Universitat politècnica de catalunya

Facultat d'Informàtica de Barcelona

SCA

Lab4: Parallelize a 3D Poisson problem

November 4, 2024

Authors: Stefano Petrilli
Javier Beiro Piñón
Professor: CARLOS ALVAREZ MARTINEZ
Academic year: 24/25
Module: Numerical

Contents

List of Figures

List of Tables

Listings

1 Introduction

The 3D Poisson solver is a computational algorithm designed to solve the Poisson equation, which is a partial differential equation of the form

$$\nabla^2 u(x, y, z) = f(x, y, z),$$

where ∇^2 denotes the Laplace operator, $u(x, y, z)$ is the unknown function to be solved for, and $f(x, y, z)$ represents the source term. The discrete form, transforms the continuous PDE into a matrix equation:

$$Au = f$$

where A is a matrix representing the discrete Laplacian operator, u is the vector of unknowns, and f is the vector representing the discrete right-hand side function f.

This lab report concerns the parallelization of a toy 3D Poisson problem solver based on preconditioned conjugate gradient iteration. All the data reported in the following report are measured in the execution queue of `BOADA`.

2 The sequential version

The starting point for this work is the basic sequential version of the solver. The solver takes as input the parameters visible in Listing 1.

```
cgp3d
  -h: print this message
  -n: mesh size (100)
  -M: maximum iteration count (200)
  -r: relative residual tolerance (1e-6)
  -p: preconditioner: as, ssor, or id (id)
  -w: SSOR relaxation parameter (1.9)
  -o: Schwarz method overlap (10)
```

Listing 1: 3D Poisson problem parameters

From Listing 1 we can see between parenthesis the default parameters. In the case of the mesh size, the default parameter is 100.

If we execute the sequential version using the default parameters, the output can be seen in Listing 2.

```
199 steps, residual reduction 9.0582e-07 (<= tol 1e-06); time 2.1691 seconds
rnorm = 8.45154e-08
```

Listing 2: Sequential version, output of default parameters execution.

From 2 we can notice that it takes **2.16 seconds** and **199 iterations** to reach a point where the relative residual is lower than the default **relative residual tolerance**, which is $1e-6$. The **maximum iteration count** is enough in this case to reach a solution, as the default parameters is **200 iterations**.

```
#!/bin/bash

#SBATCH --job-name=submit-seq.sbatch
#SBATCH -D .
#SBATCH --output=execution_result.out

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean
make -C "/scratch/nas/1/sca1011/CSA/lab4"

../cgp3d.x -n 200

../cgp3d.x -n 200 -M 500
```

Listing 3: Slurm task with bigger mesh and higher iteration count.

```
200 steps, residual reduction 0.0553634 (> tol 1e-06); time 18.6042 seconds
rnorm = 0.00372623
403 steps, residual reduction 9.50542e-07 (<= tol 1e-06); time 37.0298 seconds
rnorm = 2.65834e-07
```

Listing 4: Output of the task in Listing 3

If we run the SLURM script visible in Listing 3, we get the output visible in Listing 4. From the output, we can conclude that, with a mesh size of 500, the residual does not get lower than the residual limit within the limit of 200 iterations. On the second execution, which has a mesh size of 200 but also a maximum iteration count of 500, the residual gets below the limit after the 403rd iteration. To reach the solution with a bigger mesh, the execution time is **37.0 seconds**. It's also worth noting that the execution time, when increasing the size by two, grows by more than 8 times for the same amount of iterations. This is likely to be attributed to the asymptotic complexity of each iteration, which is $O(n^3)$. As the iterations grow linearly to the mesh size, we end up with a complexity of $O(n^4)$ to reach the residual reduction, which justify the execution time of **37.0 seconds**.

An in depth explanations of the parameters visible in Listing 1 is:

- `-h`: Prints the help message.
- `-n`: Specifies the mesh size. The default value of 100 results in a problem size of $100 \times 100 \times 100$.
- `-M`: Sets the maximum number of iterations allowed. Default is 200. If this number of iterations is reached before the residual goes below the threshold specific by `-r`, the execution is stopped, even tough the solution is not actually reached.
- `-r`: Sets the relative residual tolerance for convergence. Default is 1×10^{-6}. If the residual goes below this value, we are satisfied with the solution and we do not execute further iterations.
- `-p`: Specifies the preconditioner to use, which helps accelerate the convergence of the iterative method. Options are:
  - `as`: Additive Schwarz preconditioner. This is a domain decomposition method where the domain is split into overlapping subdomains. Each subdomain solves the problem independently, which can improve convergence by reducing the condition number. The parameter `-o` (discussed below) controls the overlap for this method.

  - `ssor`: Symmetric Successive Over-Relaxation preconditioner. SSOR is an iterative method that combines Gauss-Seidel iteration with over-relaxation, allowing faster convergence. The parameter `-w` (discussed below) controls the relaxation factor.
  - `id`: Identity preconditioner, meaning no preconditioning is applied. This can be used as a baseline or for comparison with other preconditioners. Default is `id`.

- `-w`: Specifies the relaxation parameter for SSOR. This parameter, often denoted by ω, controls the level of over-relaxation applied to the SSOR method:
  - Values of $\omega > 1$ apply over-relaxation, which can improve convergence rates but may also lead to instability if set too high.
  - Values of $\omega < 1$ apply under-relaxation, which can stabilize the iteration at the cost of slower convergence.

  The default value of 1.9 is typically chosen to balance convergence rate and stability.

- `-o`: Specifies the overlap for the Schwarz method, which is the amount of shared region between subdomains in the additive Schwarz preconditioner:
  - A larger overlap allows more information exchange between subdomains, which can improve convergence by providing a smoother transition across subdomains.
  - However, a higher overlap increases computational cost, as more subdomain boundaries need to be solved iteratively.

  The default value is 10.

The parameters `-h`, `-n`, `-M` and `-r` affects the execution regardless of the value choose for `-p`. The values attributed to `-w` and `-o` affect the execution only when `-p` is set respectively to `ssor` and `as`.

```
#!/bin/bash

#SBATCH --job-name=submit-seq.sbatch
#SBATCH -D .
#SBATCH --output=execution_result.out

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean
make -C "/scratch/nas/1/sca1011/CSA/lab4"

../cgp3d.x

../cgp3d.x -p as

../cgp3d.x -p ssor
```

Listing 5: Slurm task to execute using different preconditioners.

```
199 steps, residual reduction 9.0582e-07 (<= tol 1e-06); time 2.17695 seconds
rnorm = 8.45154e-08
41 steps, residual reduction 7.87636e-07 (<= tol 1e-06); time 1.52927 seconds
rnorm = 3.71707e-06
30 steps, residual reduction 9.02651e-07 (<= tol 1e-06); time 0.977893 seconds
rnorm = 4.93313e-06
```

Listing 6: Slurm output when executing the task in Listing 5

When executing the SLURM task in Listing 5, we get the output displayed in Listing 6. From that we can notice that, as expected, the number of iterations needed to go below the residual threshold greatly decrease when using any of the two preconditioners. As A consequence, the running time is also greatly improved.

Another factor that influences the running time is the right-hand part of the Poisson equation that the solver will solve for. Two possible right hand-hand initializations are displayed in Listing 7.

```
void setup_rhs0(int n, double* b)
{
    int N = n*n*n;
    memset(b, 0, N*sizeof(double));
    b[0] = 1;
}

void setup_rhs1(int n, double* b)
{
    int N = n*n*n;
    memset(b, 0, N*sizeof(double));
    for (int i = 0; i < n; ++i) {
        double x = 1.0*(i+1)/(n+1);
        for (int j = 0; j < n; ++j) {
            double y = 1.0*(i+1)/(n+1);
            for (int k = 0; k < n; ++k) {
                double z = 1.0*(i+1)/(n+1);
                b[(k*n+j)*n+i] = x*(1-x) * y*(1-y) * z*(1-z);
            }
        }
    }
}
```

Listing 7: Right-hand side configuration.

The `setup_rhs0` function initializes the right-hand side vector to zero everywhere except for the first element, which is set to 1. On the other hand, `setup_rhs1` sets up a right-hand side function where the source is distributed throughout the grid based on the equation $x(1-x) \times y(1-y) \times z(1-z)x(1-x) \times y(1-y) \times z(1-z)$. This results in a continuous, smoothly varying source term across the domain.

To test how the right-hand side influences the running time, the part of the code where the right-hand side has been modified as displayed in Listing 8 and the SLURM task displayed in 10 has been executed. The running time resulting from the task are displayed in 10.

```
#ifdef USE_RHS0
    setup_rhs0(n, b);
#else
    setup_rhs1(n, b);
#endif
```

Listing 8: Select the right hand side at compile time.

```
make -C "/scratch/nas/1/sca1011/CSA/lab4" clean
make -C "/scratch/nas/1/sca1011/CSA/lab4" CFLAGS="-DUSE_RHS0"

../cgp3d.x

../cgp3d.x -p as

../cgp3d.x -p ssor

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean
make -C "/scratch/nas/1/sca1011/CSA/lab4"

../cgp3d.x

../cgp3d.x -p as

../cgp3d.x -p ssor
```

Listing 9: SLURM task to execute using different right-hand side.

```
97 steps, residual reduction 9.81934e-07 (<= tol 1e-06); time 2.95219 seconds
rnorm = 3.20149e-11
25 steps, residual reduction 7.14612e-07 (<= tol 1e-06); time 1.78559 seconds
rnorm = 4.32465e-08
20 steps, residual reduction 9.9342e-07 (<= tol 1e-06); time 1.21564 seconds
rnorm = 5.41227e-08

199 steps, residual reduction 9.0582e-07 (<= tol 1e-06); time 2.21613 seconds
rnorm = 8.45154e-08
41 steps, residual reduction 7.87636e-07 (<= tol 1e-06); time 1.51608 seconds
rnorm = 3.71707e-06
30 steps, residual reduction 9.02651e-07 (<= tol 1e-06); time 0.983194 seconds
rnorm = 4.93313e-06
```

Listing 10: SLURM task to execute using different right-hand side.

From the result visible in 10, we can state that although `setup_rhs0` result in much fewer iterations, each iteration takes up more time. Therefore, `setup_rhs1` needs more iterations to converge, but iterations are faster and ultimately result in a lower running time. In the instances taken into account, it seems that the behavior described is consistent among all the preconditioners configurations.

3 Timing unpreconditioned serial code on Boada

In this section, we execute the code with varying problem sizes. Specifically, as shown in 11, we run the code with three sizes: 100, 200, and 400. We do not specify a preconditioner, so the default 'id' is used, which initializes all values to 0.

```
#!/bin/bash
// ...
for n in 100 200 400; do
    ../cgp3d.x -n $n
done
```

Listing 11: SLURM task to execute using different right-hand side.

In Figure 1, we observe that the execution time increases significantly as the problem size grows. Specifically, each time we double the size of the mesh (since the mesh is $n \times n$, doubling n multiplies the number of elements by 4), the execution time increases by more than four times. For instance, when increasing the size from 100 to 200, the execution time grows by approximately one order of magnitude. Moreover, when the size increases from 200 to 400, the execution time becomes 8 times longer.

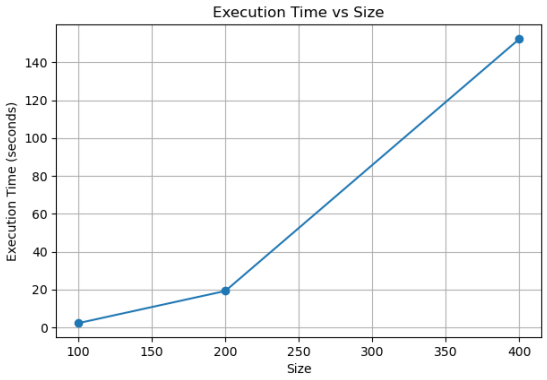

Figure 1: Execution time vs. problem size

In Table 2, we present the measured execution times of the 'Afun' function, showing that it consistently accounts for a significant portion of the total execution time, around 50% in all cases. We can observe this effect more clearly in Figure 3.

Size	Afun (s)	Dot (s)	Total Time (s)
100	0.62	0.60	2.20
200	4.97	5.06	18.51
400	39.74	40.05	151.36

Figure 2: Execution times of 'Afun' and dot product operations

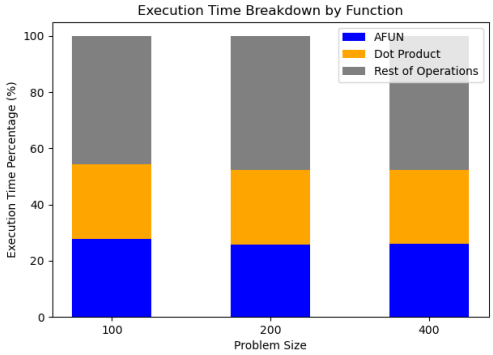

Figure 3: Execution time distribution between 'Afun' and dot product operations

4 Timing precondition serial code on Boada

In this section we measure time spent in the preconditioners and the number of steps taken by the preconditioned conjugate gradient (PCG) method for different problem sizes and preconditioners. The results are summarized in Table 1, where the preconditioners used were:

- **id**: The identity preconditioner (no preconditioning, only zero initialization).
- **ssor**: Symmetric Successive OverRelaxation (SSOR) preconditioner.
- **as**: Additive Schwarz preconditioner.

The table shows the preconditioner time in seconds and the number of steps required to reach convergence for each combination of problem size and preconditioner.

Size	Preconditioner	Preconditioner time (s)	Steps
100	id	0.283951	199
	ssor	0.675791	30
	as	1.09946	41
200	id	2.54061	200
	ssor	9.60144	51
	as	15.5157	71
400	id	20.2809	200
	ssor	138.135	90
	as	195.592	127

Table 1: Mfun Time and Steps for Different Sizes and Preconditions

The identity preconditioner, which performs no actual preconditioning, results in a high number of steps for all problem sizes, taking 199 steps for size 100 and 200 steps for sizes 200 and 400. Contrary to the initial assumption, the preconditioner time does not scale linearly with the problem size; instead, it increases rapidly, approximately with the cube of the problem size, i.e., the preconditioner time scales as $\mathcal{O}(n^3)$, where n is the problem size.

The SSOR preconditioner significantly reduces the number of steps required to converge. For size 100, it reduces the number of steps from 199 to 30. However, this comes with a corresponding increase in the preconditioner time, which scales even more rapidly with the problem size—approximately with the fourth power of the problem size, i.e., the preconditioner time scales as $\mathcal{O}(n^4)$.

The Additive Schwarz (AS) preconditioner strikes a balance between the two, taking fewer steps than the identity preconditioner but requiring more time than SSOR. For size 400, it takes 127 steps to converge, and the preconditioner time is 195.592 seconds, which is higher than that of SSOR at 138.135 seconds.

In summary, using preconditioners reduces the number of steps required for convergence, with SSOR being the most efficient in terms of steps. However, this efficiency comes at the cost of significantly increased preconditioner time, especially as the problem size grows, since the preconditioner time increases more rapidly than linearly with the problem size.

5 Parallelization of the unpreconditioned code

5.1 Basic parallelization

In this section, we implement the parallelization of the code using OMP pragmas. In particular, we parallelized the implementation of the `mul_poisson3d` function and `dot` function.

5.1.1 Execution parameters

To execute the model, we run the loop shown in 12, where we send jobs for each of the sizes that we want to study. Since we are using OMP and aiming for maximum parallelization, we allocate one node, with one task, and all the CPUs on the node with the maximum number of threads available, which in this case is 40.

```
for n in 100 200 400; do
    sbatch -W --nodes=1 --ntasks=1 --cpus-per-task=40 --export=OMP_DYNAMIC=false
     --export=OMP_NUM_THREADS=40 --wrap="../cgp3d.x -n $n"
    wait
done
```

Listing 12: Script for execution.

5.1.2 Code modifications

To implement the parallelization of the code, we use OpenMP pragmas to parallelize loops in the `dot` and `mul_poisson3d` functions.

In the `dot` function, we have a simple loop that computes the dot product of two vectors. To parallelize this loop, we use the `#pragma omp parallel for reduction(+:result)` directive as shown in Listing 13.

```
double result = 0;
#pragma omp parallel for reduction(+:result)
for (int i = 0; i < n; ++i)
    result += x[i]*y[i];
return result;
```

Listing 13: Parallelization of dot product function.

The `#pragma omp parallel for` directive tells the compiler to parallelize the following `for` loop using OpenMP. The `reduction(+:result)` clause specifies that the variable `result` should be reduced across threads using the `+` operator. This means that each thread will compute a partial sum of `result`, and at the end of the loop, these partial sums will be combined into a single value. This avoids race conditions and ensures thread safety.

In the `mul_poisson3d` function, which computes the matrix-vector multiplication for the Poisson 3D problem, we have nested loops over the indices `k`, `j`, and `i`. To parallelize these loops, we use the `#pragma omp parallel for collapse(3)` directive as shown in Listing 14.

```
#pragma omp parallel for collapse(3)
for (int k = 0; k < n; ++k) {
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            double xx = X(i,j,k);
            double xn = (i > 0)   ? X(i-1,j,k) : 0;
            double xs = (i < n-1) ? X(i+1,j,k) : 0;
            double xe = (j > 0)   ? X(i,j-1,k) : 0;
            double xw = (j < n-1) ? X(i,j+1,k) : 0;
            double xu = (k > 0)   ? X(i,j,k-1) : 0;
            double xd = (k < n-1) ? X(i,j,k+1) : 0;
            AX(i,j,k) = (6*xx - xn - xs - xe - xw - xu - xd)*inv_h2;
        }
    }
}
```

Listing 14: Parallelization of `mul_poisson3d` function.

The `#pragma omp parallel for collapse(3)` directive instructs the compiler to parallelize the nested loops over `k`, `j`, and `i`. The `collapse(3)` clause tells OpenMP to collapse the three

nested loops into a single loop over the combined iteration space. This increases the amount of parallelism available by effectively flattening the loops, allowing the iterations to be distributed among threads more efficiently.

In both cases, the use of OpenMP pragmas allows us to parallelize the computationally intensive parts of the code, leveraging multi-core processors to improve performance.

Iterating on this optimization, we applied the same optimization to the blocked version of `mul_poisson3d` described in Section 8.4.

5.1.3 Results

Table 2 presents the execution times for both the sequential and parallel versions of the code across different problem sizes leaving the maximum iteration count invaried. We observe that the parallel implementation consistently outperforms the sequential one and the parallel blocked version has slighly better performances. The Table 3 reports the same result but focused only on `mul_poisson3d` and setting stopping when the precision is reached and not due to the iteration count.

Size (n)	Time Seq. (s)	Time Par. (s)	Time Blocked Par. (s)	Speedup	Speedup Blocked
100	2.20	1.52	1.50	1.45	1.46
200	18.51	12.80	12.70	1.45	1.47
400	151.36	102.24	98.31	1.48	1.53

Table 2: Execution times and speedup for sequential and parallel code

Size (n)	Time Sequential (s)	Time Parallel (s)	Speedup
100	0.690495	0.079	8.74
200	10.975	2.151	5.10
300	55.4436	12.795	4.33

Table 3: Execution times and speedup for sequential and parallel implementation of `mul_poisson3d`

The speedup is calculated as the ratio of the sequential execution time to the parallel execution time:

$$\text{Speedup} = \frac{\text{Time Sequential}}{\text{Time Parallel}}$$

As shown in Table 2, the speedup achieved ranges from approximately 1.45 to 1.48. This indicates that the parallel code runs about 1.5 times faster than the sequential code.

While the parallelization yields performance improvements, the speedup is modest considering that we utilized 40 threads in the parallel execution. Several factors contribute to this limited speedup:

- **Amdahl's Law:** The overall speedup is constrained by the sequential portions of the code. As we show in the previous experiments the parallelized regions supposes approximately 50% of the execution time, so even if the execution time of these sections is reduces to 0 we can not get a speed-up bigger than 2. As it is possible to see from Table 3 the speedups achieved when taking into account only `mul_poisson3d` are much higher. But, as `mul_poisson3d` is only a fraction of the total execution time, this does not directly translate in a big speedup on the overall execution time.

- **Parallel Overhead:** Initiating parallel regions and managing threads introduce overhead. This overhead can offset the benefits of parallel execution, especially for smaller problem sizes where the computation per thread is minimal.
- **Load Imbalance:** If the workload is not evenly distributed among threads, some threads may complete their tasks earlier and remain idle while others are still processing.
- **Synchronization Overhead:** Although we used efficient OpenMP constructs like `reduction` and `collapse`, there is still some overhead associated with synchronizing threads and combining results.

Analysis of Performance Improvement The parallelization of the `dot` and `mul_poisson3d` functions addresses the most computationally intensive parts of the code. The `dot` function benefits from the use of the `reduction` clause, which efficiently sums partial results from each thread. The `mul_poisson3d` function utilizes the `collapse(3)` clause to maximize parallelism across three nested loops.

Despite these optimizations, the speedup is not proportional to the number of threads. This suggests that the program is not fully utilizing the available computational resources.

Conclusion The implementation of OpenMP parallelization in the `dot` and `mul_poisson3d` functions has led to measurable performance gains. However, the less-than-ideal speedup indicates that the program's scalability is limited by factors such as overhead, memory bandwidth, and remaining sequential code. Further optimizations are necessary to achieve better parallel efficiency and fully leverage the capabilities of multicore processors.

6 Parallelization of Schwarz

6.1 Basic parallelization

6.1.1 Code Modifications

To parallelize the Schwarz preconditioner for the 3D Poisson problem, we modified the code to decompose the domain into multiple overlapping subdomains and utilized OpenMP to parallelize the computations across these subdomains. Specifically, we focused on parallelizing the `pc_schwarz_poisson3d` function and its helper functions.

Domain Decomposition and Parallelization We generalized the domain decomposition to split the 3D domain along the `z`-axis into `P` overlapping subdomains, where `P` is the number of threads or processors. Each subdomain includes an overlap region to ensure that the updates in one subdomain influence neighboring subdomains, improving convergence.

To parallelize the computations over these subdomains, we employed OpenMP's `#pragma omp parallel for` directive. Listing 15 shows the modified `pc_schwarz_poisson3d` function with the parallelization.

```
void pc_schwarz_poisson3d(int N, void* data,
                          double* restrict Ax,
                          double* restrict x)
{
    int P = 20;
    pc_schwarz_p3d_t* ssor_data = (pc_schwarz_p3d_t*) data;
    int n = ssor_data->n;
    int overlap = ssor_data->overlap / 2;
    double w = ssor_data->omega;
    int subdomain_size = (n + P - 1) / P; // Ceiling division

```

```
    // Initialize Ax to zero
    memset(Ax, 0, N * sizeof(double));

    // Allocate arrays
    double** scratch = (double**) malloc(P * sizeof(double*));
    int* z_start = (int*) malloc(P * sizeof(int));
    int* z_end = (int*) malloc(P * sizeof(int));

    // Precompute subdomain boundaries
    for (int p = 0; p < P; ++p) {
        z_start[p] = p * subdomain_size - overlap;
        z_end[p] = (p + 1) * subdomain_size + overlap;

        if (z_start[p] < 0) z_start[p] = 0;
        if (z_end[p] > n) z_end[p] = n;

        int local_nz = z_end[p] - z_start[p];
        int local_nx = n;
        int local_ny = n;
        int local_size = local_nx * local_ny * local_nz;
        scratch[p] = (double*) malloc(local_size * sizeof(double));
    }

    // Parallel computation over subdomains
    #pragma omp parallel for
    for (int p = 0; p < P; ++p) {
        double* local_scratch = scratch[p];

        int local_nx = n;
        int local_ny = n;
        int local_nz = z_end[p] - z_start[p];

        int i_start_global = 0;
        int j_start_global = 0;
        int k_start_global = z_start[p];

        // Extract local data
        schwarz_get(n, i_start_global, j_start_global, k_start_global,
                    local_nx, local_ny, local_nz,
                    local_scratch, x);

        // Perform SSOR sweeps
        ssor_forward_sweep(local_nx, local_ny, local_nz,
                           local_scratch, w);
        ssor_diag_sweep(local_nx, local_ny, local_nz,
                        local_scratch, w);
        ssor_backward_sweep(local_nx, local_ny, local_nz,
                            local_scratch, w);
    }

    // Combine contributions
    #pragma omp parallel for
    for (int p = 0; p < P; ++p) {
        int local_nx = n;
        int local_ny = n;
        int local_nz = z_end[p] - z_start[p];

        int i_start_global = 0;
        int j_start_global = 0;
        int k_start_global = z_start[p];

        schwarz_add_atomic(n, local_nx, local_ny, local_nz,
                           i_start_global, j_start_global, k_start_global,
                           scratch[p], Ax);
    }

    // Free allocated memory
    for (int p = 0; p < P; ++p) {
        free(scratch[p]);
    }
    free(scratch);
    free(z_start);
```

```
84     free(z_end);
85 }
```

Listing 15: Parallelization of pc_schwarz_poisson3d function.

In this code:

- We define the number of subdomains `P` and calculate the subdomain size with an overlap.
- We allocate arrays to store subdomain boundaries and scratch space for each thread.
- We use `#pragma omp parallel for` to parallelize the loop over subdomains.
- Each thread extracts its local data using the `schwarz_get` function, performs the SSOR sweeps, and then contributes its results back to the global array `Ax` using the `schwarz_add_atomic` function.

Synchronization with Atomic Operations Because subdomains overlap, multiple threads may attempt to update the same elements in the global array `Ax`. To prevent race conditions and ensure thread safety, we use atomic operations when updating `Ax`. Listing 16 shows the `schwarz_add_atomic` function.

```
void schwarz_add_atomic(int n, int local_nx, int local_ny, int local_nz,
                        int i_start_global, int j_start_global, int k_start_global,
                        double* local_scratch,
                        double* Ax)
{
    int idx = 0; // Local index
    for (int k = 0; k < local_nz; ++k) {
        int global_k = k + k_start_global;
        for (int j = 0; j < local_ny; ++j) {
            int global_j = j + j_start_global;
            for (int i = 0; i < local_nx; ++i) {
                int global_i = i + i_start_global;
                int global_idx = global_i + n * (global_j + n * global_k);
                #pragma omp atomic
                Ax[global_idx] += local_scratch[idx];
                idx++;
            }
        }
    }
}
```

Listing 16: Atomic addition in schwarz_add_atomic function.

The `#pragma omp atomic` directive ensures that the addition of `local_scratch[idx]` to `Ax[global_idx]` is performed atomically, preventing race conditions when multiple threads update the same element.

Adjustments to SSOR Functions We modified the SSOR sweep functions to operate on local subdomains with proper indexing. The functions now use local indices and dimensions, and handle boundary conditions within the local subdomains. Listing **??** shows the modified `ssor_forward_sweep` function.

```
void ssor_forward_sweep(int local_nx, int local_ny, int local_nz,
                        double* restrict Ax, double w)
{
    #define AX(i,j,k) (Ax[(i) + local_nx * ((j) + local_ny * (k))])
    for (int k = 0; k < local_nz; ++k) {
        for (int j = 0; j < local_ny; ++j) {
            for (int i = 0; i < local_nx; ++i) {
                double xx = AX(i,j,k);
                double xn = (i > 0)              ? AX(i-1,j,k) : 0;
                double xe = (j > 0)              ? AX(i,j-1,k) : 0;
                double xu = (k > 0)              ? AX(i,j,k-1) : 0;
                AX(i,j,k) = (xx + xn + xe + xu) * w / 6.0;
            }
        }
    }
    #undef AX
}
```

Listing 17: Modified ssor_forward_sweep function for local subdomains.

In this function:

- We use local indices `i`, `j`, `k` that range over the subdomain's dimensions.
- The macro `AX(i,j,k)` maps local indices to the 1D array `Ax`.
- We handle boundary conditions by checking if the neighboring indices are within the local subdomain.

Handling of Overlapping Regions The overlapping regions between subdomains require careful handling to ensure correctness and convergence. By updating variables in the overlap regions in multiple subdomains, we allow information to propagate across the domain more effectively.

Memory Allocation and Indexing Adjustments We adjusted the memory allocation for the scratch space to match the sizes of the local subdomains. Additionally, we ensured that all indexing correctly maps between local and global indices to prevent segmentation faults and memory access violations.

Example of Data Extraction Function Listing 18 shows the modified `schwarz_get` function used to extract local data from the global array.

```
void schwarz_get(int n, int i_start_global, int j_start_global, int k_start_global,
                 int local_nx, int local_ny, int local_nz,
                 double* restrict x_local,
                 double* restrict x)
{
    int idx = 0;
    for (int k = 0; k < local_nz; ++k) {
        int global_k = k + k_start_global;
        for (int j = 0; j < local_ny; ++j) {
            int global_j = j + j_start_global;
            for (int i = 0; i < local_nx; ++i) {
                int global_i = i + i_start_global;
                int global_idx = global_i + n * (global_j + n * global_k);
                x_local[idx] = x[global_idx];
                idx++;
            }
        }
    }
}
```

Listing 18: Modified `schwarz_get` function for local subdomains.

This function maps local indices to global indices to extract the relevant portion of the global data into the local scratch space.

This way we decompose the domain into multiple overlapping subdomains and parallelizing the computations using OpenMP.

6.1.2 Results

We evaluated the performance of the sequential and parallel implementations of the Schwarz preconditioner by measuring the total execution time and the time spent specifically in the preconditioner function (MFUN) for different problem sizes. Tables 4 and 5 present the execution times and the corresponding speedups achieved by the parallel implementation.

Size (n)	Total Time Sequential (s)	Total Time Parallel (s)	Speedup
100	1.40721	1.03994	1.35
200	19.8635	10.5452	1.88
400	282.258	122.420	2.31

Table 4: Total execution times and speedup for sequential and parallel Schwarz preconditioner

Size (n)	MFUN Time Sequential (s)	MFUN Time Parallel (s)	Speedup
100	0.978714	0.555386	1.76
200	13.8039	4.81925	2.86
400	196.094	45.3204	4.33

Table 5: MFUN execution times and speedup for sequential and parallel Schwarz preconditioner

The speedup is calculated as the ratio of the sequential execution time to the parallel execution time:

$$\text{Speedup} = \frac{\text{Sequential Time}}{\text{Parallel Time}}$$

Analysis of Performance Improvement As observed in Table 4, the total execution time decreases with the parallel implementation across all problem sizes. The speedup for the total execution time increases with the problem size, achieving up to a 2.31 speedup for $n = 400$. This indicates that the parallelization becomes more effective for larger problem sizes.

Similarly, Table 5 shows that the time spent in the MFUN function is significantly reduced with the parallel implementation. The speedup for MFUN is higher than that of the total execution time, reaching up to a 4.33 speedup for $n = 400$, given that the precondition supposes around 50% of the total execution time.

Observations and Factors Affecting Speedup Several factors contribute to the observed performance improvements and the scaling behavior:

- **Increased Parallel Workload with Problem Size:** Larger problem sizes result in more computational work per thread, which improves the efficiency of parallel execution by amortizing the overhead associated with thread management and synchronization.
- **Amdahl's Law:** The overall speedup is limited by the sequential portions of the code. As the problem size increases, the proportion of time spent in parallelizable sections grows, allowing for greater speedup.
- **Parallel Overhead:** Initiating parallel regions, managing threads, and synchronizing shared data introduce overhead. This overhead has a more pronounced effect on smaller problem sizes, where the computation per thread is minimal.
- **Synchronization Overhead:** The use of atomic operations in the `schwarz_add_atomic` function introduces synchronization overhead. However, this is necessary to maintain correctness due to overlapping subdomains.

Conclusion The parallelization of the Schwarz preconditioner using OpenMP has led to significant performance gains, especially for larger problem sizes. The speedup achieved in the MFUN function demonstrates the effectiveness of the parallel implementation in reducing the computational cost of the preconditioner. While the total execution time also benefits from parallelization, the speedup is moderated by the sequential parts of the code and overhead factors.

These results indicate that further optimizations, such as parallelizing additional parts of the code or reducing synchronization overhead, could lead to even greater performance improvements.

7 Scaling Studies for Different Problem Sizes and Preconditioners

In this section, we present scaling studies of the execution time required by our code for different problem sizes n and preconditioners p. We compare the sequential and parallel execution times of the different preconditioners and analyze the performance benefits of parallelization.

7.1 Unpreconditioned Solver

For the unpreconditioned solver, we parallelize the constructor, the `mul_poisson3d` function, and the `dot` function. The unpreconditioned solver's execution time is significantly impacted by these functions, accounting for approximately 50% of the total execution time. The execution times for different problem sizes n are presented in Table 6.

Table 6: Execution times (in seconds) for the unpreconditioned solver.

n	Sequential		Parallel	
	Time	Speedup	Time	Speedup
100	2.20	1×	1.52	1.45×
200	18.51	1×	12.80	1.45×
400	151.36	1×	102.24	1.48×

7.2 SSOR Preconditioner

For the SSOR preconditioner, we use the same parallelization strategies as the unpreconditioned solver by parallelizing the `mul_poisson3d` and `dot` functions. However, the SSOR preconditioner itself is computationally intensive, accounting for approximately 66% of the execution time in the sequential code. Since we do not parallelize the preconditioner, we can only parallelize about 40% of the code, which limits the maximum achievable speedup. The execution times are shown in Table 7.

Table 7: Execution times (in seconds) for the SSOR preconditioner.

n	Sequential		Parallel	
	Time	Speedup	Time	Speedup
100	0.901	1×	0.791	1.14×
200	12.725	1×	11.053	1.15×
400	181.14	1×	157.27	1.15×

7.3 Schwarz Preconditioner

In the case of the Schwarz preconditioner, we parallelize not only the `mul_poisson3d` and `dot` functions but also the preconditioner itself. This comprehensive parallelization allows us to achieve a greater speedup compared to the other preconditioners. The execution times are detailed in Table 8.

Table 8: Execution times (in seconds) for the Schwarz preconditioner.

n	Sequential		Parallel	
	Time	Speedup	Time	Speedup
100	1.407	1×	1.040	1.35×
200	19.864	1×	10.545	1.88×
400	282.258	1×	122.420	2.31×

7.4 Comparison Between Preconditioners

Table 9 and Table 10 compares the execution times of the different preconditioners with and without parallelization. Notably, for smaller problem sizes, the more complex preconditioners like SSOR and Schwarz can lead to faster overall execution times despite their computational complexity. This is because they reduce the number of iterations required for convergence, offsetting the additional per-iteration cost.

Table 9: Comparison of sequential execution times (in seconds) between preconditioners.

n	Sequential			
	Unprecond.	SSOR	Schwarz	Best
100	2.20	**0.901**	1.407	SSOR
200	18.51	**12.725**	19.864	SSOR
400	**151.36**	181.14	282.258	Unprecond.

Table 10: Comparison of parallel execution times (in seconds) between preconditioners.

n	Parallel			
	Unprecond.	SSOR	Schwarz	Best
100	1.52	**0.791**	1.040	SSOR
200	12.80	11.053	**10.545**	Schwarz
400	**102.24**	157.27	122.420	Unprecond.

7.5 Discussion

The results demonstrate that parallelization provides performance benefits across all preconditioners, with the most significant speedups observed in the Schwarz preconditioner due to the parallelization of the preconditioner itself. However, the effectiveness of parallelization is also influenced by the proportion of the code that can be parallelized. For instance, in the SSOR preconditioner, only 40% of the code is parallelizable, limiting the speedup.

Moreover, the choice of preconditioner has a considerable impact on the execution time. For smaller problem sizes, the SSOR preconditioner outperforms others due to its ability to reduce the number of iterations required for convergence, despite its higher per-iteration computational cost. As the problem size increases, the overhead of the more complex preconditioners becomes more pronounced, and the unpreconditioned solver may offer faster execution times in parallel implementations.

These way it is importance of considering both the computational complexity of preconditioners and the extent of parallelization when optimizing solver performance for different problem sizes.

8 Improving Laplacian and SSOR sequential implementation

The `mul_poisson3d` function applies the 3D Laplacian to a $n \times n \times n$ mesh of, $[0,1]^3$ ($N = n^3, h = \frac{1}{n-1}$) assuming Dirichlet boundary conditions. The code can be seen in Listing 20.

```
void mul_poisson3d(int N, void* data, double* restrict Ax, double* restrict x)
{
    #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])

    int n = *(int*) data;
    int inv_h2 = (n-1) * (n-1);
    for (int k = 0; k < n; ++k) {
        for (int j = 0; j < n; ++j) {
            for (int i = 0; i < n; ++i) {
                double xx = X(i,j,k);
                double xn = (i > 0)   ? X(i-1,j,k) : 0;
                double xs = (i < n-1) ? X(i+1,j,k) : 0;
                double xe = (j > 0)   ? X(i,j-1,k) : 0;
                double xw = (j < n-1) ? X(i,j+1,k) : 0;
                double xu = (k > 0)   ? X(i,j,k-1) : 0;
                double xd = (k < n-1) ? X(i,j,k+1) : 0;
                AX(i,j,k) = (6*xx - xn - xs - xe - xw - xu - xd)*inv_h2;
            }
        }
    }

    #undef AX
    #undef X
}
#endif
```

Listing 19: Initial implementatino of mul_poisson3d

This is a short code, but it hides many opportunities to apply optimizations. The ones that will be explored in this report are the following:

8.1 Partial Predication

Despite branch predictors, branches are very expensive. In modern processors with: out of order execution, speculation and very deep pipelines, a branch miss can represent a substantial amount of wasted computation. As the loop in question has several branches for each iteration, these might be the cause of a big lost of performances. One way to overcome these branches is the use of partial predication. Partial predication is a technique that aims to replace conditions with arithmetic operations or selects values based on the conditions with the ultimate goal of avoiding branches. This brings the benefit of completely eliminating branches, therefore making the control flow much more predictable and allows keeping the processor in use. The drawback is that this comes at the cost of an increased amount of arithmetic operations that need to be performed. The code showed in Listing 20 is the Laplacian with the improvements. Figure 4 shows the time spent in the `mul_poisson3d` when using the normal implementation or the partially predicated one. As it's possible to appreciate in Figure 4, in this specific case, predication does not yield better results than the baseline version.

```
void mul_poisson3d(int N, void* data, double* restrict Ax, double* restrict x)
{
    tic(1);
    #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
    int n = *(int*) data;
    int inv_h2 = (n-1) * (n-1);
    for (int k = 0; k < n; ++k) {
        int k_gt_0 = k > 0;
        int k_lt_n1 = k < n - 1;
        int ku = k - k_gt_0;
        int kd = k + k_lt_n1;
        for (int j = 0; j < n; ++j) {
            int j_gt_0 = j > 0;
            int j_lt_n1 = j < n - 1;
            int je = j - j_gt_0;
            int jw = j + j_lt_n1;
            for (int i = 0; i < n; ++i) {
                int i_gt_0 = i > 0;
                int i_lt_n1 = i < n - 1;
                int in = i - i_gt_0;
                int is = i + i_lt_n1;
                double xx = X(i, j, k);
                double xn = X(in, j, k) * i_gt_0;
                double xs = X(is, j, k) * i_lt_n1;
                double xe = X(i, je, k) * j_gt_0;
                double xw = X(i, jw, k) * j_lt_n1;
                double xu = X(i, j, ku) * k_gt_0;
                double xd = X(i, j, kd) * k_lt_n1;
                AX(i, j, k) = (6 * xx - xn - xs - xe - xw - xu - xd) * inv_h2;
            }
        }
    }
    #undef AX
    #undef X
    time_in_poisson += toc(1);
}
```

Listing 20: Partial predicated implementation of mul_poisson3d

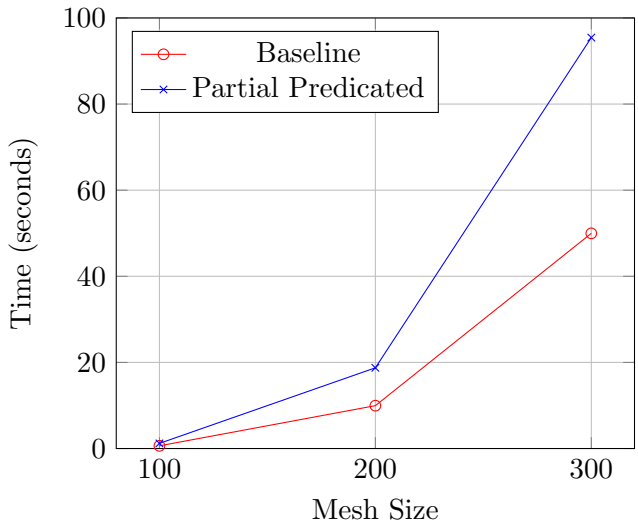

Figure 4: Comparison of Baseline and Partial Predicated mul_poisson3d execution times

8.2 Loop Interchange

Memory access can be several order of magnitude more expensive than performing an arithmetic instruction. Therefore, we want to minimize the number of cache misses that we incur in. One way to minimize it is to access the data sequentially in order to leverage the data locality and therefore minimize the cache misses and therefore the access to the further memory level in the memory hierarchy. One way to achieve this, in our case, is to order the three loops in the best way possible. To find out what is the best way possible, as we have only 6 combinations, we can implement all of them and see which one grant the best running times. This implementation is shown in Listing 21. This code is then compiled and executed once for each of the preprocessor directive, the resulting times are shown in Figure 5. Also in this case, the baseline (KJI in the plot) version already presents the best access to memory and therefore is the one with the lowest execution time, beating the second best by a very wide margin while the worst one takes almost 4x the time.

```
void mul_poisson3d(int N, void* data, double* restrict Ax, double* restrict x)
{
    tic(1);
    #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])

    int n = *(int*) data;
    int inv_h2 = (n - 1) * (n - 1);
#if defined(USE_JKI)
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
#elif defined(USE_KIJ)
    for (int k = 0; k < n; ++k) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
#elif defined(USE_IKJ)
    for (int i = 0; i < n; ++i) {
        for (int k = 0; k < n; ++k) {
            for (int j = 0; j < n; ++j) {
#elif defined(USE_IJK)
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            for (int k = 0; k < n; ++k) {
#elif defined(USE_JIK)
    for (int j = 0; j < n; ++j) {
        for (int i = 0; i < n; ++i) {
            for (int k = 0; k < n; ++k) {
#else
    for (int k = 0; k < n; ++k) {
        for (int j = 0; j < n; ++j) {
            for (int i = 0; i < n; ++i) {
#endif
                double xx = X(i,j,k);
                double xn = (i > 0)   ? X(i-1,j,k) : 0;
                double xs = (i < n-1) ? X(i+1,j,k) : 0;
                double xe = (j > 0)   ? X(i,j-1,k) : 0;
                double xw = (j < n-1) ? X(i,j+1,k) : 0;
                double xu = (k > 0)   ? X(i,j,k-1) : 0;
                double xd = (k < n-1) ? X(i,j,k+1) : 0;
                AX(i,j,k) = (6*xx - xn - xs - xe - xw - xu - xd)*inv_h2;
            }
        }
    }

    #undef AX
    #undef X
    time_in_poisson += toc(1);
}
```

Listing 21: Loop interchange implementation of mul_poisson3d

8.3 Split the loop in more loops

In addition to the problem described on Item 1 of this list, a more complex cycle iteration might prevent the compiler to automatically apply certain optimizations. The branches in the baseline implementation are necessary to handle the edge cases where we calculate points at the edges of the mesh. If we isolate the edge cases and compute them in different cycles, we can achieve a much simpler cycle for the base cases (which also represent the vast majority of the iterations) which also have no branches. The drawback in this case is that the code size increases a lot and therefore, in addition to increased pressure on the instruction cache, we also get code that is much less readable and less maintainable. This implementation is showcased in Listing 22, for readability reason only the base case is represented, the full code can be found in the file `cgp3d.c` that will be handed in

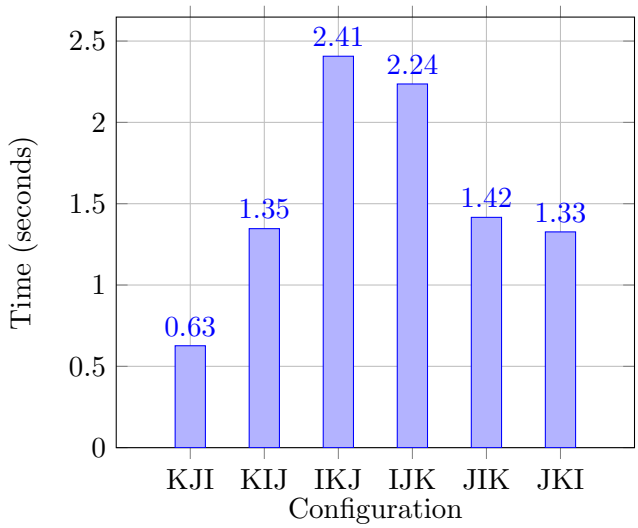

Figure 5: Different loop ordering execution time.

with this report. The execution times of this version are compared with the baseline in Figure 6. In this case, the new version performs better than the baseline version.

```
void mul_poisson3d(int N, void* data, double* restrict Ax, double* restrict x)
{
    tic(1);
    #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])

    int n = *(int*) data;
    int inv_h2 = (n - 1) * (n - 1);

    // Handle interior points where all neighbors are within bounds
    for (int k = 1; k < n - 1; ++k) {
        for (int j = 1; j < n - 1; ++j) {
            for (int i = 1; i < n - 1; ++i) {
                AX(i, j, k) = (6 * X(i, j, k)
                                - X(i - 1, j, k) - X(i + 1, j, k)
                                - X(i, j - 1, k) - X(i, j + 1, k)
                                - X(i, j, k - 1) - X(i, j, k + 1)) * inv_h2;
            }
        }
    }

    // Handle boundary faces k = 0 and k = n - 1
    // Handle boundary faces j = 0 and j = n - 1
    // Handle boundary faces i = 0 and i = n - 1
    // Handle edges where two indices are at the boundary
    // Handle corner points where all three indices are at the boundary

    #undef AX
    #undef X
    time_in_poisson += toc(1);
}
```

Listing 22: Splitted loop implementation of mul_poisson3d

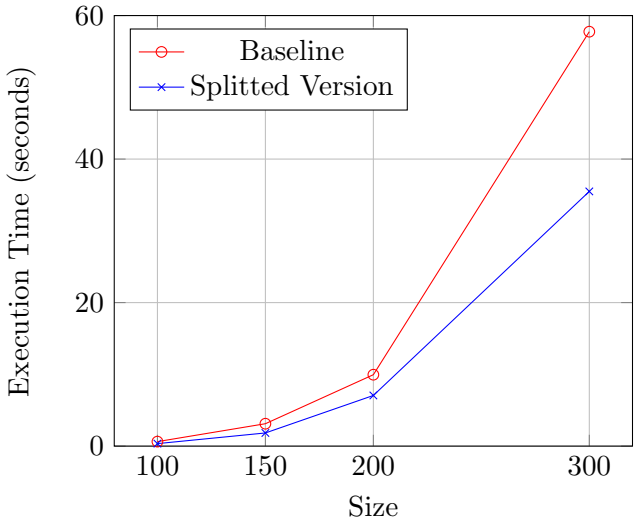

Figure 6: Comparison of Baseline and Splitted Version

8.4 Blocking

As we have seen in the previous laboratory reports, another method to improve the performances is blocking the code in order to improve the cache utilization. In Listing 23 the blocking code is displayed. The block size is defined using a preprocessing directive, which allows to programmatically test several block size values, in Figure 7 it's possible to see the results. The red bar is the non-blocked version. The blocked parameters that granted the best result are, `Bk=4,Bj=4,Bi=32` which brings a 15% improvement on the execution time.

```c
void mul_poisson3d(int N, void* data, double* restrict Ax, double* restrict x)
{
    tic(1);
    #define X(i,j,k) (x[((k)*n+(j))*n+(i)])
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])

    int n = *(int*) data;
    int inv_h2 = (n - 1) * (n - 1);

    // Set default block sizes if not defined
    #ifndef Bk
    #define Bk 8
    #endif

    #ifndef Bj
    #define Bj 8
    #endif

    #ifndef Bi
    #define Bi 32
    #endif

    for (int kk = 0; kk < n; kk += Bk) {
        for (int jj = 0; jj < n; jj += Bj) {
            for (int ii = 0; ii < n; ii += Bi) {
                int k_end = (kk + Bk < n) ? kk + Bk : n;
                int j_end = (jj + Bj < n) ? jj + Bj : n;
                int i_end = (ii + Bi < n) ? ii + Bi : n;

                for (int k = kk; k < k_end; ++k) {
                    for (int j = jj; j < j_end; ++j) {
                        for (int i = ii; i < i_end; ++i) {
                            double xx = X(i, j, k);
                            double xn = (i > 0)     ? X(i - 1, j, k) : 0;
                            double xs = (i < n - 1) ? X(i + 1, j, k) : 0;
                            double xe = (j > 0)     ? X(i, j - 1, k) : 0;
                            double xw = (j < n - 1) ? X(i, j + 1, k) : 0;
                            double xu = (k > 0)     ? X(i, j, k - 1) : 0;
                            double xd = (k < n - 1) ? X(i, j, k + 1) : 0;
                            AX(i, j, k) = (6 * xx - xn - xs - xe - xw - xu - xd) * inv_h2;
                        }
                    }
                }
            }
        }
    }

    #undef AX
    #undef X

    // Optionally undefine block sizes to avoid conflicts elsewhere
    #undef Bk
    #undef Bj
    #undef Bi

    time_in_poisson += toc(1);
}
```

Listing 23: Blocked implementation of mul_poisson3d

8.5 Applying blocking to the *SSOR preconditinoer*

The blocking optimization can also be applied to the *SSOR preconditioner*. In particular, `ssor_forward_sweep` and `ssor_backward_sweep` which respectively apply $(D/\omega + L)^{-1}$, $(D/omega + L)^{-T}$, and $(2 - \omega)D/\omega$ show the highest performance improvement when implemented using the blocking method.

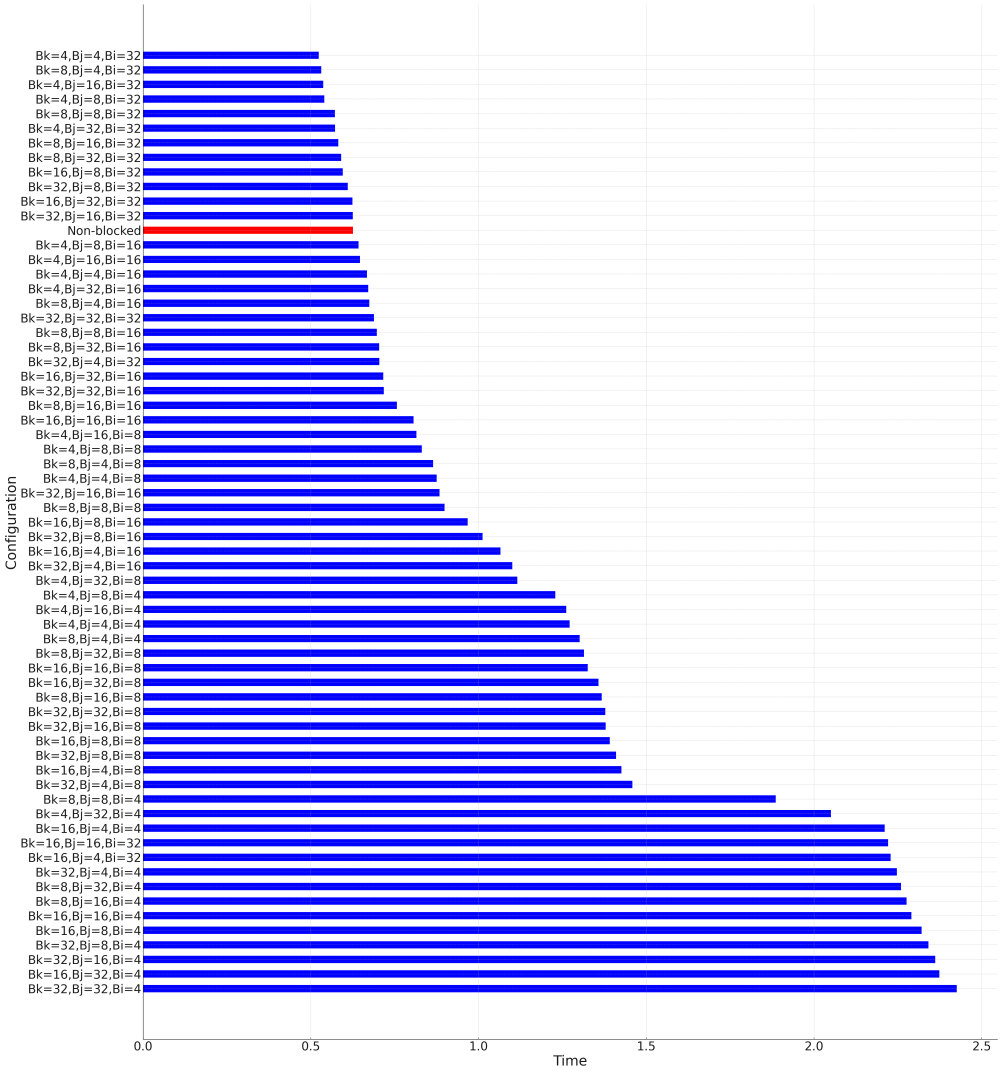

Figure 7: Time spent in `mul_poisson3d` whit different blocking parameters.

The same technique has been also applied to `ssor_diag_sweep` which applies $(2-\omega)D/\omega$ but for this function, but as no significant improvement was measured this function has been reverted to the baseline version. The blocked code for `ssor_forward_sweep` and `ssor_backward_sweep` is displayed in Listing 24. The time spent in SSOR is shown in Figure 8.

```
#define BLOCK_SIZE 8
#define MIN(a, b) (((a) < (b)) ? (a) : (b))
#define MAX(a, b) (((a) > (b)) ? (a) : (b))

void ssor_forward_sweep(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                                double* restrict Ax, double w)
{
    tic(2);
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
    for (int kk = k1; kk < k2; kk += BLOCK_SIZE) {
        int k_max = MIN(kk + BLOCK_SIZE, k2);
        for (int jj = j1; jj < j2; jj += BLOCK_SIZE) {
            int j_max = MIN(jj + BLOCK_SIZE, j2);
            for (int ii = i1; ii < i2; ii += BLOCK_SIZE) {
                int i_max = MIN(ii + BLOCK_SIZE, i2);
                for (int k = kk; k < k_max; ++k) {
                    for (int j = jj; j < j_max; ++j) {
                        for (int i = ii; i < i_max; ++i) {
                            double xx = AX(i,j,k);
                            double xn = (i > 0)   ? AX(i-1,j,k) : 0.0;
                            double xe = (j > 0)   ? AX(i,j-1,k) : 0.0;
                            double xu = (k > 0)   ? AX(i,j,k-1) : 0.0;
                            AX(i,j,k) = (xx + xn + xe + xu) / 6.0 * w;
                        }
                    }
                }
            }
        }
    }
    #undef AX
    time_in_SSOR += toc(2);
}

void ssor_backward_sweep(int n, int i1, int i2, int j1, int j2, int k1, int k2,
                                 double* restrict Ax, double w)
{
    tic(2);
    #define AX(i,j,k) (Ax[((k)*n+(j))*n+(i)])
    for (int kk = k2 - 1; kk >= k1; kk -= BLOCK_SIZE) {
        int k_min = MAX(kk - BLOCK_SIZE + 1, k1);
        for (int jj = j2 - 1; jj >= j1; jj -= BLOCK_SIZE) {
            int j_min = MAX(jj - BLOCK_SIZE + 1, j1);
            for (int ii = i2 - 1; ii >= i1; ii -= BLOCK_SIZE) {
                int i_min = MAX(ii - BLOCK_SIZE + 1, i1);
                for (int k = kk; k >= k_min; --k) {
                    for (int j = jj; j >= j_min; --j) {
                        for (int i = ii; i >= i_min; --i) {
                            double xx = AX(i,j,k);
                            double xs = (i < n-1) ? AX(i+1,j,k) : 0.0;
                            double xw = (j < n-1) ? AX(i,j+1,k) : 0.0;
                            double xd = (k < n-1) ? AX(i,j,k+1) : 0.0;
                            AX(i,j,k) = (xx + xs + xw + xd) / 6.0 * w;
                        }
                    }
                }
            }
        }
    }
    #undef AX
    time_in_SSOR += toc(2);
}
}
```

Listing 24: Blocked implementation of ssor_forward_sweep and ssor_backward_sweep

When both *SSOR* and *Laplacian* are executed using blocked code, the final execution time is shown in Figure 9.

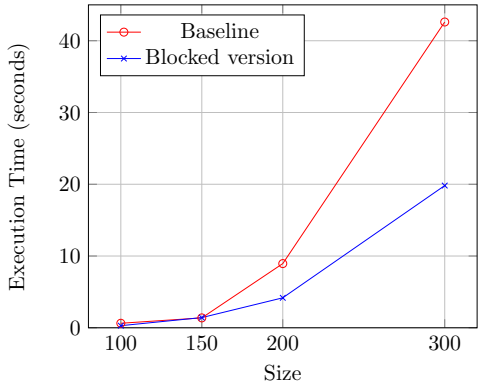

Figure 8: Comparison of baseline and blocked version of *SSOR* preconditioner.

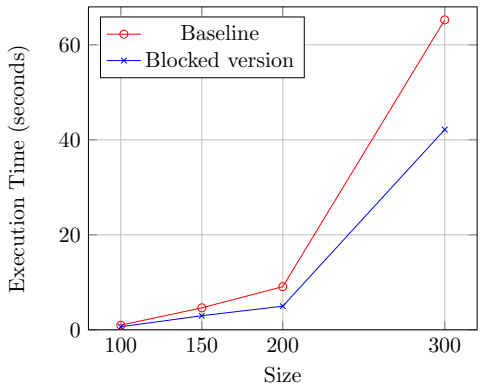

Figure 9: Comparison of baseline and blocked version of *SSOR* and *Laplacian*

8.6 Conclusion

Laplacian showed increased performance when using *blocking* and when splitting the loop in order to have simpler control sequences. *SSOR preconditioner* showed increased performances when applying blocking. Therefore, we can conclude that the baseline implementation is not the optimal one, and we can achieve better performance without relying on parallelization by improving cache temporal and space locality using blocking. These improved version represent a good starting point for parallelization.

9 Finding the best value of ω

As described in Section 2, the flag `-w` specifies the relaxation parameter for the SSOR preconditioner. The default value is 1.9 but is this the best value we could possibly choose?

We used an empirical measurement to verify if better values exists. The data have been obtained by running 4 different mesh size (50, 100, 150 and 200) with different values of `-w` at intervals of 0.02 and recording the execution times. This has been achieved by using the script reported in Listing 1. The data collected have been used to create Figure 10. From Figure 10 and from the other data collected with this experiment, we can conclude that:

- In all problem sizes, greater values of ω yield better performance as long as is below a certain breakpoint value. Once ω goes past that value, the performance degrade as the number of iterations required to reach the solution increases. This is likely due to excessive over-relaxation introduced by high values of ω.
- As the mesh size increase, the best value of ω moves towards 1.99.
- The iteration count and the execution time follow the same trend when ω changes. This is to the point that it's difficult to distinguish in Figure 10 the two lines.

We can therefore conclude that in the configuration tested, 1.90 is a good default value but is not the optimal value for the smallest mesh sizes nor for the bigger ones. We can additionally conclude that the optimal ω value, varies based on the mesh size.

If we wanted to decide a more granular default value, we should tie it to the mesh size and move it further from 1.99 as the mesh size decreases. Nevertheless, this conclusion is only for the configuration tested, more in depth test are necessary for other mesh size dimension and for other right-hand sides.

```
#!/bin/bash

#SBATCH --job-name=submit-seq.sbatch
#SBATCH -D .
#SBATCH --output=execution_result.out

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean -s
make -C "/scratch/nas/1/sca1011/CSA/lab4" CFLAGS="-std=gnu99 -O3 -march=native -Wall -m64 -g
    -DCLOCK=CLOCK_REALTIME" -s

for size in 50 100 150 200; do
  printf -- "---- Running with size: %d ----\n" "$size"
  for w in $(seq 0.05 0.02 1.99); do
    echo -n "$w "
  ../cgp3d.x -w "$w" -M 2000 -p ssor -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'
  done
done

```

Listing 1: Script used to collect data showcased in Figure 10

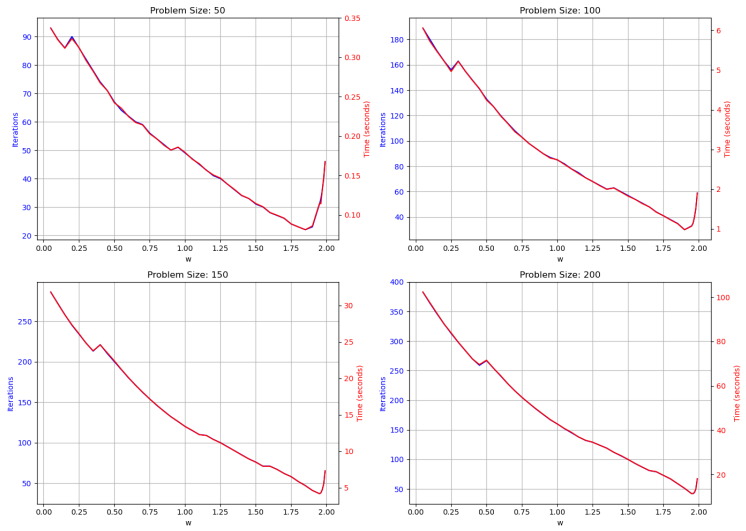

Figure 10: Execution time and iterations count trend when ω varies, on problem with mesh size 100, 150, 200 and 300.

10 Overlap Parameter for Schwarz

Similarly, as ω for the *SSOR* preconditioner, the Additive Schwarz also has a parameter. In the case of Additive Schwarz, the parameter is used to decide the overlaps between subdomains. In the case of *SSOR*, a similar experiment as the one described in Section 9 has been created. The execution time and the number of iteration taken to solve the problem using the *SSOR* preconditioner have been sampled while varying the input value `-o` from 0 (degenerate case) to the mesh size `-n`. The sample of the values of `-o` has been in steps of 2 between 0 and 30, in steps of 10 between 30 and 60 and in steps of 20 between 60 and 200. The data resulting from the experiment are plotted in Figure 11. As the first plot makes the evolution for smaller mesh sizes difficult to track, a second plot where a breakdown for each mesh size is presented in Figure 12.

When setting `-o` to zero, there is no overlap between the iterations. The domain is split with strict boundaries, meaning each subdomain only uses its local data without accessing neighboring subdomains. In this case, this is effectively a *Blocked Jacobi*, and the performances obtained are in fact similar to the ones obtained in Section 8.4.

On the mesh sizes taken into account, the values of `-o` the gives the best execution time is always either 2 or 4 for all the mesh sizes tested. It's safe to say that 10 is not a good optimal value if all the mesh sizes follow the same behavior that the mesh sizes tested have.

Another finding is that once we get past a certain value of `-o` that depends on the mesh size, the number of iterations start to decrease and in all the cases the lowest amount of iteration is obtained when `-n` is equal to `-o`. Nevertheless, as `-o` increase, the overlap increase and therefore the computation time follows suit and the iteration count stops to correlate to the execution time.

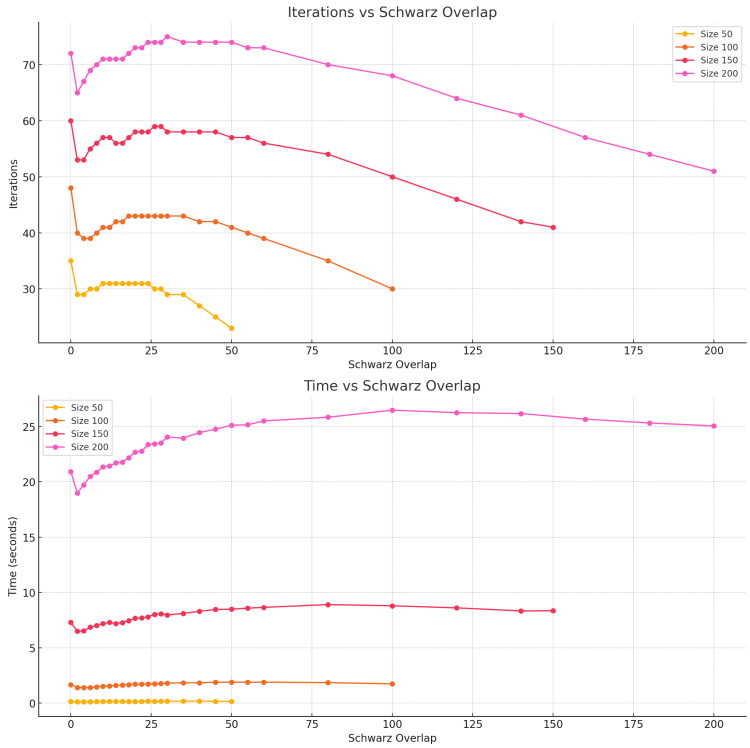

Figure 11:
On the top: evolution of iterations (y-axis) when varying the value of `-o` value (x-axis).
On the bottom: evolution of execution time (y-axis) when varying the value of `-o` value (x-axis)

11 Convergence when varying the right-hand side

Another parameter that is interesting to analyze is the impact of the right-hand side on the number of iterations and on the execution time. Again, this is done by performing an empirical test. The script used to do so is visible in Listing 2. The two version taken into account, `RHS0` and `RHS1`, are the right-hand sides described more in depth in Section 2.

The data collected are displayed in Figure 2 where it is possible to see how each preconditioner (ID, SSOR, and Schwarz) behaves differently as the right-hand side (RHS) changes from RHS0 to RHS1.

```
#!/bin/bash

#SBATCH --job-name=submit-seq.sbatch
#SBATCH -D .
#SBATCH --output=execution_result.out

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean -s
make -C "/scratch/nas/1/sca1011/CSA/lab4" CFLAGS="-std=gnu99 -O3 -march=native -Wall -m64 -g
    -DCLOCK=CLOCK_REALTIME -DUSE_RHS0" -s

printf -- "---- Running with RHS0 ----\n"

for size in 50 100 150 200; do
  printf -- "---- Running with size: %d ----\n" "$size"
    echo -n "ID: "
    ../cgp3d.x -M 500 -p id -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'

    echo -n "SSOR: "
    ../cgp3d.x -w 1.90 -M 500 -p ssor -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'

    echo -n "SCHWARZ : "
    ../cgp3d.x -o 2 -M 500 -p as -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'
done

make -C "/scratch/nas/1/sca1011/CSA/lab4" clean -s
make -C "/scratch/nas/1/sca1011/CSA/lab4" CFLAGS="-std=gnu99 -O3 -march=native -Wall -m64 -g
    -DCLOCK=CLOCK_REALTIME" -s

printf -- "---- Running with RHS1 ----\n"

for size in 50 100 150 200; do
  printf -- "---- Running with size: %d ----\n" "$size"
    echo -n "ID: "
    ../cgp3d.x -M 500 -p id -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'

    echo -n "SSOR: "
    ../cgp3d.x -w 1.90 -M 500 -p ssor -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'

    echo -n "SCHWARZ : "
    ../cgp3d.x -o 2 -M 500 -p as -n "$size" |
      grep "residual reduction" |
      sed -n 's/\([0-9]*\) steps.*time \([0-9.]*\) seconds.*/Iterations: \1, Time: \2 seconds/p'
done


```

Listing 2: Script used to collect data showcased in Figure 13

For the ID preconditioner, both the iteration count and time increase significantly with problem size, especially noticeable for RHS1. This is likely due to its lack of strong convergence properties, requiring more iterations and resulting in much higher times as the problem size grows.

The SSOR preconditioner, on the other hand, maintains a similar iteration count for RHS0 across all sizes. However, with RHS1, both the iteration count and the time grow moderately but remain relatively manageable compared to the ID preconditioner. This indicates that SSOR offers more robustness and stability with varying problem sizes and RHS configurations.

The Schwarz preconditioner is somewhere in between the two. It requires slightly more iterations than SSOR, especially with RHS1, but the time taken remains lower than ID and only moderately higher than SSOR. Schwarz seems to adapt better to changes in the RHS while still controlling the time growth across problem sizes, showcasing its effectiveness in handling different configurations with a stable convergence rate.

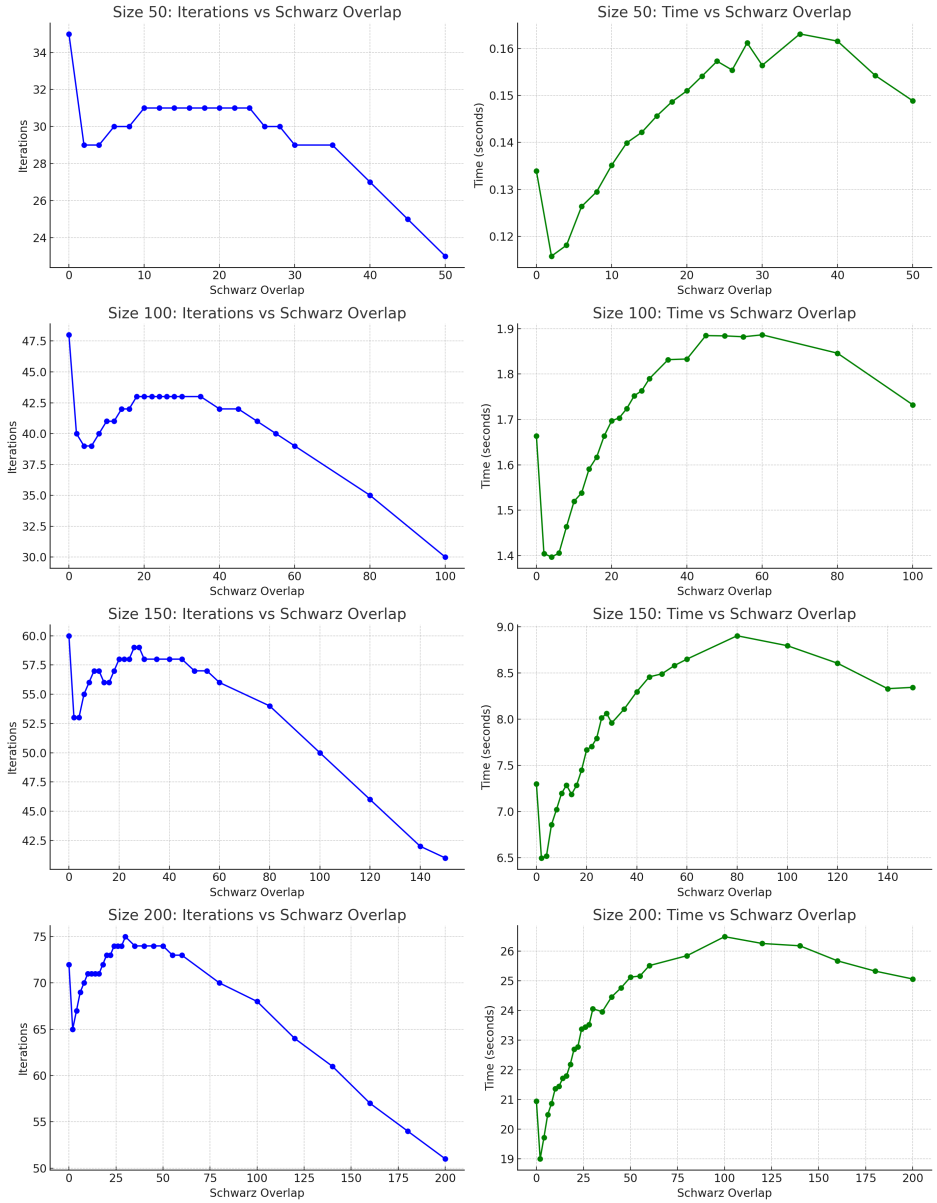

Figure 12:
On the top: breakdown for each mesh size of the evolution of iterations (y-axis) when varying the value of `-o` value (x-axis).
On the bottom: breakdown for each mesh size of the evolution of execution time (y-axis) when varying the value of `-o` value (x-axis)

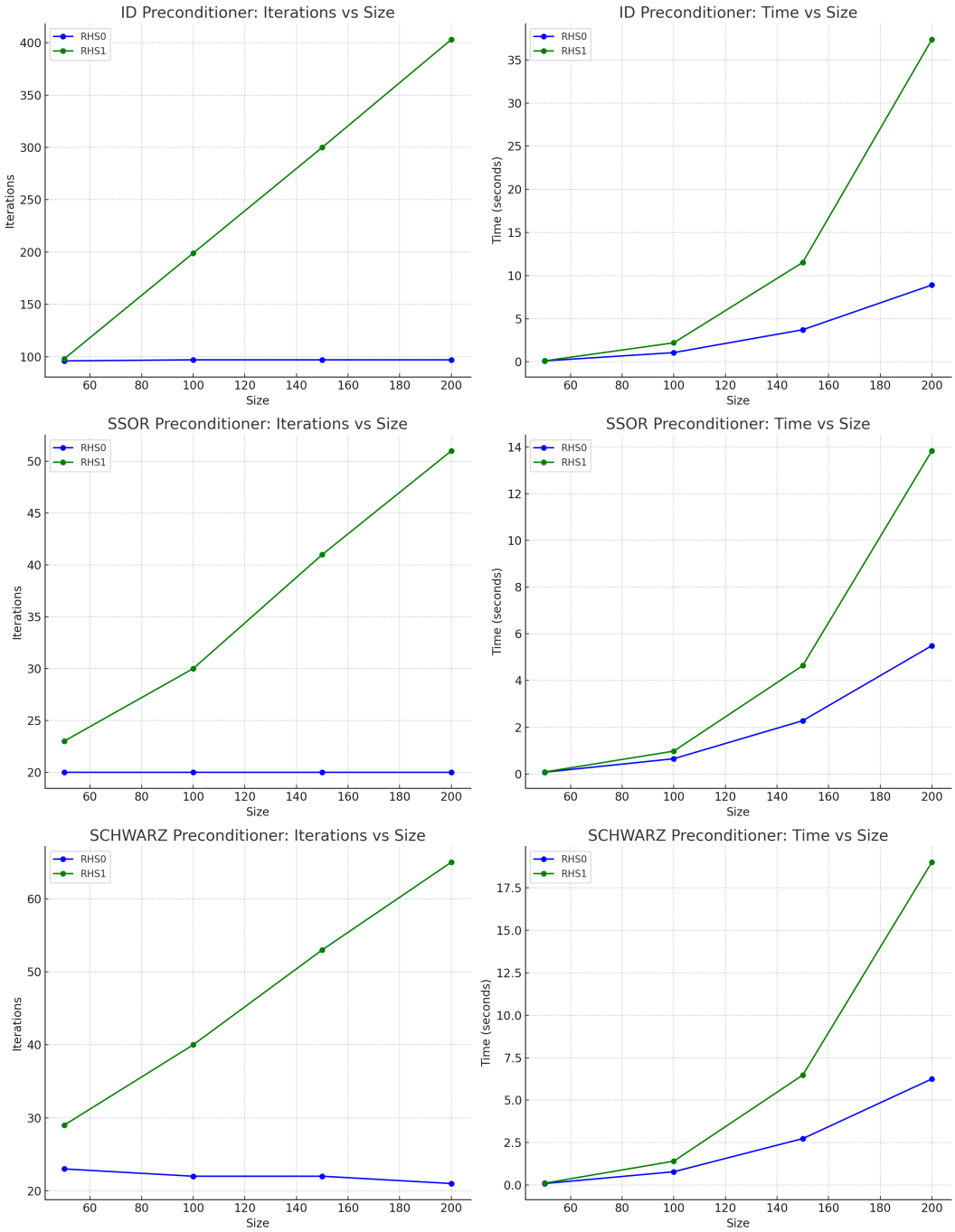

Figure 13: Trend of iteration and computing time when varying right-hand side.