Parallel and distributed systems: paradigms and models

Project; Applications Jacobi (AJ), 2021/22

Carlo Tosoni 644824

August 2022

1 Analysis of the problem

1.1 Choice of the parallel patterns

In this first part of the report I will analyze the pseudocode of the Jacobi algorithm to understand which could be the possible strategies to parallelise it. Below I wrote the pseudocode of the Jacobi Iterative Method.

Algorithm 1: The Sequential Jacobi Method

```
Data: a linear system $Ax = b$, which admits only one solution and whose matrix $A$ has
      its entries $a_{ii} \neq 0$, an initial vector $xo = x^0$, a maximum number of iterations $N$, a
      maximum tolerance $TOL$ for convergence.
Result: The solution of the linear system, when convergence is reached.
1  $k \leftarrow 1$;
2  while $k \leq N$ do
3      for $i \leftarrow 1$ to $n$ do
4          $x_i \leftarrow 0$;
5          for $j \leftarrow 1$ $n$ do
6              if $i \neq j$ then
7                  $x_i = x_i + A_{ij} \times xo_j$;
8              end
9          end
10         $x_i \leftarrow \frac{1}{A_{ii}}(b_i - x_i)$;
11     end
12     if $\frac{||x - x\_old||}{||x||} \leq TOL$ then
13         return $x = (x_1, x_2, ..., x_n)$;
14     end
15     $k \leftarrow k + 1$;
16     $xo \leftarrow x$;
17 end
18 return $x = (x_1, x_2, ..., x_n)$;
```

The first thing that we can notice is that the vector x depends on x_{old}, therefore the external while loop at line 2 cannot be parallelised and it must be executed sequentially. On the contrary, the two internal for loops (lines 3 and 5) could be executed in parallel, because there are no dependencies among different iterations of these for loops. Indeed, at each iteration of the Jacobi method is possible to compute the element x_i independently from x_j, $\forall i, j \leq n, i \neq j$. The for loop at line 3 can be parallelised using a map parallel pattern, indeed here we have a single input task completely available when the for loop starts which can be divided into different subtasks that can be executed in parallel by different threads. On the other hand, the second for loop at line 5 can be parallelised using a reduce parallel pattern; the reduce variable is x_i and the operation to compute is the sum, which is both commutative and associative; these are mandatory properties required to use a reduce parallel pattern. I chose to not parallelise the second for loop, because I had expected a overhead too large and a grain too small if I adopted this solution, therefore I decided to parallelise only the first for loop at line 5 using a map parallel pattern.

In the Jacobi algorithm is also possible to add a stopping criterion, which has to be computed at each Jacobi iteration. The stopping criterion corresponds to line 12 of the pseudocode above. There are different stopping criteria that could be used, in the pseudocode above I decided to use $||x - x_{old}||/||x||$, another possible stopping criterion could have been $||x - x_{old}||$. The pseudocode below shows us how we can compute the norm of the two vectors $x - x_{old}$ and x.

Algorithm 2: Compute the stopping criterion $||x - x_{old}||/||x||$

```
1 $num \leftarrow 0$;
2 $den \leftarrow 0$;
3 for $i \leftarrow 1$ to $n$ do
4     $num \leftarrow num + ((x_i - x_{oldi}) * (x_i - x_{oldi}))$;
5     $den \leftarrow den + (x_i * x_i)$;
6 end
7 $num \leftarrow \sqrt{num}$;
8 $den \leftarrow \sqrt{den}$;
9 return $num/den$;
```

from this pseudocode is clear that is possible to parallelise the computation of the norm using a map parallel pattern, to calculate the square of the components of each vector (i.e. $(x_i - x_{oldi}) * (x_i - x_{oldi})$ and $x_i * x_i$), followed by a reduce parallel pattern, to sum all the values. The speed up achieved by parallelising the computation of the norm using these two parallel patterns becomes negligible when $N \to +\infty$, where N is the linear system's dimension, and in general it should not affect much the sequential fraction. For these reason I decided to not parallelise the computation of the norm, but to compute it sequentially.

1.2 Analysis of model performance

Let us now analyze which can be the performance of the parallel patterns described in the previous section.

Firstly, let's consider which could be the achievable speed up using a map to parallelise the first internal loop of Algorithm 1. The number of operations that a sequential execution of the Jacobi method has to execute, omitting constants, is given by $\#seq\_op = M \times N \times N + M \times N + M$, where M is the total number of iterations to execute and N is the linear system's dimension. If we parallelise the for loop at line 3 of Algorithm 1, this number drops down to $\#par\_op = \frac{M \times N \times N}{Nw} + \frac{M \times N}{Nw} + M \times N + M$, where Nw is the parallel degree. Therefore, the achievable speed up can be approximated to;

$$speed\_up(N, M, Nw) \approx \frac{M \times N \times N + M \times N + M}{\frac{M \times N \times N}{Nw} + \frac{M \times N}{Nw} + N \times M + M}$$

which is equal to Nw if $N \to \infty$. Thence, theoretically speaking, if we consider negligible the costs to initialise and to join threads and to split subtasks among threads, a map could achieve almost a linear speed up when N is sufficiently large. This is a direct consequence of the fact that the sequential fraction goes to 0 when N becomes large. These results does not change if we want also to compute at each iteration the stopping criterion at line 12 of Algorithm 1. Indeed, whether we compute the norm of $x - x_{old}$ and x sequentially of with a map-reduce parallel pattern, the achieved speed up should not change much, and in both cases the speed up should tend to Nw when N is particularly large.

Despite that, there are other sources of overhead that could let plummet the speed up of this parallel program. First of all, we have to take into account that the iterations of the while loop of Algorithm 1 at line 2, must be executed sequentially. In other words, it means that before starting a new Jacobi iteration, we have to wait that all the threads have finished performing all the calculations of the previous one. Therefore, if there is only one thread which is much slower than the others, it means that all the others have to wait it to start computing the next Jacobi iteration. That could be a huge problem, since if we do not choose an appropriate policy for load balancing, some threads are likely to have more works to carry out than others. To face this problem, I will implement two different load balancing policies; a static policy and a dynamic one, expecting to measure better performance with the dynamic policy rather than with the static one, since load balancing is a crucial problem for this project. However, I expect to measure a speed up not close to the linear speed up, since the dependencies among different Jacobi iterations should impact negatively the performance of the algorithm.

2 Code implementation

Now that I have analyzed the problem, I can start to implement the code of the project. In this section I am going to describe which are the files that I wrote for this project and which are their main features. I wrote 4 different versions of the Jacobi method; 1 sequential and 3 parallel (the parallel versions parallelise only the for loop at line 3 of Algorithm 1). The table below summarises the most important features of each version.

Name	**Implementation**	**Load Balancing**
seq_jacobi.cpp		
par_jacobi.cpp	Barriers and Native C++ Threads	Static Policy
par_jacobi2.cpp	Thread Pool (implemented using Native C++ Threads)	Dynamic Policy
par_jacobi_ff.cpp	FastFlow library	Dynamic Policy (also possibility to choose a Static Policy)

2.1 Data creation

All the 5 versions of the Jacobi method use the same code to instantiate the linear system. The matrices A, b are created randomly using a seed provided by the user, the matrix A is created in such a way that it is strictly diagonally dominant, because the Jacobi method can more easily converge when the matrix A satisfies this property. Finally, the initial vector xo passed to the Jacobi method is the null vector. All the 4 implemented versions use the file `my_timer.cpp` to measure the elapsed time during their execution; in the experiments that I carried out, the time elapsed to initialise the linear system has never been measured. `my_timer.cpp` has been created using the Chrono library.

2.2 Jacobi algorithms

seq_jacobi.cpp

It implements the sequential version of the Jacobi method. The implementation of the sequential code is very similar to the pseudocode of Algorithm 1, with the only exception that the user can specify if the program has to compute the stopping criterion at line 12 at each iteration of Jacobi or not. For more details, look at the function `seq_jacobi()`.

par_jacobi.cpp

This version of the Jacobi method has been implemented using barriers to synchronize the native threads. The threads compute in parallel the for loop from line 3 to line 10 of Algorithm 1. At the end of this for loop a barrier awaits all the threads to prepare the next iteration of the while loop. For the load balancing I used a static policy, and in particular the subtasks are distributed among the threads using a cyclic distribution. the user can specify to the program if it has to compute the stopping criterion at the end of each iteration; in case the stopping criterion is computed sequentially.

par_jacobi2.cpp

This file parallelises the for loop at line 3 of Algorithm 1 using a thread pool. The thread pool has been implemented using native C++ threads. In this version of the Jacobi method, the main thread pushes subtasks inside a queue, each subtask corresponds to a certain number of consecutive iterations of the for loop that have to be executed by a single worker. When a worker has finished to execute its subtask, it can extract another subtask from the queue. At the end of each Jacobi iteration (i.e. at the end of each iteration of the while loop at line 2), the main thread refills again the queue with new subtasks so the workers can compute them. This file uses auto scheduling as load balancing policy, which is a dynamic policy. This version of the Jacobi method has not been programmed to compute any stopping criteria, therefore its execution finishes only when all the Jacobi iterations have been computed.

par_jacobi_ff.cpp

Parallel version of the Jacobi method implemented using the FastFlow library. The for loop has been parallelised using the class `ParallelFor` and its method `parallel_for()`. The method `parallel_for()` requires to know the chunks' size; if the passed number is positive, then `parallel_for()` uses a dynamic policy for load balancing, otherwise, if it is negative, the policy adopted is static. In my experiments, I will always use positive numbers to represent the chunks' size, therefore the policy will always be dynamic. Also this version of the Jacobi method does not compute any stopping criteria.

3 Experiments and expected results

With the files described in the previous section, I will carry out two different experiments.

In both experiment I will analyze the perfomance of the methods that I had implemented, in the first experiment I will use a smaller linear system and run less iterations, while in the second experiment I will use a larger linear system and run more Jacobi iterations. I expect to measure a higher speed up on the second experiment due to the fact that the overhead caused by the sequential execution of the jacobi iterations should become more negligible as the size of the linear system increases. In both experiments I will not compute any stopping criteria, since I noticed that it does not impact much the performance of the algorithm, especially when the linear system is particularly large. I expect that the load balancing policy will have a great impact on the performance of these algorithms; in particular I expect that `par_jacobi.cpp` (the version with barriers), will fail to achieve a linear speed up, due to the fact that this algorithm uses a static policy and therefore is likely that a thread will be much slower than the others. On the other hand, `par_jacobi2.cpp` (the version with a thread pool) should achieve a speed up closer to the linear speed up when the number of workers grow due to its dynamic policy for load balancing. Despite that, this version has to deal with an additional overhead caused by the mutually exclusive access to the shared queue, therefore the perfomance of this algorithm will likely decrease when the number of workers is too high, especially when the linear system is small. Also `par_jacobi_ff.cpp` could perform well, since even this version uses a dynamic policy for load balancing.

In both experiments to measure the performance of these algorithms I will use the completion time, the speed up, the scalability and the efficiency.

4 Results of the experiments

In this section I will plot the results of both experiments. The experiments have been carried out on a machine with 32 processors, each processor is a: Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz.

For both experiment I used up to 32 workers, each value plotted in the graphs below was obtained averaging three different executions using different seeds. The first experiment was carried out on 1024×1024 linear systems, performing 32 Jacobi iterations, while for the second experiment I used 8192×8192 linear systems, performing 256 Jacobi iterations.

4.1 Experiment 1

Below there are all the graphs related to the first experiment.

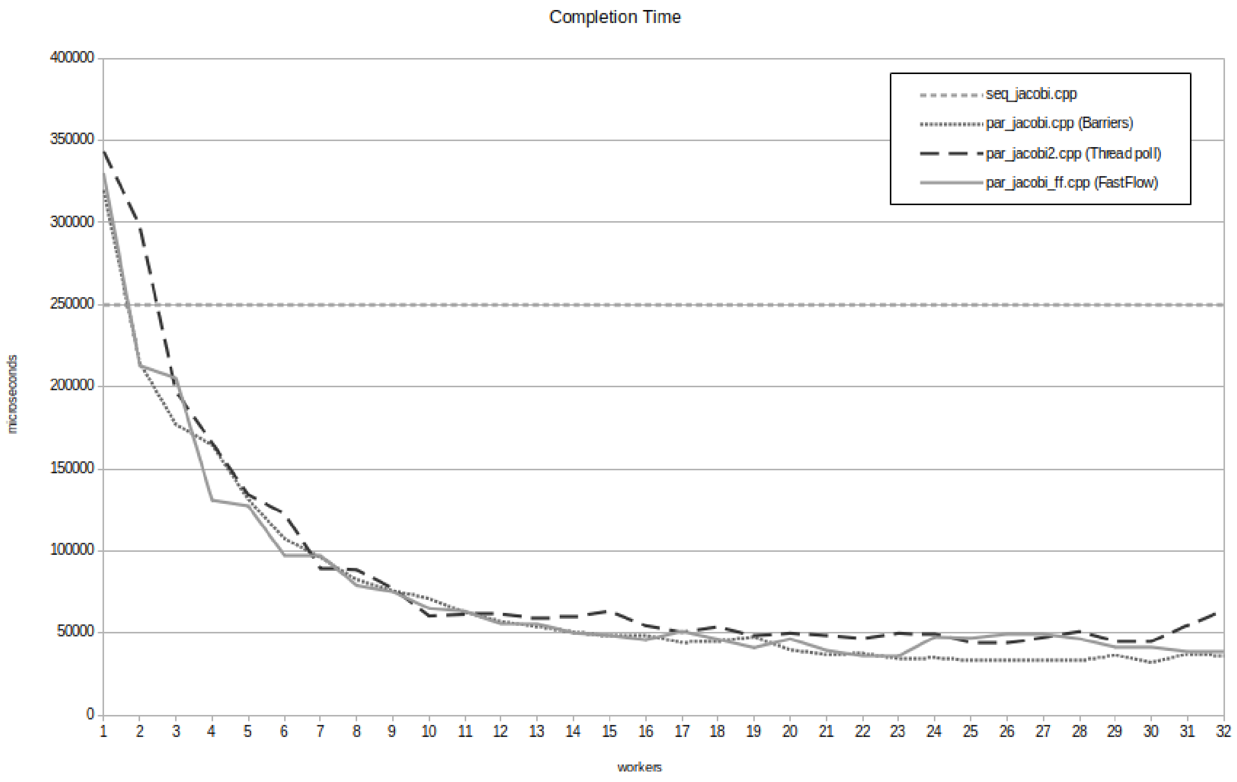

Figure 1: Elapsed time, N=1024, M=32.

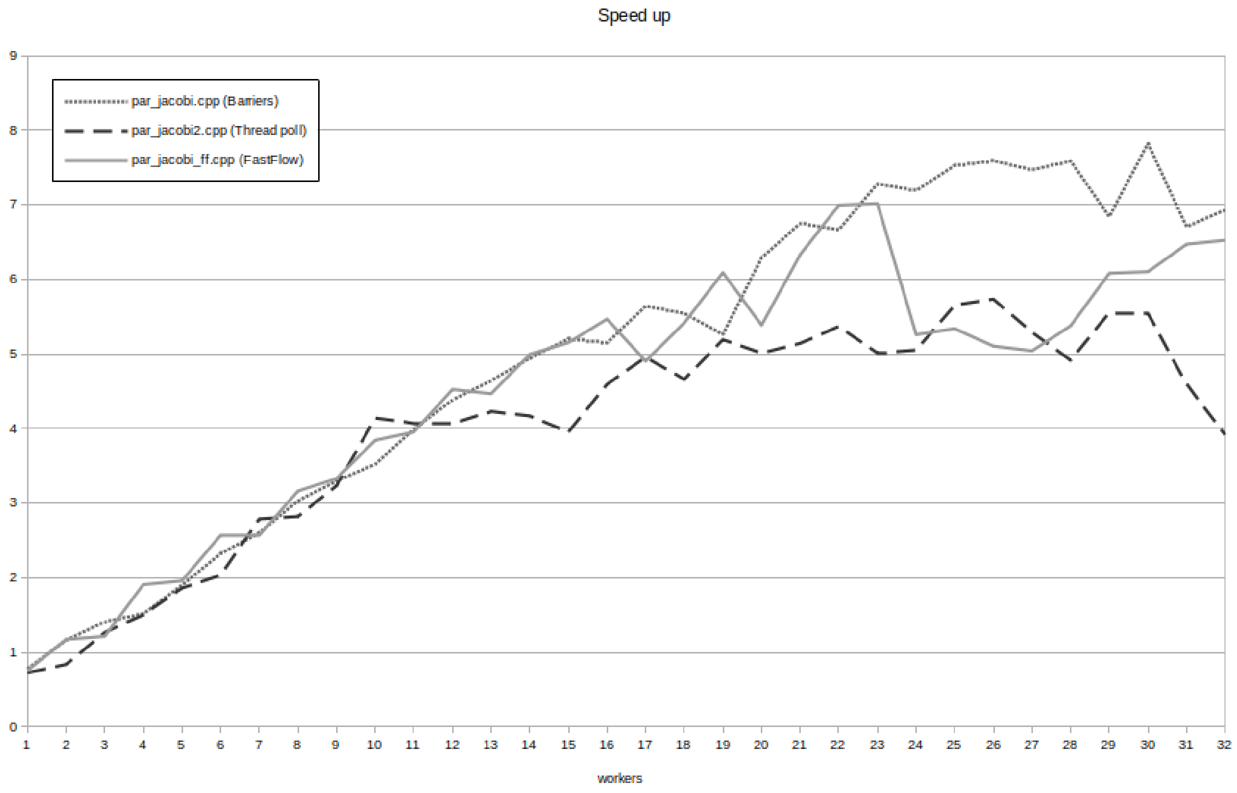

Figure 2: Speed up, N=1024, M=32.

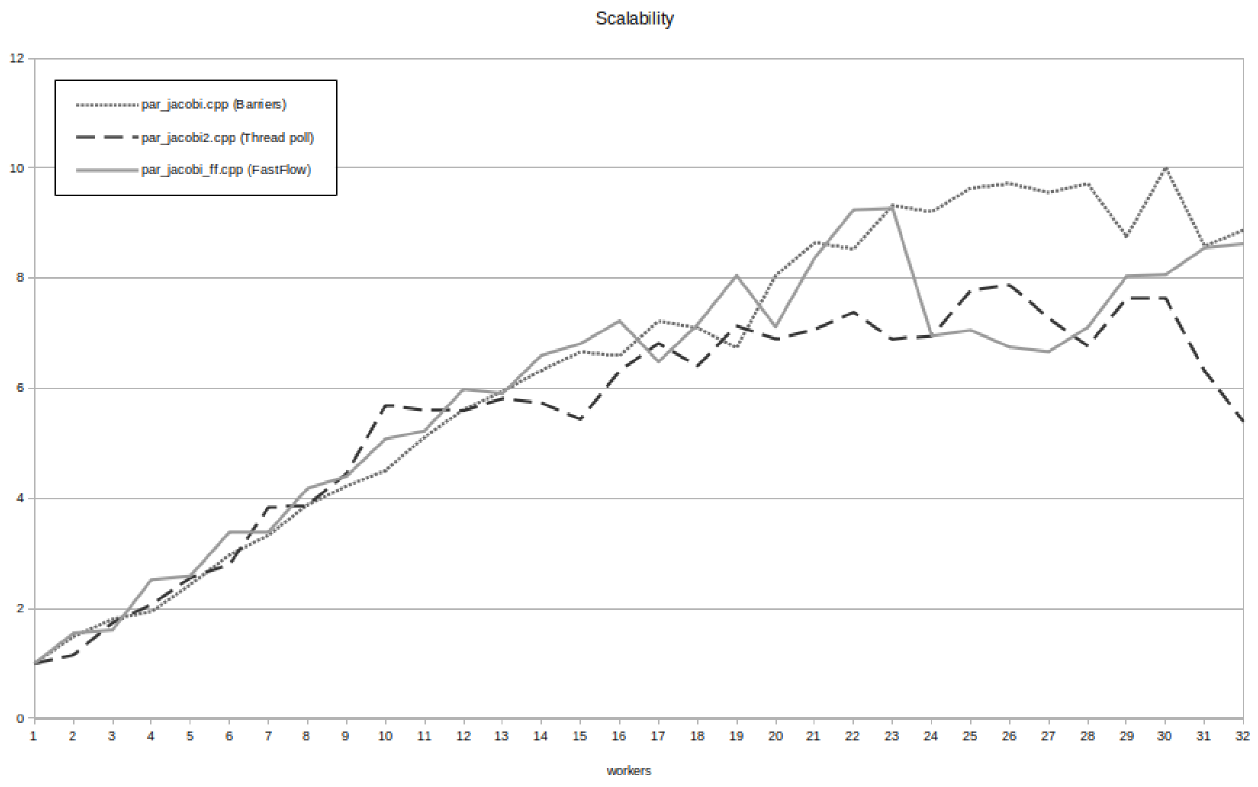

Figure 3: Scalability, N=1024, M=32.

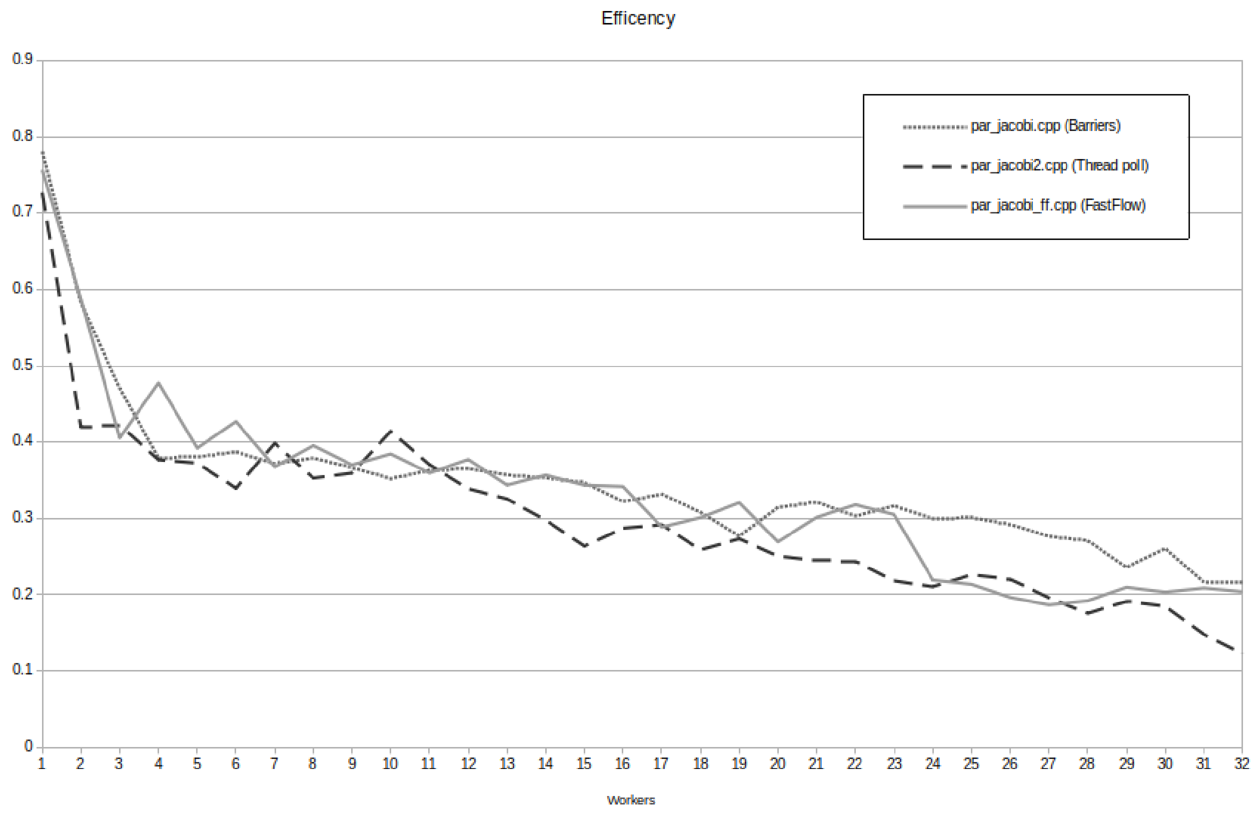

Figure 4: Efficency, N=1024, M=32.

Comments

In the first experiment I achieved a speed up far from the ideal one, here `par_jacobi.cpp` (the version that uses barriers), achieved the highest speed up, with a maximum peak of 7.8. On the other hand `par_jacobi2.cpp` (the version that uses a thread pool) struggled much more, reaching only a 5.7 of speed up and its performance started to drop sharply when the number of workers became too large, probably due to the presence of the shared queue. Another remarkable fact is that the efficiency decreased steadily as the number of workers grew, plummeting to about 0.2 when the programs used 32 threads.

4.2 Experiment 2

Now let's see how these results change when the linear system is 8192×8192 and the number of iteration is 256. Below there are all the graphs related to this second experiment.

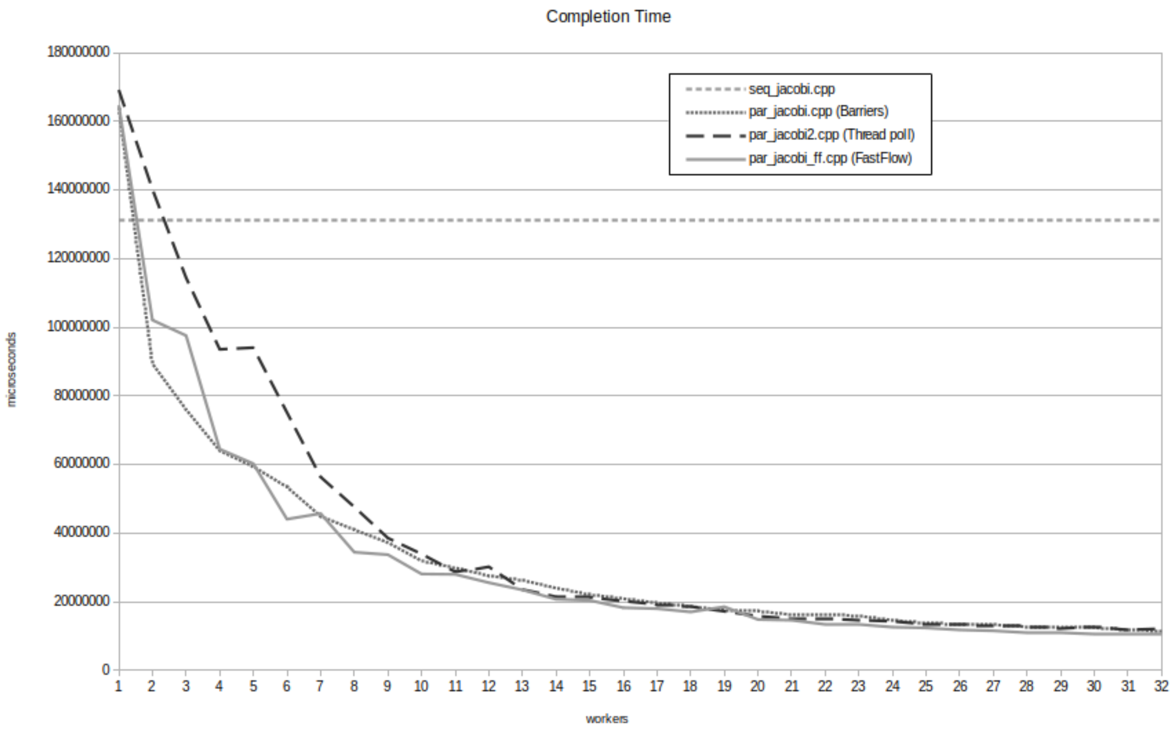

Figure 5: Elapsed time, N=8192, M=256.

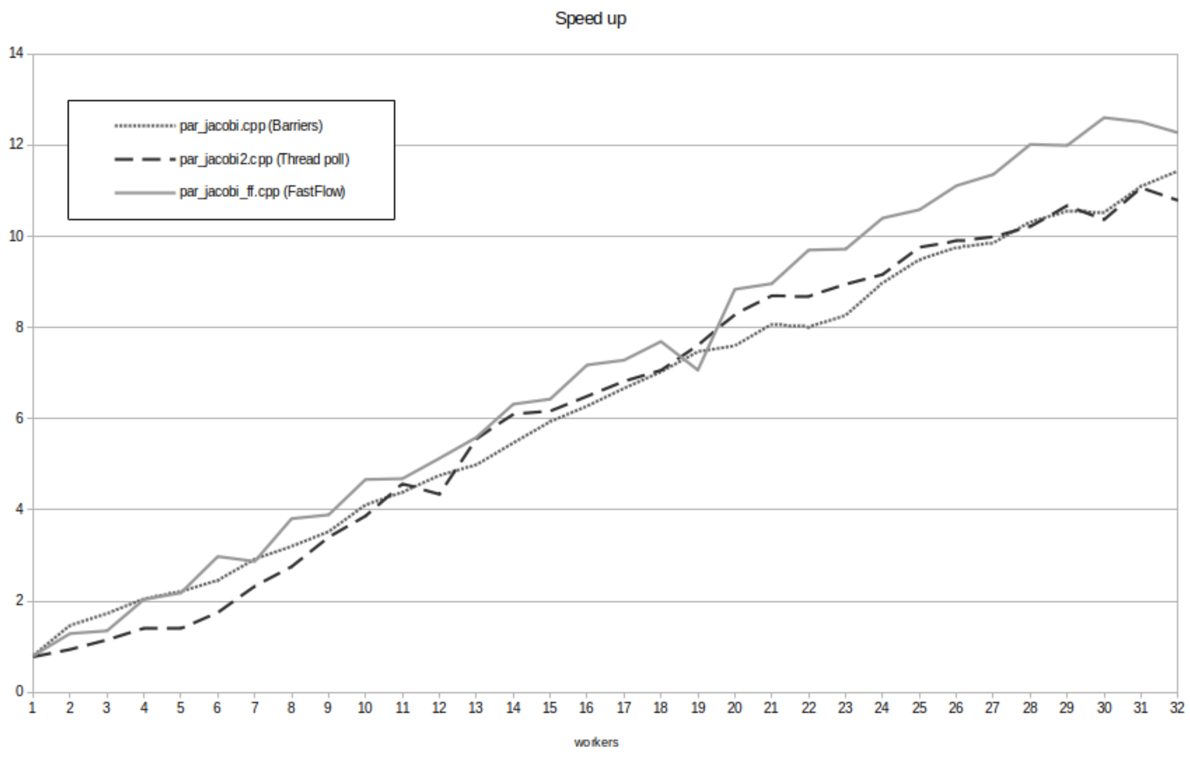

Figure 6: Speed up, N=8192, M=256.

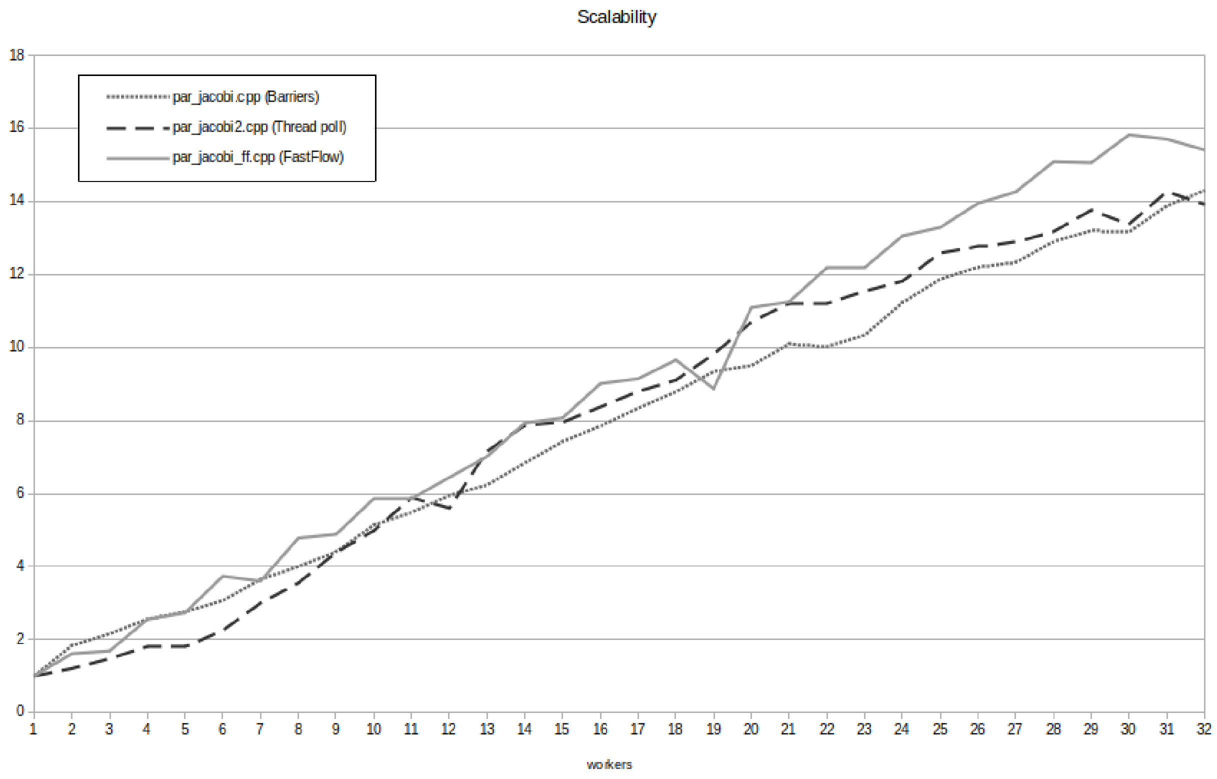

Figure 7: Scalability, N=8192, M=256.

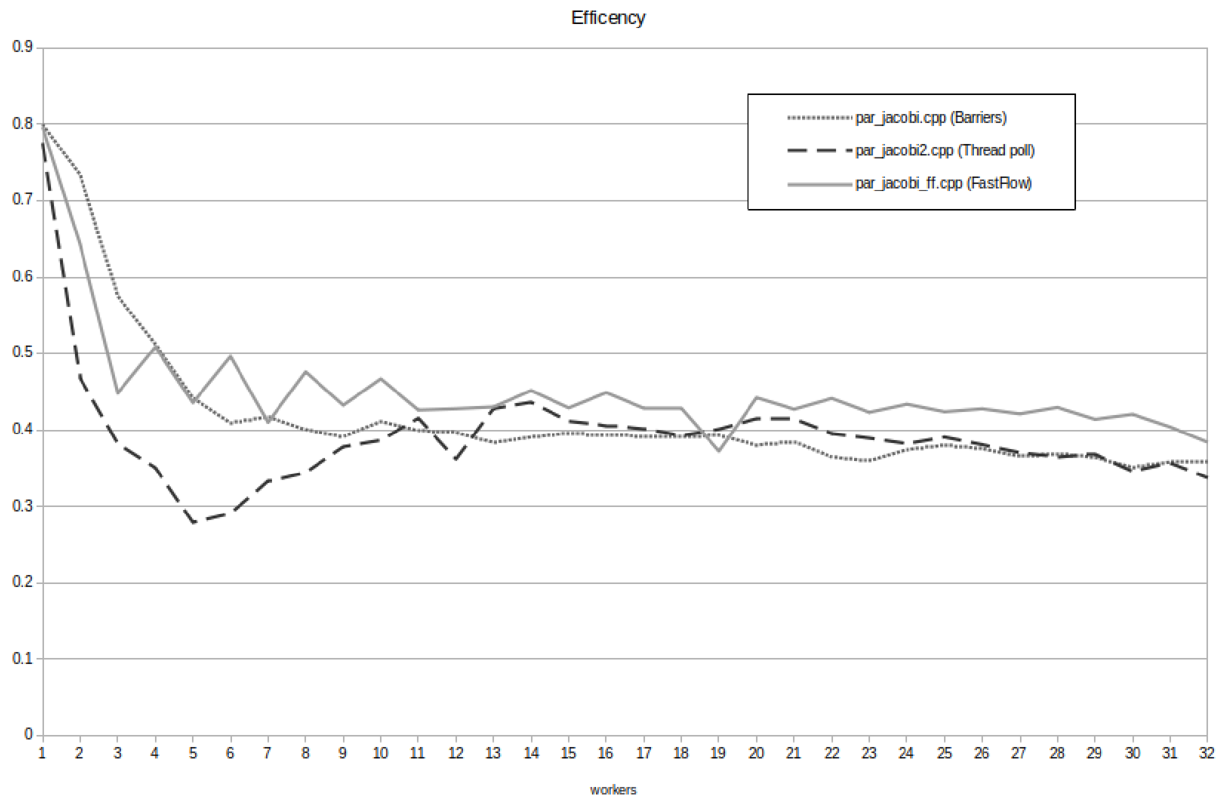

Figure 8: Efficiency, N=8192, M=256.

Comments

These time the experiment produced a different result. First of all, surprisingly, the three versions of the Jacobi method performed more or less the same. `par_jacobi_ff` (FastFlow) revealed to be slightly faster than the two other methods, reaching a maximum speed up of 12.6, while in the first experiment it had achieved a speed up of 6.9, therefore the speed up of `par_jacobi_ff` almost doubled in this second experiment. Despite that, the speed up remained significantly lower than the ideal speed up. `par_jacobi` (barriers) and `par_jacobi2` (thread pool) achieved a maximum speed up of, respectively, 11.4 and 11.0. Moreover, in this second experiment the efficiency remained more or less stable as the number of workers grew, in fact the measured efficiency is about 0.4 for all the three implementations of the Jacobi method. Despite the fact that the efficiency improved a lot compared to the first experiment, it remained quite low even in this second experiment.

5 Conclusions

In conclusion, all the three implementations of the Jacobi method struggled to parallelise the Jacobi iterations in the first experiment, where the linear system was quite small. However, they performed much better in the second experiment, even though they failed to reach a speed up close to the ideal one. Probably, if I increased even more the linear system's dimensions, the three versions of the Jacobi methods could have performed even better, reaching almost a linear speed up. It is likely that `par_jacobi_ff` performed better with the larger system due to its dynamic policy for load balancing, and it could have outclassed the other two implementations with a even larger linear system.