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 ← 1;
2  while k ≤ N do
3      for i ← 1 to n do
4          x_i ← 0;
5          for j ← 1 n do
6              if i ≠ j then
7                  x_i = x_i + A_ij × xo_j;
8              end
9          end
10         x_i ← 1/A_ii (b_i − x_i);
11     end
12     if ||x − x_old|| / ||x|| ≤ TOL then
13         return x = (x_1, x_2, ..., x_n);
14     end
15     k ← k + 1;
16     xo ← 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.

Now let's analyze which of the two patterns could be the best option to parallelise the Jacobi method and hence let us consider the time required to execute the Jacobi method using either a map parallel pattern or a reduce parallel pattern. From now on I will indicate with T_{total} the total time required to execute the Jacobi method using one of the two parallel patterns, now $T_{total} = T_{setup} + T_{par}$, where T_{setup} is the time needed to setup the parallel activities, while T_{par} is the time needed to execute the Jacobi method using multiple threads. T_{par} can be easily approximated to T_{seq}/nw, where nw is the parallel degree, while to approximate T_{setup} we have to take into account some aspects of the algorithm that we want to parallelise. Now let us imagine that we create the threads once, at the beginning of the Jacobi method, and we use them for the entire execution of the program; I have measured that, the time required by the remote machine to instantiate a single thread is roughly equal to $50\mu sec$, therefore the time required to instantiate threads is given by the formula $nw \times 50\mu sec$. Now whenever a thread has to wait that the others finish their calculations, we could imagine to use a condition variable to notify them when every thread is ready for the next iteration. I have measured that the time required to notify a thread is approximately equal to $7\mu sec$, therefore the overhead introduced by the condition variable can be expressed as $nw \times 7\mu sec \times \#times$, where $\#times$ is the number of times that we have to wait that each thread has finished its computations. It follows that T_{setup} can be expressed as $T_{setup} = nw \times 50\mu sec + nw \times 7\mu sec \times \#times$
and therefore, $T_{total} = nw \times 50\mu sec + nw \times 7\mu sec \times \#times + T_{seq}/nw$.
Now if we want to use a map parallel pattern, $\#times$ must be equal to the variable N of the pseudocode above, which represents the number of Jacobi iterations that we want to execute. Anyway, this number is much larger in the case of the reduce parallel pattern, in fact, since we have to wait all the threads at the end of each iteration of the for loop at line 5, it follows that in the case of the reduce parallel pattern $\#times$ is equal to $N \times n$ where n is the linear system's dimension. Hence, the two formulas for the two parallel patterns are;

$$T_{total\_map} = nw \times 50\mu sec + nw \times 7\mu sec \times N + \frac{T_{seq}}{nw}$$

$$T_{total\_reduce} = nw \times 50\mu sec + nw \times 7\mu sec \times N \times n + \frac{T_{seq}}{nw}$$

From these formulas is clear that the best option is the map parallel pattern, while the grain of the reduce is too low. For instance, let us suppose that we have a linear system 2048x2048 and that we want to execute 16 Jacobi iterations (i.e. $N = 16$ and $n = 2048$), using the two formulas above we can figure out which is the lowest completion time we can hope to achieve using the two parallel patterns. Firstly, I implemented a file called `seq_jacobi.cpp` (that I will introduce later on this relation) to measure the elapsed time required to execute sequentially the Jacobi method on this system. This program requires about $100000\mu sec$ to execute 16 Jacobi iterations; now we have all the ingredients to compare the two parallel patterns. As we can see on the chart below (image 1), the map parallel pattern outclassed the reduce, therefore, to parallelise this program, I decided to use a map.

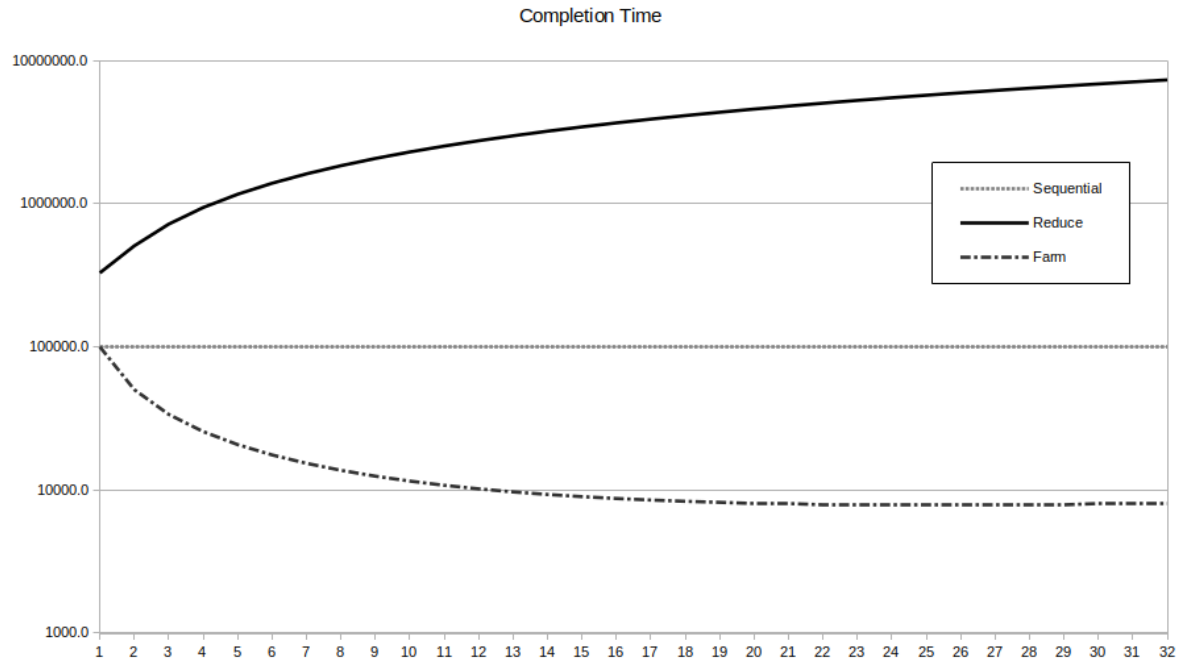

Figure 1: Expected elapsed time, n=2048, N=16, y-axis on a logarithmic scale

Moreover, there is also another aspect of the reduce parallel pattern that we have not considered yet. let us suppose that we have a vector of n elements. To sum its elements, we could build a logarithmic tree, where at the each level, we create pairs of elements and we sum them, and so we proceed until only one element remains. If we suppose to have $n/2$ available threads, this tree can be computed only using $log_2(n)$ steps, i.e. one step per level. Another possible approach would be to divide the vector of n elements in nw chunks and to assign each chunk to a thread. To compute the global sum, each thread can compute the local sum of its chunk, and then the main thread can sum all the chunks' local sums, finding the global sum of our vector. In this latter case, the number of steps required to follow this approach would be n/nw to compute the local sums and nw to sum all the local sums. If we consider that a sequential execution would require n steps, it follows that, independently from the approach we use, with a reduce parallel pattern is not possible to reach a plain linear speed up, therefore the real completion time of the reduce is even higher than the one plotted in fig 1.

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 (Algorithm 2) shows us how we can compute the norm of the two vectors $x - x_{old}$ and x. 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. But in this case the overhead introduced by this solution would be higher than the achieved grain, indeed, the computation of the norm requires only a few microseconds and it becomes negligible when $n \to +\infty$, where n is the linear system's dimension. Therefore it would make no sense to parallelise the stopping criterion.

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

```
1 num ← 0;
2 den ← 0;
3 for i ← 1 to n do
4 |   num ← num + ((x_i − x_oldi) ∗ (x_i − x_oldi));
5 |   den ← den + (x_i ∗ x_i);
6 end
7 num ← √num;
8 den ← √den;
9 return num/den;
```

It follows that I will parallelise only the for loop at line 3 of Algorithm 1 using a map parallel pattern. Using the expected completion time plotted in figure 1, I computed the maximum speed up achievable using a map parallel pattern on a system 2048x2048 with 16 Jacobi iterations. As we can see on the image below, this speed up is quite lower than the linear speed up, in fact, in this case we have to deal with a non negligible sequential fraction required to orchestrate the treads, which affect the performance of the program when the number of workers increase.

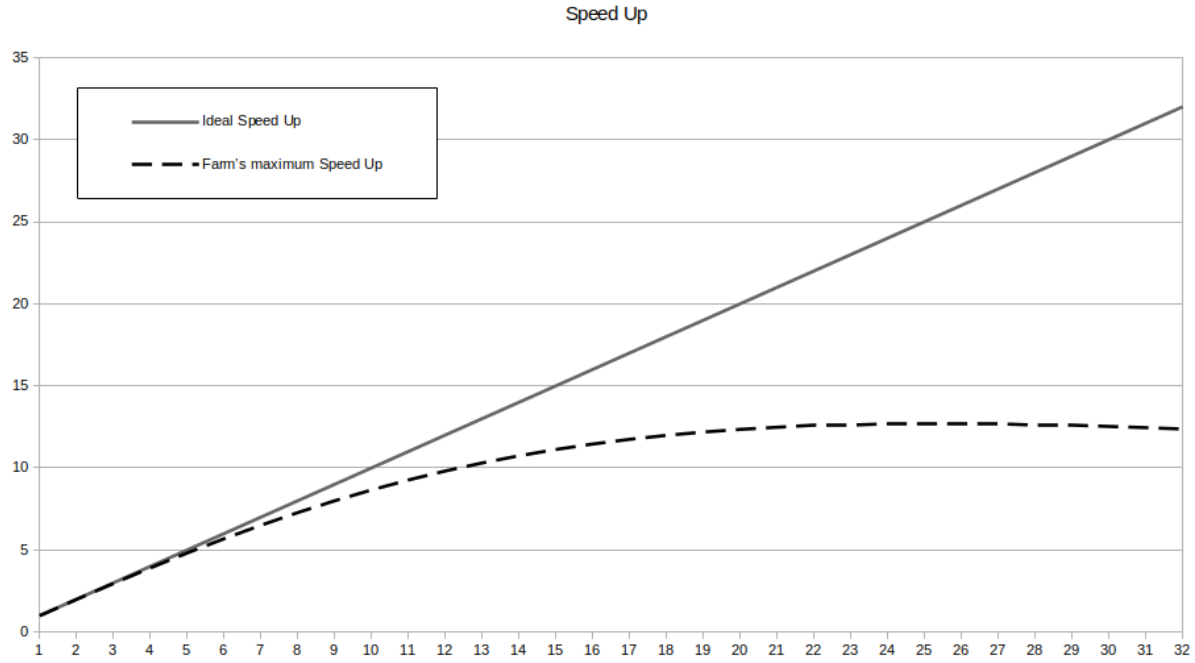

Figure 2: Maximum speed up achievable, n=2048, N=16

Moreover there are other sources of overhead that I have not taken into account yet, like: the load balancing policy's issues, the false sharing problem and more in general the cache coherency protocol, the migration of thread to other cores, the allocation/deallocation of memory from the global heap and so on. Therefore, I expect that the real speed up will be quite lower than the one plotted on figure 2.

1.2 Analysis of model performance

Now that we have outlined the solution to parallelise the Jacobi method, let us analyze better this solution and its possible main issues.

Firstly, let's consider which could be the achievable speed up of our model, when the linear system's dimension grows. 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, and if we omit the operations required to fork-join threads and to orchestrate them (which become negligible when N is particularly high), 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, with these assumptions, 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, a map could achieve almost a linear speed up when N is particularly large. This is a direct consequence of the fact that the sequential fraction goes to 0 when N becomes large. Therefore, despite the fact in the previous example the map's maximum speed up was not very close to the ideal one; the map's maximum speed up should become almost linear when N is very large.

Now let's consider which can be the other main sources of overhead that could let plummet the speed up of this parallel program. First of all, we have to remember 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 iteration. 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 work 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 seems to be a crucial problem for this project. Other sources of overhead should not be important as the ones that I have already underlined in this first section of this relation, therefore it is likely that if we find a proper load balancing policy, we could measure a speed up pretty close the maximum speed up reachable using a map parallel pattern.

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 4 versions of the Jacobi method use the same code to instantiate the linear system, this code is provided by the files `utils.h` and `utils.cpp`. 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. 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 initialize the linear system has never been measured. `my_timer.cpp` has been created using the Chrono library. If specified by the user, all the 4 versions of the Jacobi method can compute the stopping criterion at line 12 of Algorithm 1, the function that computes the stopping criterion is provided by the files `utils.h` and `utils.cpp`. Finally, `seq_jacobi.cpp`, `par_jacobi.cpp` and `par_jacobi2.cpp` can print some data about their execution time if specified by the user.

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. For more details, look at the function `par_jacobi()`.

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 uses a condition variable and a mutex for the synchronization of the threads.

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.

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 initialization of threads and their orchestration 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. Due to the reasoning of section 1, the maximum speed up achievable will be pretty lower than the linear speed up when we have to deal with a smaller linear system, and pretty close the linear speed up when the system is larger. Furthermore, 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 speed up close to the maximum achievable due to the fact that this algorithm uses a static policy and therefore it 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 and the time needed at each iteration to refill the queue with new subtasks, which is not present in the other versions of the Jacobi method. Due to the shared queue, I expect that the performance 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 I will plot also the maximum speed up achievable, which is bounded by the necessity to initialise threads and to notify them.

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 2048×2048 linear systems, performing 32 Jacobi iterations, while for the second experiment I used 16384×16384 linear systems, performing 256 Jacobi iterations.

4.1 Experiment 1

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

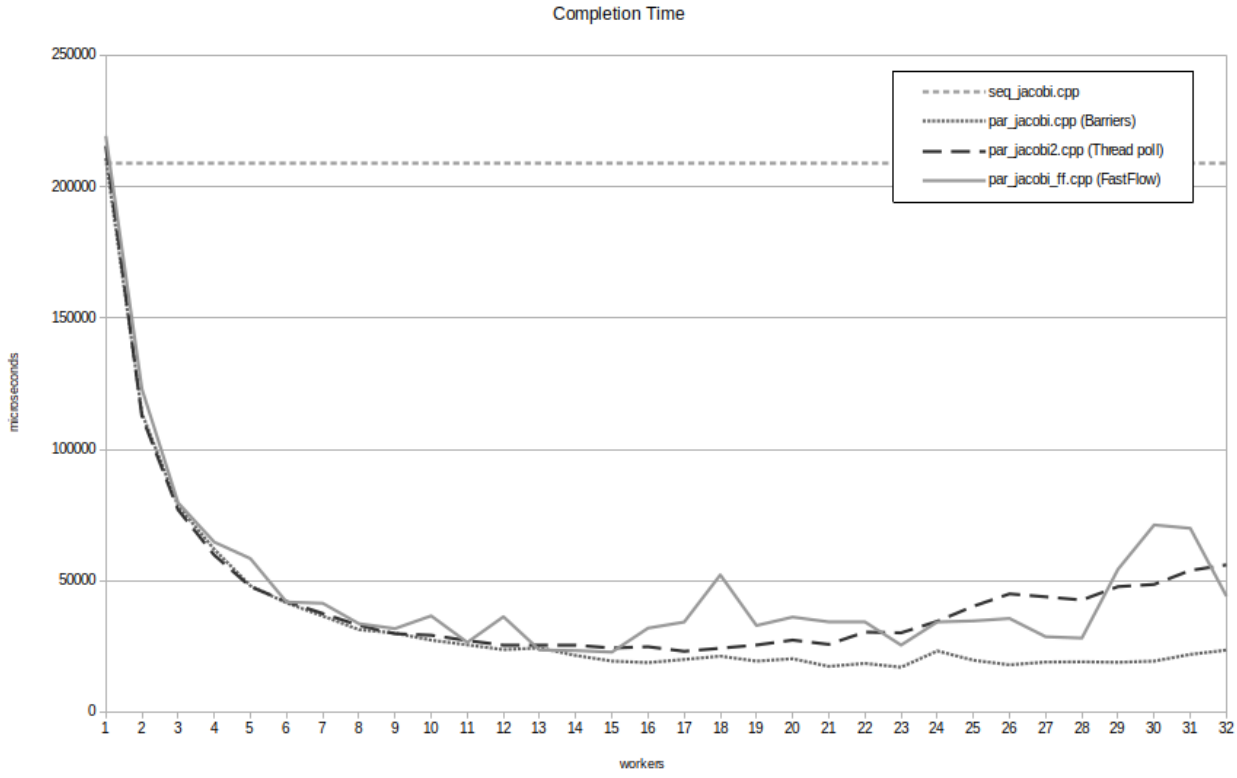

Figure 3: Elapsed time, N=2048, M=32.

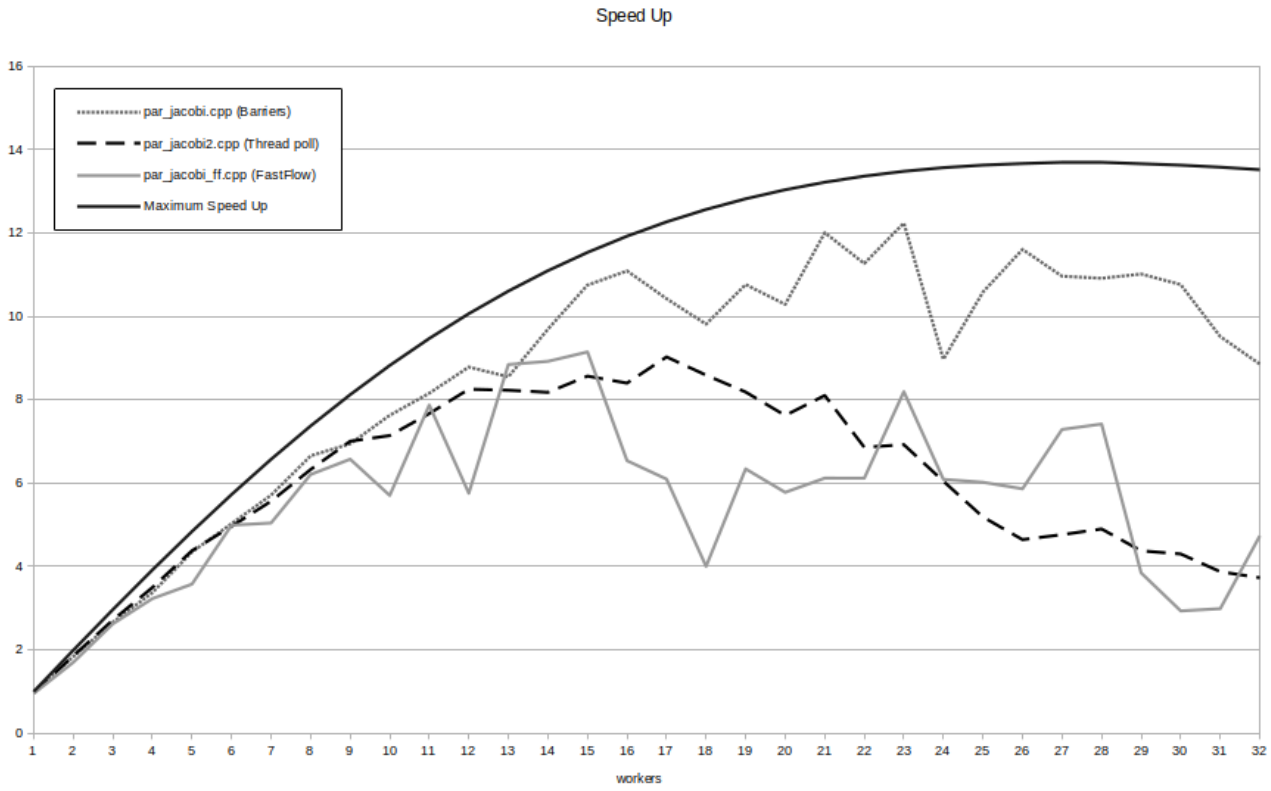

Figure 4: Speed up, N=2048, M=32.

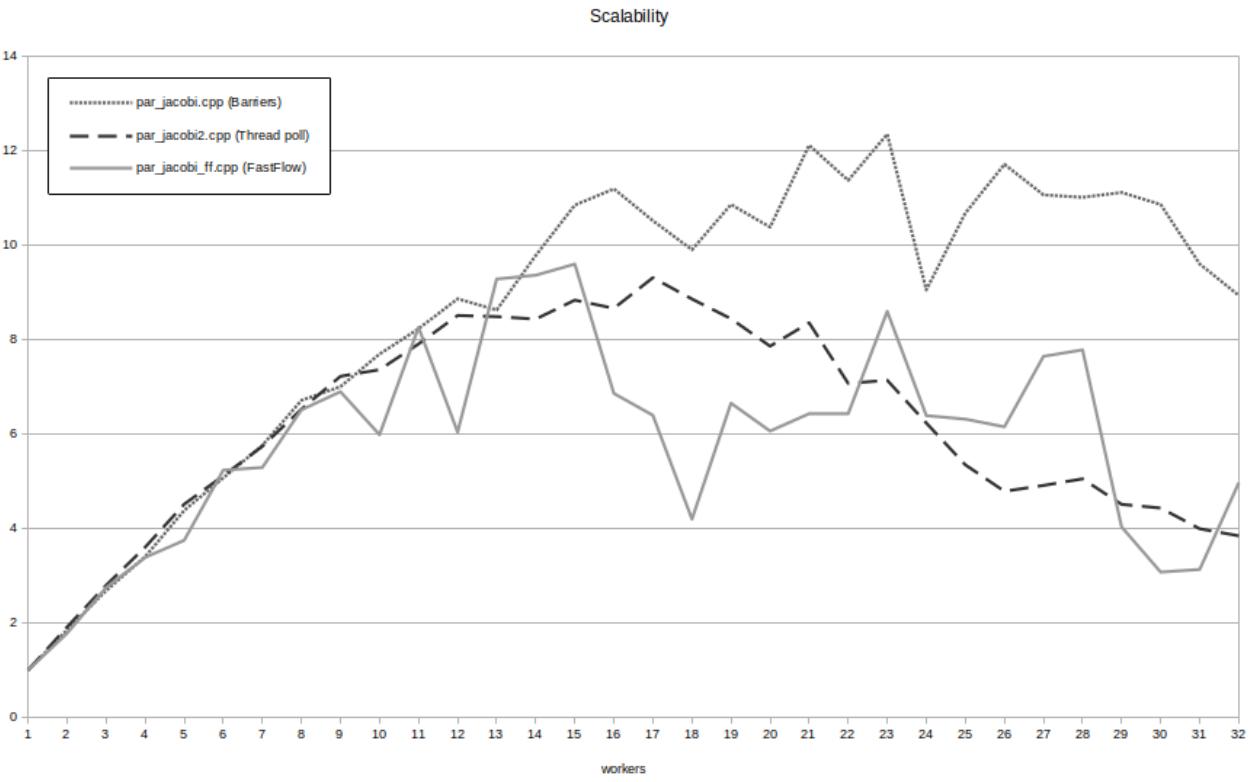

Figure 5: Scalability, N=2048, M=32.

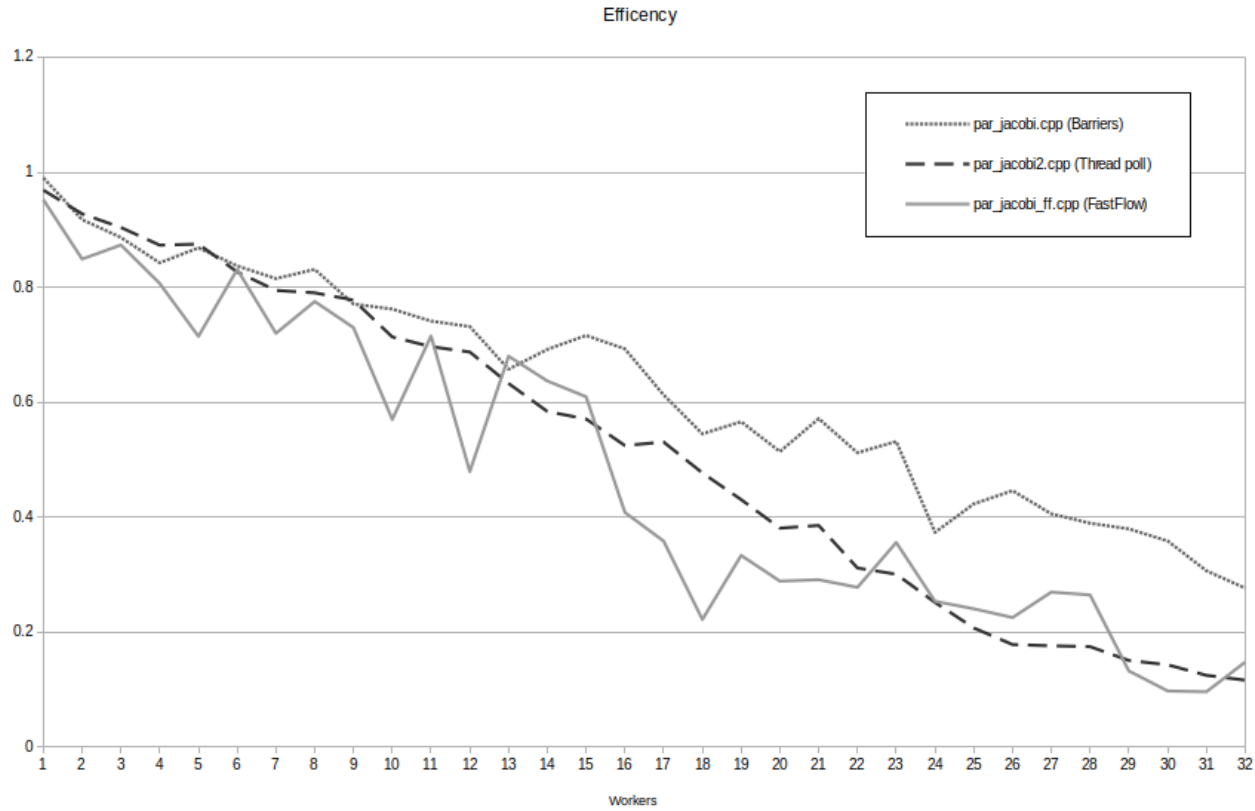

Figure 6: Efficency, N=2048, M=32.

Comments

In this first experiment the maximum speed up attainable was pretty lower than the linear speed up, indeed, following the same reasoning of section 1 of this relation, I computed that the maximum speed up achievable is about 13.7 when we use 27 workers and it decreases when we the number of workers increase. `par_jacobi.cpp` (barriers) achieved a speed up pretty close to the maximum speed up attainable, it peaked at 23 workers, reaching a speed up of 12.3. On the other hand, the other two versions performed worse; `par_jacobi2.cpp` (thread pool) achieved a speed of 9.0 using 17 workers, while `par_jacobi_ff.cpp` (FastFlow) achieved a speed up of 9.1 using 15 workers. Moreover, the performance of these two latter methods started to drop sharply when the number of workers became too large; in `par_jacobi2.cpp` that was probably caused by the presence of the shared queue. Another remarkable fact is that the efficiency decreased steadily in all the three methods as the number of workers grew, plummeting to about 0.1 in `par_jacobi2.cpp` and to 0.3 in `par_jacobi.cpp`, when the programs used 32 threads.

4.2 Experiment 2

Now let's see how these results change when the linear system is 16384×16384 and the number of iterations is 256. Below there are all the graphs related to this second experiment.

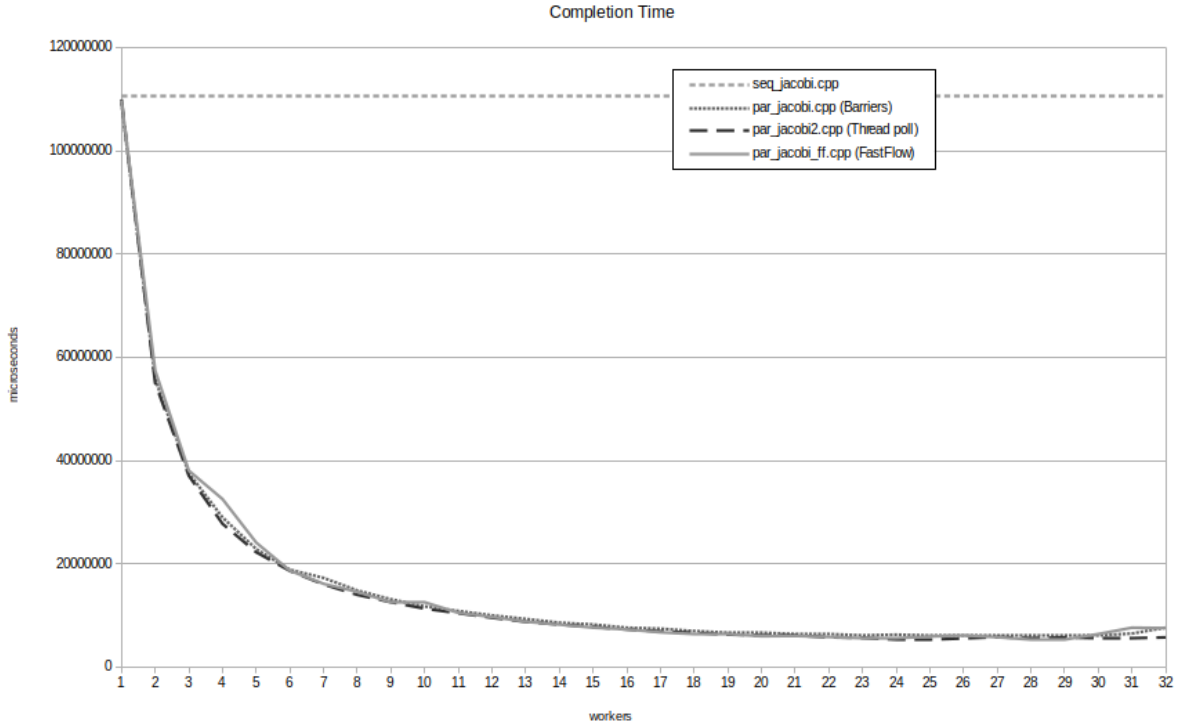

Figure 7: Elapsed time, N=16384, M=256.

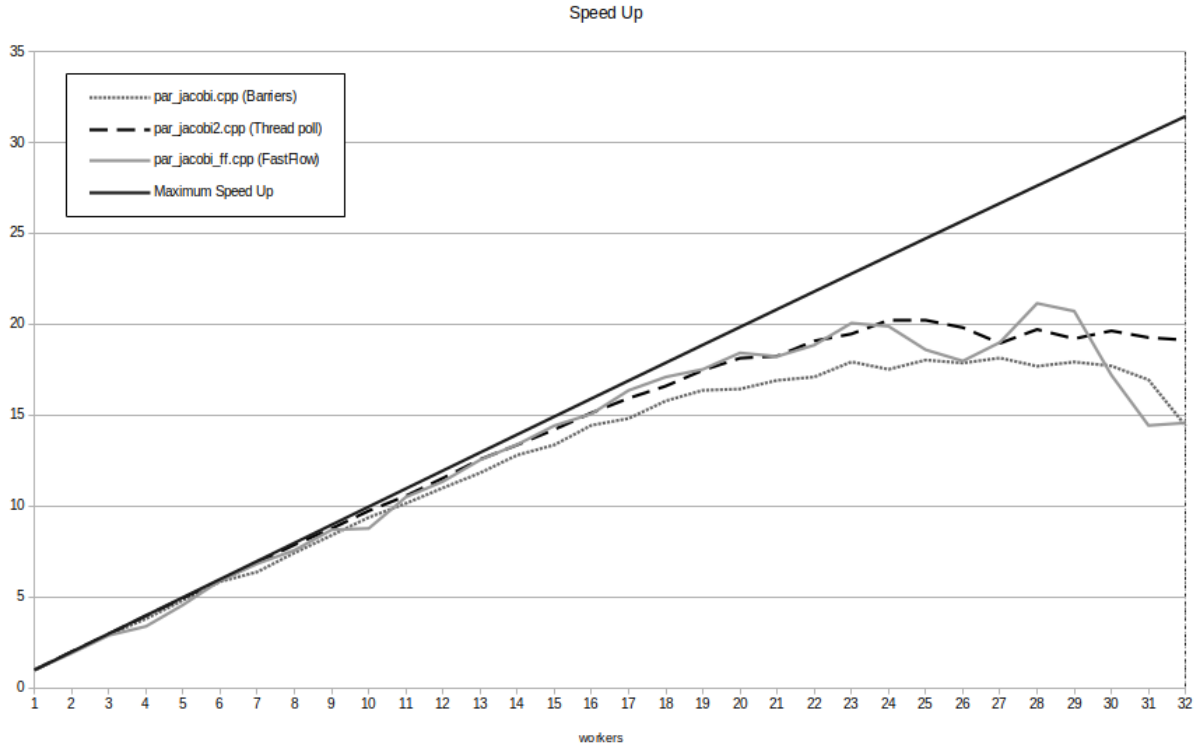

Figure 8: Speed up, N=16384, M=256.

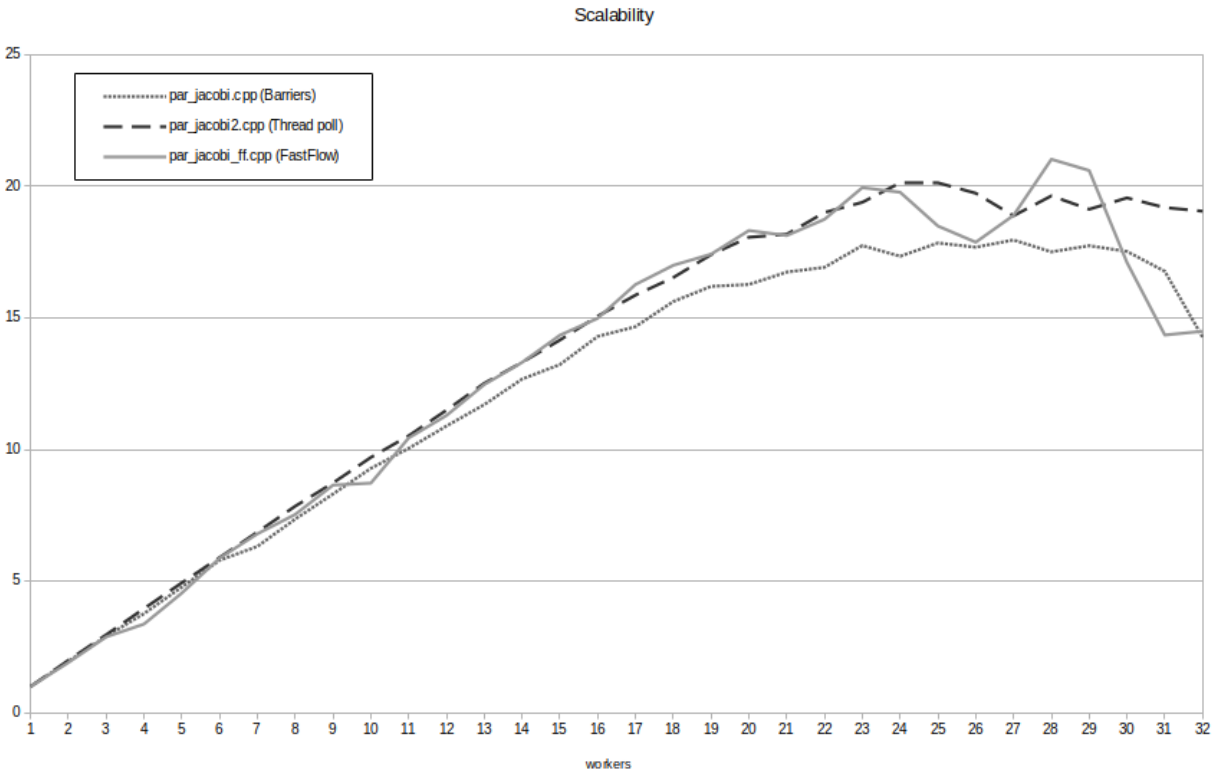

Figure 9: Scalability, N=16384, M=256.

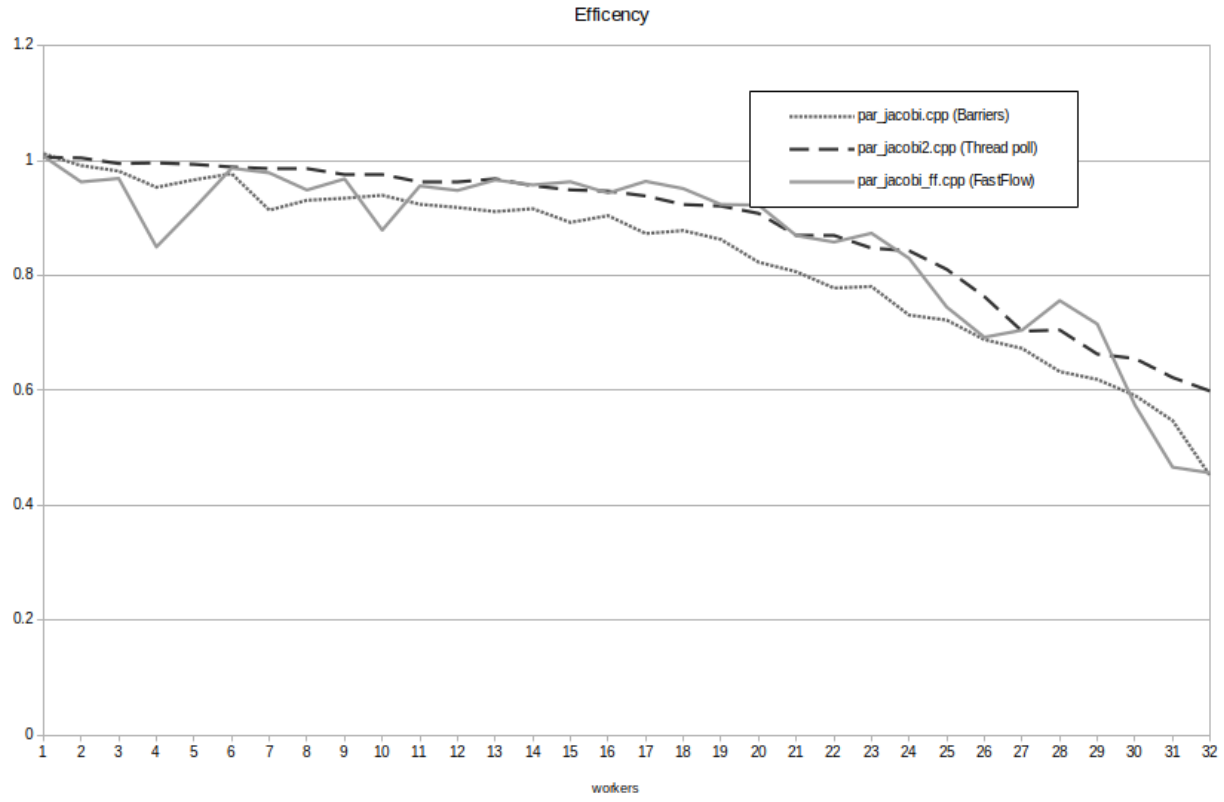

Figure 10: Efficiency, N=16384, M=256.

Comments

In this second experiment the cost to initialize and to notify threads revealed to be almost negligible, indeed, this time the maximum speed up that we can hope to achieve is 31.4 using 32 threads. the first noteworthy aspect of this second experiment is that `par_jacobi2` (thread pool) and `par_jacobi_ff` (FastFlow) perfomed more or less the same, while `par_jacobi` (barriers) was slower than the other two versions. `par_jacobi_ff` (FastFlow) achieved the highest speed up among them, achieving a speed up of 21.2 using 28 threads, while `par_jacobi2` (thread pool) achieved a speed up of 20.2 using 25 threads. These values are pretty lower than the maximum speed up attainable, therefore probably other sources of overhead slowed down the computations. In this experiment the dynamic load balancing policies proved to be faster, indeed it is likely that `par_jacobi` (barriers) paid the fact it had to pre-assign the subtasks among its threads. Finally, this time the efficiency remained more or less constant in the first half of the experiment, while it plummeted to about 0.5 when the number of workers was, probably because some threads had to share the same core.

5 Analysis of the results

In this section of the relation I am going to analyze in detail the execution of `par_jacobi` (barriers) and `par_jacobi2` (thread pool). In the former program, I want to study how much its static load balancing policy affected its performance, while in the latter I want to figure out how much overhead was introduced by the presence of the shared queue. In both programs is possible to print some data about their execution passing the argument stats == 1.

5.1 par_jacobi.cpp (barriers)

Firstly, I have measured the time needed to initialize a barrier and I figured out that a barrier can be created using about 5 μsec, therefore it can be considered negligible in our experiments. In the first two graphs below, I plotted the average time spent by the threads to execute tasks compared to the average time spent by the threads to wait the slowest thread on the barrier at the end of each Jacobi iteration. The first graph is related to the first experiment (N=2048, M=32), while the second graph to the second experiment (N=16384, M=256).

Figure 11: Execution time versus waiting time, N=2048, M=32. y-axis on a logarithmic scale

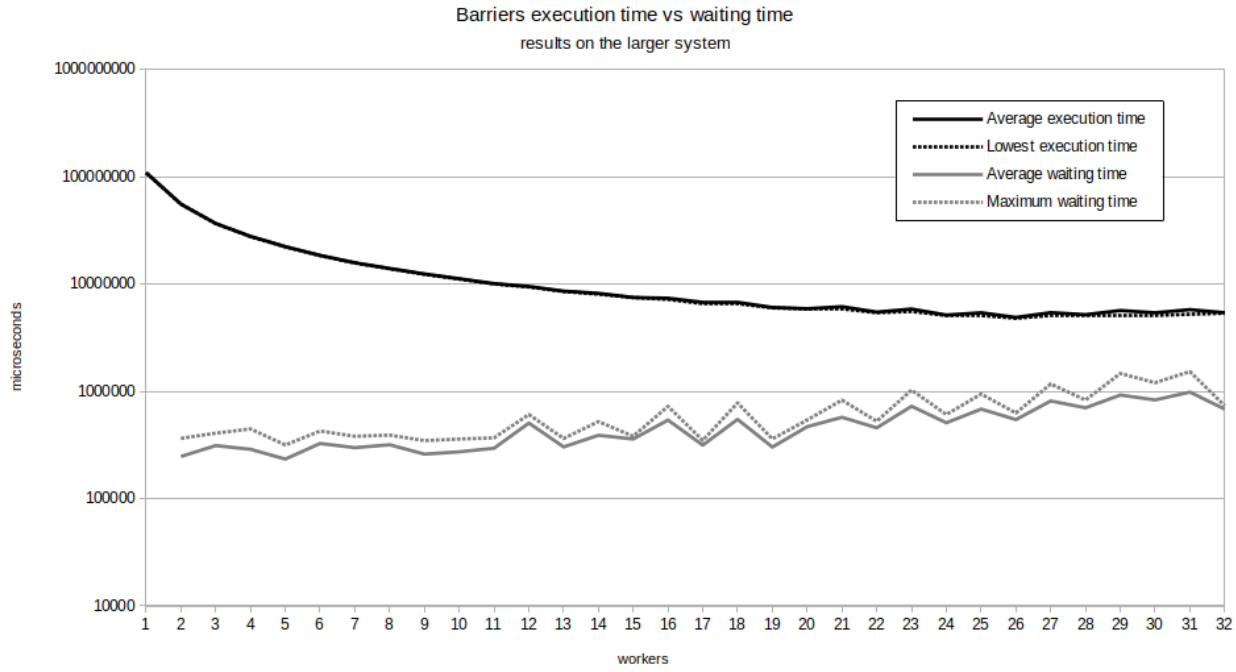

Figure 12: Execution time versus waiting time, N=16384, M=256. y-axis on a logarithmic scale

As we can see, in both experiments the wasted time caused by the necessity to wait the slowest thread on the barrier was absolutely non-negligible. The average waiting time on the barrier soared as the number of workers grew, and especially on the smaller system, it became almost higher than the execution time when the number of workers involved was around 30 units. In the graph below I plotted the ratio between the average time spent by the threads to execute tasks and the total time (waiting time + execution time).

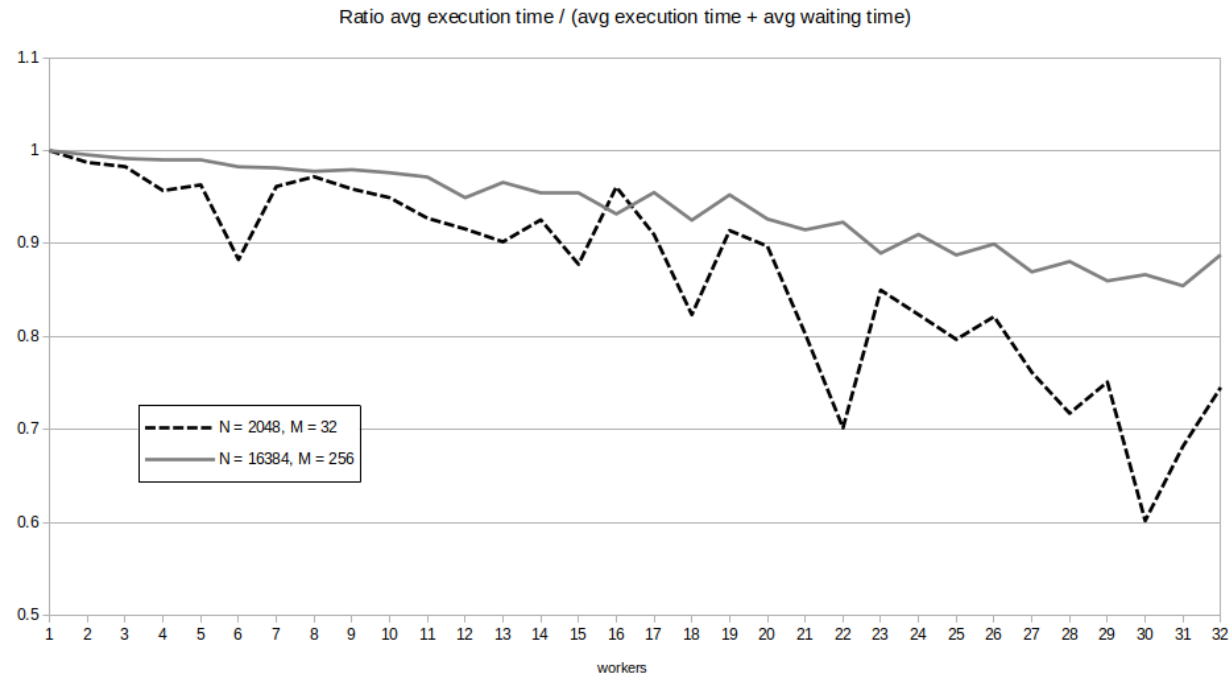

Figure 13: Ratio between execution time and total time

from the graph above, it is clear that in both experiments the performance of `par_jacobi.cpp` (barriers) has suffered from its cyclic load balancing policy, this is the main reason why `par_jacobi.cpp` (barriers) was outclassed by the two dynamic implementations of Jacobi in the second experiment.

5.2 par_jacobi2 (thread pool)

In the first two graphs, I plotted the average time spent by the threads to execute tasks and the average time spent by the threads to access the shared queue. The first graph is related to the first experiment (N=2048, M=32), while the second graph to the second experiment (N=16384, M=256).

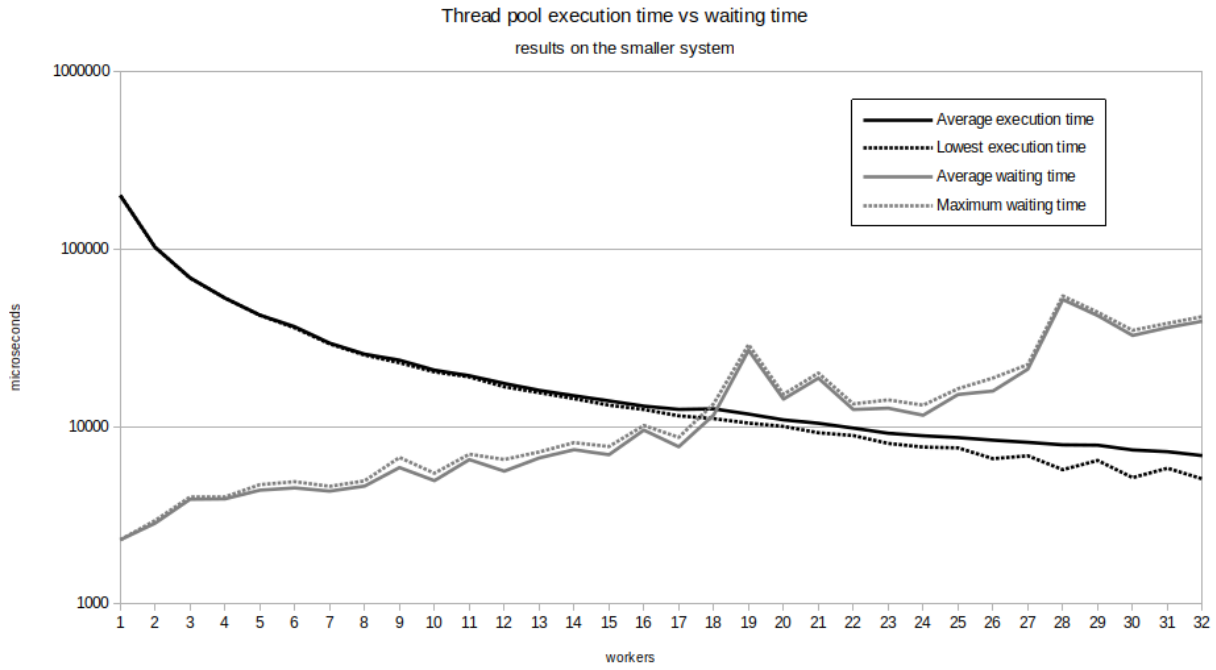

Figure 14: Execution time versus waiting time, N=2048, M=32. y-axis on a logarithmic scale

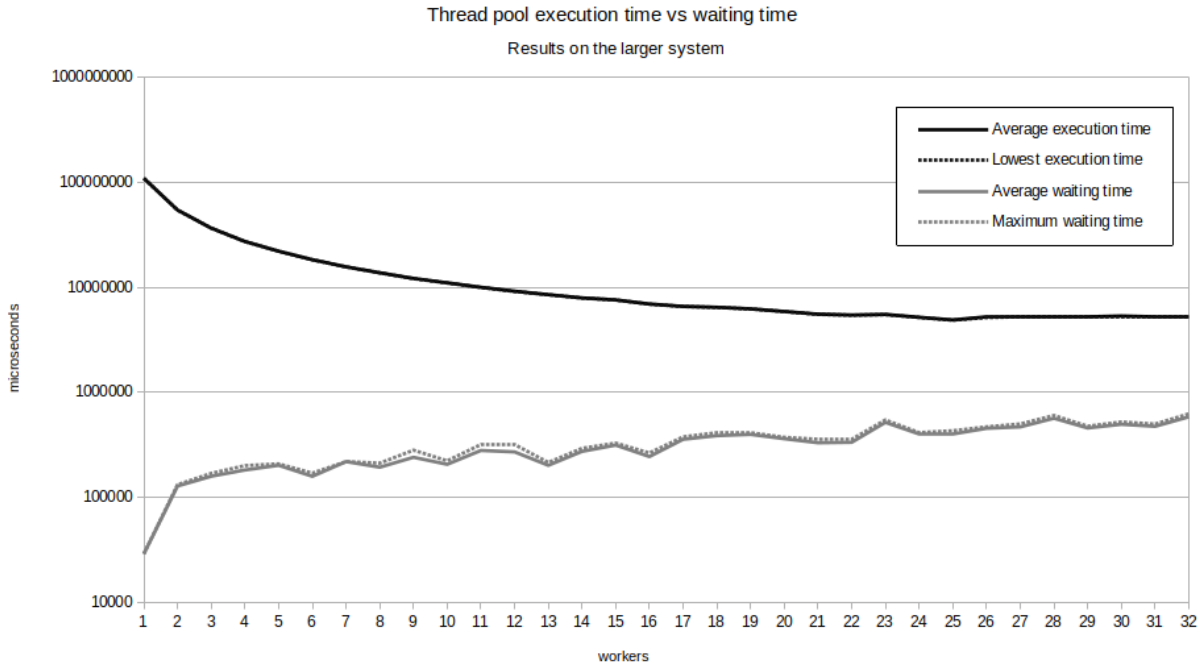

Figure 15: Execution time versus waiting time, N=16384, M=256. y-axis on a logarithmic scale

As I expected, in both cases the time wasted trying to access the shared queue soared drastically as the number of workers grew. This was particular true in the case of the smaller system, where the threads started to spend more time on the shared queue rather than to compute subtasks. In the graph below I plotted the ratio between the average time spent by the threads to execute tasks and the total time (waiting time + execution time).

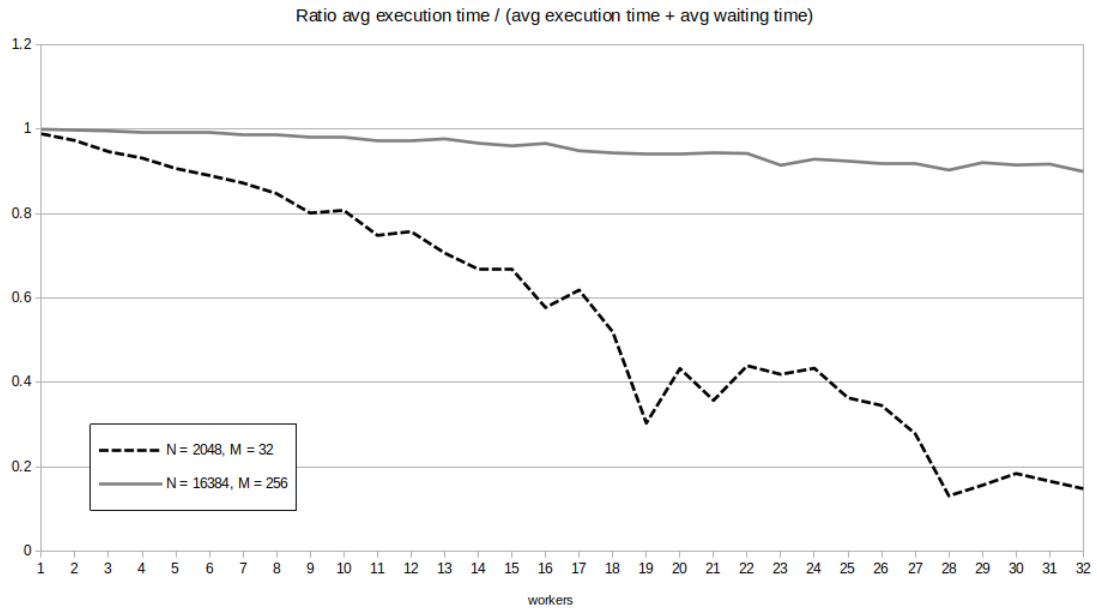

Figure 16: Ratio between execution time and total time

In the last chart I analyzed the time spent by the main thread to refill the shared queue. This is a source of overhead which is not present in the other programs, therefore I wanted to analyze how much it can impact the performance of the algorithm.

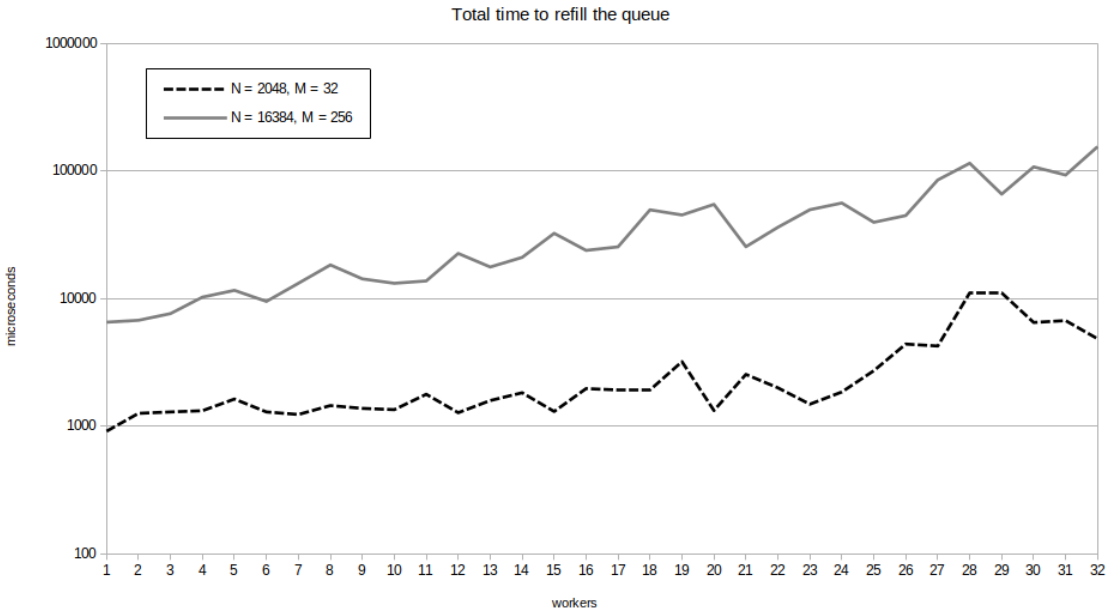

Figure 17: Microseconds spent to refill the shared queue, y-axis on a logarithmic scale

This last chart shows that the time required to refill the queue was not particular high compared to the completion time. This time soared as the number of workers grew, due to the necessity to orchestrate the threads.

6 Conclusions

In conclusion, the static load balancing policy revealed the be the best option in case of small system, indeed here the static strategy of pre-allocation achieved a speed up pretty close to the maximum achievable. Anyway, when the dimension of the system increased, the two dynamic implementations of the Jacobi method had better performance than the version with barriers, achieving a higher speed up. Probably with a even larger linear system, the two dynamic versions of the Jacobi method would have performed even better, outclassing their static version counterpart.