Università della Svizzera italiana | Institute of Computing CI

High-Performance Computing **2022**

Student: Nicolai Hermann Discussed with: Michele Cattaneo

Solution for Project 3

Due date: 02.11.2022, 23:59

HPC 2022 — Submission Instructions
(Please, notice that following instructions are mandatory: submissions that don't comply with, won't be considered)

- Assignments must be submitted to iCorsi (i.e. in electronic format).
- Provide both executable package and sources (e.g. C/C++ files, Matlab). If you are using libraries, please add them in the file. Sources must be organized in directories called:
  Project_number_lastname_firstname
  and the file must be called:
  project_number_lastname_firstname.zip
  project_number_lastname_firstname.pdf
- The TAs will grade your project by reviewing your project write-up, and looking at the implementation you attempted, and benchmarking your code's performance.
- You are allowed to discuss all questions with anyone you like; however: (i) your submission must list anyone you discussed problems with and (ii) you must write up your submission independently.

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

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

1.1. Linalg

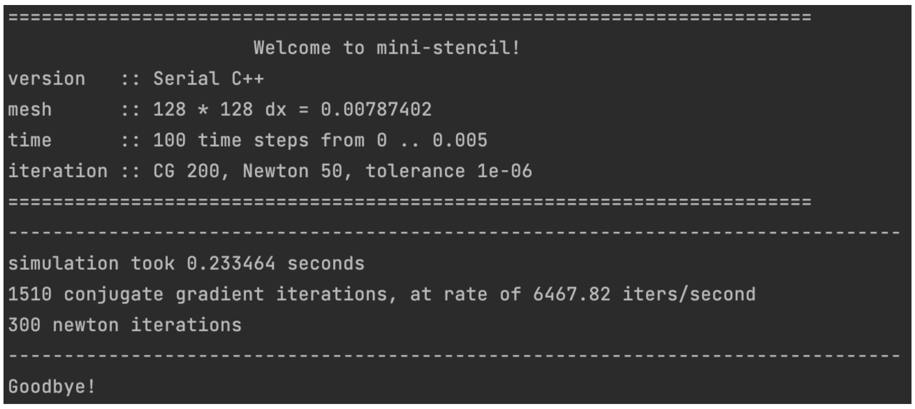
Implementing all `hpc_xxxxx()` functions could be solved by using a simple for loop, iterating through all components and implementing the formula given in the docsting of the function. If it was an array it was indexed with `i` otherwise it was a constant which could be used immediately. A high potential for parallelization was noted immediately.

1.2. Stencil Kernel

For the kernel multiple scenarios were needed to be covered. First of all, for all the interior points the current point and all its neighbours were required for the computation. The point value of the last time step $s_{i,j}^k$ was given by the variable `y_old`. For the outermost ring of the grid, all of the points needed at least one boundary point. The corners required two boundary points and were already given in the code. For the remaining points one of the neighbours was given by a boundary point. For example the points at $i = 0$ relied on the western boundary points. Therefore, the access to the grid points at $(-1, j)$ (the left neighbours) was replace by the western boundary points $boundW_j$. The same principle was applied for all other exterior points: North boundary for $s_{i,j+1}$, East boundary for $s_{i+1,j}$, South boundary for $s_{i,j-1}$.

1.3. Validation

After all changes were made the code was compiled and run for a 128×128 grid, 100 time steps with a simulation time from 0 - 0.005 seconds. The console output of the mini-app can be found in Fig. 1. The execution was twice as fast as the reference output provided in the assignment but the conjugate gradient iterations are approximately the same and the newton iterations match perfectly. Furthermore, the results were plotted using the provided `plotting.py` script and the created plot can be seen in Fig 2. The plot matches the reference plot perfectly. The strong alignment of the results with the references suggests that the stencil and linalg operations were implemented correctly.



```
========================================================================
                      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 0.233464 seconds
1510 conjugate gradient iterations, at rate of 6467.82 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 1: Console output after running the mini-app post implementation.

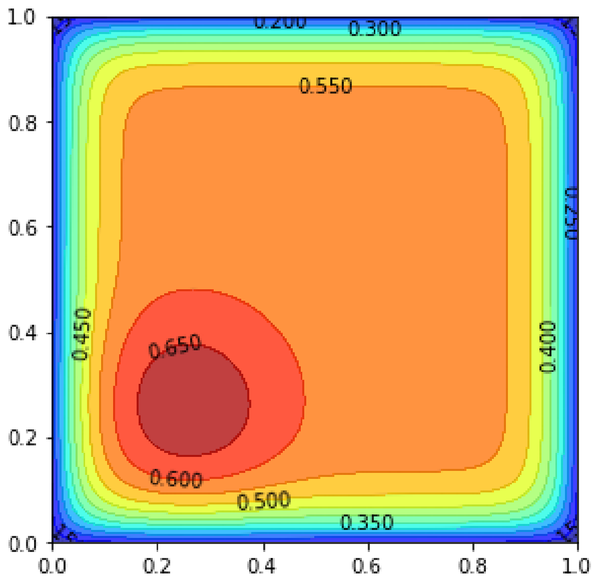


Figure 2: Output of the mini-app for a domain discretization into 128×128 grid points, 100 time steps with a simulation time from 0 - 0.005 s after code was correctly implemented.

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

2.1. Linalg

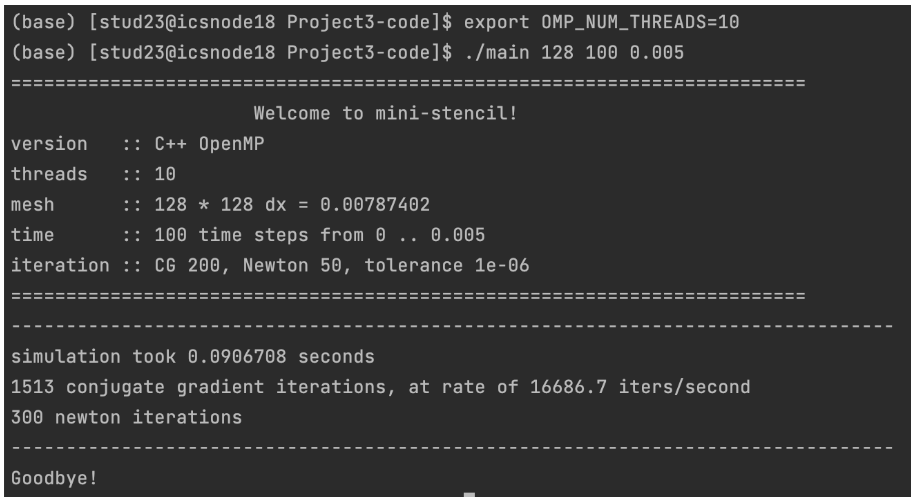
Starting with the `hpc_dot` and `hpc_norm2` where the arrays are reduced to one value. To parallelize the computation the pragma `omp parallel for reduction(+: result)` was used. In all other cases a simple `omp parallel for` was chosen since all for loop iterations were independent from each other and had the same computational complexity.

2.2. Stencil Kernel

Multiple regions of the kernel were identified for parallelization. The computation of the interior gird points and the for loops computing the boundary points except for the corner points. Since in general all points can be computed independently it makes sense to distribute the points among the threads. To achieve that the `collapse(2)` pragma was used in combination with `omp parallel for` to make sure that the points are parallelized and not the rows. The same idea was applied for the borders, but since they are only iterating through one column or row, `collapse` is not needed and the `omp parallel for` is sufficient.

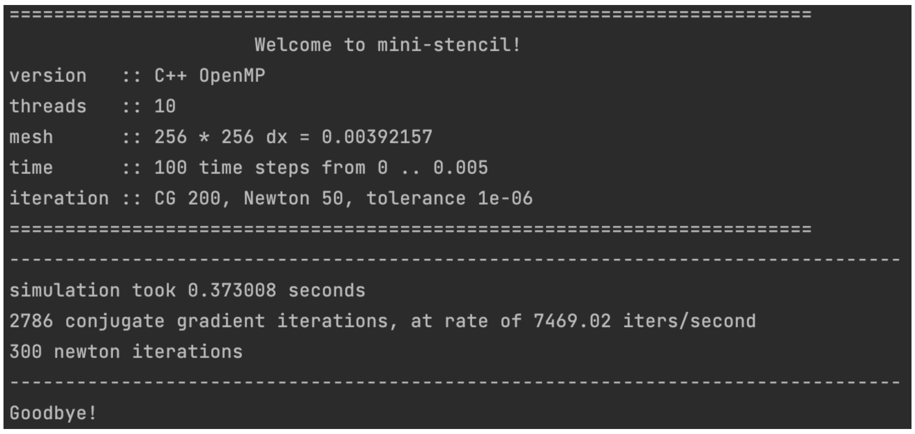
2.3. Results

After adapting and extending the welcome message of the mini-app the following output using 10 threads for a grip size of 128 and 256, 100 time steps with a simulation time from + - 0.005 seconds was obtained. The result for a grid size of 128 can be seen in Fig. 3 and for 256 in Fig. 4. Significant speedups can be noticed. The plot also remained the same indicating that the parallelization of the mini-app was implemented correctly.



```
(base) [stud23@icsnode18 Project3-code]$ export OMP_NUM_THREADS=10
(base) [stud23@icsnode18 Project3-code]$ ./main 128 100 0.005
========================================================================
                      Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 10
mesh      :: 128 * 128 dx = 0.00787402
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
========================================================================
--------------------------------------------------------------------------------
simulation took 0.0906708 seconds
1513 conjugate gradient iterations, at rate of 16686.7 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 3: Console output of the parallelized mini-app with a grid size of 128.



```
========================================================================
                      Welcome to mini-stencil!
version   :: C++ OpenMP
threads   :: 10
mesh      :: 256 * 256 dx = 0.00392157
time      :: 100 time steps from 0 .. 0.005
iteration :: CG 200, Newton 50, tolerance 1e-06
========================================================================
--------------------------------------------------------------------------------
simulation took 0.373008 seconds
2786 conjugate gradient iterations, at rate of 7469.02 iters/second
300 newton iterations
--------------------------------------------------------------------------------
Goodbye!
```

Figure 4: Console output of the parallelized mini-app with a grid size of 128.

2.4. Strong Scaling

For the strong scaling analysis the mini-app was run for multiple grid sizes and thread counts. The results are depicted in Fig 5.

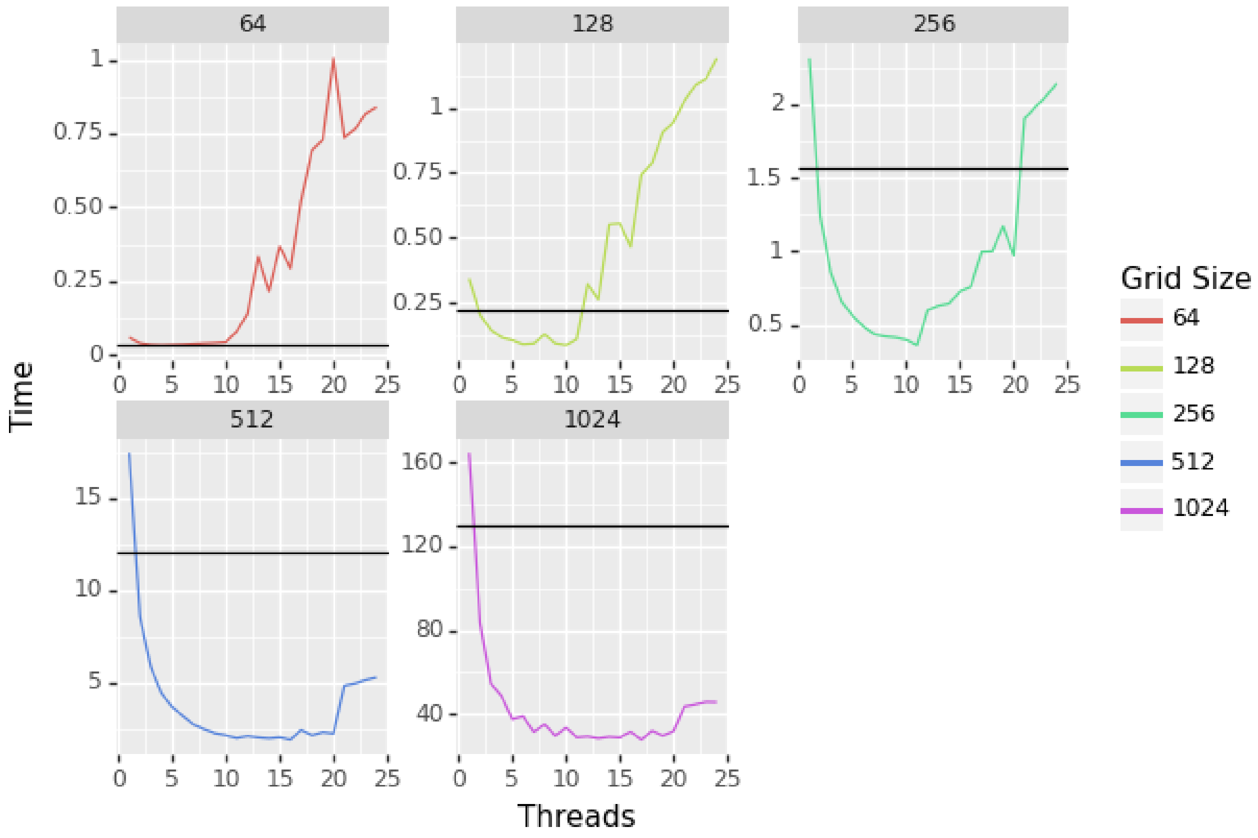


Figure 5: Benchmark plots of the parallel implementation for different numbers of threads and grid sizes. The black horizontal line is the serial performance and serves as a baseline.

The black line in each subplot serves as a baseline and indicates the performance of the serial performance for the given grid size. For the 64×64 grid we can observe that using multiple threads does not lead to a significant speedup. There is a benefit for small thread counts but the overhead of managing threads quickly takes over and dominates the run time. This is mainly due to the problem not being large enough. However, as the grid size increases the work to distribute increases with it and the overhead gets neglectable. So generally it can't be said that more threads increase the performance. This is only true as long as there is enough work to distribute.

Bitwise identical results are hard to ensure. By chance it might be possible but for the most cases it won't be true. Through the parallelization the order in which the arithmetics are performed is not given. This could introduce floating point imprecision earlier or later in the execution leading to the addition or multiplication of different values which could trigger imprecision. The main source of the imprecision variance is introduced in the `hpc_norm2` and `hpc_dot`. In those loops the order of adding to the `result` variable can change frequently. On the contrary for the other for loops we had to implement, all iterations were independent of each other, meaning that changing the order of execution has no impact whatsoever. So if we remove the parallelization from `hpc_norm2` and `hpc_dot` no imprecision variance is introduced from the code I have implemented. In this case the results might be bitwise identical. (If not the imprecision was not caused by my parallelization.)

2.5. Weak Scaling

For the weak scaling study the number of grid points was increased relative to the thread count, meaning that the workload per thread stays the same throughout the experiment. (See Table 1 for the exact values) For the first 20 threads we can observe a linear increase in computation time (See Fig. 6). Shortly before 20 threads the slope increases and gets much steeper after 20 threads. This increase could be caused by multiple things. At first sight, setting the initial condition is an $O(n^2)$ operation which could cause an increase in overhead, however, the timer is only started after initialization, thus is not measured. Apart from that, the overhead from setting up and managing more and more threads could be an explanation for the linear and stronger increase. Also, in some occasion not all threads get the same cpu time, meaning that it might happen that all threads have to wait for some thread to finish late thereby slowing down the entire process. Lets say there is some probability p that this happens to a thread, then the chance that it occurs during one time step, increases with the number of threads. Another reason could be the memory growth as the problem scales. This subsequently forces the process to fetch its elements from more expensive memory.

Threads	1	2	3	4	5	6	7	8	9	10	11	12
Grid Size	209	295	362	418	467	512	553	591	627	660	693	724
Threads	13	14	15	16	17	18	19	20	21	22	23	24
Grid Size	753	782	809	836	861	886	911	934	957	980	1002	1024

Table 1: Grid sizes relative to the thread counts used during the weak scaling study to obtain Fig. 6

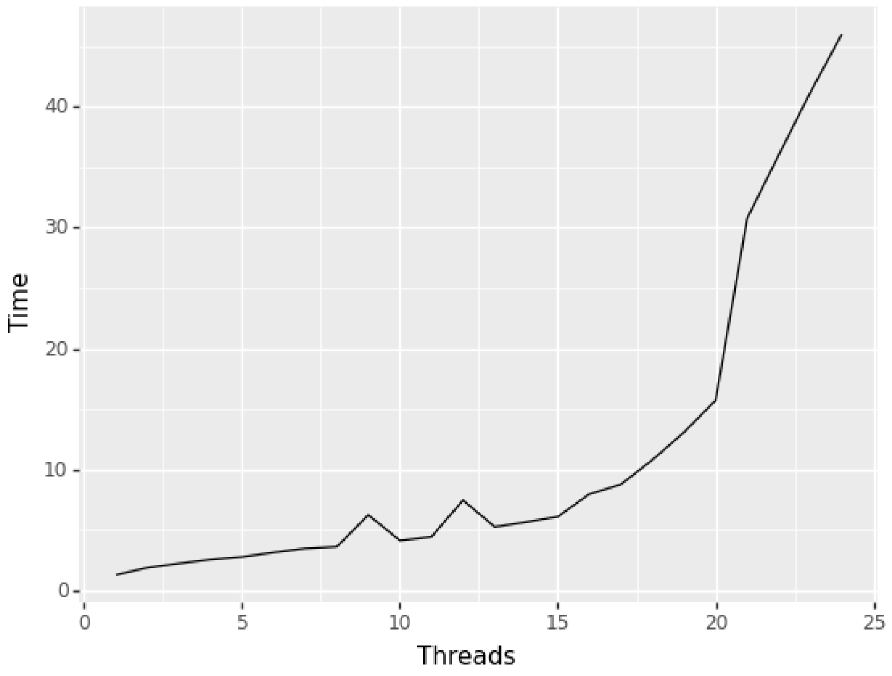


Figure 6: Weak scaling results using 1-24 threads where the grid size was kept relative to the thread count such that the workload per thread remained constant.

3. Bonus Question

After some research I started implementing most of the linalg function with SIMD instructions. After running some benchmarks there was a slight increase in performance, however, not significant. Later I found out that the compiler probably already inserted most of the instructions which explains the only marginal speedup. Therefore, I attempted to vectorize the the stencil kernel. Firstly I tried it for the interior points as the change to the border points would have been easy afterwards. The general idea for SIMD is to perform a single cpu instruction to multiple data points at the same time. In case of the cluster it is possible to perform the same operation on four double at the same time (or 8 float, ...). To ensure that the accessed data is contiguous in the memory it was chosen to vectorize the x dimension (rows). Through the `data()` getter of the `Field` class it was possible to access the pointer to the double array. The entire for loop was implemented using SIMD instructions however the CG failed to converge. I tried to debug it for hours but had no luck. The SIMD instructions are kept as comments.

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

Additional notes and submission details

Submit the source code files (together with your used `Makefile`) in an archive file (tar, zip, etc.) and summarize your results and the observations for all exercises by writing an extended Latex report. Use the Latex template from the webpage and upload the Latex summary as a PDF to iCorsi.

- Your submission should be a gzipped tar archive, formatted like project_number_lastname_firstname.zip or project_number_lastname_firstname.tgz. It should contain:
  - all the source codes of your OpenMP solutions.
  - your write-up with your name project_number_lastname_firstname.pdf,

- Submit your .tgz through Icorsi.