ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

A Matrix Free Preconditioner for Exascale Computing

Bachelor Thesis

CSE, Department of Mathematics

Bolliger Matteo

December 16, 2024

Supervisor: Dr. Andreas Adelmann
Scientific Advisor: Sonali Mayani

Abstract

This thesis focuses on improving the scalability and performance of the Preconditioned Conjugate Gradient (PCG) solver in the Independent Parallel Particle Layer (IPPL) framework, making it better suited for exascale computing. Apart from improving the performance of the existing Richardson and Tow-step Gauss-Seidel preconditioners, a new Symmetric Successive Over-Relaxation (SSOR) method is introduced, following the same Matrix-free approach. The correctness of the methods is validated by comparing numerical solutions to analytical results. On the other hand, scaling studies reveal significant improvement on performance especially using a GPU system. SSOR reduces the iteration count of the CG by a large amount showcasing the capabilities of preconditioning while still being more scalable than the base CG. However, limitations in load balancing highlight points for further optimization. These findings contribute to advancing the IPPL framework toward the usage on even bigger and faster systems and lay the groundwork to extend the PCG solver to solve different relevant PDEs other then Poisson-type problems.

Contents

Chapter 1

Introduction

This thesis aims to improve the performance and efficiency of the Preconditioned Conjugate Gradient (PCG) solver, which was implemented in the Independent Parallel Particle Layer framework (IPPL) [4] by Alessandro Vinciguera [6] and Bob Schreiner [2], with the ultimate goal of scaling to exascale architectures. IPPL is a C++ library built around Particle-In-Cell solvers for solving plasma physics problems, specifically designed for large-scale distributed systems. Building on this, the focus is now on two main objectives: refining the existing preconditioners to improve their efficiency and speed, and introducing a new method to this framework called Symmetric Successive Over-Relaxation (SSOR) which is based on [3] and combined with ideas introduced by [1].

A description of the PCG solver as well as an in-depth derivation of the existing Richardson and Two-step Gauss-Seidel preconditioners and the SSOR method will be provided in Chapter 2. Chapter 3 will provide the details about the implementation of the SSOR preconditioner and the matrix-free approach used by all methods, which has proven to be efficient on large scale systems, particularly on GPUs, as it avoids building the entire matrix and reduces memory-bound limitations.. Following the theory the base CG and the preconditioners are tested on multiple plasma physics problems [4] for correctness and scalability, which will be given in Chapter 4 with a discussion of the results. The final notes and conclusion of the thesis can be found in Chapter 5 with some suggestions to further improve and extend this part of IPPL.

Chapter 2

Preconditioned Conjugate Gradient

In this chapter a brief explanation of the Preconditioned Conjugate Gradient solver is given to understand the problem we are trying to solve. An extensive description of the CG solver can be found in [5]. In addition a precise explanation and definition based on [1, 3] of the different preconditioners used are given. This includes a pseudo-code for each method corresponding to its implementation. Throughout this chapter the use of the function $F_A(\vec{x})$ represents the matrix-free implementation of the multiplication of matrix A with vector $\vec{x}$.

2.1 Conjugate Gradient

The (Preconditioned) Conjugate Gradient solver is used for solving linear systems of equations of the form

$$A\vec{x} = \vec{b}, \tag{2.1}$$

where $A \in \mathbb{R}^{n\times n}$ is symmetric positive definite, $\vec{b} \in \mathbb{R}^n$ is the right-hand side vector and we solve for $\vec{x}$. An exact explanation of the solver can be found in [5]. The pseudo-code in Algorithm 1 shows the iterative steps of generating conjugate directions, which are search directions orthogonal with respect to A, and updating the solution along these directions until convergence.

One can precondition the matrix by applying a matrix M^{-1} to the system:

$$M^{-1}A\vec{x} = M^{-1}\vec{b}.$$

This matrix M is known as the preconditioner matrix and should closely approximate A and its inverse needs to be efficient to compute, because the PCG algorithm is an iterative method for which the time to convergence and

to solution is mainly dependent on the condition number of A:

$$\kappa(A) = \frac{\lambda_{\max}}{\lambda_{\min}},$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of A respectively. The preconditioners goal is to lower this condition number each iteration such that the PCG solver takes less of them to converge to the solution. The key difference between the PCG method and the base CG method is that the PCG uses the preconditioned residual to update the search directions, rather than the residual from the original system.

Algorithm 1 Preconditioned Conjugate Gradient (PCG)

Require: F_A, $F_{M^{-1}}$, $\vec{b}$, $\vec{x}$, ε, $i_{\max}$
  $i \leftarrow 0$
  $\vec{r} \leftarrow \vec{b} - F_A(\vec{x})$
  $\vec{d} \leftarrow F_{M^{-1}}(\vec{r})$
  $\delta_{\text{new}} \leftarrow \vec{r}^\top \vec{d}$
  $\delta_0 \leftarrow \delta_{\text{new}}$
  while $i < i_{\max}$ and $\delta_{\text{new}} > \varepsilon^2 \delta_0$ **do**
    $\vec{q} \leftarrow F_A(\vec{d})$
    $\alpha \leftarrow \frac{\delta_{\text{new}}}{\vec{d}^\top \vec{q}}$
    $\vec{x} \leftarrow \vec{x} + \alpha \vec{d}$
    $\vec{r} \leftarrow \vec{r} - \alpha \vec{q}$
    $\vec{s} \leftarrow F_{M^{-1}}(\vec{r})$
    $\delta_{\text{old}} \leftarrow \delta_{\text{new}}$
    $\delta_{\text{new}} \leftarrow \vec{r}^\top \vec{s}$
    $\beta \leftarrow \frac{\delta_{\text{new}}}{\delta_{\text{old}}}$
    $\vec{d} \leftarrow \vec{s} + \beta \vec{d}$
    $i \leftarrow i + 1$
  end while
  return $\vec{x}$

2.2 Preconditioners

2.2.1 Preliminaries

Before starting with a specific preconditioner there are some similarities between all of them which help us understand their connection to each other.

In the case of this thesis the structure of applying the preconditioner to the system (2.1) is done by passing the residual used in the CG iterations,

$$\vec{r} = \vec{b} - A\vec{x},$$

to the preconditioner which then in return solves a linear system using the residual as the right hand side and the matrix A as in the original problem (2.1). For the rest of this chapter the problem solved by the preconditioners takes the following form:

$$A\vec{x} = \vec{b},$$

where we define $\vec{b}$ as the residual passed to the preconditioner.

All considered preconditioners are iterative methods using the following splitting of the matrix A:

$$A = M - N.$$

Taking this splitting we can rewrite our problem into a new form

$$\vec{x} = M^{-1}(\vec{b} + N\vec{x}),$$

which be can used to get the following iterative scheme:

$$\vec{x}^{(k+1)} = M^{-1}(\vec{b} + N\vec{x}^{(k)}), \quad k = 0, 1, \ldots \text{ and } \vec{x}^{(0)} \text{ as an initial guess.} \tag{2.2}$$

To derive the specific preconditioners each uses an additional splitting of A in to $A = L + D + U$, with D as the diagonal of A and L and U the strictly upper and lower triangular matrices respectively.

2.2.2 Jacobi-Richardson

The simplest case of our preconditioners is the Jacobi-Richardson method. It uses the derivation 2.2 with $M \equiv D$ and consequently $N \equiv -(L + U)$ which results in the following iterative scheme:

$$\vec{x}^{(k+1)} = D^{-1}(\vec{b} - (L + U)\vec{x}^{(k)}). \tag{2.3}$$

In Algorithm 2 a pseudo-code of the exact implementation of (2.3) can be found.

Algorithm 2 Jacobi-Richardson

Require: $F_{D^{-1}}$, F_{L+U}, $\vec{b}$, k_{max}
  $\vec{x} \leftarrow 0$
  for $k \leftarrow 1$, k_{max} **do**
    $\vec{r} \leftarrow \vec{b} - F_{L+U}(\vec{x})$
    $\vec{x} \leftarrow F_{D^{-1}}(\vec{r})$
  end for
  return $\vec{x}$

2.2.3 Symmetric Two-step Gauss-Seidel

The symmetric Gauss-Seidel method uses the same idea as the other preconditioners but uses a different splitting of A.

To be exact, it uses $M \equiv L + D$ and $M \equiv U + D$ for its forward and backward sweeps, respectively, which will be explained later. For now, we focus only on the former and obtain:

$$\vec{x}^{(k+1)} = (L + D)^{-1}(\vec{b} - U\vec{x}^{(k)}). \tag{2.4}$$

In an ordinary setting this equation is transformed further to be able to solve it with a simple forward substitution by making use of the forms of L and U

$$\vec{x}^{(k+1)} = D^{-1}(\vec{b} - U\vec{x}^{(k)} - L\vec{x}^{(k+1)}). \tag{2.5}$$

For an efficient and matrix-free implementation of the Gauss-Seidel method the equations (2.4) and (2.5) are not sufficient as they either need a function $F_{(L+D)^{-1}}(x)$ or a sequential implementation.

The symmetric two-stage Gauss-Seidel tries to eliminate these problems by replacing the forward substitution with an additional iterative solve which is inspired by [1]. This is done by defining $\vec{g} := \vec{x}^{(k+1)}$ and $r := \vec{b} - U\vec{x}^{(k)}$ which yields a new system:

$$(L + D)\vec{g} = \vec{r}.$$

Furthermore we solve this equation with the Jacobi-Richardson method from subsection 2.2.2 which becomes

$$\vec{g}^{(j+1)} = D^{-1}(\vec{r} - L\vec{g}^{(j)}).$$

The solution for $\vec{g}$ after a predefined number of iterations n_{inner} will be $\vec{x}^{(k+1)}$.

As mentioned in the beginning this is only one part of the symmetric Gauss-Seidel method and to make it symmetric the same steps are done with $M \equiv U + D$. The reason for this is best seen if we look at the forward substitution which can be used to solve (2.5):

$$x_i^{(k+1)} = \frac{1}{a_{ii}}(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}). \tag{2.6}$$

This equation shows the computation for one element of $\vec{x}^{(k+1)}$ with indices $i = 1...n$. It shows that certain parts of $\vec{x}^{(k+1)}$ with lower indices get updated mostly by $\vec{x}^{(k)}$ and others with higher by already computed values of $\vec{x}^{(k+1)}$, due to the structure of L and U.

Like the Jacobi-Richardson method in practice this procedure is done in-place and the implementation of it with matrix-free operators can be viewed in Algorithm 3.

Algorithm 3 Symmetric Two-step Gauss-Seidel

Require: $F_{D^{-1}}$, $F_{L,}$, F_U, $\vec{b}$, k_{inner}, k_{outer}
  $\vec{x} \leftarrow 0$
  for $k \leftarrow 1$, k_{outer} **do**
    $\vec{r} \leftarrow \vec{b} - F_U(\vec{x})$
    for $j \leftarrow 1$, k_{inner} **do**
      $\vec{r}_{inner} \leftarrow \vec{r} - F_L(\vec{x})$
      $\vec{x} \leftarrow F_{D^{-1}}(\vec{r}_{inner})$
    end for
    $\vec{r} \leftarrow \vec{b} - F_L(\vec{x})$
    for $j \leftarrow 1$, k_{inner} **do**
      $\vec{r}_{inner} \leftarrow \vec{r} - F_U(\vec{x})$
      $\vec{x} \leftarrow F_{D^{-1}}(\vec{r}_{inner})$
    end for
  end for
  return $\vec{x}$

2.2.4 Symmetric Successive Over-Relaxation

Finally the symmetric successive over-relaxation takes the same idea as the Gauss-Seidel method but extends the problem with a relaxation parameter ω. To understand the method further we start with the modified problem

$$\omega A\vec{x} = \omega b$$

and follow the same steps but use a slightly different reordering made possible by ω:

$$(D + \omega L)\vec{x} = \omega\vec{b} - \omega U\vec{x} + (1 - \omega)D\vec{x},$$

$$\vec{x} = (D + \omega L)^{-1}(\omega\vec{b} + ((1 - \omega)D - \omega U)\vec{x}).$$

This further leads to this iterative scheme:

$$\vec{x}^{(k+1)} = (D + \omega L)^{-1}(\omega\vec{b} + ((1 - \omega)D - \omega U)\vec{x}^{(k)}),$$

which gives us the splitting of A as $M \equiv \frac{1}{\omega}D + L$ and $N \equiv (\frac{1}{\omega} - 1)D - U$. Now the same as in (2.4) we can write this problem differently to be able to use a forward substitution to solve it:

$$\vec{x}^{(k+1)} = D^{-1}(\omega\vec{b} + ((1 - \omega)D - \omega U)\vec{x}^{(k)} - \omega L\vec{x}^{(k+1)}).$$

Again this approach leaves us with the same problems as the Gauss-Seidel method. To solve these the same two-stage method is applied and we end

up with an inner solve for $\vec{g} := \vec{x}^{(k+1)}$ with $\vec{r} := \omega\vec{b} + ((1-\omega)D - \omega U)\vec{x}^{(k)}$, which gives rise to the following iterative scheme:

$$\vec{g}^{(j+1)} = D^{-1}(\vec{r} - \omega L\vec{g}^{(j)}).$$

It is important to know for the SSOR method to converge ω needs to fulfill the condition $|\omega - 1| < 1$ or in other words $\omega \in (0, 2)$ [3].

The implementation is based on the Gauss-Seidel method and can be seen in the pseudo-code in Algorithm 4.

Algorithm 4 Symmetric successive over-reaxation

Require: F_D, $F_{D^{-1}}$, F_L, F_U, ω, $\vec{b}$, k_{inner}, k_{outer}
  $\vec{x} \leftarrow 0$
  for $k \leftarrow 1$, k_{outer} **do**
    $\vec{r} \leftarrow \omega\vec{b} - \omega F_U(\vec{x}) + (1-\omega)F_D(\vec{x})$
    for $j \leftarrow 1$, k_{inner} **do**
      $\vec{r}_{inner} \leftarrow \vec{r} - \omega F_L(\vec{x})$
      $\vec{x} \leftarrow F_{D^{-1}}(\vec{r}_{inner})$
    end for
    $\vec{r} \leftarrow \omega\vec{b} - \omega F_L(\vec{x}) + (1-\omega)F_D(\vec{x})$
    for $j \leftarrow 1$, k_{inner} **do**
      $\vec{r}_{inner} \leftarrow \vec{r} - \omega F_U(\vec{x})$
      $\vec{x} \leftarrow F_{D^{-1}}(\vec{r}_{inner})$
    end for
  end for
  return $\vec{x}$

Chapter 3

IPPL Implementation

The IPPL framework already provides a good basis for extending to an additional preconditioner with the matrix-free approach. The following section will showcase how matrix-free operators are implemented in IPPL and provide the detailed implementation of the SSOR method.

3.1 Matrix Free Operators

Currently the PCG solver is only concretely used to solve problems of type Poisson:

$$\nabla^2 \varphi = \rho,$$

where ∇^2 is the Laplace operator giving us the the second order derivative in each dimension. When discretizing this PDE to the form $A\vec{x} = \vec{b}$, the matrix A is defined by the Laplacian stencil used for numerical evaluation of the Laplace operator:

$$\nabla^2 u(\vec{x}) = \sum_{j=1}^{n} \frac{u(\vec{x} + h_j \vec{e}_j) - 2u(\vec{x}) + u(\vec{x} - h_j \vec{e}_j)}{h_j^2} + \mathcal{O}(h^2),$$

with h_j being the mesh spacing and $\vec{e}_j$ the unit vectors of the n dimensions. Using this evaluation it is clear to see that the matrix gets sparse because every grid-point depends only on the value at the grid-point and the immediate neighbours. This approach allows us to apply the stencil efficiently at every point without constructing or storing the full matrix.

Using this stencil we can do the same for each of the needed operators by the preconditioners. The upper and lower triangular operators benefit most from this matrix-free approach as they need to do only one calculation per point in each dimension to get to the solution.

Code 3.1 outlines the operator for the diagonal of the Laplacian matrix and implements the following:

$$Du(\vec{x}) = -2 \cdot (\sum_{j=1}^{n} \frac{1}{h_j^2})\, u(\vec{x}).$$

```
1   /*!
2    * Diagonal Laplace operator
3    */
4   template <typename Field>
5   double diagonal_laplace(Field& u) {
6       constexpr unsigned Dim = Field::dim;
7       using mesh_type        = typename Field::Mesh_t;
8       mesh_type& mesh        = u.get_mesh();
9       double sum     = 0.0;
10      for (unsigned d = 0; d < Dim; ++d) {
11          sum += 1/(Kokkos::pow(mesh.getMeshSpacing(d), 2));
12      }
13      return - 2.0 * sum;
14  }
```

Code 3.1: Diagonal Laplace operator in IPPL.

When working with a distributed domain these operators run into a problem when calculating values on the boundary of their subdomain as some of the neighbouring grid-points are part of a different subdomain controlled by an other MPI rank. This problem get mitigated by introducing halo cells containing all necessary values form the other rank. The challenge arising from this is the communication overhead created by updating the halo cells. A detailed description of how this is handled in IPPL can be found in [6]. For simplicity and improved performance [2] all preconditioners use no communication among ranks by not using halo-cells because we assume that their contribution does not affect our methods in a significant way.

3.2 Preconditioners

The structure of the already provided preconditioners has been refined to reduce re-allocating memory each iteration. This gets rid of the initialization overhead and is also used for implementing the SSOR method. In Code 3.2 the function implementing the SSOR preconditioner, shown in Algorithm 4, in IPPL can be found, showcasing the in-situ computation of the solution.

```
1   /*!
2    * Symmetric successive over-relaxation (SSOR)
3    */
4   template <typename Field, typename LowerF, typename UpperF, typename
        InvDiagF, typename DiagF>
5   struct ssor_preconditioner : public preconditioner<Field> {
6
7       // Constructor, Initialization
8
```

```
9           Field operator()(Field& b) override {
10
11              double D; // Diagonal term
12              layout_type& layout = b.getLayout();
13              mesh_type& mesh     = b.get_mesh();
14              Field x(mesh, layout); // Solution vector
15              x = 0;  // Initial guess
16
17              // Outer iterations for SSOR
18              for (unsigned int k = 0; k < outerloops_m; ++k) {
19                  // Update residual r
20                  UL_m = upper_m(x);  // Apply upper triangular
21                  D = diagonal_m(x);  // Apply diagonal
22                  r_m = omega_m * (b - UL_m) + (1.0 - omega_m) * D * x;
23
24                  // Inner iterations
25                  for (unsigned int j = 0; j < innerloops_m; ++j) {
26                      UL_m = lower_m(x);  // Apply lower triangular
27                      x = r_m - omega_m * UL_m;  // Update solution
28                      x = inverse_diagonal_m(x) * x;  // Apply inverse
                            diagonal
29                  }
30
31                  // Update residual for second pass
32                  UL_m = lower_m(x);  // Apply lower triangular
33                  D = diagonal_m(x);  // Apply diagonal
34                  r_m = omega_m * (b - UL_m) + (1.0 - omega_m) * D * x;
35
36                  // Second inner iteration
37                  for (unsigned int j = 0; j < innerloops_m; ++j) {
38                      UL_m = upper_m(x);  // Apply upper triangular
39                      x = r_m - omega_m * UL_m;  // Update solution
40                      x = inverse_diagonal_m(x) * x;  // Apply inverse
                            diagonal
41                  }
42              }
43              return x;
44          }
45
46      protected:
47          // Member variables, etc.
48      };
```

Code 3.2: Symmetric successive over-relaxation implemented in IPPL.

Chapter 4

Results

In this chapter we show correctness and performance analyses of the aforementioned preconditioners when used in conjunction with the CG solver for three different problems. A comparison of the different preconditioners is also presented.

4.1 Setup

All tests were run on two of PSI's distinct high-performance computing platforms to show portability using multiple MPI ranks for parallel execution. For testing CPU scalability, they were conducted on the Merlin6 cluster. Each node of the cluster is using an Intel® Xeon® Gold 6152 Scalable Processor at 2.10 GHz with 44 cores over two sockets and 384GB RAM and NVMe scratch memory. The second architecture is given by the Gwendolen GPU cluster, where each rank is handled by on of the eight available NVIDIA A100 Tensor Core GPUs combining to 320 GB of total memory. They are managed by dual AMD Rome 7742 processors at 2.25 GHz with a total of 128 cores and access to 1 TB of system memory.

The first problem we will solve is the Poisson equation from [6]:

$$
\begin{aligned}
-\Delta\varphi = \pi^2 \Big[& \cos(\sin(\pi z))\sin(\pi z)\sin(\sin(\pi x))\sin(\sin(\pi y)) \\
& + (\cos(\sin(\pi y))\sin(\pi y)\sin(\sin(\pi x)) + (\cos(\sin(\pi x))\sin(\pi x) \\
& + (\cos^2(\pi x) + \cos^2(\pi y) + \cos^2(\pi z))\sin(\sin(\pi x)))\sin(\sin(\pi y)))\sin(\sin(\pi z)) \Big],
\end{aligned}
$$

for which the solution is known and can be used to test for correctness by evaluating the relative error of the computed solution with respect to the analytical result, given by:

$$\varphi = \sin(\sin(\pi x))\sin(\sin(\pi y))\sin(\sin(\pi z)) \tag{4.1}$$

In addition, we use two mini-apps from ALPINE [4] for testing. We use Landau Damping, a benchmark problem in plasma physics, for correctness analysis and scaling studies. The third problem is is the Penning Trap simulation, which we also use for scaling, as it uses a less uniformly distributed system compared to Landau Damping, where load balancing becomes more of an issue.

4.2 Correctness

In Figure 4.1 the relative error of the CG solvers solution to the analytical solution in (4.1) can be found. The results show that all methods achieve second-order convergence, as evidenced by the linear slope in the log-log plot. This indicates that the solvers converge at the expected rate, achieving the desired accuracy consistent with second-order convergence.

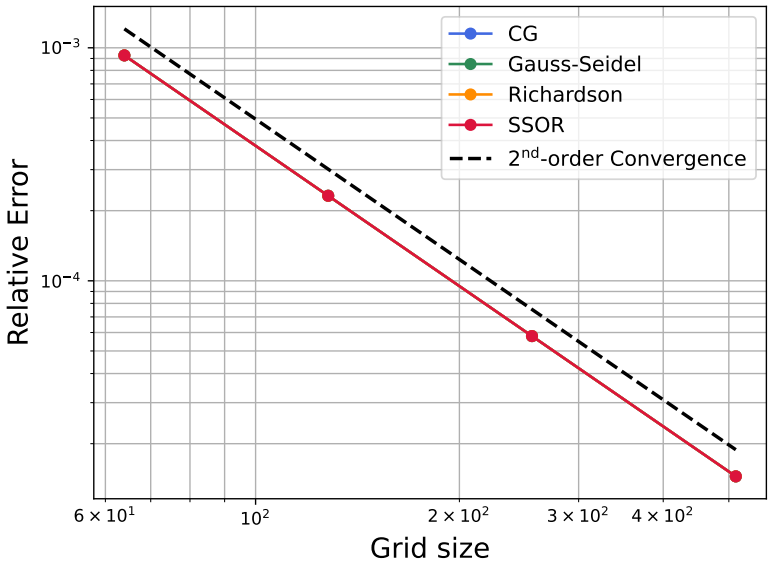

Figure 4.1: Relative error to analytical solution.

The same conclusion can be drawn when looking at the electric field energy in x-direction of Landau Damping using a 32^3 grid with 83′886′080 particles in Figure 4.2, where the damping rate matches the known analytical rate for all preconditioners.

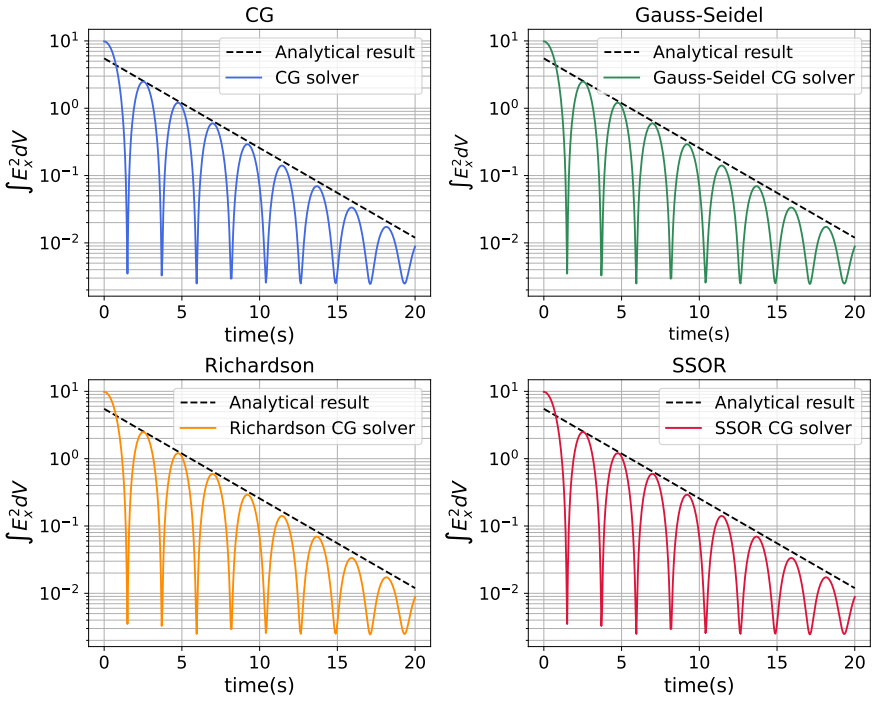

Figure 4.2: Electric field energy in x-direction of Landau Damping.

4.3 Scaling

The scaling studies are separated into the two tested problems, laying out both CPU and GPU efficiency of their usage with the PCG solver and some insight into the different preconditioners. To quantify the scaling, we use speedup S_p and efficiency E_p defined by:

$$S_p = \frac{T_1}{T_p}, \quad E_p = \frac{S_p}{p},$$

where p is the number of ranks and T_p the time to solution using p ranks. All test are conducted on the same 256^3 grid with 134′217′728 particles.

4.3.1 Landau Damping

In Figure 4.3 we can see the total time to solution of the solvers compared to each other. One should note that all preconditioners overtake and are faster than the base CG when using more GPUs. This is nicely portrayed in Figure 4.3 on the right where it clearly shows the speedup of all preconditioners being much better, with the SSOR scaling the best and keeping efficiency above 40% up to 8 GPUs.

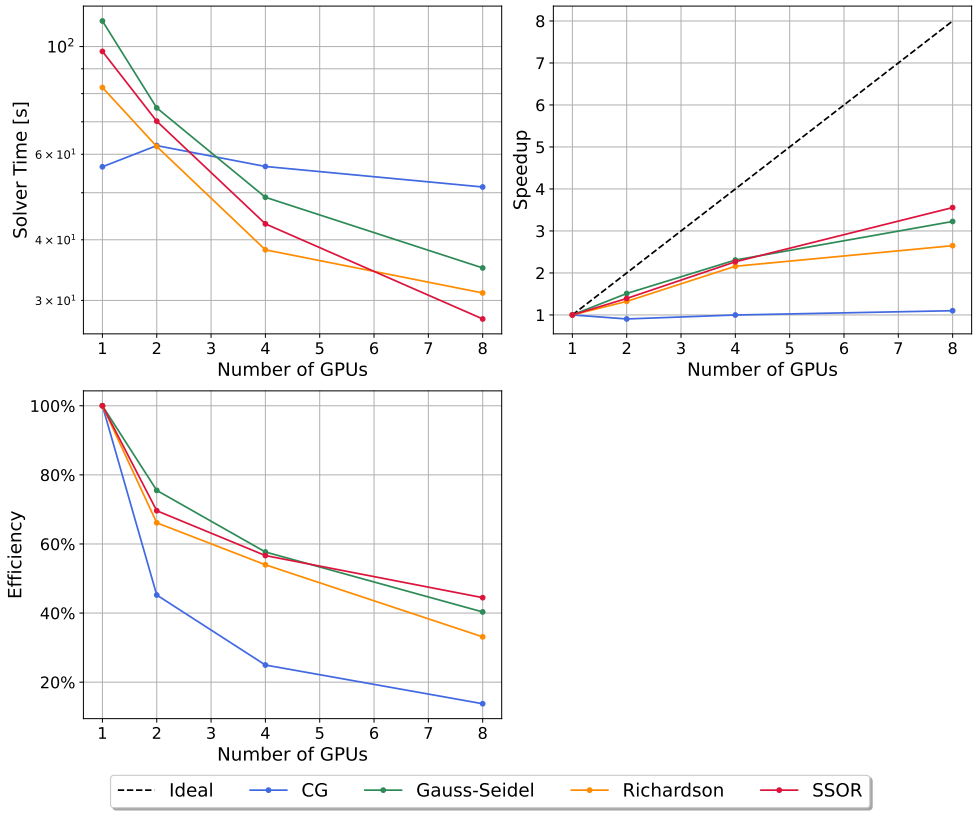

Figure 4.3: Strong scaling (top left), speedup top right) and efficiency (bottom left) for Landau Damping on GPU.

The same comparison using the OpenMP build for CPU testing can be viewed in Figure 4.4. In this case, while the preconditioners still show very good efficiency, the absolute time of the base CG is overall faster. The speedup for all preconditioners even show superlinear scaling suggesting that the domain decomposition and the locality of the preconditioner have an influence on the final time. This became more evident as the load balancing threshold needed to be adjusted due to problems with unbalanced workload across MPI rank when using 16 ranks.

Overall the preconditioners show very similar results with good scalability. To better see their performance on GPU in Landau Damping, their time per iteration to calculate the preconditioned system can be found in Figure 4.5. There unsurprisingly we can tell the different complexity of each has an influence on its calculation time and less of an impact on their efficiency which stays above 80% for all of them.

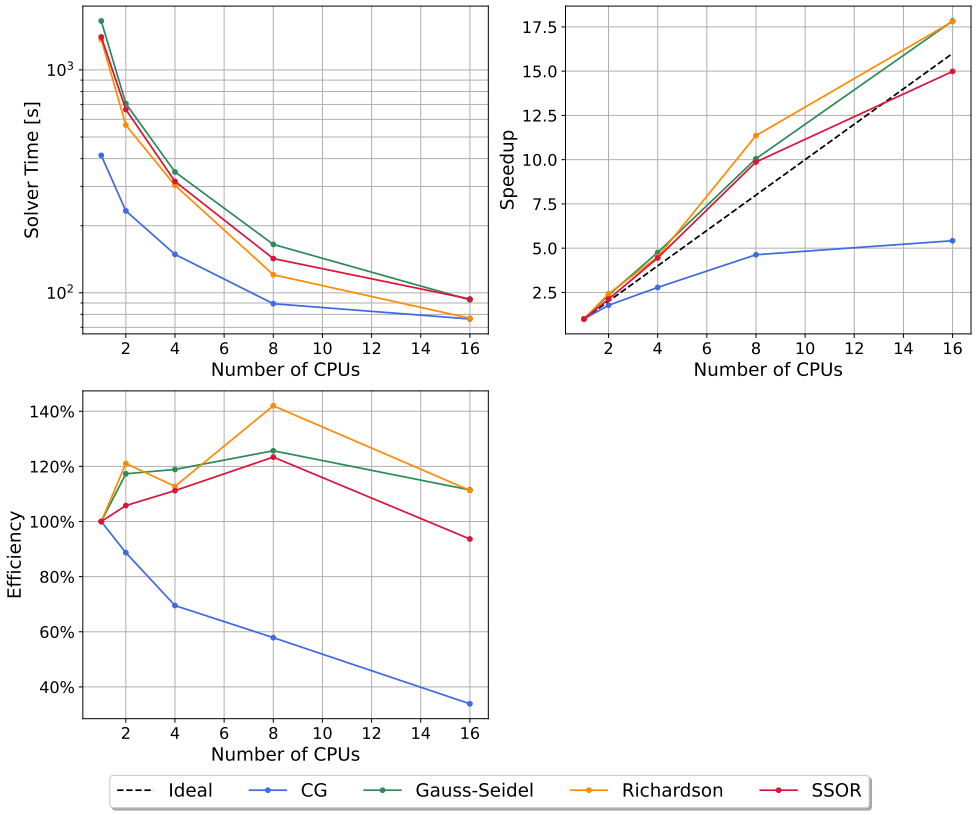

Figure 4.4: Strong scaling (top left), speedup top right) and efficiency (bottom left) for Landau Damping on CPU.

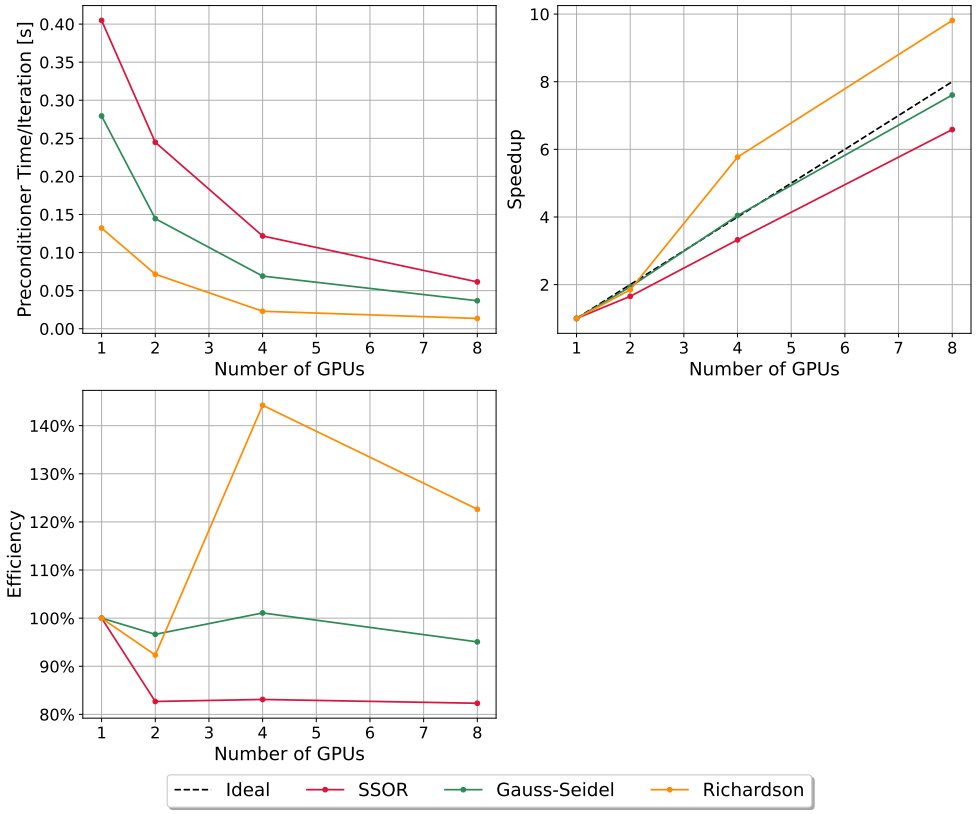

Figure 4.5: Strong scaling (top left), speedup top right) and efficiency (bottom left) of preconditioners for Landau Damping on GPU.

Important to note here is that they do have different number of iterations to get to the solution which can be found in Table 4.1. Here it shows us the effect each preconditioner has on the PCG solver by lowering the condition number of the matrix. The methods again behave as expected and the SSOR stands out by reducing the iteration count to less than 25% of the base CG.

Test	CG	Gauss-Seidel	Richardson	SSOR
Landau Damping	768.76	316.67	386.12	203.33
Penning Trap	747.11	424.18	508.21	262.49
CG Test	225	226	176	171

Table 4.1: Average number of CG iterations to solution.

Examining those results there is a clear trade off between iteration count and computational overhead by the preconditioners. With good scalability the SSOR solver benefits on bigger machines where the number of iterations has a greater influence on the time as the computational overhead gets smaller.

4.3.2 Penning Trap

When looking at the Penning Trap tests we get to similar results for GPU in Figure 4.6 and CPU in Figure 4.7 as for Landau Damping. The GPU performance for the Penning Trap gives us the SSOR method as the best method for larger system due to its scaling capabilities. For CPU architectures the base CG remains faster throughout every number of CPU nodes but there are again problems with load balancing when exceeding 8 nodes. Contrary to Landau Damping using a different load balancing threshold did not fix the problem and slowed down all other runs. This is due the very non-uniform distribution of the Penning Trap problem [4] making it more difficult to distribute work equally among ranks.

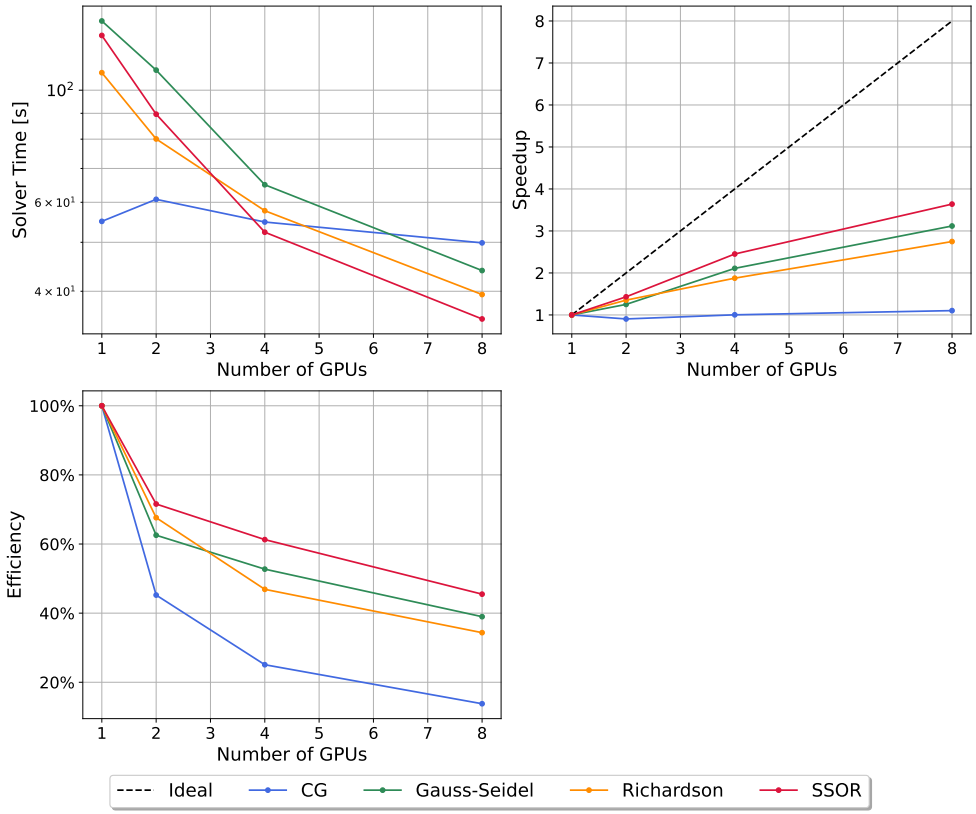

Figure 4.6: Strong scaling (top left), speedup top right) and efficiency (bottom left) for Penning Trap on GPU.

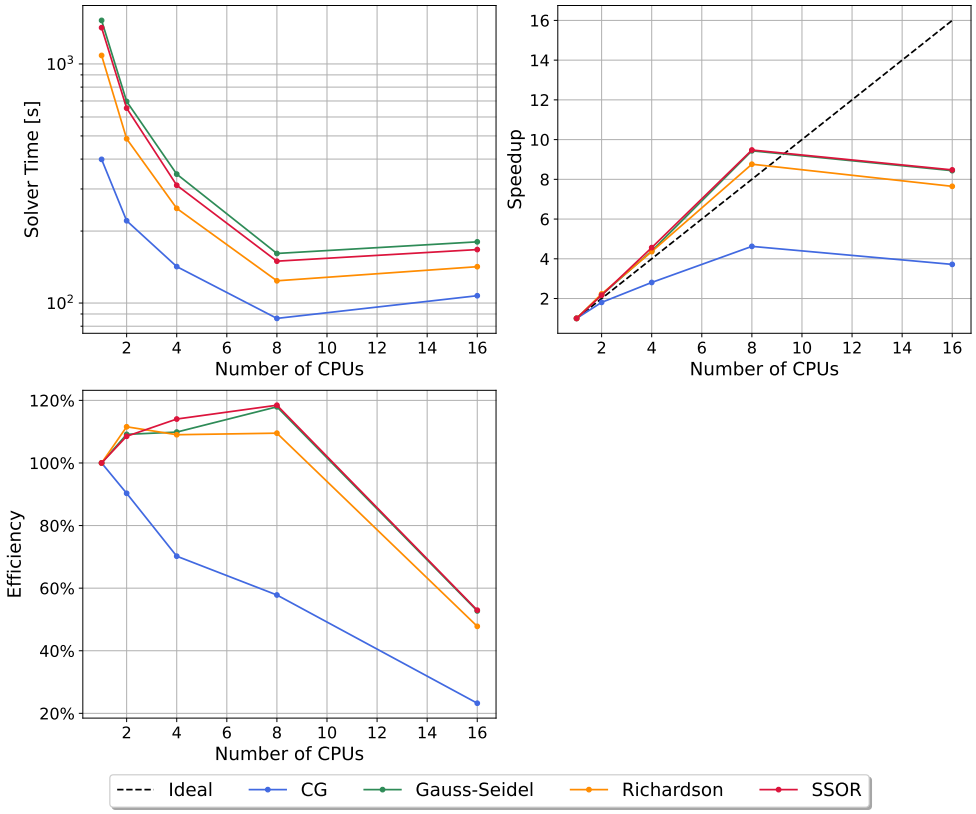

Figure 4.7: Strong scaling (top left), speedup top right) and efficiency (bottom left) for Penning Trap on CPU.

Chapter 5

Conclusion

The results demonstrate significant progress in efficiency and performance of the PCG solver in the IPPL framework. Incorporating the new SSOR method proved to be highly effective, since it significantly reduced the iteration count of the CG solver when using a preconditioner. Correctness studies confirmed the reliability and consistency across multiple problems when comparing to analytical results. Scaling studies emphasize the potential of SSOR and the other preconditioners on even larger systems especially in GPU environments.

However, there is room for improvement in terms of absolute timings and scaling, especially visible on the CPU scalings, which require further optimization for the preconditioners to be more effective. To better understand this and other potential shortcomings, it would be beneficial to construct a flow-graph outlining more of the structure of each preconditioner. A detailed analysis of time spent per iteration in the preconditioner would also help pinpoint performance bottlenecks in the current code which could explain the high absolute timings. A different and interesting possibility for further improvement suggested by [1], was to limit the preconditioners to single precision making the computations even faster. Additionally, further research could be done on the parameters of each method and how we can change them to better control the balance between cutting down the number of iterations and the computational overhead of the preconditioners.

An intriguing next step would be to analyze the performance of the preconditioners on larger problems and more powerful computing systems, to see how they will scale further than the used resources. But the PCG solver is currently limited to solving Poisson-type problems because only the Laplacian matrix has the matrix-free operators needed to utilize the preconditioners. By implementing other operators in a matrix-free manner, we could extend IPPL to use preconditioning and iterative solvers to solve other relevant PDEs.

Acknowledgements

I would like to thank the IPPL and AMAS group for their valuable support and insights during this work. I want to specially thank Dr. Andreas Adelmann for providing me with the opportunity to work on this project, and to Sonali Mayani, who was always available to assist whenever I encountered difficulties. I am also appreciative of everyone who took time to proofread my thesis.

Bibliography

[1] Berger-Vergiat, L., Kelley, B., Rajamanickam, S., Hu, J., Swirydowicz, K., Mullowney, P., Thomas, S., and Yamazaki, I. Two-stage gauss–seidel preconditioners and smoothers for krylov solvers on a gpu cluster, 2021.

[2] B.Schreiner. A performance portable matrix-free preconditioner for the conjugate gradient solver, 2024.

[3] Hadjidimos, A. Successive overrelaxation (sor) and related methods. *Journal of Computational and Applied Mathematics 123*, 1 (2000), 177–199. Numerical Analysis 2000. Vol. III: Linear Algebra.

[4] Muralikrishnan, S., Frey, M., Vinciguerra, A., Ligotino, M., Cerfon, A., Stoyanov, M., Gayatri, R., and Adelmann, A. Scaling and performance portability of the particle-in-cell scheme for plasma physics applications through mini-apps targeting exascale architectures" , 2022.

[5] Shewchuk, J. R. An introduction to the conjugate gradient method without the agonizing pain. Tech. rep., USA, 1994.

[6] Vinciguerra, A. A performance portable conjugate gradient solver, 2021.

Appendix

Reproducing Results

The source code of the IPPL library and the ALPINE mini-apps both can be found on GitHub at https://github.com/mbollige/ippl. The Readme includes a run through of how to build and compile IPPL with APLINE for different machines. In our test, this involved a build for OpenMP and Cuda on the Ampere architecture.

Running Tests

Both ALPINE tests are run the same way and only differ in their executable which are `./LandauDamping` and `./PenningTrap` respectively. The following description is only for running Landau Damping but can be easily changed for Penning Trap.

For testing correctness a new data directory in our build folder needs to be created, which the test will dump the data diagnostics in for each timestep. To run the test following commands need to be run for the OpenMP build:

```
$ cd ./ippl/build_openmp/alpine
$ mkdir data
$ srun -n 2 ./LandauDamping 256 256 256 134217728 20 PCG 0.01
    LeapFrog ssor 4 2 1.57079632679 --overallocate 2.0 --info 10
```

This runs a simulation on 2 CPU nodes with a 256^3 grid and 134217728 particles using the SSOR preconditioner with 4 inner, 2 outer iterations and a dampening factor of $\frac{\pi}{2}$ and put the resulting data in the `data/` folder.

For simplicity, slurm jobscripts were used to run on the PSI clusters. All scripts used to run and extract timing data can be found in https://gitlab.psi.ch/AMAS-students/bolliger_bsc including all data obtained and used for this work.

The following is an example for batch scripts used in this work. It is called by executing `strong.sh` given in Code 1:

```
$ sh strong.sh
```

This will run through all preconditioners with 1-8 GPUs calling `run.sh` in Code 2 each time and run them on the GPU cluster Gwendolen of PSI:

```
for i in 0 1 2 3
  do
  N=$((2**$i))
    sbatch --gpus=$N --ntasks=$N --output=scaling_Gauss-Seidel_${N}.out --error=
        scaling_error_Gauss-seidel_${N}.err run.sh 256 256 256 gauss-seidel 2 2 0
    sbatch --gpus=$N --ntasks=$N --output=scaling_Richardson_${N}.out --error=
        scaling_error_Richardson_${N}.err run.sh 256 256 256 richardson 4 0
    sbatch --gpus=$N --ntasks=$N --output=scaling_SSOR_${N}.out --error=
        scaling_error_SSOR_${N}.err run.sh 256 256 256 ssor 4 2 1.57079632679
done
```

Code 1: strong.sh used to initialize all different tests

```
#!/bin/bash
#SBATCH --time=00:40:00
#SBATCH --nodes=1
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen # Running on the Gwendolen partition of the GPU cluster
#SBATCH --account=gwendolen
#SBATCH --exclusive

N1=$1
N2=$2
N3=$3
preconditioner=$4
P1=$5
P2=$6
P3=$7

srun ./LandauDamping $N1 $N2 $N3 134217728 20 PCG 0.05 LeapFrog $preconditioner $P1
    $P2 $P3 --overallocate 2.0 --info 10 --kokkos-map-device-id-by=mpi_rank
```

Code 2: run.sh used to run tests with preconditioner

As in Code 2 can be seen the preconditioner parameters are passed after the time-stepping method. For our preconditioners this includes:

- Richardson:
  - Number of iteratrions
  - Communication
- Gauss-Seidel:
  - Number of inner iterations
  - Number of outer iterations
  - Communication

- SSOR:
  - Number of inner iterations
  - Number of outer iterations
  - Dampening factor

For Richardson the unused parameter can be left empty. To run the base CG the run command needs to be changed a bit to clarify only CG:

```
#!/bin/bash
#SBATCH --time=00:40:00
#SBATCH --nodes=1
#SBATCH --clusters=gmerlin6
#SBATCH --partition=gwendolen # Running on the Gwendolen partition of the GPU cluster
#SBATCH --account=gwendolen
#SBATCH --exclusive

N1=$1
N2=$2
N3=$3
preconditioner=$4
P1=$5
P2=$6
P3=$7

srun ./LandauDamping $N1 $N2 $N3 134217728 20 CG 0.05 LeapFrog --overallocate 2.0 --
     info 10 --kokkos-map-device-id-by=mpi_rank
```

Code 3: run.sh used to run tests with CG

Code overview

The code described in Chapter 3. can be found in the `src/` folder of IPPL. The PCG algorithm using the preconditioners is located in `src/LinearSolvers/PCG.h` wich uses the preconditioner classes defined in `src/LinearSolvers/Preconditioner.h`. The mentioned matrix-free Laplace and decomposed operators passed to the PCG are located in the file `src/PoissonSolvers/LaplaceHelpers.h`.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Eigenständigkeitserklärung

Die unterzeichnete Eigenständigkeitserklärung ist Bestandteil jeder während des Studiums verfassten schriftlichen Arbeit. Eine der folgenden drei Optionen ist in Absprache mit der verantwortlichen Betreuungsperson verbindlich auszuwählen:

- (●) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Es wurden keine Technologien der generativen künstlichen Intelligenz[1] verwendet.
- ( ) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[2] verwendet und gekennzeichnet.
- ( ) Ich bestätige, die vorliegende Arbeit selbständig und in eigenen Worten verfasst zu haben, namentlich, dass mir niemand beim Verfassen der Arbeit geholfen hat. Davon ausgenommen sind sprachliche und inhaltliche Korrekturvorschläge durch die Betreuungsperson. Als Hilfsmittel wurden Technologien der generativen künstlichen Intelligenz[3] verwendet. Der Einsatz wurde, in Absprache mit der Betreuungsperson, nicht gekennzeichnet.

Titel der Arbeit:

A Matrix Free Preconditioner for Exascale Computing

Verfasst von:
Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich.

Name(n):	**Vorname(n):**
Bolliger	Matteo

Ich bestätige mit meiner Unterschrift:
- Ich habe mich an die Regeln des «Zitierleitfadens» gehalten.
- Ich habe alle Methoden, Daten und Arbeitsabläufe wahrheitsgetreu und vollständig dokumentiert.
- Ich habe alle Personen erwähnt, welche die Arbeit wesentlich unterstützt haben.

Ich nehme zur Kenntnis, dass die Arbeit mit elektronischen Hilfsmitteln auf Eigenständigkeit überprüft werden kann.

Ort, Datum	**Unterschrift(en)**
Volketswil, 16.12.2024	

Bei Gruppenarbeiten sind die Namen aller Verfasserinnen und Verfasser erforderlich. Durch die Unterschriften bürgen sie grundsätzlich gemeinsam für den gesamten Inhalt dieser schriftlichen Arbeit.

[1] z. B. ChatGPT, DALL E 2, Google Bard
[2] z. B. ChatGPT, DALL E 2, Google Bard
[3] z. B. ChatGPT, DALL E 2, Google Bard