Università della Svizzera italiana

Institute of Computing CI

**High-Performance Computing** **2022**

Student: SIMONE TARENZI Discussed with: LUCA PERNIGO

## Solution for Project 3

Due date: 09.11.2022, 23:59

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

Nothing really interesting to be said about implementing the hpc functions and stencil operators.

```
simulation took 0.224768 seconds
1510 conjugate gradient iterations, at rate of 6718.05 iters/second
300 newton iterations
```

Figure 1: the output of the terminal for the 128x128 grid

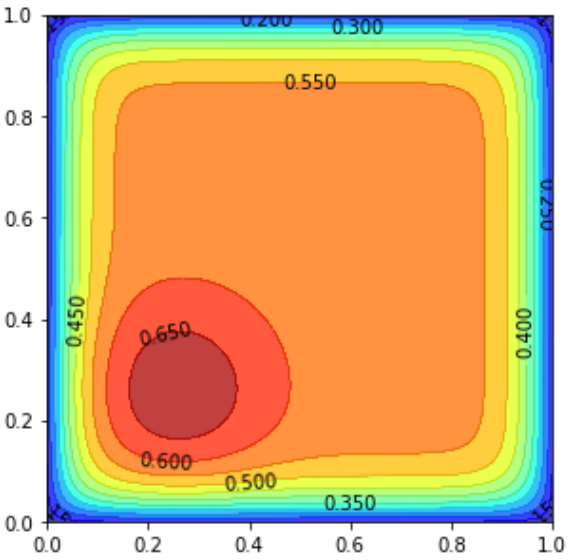



Figure 2: the output.png file of the 128x128 grid created by plotting.py

I've also tried doing a test with a 500x500 grid size.

```
simulation took 11.0875 seconds
5234 conjugate gradient iterations, at rate of 472.064 iters/second
300 newton iterations
```

Figure 3: the output of the terminal for the 500x500 grid

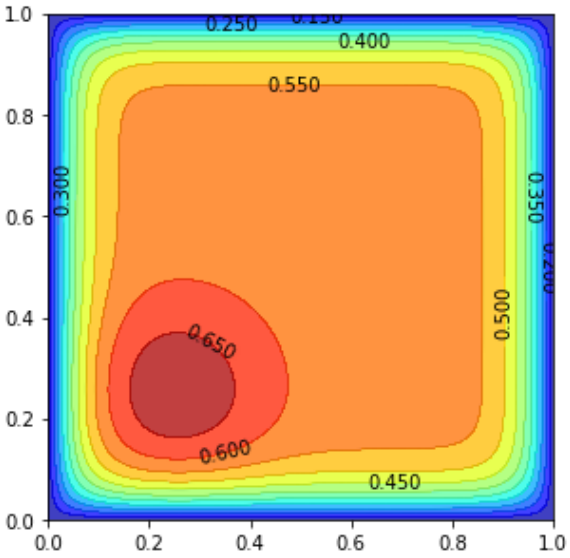



Figure 4: the output.png file of the 500x500 grid created by plotting.py

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

First of all I've added the int numThreads in the data header that can be set directly by calling the main from the terminal. This makes it way easier to run and plot the result of the program using a varying number of OpenMP threads.
After that, implementing the parallel versions of the hpc_xxxx() functions was very straightforward.

Using 4 threads showed great results for both the 128 and 256 grids, but the 512 had its best ones while using 16 threads.

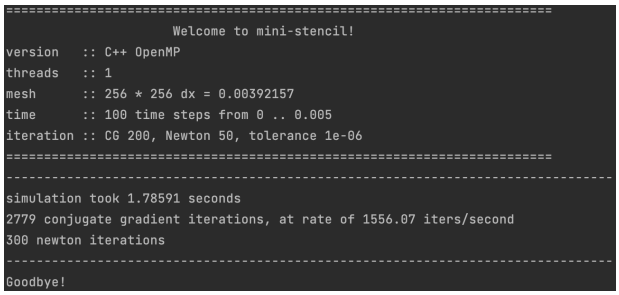
```
================================================================================
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 1
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.23751 seconds
1510 conjugate gradient iterations, at rate of 6357.63 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

(a) Using 1 thread on a 128x128 grid

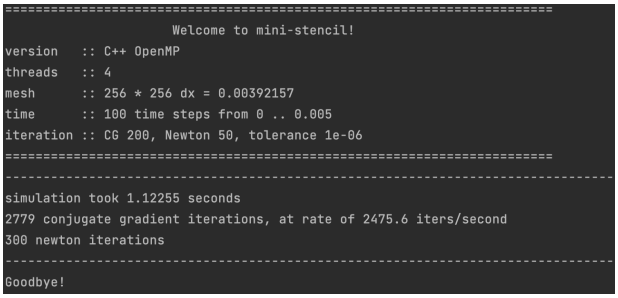
```
================================================================================
                        Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 4
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
================================================================================
--------------------------------------------------------------------------------
simulation took 0.125471 seconds
1514 conjugate gradient iterations, at rate of 12066.6 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

(b) Using 4 threads on a 128x128 grid

Figure 5: comparison between two runs of the program on a 128x128 grid with 1 and 4 threads

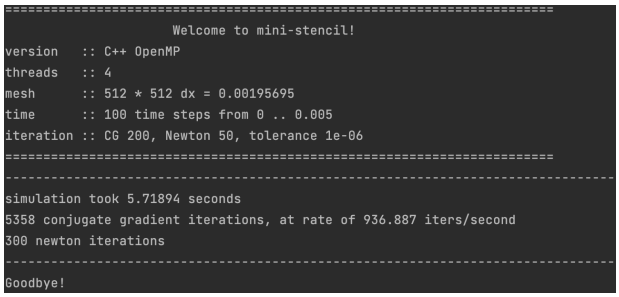



(a) Using 1 thread on a 256x256 grid

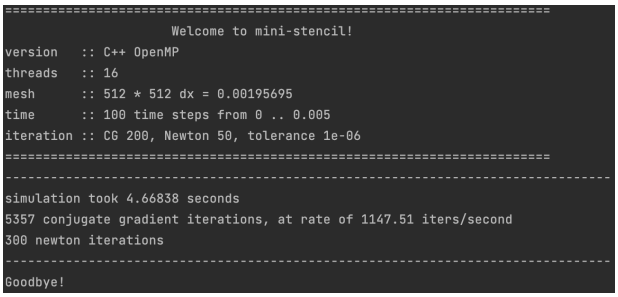



(b) Using 4 threads on a 256x256 grid

Figure 6: comparison between two runs of the program on a 256x256 grid with 1 and 4 threads




(a) Using 4 threads on a 512x512 grid




(b) Using 16 threads on a 512x512 grid

Figure 7: comparison between two runs of the program on a 512x512 grid with 4 and 16 threads

Next, the exercise asked to implement the parallelization for the stencil operators of the grid and plot the performance of various grid sizes while using a number of threads going from 1 to 24.
First I implemented parallelization for the interior grid points and the inner east, west, north and south boundaries.
Then I created a manual_plot.py file that shows the performance for the various thread numbers.
I also modified the main class so that it writes to a txt file the time taken for the program to run.
Being the first time I used Python, I simply copy-pasted the results in the manual_plot.py file.

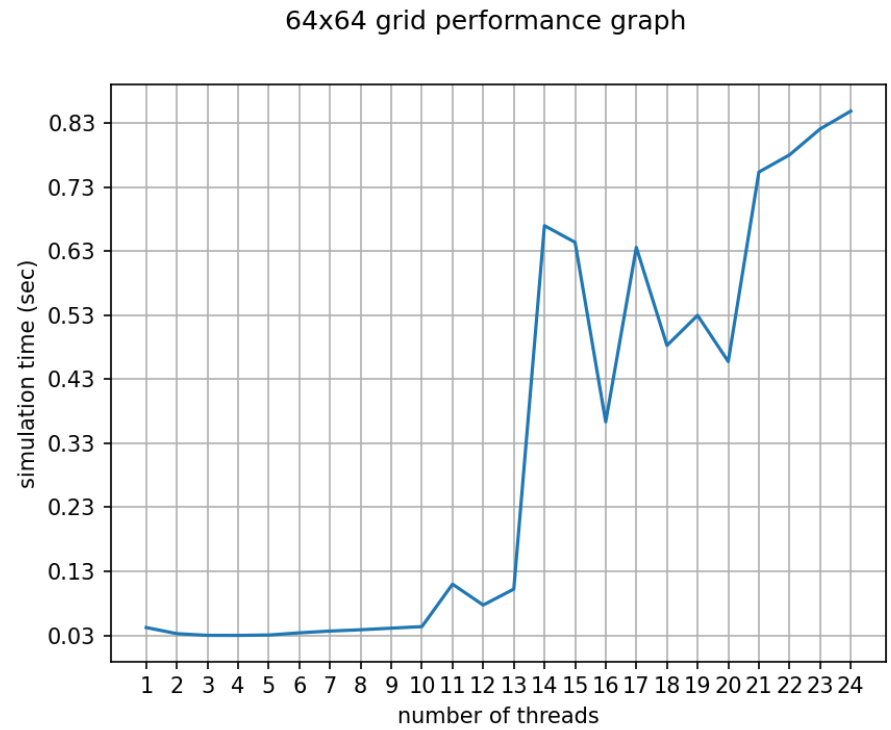



Figure 8: performance of the parallel version of the program for a 64x64 grid

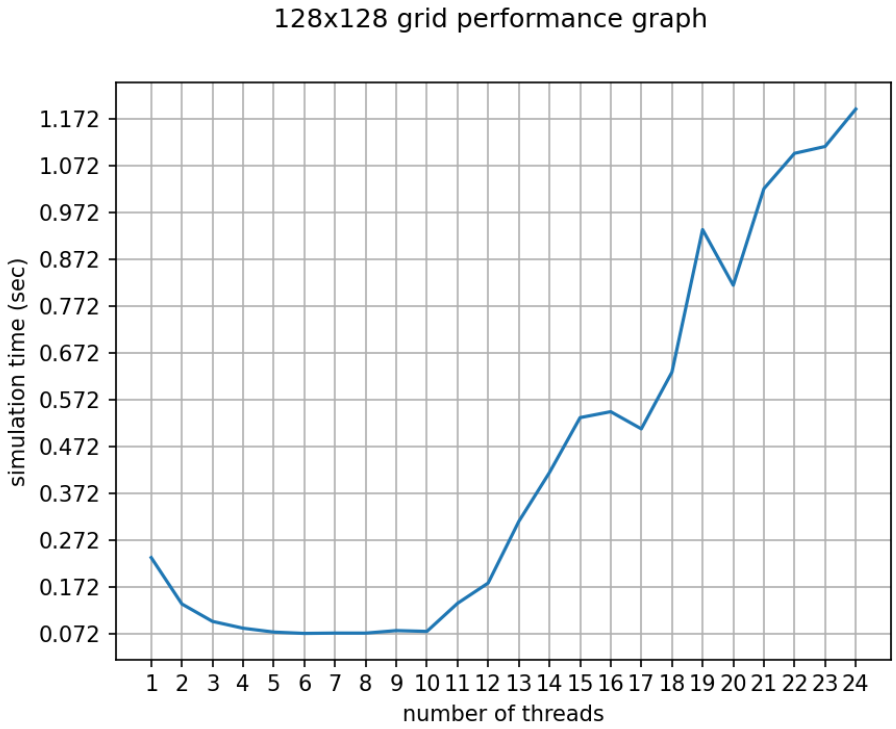



Figure 9: performance of the parallel version of the program for a 128x128 grid

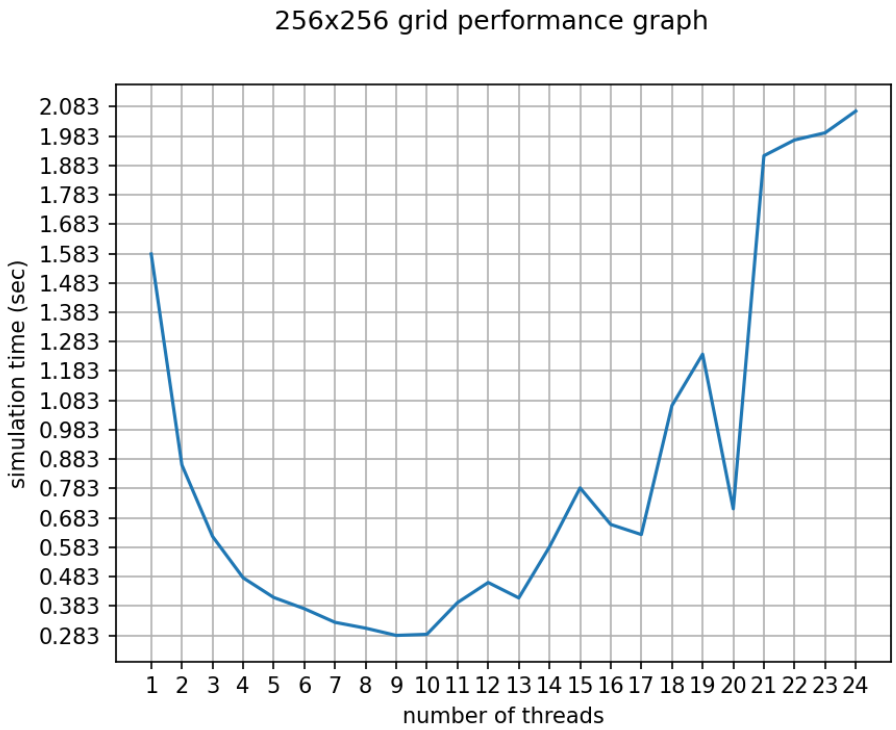



Figure 10: performance of the parallel version of the program for a 256x256 grid

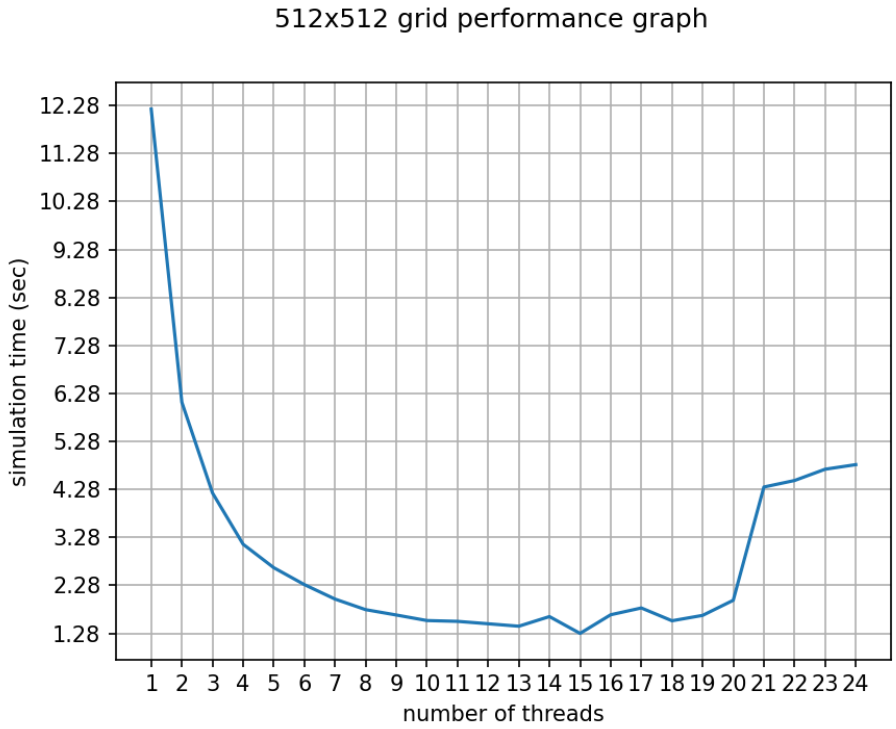



Figure 11: performance of the parallel version of the program for a 512x512 grid

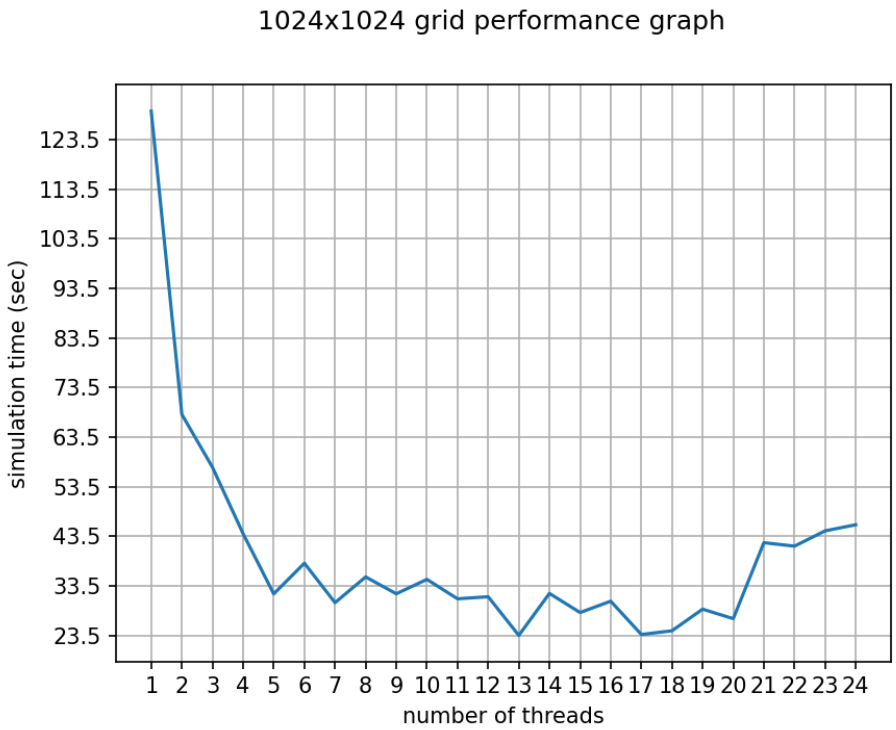



Figure 12: performance of the parallel version of the program for a 1024x1024 grid

The 64x64 grid (Fig.8) clearly shows the better results when using a small number of threads: in particular, using 4 threads consistently demonstrates the best performance, while getting progressively worse with each added thread, with a pretty odd dip when going from 13 to 14 threads.

The 128x128 grid (Fig.9) 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.

The same can be said for the 256x256 grid (Fig.10), which shows an even bigger relative difference in performance between 1 and multiple threads.

The 64 and 128 grid tests in particular exhibited worse performance when using an 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.

Both the 512x512 (Fig.11) and 1024x1024 grid (Fig.12) instead, 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 bigger than the cost of thread creation.
However, after the 20 threads mark, I noticed that all tests show a dip in performance: this is probably because the cluster node doesn't support more than 20 threads, and so adding them is not only useless, since they are not used, but it also worsen performance because creating a thread is never a free operation.

Next, the exercise asked to show a weak scaling plot of the program.
I wanted to work with a 32x32 grid size for each thread, since the 64x64 grid showed the best performance when using 4 threads, but that would mean that the 256x256 grid would already use 64 threads, and we already know that the limit is 20.
We can at least work with 1 thread for each 64x64 grid, so that would mean 4 threads for the 128x128 grid and 16 threads for the 256x256 one.

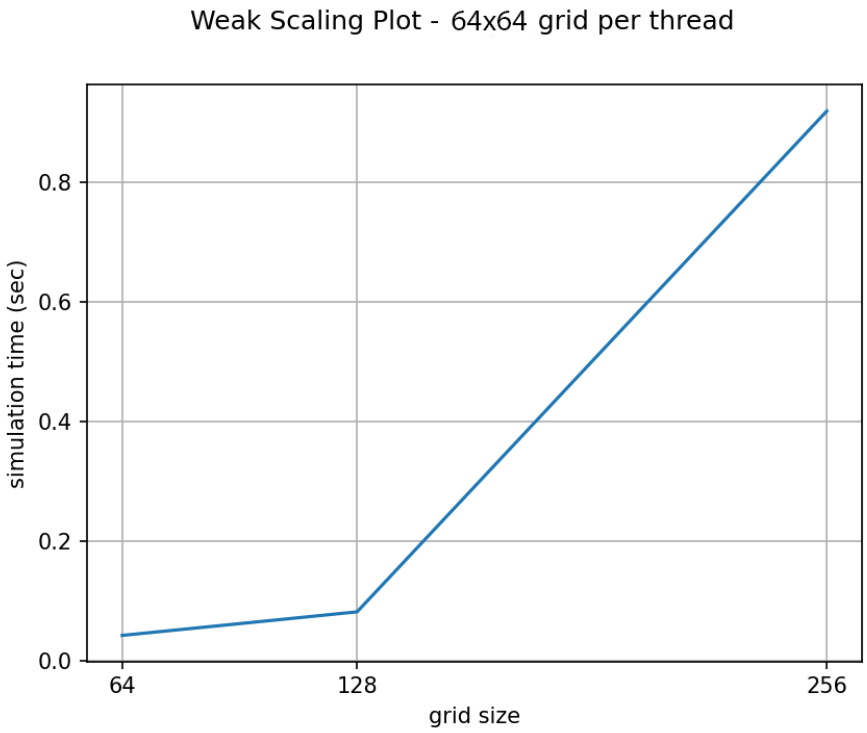



Figure 13: performance of the 64x64 grid with 1 thread, the 128x128 grid with 4 threads and 256x256 grid with 16 threads

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 time needed to run the program.

## 3. Bonus Question [5-10 Points]

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 even on the stencil operators. The real question is if it can work faster than the parallel version, but it probably does.

## 4. Task: Quality of the Report [15 Points]