Project of Distributed Systems: Paradigms and Models
Parallel Iterative Jacobi Method

Student: Diego Arcelli
Student ID: 647979
Professor: Marco Danelutto

Academic Year 2021-2022

UNIVERSITÀ DI PISA

Contents

1 Analysis of the problem

Notation premise: with $x_i^{(k)}$ we indicate the i-th entry of vector x at iteration k. The pseudo-code to execute the Jacobi method sequentially, for a fixed amount of iterations, is shown in 1. Note that we are forced to use an auxiliary vector x' to compute the updated elements of x for the current iteration, and then, before starting the following iteration, we copy the elements of x' in x. This is necessary because if at iteration k we compute $x_1^{(k)}$ modifying x in-place, then when we'll compute $x_2^{(k)}$, instead of using $x_1^{(k-1)}$, we'll use $x_1^{(k)}$.
The algorithm is composed by three nested loops, where the first iterates *limit* times, and the other two iterate n times, so the time complexity is $O(limit \cdot n^2)$. The *swap* function used to copy the elements of x' in x, can be implemented in $O(1)$ time complexity, by swapping the pointers of the two arrays.

Algorithm 1 Sequential code for Jacobi method

Require: A strictly diagonally dominant matrix, b vector, *limit* positive integer

```
1:  for k ← 1 to limit do
2:      for i ← 1 to n do
3:          x'[i] ← 0
4:          for j ← 1 to n do
5:              if j ≠ i then
6:                  x'[i] ← x'[i] − A[i, j] ∗ x[j]
7:              end if
8:          end for
9:          x'[i] ← (x'[i] + b[i])/A[i, i]
10:     end for
11:     swap(x, x')
12: end for
```

Let's now analyze how we can exploit the properties of the algorithm in order to parallelize it. The first important thing we notice is that in order to compute the updated values for the vector x at iteration k, we need the full updated vector x at iteration $k-1$, therefore the iterations of the algorithm need to be strictly sequential.
A relevant property of the algorithm that we can notice from the pseudo-code, is that each element $x_i^{(k)}$ is computed independently from the others elements of the vector $x^{(k)}$, since to compute $x_i^{(k)}$ we just need the old vector $x^{(k-1)}$ and the matrix A. Therefore the iterations of the loop at line 2 of the pseudo-code, can be executed in parallel, which is the classical application of a map pattern. So, if we have nw workers available, we can assign to each worker a partition of indices of x to compute.
The innermost loop at line 4 is used to compute a sum of $n-1$ elements, which is the classical application of a reduce pattern. This is done inside the loop that we already said we can parallelize using a map pattern, so if we want to implement also the reduce, for each one of the nw workers of the map, we have to arrange other nw' additional workers that can be used to compute in parallel the summation.
The introduction of a reduce inside each map would increase the overhead, since each worker assigned to the map would have to coordinate its computations with the workers that it uses for the reduce. Moreover, if the additional $nw \cdot nw'$ workers arranged for the reduce, are just used to increase the parallel degree of the map, we would obtain the same theoretical speedup (this is shown in the time analysis section) but which much less overhead. Therefore I decided to do not implement the reduce.
Even if we only implement the map, there some synchronization problems that we need to care

about:

1. Start iteration $k+1$ only after all the workers finished their computations for iteration k, since the iterations must be strictly sequential

2. Executing the swap of the main and auxiliary vectors only after all the workers finished their computations for the current iteration, otherwise it might happen that the vectors are swapped while some workers are still computing their portion of $x^{(k)}$

3. Keep track of the number of iterations so that all the workers will stop after *limit* iterations

The first problem can be solved simply by putting a barrier after the execution of the map, so that all the worker will be forced to wait the others before starting the next iteration. In this way we can also solve the third problem by selecting one of the worker to count the number of iterations of the algorithm executed so far. The designated worker can simply use a shared variable among the workers, that it will increase by one before the barrier. Since the variable is modified by just one worker it doesn't need synchronization mechanism to be accessed, even if it is shared. This solution guarantees that before starting iteration k the shared variable will have value k for all the workers, so that every worker can individually check if $k = limit$ and stop its execution if it is the case. To solve the second problem we need to introduce a second barrier, which has to be placed between the end of the map an the previously defined next iteration barrier. Between this new defined barrier and the next iteration barrier, one of the workers will be selected to execute the swap (it can be the same one that counts the iterations). In this way the swap will happen only after every worker finished its computations for the current iteration and before starting the next iteration.
The pseudo-code of the routine executed by a worker is showed in 2.

Algorithm 2 Worker pseudo-code

Require: A strictly diagonally dominant matrix, b vector, *limit* positive integer

```
1:  while k < limit do
2:      for i ← start to end do
3:          x'[i] ← 0
4:          for j ← 1 to n do
5:              if j ≠ i then
6:                  x'[i] ← x'[i] − A[i, j] ∗ x[j]
7:              end if
8:          end for
9:          x'[i] ← (x'[i] + b[i])/A[i, i]
10:     end for
11:     Vectors swap barrier
12:     if start = 0 then
13:         swap(x, x')
14:         k ← k + 1
15:     end if
16:     Next iteration barrier
17: end while
```

The variables *start* and *end* represent respectively the first and the last index of x assigned to the worker. The worker that will update the iterations counter and execute the swap of the vectors is the one assigned to the leftmost chunk (the one with $start = 0$). Note that both of

the barriers are needed because if we only use the copy barrier, it could happen that one of the workers might start the next iteration before the designated worker could swap the vectors and update the iteration counter, so both barriers are needed.

2 Time analysis

In the sequential case, the time required to execute the algorithm is:

$$T_{seq} = T_{init} + k\Big(n^2 T_{\oplus} + T_{swap}\Big)$$

where k is the number of iterations, n is the number of rows and columns of the matrix, T_{init} is the time required to initialize the computations, $T_{\oplus}$ is the time required to compute the expression at line 6 of 1 and T_{swap} is the time required to perform the swap of the vectors. T_{init} includes the declaration of some variable and the creation of the auxiliary array, an operation whose cost depends by n. If we consider negligible the initialization and the swap times, then the execution time is more or less:

$$T_{seq} \approx kn^2 T_{\oplus}$$

In the parallel case, the loop at line 2 of 1 is parallelized, so, with nw workers available, the program will execute at the same time nw parallel loops, each of which will iterate more or less $\frac{n}{nw}$ times. The overhead introduced is given by the forking and joining of the threads, by the time required to compute and assign tasks to the workers and by the time required to manage the barrier, so overall we have that the parallel time is:

$$T_{par}(nw) = T_{init} + T_{split} + T_{fork} + k\Big(\frac{n}{nw} n T_{\oplus} + T_{synch} + T_{swap} + T_{synch}\Big) + T_{join}$$

where T_{fork} and T_{join} are the times for the fork-join mechanism, T_{split} is the time required to compute and split the tasks among the workers and T_{synch} is the time required to handle the barrier. Again, if we consider negligible the times required for the preparations, the synchronization and the swap, then the parallel time is approximately:

$$T_{par}(n) \approx k\frac{n}{nw} n T_{\oplus}$$

If we had decided to implement the reduce too, assigning nw_1 workers to the map and nw_2 workers to each worker of the map for the reduce, then the parallel time would have changed as follows:

$$T'_{par}(nw) \approx k\Big[\frac{n}{nw_1}\Big(\frac{n}{nw_2} T_{\oplus} + nw_2 T_{\oplus}\Big)\Big] = k\frac{n}{nw_1}\frac{n}{nw_2} T_{\oplus} + knw_2 T_{\oplus} \approx k\frac{n}{nw_1}\frac{n}{nw_2} T_{\oplus}$$

since for each map we could have computed in parallel nw_2 partial sums of $\frac{n}{nw_2}$ elements each, and then we would have computed the final sum summing up the nw_2 partial sums. If in T_{par} we set $nw = nw_1 \cdot nw_2$, then the time is the same of T'_{par}. Actually if we don't exclude from T'_{par} the time needed to compute the partial sums, we also have the term $knw_2 T_{\oplus}$, and so T_{par} is even better than T'_{par}. From this analysis we draw the conclusion that if we only implement the map, with the same resources we get an even better theoretical execution time than implementing both the map and the reduce. Furthermore the implementations of the reduce would introduce overheads and more synchronization problems that are not present if we only implement the map.
The speedup we can achieve is:

$$Speedup(nw) = \frac{T_{init} + k\Big(n^2 T_{\oplus} + T_{swap}\Big)}{T_{init} + T_{split} + T_{fork} + k\Big(\frac{n}{nw} n T_{\oplus} + T_{synch} + T_{swap} + T_{synch}\Big) + T_{join}}$$

If we exclude the negligible times then we get:

$$Speedup(nw) \approx \frac{kn^2 t_{\oplus}}{k\frac{n}{nw} n t_{\oplus}} = nw$$

which is the ideal speedup which we can theoretically obtain with a map pattern. Of course to have a better estimate of the speedup we should consider the serial fraction of the program. We can make a rough estimation by considering that the total number of operations done by the algorithm is more or less in the order of kn^2. since we have to do some fixed amount of operations inside the three loops. Ideally, With n workers, every worker could compute its own $x_i^{(k)}$ in more or less n operations, reducing the number of operations to kn. Therefore a rough estimation of the serial fraction is:

$$f = \frac{kn}{kn^2} = \frac{1}{n}$$

if we now consider the speedup given by the Amdahl's law, we have that in the best case the maximum speedup is:

$$Speedup < \frac{1}{f} = \frac{1}{\frac{1}{n}} = n$$

So the speedup is upper-bounded by the number of rows/columns of the matrix. This makes sense, since in an ideal scenario, we would have a worker for each component of the vector x, reducing the cost of a single iteration of the algorithm from $O(n^2)$ to $O(n)$. So we should expect a better speedup as n increases.

3 Implementation details

The parallel program has been implemented in three versions:

- The first using native C++ threads
- The second using FastFlow
- The third using OpenMP

The program has been implemented defining a class called `Jacobi` which has as attributes the coefficients matrix A, the vector of known terms b and an integer n that is the size of A and b. For each implementation of the algorithm (the sequential and the three parallel ones) there's a corresponding method to solve the linear system $Ax = b$, using that specific implementation, by providing as parameters the number of iterations of the algorithm and the number of workers (for the parallel versions). For all the implementations the solution vector x is initialized setting all the elements to 0, and to assure that algorithm will converge, A it's generated randomly to be strictly diagonally dominant. tends to

3.1 C++ threads

In the implementation with native C++ threads I used the fork-join mechanism to spawn nw threads, each of them is assigned to a contiguous chunk of x of size $\lfloor \frac{n}{nw} \rfloor$ according to the following procedure:

$\Delta \leftarrow \lfloor \frac{n}{nw} \rfloor$
for $i \leftarrow 0$ to nw **do**
    $start_i \leftarrow i * \Delta$
    $end_i \leftarrow start_i + \Delta - 1$

end for
$end_{nw} \leftarrow n - 1$

Then each thread will execute code identical to the one in 2. For the implementation of the barrier I used the `barrier` class introduced in C++20.

3.2 FastFlow

For the FastFlow implementation I used the `ParalleFor` class to parallelize the loop at line 2 of 1. An object of class `ParalleFor` is created before the loops start, in order to prepare the nw workers in advance, and then we call the `parallel_for` method, inside the main loop of of the algorithm. The function passed to the parallel:

function PARALLEL_FOR_ITERATION(i)
  $x'[i] \leftarrow 0$
  for $j \leftarrow 0$ to $n - 1$ **do**
    if $i \neq j$ **then**
      $x'[i] \leftarrow x'[i] - A[i, j] * x[j]$
    end if
  end for
  $x'[i] \leftarrow (x'[i] + b[i])/A[i, i]$
end function

After the call of the parallel for method, the counter of the loop is increased and the vectors are swapped, and this is done sequentially by the main thread of the program. Note that we do not need to explicitly define the barrier since the call of the `parallel_for` method is an implicit barrier for the workers.

3.3 OpenMP

For the OpenMP implementation I simply parallelized the second innermost loop at line 2 of algorithm 1, using the `#pragama omp paralle for` directive, with the default static scheduling, which divides the loop in nw chunks of equal size. Like in the case of FastFlow, the end of the parallelized for is an implicit barrier for the threads, so we do not need to define the barrier explicitly.

4 Manual

The source code, the scripts written for the experiments and a detailed manual of the program are available at this repository.
To compile the code is sufficient to execute the command `make` in the `src` directory of the project. The compilation will produce the executable `main` which can be launched to execute the program, providing the following arguments:

- `n`: the number of row and columns of the matrix
- `nw`: the number of workers that will be used in the parallel versions of the algorithm
- `iters`: the number of iterations of the algorithm
- `mode`: an integer number which specifies which implementations of the algorithm will be executed

The possible values of `mode`, with the corresponding effects, can be showed by launching the program with no argument. Once executed the program will run the versions of the algorithm required (according to the value of `mode`), and it will print their execution times.

5 Experiments

The main goals of the experiments that we will execute are the following:

- Comparing the performances obtained by the three implementations
- Check how the performances change with respect to the size of the matrix
- Comparing the actual performances to the ideal ones

To compare the performances of the three implementations we'll measure the times obtained when executing the algorithm with nw workers, varying nw from 1 to 32 (which is the number of cores of the machine used for the experiments). Then from the times we'll derive the speedup and the efficiency of the three implementations. To see if the size of the matrix impact on the performances, we'll repeat this procedure twice, first on 1000×1000 matrix and then on a 10000×10000 matrix.
After that we'll execute the algorithms with a fixed number of workers on matrices of different sizes (500, 1000, 5000, 10000, 15000, 20000) to better understand the impact that the size of the matrix has on the performances.

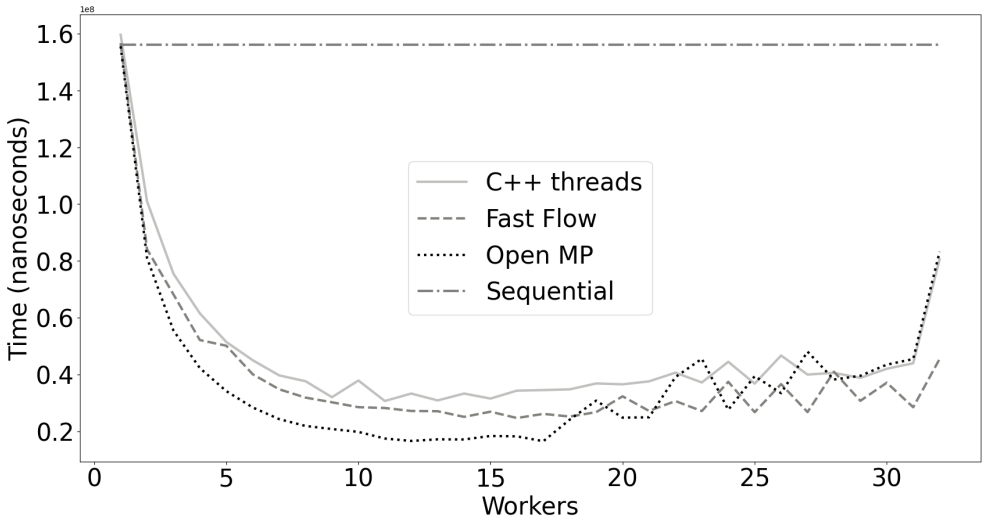

Figure 1: Time vs workers for matrix size 1000

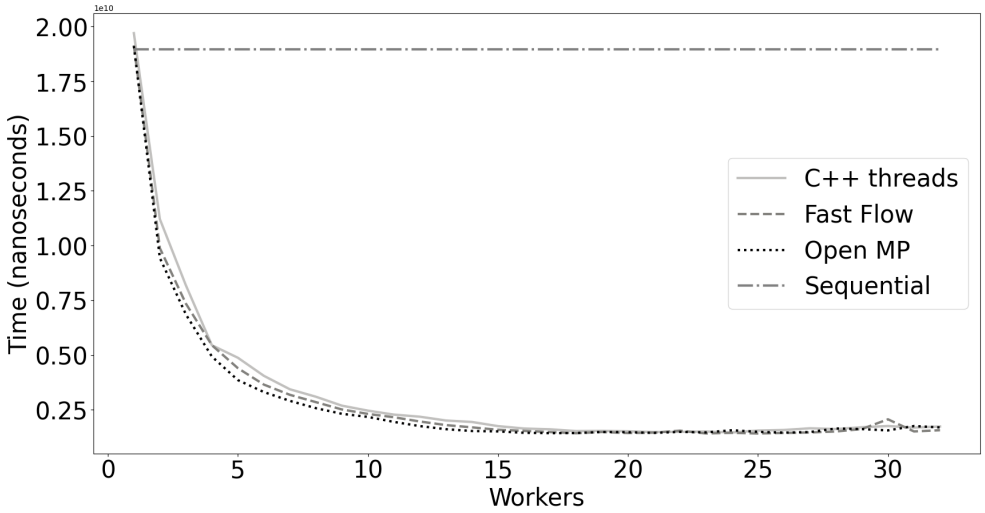

Figure 2: Time vs workers for matrix size 10000

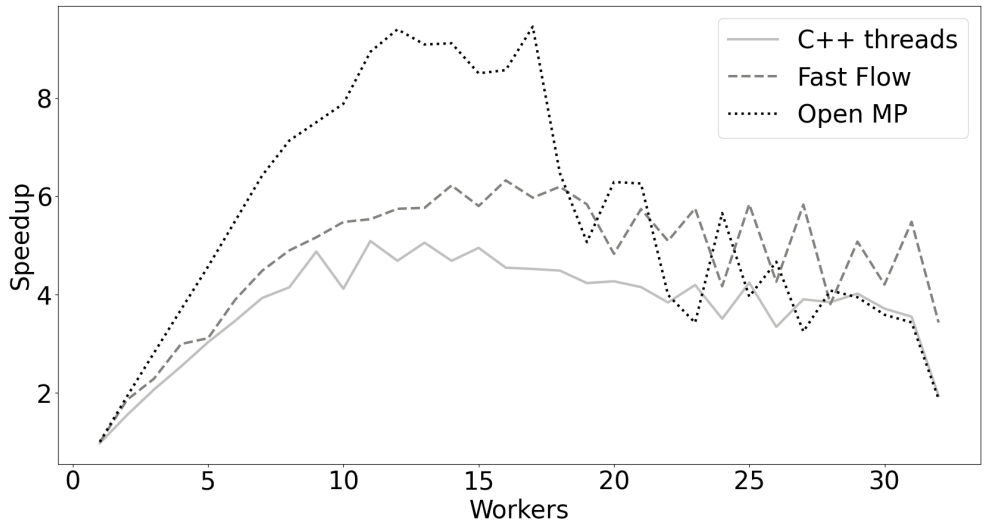

Figure 3: Speedup vs workers for matrix size 1000

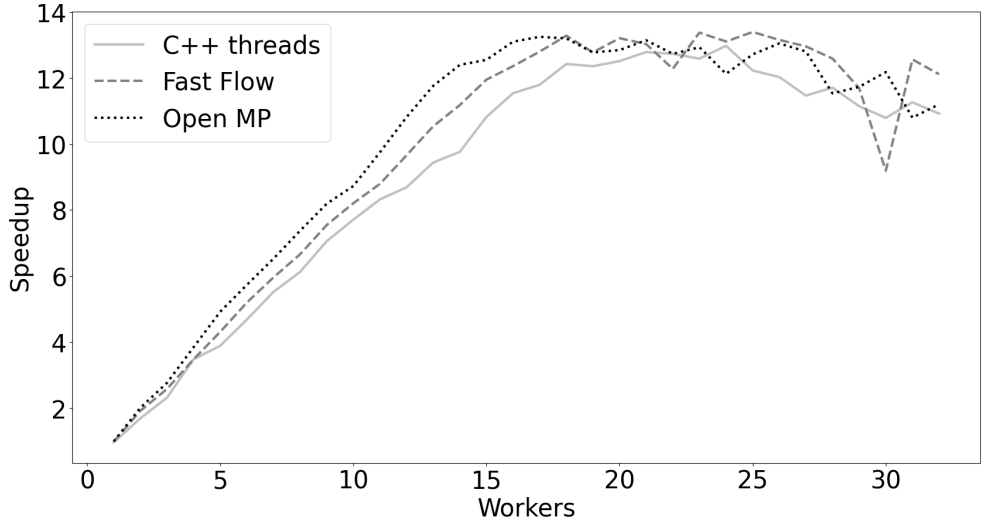

Figure 4: Speedup vs workers for matrix size 10000

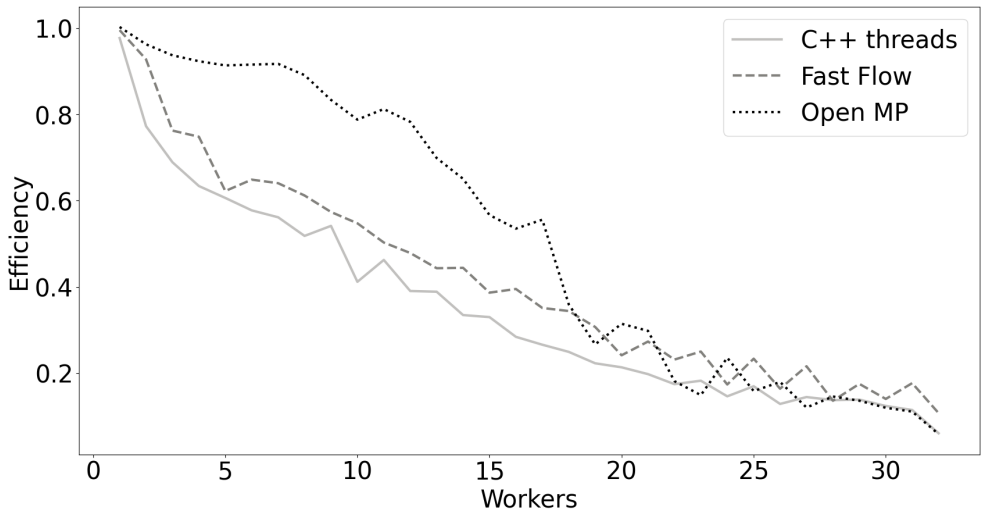

Figure 5: Efficiency vs workers for matrix size 1000

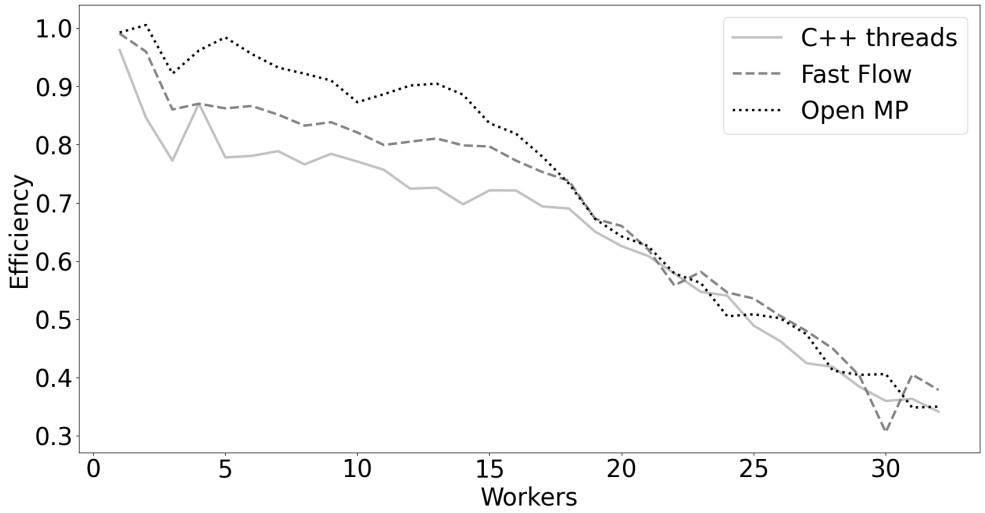

Figure 6: Efficiency vs workers for matrix size 10000

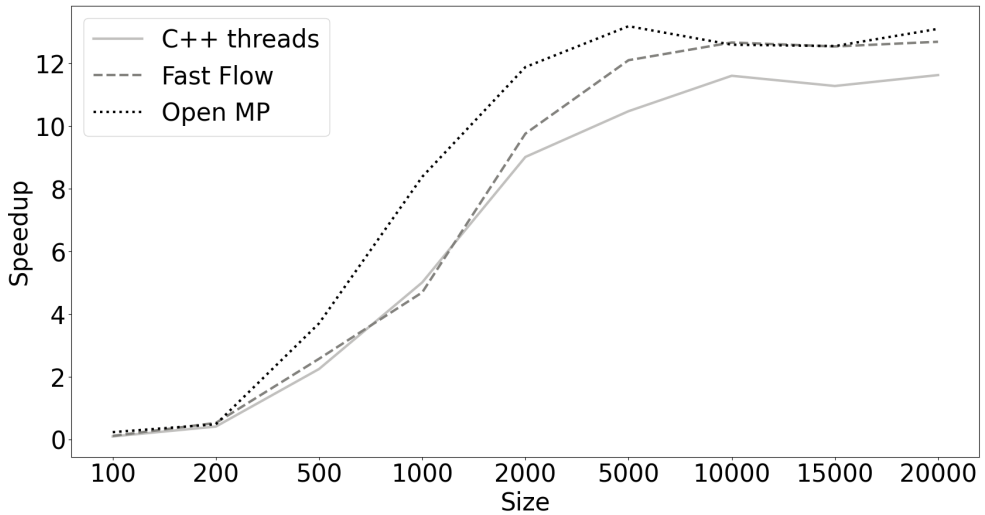

Figure 7: Speedup vs matrix size with 32 workers

6 Conclusions

The first thing that we notice from the experiments is that the performances gets better as the dimension of the matrix increases. The speedups reached when n is large are better than those reached when n is small. Because of this, also the efficiency decreases slower when n is large. Moreover figure 7 shows that when we fix the number of worker, increase the size of the matrix produces better speedups. This is probably due to the fact that as n increases, also the coarse of the computations increases, since a single worker has to compute bigger chunks.
In the case of $n = 1000$, the OpenMP implementations tends to do much better then the other two, and FastFlow does better than the native C++ threads implementation. For the experiments with $n = 10000$ the performances of the three implementations are very similar, even if also in this case the OpenMP versions does a little bit better than the other two, and FastFlow does a little bit better of the native C++ threads implementation. The fact that OpenMP is the best version is probably due to the fact that is though to optimize data parallel computations.
From the plots we can see that the the speedups tends to be close to the linear one, when the number of workers is small. As the number of workers increases, the efficiency tends to decrease. One possible causes of this is the fact that with few workers the grain of the computations is larger, since we have to handle less threads, and each threads has to do more computations. Since the speedup increases with n, maybe with larger values of n, even with an higher number of workers the speed up would have remained close to the linear one (I couldn't test this hypothesis since with $n = 50000$ the remote machine killed the process).