Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing Lab **Institute of Computing**

Student: Kaan Egemen Sen

Solution for Project 3

This project will introduce you to a parallel space solution of a nonlinear PDE using OpenMP.

1. Task: Implementing the linear algebra functions and the stencil operators [35 Points]

In this part, the missing functions *hpc_xxxxx*() in the file *linalg.cpp* are completed, as well as the missing stencil kernels in the file *operators.cpp*. The following is the output for domain size 128 × 128, 100 time steps, and simulation time $0 - 0.005$ s. The visualization can be found below figure :

```
kaans-MBP-1323:Project 3 usi$ ./main 128 100 0.005
================================================================================
                        Welcome to mini-stencil!
version   :: Serial C++
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.128283 seconds
1511 conjugate gradient iterations, at rate of 11778.6 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

The Figure that outputs this operations can be seen below :

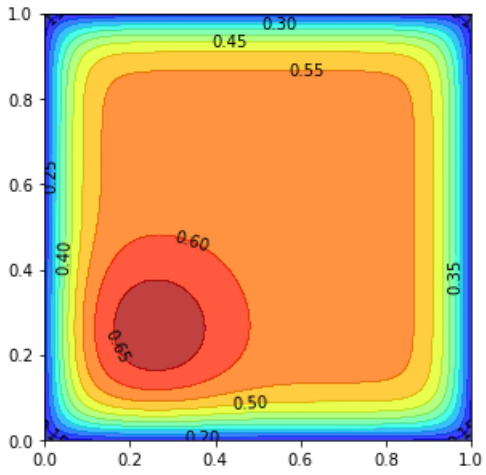

Figure 1: Heatmap result with serial computation

1.1. The diffusion stencil [10 Points]

- Parallelized the stencil operator in *operators.cpp*

What role do the boundary loops play?

Boundary loops are responsible for handling the grid points at the edges or boundaries of the computational domain. These are the grid points that don't have a full set of neighboring points in all directions, which is necessary for applying the stencil. Boundary loops are essential for handling edge cases. They ensure that boundary grid points are updated correctly despite having fewer neighbors. In other words, they apply a modified version of the stencil operator tailored to the specific boundary conditions.

2. Task: Adding OpenMP to the nonlinear PDE mini-app [50 Points]

The welcome message in *main.cpp* is modified to include the OpenMP version title and the number of threads. The following is the output for domain size 128×128, 100-time steps, simulation time $0 - 0.005$s , and for 8 threads.

- That this is the OpenMP version.
- The number of threads it is using.

```
kaans-MBP-1323:Project 3 usi$ export OMP_NUM_THREADS=8
kaans-MBP-1323:Project 3 usi$ ./main 128 100 0.005
================================================================================
                        Welcome to mini-stencil!
version   :: C++ OpenMP
Threads   :: 8
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================

--------------------------------------------------------------------------------
simulation took 0.14885 seconds
1511 conjugate gradient iterations, at rate of 10151.2 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Then added OpenMP directives to all functions hpc_xxxxx() in the file linalg.cpp

```
        #pragma omp parallel for private(i) default(shared)
```

As a final step parallelized the stencil operator in operators.cpp with adding the line of code below

```
        #pragma omp parallel for private(j) default(shared)
```

The output of parallelized computation can be seen below

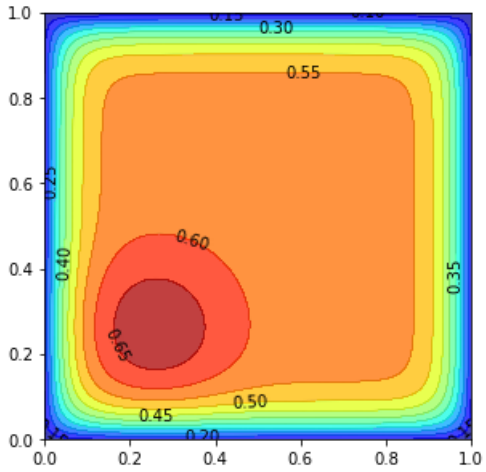

Figure 2: Heatmap result with parallel computation

2.1. Strong Scaling

The program is executed for $p = 1, 2, ..., 24$, where p is the number of threads. Different combinations of grid size and number of threads are experimented with. Where N is the grid size, the benchmarks used are:

- N = 64×64
- N = 128x128
- N = 256x256
- N = 512x512
- N = 1024x1024

The parallel scaling charts for all of the experiments, grouped by N size, can be found in the Figures below :

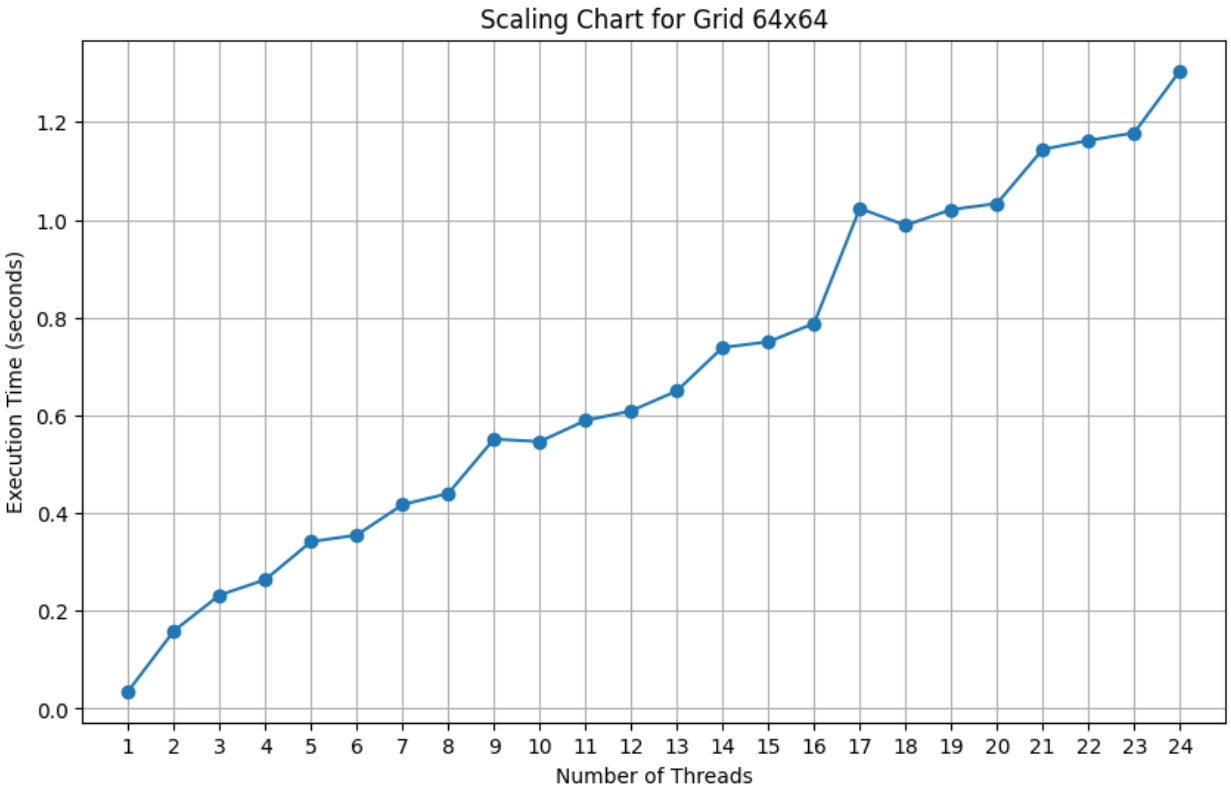

Figure 3: Grid 64x64

The 64x64 grid (3) clearly shows better results when using a small number of threads: in particular, using 1 thread consistently demonstrates the best performance, while getting progressively worse with each added thread, with a pretty odd dip when going from 16 to 17 threads.

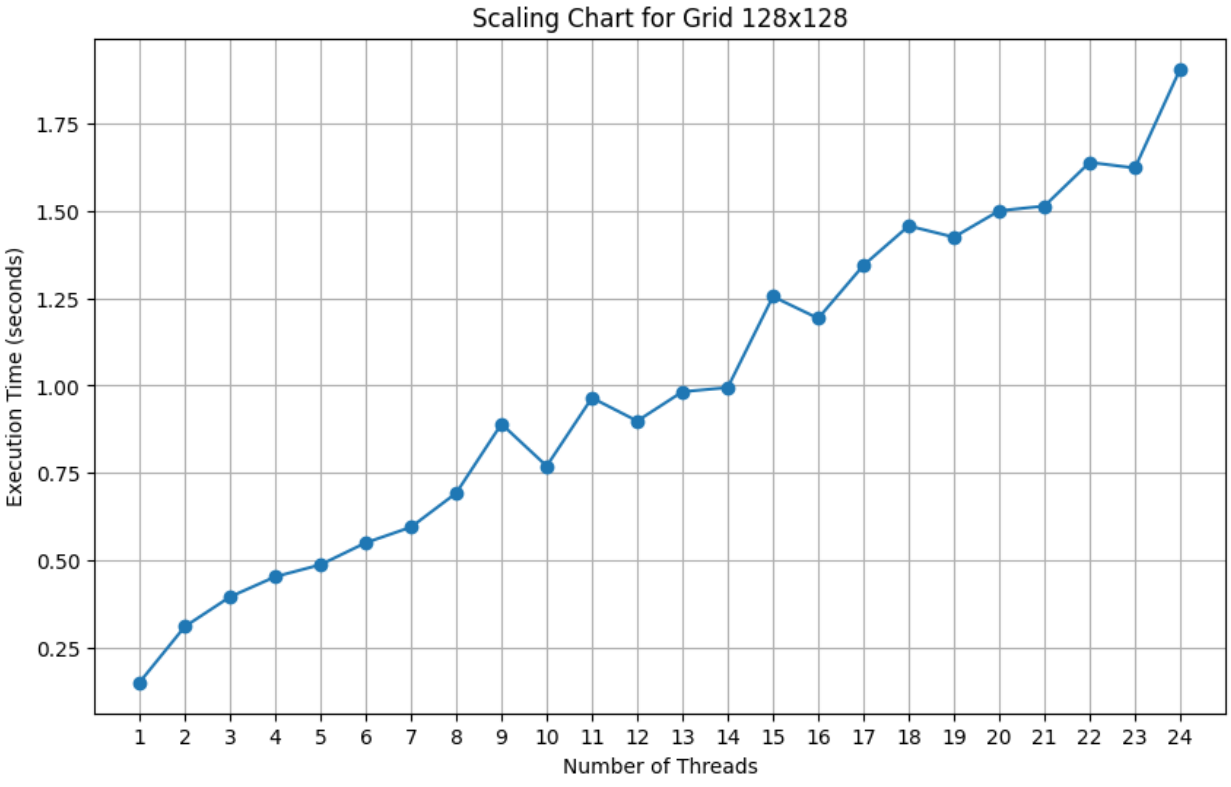

Figure 4: Grid1 128x128

The 128x128 grid (4) presented a similar performance graph to the 64x64 one, but with a more prominent increase in performance when going from a single thread to multiple ones.

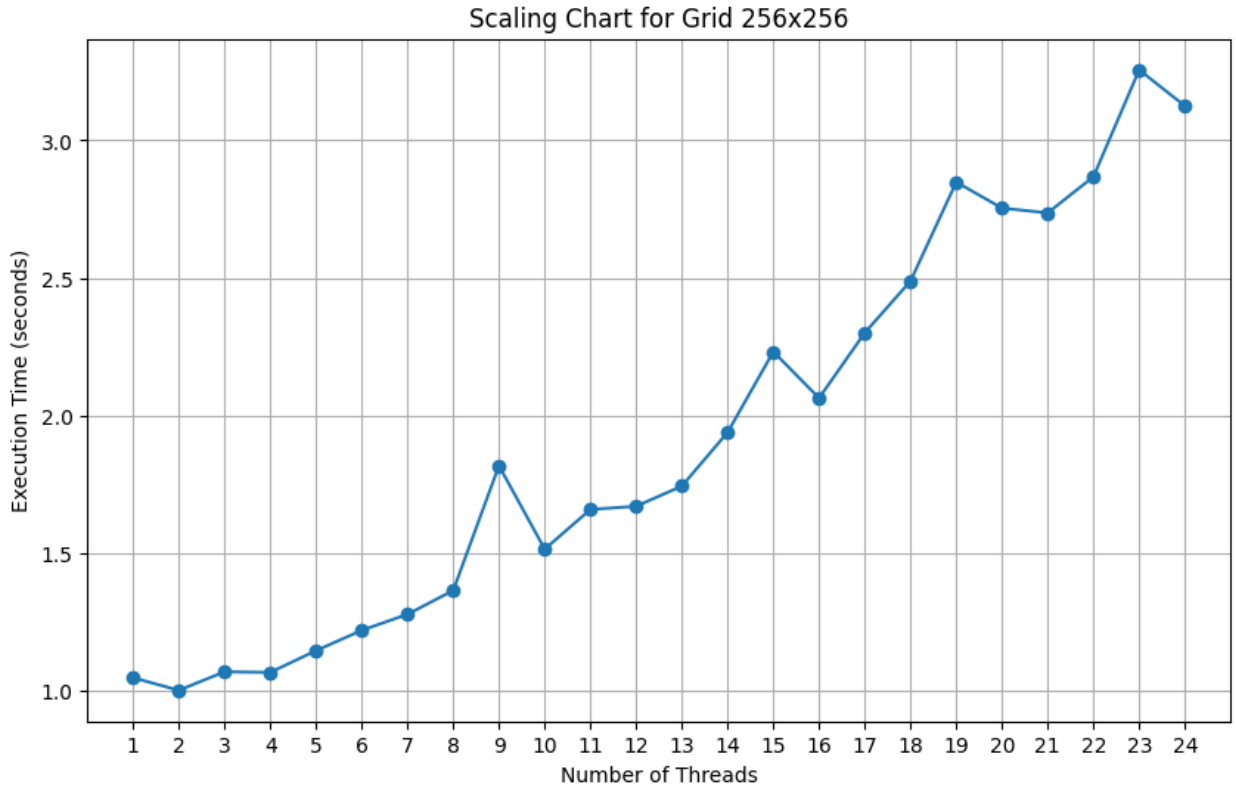

Figure 5: Grid 256x256

The same can be said for the 256x256 grid (5), which shows an even bigger relative difference in performance between 1 and multiple threads. But for this specific case, we have the fastest execution time by using 2 threads.

The 64 and 128 grid tests in particular exhibited worse performance when using a higher number of threads than their sequential implementation, and this is because the cost of creating those threads exceeded the actual cost of the operations.

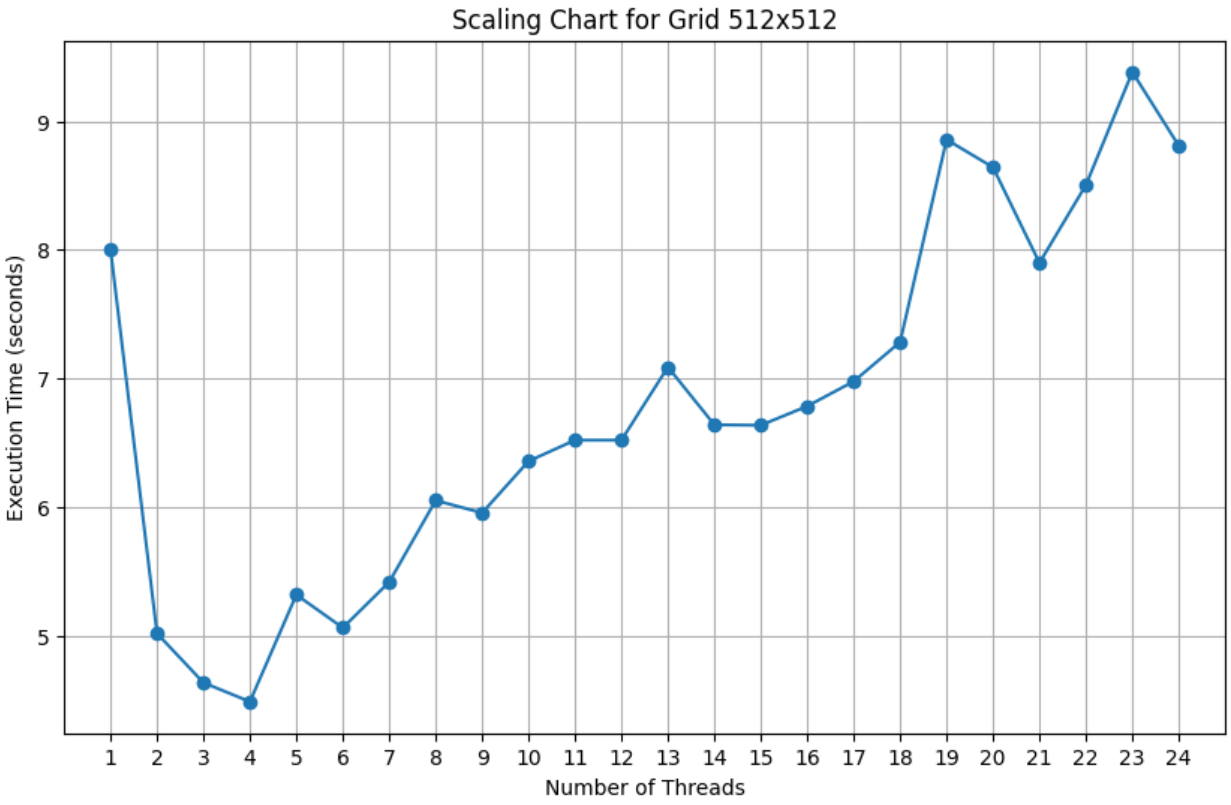

Figure 6: Grid 512x512

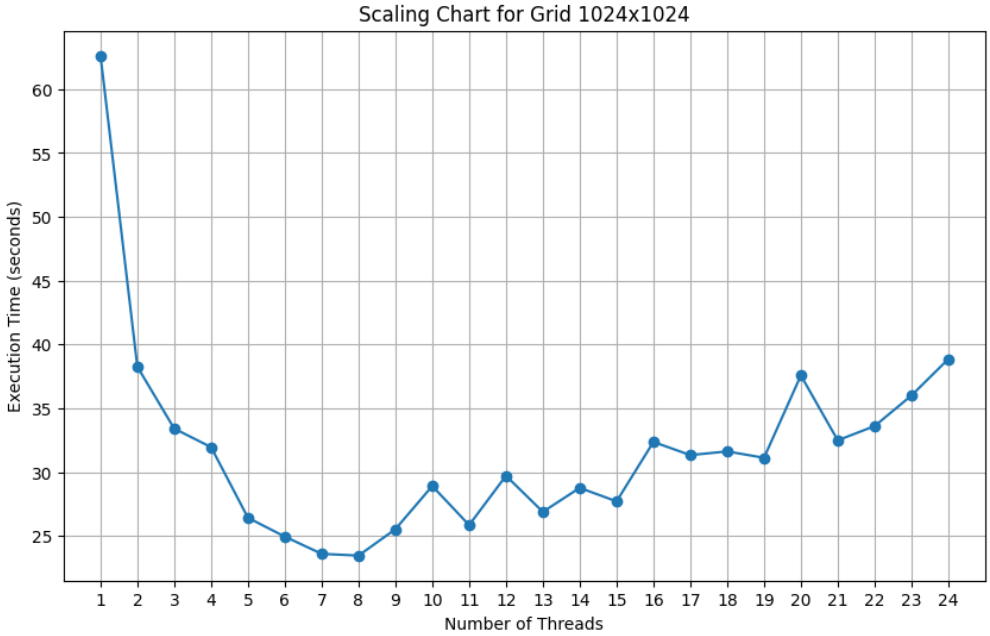

Figure 7: Grid 1024x1024

Both the 512x512 (6) and 1024x1024 grid (7), no matter the number of threads used in the test, always show better performance than their sequential version. The cost of the operations in this case is higher than the cost of thread creation.The best performance with calculation is at 8 threads for 1024 and 4 threads for 512 grid we can clearly see that

The implementation of a threaded OpenMP PDE solver that produces bitwise-identical results

without any parallel side effects would not be possible. This is because the execution of the program is not done in an ideal environment. There are various processes running in the background, and insignificant differences are expected in the results when working with multiple threads. If bitwise- identical results are desired, the measurements taken will cause side effects in terms of performance, which will make using threads useless.

2.2. Weak Scaling

Next, the exercise asked to show a weak scaling plot of the program. I wanted to work with a 64x64 grid size for each thread since the 64x64 grid showed the best performance when using 1 thread. So that would mean 1 thread for the 64x64 grid, 4 threads for the 256x256 grid, 6 threads for the 384x384, and 8 threads for the 512x512 one. Because the local machine has a maximum of 8 performance cores.

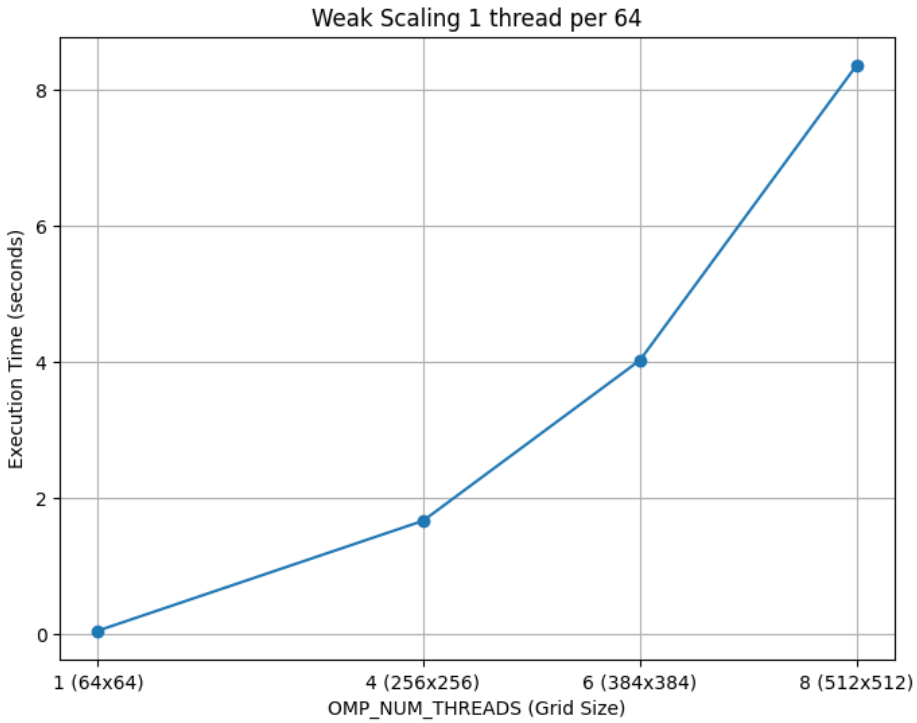

Figure 8: Weak Scaling

We can see however that the performance worsens. While each thread is working on the same 64x64 grid, my guess is that the number of operations still increases exponentially, hence the exponential growth in the time needed to run the program.

Bonus

Theoretically yes, SIMD works by performing the same operation simultaneously on multiple data points, like vectors. It could be implemented for both the linear algebra functions, which work on vectors, and stencil operators. The real question is if it can work faster than the parallel version, but it probably does.

Ref

Wikipedia for weak-strong scaling