
Solution for Project 3

HPC Lab — 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 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]

In this task we have to finish the implementation of the BLAS level 1 kernels and complete the stencil operator for the discretization of the nonlinear PDE.

For the functions in `linalg.cpp` the implementation was straightforward following the comments provided in the code. In the `operators.cpp` file, we were asked to implement the kernel to compute the inner points. I did so by looking at the already provided code for the boundary points and adapting it to the inner points. The final kernel is shown below:

```

1 // the interior grid points
2 for (int j=1; j < jend; j++) {
3     for (int i=1; i < iend; i++) {
4         f(i,j) = -(4. + alpha) * s_new(i,j)
5                 + s_new(i-1,j) + s_new(i+1,j)
6                 + s_new(i,j-1) + s_new(i,j+1)
7                 + alpha * s_old(i,j)
8                 + beta * s_new(i,j) * (1.0 - s_new(i,j));
9
10    }
11 }

```

Listing 1: Stencil kernel for the interior points

After completion of the above steps, We can run the code and plot it using the provided Python script. The result is shown in the following figure:

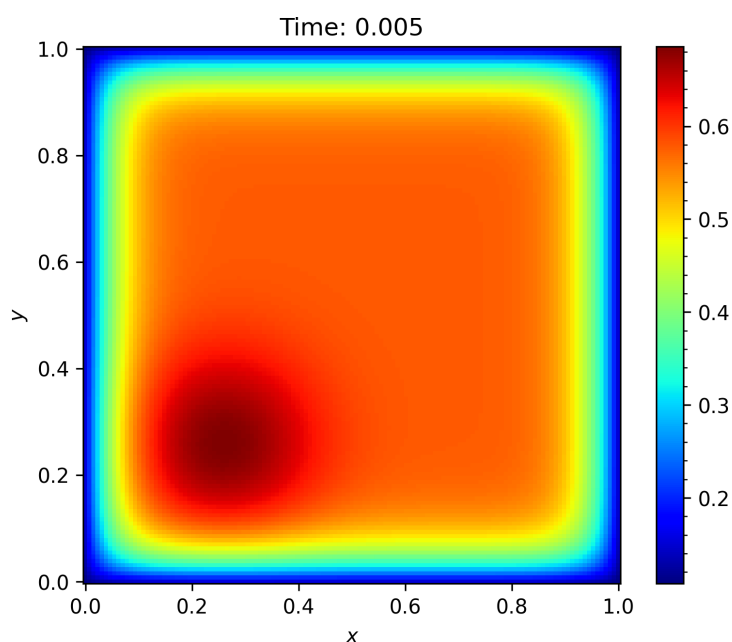


Figure 1: Solution of the nonlinear PDE after completing Task 1.

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

The main goal of this section is to parallelize the mini-app using OpenMP.

2.1. Welcome message in main.cpp and serial compilation

As instructed, I added a welcome message in the `main.cpp` file that is printed at the start of the program execution. This messages indicates the version that we are using together with the number of threads used. The message is as follows:

```

=====
                        Welcome to mini-stencil!
version    :: C++ OpenMP
threads    :: 16
mesh       :: 1024 * 1024 dx = 0.000977517
.....

```

2.2. Parallelization of the Linear algebra kernels

To parallelize the linear algebra kernels in `linalg.cpp`, I used OpenMP directives to distribute the workload across multiple threads. We could see from the function definitions that they all take as input constant values except for what they will write to. Moreover, the computation for each element is independent of the others. We know that `const` variables are shared by default in OpenMP. Therefore, I added the `#pragma omp parallel for` directive before the main loops in each function and I declared as shared the variable that will be modified.

2.3. Parallelization of the stencil kernel in `operators.cpp`

To parallelize the stencil kernel in `operators.cpp`, I applied a similar approach as for the linear algebra kernels, but also adding the `collapse(2)` clause to the OpenMP directive. This can be done because we have two nested loops iterating over the grid points and the inner and outer loops are independent.

The output of the parallelized version is shown below and is visually identical to the serial one, as expected.

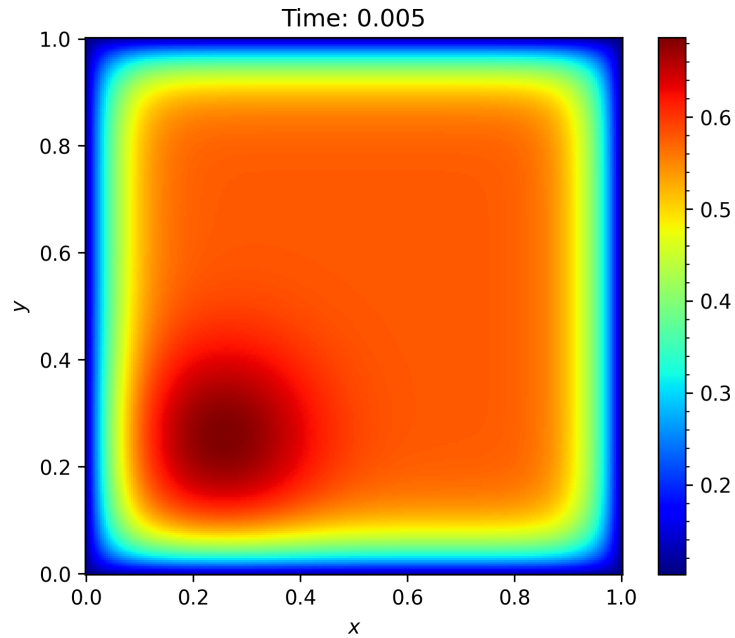


Figure 2: Solution of the nonlinear PDE after completing Task 2 with OpenMP.

2.4. Bitwise identical results

To answer this question, we must first understand that floating point addition and multiplication are not associative operations [1].

This can also be easily observed, not every float number can be represented exactly, this introduces rounding error during computation, and so the order of operations can affect the final result due to the accumulation of these rounding errors.

Then we might try to use static scheduling to ensure that the order of operations is consistent across different runs. However, even this approach won't guarantee bitwise identical results because of the reduction clause used in parallelization. Looking at the details specified in the OpenMP specification for the reduction scoping clauses [2], we see that the location in the OpenMP program at which values are combined and the order in which values are combined are unspecified. Therefore, even with static scheduling, the non-deterministic order of combining values in the reduction operation can lead to different rounding errors and thus different final results.

2.5. Strong scaling

In this section, we analyze the strong scaling of our OpenMP implementation. This means analyze how the execution time improves with the number of threads for a fixed problem size. What we do is keep the problem size constant and measure the execution time while varying the number of threads used.

By using strong scaling we can observe how efficient is our parallel implementation, and determine the optimal number of threads to use for a given problem size.

In an ideal scenario, we expect the execution time to halve as we double the number of threads. However, this is not always the case due to overheads associated with thread management, load balance between threads and bandwidth limitations.

In my execution, we can see that the as the size of the image increases, adding more threads scales better. This is because larger problems have more work to distribute among the threads, which helps to amortize the overhead of managing multiple threads. However, we can also see that after a certain number of threads, the performance improvement starts to diminish. This is likely due to the reasons stated before.

I plotted the results obtained from the strong scaling analysis in the following graphs:

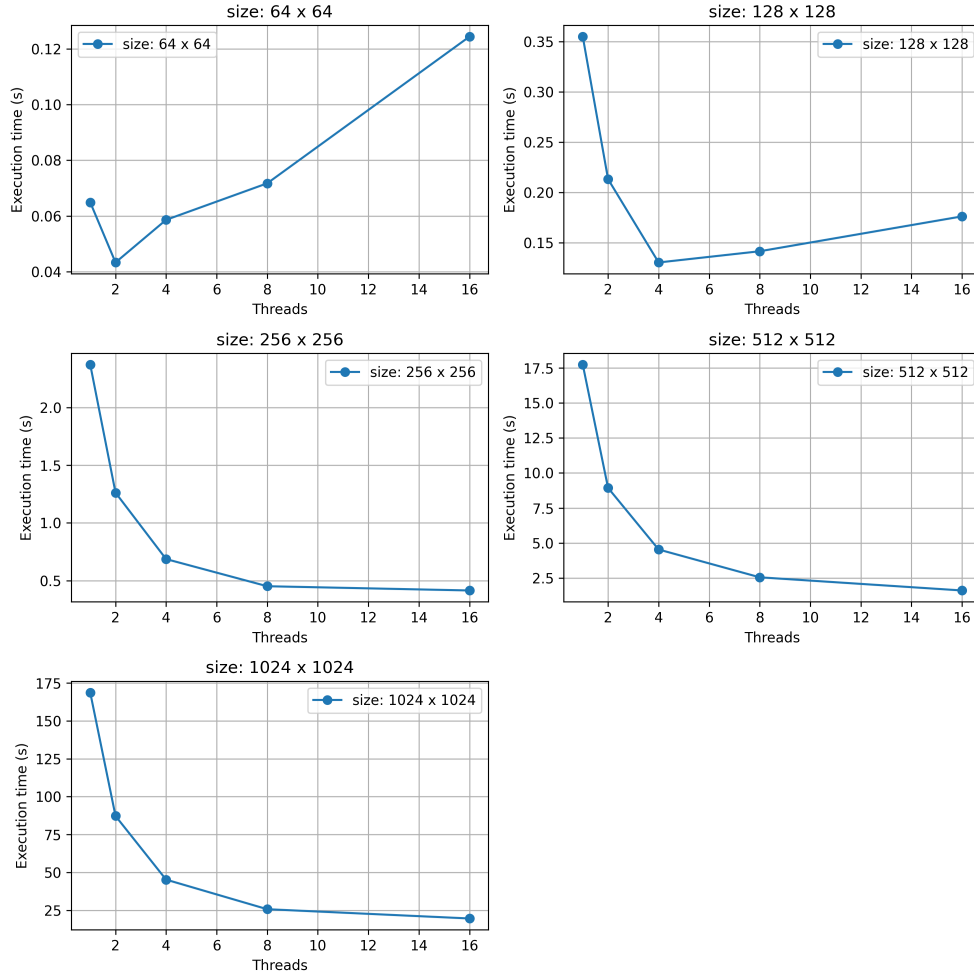


Figure 3: Strong scaling of the OpenMP implementation.

2.6. Weak scaling

In weak scaling, we analyze how the execution time changes as we increase both the problem size and the number of threads proportionally. The goal is to keep the workload per thread constant and observe how well the implementation handles larger problems.

In an ideal weak scaling scenario, we expect the execution time to remain constant as we increase both the problem size and the number of threads. However, adding more threads introduces parallelization overhead and memory access becomes more expensive, which can lead to worse performance as more threads are added.

In my execution we can observe execution time that increases constantly as we add more threads, while in an ideal scenario it should remain constant. Lastly I plotted the efficiency of the weak scaling, which is defined as the ratio of the execution time with one thread to the execution time with N threads. We expect the efficiency to be 1 (or 100%) in an ideal scenario, but in practice it decreases as we add more threads due to the overheads mentioned before. I plotted both the execution time and the efficiency obtained from the weak scaling analysis in the following graphs:

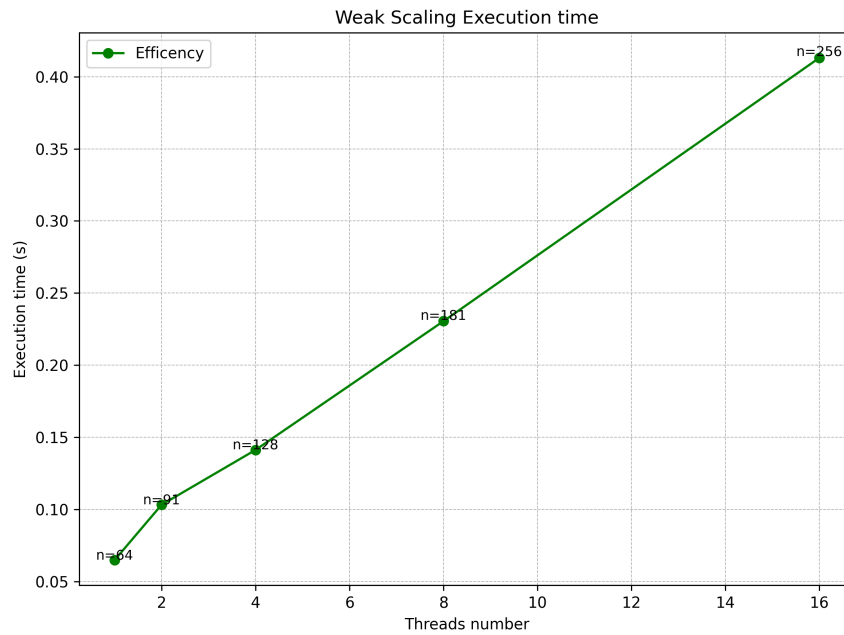


Figure 4: Weak scaling execution time of the OpenMP implementation.

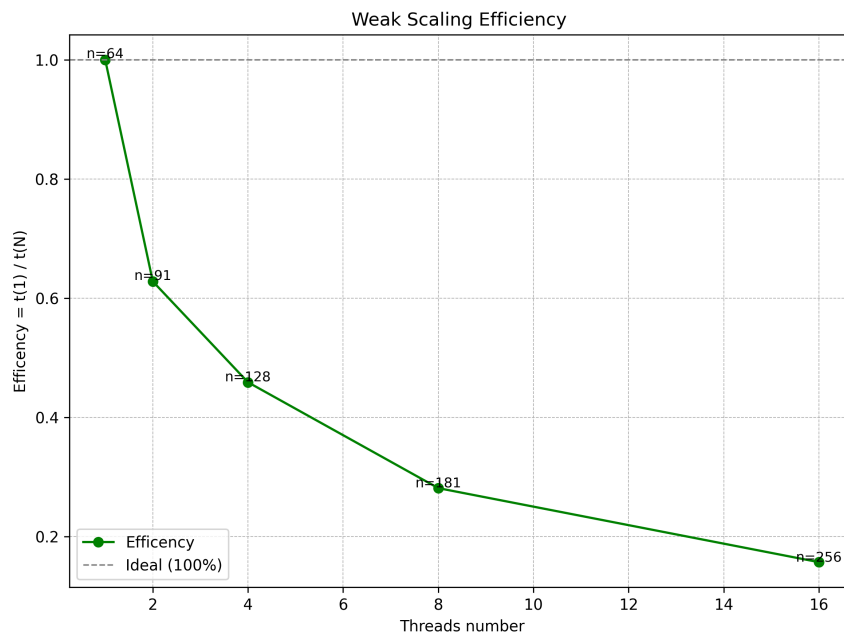


Figure 5: Weak scaling efficiency of the OpenMP implementation.

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

References

- [1] David Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. URL: https://docs.oracle.com/cd/E19957-01/806-3568/ncg_goldberg.html
- [2] OpenMP Architecture Review Board. *OpenMP Application Programming Interface Specification Version 5.2*. Novembre 2021. URL: <https://www.openmp.org/spec-html/5.2/openmpsu50.html> (consultato il 4 novembre 2025).