Parallelization of the Jacobi's Algorithm

Roberto Esposito - 636672 - r.esposito8@studenti.unipi.it - MSc. in Artificial Intelligence
Parallel and Distributed Systems - 9 CFU - A.Y. 2021/2022

1 Introduction

The following report aims to develop a parallel version of the Jacobi method that is used to solve systems of linear equations $Ax = b$ where $A \in \mathbb{R}^{n \times n}$ is a strictly diagonally dominant square matrix, $b \in \mathbb{R}^n$ is the vector of known terms and $x \in \mathbb{R}^n$ is the vector of the variables.
The Jacobi's Algorithm is an iterative method that at each step computes a more accurate approximation of the solution vector with respect to the previous step.
The formula to update the i-th element of the vector x at step k is the following:

$$x_i^{(k)} = \frac{1}{a_{ii}} \left[\sum_{\substack{j=1, \\ j \neq i}}^{n} (-a_{ij} \cdot x_j^{(k-1)}) + b_i \right]$$

At the step 0 the solution vector is initialized with all zeros.
The stopping criteria can be chosen arbitrarily and in this case I considered two ways to stop the execution of the algorithm: fixing a maximum number of iterations (K) or computing the norm of the difference between the vectors at steps (k) and $(k-1)$ normalized with the norm at the step (k) $\left(\frac{||x^{(k)} - x^{(k-1)}||}{||x^{(k)}||} \right)$.

The process to develop the following project can be divided mainly into three different phases:

- **Design Phase**: it is analyzed from a theoretical point of view the algorithm and then it is designed a possible parallel implementation of it;
- **Implementation**: are described the implementation details of the parallel versions produced;
- **Experiments**: are executed different versions of the model with different parameters in order to collect all the results and make a comparison between them.

2 Design Phase

In this section we are going to talk about the sequential version of the Jacobi's Algorithm, reported in Algorithm 1, such that we can analyze its features in order to design the parallel version.

Algorithm 1 Jacobi Algorithm

```
Require: $A \in \mathbb{R}^{n \times n},\ b \in \mathbb{R}^n,\ K \in \mathbb{N}^*,\ \text{tolerance} \in \mathbb{R}$
Ensure: $x$ s.t. $Ax = b$
1:  for $k \leftarrow 0$ to $< K$ do
2:      for $i \leftarrow 0$ to $< n$ do
3:          $sum \leftarrow 0$
4:          for $j \leftarrow 0$ to $< n$ do
5:              if $i \neq j$ then
6:                  $sum \leftarrow sum + A_{i,j} \cdot x_j^{(k-1)}$
7:              end if
8:          end for
9:          $x_i^{(k)} = \frac{(b_i - sum)}{A_{i,i}}$
10:     end for
11:     if $\frac{||x^{(k)} - x^{(k-1)}||}{||x^{(k)}||} < tolerance$ then
12:         break
13:     end if
14: end for
```

The algorithm is characterized by three cycles: the first one (lines $1 - 14$) allows us to go from iteration k to iteration $k+1$ and this must be executed sequentially because the vector $x^{(k+1)}$ is computed using $x^{(k)}$ so if we parallelize the cycle the results would not be consistent; the second loop (lines $2 - 10$) can be computed in parallel splitting the computation of the vector $x^{(k)}$ between different number of workers, but we must ensure that before we pass to the iteration $(k+1)$ the whole vector x at iteration k is computed; the last cycle (lines $4 - 8$) can be executed in parallel too, but it makes sense to use multiple workers only if the size of the linear system (n) is very large.

The two loops (lines $2 - 10$ and lines $4 - 8$) correspond respectively to the map parallel pattern and reduce parallel pattern which are both data parallel patterns. Furthermore even the computation of the stopping criteria could be done in parallel since it consists of computing two norms which are mainly composed by squares and sums.

In order to do some considerations about the performance of the algorithm we will consider some metrics as speedup, scalability and efficiency. To compute those metrics two ingredients are needed: $T_{\text{seq}}(n)$ (the sequential time with a linear system of dimension n)

and $T_{\text{par}(n,nw)}$ (the parallel time with nw workers for a linear system of dimension n).
Let's see the sequential time:

$$T_{\text{seq}}(n) = T_{\text{init}} + K \cdot (n \cdot T_{\text{iter}} + T_{\text{stop}})$$

where T_{init} is the time needed to initialize the variables, K is the number of iterations, T_{iter} is the time required to compute a single element i of the vector x at step k ($x_i^{(k)}$) and T_{stop} is the time to compute the stopping criteria.
Instead the parallel time with nw workers is:

$$T_{\text{par}}(n, nw) = T_{\text{init}} + T_{\text{split}} + K \cdot \left(\frac{n}{nw} \cdot T_{\text{iter}} + T_{\text{stop}}\right) + T_{\text{merge}}$$

where T_{split} and T_{merge} is the overhead to split and merge the work among the nw workers. Notice that the number of iterations K multiplies only T_{iter} because are created only nw workers and then we can use a synchronization method to force all the threads to wait the end of the others. This is better rather than `fork` and `join` at each step nw workers.

3 Implementation

In the following section it is described in details the implementation of the Jacobi's Algorithm, I have built three different versions:

- Sequential: it is the sequential implementation of the pseudo-code reported in Algorithm 1;
- Native Threads: it is a parallel implementation that uses the native `thread` library;
- FastFlow: it is a `C++` API that allows to use high-level parallel patterns simplifying the development of parallel application ([1]).

3.1 Preliminaries

First of all if we want to execute the algorithm we need to generate the linear system and in order to do that we need to generate the matrix A and the vector b. It is done thanks to the functions `generate_matrix` and `generate_vector` that require four parameters: the size of the linear system, the seed to generate random numbers, the minimum and the maximum values; those functions are implemented in the file `utility.cpp`.

3.2 Native Threads Implementation

At this point we should describe in more details the implementation that uses the native threads and it is done using the library `thread`. As we said in the second section of this

report the second cycle can be parallelized using a map parallel pattern.
The implementation of this version is done in `jacobi_threads.cpp` and it consists of creating different workers (`num_threads`) and for each of them it is assigned a lambda expression (`body`). The whole system is split into chunks such that each thread computes the assigned section of the vector x. This step is not so trivial because since we are computing K iterations we need that each thread waits that all the others have finished the step k before they pass to step $(k + 1)$. The first idea could be to fork and join the threads at each iteration, but the overhead of this approach is huge; a much better idea is to use some synchronization methods to force all the threads to wait the termination of a specific step. This is done using the `barrier`, it allows to solve the problem described above just adding the function `arrive_and_wait` at the end of each iteration.

3.3 FastFlow Implementation

In order to parallelize the Jacobi's Algorithm with the library `FastFlow` I used the class `ff::ParallelFor` that requires a lambda function corresponding to the iterations of the associated cycle that should be parallelized; in other words it is the function executed by the workers. The implementation of the FastFlow version is done in `jacobi_ff.cpp`. In our case the lambda function corresponds to the execution of the most inner loop (the third one) in which is computed a single element i of the vector x at step k, $(x_i^{(k)})$.

4 Experiments

In the following section are analyzed the results obtained running the Jacobi's Algorithm with different parameters. All the tests have been executed on the Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz with 32 cores.
The experiments were performed using the three modalities: sequential, native threads and FastFlow with the same parameters and then they have been compared.
The parameters considered to perform the tests are the ones reported in the Table 1, the number of workers starts using just 1 thread. After that the algorithm is executed with 2 threads then for the others run the number is updated in the following way: `num_threads += 2`.

The experiments have been repeated using 5 runs and then it is computed the average time to compensate any outliers if present. Furthermore all the tests have been executed without including the tolerance since its computation did not affect the completion time of the algorithm and in this way it is ensured that all the trials are equal because all of them finish with the same number of iterations.
At compilation time the files have been compiled using the flag $-O3$ in order to optimize where possible the code such that the execution time could be the best possible; at the

Parameters	**Values**
Linear System size	1.000 - 5.000 - 15.000
Number of iterations	100
Number of threads (for parallel versions)	1 - [2 - 32]

Table 1: Values considered for the different parameters

following Table 2 are reported the differences in time using the $-O3$ flag and without it for the sequential implementation.

Size (n)	**n=1000**	**n=5000**	**n=15000**
Time with $-O3$ flag (mu sec)	157.644	4.207.560	36.605.800
Time without $-O3$ flag (mu sec)	172.179	4.403.680	36.804.400

Table 2: Differences with the usage of the $-O3$ flag and without it.

In the Table 3 reported below it is computed a rough approximation of the overhead. These values are obtained doing `fork` and `join` of several threads and it is added a `barrier` to simulate additional overhead, but as I said before this is just an approximation.

Threads	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**
Average Time (mu sec)	125	160	258	431	536	528	683	779	870

Threads	**20**	**22**	**24**	**26**	**28**	**30**	**32**
Average Time (mu sec)	1121	1357	1517	1777	1832	1922	2401

Table 3: Overhead

At this point it is useful to see the plots obtained executing the algorithm with different parameters. In the following Table 4 are reported the results of the Jacobi's Algorithm using the sequential version and the plots of the parallel versions are the following ones Figure 1, 2, 3.

Sequential Time	**Time (mu sec)**
n=1000	155322
n=5000	3.98e+06
n=15000	3.59e+07

Table 4: Sequential Time with K=100.

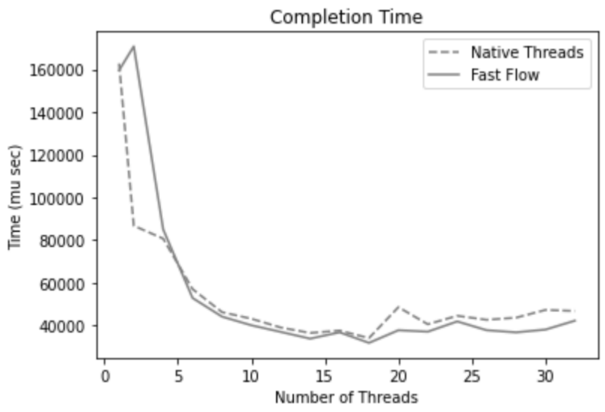

Figure 1: n=1000, K=100.

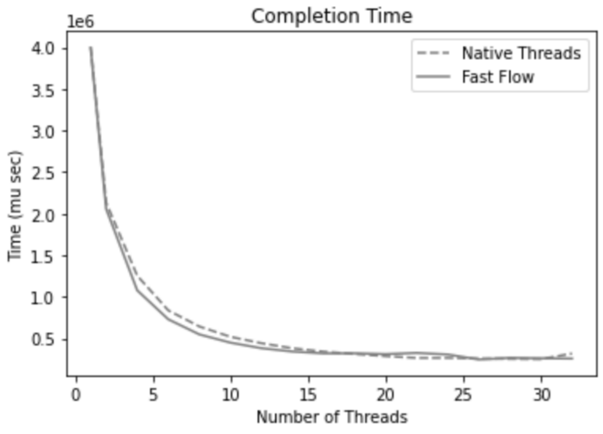

Figure 2: n=5000, K=100.

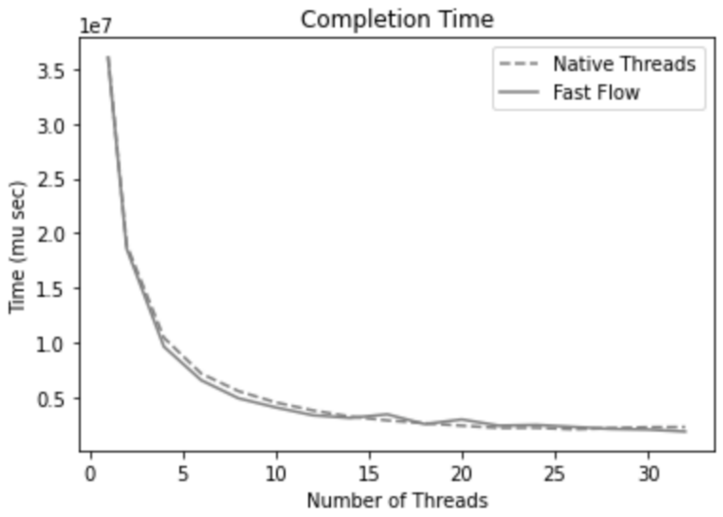

Figure 3: n=15000, K=100.

From this point on are reported the results obtained considering different metrics as speedup, scalability and efficiency and the tests are executed changing the size of the linear system (1.000, 5.000, 15.000).

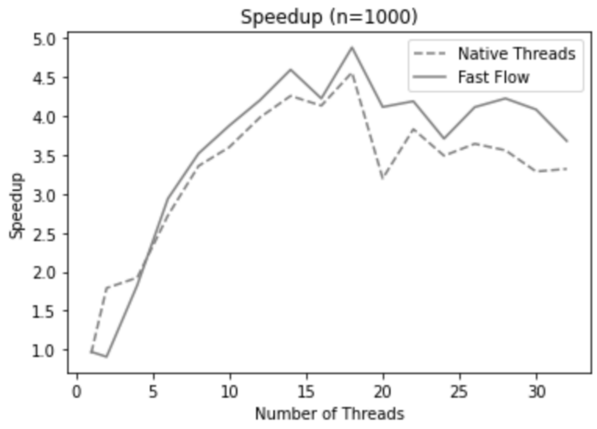

Figure 4: n=1000, K=100.

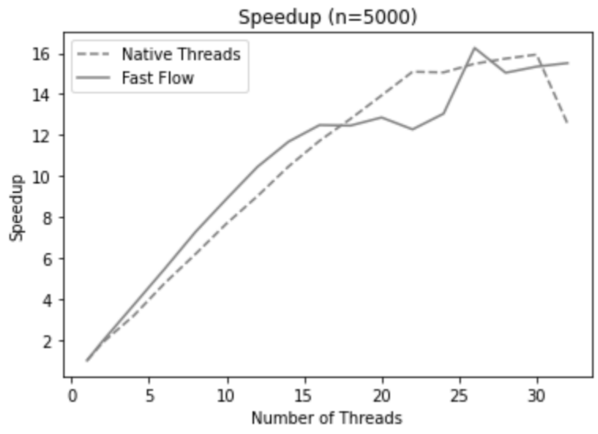

Figure 5: n=5000, K=100.

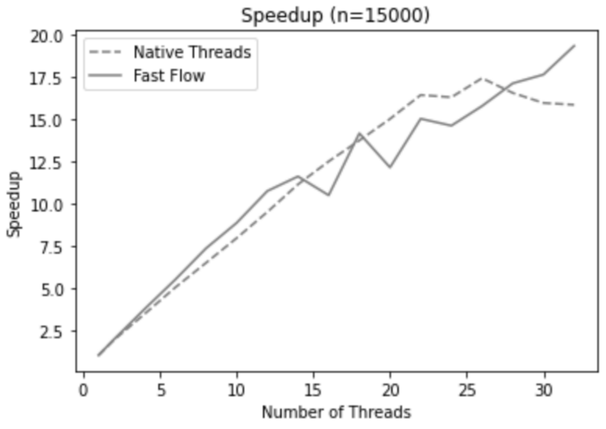

Figure 6: n=15000, K=100.

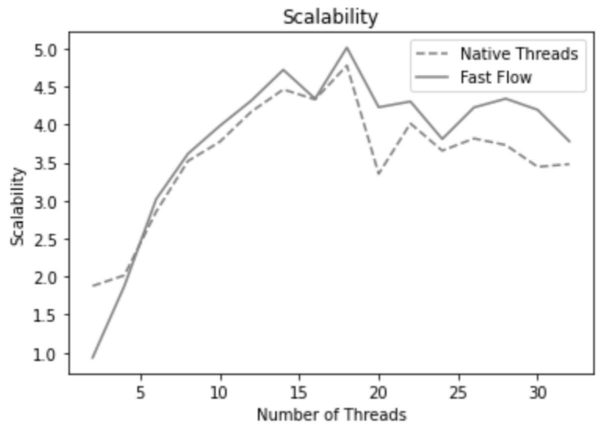

Figure 7: n=1000, K=100.

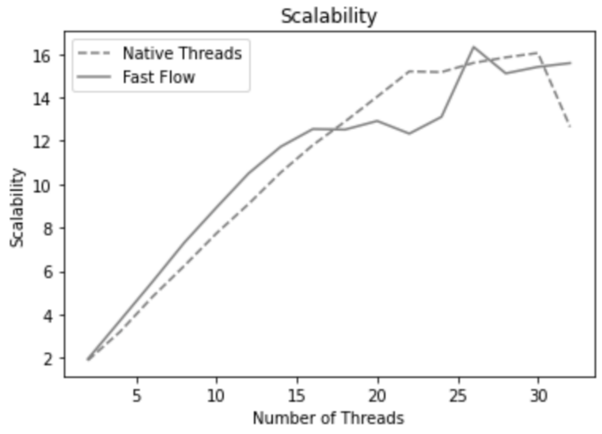

Figure 8: n=5000, K=100.

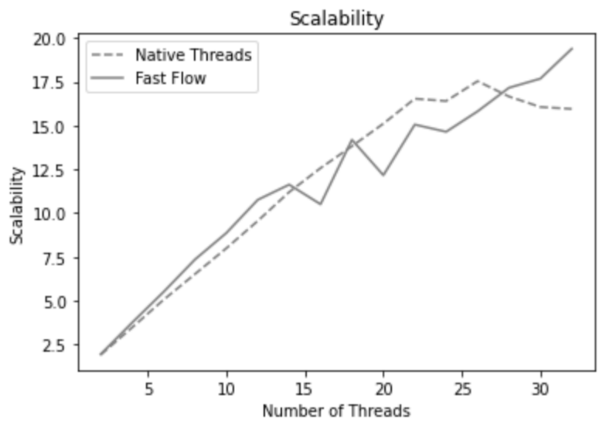

Figure 9: n=15000, K=100.

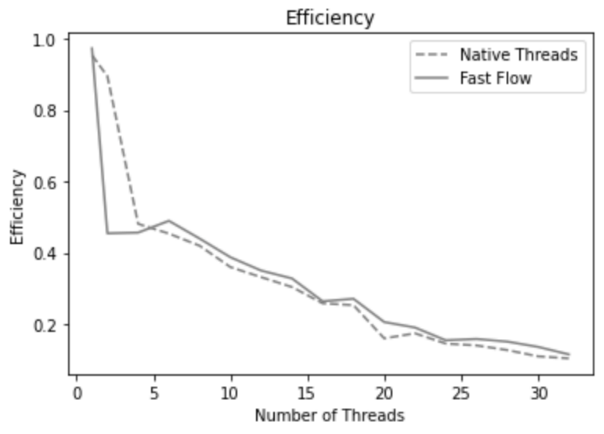

Figure 10: n=1000, K=100.

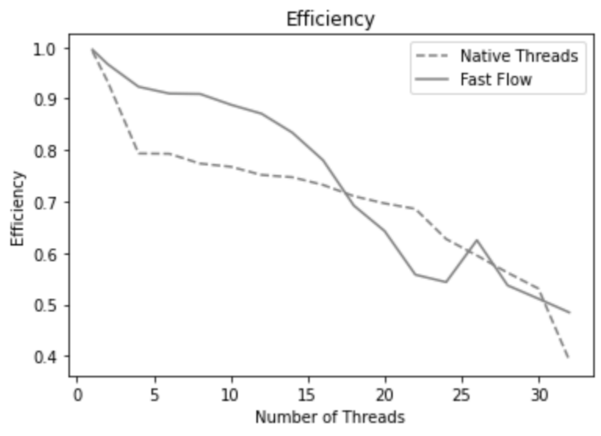

Figure 11: n=5000, K=100.

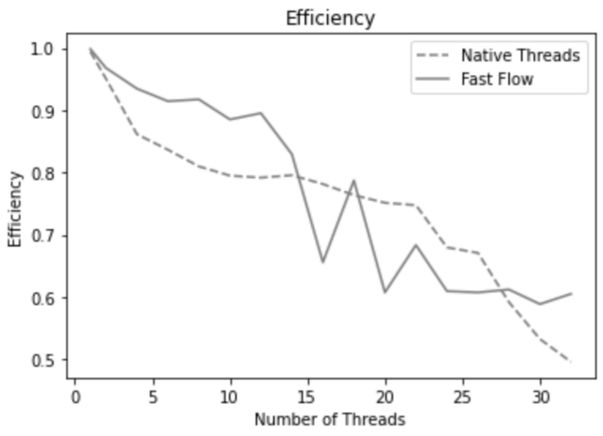

Figure 12: n=15000, K=100.

5 Conclusions

In conclusion we can see how using small matrices the performances are not very good due to synchronization overhead. On the other hand if we use bigger matrix then it leads to a smaller difference between the ideal performance and the real one.
The two parallel versions seems to behave in the same way even though sometimes FastFlow seems to perform slightly better rather than the Native Threads implementation.
It turns out that using a large matrix ($n = 15.000$) the best performance from the Fast Flow implementation are obtained when the number of workers is 32 and for the Native Threads implementation the best values are reached with 26 workers.

[1] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Fastflow: High-level and efficient streaming on multicore. *Programming multi-core and many-core computing system*, 2017.