Università della Svizzera italiana

Institute of Computing CI

High-Performance Computing **2022**

Student: Fabio Marchesi Discussed with: Raffaele Morganti, Gioele De Pianto, Nicholas Carlotti

Solution for Project 3

Due date: 02.11.2022, 23:59

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]

Let's look at the linear algebra functions implemented.

- $||x||^2$.

```
double hpc_norm2(Field const& x, const int N) {
    double result = 0.0;
        for (int i = 0; i < N; i++) {
                result += x[i] * x[i];
        }
    return sqrt(result);
}
```

- $\vec{x} = \{value, value, \ldots, value\}$.

```
void hpc_fill(Field& x, const double value, const int N) {
        for (int i = 0; i < N; i++) {
                x[i] = value;
        }
}
```

- $\vec{y} = \vec{y} + \vec{x} \cdot \alpha$

```
void hpc_axpy(Field& y, const double alpha, Field const& x, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] += x[i] * alpha;
        }
}
```

- $\vec{y} = \vec{x} + \alpha \cdot (\vec{l} - \vec{r})$

```
void hpc_add_scaled_diff(Field& y, Field const& x, const double alpha,
    Field const& l, Field const& r, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] = x[i] + alpha * (l[i] - r[i]);
        }
}
```

- $\vec{y} = \alpha \cdot (\vec{l} - \vec{r})$

```
void hpc_scaled_diff(Field& y, const double alpha,
    Field const& l, Field const& r, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] = alpha * (l[i] - r[i]);
        }
}
```

- $\vec{y} = \alpha \cdot \vec{x}$

```
void hpc_scale(Field& y, const double alpha, Field const& x, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] = alpha * x[i];
        }
}
```

- $\vec{y} = \alpha \cdot \vec{x} + \beta \cdot \vec{z}$

```
void hpc_lcomb(Field& y, const double alpha, Field const& x, const double beta,
    Field const& z, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] = alpha * x[i] + beta * z[i];
        }
}
```

- $\vec{y} = \vec{x}$

```
void hpc_copy(Field& y, Field const& x, const int N) {
        for (int i = 0; i < N; i++) {
                y[i] = x[i];
        }
}
```

Instead the stencil operators are implemented as follows:

- The interior grid points for the $k-th$ iteration are updated using the following formula: $f_{i,j} = [-(4+\alpha)s_{i,j} + s_{i-1,j} + s_{i+1,j} + s_{i,j-1} + s_{i,j+1} + \beta s_{i,j}(1 - s_{i,j})]^{k-1} + \alpha s^k_{i,j}$
  The code looks like this:

```
for (int j = 1; j < jend; j++) {
    for (int i = 1; i < iend; i++) {
                f(i, j) = -(4. + alpha) * s(i, j)
                        + s(i - 1, j) + s(i + 1, j) + s(i, j - 1) + s(i, j + 1)
                        + beta * s(i, j) * (1.0 - s(i, j)) +
                        alpha * y_old(i, j);
    }
}
```

- When we update the boundaries points we need to remember that at least one of the adjacent cells of (i, j) (two if (i, j) is one of the corners) isn't store on the operator s, so we need to use the four boundaries $bndW, bndE, bndN, bndS$ which corresponds to, respectively, the west, east, north and south boundaries. Apart from that, the formula remains the same one of the inner points.
  - East boundary:

```
int i = nx - 1;
for (int j = 1; j < jend; j++) {
    f(i, j) = -(4. + alpha) * s(i, j) +
```

```
            s(i - 1, j) + s(i, j - 1) + s(i, j + 1) +
            alpha * y_old(i, j) + bndE[j] +
            beta * s(i, j) * (1.0 - s(i, j));
    }
```

– West boundary:

```
    int i = 0;
    for (int j = 1; j < jend; j++) {
        f(i, j) = -(4. + alpha) * s(i, j) +
            s(i + 1, j) + s(i, j + 1) + s(i, j - 1) +
            alpha * y_old(i, j) + bndW[j] +
            beta * s(i, j) * (1.0 - s(i, j));
            }
```

– North boundary:

```
    int j = nx - 1;
        for (int i = 1; i < iend; i++) {
        f(i, j) = -(4. + alpha) * s(i, j) +
            s(i + 1, j) + s(i - 1, j) + s(i, j - 1) +
            alpha * y_old(i, j) + bndN[i] +
            beta * s(i, j) * (1.0 - s(i, j));
            }
```

– South boundary:

```
    int j = 0;
    for (int i = 1; i < iend; i++) {
        f(i, j) = -(4. + alpha) * s(i, j) +
            s(i - 1, j) + s(i + 1, j) + s(i, j + 1) +
            alpha * y_old(i, j) + bndS[i] +
            beta * s(i, j) * (1.0 - s(i, j));
            }
```

- The last cases that we need to cover are the four corners:
  – North-West Corner:

```
int j = nx - 1, i = 0;
f(i, j) = -(4. + alpha) * s(i, j) +
    s(i + 1, j) + s(i, j - 1) +
    alpha * y_old(i, j) + bndW[j] + bndN[i] +
    beta * s(i, j) * (1.0 - s(i, j));
```

  – North-East corner:

```
int j = nx - 1, i = nx - 1;
f(i, j) = -(4. + alpha) * s(i, j) +
    s(i - 1, j) + s(i, j - 1) +
    alpha * y_old(i, j) + bndE[j] + bndN[i] +
    beta * s(i, j) * (1.0 - s(i, j));
```

  – South-West corner:

```
int j = 0, i = 0;
f(i, j) = -(4. + alpha) * s(i, j) +
    s(i + 1, j) + s(i, j + 1) +
    alpha * y_old(i, j) + bndW[j] + bndS[i] +
    beta * s(i, j) * (1.0 - s(i, j));
```

  – South-East corner:

```
int j = 0, i = nx - 1;
f(i, j) = -(4. + alpha) * s(i, j) +
    s(i - 1, j) + s(i, j + 1) +
    alpha * y_old(i, j) + bndE[j] + bndS[i] +
    beta * s(i, j) * (1.0 - s(i, j));
```

The final program allows us to run a simulation of the reaction-diffusion equation where the initial condition s^{init} are set to 0.1 over the entire domain, except for a circular region in the bottom left corner that is initialized to the value 0.2. the domain Ω is discretized using a uniform grid with $(n+2) \times (n+2)$ points, where a grid point is indicated by $x_{i,j}$, for $i, j \in \{0, 1, \ldots, n+1\}$, and where $s^k_{i,j}$ is the approximation of s at the grid-point $x_{i,j}$ at time-step k. The mini-application has three parameters: the grid's side size, the number of iterations used to simulate the process and the final time. Increasing the number of iterations will lead to a more accurate simulation, while increasing the end time will allow the "heat" to spread more.

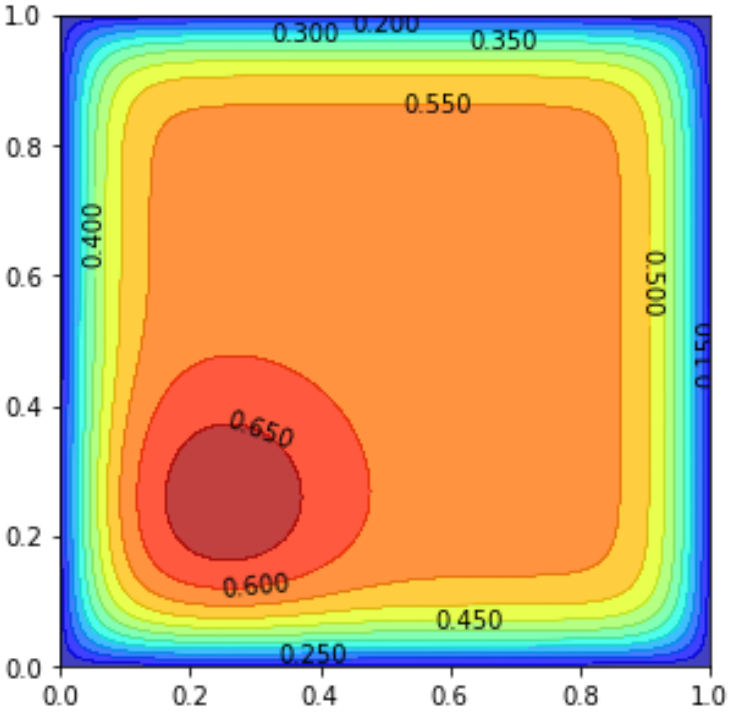


Figure 1: This is the result of the simulation on a grid of size 256x256, with 100 iterations and final time equals to 0.005

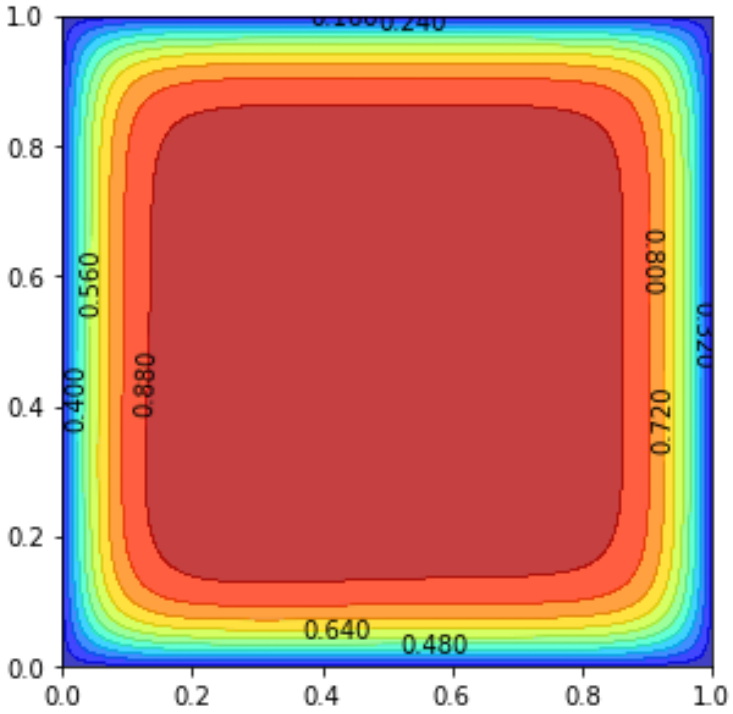


Figure 2: This is the result of the simulation on a grid of size 256x256, with 100 iterations and final time equals to 0.01

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

Looking at the code proposed before, we can see how there are many aspects that can be parallelized. For example, all the functions that we wrote in **linalg.cpp** contain a for that can be parallelized using OpenMP. In order to do that, the functions have been modified as follows.

- $\vec{x}\dot{\vec{y}}$. Here the for-loop is splitted among all the threads and we use **reduction** to update `result` correctly.

```
double hpc_dot(Field const& x, Field const& y, const int N) {
    double result = 0.0;
    #pragma omp parallel for reduction (+ : result)
    for (int i = 0; i < N; i++) {
            result += x[i] * y[i];
    }
    return result;
}
```

- $||x||^2$. This function is paralellized in a similar way to the previous one.

```
double hpc_norm2(Field const& x, const int N) {
    double result = 0.0;
        #pragma omp parallel for reduction (+ : result)
        for (int i = 0; i < N; i++) {
                result += x[i] * x[i];
        }
    return sqrt(result);
}
```

- $\vec{x} = \{value, value, \ldots, value\}$.

```
void hpc_fill(Field& x, const double value, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                x[i] = value;
        }
}
```

- $\vec{y} = \vec{y} + \vec{x} \cdot \alpha$

```
void hpc_axpy(Field& y, const double alpha, Field const& x, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] += x[i] * alpha;
        }
}
```

- $\vec{y} = \vec{x} + \alpha \cdot (\vec{l} - \vec{r})$

```
void hpc_add_scaled_diff(Field& y, Field const& x, const double alpha,
    Field const& l, Field const& r, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] = x[i] + alpha * (l[i] - r[i]);
        }
}
```

- $\vec{y} = \alpha \cdot (\vec{l} - \vec{r})$

```
void hpc_scaled_diff(Field& y, const double alpha,
    Field const& l, Field const& r, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] = alpha * (l[i] - r[i]);
        }
}
```

- $\vec{y} = \alpha \cdot \vec{x}$

```
void hpc_scale(Field& y, const double alpha, Field const& x, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] = alpha * x[i];
        }
}
```

- $\vec{y} = \alpha \cdot \vec{x} + \beta \cdot \vec{z}$

```
void hpc_lcomb(Field& y, const double alpha, Field const& x, const double beta,
    Field const& z, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] = alpha * x[i] + beta * z[i];
        }
}
```

- $\vec{y} = \vec{x}$

```
void hpc_copy(Field& y, Field const& x, const int N) {
        #pragma omp parallel for
        for (int i = 0; i < N; i++) {
                y[i] = x[i];
        }
}
```

A part from the first two functions where we had to use a reduction all the other functions are parallelized in the same way using a simple **omp parallel for**. This solution works well because the default way that that directive uses to split the workload among the threads is static. This means that the iterations will be split in chucks of the same size ($N/\#threads$) and will be assigned to each thread. We can split the loops in this way because there aren't any loop carried dependencies, and it's fast because the operations inside the loop are simple, so, we expect each thread to use, roughly, the same amount of time. To parallelize the code inside `operators.cpp` the following changes has been made.

- The interior grid points for the $k-th$ iteration are updated using two nested loops. In order to use multiple threads we exploit the fact that all the computations inside the loops are independent, so we don't need to worry about the way that iterations are assigned to the threads. The simplest way to use more threads is to use the directive **pragma omp parallel for collapse(2)**, in this way each iteration (that is, each pair (i, j)) will be assigned to the first free thread.

```
#pragma omp parallel for collapse(2)
for (int j = 1; j < jend; j++) {
    for (int i = 1; i < iend; i++) {
        f(i, j) = -(4. + alpha) * s(i, j)
            + s(i - 1, j) + s(i + 1, j) + s(i, j - 1) + s(i, j + 1)
            + beta * s(i, j) * (1.0 - s(i, j)) +
            alpha * y_old(i, j);
    }
}
```

- When we update the boundaries we have only one loop, so we can simply use **omp parallel for** to parallelize it.
  - East boundary:

```
int i = nx - 1;
#pragma omp parallel for
for (int j = 1; j < jend; j++) {
    f(i, j) = -(4. + alpha) * s(i, j) +
        s(i - 1, j) + s(i, j - 1) + s(i, j + 1) +
        alpha * y_old(i, j) + bndE[j] +
        beta * s(i, j) * (1.0 - s(i, j));
}
```

  - West boundary:

```
int i = 0;
#pragma omp parallel for
for (int j = 1; j < jend; j++) {
    f(i, j) = -(4. + alpha) * s(i, j) +
        s(i + 1, j) + s(i, j + 1) + s(i, j - 1) +
        alpha * y_old(i, j) + bndW[j] +
        beta * s(i, j) * (1.0 - s(i, j));
        }
```

  - North boundary:

```
int j = nx - 1;
#pragma omp parallel for
    for (int i = 1; i < iend; i++) {
    f(i, j) = -(4. + alpha) * s(i, j) +
        s(i + 1, j) + s(i - 1, j) + s(i, j - 1) +
        alpha * y_old(i, j) + bndN[i] +
        beta * s(i, j) * (1.0 - s(i, j));
        }
```

– South boundary:

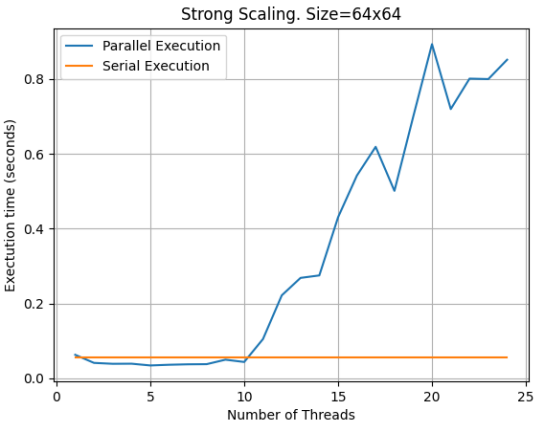
```
int j = 0;
#pragma omp parallel for
for (int i = 1; i < iend; i++) {
    f(i, j) = -(4. + alpha) * s(i, j) +
        s(i - 1, j) + s(i + 1, j) + s(i, j + 1) +
        alpha * y_old(i, j) + bndS[i] +
        beta * s(i, j) * (1.0 - s(i, j));
        }
```

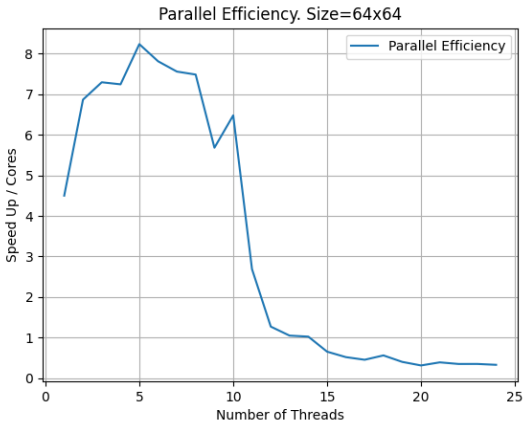
The computation of the corner is done in $O(1)$, so it doesn't make sense to parallelize it. In regard to the question "The nested for loop and the inner grid points might be obvious targets. What role do the boundary loops play?", after a bit of testing, I found out that make the boundary computation parallel doesn't make a big difference in terms of execution time. The reason behind this result is that the number of boundary points is $O(N)$ while the number of inner grid points is $O(N^2)$, so the latter plays a much bigger factor in terms of execution time.

2.1. Strong Scaling

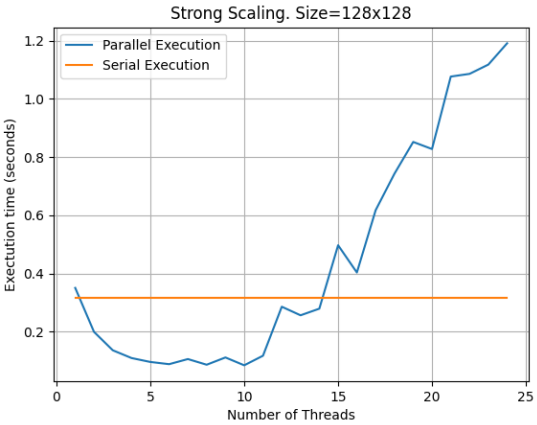
We measure the strong scaling of our parallel implementation, this means that we look at the time required to do a simulation on a fixed grid size with an increasing number of threads. We analyzed the results with size sxs and t threads, where $s \in \{64, 128, 256, 512, 1024\}$ and $t \in \{1, 2, \ldots, 24\}$. For each pair (s, t) we plotted the time required and also the parallel efficiency. The parallel efficiency can be used to estimate how much of the computing capacity is actually used to carry out a calculation and is computed as $\frac{Speed\_up}{Number\_of\_cores}$ (the number of cores on each node of the cluster is 20). All the results reported are computed with 100 iterations and end time equals to 0.005. A common consideration for all the plots is that when the number of threads is greater than 20 the performance get worse, this is because the number of cores is equals to 20.

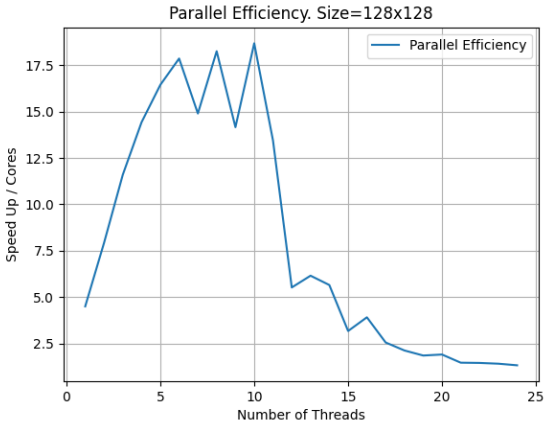
- $64x64$: The grid here is quite small, and we can see that if the number of threads is greater or equal than 10 then the added overhead is bigger than the improvement of the parallelization. This is also observable looking at the parallel efficiency graph, indeed with $t > 10$ the parallel efficiency is less than one. So if we need to do a simulation of this size, is fine to use the serial version.



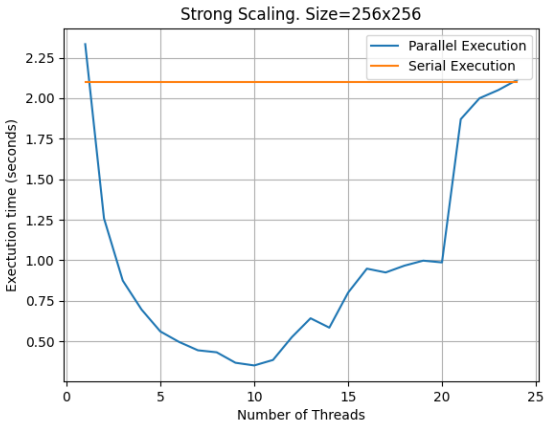


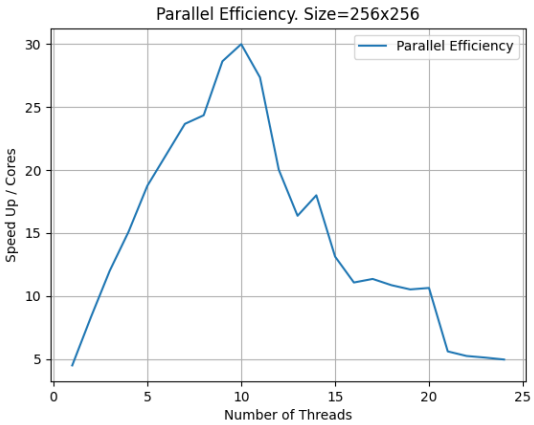
- $128x128$: Here the grid is a little bit bigger than before and the time improvement using the parallel version with the "correct" number of threads is much bigger than before. In this case 15 seems to be the maximum number of threads for which the parallel version results more efficient than the serial one. Also, the parallel efficiency is better than the one we had in the grid of size 64x64, with a peak of 18



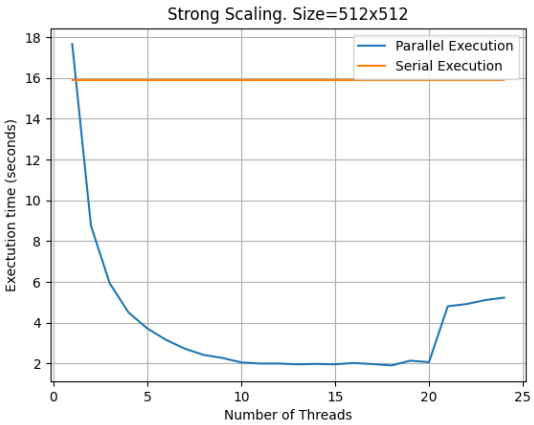


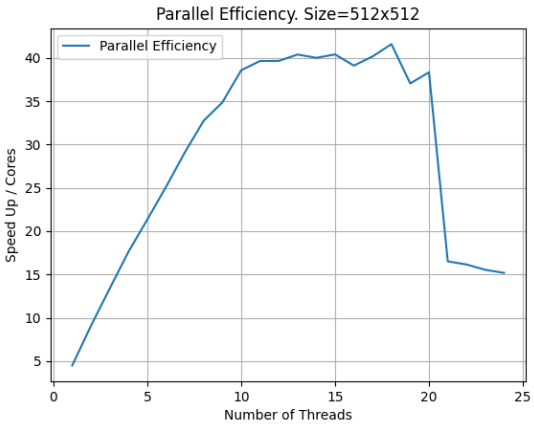
- $256x256$: Here the matrix starts to get quite big, indeed the serial version takes more than two seconds to compute the simulation. Here the parallel version is faster than the serial one for each possible number of threads except one. With the parallel version using one thread (this consideration is independent of the grid size) we have the same computational power of the serial version but with the added overhead of the thread's management, so the computation is slower than the serial one. Also, the parallel efficiency is getting better, reaching 30% with ten threads.



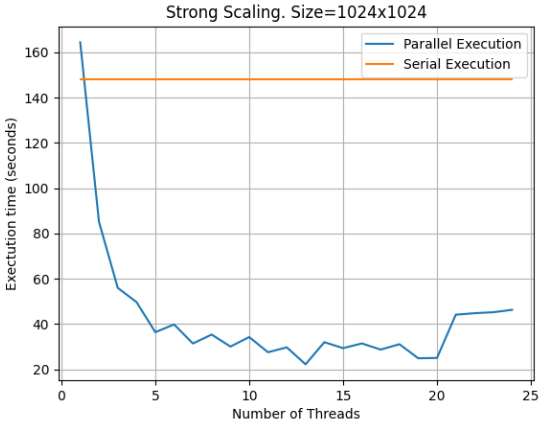


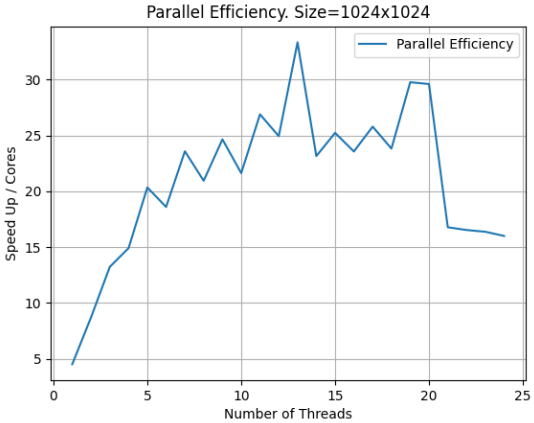
- $512x512$ Here we start to see the power of parallelization, the serial version takes 16 seconds to compute while the parallel version with 10-20 threads takes only two seconds, reaching a parallel efficiency of more than 40%. Like said before, if the number of threads is bigger than 20 then the performance starts to get worse only because the number of cores on the cluster's nodes is 20.





- $1024x1024$ With such a big matrix, the improvement of the parallelization is even bigger. We go from 2 and half minutes of the serial version to less than 30 seconds with 13 threads. The parallel efficiency is a little bit worse than the one we got on the 512x512 matrix but is still pretty high, reaching almost 35





We can conclude that our mini-application has a good strong scaling, especially if the matrix is

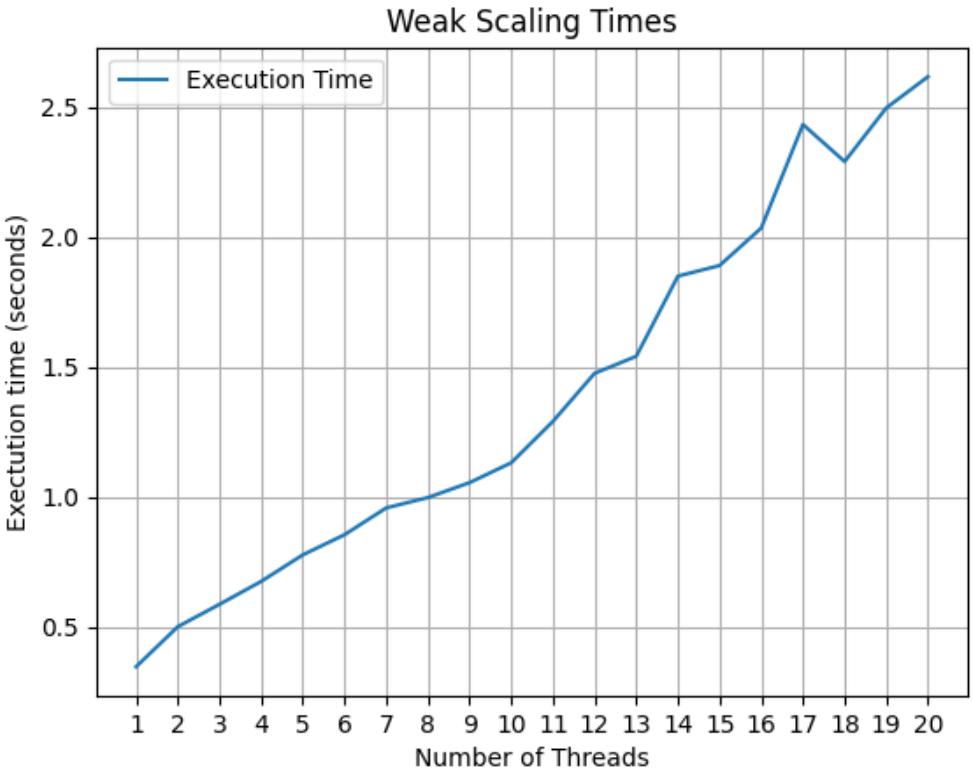
big. Note that with the added parallelization, the simulations don't produce bitwise identical results to the serial version. This is because `hpc_dot` and `hpc_norm2` compute `result` without any guarantee about the order of the iterations, and when working with floating point values we have to do approximations and the order of operations matters for the final result. All the other parallelized functions don't have this problem thought, because for all the other functions in **linalg.cpp** the iteration order doesn't matter. Also, for the functions in **operators.cpp** we have that each iteration works with values of the previous iterations and also here the iteration order does't affect the results. So if we would want to have bitwise identical results to the serial version we could avoid parallelizing those two functions, this would make the performance a little bit slower. In general, the generated difference is minimal.

2.2. Weak scaling

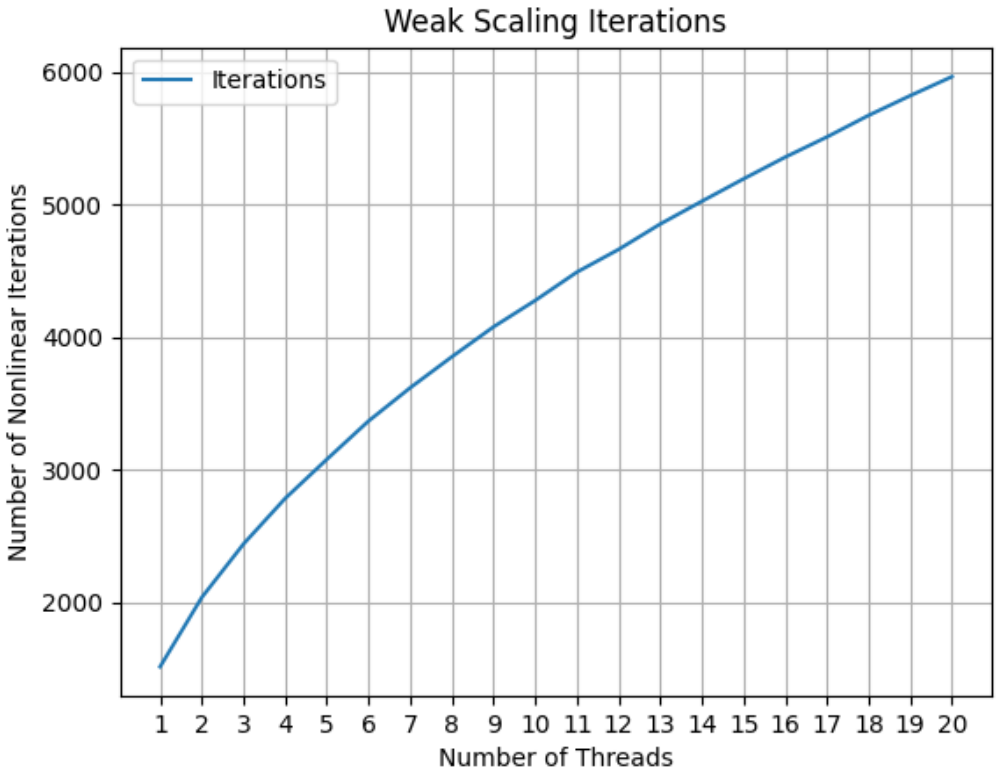
In order to measure how our code with weak scaling, we need to run simulations with different number of threads and different matrix sizes in a way that the workload for each core remains the same. In order to do that, we used the following values.

Number of Threads	Side of the matrix	Cells for each core
1	128	16384.0
2	181	16380.5
3	222	16428.0
4	256	16384.0
5	286	16359.2
6	314	16432.67
7	339	16417.29
8	362	16380.5
9	384	16384.0
10	405	16402.5
11	425	16420.45
12	443	16354.08
13	462	16418.77
14	479	16388.64
15	496	16401.07
16	512	16384.0
17	528	16399.06
18	543	16380.5
19	558	16387.58
20	572	16359.2

The grid processed by our simulation is a square, so it isn't possible to have the exact number of cells for each core for each possible number of threads, but the approximations lead to really similar values.



We can see that the application doesn't seem to weak scale well, indeed, if we look at the execution time with 20 threads is 8 times higher than the one with one thread, even though the workload for each core is the same, let's try to understand why. For each of these simulations, the number of step is the same (100) but when we have more threads the matrix is much bigger. For each step during the simulation, we have to solve a system of equations $Ax = b$ and we solve this system using an iterative method. So, it's useful to check whether the number of iterations required to solve each system is the same or not. This is the plot of this analysis.



As we can see, there seems to be a strong correlation between the size of the matrix and the number of iterations to solve the system. This is probably due to the fact that when we use an iterative method, we stop when the approximate solution's error is smaller than a certain tolerance. We compute that error as the norm of the difference between 2 vectors (the solution and the approximate solution), and with bigger vectors we have and higher norm and the tolerance is independent of the vector's size. So we can say that the slowdown on the weak scaling graph is partially caused by the different number of iterations on the conjugate gradient method. But still we can see that the time with 20 threads is worse than the time with one thread by a factor of 8 while the number of iterations of the conjugate gradient method with 20 threads is higher than the number of iterations with one thread by a factor of 4. This means that remains a factor of 2 on the slowdown when we analyze the weak scaling. The conclusion is that the mini-application doesn't weak scale well.

2.3. SIMD instructions

The main idea behind the SIMD (Single Instruction, Multiple Data) concept is that if we need to perform the same operation on multiple data, we can do it concurrently. The idea is that instead of only applying the operation on simply two values, we can apply it to two vector of operands. This is possible because the computational power of the CPU at each cycle is often much bigger than the simple operations that we have to perform. So, I also tried to use SIMD instructions to improve the performance, but I didn't get any real improvement. I tried to use the directive **omp simd** on the loops in `linalg` and I got results that are better than the serial version for grid side equals to $64, 128, 256, 512, 1024$ and 100 iterations. But the parallel version with a reasonable number of threads is much faster. So I tried to combine the two concept using **omp parallel for simd**, here there is a visualization of what it actually does.

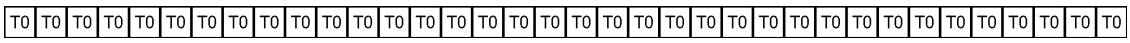


Figure 3: Without using any directive, the main thread will carry out all the iterations

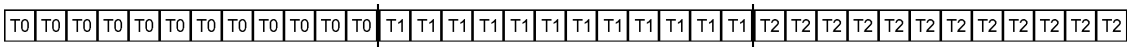


Figure 4: The first concept that is applied is the **parallel for**. In this way, the iterations are split among threads (three in the example)

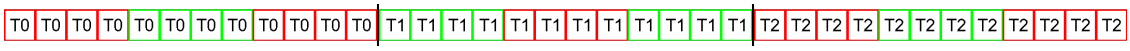


Figure 5: In the end, there is the effect of **simd**, this concept applies to each thread and basically groups continuous iterations in small chunks that will be executed at the same time. The new SIMD modifier automatically adjusts the chunk size to match it with the length of the SIMD register

But also in this case, if I had a significant number of threads, the parallel version that doesn't use **SIMD** perform better than the one tha uses **SIMD**. I tried also to use them on the linear algebra operations or only on the inner grid points computation, but it doesn't seem to work well with a big number of threads.

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.