The `repeat_n` sender adaptor was also trialled but proved detrimental to performance.

```
stdexec::sync_wait(stdexec::schedule(scheduler)
| SYMGS(color = 0) | SYMGS(color = 1)
| SYMGS(color = 2) | SYMGS(color = 3)
| SYMGS(color = 4) | SYMGS(color = 5)
| SYMGS(color = 6) | SYMGS(color = 7)
| SYMGS(color = 0) | SYMGS(color = 1)
| SYMGS(color = 2) | SYMGS(color = 3)
| SYMGS(color = 4) | SYMGS(color = 5)
| SYMGS(color = 6) | SYMGS(color = 7));
```

Listing 4.8: Illustrative pseudocode showing structure of parallel SYMGS pipeline in `stdexec` HPCG implementation. In practice, each `SYMGS` sender adaptor is followed by a `stdexec::then` invocation to update the `color` value, resulting in 32 sender adaptors (Listing E.7 provides a full example for the first V-cycle stage).

Another code configuration was created with baseline HPCG code using the parallel SYMGS algorithm instead of the serial SYMGS algorithm. The only change to the baseline code was calling `ComputeSYMGS_par.cpp` rather than `ComputeSYMGS_ref.cpp`.

4.4 Memory Layout Optimisation

Memory Layout Optimisation

Because rows of a given colour are generally not adjacent in matrix A, parallel SYMGS introduces a more discontinuous memory access pattern compared to serial SYMGS. Cache performance can be negatively impacted due to reduced spatial locality. To provide continuous memory access, contiguous blocks of memory were allocated for both the matrix values of A and the column indexes of those values (see Listing 4.9). Rows of A were then assigned to subregions of these two contiguous blocks. The rows were assigned in order of colour, maintaining the relative ordering of rows within each color group as they appeared in the original matrix. Figure 4.4 illustrates this process.

```
//allocate memory in large blocks
local_int_t *tmpIndL = new local_int_t[total_nnz];
double *tmpVals = new double[total_nnz];

//assign memory in colour-order
for(int color = 0; color < NUM_COLORS; color++){
  for (local_int_t i = 0; i < localNumberOfRows; i++){
    if(A.colors[i] == color){
      mtxIndL[i] = tmpIndL;
      matrixValues[i] = tmpVals;
      tmpIndL += numberOfNonzerosPerRow;
      tmpVals += numberOfNonzerosPerRow;
    }
  }
}
```

Listing 4.9: Simplified code illustrating the colour-optimised memory layout policy used with parallel SYMGS. Blocks of contiguous memory are allocated for `total_nnz` elements of `A` and their associated column index values, accessible by pointers `tmpVals` and `tmpIndL`. Assignment of rows of `A` to subregions of these blocks then occurs in passes. `matrixValues[i]` and `mtxIndL[i]` are pointers to arrays containing the elements and column index values respectively of row `i`.

This scheme was included within `GenerateProblem_ref.cpp` as an alternative to the default memory allocation policy, selectable with the compiler flag `-DPARALLEL_SYMGS`. Because `GenerateProblem` was also used when creating the coarse representations of A, the optimised memory allocation policy applied to all levels of the multigrid hierarchy.

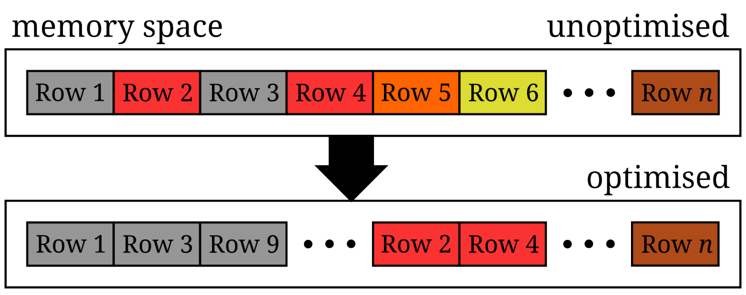


Figure 4.4: Visualisation of memory layout for data of `SparseMatrix A` with and without colour-based optimisation.

Chapter 5

Results and Discussion

5.1 Absolute Runtime on Reference Platform

The value of absolute runtime is clear: shorter runtimes enable more simulations, faster results, and reduced costs. Figure 5.1 shows average runtimes for a single CG solve with `stdpar`, `stdexec` and baseline implementations. The runs were performed on a machine with a 12-core AMD Ryzen 9 CPU and RTX 3060 GPU*[1]. Although this hardware is consumer-grade, parallel programming models are increasingly intended to support accessible, general-purpose heterogeneous systems, not only supercomputers. Using such hardware therefore provides insight into the utility of such models for the broader HPC and general computing communities.

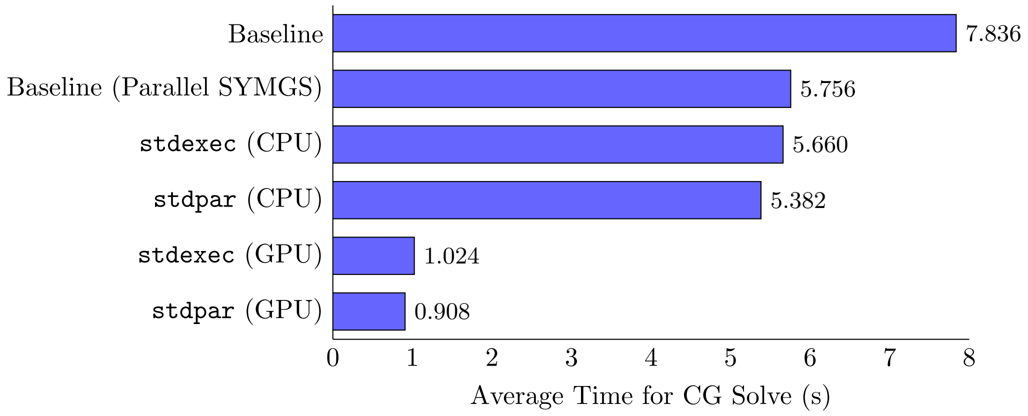


Figure 5.1: Average time for CG solve when performing 10 back-to-back solves at default problem size ($104 \times 104 \times 104$). Performed on remote development machine (AMD Ryzen 9 3900X 12-core CPU with 64GB RAM, NVIDIA RTX 3060 GPU with 12GB RAM*).

The three CPU-based implementations using parallel SYMGS (`stdpar` CPU, `stdexec` CPU and baseline

[1]Some results were obtained using cloud-based computing instances outside of those from the University of Bristol clusters. The hardware used has been stated in each figure caption and is marked with an asterisk (*) when this is the case.

with parallel SYMGS) give similar runtimes falling within $\sim$7% of each other. This is expected since they mostly implement equivalent kernels (for example, the SPMV kernel follows the same algorithmic steps across all three implementations) and are executed on the same CPU target. The `stdexec` code is $\sim$5% slower than `stdpar` and $\sim$2% faster than the baseline with parallel SYMGS.

The `stdexec` GPU code runs $\sim$12% slower than the `stdpar` GPU code. These GPU implementations use the same source code as their CPU counterparts. The `stdpar` GPU implementation is $\sim$5.9 times faster than its CPU counterpart; `stdexec` GPU is $\sim$5.5 times faster than `stdexec` CPU. Given that HPCG is predominantly memory-bound, the maximum theoretical memory bandwidths of the CPU and GPU provide a simple basis for estimating the expected speedup of the GPU over the CPU. For the CPU, this may be calculated by:

$$\text{Bandwidth} = \text{Data Rate (MT/s)} \times \text{Bytes per transfer} \times \text{Number of channels}$$

Using values of 3200 MT/s (for DDR4-3200 RAM), 8 bytes per transfer and 2 channels gives 51.2 GB/s. For the GPU, the formula is:

$$\text{Bandwidth} = \text{Data Rate (Gb/s per pin)} \times \text{Bus width (bits)} \div 8$$

Using values of 15 Gb/s (GDDR6 effective rate per pin) and 192 bits bus width gives 360 GB/s. Diving the two gives a theoretical speedup of $\sim$7. While the theoretical GPU bandwidth is 360 GB/s, the system vendor reported an effective bandwidth of 317.7 GB/s for the specific machine. Using this value instead gives an expected speedup of $\sim$6.

5.2 Kernel Runtimes

To gain a clearer understanding of the differences between the implementations, the execution times of the kernels have been examined. This analysis can reveal whether particular kernels perform especially well or poorly on certain implementations or target systems. This section begins with an analysis of the kernel breakdown for the baseline code, followed by the results for the CPU parallel SYMGS implementations. The discussion then turns to the GPU implementations. To ensure consistency and enable meaningful cross-comparisons, all kernel data presented in this section was collected on the same development machine[2].

Figure 5.2 presents the kernel breakdown for the baseline code. The runtime is dominated by SYMGS, which accounts for 81.0% of the total execution time, as expected given its serial nature. The next most expensive kernel is SPMV, contributing 17.5%.

[2]AMD Ryzen 9 3900X 12-core CPU with 64GB RAM, NVIDIA RTX 3060 GPU with 12GB RAM*

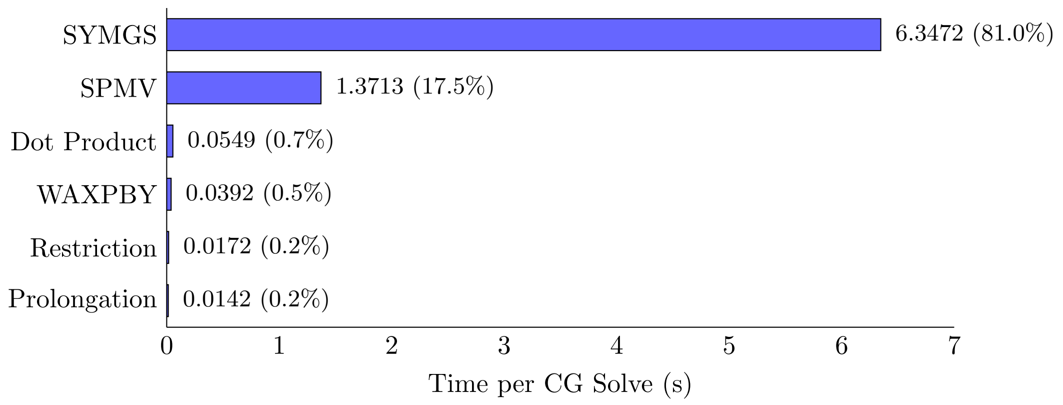


Figure 5.2: Time spent per kernel for a typical CG solve using the baseline code with default problem size (104 × 104 × 104). Percentages shown in brackets.

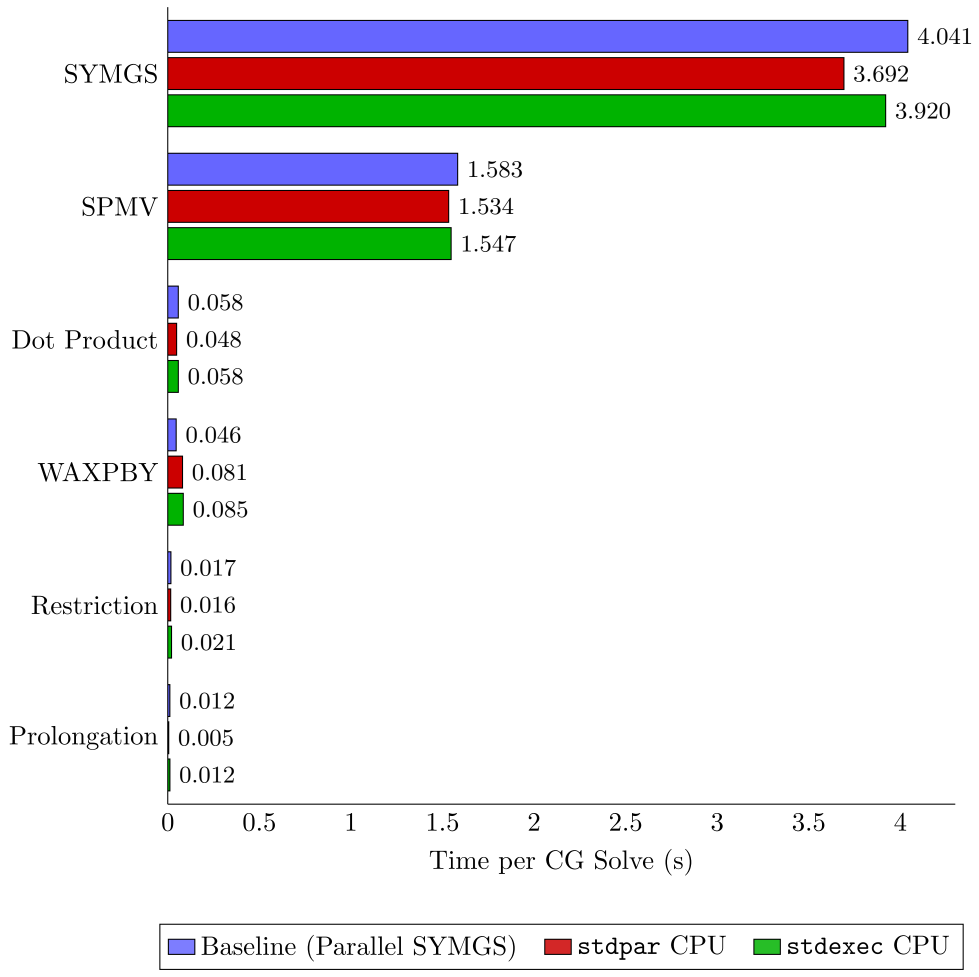


Figure 5.3: Time spent per kernel for CPU implementations using parallel SYMGS on a typical CG solve with default problem size (104 × 104 × 104).

Figure 5.3 presents the kernel runtime breakdowns for the CPU implementations using parallel SYMGS. Compared with Figure 5.2, the SYMGS kernel is less dominant, accounting for ∼70% of the total solve time.

The other kernels exhibit similar orders of magnitude across all implementations. Notably, the `stdpar` version of SYMGS is ∼9% faster than the baseline OpenMP implementation and ∼6% faster than the `stdexec` implementation.

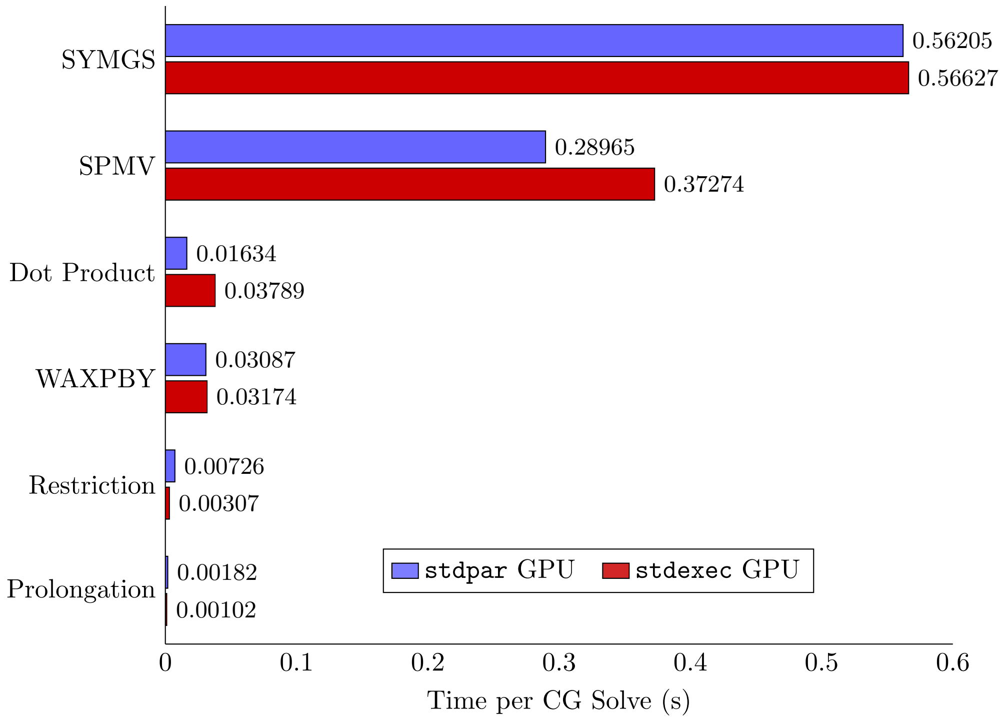


Figure 5.4: Time spent per kernel for GPU implementations on a typical CG solve with default problem size $(104 \times 104 \times 104)$.

Figure 5.4 presents the kernel-level runtime breakdowns for the GPU implementations. Overall, the distributions are quite similar, with two notable exceptions: SPMV and dot product. For SPMV, the `stdpar` implementation is ∼25% faster than `stdexec`, despite the two kernels containing nearly identical code and following the same algorithmic steps. The source of this discrepancy is not yet understood and warrants further investigation. In the case of dot product, the `stdpar` kernel achieves ∼2.3 faster performance than `stdexec`. This difference can be explained by implementation choices: `stdpar` leverages the optimized `std::transform_reduce` algorithm, whereas `stdexec` relies on a user-defined routine (see Section 4.2), which is likely suboptimal. A performance penalty in the latter case is therefore not unexpected.

Since NVIDIA NVTX markers cannot be used within GPU-executed senders, and since `nsys` presented the kernel wrapper functions of `CG_stdexec` (see Listing G.1) only under stripped, type-based signatures, the task of mapping profiler-reported kernels back to their corresponding source code names was a manual and cumbersome process. During this analysis, it was observed that a `stdexec::then` invocation executing 59 times within the main CG loop accounted for ∼0.3% of total kernel runtime. Further inspection revealed this invocation corresponded to `then([=](){ *oldrtz = *rtz; })`. Given the simplicity of this step, and the presence of similar operations in the loop that incurred negligible overhead, its relative cost was unexpected. Further investigation is warranted, though its position as the first sender adaptor in the main CG loop's pipeline may be a contributing factor.

5.3 Scaling

This section presents the results of a strong-scaling test and a test where the problem size has been increased on a machine with fixed computational resources.

Figure 5.5 presents the results of a strong-scaling experiment for the three CPU-based parallel SYMGS implementations. As is typical with HPCG, the curves exhibit memory-bound behavior, reaching peak speedup at around 8 threads before stagnating due to additional overheads at higher thread counts. At 8 threads, the difference between the highest speedup (2.97 for `stdpar`) and the lowest (2.57 for the baseline) is ∼14%, which is significant. The baseline implementation relies on the default (serial) versions of the `CopyVector` and `ZeroVector` functions, which explains its consistently lower performance relative to `stdpar` and `stdexec`. The `stdexec` implementation may also incur additional serial-like overheads from launching and managing sender pipelines, although this alone does not fully account for its deviation from `stdpar`. Another factor may be the user-designed reduction strategy in the `stdexec` dot product kernel (Section 4.2), which employs a fixed number of bins and may scale poorly, thereby further limiting performance.

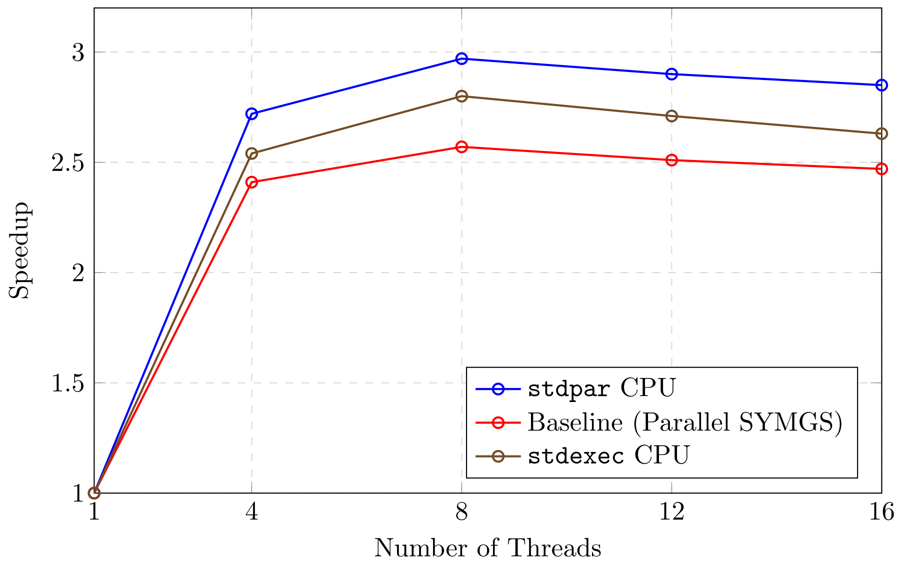


Figure 5.5: Strong-scaling speedup for CPU implementations using parallel SYMGS. Speedup is $T(1)/T(p)$ where $T(1)$ is the runtime with 1 thread and $T(p)$ is the runtime with p threads. Default problem size ($104 \times 104 \times 104$) on development machine (AMD Ryzen 9 5950X 16-Core CPU with 128GB RAM*).

A rough estimate of HPCG's memory bandwidth usage can be made as follows. Since the sparse matrix `A` is the dominant data structure in terms of memory, the operations that traverse `A` account for the bulk of bandwidth consumption. These operations are SYMGS and SPMV. In each CG iteration, two SYMGS calls and two SPMV calls are executed. The SPMV kernel traverses the values of `A` once, while SYMGS traverses them twice (once forward and once backward). Consequently, the values of `A` are accessed approximately six times per iteration. Most of the data associated with `A` resides in the `mtxIndL` and `matrixValues` structures, which store the column indices and matrix values, respectively. The total memory traffic per CG solve can therefore be estimated as:

$$\text{Data (bytes)} = \text{(No. of nodes)} \times \text{(Non-zeros per row)} \times \text{(Bytes per value)} \times 6 \times 2 \times \text{(Iterations per solve)}$$

Substituting 104^3 nodes, 27 non-zeros per row, 8 bytes per value and 60 iterations per solve yields a total memory usage of ~175 GB. The single-thread run on the AMD Ryzen machine shown in Figure 5.5 completed (on average) in ~12.5 seconds, corresponding to an effective per-thread bandwidth of ~14.0 GB/s. Given that the Ryzen CPU has a peak bandwidth of 51 GB/s, saturation would occur after ~3.64 threads if bandwidth scaled linearly with thread count. This calculation is, of course, a simplified approximation that ignores many architectural factors and relies on several assumptions (notably, cache will significantly reduce the memory traffic giving a higher saturation thread count). Nevertheless, it serves to illustrate the substantial memory demands of HPCG and highlights how quickly the available CPU bandwidth can become saturated as the thread count increases.

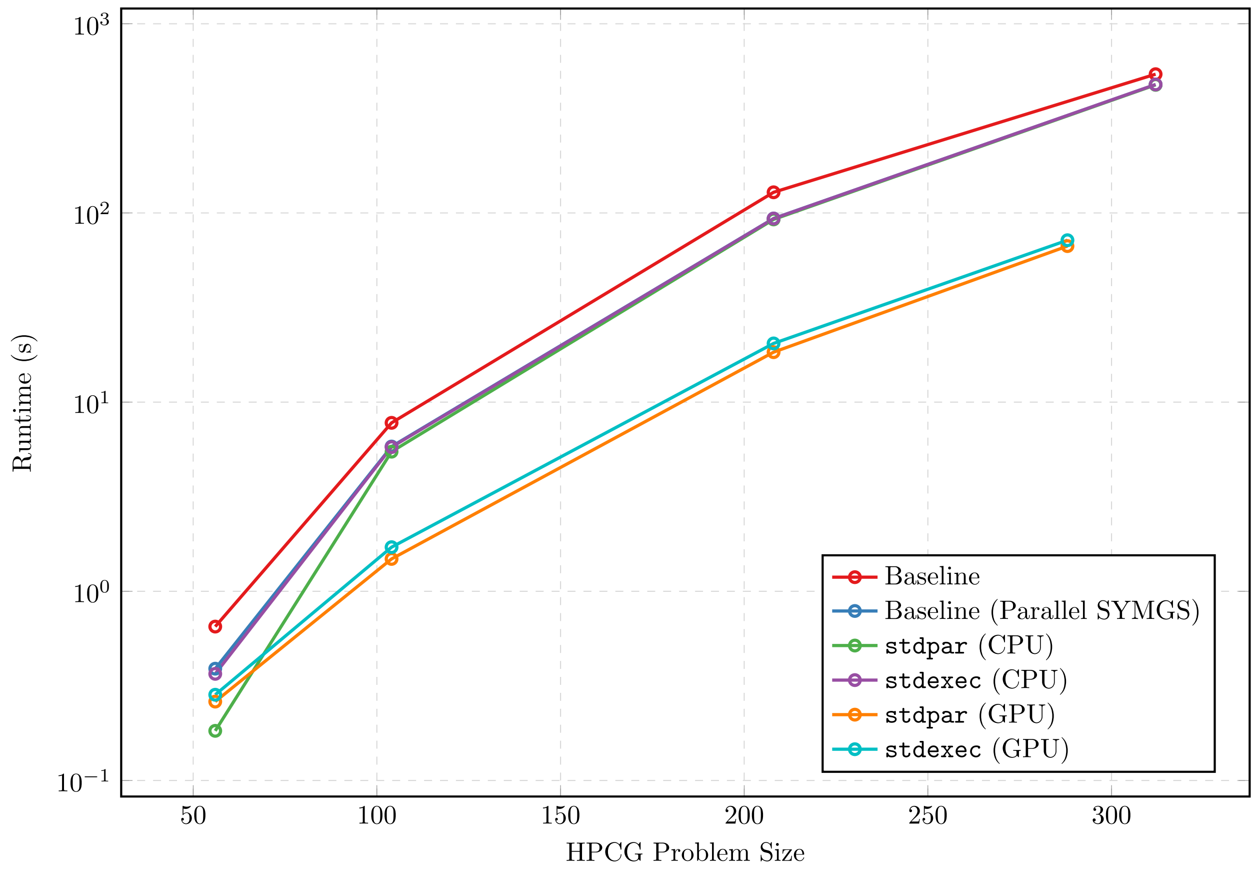


Figure 5.6: Typical runtime for one CG solve with varying problem size. Problem sizes of 56 (i.e. $56 \times 56 \times 56$), 104 and 208 were performed on both CPU and GPU. 312 was performed on CPU but consistently failed on GPU, likely due to an exhaustion of device resources; 288 was performed on GPU instead. Data collected on remote development machine (AMD Ryzen 9 3900X 12-core CPU with 64GB RAM, NVIDIA RTX 3060 GPU with 12GB RAM*).

Figure 5.6 shows how the runtimes for the implementations vary with problem size. The three CPU-based parallel SYMGS implementations (`stdpar` CPU, `stdexec` CPU and baseline with parallel SYMGS) broadly deliver similar performance, except for small problem sizes where `stdpar` CPU outperforms all implementations including the GPU implementations. With smaller problem size, the GPU implementations give smaller speedups (30-50%) against the CPU implementations, compared to larger problem sizes where the speedup is greater (typically 3-5 times faster). The performance difference between the `stdexec` and `stdpar` GPU implementations is consistent across all problem sizes meaning the additional overhead of `stdexec` relative to `stdpar` scales with the problem.

Increasing problem size produces a substantial increase in computational work since both number of nodes and number of CG iterations increase. For example, the doubling of problem size from 104 to 208 increases the number of nodes by $2^3 = 8$ and also the number of iterations from 60 to 103. The increase in computational work (and therefore runtime) may, to a first approximation, be estimated as $\frac{103}{60} \times 8 = 13.7$.

5.4 Heterogeneous Programming

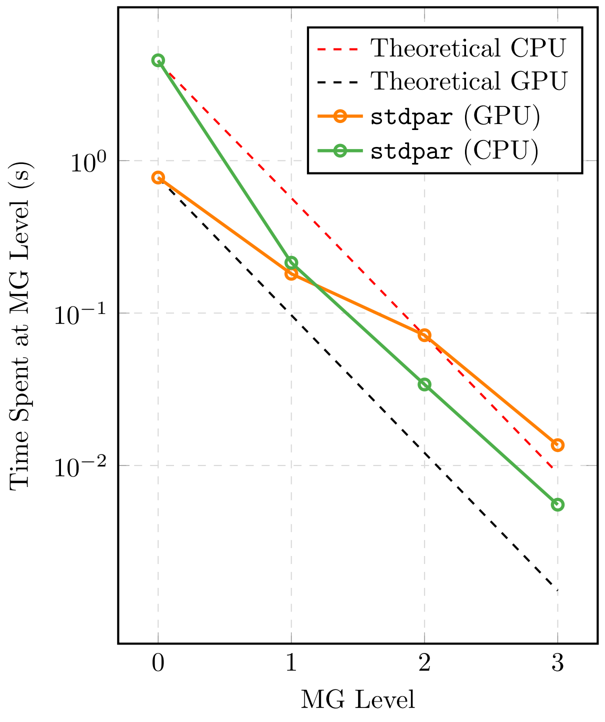
The ability to compile `stdpar` and `stdexec` codes for both CPU and GPU execution enables heterogeneous programming. In the case of `stdexec`, this is particularly convenient, as senders can be redirected to different schedulers (each representing a distinct execution resource) using `stdexec::continues_on`. While this section does not present experiments with fully heterogeneous code, it examines the multigrid preconditioning process as a case study to assess the feasibility of switching from GPU to CPU execution at higher multigrid levels.

A known strategy [27] for improving performance of GPU-offloaded MG preconditioning is to execute the coarsest levels on the CPU. For memory-bound applications like HPCG, the GPU hides memory access latency by switching execution between warps (NVIDIA) or wavefronts (AMD), allowing some warps to proceed while others wait for data. With workloads that are too small to provide sufficient active warps, this mechanism degrades and memory stalls negatively impact performance. Occupancy is defined as the ratio of active warps on a Streaming Multiprocessor (SM) to the maximum number of active warps supported by the SM[3]. The coarsest levels of MG preconditioning have small numbers of parallel threads and are therefore likely low-occupancy.

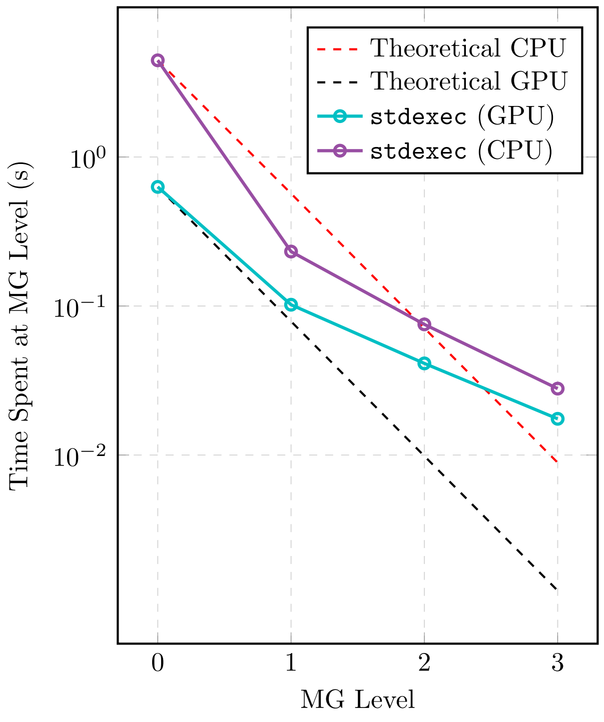
Figure 5.7 shows times spent at each MG level (which include all kernels executed at these levels) for the `stdpar` and `stdexec` CPU and GPU implementations. Figure 5.7a (left) shows the `stdpar` times while 5.7b (right) shows the `stdexec` times. Since the number of grid points decreases by a factor of eight at each coarser MG level, a rough expectation is that the time spent would also decrease by a factor of eight with each transition to a coarser level. In Figure 5.7a, this expectation is shown by the black and red dashed lines, which begin at the level 0 values of `stdpar` GPU and `stdpar` CPU respectively and consistently decrease by a factor of eight. As expected, `stdpar` CPU time (green) seems to follow (and even outperform) the equivalent theoretical line (red). The `stdpar` GPU line fails to follow the theoretical (black) line, suggesting low occupancy might be degrading performance at higher MG levels. This suggests that switching to the CPU for higher levels of MG preconditioning may yield performance benefits on an otherwise GPU-executed implementation.

[3]As defined by NVIDIA since definitions vary.

In 5.7b, the `stdexec` GPU line (cyan) departs from its theoretical prediction (black), consistent with a low-occupancy performance degradation effect. However, the CPU line (purple) also deviates from its theoretical reference (red), thereby eliminating the expected crossover between the GPU and CPU curves. One possible contributing factor could be overheads introduced by the sender execution model, though the precise cause remains unclear and requires further investigation. This result also suggests that switching to a CPU-based scheduler for coarser MG levels does not yield performance benefits; experiments performed to this effect (though not presented here in detail) confirmed this finding.



(a) `stdpar`



(b) `stdexec`

Figure 5.7: Average time spent at each level during MG preconditioning per CG solve (default size $104 \times 104 \times 104$). Data collected on remote development machine (AMD Ryzen 9 3900X 12-core CPU with 64GB RAM, NVIDIA RTX 3060 GPU with 12GB RAM*.)

On another note, the `stdpar` CPU line (green) shows a much steeper gradient when transitioning between levels 0 and 1 compared to transitions between the other levels. This may be caused by a regime change in the cache utilisation of the processor. The data structure which dominates memory usage is the sparse matrix `A`. Its memory footprint may be roughly calculated by:

$$\text{Size of A (bytes)} = (\text{No. of nodes}) \times (\text{Non-zeros per row}) \times (\text{Bytes per value}) \times 2$$

The factor of two accounts for the two arrays `mtxIndL` and `matrixValues` within `A`. For level 0 (104^3 nodes, 27 non-zeros per row and 8 bytes per value), `A` occupies ~486 MB. At level 1, the number of nodes decreases to 52^3 and the size of `A` becomes ~61MB. Given that the AMD Ryzen 9 3900X CPU provides a total of 64 MB of L3 cache, it is plausible that the program is able to retain the majority of its working data within this cache level, resulting in the pronounced reduction in runtime observed between levels 0 and 1. Similar cache regime transitions have been reported by other researchers, with a notable example provided in [24].

5.5 Portability

Code portability (the ability to compile and run code on many systems) is essential in ensuring applications run on successive generations of computers and diverse hardware architectures. Since both the `stdexec` and `stdpar` versions use standard C++, they can be compiled and executed with any sufficiently recent C++ implementation. This contrasts with OpenMP (used by the baseline), which is widely supported by C++ compilers but not universally guaranteed.

However, portability alone is insufficient — code must also execute efficiently on the systems to which it is portable. Pennycook et al. [22] define this property as **performance portability**, which they describe as follows:

“A measurement of an application’s performance efficiency for a given problem that can be executed correctly on all platforms in a given set.”

Pennycook also defines an associated performance portability metric $\mathbf{\Phi}$. For a set of platforms H (representing all computer systems over which code is executed), $\mathbf{\Phi}$ is:

$$\mathbf{\Phi}(a, p, H) = \frac{|H|}{\sum_{i \in H} \frac{1}{e_i(a,p)}}$$

An implementation a which solves some problem p on one the i^{th} platform of the set has performance efficiency $e_i(a, p)$. Although performance efficiency can be derived in two ways (architectural efficiency and application efficiency), here the application efficiency is used. Application efficiency is the performance of a given implementation on given a platform as a percentage of the best known implementation for that platform [21]. In this case, this means:

$$e_i(a, p) = \frac{T_{min}}{T}$$

For the given platform, T is the runtime of the chosen implementation and T_{min} is the runtime of the fastest known implementation.

Two important distinctions are made from Pennycook’s methodology. Firstly, the same application or executable is not transferred and run on many platforms, but rather the same source code is compiled and run (by the same compiler) on different systems. Secondly, Pennycook defines T_{min} as the performance of the fastest implementation ever produced for a given problem. In this work, however, T_{min} is taken to be the fastest among the three implementations considered — `stdpar`, `stdexec`, and the baseline (where applicable). Consequently, performance portability ($\mathbf{\Phi}$) is treated here as a relative metric between the implementations evaluated in this thesis, rather than as an absolute metric based on the globally fastest known implementations.

Table 5.1 shows the application efficiencies of the CPU codes implementing parallel SYMGS on four CPU platforms, while 5.2 shows the efficiencies of the GPU codes on four GPU platforms. Φ values for the `stdpar` CPU, `stdexec` CPU and baseline (parallel SYMGS) codes are 1.000, 0.811 and 0.756 respectively, while those for the `stdpar` GPU and `stdexec` GPU codes are 0.997 and 0.963 respectively. For the CPU codes, `stdpar` is always fastest. Furthermore, on the AMD EPYC platform, the splitting of memory into four NUMA regions caused all data to be placed onto one region by first-touch, giving large slowdowns. Since changing the codes to rectify this issue was non-trivial, the codes were run using `numactl --interleave=all`, which evenly distributed the data among all NUMA regions. This modification appeared to have a much greater impact on the `stdexec` and baseline implementations than on the `stdpar` implementation, shown by the efficiency values of 0.560 and 0.460. However, for GPU execution the performance difference between `stdpar` and `stdexec` is less pronounced, as shown by the `stdexec` Φ value of 0.963.

	Runtime (s)			**Application Efficiency**		
Platform	`stdpar`	`stdexec`	**Baseline**	`stdpar`	`stdexec`	**Baseline**
AMD Ryzen 9 3900X 64GB*	5.382	5.660	5.756	1.000	0.951	0.935
AMD EPYC 7543P 256GB	0.801	1.431	1.740	1.000	0.560	0.460
AMD Ryzen 9 5950X 64GB*	4.559	4.839	4.740	1.000	0.942	0.962
Intel i5-11400 64GB*	5.564	5.756	5.627	1.000	0.967	0.989

Table 5.1: Execution times and application efficiencies for the `stdpar` CPU, `stdexec` CPU and baseline (parallel SYMGS) implementations across four platforms.

	Runtime (s)		**App. Efficiency**	
Platform	`stdpar`	`stdexec`	`stdpar`	`stdexec`
NVIDIA RTX 3060 12GB*	0.908	1.024	1.000	0.887
NVIDIA H100 PCIe 80GB	0.382	0.391	1.000	0.977
NVIDIA A100 SXM4 40GB	0.463	0.464	1.000	0.998
NVIDIA RTX 4090 24GB*	0.394	0.390	0.990	1.000

Table 5.2: Execution times and application efficiencies for GPU-executed implementations across four platforms.

Code portability can sometimes come at the expense of performance, particularly when targeting both CPU and GPU architectures. For example, Table 5.3 compares the CG solve times of the `stdpar` and `stdexec`

GPU implementations with an optimized HPCG GPU implementation developed by NVIDIA. The NVIDIA implementation achieves a speedup of approximately ∼2.9 relative to `stdpar` GPU. This comparison should be interpreted cautiously, as NVIDIA's implementation may employ different algorithms to perform the solve. Moreover, the aim of this project was not to develop a GPU HPCG implementation that outperforms NVIDIA's. Nonetheless, the result illustrates the substantial performance gains that can be realized when code is carefully optimized for a specific platform.

Implementation	Avg. Solve Time (s)
`stdpar` (GPU)	0.914
`stdexec` (GPU)	1.0254
NVIDIA HPCG	0.316

Table 5.3: Average CG solve time for `stdpar` GPU, `stdexec` GPU and NVIDIA implementations. Default problem size (104×104×104) used. NVIDIA implementation downloaded from the HPCG website with filename `xhpcg-3.1_cuda-11_ompi-4.0_sm_60_sm70_sm80`. Testing performed on remote development machine (Intel Xeon E5-2699 v4 CPU (11/44 cores) with 192GB RAM, NVIDIA RTX 2060S GPU with 8GB RAM*.)

5.6 Compiler and Toolchain Maturity

The final perspective from which we evaluate the `stdpar` and `stdexec` models is the maturity of their programming environments. Maturity is important because it reflects the ease with which developers can adopt these libraries in practice and therefore produce reliable benchmark results. This section begins by considering the execution of a simple test kernel with both `stdpar` and `stdexec` — and notes the difference in their performance. The discussion then examines the flexibility provided by `stdexec` for constructing and executing task graphs, and highlights how alternative design choices can lead to variations in performance. Finally, two issues which were identified with `stdexec` during development are detailed.

5.6.1 Comparing GPU Kernel Performance with `stdpar` and `stdexec`

A minimal GPU kernel (see Appendix I) was written to assess GPU offloading performance under the `stdpar` and `stdexec` approaches. For both `stdpar` and `stdexec`, the kernel was executed 100 times. The resulting runtimes were plotted as overlapping histograms for comparison (Figure 5.8). On average, the kernel executed roughly 10.1% slower with `stdexec` compared to `stdpar`. This observation could suggest reduced efficiency in GPU offloading; however, the evidence remains inconclusive without further investigation.

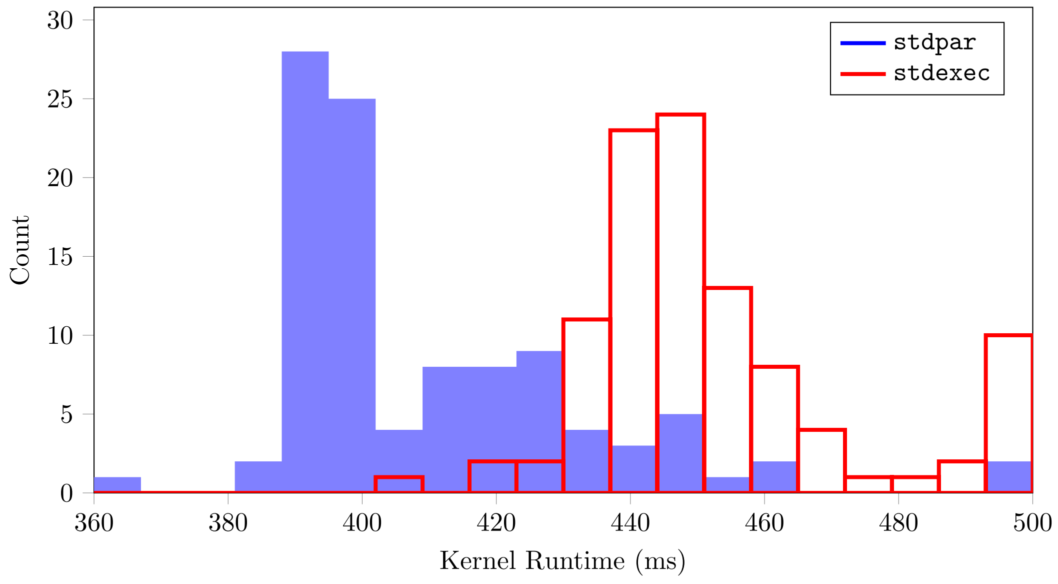


Figure 5.8: Runtime histograms for test GPU kernel with `stdpar` and `stdexec` execution. 100 kernel runs of each were performed in alternating sequence (`stdpar-stdexec-stdpar`...) to control against performance drift of the host system. Mean runtimes for `stdpar` and `stdexec` execution were 410.7ms and 454.5ms respectively giving a percentage difference of 10.1%. Standard deviations were 24.4ms and 25.1ms for `stdpar` and `stdexec` respectively. Data collected on Intel Xeon E5-2699 v4 CPU (11/44 cores) with 258GB RAM, NVIDIA RTX 2060S GPU with 8GB RAM*. Code for kernel given in Appendix I.

5.6.2 Impact of Sender Construction on Performance

Table 5.4 shows three iterations of the `stdexec` GPU code (plus the mature `stdpar` GPU code for comparison) and illustrates how different structuring of sender adaptor pipelines impacted performance. Commit 5f7f331 used an `exec::repeat_n` sender adaptor for SYMGS such that the 16 sender adaptors of Listing 4.8 could be reduced to a single, repeated sender adaptor. Since `repeat_n` only works on CPU, execution was switched to CPU before `repeat_n` in the pipeline. Changing `repeat_n` to a `for` loop with 16 `sync_wait`'s (commit a071165) reduced CG solve time from 2.944s to 1.858s. Then, commit 248d795 removed the `for` loop and made a long pipeline with 16 stages (Listing 4.8), as well as pre-constructing senders in the manner of Listing G.2. This further reduced solve time to 1.722s.

The performance differences among these three iterations are not straightforward to explain, nor are they easy to predict in advance. Since the performance characteristics of the sender-based abstractions provided by the `stdexec` library are not always transparent to the user, an initial `stdexec` implementation may perform poorly, requiring empirical optimisation through trial and error.

An interesting pattern was observed: less efficient implementations tended to produce executables with larger `.text` sections, despite (i) having similarly sized `.nv_fatbin` sections (which contain GPU instructions) and (ii) performing comparable GPU memory transfers (see CUDA H2D and D2H columns of Table 5.4). For these versions, CPU L1 data cache loads were also higher (see L1D column). These trends may suggest less efficient CPU-side performance, although the evidence is not conclusive.

Commit	Commit Summary	Avg. CG Solve (s)	`.text` Size (kB)	`.nv_fatbin` Size (MB)	CUDA U H2D (MB)	CUDA U D2H (MB)	L1D Loads
5f7f331	using `repeat_n`	2.944	790	2.08	1233	743.6	1.56e10
a071165	using `for` loops	1.858	610	1.94	1220	730.7	1.47e10
248d795	16 long pipeline + pre-constructed senders	1.722	548	2.21	1203	713.3	1.39e10
248d795	`stdpar` GPU code	1.473	376	0.85	1180	680.4	1.24e10

Table 5.4: File and runtime details for three commits during `stdexec` development; `stdpar` code is shown for reference. CG solve time is the average of 10 back-to-back solves. The `.text` and `.nv_fatbin` sections (measured with `bloaty`) contain CPU and GPU instructions, respectively. "H2D" and "D2H" denote host-to-device and device-to-host transfer within CUDA Unified Memory. "L1D" loads are accesses to the L1 data cache. Runtime statistics were collected with `nsys` and `perf` on an AMD Ryzen 9 3900X (12 cores, 64GB RAM) with an NVIDIA RTX 3060 (12GB RAM)*.

With many sender adaptor types available in `stdexec`, constructing pipelines that exhibit less-than-optimal performance is a common risk. For example, an approach trialled during development was to make each of the seven stages in multigrid preconditioning a distinct sender. NVIDIA defines `exec::variant_sender` which can be one of several different senders; the seven senders were then packaged into a single `variant_sender`. `stdexec::let_value` was then used to perform runtime selection of the appropriate member of the `variant_sender`. While in principle this reduced the sender pipeline length, it proved to be slow in practice and was not pursued further.

5.6.3 `stdexec` Limitations and Workarounds

Sender Adaptor Pipeline Length

Sender adaptor pipelines of sufficient length or complexity have been observed to fail compilation, typically resulting in extremely verbose error messages. These errors often stem from internal assertions within the library or unresolved template instantiations. Notably, these errors do not originate from incorrect user code, but rather from limitations in the current implementation of `stdexec` when handling deeply nested or complex pipeline expressions.

```
#define ARR_SZ 10
static int count[ARR_SZ];
for(int ind = 0; ind < ARR_SZ; ind++)
  count[ind] = 0;

stdexec::sender auto mysender = stdexec::schedule(sched)
| stdexec::bulk(stdexec::par, ARR_SZ, [&](int ind){ count[ind]++; })
| stdexec::then([&](){
  for(int ind = 0; ind < ARR_SZ; ind++){
    std::cout << count[ind] << " ";
}})
| stdexec::then([](){ std::cout << "\n"; });

stdexec::sync_wait(mysender);
```

Listing 5.1: Sequence of `bulk-then-then` invocations. This code example was repeated multiple times (Listing 5.2) to illustrate a compilation issue in correspondence with NVIDIA.

Listing 5.1 contains a pipeline of three sender adaptors. The `bulk-then-then` sequence increments all elements of array `count` and prints them. However, when the sequence is repeated multiple times and pipelined together (Listing 5.2) `nvc++` gives an error when 11 `bulk-then-then`'s are reached. The code works with 10 repetitions of this sequence. Correspondence with NVIDIA revealed the error was caused by a hard-coded limit of the stack size for pending template instantiations of 256. This limit cannot be configured by the user. More recent versions of the `stdexec` library have reduced the depth of template instantiation required by the library's internals, making such failures less likely, though the limit can still be reached in sufficiently complex cases. When encountered in code development, the workaround has been to introduce additional synchronisation points (using `sync_wait`) to break-down a long sender pipeline into multiple shorter ones.

```
stdexec::sender auto mysender = stdexec::schedule(sched)
| stdexec::bulk(...)
| stdexec::then(...)
| stdexec::then(...)
| stdexec::bulk(...)
| stdexec::then(...)
| stdexec::then(...)
...
| stdexec::bulk(...)
| stdexec::then(...)
| stdexec::then(...)

stdexec::sync_wait(mysender);
```

Listing 5.2: Repeated `bulk-then-then` invocations which cause `nvc++` compilation failure. Each `bulk-then-then` sequence is of the form shown in Listing 5.1.

GPU-CPU-GPU Switching

Switching from GPU to CPU to GPU again (as shown in Listing 5.3) fails compilation giving a number of `nvlink` errors. However when the code is compiled without the `-DSWITCH_BACK` compiler flag, no errors are thrown and the code functions as expected.

```
//do some work on GPU:
stdexec::sender auto incr_vals = stdexec::schedule(scheduler_gpu)
| stdexec::then([=](){ (*color)++; })

//now switch to CPU and do some work:
| stdexec::continues_on(scheduler_cpu)
| stdexec::then([=](){ (*color)++; })

//now swtich back to GPU and do work:
#ifdef SWITCH_BACK
| stdexec::continues_on(scheduler_gpu)
| stdexec::then([=](){ (*color)++; })
#endif
;

stdexec::sync_wait(incr_vals);
```

Listing 5.3: Code example which fails compilation when switching from GPU to CPU to GPU.

Chapter 6

Conclusions and Further Work

This chapter discusses the main findings of comparing the performances of the `stdpar` and `stdexec` parallelism models, before touching upon further research directions.

6.1 Conclusions

The comparison between the `stdpar` and `stdexec` models was conducted along five dimensions: absolute runtime, scalability, heterogeneous programming support, portability, and the maturity of their programming environments. The key findings from this study are summarised below:

- **While the `stdpar` and `stdexec` parallelism models achieve comparable runtimes, this study suggests a slight performance benefit when using `stdpar`.** Both `stdpar` and `stdexec` produce correct solutions for the HPCG benchmark problem capable of CPU and GPU execution. Runtimes between them are similar. Runtimes are also similar to the baseline, which uses OpenMP. However, `stdpar` seems slightly faster than `stdexec`. While understanding underlying causes demands further investigation, possible reasons include runtime overheads with sender abstractions and a sub-optimal arrangement of senders having been employed.
- **For certain kernels, the performance gap between `stdpar` and `stdexec` is more pronounced, such as those involving reduction operations.** Kernel-based runtime breakdowns are similar between `stdexec` and `stdpar` HPCG CPU code, with SYMGS contributing the biggest difference. With `stdpar` and `stdexec` GPU code, `stdexec` SPMV is notably slower than its `stdpar` equivalent. The reasons are not understood; this warrants further investigation. `stdexec` GPU dot product (which is user implemented, owing to the absence of a reduction operation in the `stdexec` library) is also notably slower than its `stdpar` counterpart.
- **While `stdpar` and `stdexec` generally display comparable scaling behavior, for small problem sizes `stdpar` demonstrates superior CPU scaling relative to `stdexec`, with CPU performance exceeding GPU performance; this effect is not observed with `stdexec`.** Both `stdpar` and

`stdexec` observe similar scaling behaviour with one exception: for small problem sizes, `stdpar` CPU becomes much faster than `stdexec` CPU and even outperforms the GPU implementations. While `stdpar` CPU code outperforms `stdpar` GPU code for small problem sizes (making the switch from GPU to CPU execution viable), a performance crossover point between `stdexec` CPU and GPU code fails to occur. Understanding this crucial difference demands further work.

- **While `stdpar` code seems to consistently outperform `stdexec` code across different platforms, the performance difference is less on GPU platforms compared to CPU platforms.** While the `stdpar` CPU code has notably higher performance portability than `stdexec` CPU code, the respective GPU implementations have similar performance portability. This suggests that for GPU execution, the differences between the two parallelism models across different platforms is less noticeable.
- The `stdexec` library provides a broader set of features and functionality than `stdpar`, reflecting its more ambitious design goals. However, as it is still in development, its performance is likely not yet as highly optimised, and occasional bugs are present. Continued development is likely to address these limitations over time.

6.2 Further Work

Directions for further work include:

- Understanding why `stdexec` code is sometimes slower than `stdpar` code.
- Investigating strategies where the CPU and GPU are utilised in parallel.
- Implementing user-defined senders and receivers for leveraging specific hardware capabilities.
- Comparing `stdexec` against other parallelism models for heterogeneous execution such as SYCL, Kokkos and RAJA.
- Studying scaling on multi-node and distributed systems.

Bibliography

[1] Yuuichi Asahi, Thomas Padioleau, Guillaume Latu, Julien Bigot, Virginie Grandgirard, and Kevin Obrejan. Performance portable Vlasov code with C++ parallel algorithm. In *2022 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 68–80, 2022.

[2] David Bailey, E. Barszcz, Barton J.T, Browning D.S, Carter R.L, Dagum D, Fatoohi R.A, Paul Frederickson, Lasinski T.A, Robert Schreiber, Horst Simon, Venkat Venkatakrishnan, and Weeratunga K. The Nas Parallel Benchmarks. *International Journal of High Performance Computing Applications*, 5:63–73, 09 1991.

[3] Marcel Breyer, Alexander Van Craen, and Dirk Pflüger. Short Paper: Evaluation of stdpar Compilers on a Kernel Matrix Assembly and a BLAS Level 3 SYMM Kernel. In *2024 23rd International Symposium on Parallel and Distributed Computing (ISPDC)*, pages 1–5, 2024.

[4] William L. Briggs, Van Emden Henson, and Steve F. McCormick. *A Multigrid Tutorial, Second Edition*. Society for Industrial and Applied Mathematics, second edition, 2000.

[5] Gonzalo Brito Gadeschi and Jonas Latt. Portable GPU Acceleration of HPC Applications with ISO C++. Presentation, NVIDIA, 2023. PDF slides, available at https://www.nvidia.com/en-us/on-demand/session/gtcspring23-dlit51169/ [Accessed 31-Aug-2025].

[6] Michal Dominiak, Kirk Shoop, Eric Niebler, Bryce Adelstein Lelbach, Lewis Baker, and Michael Garland. P2300R10: std::execution. Technical report, ISO/IEC JTC1/SC22/WG21, October 2024. [Accessed 01-Sep-2025].

[7] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. An iterative solver benchmark. *Sci. Program.*, 9(4):223–231, December 2001.

[8] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK Benchmark: past, present and future. *Concurrency and Computation: Practice and Experience*, 15, 2003.

[9] Meta (Facebook Experimental). libunifex. GitHub repository, 2024. Available at https://github.com/facebookexperimental/libunifex [Accessed 25-Aug-2025].

[10] Toshihiro Hanawa, Kengo Nakajima, Yohei Miki, Takashi Shimokawabe, Kazuya Yamazaki, Shinji Sumimoto, Osamu Tatebe, Taisuke Boku, Daisuke Takahashi, Akira Nukada, Norihisa Fujita, Ryohei Kobayashi, Hiroto Tadano, and Akira Naruse. Preliminary Performance Evaluation of Grace-Hopper

GH200. In *2024 IEEE International Conference on Cluster Computing Workshops (CLUSTER Workshops)*, pages 184–185, 2024.

[11] Muhammad Haseeb, Weile Wei, Jack Deslippe, and Brandon Cook. That's right, the same C++ STL asynchronous parallel code runs on CPUs & GPUs. 2023.

[12] Intel. cpp-baremetal-senders-and-receivers. GitHub repository, 2024. Available at https://github.com/intel/cpp-baremetal-senders-and-receivers [Accessed 25-Aug-2025].

[13] ISO/IEC JTC1/SC22/WG21. Draft—Clause 33: Execution Control Library. https://eel.is/c++draft/exec, 2025. [Online; accessed 25-Aug-2025].

[14] Kiyoshi Kumahata, Kazuo Minami, and Naoya Maruyama. High-performance conjugate gradient performance improvement on the K computer. *Int. J. High Perform. Comput. Appl.*, 30(1):55–70, February 2016.

[15] Júnior Löff, Renato B. Hoffmann, Arthur S. Bianchessi, Leonardo Mallmann, Dalvan Griebler, and Walter Binder. NPB-PSTL: C++ STL Algorithms with Parallel Execution Policies in NAS Parallel Benchmarks. In *2025 33rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 162–169, 2025.

[16] Giulio Malenza, Valentina Cesare, Marco Aldinucci, Ugo Becciani, and Alberto Vecchiato. Toward HPC application portability via C++ PSTL: the Gaia AVU-GSR code assessment. *J. Supercomput.*, 80(10):14369–14390, March 2024.

[17] Nicco Mietzsch and Karl Fuerlinger. Investigating Performance and Potential of the Parallel STL Using NAS Parallel Benchmark Kernels. *2019 International Conference on High Performance Computing & Simulation (HPCS)*, pages 136–144, 2019.

[18] Eric Niebler et al. GitHub - NVIDIA/stdexec: std::execution, the proposed C++ framework for asynchronous and parallel programming. https://github.com/NVIDIA/stdexec, 2025. [Accessed 01-Sep-2025].

[19] NVIDIA. NVIDIA HPC-Benchmarks Container (including HPCG). https://catalog.ngc.nvidia.com/orgs/nvidia/containers/hpc-benchmarks, Apr 2025. [Online; accessed 01-Sep-2025].

[20] NVIDIA Corporation. C++ Parallel Algorithms. Technical Report PR-10011-001-V20.7, NVIDIA Corporation, August 2020. Available at https://docs.nvidia.com/hpc-sdk/archive/20.7/pdf/hpc207c++_par_alg.pdf.

[21] S. John Pennycook, Jason D. Sewall, Douglas W. Jacobsen, Tom Deakin, and Simon McIntosh-Smith. Navigating Performance, Portability, and Productivity. *Computing in Science & Engineering*, 23(5):28–38, 2021.

[22] Simon John Pennycook, Jason D. Sewall, and Victor W. Lee. A Metric for Performance Portability. *ArXiv*, abs/1611.07409, 2016.

[23] pika-org (GitHub). pika: A C++ tasking library built on std::execution (P2300). https://github.com/pika-org/pika, 2025. Latest release: v0.34.0 (11 June 2025).

[24] Long Qu, Hatem Ltaief, and David Keyes. Leveraging Stencil Computation Performance with Temporal Blocking using Large Cache Capacity on AMD EPYC 7003 Processors with AMD 3D V-Cache Technology. In *Research Posters, SC '22 Conference*, 2022. Poster; Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '22).

[25] Sandia Report, M. Heroux, Jack J. Dongarra, and Piotr Luszczek. HPCG Technical Specification. 2013.

[26] Yousef Saad. *Iterative Methods for Sparse Linear Systems.* Society for Industrial and Applied Mathematics, second edition, 2003.

[27] Nikolay Sakharnykh. High-Performance Geometric Multi-Grid with GPU Acceleration. NVIDIA Developer Blog, 2016. Available at https://developer.nvidia.com/blog/high-performance-geometric-multi-grid-gpu-acceleration/ [Accessed 31-Aug-2025].

[28] 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, 2023.

[29] Jonathan R Shewchuk. An Introduction to the Conjugate Gradient Method Without the Agonizing Pain. Technical report, USA, 1994.

[30] Raffaele Solcà, Mikael Simberg, Rocco Meli, Alberto Invernizzi, Auriane Reverdell, and John Biddiscombe. DLA-Future: A Task-Based Linear Algebra Library Which Provides a GPU-Enabled Distributed Eigensolver. In *Workshop on Asynchronous Many-Task Systems and Applications*, pages 135–141. Springer, 2024.

[31] Anthony Williams. *C++ Concurrency in Action.* Simon and Schuster, 2019.

[32] Ziheng Yuan and Takashi Shimokawabe. Accelerating lattice Boltzmann method with asynchronous stdexec sender/receiver library. In *Proceedings of the Conference on Computational Engineering and Science*, volume 29, pages 962–965, 2024.

[33] Yuliana Zamora and Robert Robey. *Parallel and High Performance Computing.* 2021.

Appendices

Appendix A

Source Code and Thesis Availability

The code is a fork of the Official HPCG Benchmark and is available at https://github.com/ab2163/hpcg. Unless otherwise stated, results in figures and tables (including numbers provided in the discussion) have been generated using commit 63a0452. The LaTeX source code of this thesis, together with figures and datasets, are available upon request.

Appendix B

Program Structure

Figure B.1 shows a call graph linking the functions executing individual kernels to `main`. Functions for the baseline, `stdpar` and `stdexec` implementations are given in red, blue and green respectively. When compiling `CG`, the compiler calls selector functions which choose the relevant implementation depending upon the provided compiler flags. For example, `ComputeDotProduct` selects either `ComputeDotProduct_ref` or `ComputeDotProduct_stdpar` during compilation. Because of the asynchronous nature of the `stdexec` implementation, `CG_stdexec` is implemented as a separate function (called directly by `CG`) and invokes no further functions since it contains all relevant kernels as internal lambda functions. `ComputeMG_ref` may also call the `ComputeSYMGS_par` function (shown in yellow) which provides a parallel SYMGS implementation, instead of `ComputeSYMGS_ref`.

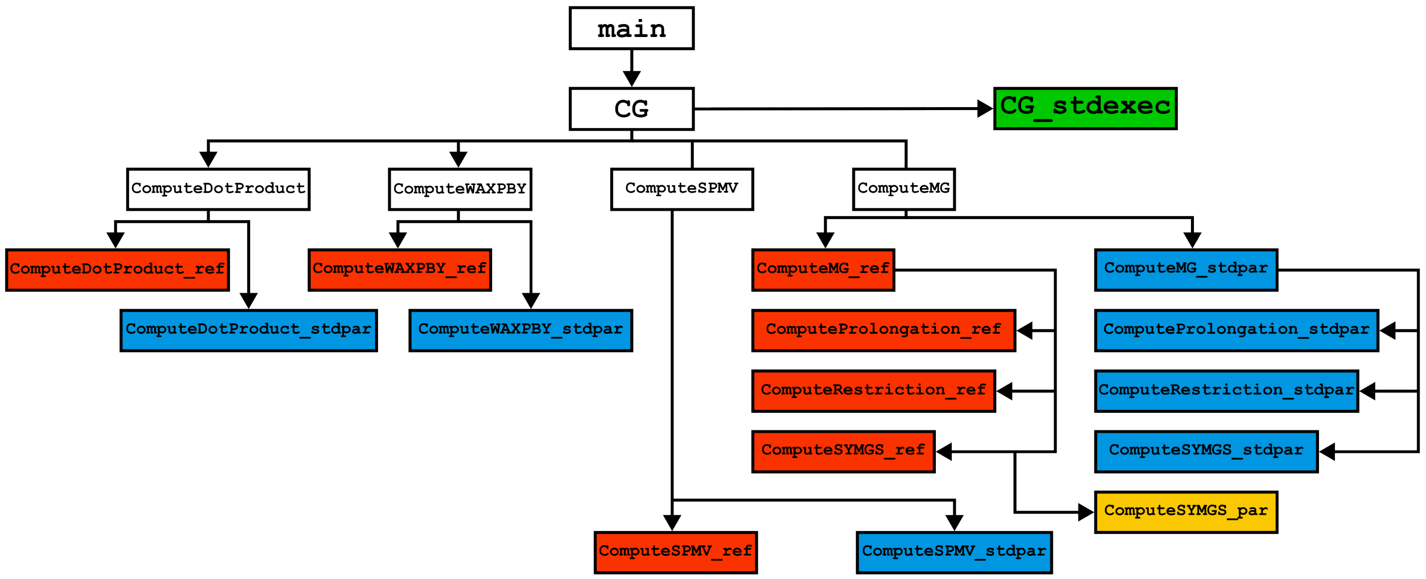


Figure B.1: Call graph for execution of Conjugate Gradient (`CG`) algorithm. The user can specify which implementation of the program to build. Based on this choice, only the functions required for the selected implementation are compiled into the final executable.

Appendix C

Build and Execution Instructions

Since `nvc++` compiler is only available for Linux, these instructions only apply to Linux-based systems. Compilation steps to produce and run the `xhpcg` executable are:

1. Clone the GitHub repo (using `--recursive` to also clone the `stdexec` submodule):
   `git clone --recursive https://github.com/ab2163/hpcg`
2. Install NVIDIA `nvc++` compiler and an MPI implementation.
3. Modify the `Make.bris` file within the `setup` subdirectory of the repo such that:
   (a) The `CXX` variable points to the correct location of the `nvc++` compiler:
   e.g. `CXX = /opt/nvidia/hpc_sdk/Linux_x86_64/2025/compilers/bin/nvc++`
   (b) The `MPinc` and `MPlib` variables point to the `include` and `lib` directories of the MPI installation:
   e.g. `MPinc = -I/usr/lib/x86_64-linux-gnu/openmpi/include`
   e.g. `MPlib = /usr/lib/x86_64-linux-gnu/openmpi/lib`
   (c) In `CXXFLAGS`, the include path for the `stdexec/include` folder within HPCG repo is correctly set:
   e.g. `CXXFLAGS = ... -I/root/hpcg/stdexec/include`
4. Change the relevant line in `bris_build.sh` (which calls the `configure` file within the HPCG repo) to match the location of the `configure` file on the system:
   e.g. `/root/hpcg/configure bris`
5. Run one of the commands given in Table C.1 to build the code.
6. Adjust the environment variables in `bris_run.sh` (refer to Appendix F). Also adjust the HPCG problem size in the `hpcg.dat` file within the `build/bin` folder.
7. Run the `bris_run.sh` script, after which an output file will be produced in the `build/bin` folder containing the benchmark results.
8. To remove all files generated during building and execution of the program run `bris_clean.sh`.

Code Implementation	Build Command
Baseline	`bris_build.sh baseline`
Baseline with Parallel SYMGS	`bris_build.sh baseline_par`
`stdpar` (CPU execution)	`bris_build.sh stdpar_cpu`
`stdpar` (GPU execution)	`bris_build.sh stdpar_gpu`
`stdexec` (CPU execution)	`bris_build.sh stdexec_cpu`
`stdexec` (GPU execution)	`bris_build.sh stdexec_gpu`

Table C.1: Commands to build the `xhpcg` executable file using the `bris_build.sh` script.

Appendix D

stdpar Kernel Implementation Code

```
int ComputeDotProduct_stdpar(const local_int_t n, const Vector &x, const Vector &y,
  double &result, double &time_allreduce){

  double local_result = 0.0;
  const double * const xv = x.values;
  const double * const yv = y.values;
  if(yv == xv){
    local_result = std::transform_reduce(std::execution::par_unseq,
      xv, xv + n, xv, 0.0);
  }
  else{
    local_result = std::transform_reduce(std::execution::par_unseq,
      xv, xv + n, yv, 0.0);
  }

  result = local_result;
  return 0;
}
```

Listing D.1: `stdpar` dot product kernel.

```
int ComputeProlongation_stdpar(const SparseMatrix &Af, Vector &xf){

  double * const xfv = xf.values;
  const double * const xcv = Af.mgData->xc->values;
  const local_int_t * const f2c = Af.mgData->f2cOperator;
  const local_int_t nc = Af.mgData->rc->localLength;
  auto range = std::views::iota(0, nc);

  std::for_each(std::execution::par_unseq, range.begin(), range.end(),
    [=](int i){ xfv[f2c[i]] += xcv[i]; });

  return 0;
}
```

Listing D.2: `stdpar` prolongation kernel.

```
int ComputeRestriction_stdpar(const SparseMatrix &A, const Vector &rf){

  const double * const Axfv = A.mgData->Axf->values;
  const double * const rfv = rf.values;
  double * const rcv = A.mgData->rc->values;
  const local_int_t * const f2c = A.mgData->f2cOperator;
  const local_int_t nc = A.mgData->rc->localLength;
  auto range = std::views::iota(0, nc);

  std::for_each(std::execution::par_unseq, range.begin(), range.end(),
    [=](int i){ rcv[i] = rfv[f2c[i]] - Axfv[f2c[i]]; });

  return 0;
}
```

Listing D.3: `stdpar` restriction kernel.

```
int ComputeSPMV_stdpar(const SparseMatrix &A, Vector &x, Vector &y){

  const double * const xv = x.values;
  double * const yv = y.values;
  const double * const * const amv = A.matrixValues;
  const local_int_t * const * const indv = A.mtxIndL;
  const char * const nnz = A.nonzerosInRow;
  const local_int_t nrow = A.localNumberOfRows;
  auto rows = std::views::iota(local_int_t{0}, nrow);

  std::for_each(std::execution::par_unseq, rows.begin(), rows.end(),
    [=](local_int_t i){
    double sum = 0.0;
    for(int j = 0; j < nnz[i]; j++){
      sum += amv[i][j] * xv[indv[i][j]];
    }
    yv[i] = sum;
  });

  return 0;
}
```

Listing D.4: `stdpar` SPMV kernel.

```
int ComputeWAXPBY_stdpar(const local_int_t n, const double alpha, const Vector &x,
  const double beta, const Vector &y, Vector &w){

  const double * const xv = x.values;
  const double * const yv = y.values;
  double * const wv = w.values;

  if(alpha == 1.0){
    std::transform(std::execution::par_unseq, xv, xv + n, yv, wv,
      [=](double xi, double yi){ return xi + beta*yi; });
  }else if(beta == 1.0){
    std::transform(std::execution::par_unseq, xv, xv + n, yv, wv,
      [=](double xi, double yi){ return alpha*xi + yi; });
  }else{
    std::transform(std::execution::par_unseq, xv, xv + n, yv, wv,
      [=](double xi, double yi){ return alpha*xi + beta*yi; });
  }

  return 0;
}
```

Listing D.5: `stdpar` WAXPBY kernel.

```cpp
#define NUM_COLORS 8
#define FWD_AND_BACK_SWEEPS 2

int ComputeSYMGS_stdpar(const SparseMatrix &A, const Vector &r, Vector &x){

  const local_int_t nrow = A.localNumberOfRows;
  const double * const * const matrixDiagonal = A.matrixDiagonal;
  const double * const rv = r.values;
  double * const xv = x.values;
  const double * const * const amv = A.matrixValues;
  const local_int_t * const * const indv = A.mtxIndL;
  const char * const nnz = A.nonzerosInRow;
  const unsigned char * const colors = A.colors;
  auto rows = std::views::iota(0, nrow);

  for(int sweeps = 0; sweeps < FWD_AND_BACK_SWEEPS; sweeps++){
    for(int color = 0; color < NUM_COLORS; color++){
      std::for_each(std::execution::par_unseq, rows.begin(), rows.end(),
        [=](local_int_t i){
        if(colors[i] == color){
          const double currentDiagonal = matrixDiagonal[i][0];
          double sum = rv[i]; //RHS value

          for(int j = 0; j < nnz[i]; j++){
            local_int_t curCol = indv[i][j];
            sum -= amv[i][j] * xv[curCol];
          }
          sum += xv[i]*currentDiagonal;
          xv[i] = sum/currentDiagonal;
        }
      });
    }
  }

  return 0;
}
```

Listing D.6: stdpar SYMGS kernel.

```
int ComputeMG_stdpar(const SparseMatrix &A, const Vector &r, Vector &x){

  ComputeWAXPBY_stdpar(x.localLength, 0, x, 0, x, x);

  int ierr = 0;
  if (A.mgData != 0){ //go to next coarse level if defined
    int numberOfPresmootherSteps = A.mgData->numberOfPresmootherSteps;
    for(int i = 0; i < numberOfPresmootherSteps; i++)
      ierr += ComputeSYMGS_stdpar(A, r, x);
    if (ierr != 0) return ierr;
    ierr = ComputeSPMV_stdpar(A, x, *A.mgData->Axf);
    if (ierr != 0) return ierr;
    //perform restriction operation using simple injection
    ierr = ComputeRestriction_stdpar(A, r);
    if (ierr != 0) return ierr;
    ierr = ComputeMG_stdpar(*A.Ac, *A.mgData->rc, *A.mgData->xc);
    if (ierr != 0) return ierr;
    ierr = ComputeProlongation_stdpar(A, x);
    if (ierr != 0) return ierr;
    int numberOfPostsmootherSteps = A.mgData->numberOfPostsmootherSteps;
    for (int i = 0; i < numberOfPostsmootherSteps; i++)
      ierr += ComputeSYMGS_stdpar(A, r, x);
    if (ierr != 0) return ierr;
  }
  else{
    ierr = ComputeSYMGS_stdpar(A, r, x);
    if(ierr != 0) return ierr;
  }
  return 0;
}
```

Listing D.7: stdpar MG kernel.

Appendix E

stdexec Kernel Implementation Code

```
#define NUM_BINS 1000

auto dot_prod_stg1 = [=](local_int_t i, const double * const vec1_vals,
  const double * const vec2_vals){

  local_int_t minInd = i*(nrow/NUM_BINS);
  local_int_t maxInd = ((i + 1) == NUM_BINS) ? nrow : (i + 1)*(nrow/NUM_BINS);
  double bin_sum = 0.0;
  for(local_int_t j = minInd; j < maxInd; ++j){
    bin_sum = std::fma(vec1_vals[j], vec2_vals[j], bin_sum);
  }
  bin_vals[i] = bin_sum;
};

auto dot_prod_stg2 = [=](double *result){

  double result_cpy = 0.0;
  for(local_int_t i = 0; i < NUM_BINS; ++i) result_cpy += bin_vals[i];
  *result = result_cpy;
};
```

Listing E.1: `stdexec` dot product kernels.

```
auto waxpby = [=](local_int_t i, double alpha, const double * const xvals,
  double beta, const double * const yvals, double *wvals){

  wvals[i] = alpha*xvals[i] + beta*yvals[i];
};
```

Listing E.2: `stdexec` WAXPBY kernel.

```cpp
auto spmv = [=](local_int_t i, const double * const * const avals,
  const double * const xvals, double *yvals,
  const local_int_t * const * const indvals, const char * const nnz){

    double sum = 0.0;
    for(int j = 0; j < nnz[i]; ++j){
      sum += avals[i][j] * xvals[indvals[i][j]];
    }
  yvals[i] = sum;
};
```

Listing E.3: `stdexec` SPMV kernel.

```cpp
auto restriction = [=](local_int_t i, double *rcv_vals, const double * const r_vals,
  const local_int_t * const f2c_vals, const double * const Axfv_vals){

  rcv_vals[i] = r_vals[f2c_vals[i]] - Axfv_vals[f2c_vals[i]];
};
```

Listing E.4: `stdexec` restriction kernel.

```cpp
auto prolongation = [=](local_int_t i, double *z_vals,
  const local_int_t * const f2c_vals, const double * const xcv_vals){

  z_vals[f2c_vals[i]] += xcv_vals[i];
};
```

Listing E.5: `stdexec` prolongation kernel.

```cpp
auto symgs = [=](local_int_t i, const double * const * const avals, double *xvals,
  const double * const rvals, const char * const nnz,
  const local_int_t * const * const indvals, const double * const * const diagvals,
  const unsigned char * const colors){

  if(colors[i] == *color){
    const double currentDiagonal = diagvals[i][0];
    double sum = rvals[i];
    for(int j = 0; j < nnz[i]; ++j){
      local_int_t curCol = indvals[i][j];
      sum -= avals[i][j] * xvals[curCol];
    }
    sum += xvals[i]*currentDiagonal;
    xvals[i] = sum/currentDiagonal;
  }
};
```

Listing E.6: `stdexec` SYMGS kernel.

```
auto zerovector_0 = [=](local_int_t i){
  waxpby(i, 0, z_vals0, 0, z_vals0, z_vals0); };
auto symgs_0 = [=](local_int_t i){
  symgs(i, A_vals0, z_vals0, r_vals0, A_nnzs0, A_inds0, A_diags0, A_colors0); };
auto spmv_mg0 = [=](local_int_t i){
  spmv(i, A_vals0, z_vals0, Axfv_vals0, A_inds0, A_nnzs0); };
auto restriction_0 = [=](local_int_t i){
  restriction(i, rcv_vals0, r_vals0, f2c_vals0, Axfv_vals0); };

sender auto mg_stg0 = schedule(scheduler)
| bulk(par_unseq, A_nrows0, zerovector_0)
| then([=](){ *color = 0; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ *color = 0; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| then([=](){ (*color)++; })
| bulk(par_unseq, A_nrows0, symgs_0)
| bulk(par_unseq, A_nrows0, spmv_mg0)
| bulk(par_unseq, A_nrows1, restriction_0);
```

Listing E.7: First stage of the `stdexec` MG V-Cycle. The relevant kernel wrapper functions used by `mg_stg0` are also shown.

```
sender auto rest_of_loop = schedule(scheduler)
| then([=](){ *oldrtz = *rtz; })
| bulk(par_unseq, NUM_BINS, dot_prod_rz_stg1)
| then(dot_prod_rz_stg2)
| then([=](){ *beta = *rtz/(*oldrtz); })
| bulk(par_unseq, nrow, waxpby_peqbppz)
| bulk(par_unseq, nrow, spmv_Ap)
| bulk(par_unseq, NUM_BINS, dot_prod_pAp_stg1)
| then(dot_prod_pAp_stg2)
| then([=](){ *alpha = *rtz/(*pAp); })
| bulk(par_unseq, nrow, waxpby_xeqxpap)
| bulk(par_unseq, nrow, waxpby_reqrmaAp)
| bulk(par_unseq, NUM_BINS, dot_prod_rr_stg1)
| then(dot_prod_rr_stg2)
| then([=](){ *normr_cpy = sqrt(*normr_cpy); });

for(k = 2; k <= max_iter && *normr_cpy/(*normr0_cpy) > tolerance; k++){
  sync_wait(mg_stg0);
  sync_wait(mg_stg1);
  sync_wait(mg_stg2);
  sync_wait(mg_stg3);
  sync_wait(mg_stg4);
  sync_wait(mg_stg5);
  sync_wait(mg_stg6);
  sync_wait(rest_of_loop);
}
```

Listing E.8: Loop of stdexec CG algorithm. The rest_of_loop sender is pre-defined to avoid construction within the loop each time.

Appendix F

Compiler Flags and Environment Variables

Compiler Flags for `nvc++` and `stdpar`

`-stdpar=multicore` was specified for CPU and `-stdpar=gpu` was specified for GPU execution. `-O3` optimisation was used. User-defined flags `-DSELECT_STDPAR` and `-DPARALLEL_SYMGS` were also specified.

Compiler Flags for `nvc++` and `stdexec`

`-stdpar=multicore` was specified for CPU and `-stdpar=gpu` was specified for GPU execution. `-std=c++20` and the path for `stdexec/include` were specified. User-defined flags `-DSELECT_STDEXEC` and `-DPARALLEL_SYMGS` were also specified. `-O3` optimisation was used.

Environment Variables for `stdpar` and Baseline

`OMP_NUM_THREADS` was set to the number of hardware cores of the machine. `nproc` was not used to provide this value because `nproc` gives the number of logical cores available rather than hardware cores. Multiple logical cores using the same hardware core on a memory-bound application like HPCG is undesirable since threads on the same physical core will contend for memory bandwidth.

`OMP_PROC_BIND=spread` was used to allocate one thread per hardware core. `OMP_PLACES=threads` was specified to prevent threads from moving between different logical cores within the same hardware core.

On machines where NUMA regions introduced slowdowns due to undesirable first-touch effects (e.g. the Isambard MACS3 cluster), the `bris_run.sh` script was executed with `numactl --interleave=all ./bris_run.sh` to evenly allocate data across all NUMA regions.

Environment Variables for `stdexec`

The command `ulimit -s 16384` (which increases the maximum stack size `nvc++` can use from default 8192 to 16384 kilobytes) was performed before performing compilation since long sender pipelines could exhaust the compiler stack.

`OMP_NUM_THREADS` was set to the number of hardware cores of the machine. When instantiating a thread pool, `omp_get_max_threads()` was used to specify the number of threads used. `OMP_PROC_BIND=spread` was used with `OMP_PLACES` set to `cores` because `threads` seemed to have no effect. Fortunately, using `cores` also seemed to bind the threads to specific logical cores and avoid the issue of threads switching between different logical cores within the same physical core (as observed with `stdpar` when `cores` was used).

On machines where NUMA regions introduced slowdowns due to undesirable first-touch effects (e.g. the Isambard MACS3 cluster), the `bris_run.sh` script was executed with `numactl --interleave=all ./bris_run.sh` to evenly allocate data across all NUMA regions. `OMP_PROC_BIND=false` was also specified since other settings were detrimental to performance.

Appendix G

Optimisations to `stdexec` Code

Several design choices were made with the primary goal of improving performance:

- **Kernels as named lambda functions in `CG_stdexec` function.** Each kernel needs pointers to the data it operates upon. Named lambdas allowed a common set of data pointers to be defined in `CG_stdexec` which the lambdas captured as needed.
- **Kernel wrapper functions.** The callable passed to `stdexec::bulk` can only have a single argument (the element index, typically denoted `i`). However, the kernel lambdas generally take multiple arguments corresponding to all the parameters and data pointers they require (for example, the `waxpby` kernel needs index `i`, the `alpha` and `beta` parameters, plus the `xvals`, `yvals` and `wvals` pointers). A simple approach was taken involving kernel wrapper functions which specified all kernel arguments except `i`. The wrapper functions were then passed to sender adaptors like `stdexec::bulk`. Listing G.1 shows the `waxpby` kernel and two such wrapper functions.
- **Named pointers.** To minimise any potential indexing of pointers, named pointers were declared directly pointing to relevant data. For example, `A_vals1` points directly to values of `SparseMatrix A` at the second-finest MG level and avoids equivalent expressions such as `A->Ac->values` from being passed to or captured by kernel lambda functions.
- **Pre-constructed senders.** Named senders were constructed in advance and reused within the CG loop. Listing G.2 illustrates this procedure.
- **Sender pipelines as long as possible.** Within `nvc++` compiler limits (see Section 5.6.3), sender pipelines were constructed to be as long as possible to minimise explicit synchronisation points.
- **`const` lambda function parameters.** Parameters of kernel lambda functions were declared `const` whenever possible to allow the compiler to make aggressive optimisations.

```
//waxpby kernel definition
auto waxpby = [=](local_int_t i, double alpha, const double * const xvals,
double beta, const double * const yvals, double *wvals){

       wvals[i] = alpha*xvals[i] + beta*yvals[i];
};

//kernel wrapper function: p = x
auto waxpby_peqx = [=](local_int_t i){
       waxpby(i, 1, x_vals, 0, x_vals, p_vals);
};

//kernel wrapper function: r = b - Ap
auto waxpby_reqbmAp = [=](local_int_t i){
       waxpby(i, 1, b_vals, -1, Ap_vals, r_vals0);
};
```

Listing G.1: WAXPBY kernel lambda function and two wrapper functions. The wrapper functions specify all arguments for the kernel except the element index `i`.

```
//constructs loop_work every time
while(residual > tolerance){
  stdexec::sender auto loop_work = ...;
  stdexec::sync_wait(loop_work);
}

//pre-construct loop_work and reuse within loop
stdexec::sender auto loop_work = ...;
while(residual > tolerance){
  stdexec::sync_wait(loop_work);
}
```

Listing G.2: Simplified code illustrating the usage of pre-constructed senders. The second `while` loop is equivalent to the first but executes faster because `loop_work` does not require construction every loop iteration.

Appendix H

Designing GPU-Compatible Kernels

CUDA Unified Memory

The `nvc++` compiler uses CUDA Unified Memory to automatically manage data shared between CPU and GPU. NVIDIA guidelines [20] specify that only heap memory dynamically allocated in CPU code is supported by this feature. CPU stack memory and memory used by global objects cannot be managed by this feature. This means when a lambda function captures a variable by reference, the variable must have been dynamically allocated in heap memory. Otherwise, the variable should be captured by value.

The approach taken was to only capture variables by value in lambda functions passed to kernels. This was straightforward for constants and individual parameters used by the kernels. Whenever the kernel needed to modify data to be used after the kernel finished, the data was placed in heap memory[1] and a pointer passed to the lambda function by value.

The baseline HPCG kernels generally take `SparseMatrix` and `Vector` objects by reference. These objects do not hold their underlying data (which is held in heap memory) but do contain various metadata related to the underlying data. Rather than capturing these objects by value (and copying them to GPU memory), the relevant pointers of the underlying data used by the kernel were calculated in advance and directly passed to the `stdpar` lambda function. This reduces data transfer to GPU and removes the overhead of repeated array indexing within the kernel.

Trivially Copyable Data Structures

Variables which are not trivially copyable cannot be captured by value in lambda functions passed to GPU kernels. A `std::vector` added to the `SparseMatrix` object for storing colours of elements (used within parallel SYMGS) failed this assertion (Listing H.1) and was therefore changed to a regular array (allocated using `new`).

[1]The baseline implementation of the HPCG algorithm generally allocates data in heap memory using `new` which lends to this approach.

```
"/home/ajinkya/Documents/hpcg/stdexec/include/nvexec/stream/bulk.cuh", line 40: error: static
    assertion failed
static_assert(trivially_copyable<Shape, Fun, As...>);
```

Listing H.1: Failure of `trivally_copyable` assertion

Random Access Iterators

While the C++ standard requires that iterators passed to `stdpar` algorithms are forward iterators, GPU offloading requires random access iterators. Raw pointers and iterators generated from `std::views::iota` were used for this purpose.

Reduced C++ Standard Library Support

While certain standard library features (notably output using `std::cout`) are unsupported within GPU kernels, `printf` is supported and proved useful for debugging.

CPU-Only Sender Adaptors

Some sender adaptors fail GPU execution (e.g. `exec::repeat_n`). A GPU pipeline must switch to a CPU-derived scheduler to execute them.

NVTX Markers Incompatibility

NVTX markers, used by the NVIDIA `nsys` profiler, were unsupported within GPU-executed sender adaptors.

Appendix I

Kernel for Testing stdpar and stdexec GPU Execution

```
#define ARR_SZ 100000000

int main(void){
  double *long_arr = new double[ARR_SZ];
  double *transformed_arr = new double[ARR_SZ];

  //initialise the input array
  for(int ind = 0; ind < ARR_SZ; ++ind)
    long_arr[ind] = rand() % 100;

  //kernel for gpu testing
  auto transform_func = [=](int ind){
    transformed_arr[ind] = long_arr[ind]*long_arr[ind]
      + 0.5*long_arr[ARR_SZ-(ind+1)];
  };

  nvexec::stream_context ctx;
  auto sched_gpu = ctx.get_scheduler();
  auto indices = std::views::iota(0, ARR_SZ);

#ifdef STDPAR
  std::for_each(std::execution::par_unseq, indices.begin(), indices.end(), transform_func);
#endif

#ifdef STDEXEC
  stdexec::sync_wait(stdexec::schedule(sched_gpu)
  | stdexec::bulk(stdexec::par_unseq, ARR_SZ, transform_func));
#endif
}
```

Listing I.1: Kernel for testing of `stdpar` and `stdexec` GPU offloading performance.