Parallel And Distributed Systems Project: Jacobi iterative method implementation

Giuliano Galloppi - 646443 - g.galloppi@studenti.unipi.it - MSc in Big Data Technologies

Parallel and Distributed Systems: Paradigms and Models (305AA) - 9 CFU - A.Y. 2021/2022

Instructor: Professor Marco Danelutto

1 Introduction

In this report, the iterative Jacobi method is described, which in this project was developed with the aim of solving systems of linear equations by means of different types of implementations studied during the course, providing results and considerations about their performance.

The Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations, as $Ax = B$, where both the $x \in \mathbb{R}$ and B are vectors of length $n \in \mathbb{R}$, and the matrix has an $n \times n$ size.

With the formula:

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

a new approximation of the values of the different variables is calculated iteratively. In other words, we can calculate the approximation of the $i - th$ value of the unknown variable x belonging to the next iteration $k + 1$ respect the actual iteration k. The following pages will describe the project development process, which is divided into:

- **Analisys and Design**: understanding of the problem, also from a theoretical point of view, and approach to the solution.
- **Implementation**: description of the different implementations details of the parallel solution/algorithm regarding to the previous consideration.

- **Test and Results**: evaluation on the execution of different versions of the solution using several input parameters in order to collect all the results, and then make a comparison.

2 Analisys and Design

In this phase, the initial analysis of Jacobi's method and the subsequent algorithm from its sequential version is discussed. After some reading of the mathematical formula, a design of the sequential algorithm was made in a pseudocode version, that is following, from which the C++ implementation was then transposed.

Algorithm 1 Jacobi Sequential Algorithm

Require: $n > 0$

```
1:  function JacobiSequential(A, B, K)
2:      for k ← 0 to < K do
3:          for i ← 0 to < n do
4:              sum ← 0
5:              temp1 ← 1/A_ii
6:              for j ← 0 to < n do
7:                  if j ≠ i then
8:                      sum ← sum + A_ij · x_j^k
9:                  end if
10:             end for
11:             temp2 = B[i] − sum
12:             x_i^{k+1} = temp1 ∗ temp2
13:         end for
14:     end for
15: end function
```

As we can see from the pseudocode, the algorithm uses 3 loops, linked respectively to K which is the number of maximum iterations that incorporates the other loops (lines 2 - 14). This is the loop that allows us to process the elements in iteration k, i.e. the vector x^k, and then move on to iteration k+1, for which it uses a different vector x^{k+1}. This cycle is executed sequentially because the vector x^{k+1} is calculated using x^k; if we parallelised it, the results would be out of sync and therefore inaccurate.
The second cycle (lines 3 - 13) starts processing the matrix by calculating the vector x^{k+1}. This can be parallelised as we can subdivide the matrix into *portions*/chunks to be assigned to a different number of *workers*, ensuring that before the iteration change the calculation is completed and the vector of x for iteration k is updated.
The third and final loop (6-10) which performs part of the calculation to obtain the vector of the next iteration, can itself be parallelised.

Taking into account of the fact that we haven't a stream of data but the whole matrix, that has to be processed in the two central loops, we can observe that these corresponds respectively to the *map* and *reduce* parallel pattern, which their purpose is to minimize the completion time of the single iteration respect to their time spent on a sequential execution. In order to evaluate the performance of the algorithm and draw considerations on these we will taking into account the metrics of: speedup, scalability and efficiency. These metrics are obtained using $T_{seq}(n)$ as the sequential time having n as dimension for the linear system, which includes the time to initialise the variables added to the iteration $K \times (T_{iter} \cdot n)$. While $T_{par}(n, nw)$ is the parallel time using nw workers and having n as dimension for the linear system, that consists of T_{split} and T_{merge} as overhead to repart and merge again the work among the workers, and having the nw that divides the iteration time.

3 Implementation

In this section are described in detail the implementations about different version of the Jacobi's algorithm. They are:

1. *Sequential*: is the sequential implementation that refers pseudocode in 1.
2. *Native C++ Threads*: this implementation use the standard `thread` and `barrier` library.
3. *FastFlow*: this implementation use `FastFlow` that is a C++ parallel programming library, that let to use parallel pattern at high-level making development easy.

The project contains, in addition to the project main "`main_all.cpp`" that involves togheter these versions, separate mains for each of these versions in their respective folders ∼`\seq`; ∼`\thread`; ∼`\ff`.

3.1 User Guide

In the project is present the file `main_all.cpp` that is the principal "main file" of the project. To compile and execute:

- **Compile**: g++ -std=c++20 -O3 -o main_all.out main_all.cpp util.hpp util.cpp utimer.cpp -pthread
- **Execute**: ./main_all.out [K_MAX_ITER] [N_EXECUTIONS] [N] [N_THREAD]

where flags `[K\_MAX\_ITER]` are the iterations, `[N\_EXECUTIONS]` are the total runs of the algorithm to give an average of the times, `[N]` the size of matrix and vectors and `[N_THREAD]` the number of working thread involved in the parallel executions.

When the main is compiled and then executed, it will be possible choose which version of the algorithm execute, and after each execution will be displayed the execution time in average of the `[N\_EXECUTIONS]` inserted.

3.2 Premise

As a premise, for the correct execution of the algorithm we need its arguments. Both matrix A and vector B are randomly generated by the respective functions, which are based on the use of a seed given as input, followed by a minimum and maximum value. Seed and this lower and upper bound are defined as constant in each main of the project. Generators and other useful functions are defined in the `util.cpp` file.

3.3 Native C++ Threads Implementation

The native thread implementation uses the native thread library of C++. As mentioned in the previous sections, the central loop can be parallelized referring to the *map parallel pattern*. The implementation of this version is in a stand-alone way in the file `main_th.cpp` and in the principal file `main_all.cpp` as the function `jacobiThread`. The intuition of this version is to create as many workers as there are `[N_THREAD]` specified and each of them is assigned a function in the form of a lambda expression (function body). The idea developed is one in which the system is divided into parts, so that each thread calculates its assigned portion of the vector of x's of the next iteration(or say x's of $k+1$). Since the parallelisation works on a shared variable it requires a synchronisation mechanism, in particular the different threads must be synchronised on the current iteration, since while calculating the vector x^{k+1} no thread can be allowed to move on to the next iteration creating a data inconsistency. As an initial idea, the use of mutexes with condition variables was assumed, which together with wait() and notify() mechanisms allow good management of race conditions. A similar synchronisation mechanism that was the final choice is the barrier mechanism; it forces all threads to wait at a specific point and then grants progress from that point onwards. The `.arrive_and_wait` function placed at the end of each iteration therefore allowed the synchronisation of the threads.

3.4 FastFlow Implementation

The implementation with the library FastFlow uses a ff:ParallelFor object that is used to implement the map data parallel pattern. This lambda function matches with the execution of second loop that embody the third, where is computed the element i of the vector x about the next iteration $k+1$, at the step k. Pratically the lambda function corresponds to the iterations of the associated cycle that should be parallelized, that is to say that takes the function executed by thread workers. This implementation is in the

principal file `main_all.cpp` as the function `jacobiFastFlow` and in its stand-alone way in the file `main_ff.cpp`.

4 Test and Results

In the following section are reported the analysis of the results obtained by running the Jacobi's Algorithm making comparisons between the implementations previously described, using the same parameters.
The parameters considered to execute the tests are:

Parameters	**Values**
Number of iterations (K)	70
Linear System size (n)	1.000 / 5.000 / 15.000
Number of threads (only for parallel cases)	[1 - 32]

Table 1: Parameters of the tests

The number of workers starts using only 1 thread. Each experiment is repeated 5 times by averaging the completion times obtained to get more reliable results. Files have been compiled using the -O3 flag in order to optimize where possible the code such that to achieve the best possible execution time, and the -pthread flag for parallel version. All the tests have been executed on an Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz with 32 cores. In the next table are reported the time differences using the -O3 flag or not for the sequential implementation:

Size n	**n=1000**	**n=5000**	**n=15.000**
Time without -O3 flag (mu sec)	691153	1.92469e+07	1.67713e+08
Time with -O3 flag (mu sec)	110583	2.89467e+06	2.56029e+07

Table 2: Difference with the -O3 flag

Now it will be shown in the next plots the results obtained from the execution of the algorithm, using the measures of T_{seq} as the completion time spent by the sequential version of the program T_{par} as the completion time spent by the parallel version and the derived measures of:

- **Speedup**: $Sp(nw) = \frac{T_{seq}}{T_{par}(nw)}$ It gives a measure of how optimal parallelisation is compared to the best sequential computation.

- **Scalability**: $Sc(nw) = \frac{T_{par}(1)}{T_{par}(nw)}$ It gives a measure of how effective parallel implementation is in achieving better performance on larger parallelism degree.
- **Efficiency**: $\epsilon(nw) = Sp(nw)/nw$ It measures the aptitude of the parallel application in making good usage of the available resources.

In the following table are shown the timing results of the sequential version about the Jacobi's Algorithm:

Size n	**n=1000**	**n=5000**	**n=15000**
Sequential time	111939	2.9048e+06	2.56036e+07

Table 3: Sequential Time of Jacobi method

Here instead there are the results of the parallel implementations with $\boldsymbol{n = 1000}$ with some number of threads:

Number of Threads	**1**	**2**	**4**	**10**	**16**	**32**
Native thread par. time	110.378	71.922,4	36.757.4	33.227.4	24.960.9	41.168.2
Fast Flow par. time	109.743	130.442	108.005	26.942.4	26.370,5	33.104.2

Table 4: Parallel Time of Jacobi method

Regarding the time spent to set up 2 threads is around 130μs while the time spent to set up 32 threads is about 1928μs. Moreover about threads implementation we report also the overhead measures of the barrier synchronization method adopted, here are shown the average time that a thread has to wait the other ones:

Number of Threads	**2**	**4**	**10**	**16**	**20**	**32**
Overhead Average Time (mu sec)	303	340	406	521	630	1232

Table 5: Overhead on Number of thread

Now,in the next pages, it will presented plots regarding some of the derived metrics of the thread and fastflow implementations:

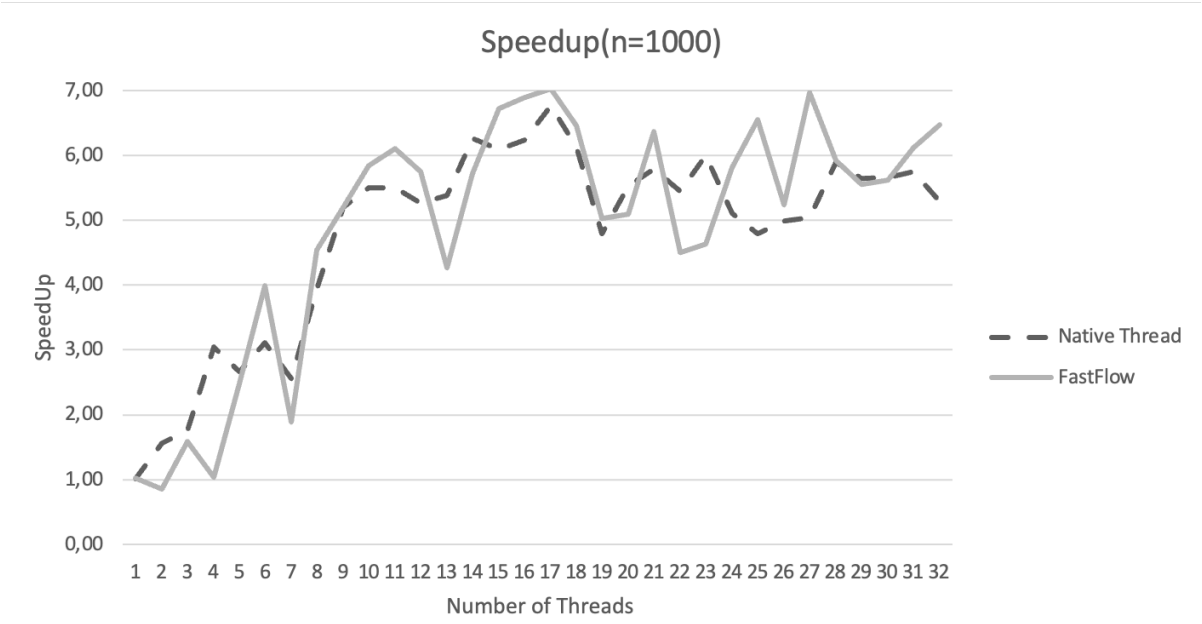

Figure 1: n=1000 , K=70

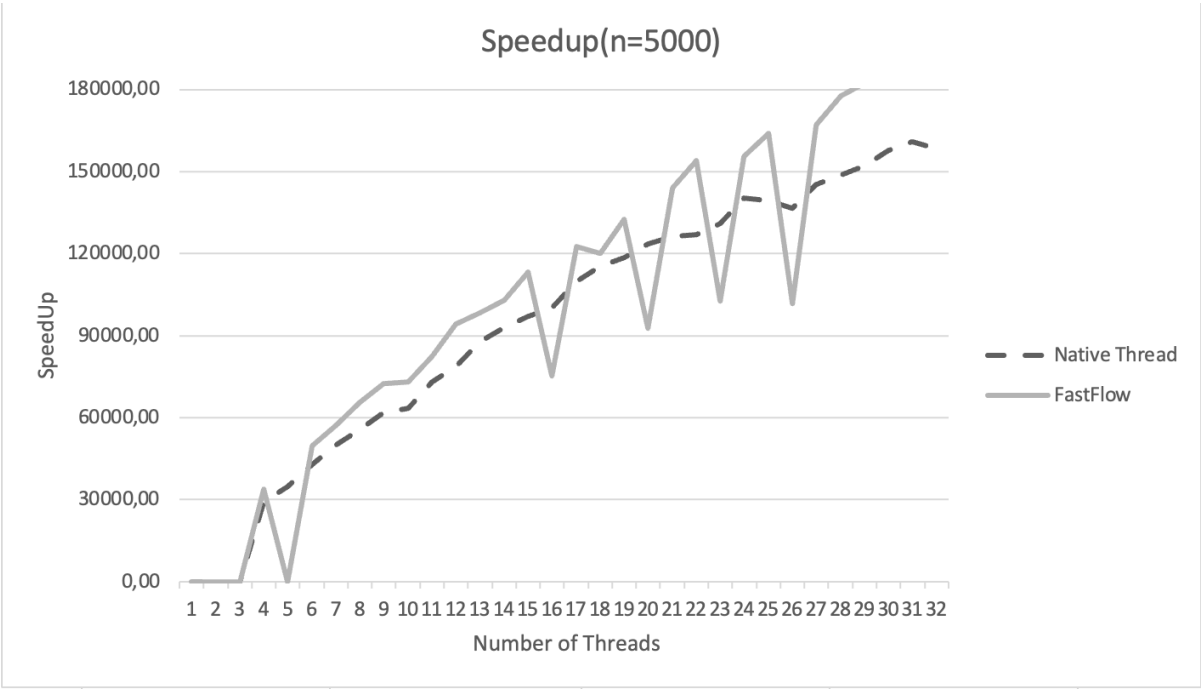

Figure 2: n=5000 , K=70

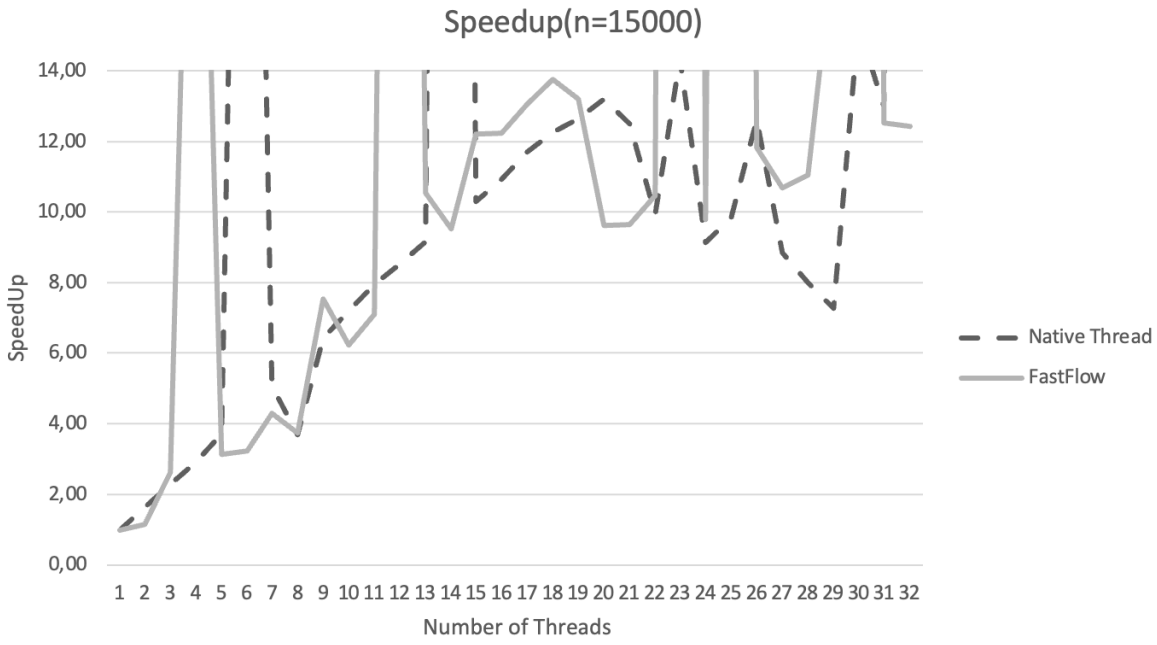

Figure 3: n=15000 , K=70

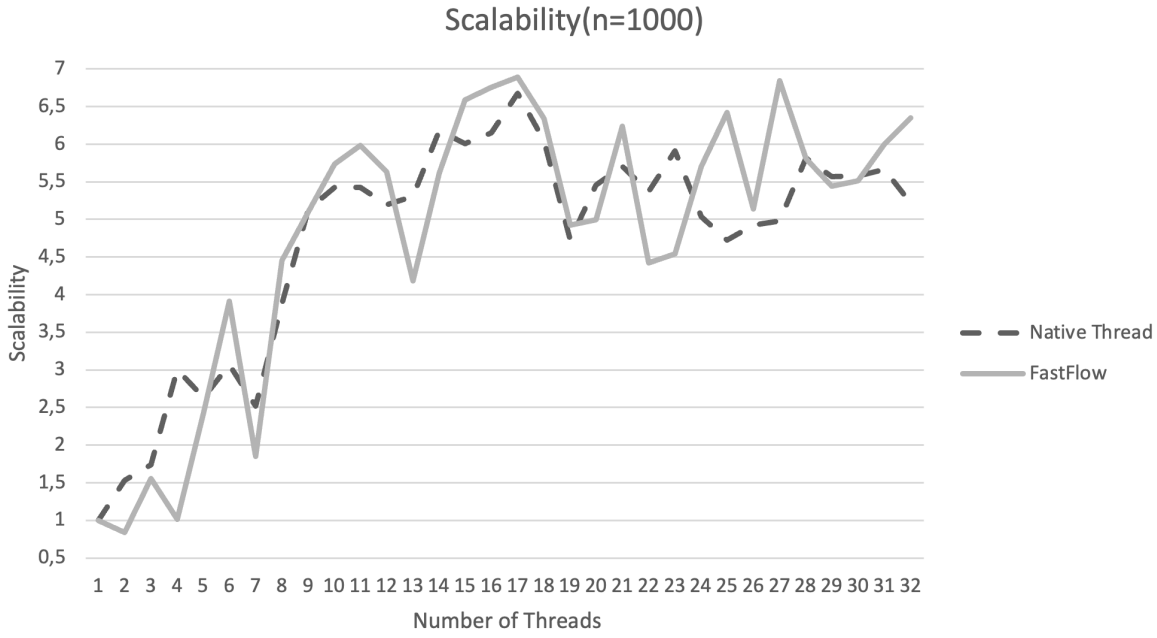

Figure 4: n=1000 , K=70

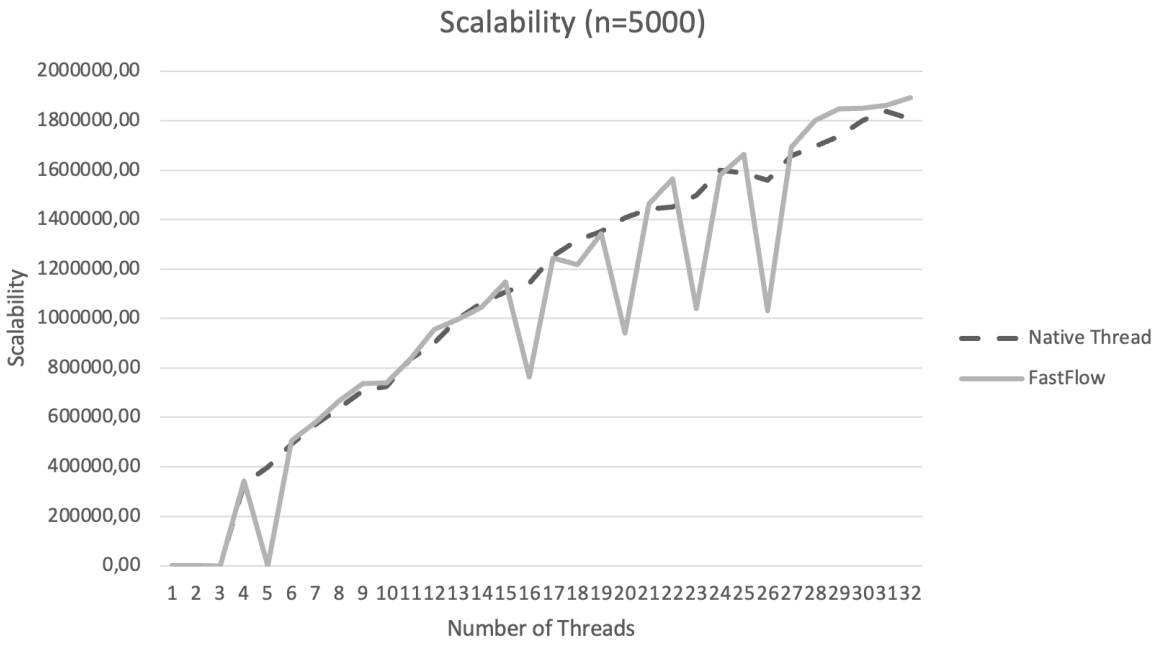

Figure 5: n=5000 , K=70

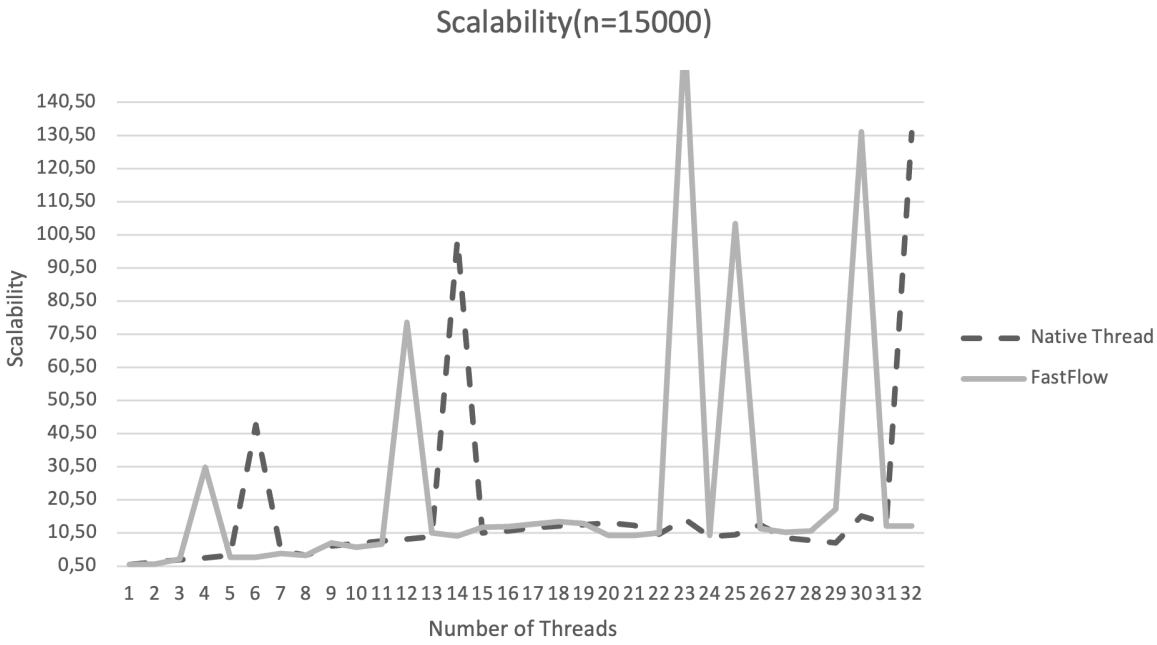

Figure 6: n=15000 , K=70

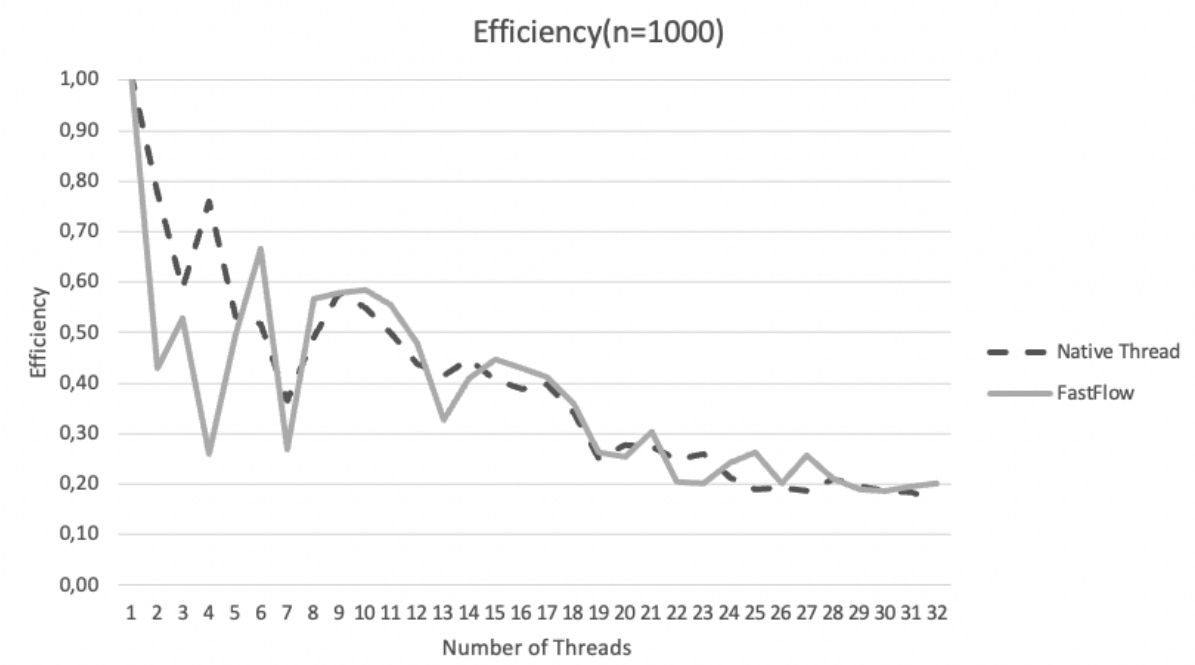

Figure 7: n=1000 , K=70

Conclusions: Considering the possible overhead, the analysis led to the choice of an implementation using the fastflow library over the native thread library, even though these seem to behave in the same way. The experiments revealed that the fastflow implementation performs worse with small matrix sizes and when few threads are involved in the computation. On the other hand, if larger matrices are used, the difference between ideal and actual performance is smaller. The best performance of the implementation with FastFlow occurs when the number of workers is set to 32, while for the Native Threads implementation the best values are achieved from 17 workers upwards. Thus the parallelisation of Jacobi's iterative method gives benefits on completion time.