Università di Pisa

DEPARTMENT OF COMPUTER SCIENCE

SPM FINAL REPORT

COMPUTING SOLUTION OF LINEAR SYSTEMS VIA JACOBI ITERATIVE METHOD

April 4, 2018

Marco Barbone

Student ID: 505575

April 4, 2018

MAUAL

Compilation There is a CMakeLists.txt included. Before compiling there are some variables that need to be set:

- **ff:** specify the FastFlow directory.
- **omp:** Enable or disable OpenMP (default disabled).
- **mic:** Enables flag for intel xeon phi compilation (default disabled).
- **intel:** Enables intel compiler (default gcc).

The makefile can be generated using the standard cmake command *cmake .* and the variables can be set using the standard syntax -D<name>=<value> (e.g. cmake . -Domp=ON).
The target can be generated also with the standard cmake command *cmake –build . – -j4*.

Execution **main** <algorithm> [-w <workers>] [-s <size>] « [-p <filename>][-i <iterations>] [-t <tolerance> [-h] [-d] [-c <seed>] »
The required argument is **algorithm** that indicates the algorithm executed from the following list.

- **sequential:** sequential jacobi algorithm.
- **omp:** OpenMP multi-thread implementation of jacobi algorithm.
- **thread:** plain thread implementation of jacobi algorithm.
- **fastflow:** fastflow implementation of jacobi algorithm.

The optional arguments are:

- **[-w]** number of threads used, default 8.
- **[-s]** size of the matrix, default 1024.
- **[-i]** iteration performed, default 50.
- **[-t]** error tolerated, default -1.
- **[-p]** filename in case of csv exporting, default null.
- **[-h]** prints the helper.
- **[-d]** enable debug prints, solution and error.
- **[-c]** seed used to generate the matrix, default. 42.

INTRODUCTION

The aim of this project is to implement the Jacobi algorithm to solve linear systems.
The entire work-flow of the project went through multiple phases:

1. **Sequential implementation:** a sequential implementation of the algorithm is produced.
2. **Design phase:** design and model a possible parallel implementation of the algorithm.
3. **Evaluation:** evaluate the model and derive the expected performances.
4. **Implementation:** Implement multiple parallel versions of the algorithm using the model designed before.
5. **Testing:** all the different versions are executed using different input and parameters combination in order to measure the actual performance.
6. **Comparison:** all the empirical results are collected and compared with the theoretical ones.

1 DESIGN PHASE

The aim of this phase is to analyze the sequential Jacobi's algorithm and produce a parallel model.

analysis Starting from the sequential code can be observed that the sources of parallelism are:

- **Line 6** the iterations of this for loop are independent of each other. Because the element $x_i^{(k)}$ is calculated using only the row i of the iteration matrix and the solution $x^{(k-1)}$.
- **Line 8** the operator used in this for is associative and commutative this loop can be parallelized using a reduction.
- **Line 12** the norm operator can be parallelized using a reduction.

The loops in *line 8* and in *line 12* are not been parallelized because are too fine grained.
The cost of the loop in *line 12* is also negligible respect to the rest of the computation. Suppose a matrix with size M then the *line 6* loop has complexity M^2 but the *line 12* loop has

Algorithm 1 Jacobi algorithm sequential version

Input:
A linear system in matrix form Ax=b of size MxM
$\boldsymbol{\epsilon}$ the maximum accepted error
K the maximum number of iterations

Output:
$\boldsymbol{x^{(k)}}$ the solution vector

```
1:  function JACOBI(A, b, ϵ, K)
2:      x^(0) ← initial guess
3:      k ← 0
4:      repeat
5:          error ← 0
6:          for i=0 to M do
7:              β ← 0
8:              for j=0 to M do
9:                  if i ≠ j then
10:                     β ← β + a_ij x_j^(k)
11:             x_i^(k+1) ← (b_i − β) / a_ii
12:         error ← ||x^(k+1) − x^(k)||_1 / N
13:         k ← k + 1
14:     until error > ϵ and k < K
15:     return x^(k)
```

complexity M (e.g. if M is 1000 then the norm consumes the 0.001% of the computation time).
Parallelize the *loop 6* means to assign different rows to the different processors in order to calculate some components of the solution vector in parallel.

model The operations that the sequential Jacobi algorithm performs are:

- **Read:** read or generate the linear system matrix, the known terms vector, and store it in Ram.
- **Alloc:** initialize the solution vector (*line 6*).
- **Calc:** compute the solution (*line 7-11*).
- **Norm:** calculate the norm (*line 12*).

Define *f[x]* a set of instructions and $T(f[x])$ the function that returns the time to execute f with parameter x. Define also **init** that contains all the initialization operations (in this case only alloc).
The total time required by the algorithm can be modeled as follows:

$$T(\boldsymbol{Jacobi}[\boldsymbol{k},\boldsymbol{m}]) = T(\boldsymbol{read}[\boldsymbol{m}]) + T(\boldsymbol{init}[\boldsymbol{m}]) + k\{T(\boldsymbol{calc}[\boldsymbol{m}]) + T(\boldsymbol{norm}[\boldsymbol{m}])\} \quad (1)$$

where k is the number of iterations performed and m the size of the matrix.

Now suppose that **calc** is parallelized using n workers. Three other operations are introduced:

- **Split:** partition the original data and assign a set of rows to each worker.
- **Spawn:** create the corresponding threads.
- **Collect:** gather all the result and reconstruct the solution vector.

Algorithm 2 Worker pseudo code

Input:
i: first row assigned to the worker.
j: last row assigned to the worker.
Output:
$\{\boldsymbol{x_i}, \boldsymbol{x_j}\}$ updated subset of the solution vector.

```
1: function WORKER(i, j)
2:     check termination
3:     for index=i to j do
4:         β ← 0
5:         for j=0 to M do
6:             if i ≠ j then
7:                 β ← β + a_ij x_j^(k)
8:         x_i^(k+1) ← (b_i − β) / a_ii
9:     sync
```

While the single worker performs the following operations:

- **Check:** check if the termination has been signaled
- **Calc:** compute the solution.
- **Sync:** wait that all workers finish their job.

The **init** now contains not only the **alloc** operation but also **split** and **spawn**.
In this case the total time can be modeled as:

$$T(\boldsymbol{Jacobiparallel}[\boldsymbol{k},\boldsymbol{m},\boldsymbol{n}]) = \tag{2}$$

$$T(\boldsymbol{read}[\boldsymbol{m}]) + T(\boldsymbol{init}[\boldsymbol{m},\boldsymbol{n}]) + \tag{3}$$

$$+\boldsymbol{k}\left[\frac{T(\boldsymbol{calc}[\boldsymbol{m}])}{\boldsymbol{n}} + T(\boldsymbol{collect}[\boldsymbol{m}]) + T(\boldsymbol{norm}[\boldsymbol{m}])\right] \tag{4}$$

Where k is the number of iterations and n is the number of workers. T(**check**) is a constant time operation. Assuming a shared memory architecture also T(**collect**) become negligible. Substituting T(**init**) and T(**calc**) with the worker code, the formula becomes:

$$T(\boldsymbol{Jacobiparallel}[\boldsymbol{k},\boldsymbol{m},\boldsymbol{n}]) = \tag{5}$$

$$T(\boldsymbol{read}[\boldsymbol{m}]) + T(\boldsymbol{alloc}[\boldsymbol{m},\boldsymbol{n}]) + T(\boldsymbol{split}[\boldsymbol{m},\boldsymbol{n}]) + T(\boldsymbol{spawn}[\boldsymbol{n}]) + \tag{6}$$

$$+\boldsymbol{k}\left[\frac{T(\boldsymbol{calc}[\boldsymbol{m}])}{\boldsymbol{n}} + T(\textbf{sync}[\textbf{n}]) + T(\boldsymbol{norm}[\boldsymbol{m}])\right] \tag{7}$$

2 EVALUATION PHASE

In order to simplify the evaluation two assumption are taken:

1. Read is excluded from the evaluation because the they are common to all Jacobi's implementation.
2. The input data is a square matrix of size MxM.

Sequential Referring the formula 1 let's analyze the complexity of each part of the algorithm.

- **Calc:** performs a matrix-vector multiplication than the complexity is O(M^2).
- **Norm:** Standard vectorial norm that takes O(M).

Parallel Let's now analyze the formula 5 which models the parallel version of the algorithm. The split can be neglected because is a constant time operation that is executed only one time. This leaves with calc, sync, and norm let's analyze them separately.

- **Calc:** The matrix is divided by rows so the smallest amount of work that a worker performs is a row by column multiplication which leads to a complexity $\Theta(M * rows)$.

- **Synk:** The evaluation of the synchronization time is difficult since it depends mostly on the underlying architecture.
- **Norm:** The norm is not parallelized so the time that it needs is the same as the one stated for the sequential version ($\Theta(M)$ time).

The main overhead introduced parallelizing the algorithm (on a multi-core shared memory architecture) is given by the synchronization mechanism. It is a step function that depends mostly on the underlying architecture and the number of workers. Two situations are possible:

1. It grows slower or equal respect to the number of workers.
2. It grows faster than the number of workers.

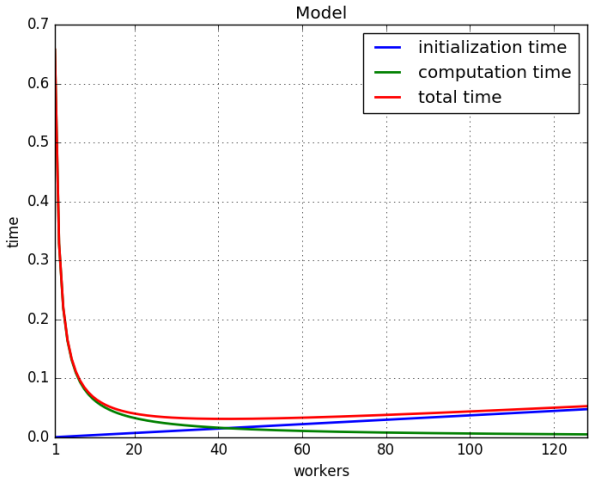
Moreover, it is not very useful evaluate the sync time alone but is more interesting the comparison between calc and sync time. Because even if the sync time is very big but the computation takes really long it is not a problem but instead if the calc time is very short then the synchronization is a bottleneck. Let's evaluate the model discussed in 1. The init time has been calculated using the following procedure:

$T(init) = T(alloc) + T(split) + T(spawn)$
where:
T(split) is negligible.
$T(alloc) = m * (alloctime) = m * \frac{alloc\ vector\ of\ size\ k}{k}$
$T(spawn) = n * (spawntime) = n * \frac{create\ k\ threads}{k}$

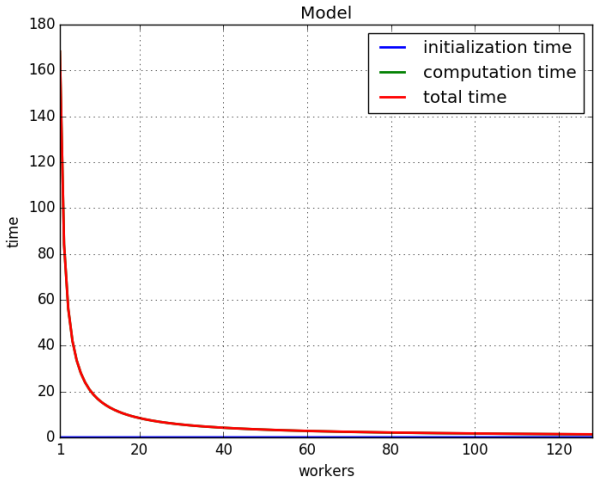
The computation time has been calculated by:

T(calc)=T(multiplication time)*M^2

Where the multiplication time has been obtained from the sequential implementation.
These are very rough measurements because T(init) is supposed to be linear both in the number of workers and in the size of the matrix, which is not always true (might be linear in some intervals). T(calc) overestimates the multiplication time (also ADD, IF, GOTO and other instructions are executed and bias the measurement). It is also supposed to be linear and does not takes into account the sync overhead.
Using the parameters stated before and imposing T(sync)=0 and the number of iteration to 50, the following images show the plot of the expected behavior.



(a) Matrix size 1024



(b) Matrix size 16384

Using a small matrix can be noticed that even without the sync overhead the initialization time became a bottleneck when the number of workers grows.

3 IMPLEMENTATION

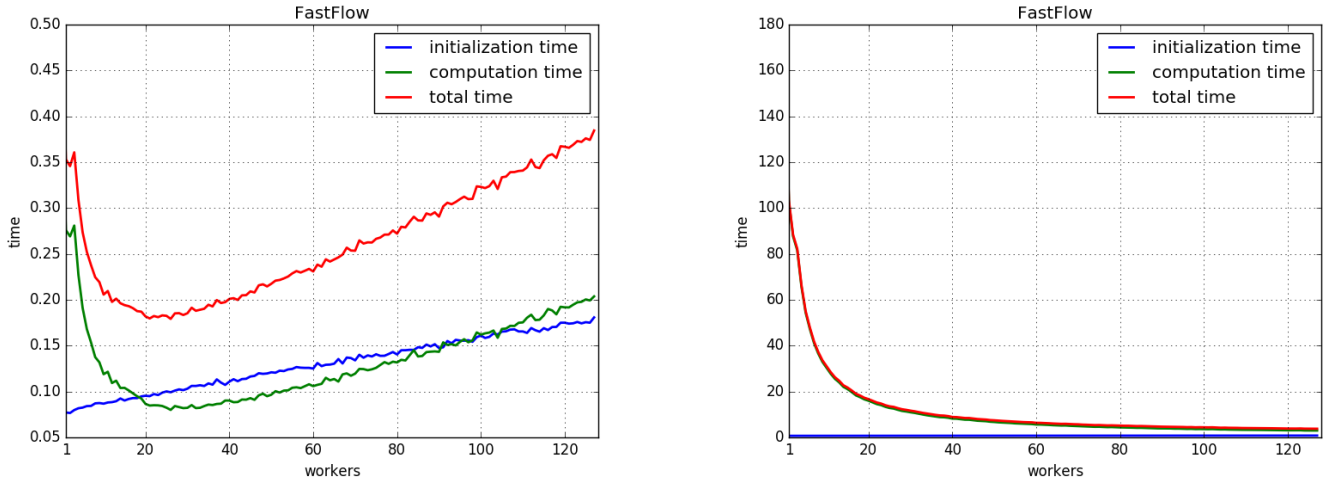
Four implementations of the algorithm are provided:

- **sequential** simple sequential code.
- **fastflow** parallel code developed using the fastflow framework.
- **openmp** parallel code developed using openmp.
- **thread** low-level thread implementation.

Several optimizations are used during the development of the various versions. The row-column multiplication is always vectorized and when possible the matrix and the solution vector are aligned. The fastflow version is implemented by means a parallel_for skeleton which parallelizes the matrix-vector multiplication present in line 6. The same is done with openmp using "#pragma omp parallel for".
More interesting is the thread implementation which allows more handmade optimization. As for the other version, only the line 6 loop is parallelized. The waiting is always active (no wait, sleep...) in order to minimize the context switch overhead. There is no explicit locking, atomic variables and fences are used for synchronization in order to minimize the sync overhead.

4 TESTING

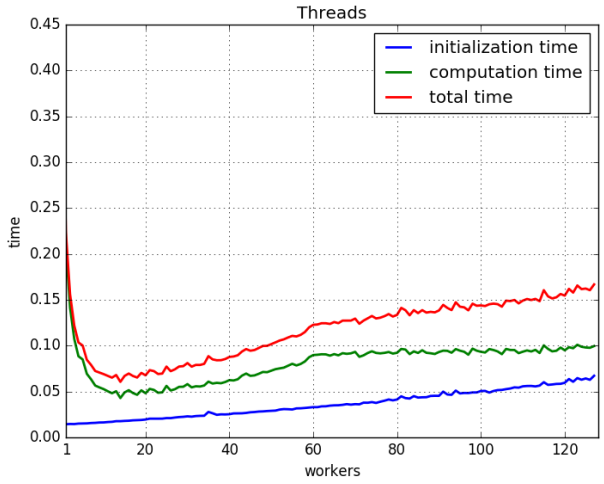
All tests are executed on the Intel Xeon Phi processor. The number of iteration executed during the test is always 50 which is reasonable because using a smaller amount of iteration on the small matrix lead to a computation time smaller than the initialization time even with few workers. The number of workers varies in range 1 and 128 to test the speedup achieved. Time, speedup, and efficiency are measured using two different matrices one small (1024x1024) and the other big (16384x16384). When OpenMP is used the initialization time does not contains the spawn time.



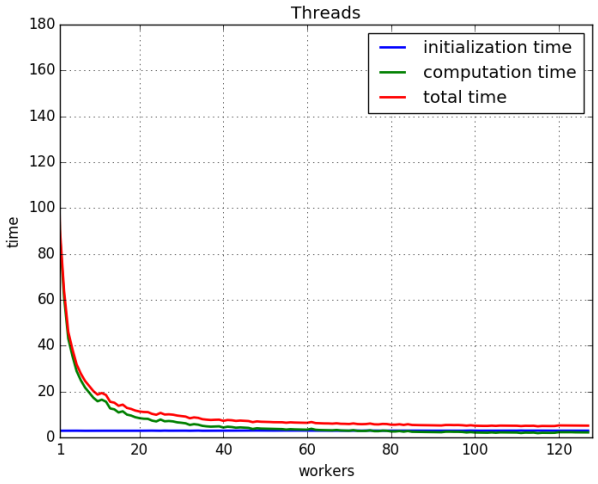
(a) Matrix size 1024

(b) Matrix size 16384

Figure 2: FastFlow behavior varying the number of workers threads

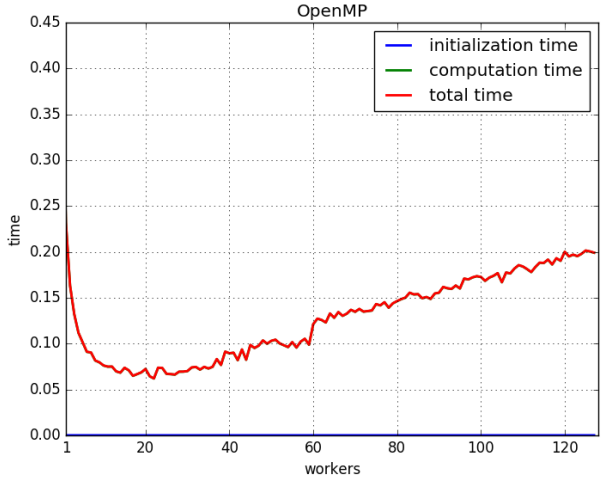


(a) Matrix size 1024

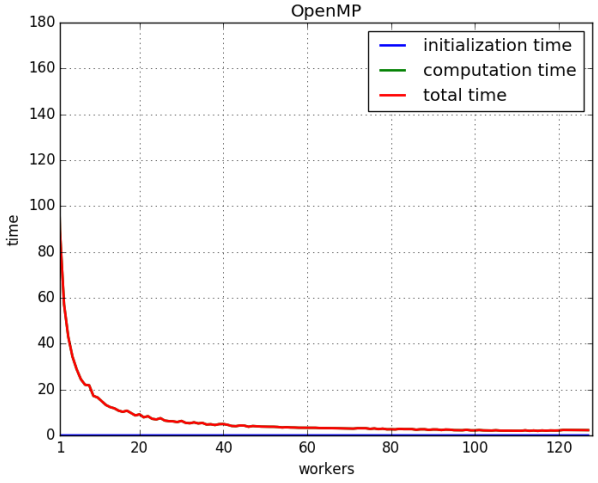


(b) Matrix size 16384

Figure 3: Threads behavior varying the number of workers threads



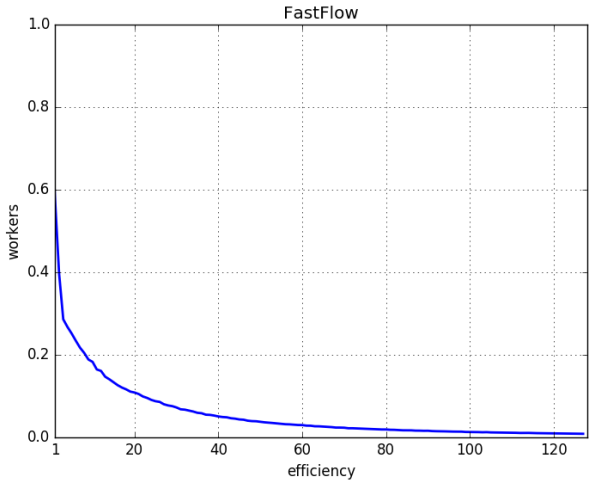
(a) Matrix size 1024



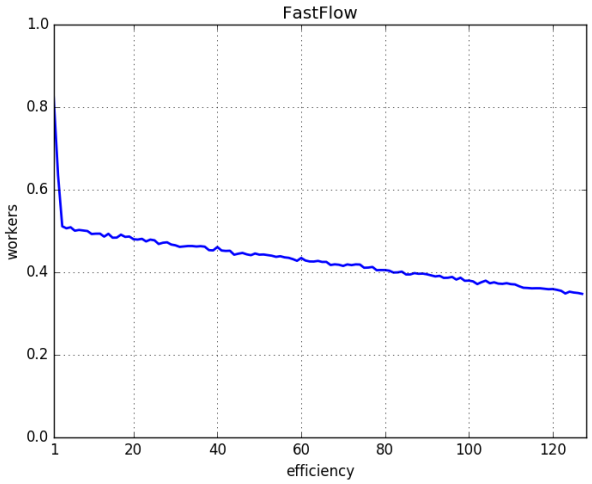
(b) Matrix size 16384

Figure 4: OpenMP behavior varying the number of workers threads

All the framework used show the same behavior: using a small matrix after 20 cores they stop scaling. This happens because the sync overhead and the initialization time become bigger than the calc time. Using a bigger matrix leads to better results because of the reduced impact of the synk and init time.

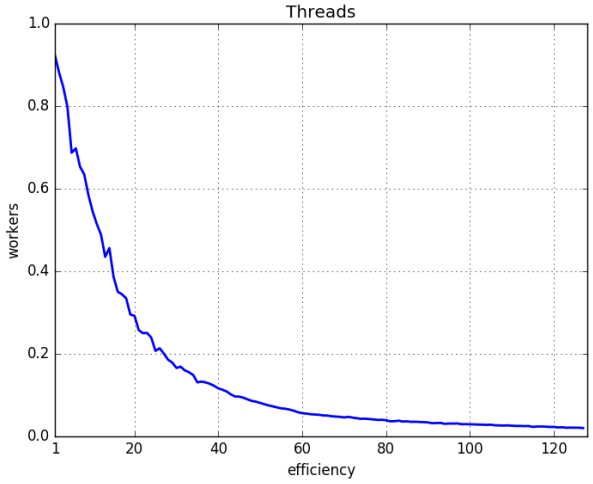


(a) Matrix size 1024

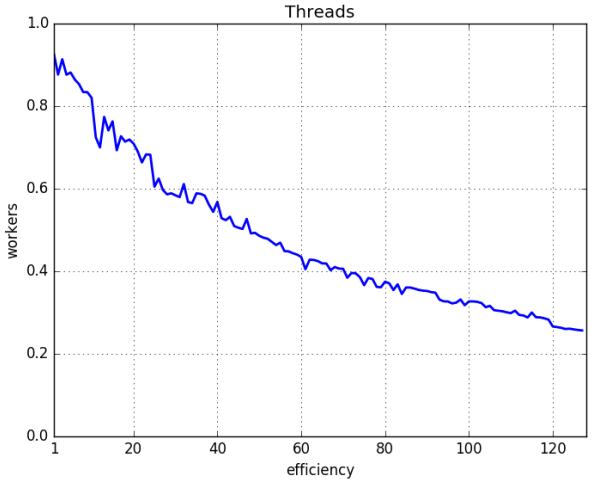


(b) Matrix size 16384

Figure 5: FastFlow efficiency varying the number of workers threads

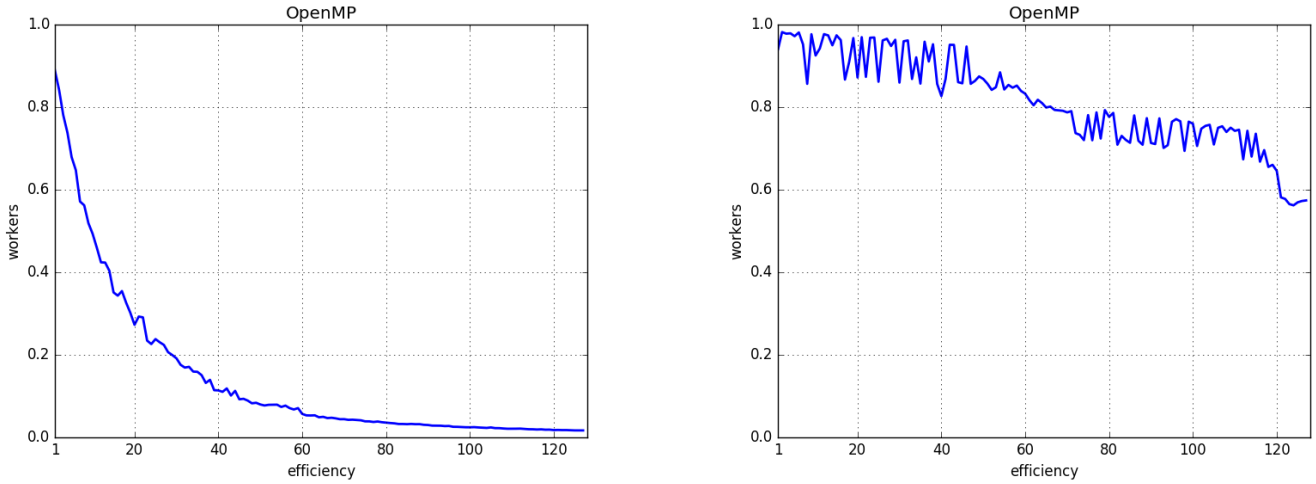


(a) Matrix size 1024



(b) Matrix size 16384

Figure 6: Threads efficiency varying the number of workers threads



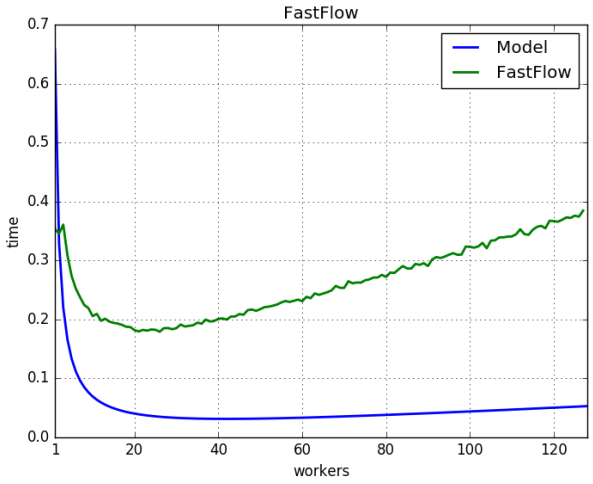
(a) Matrix size 1024 **(b)** Matrix size 16384

Figure 7: OpenMP efficiency varying the number of workers threads

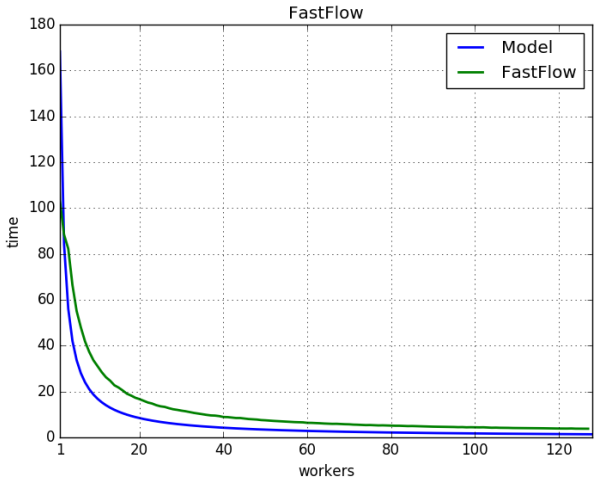
The efficiency instead shows some interesting results. FastFlow drops from 1 to 2 workers. The manual thread implementation drops very fast using a small matrix, on the contrary, using a bigger one decreases linearly. OpenMP shows the same results of the thread implementation using a small matrix. Using a big matrix it achieves very good results with a very high efficiency.

5 Comparison

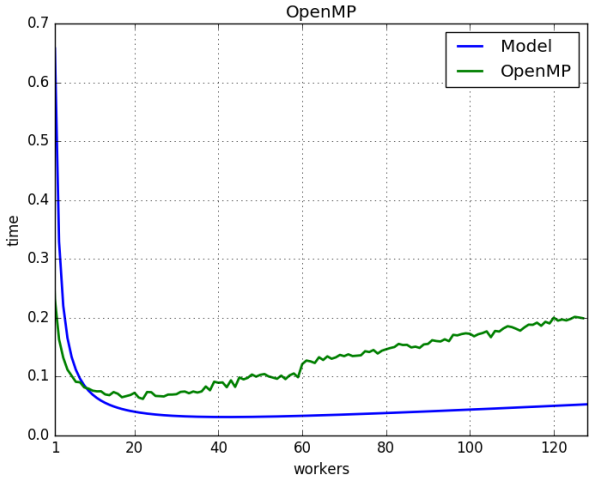
From the comparison between the model depicted before and the experimental results can be noticed that with small matrices the synchronization time is not negligible and its impact on the performance lead to very poor scalability. Contrarily using big matrices the results almost fits with the model and in the case of OpenMP, the fitting is practically perfect.



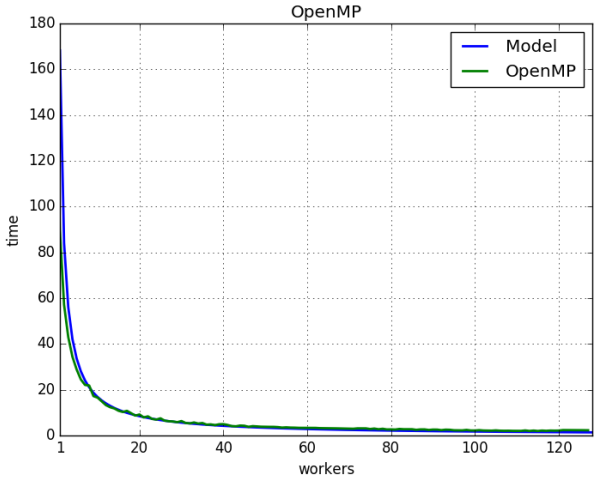
(a) Matrix size 1024



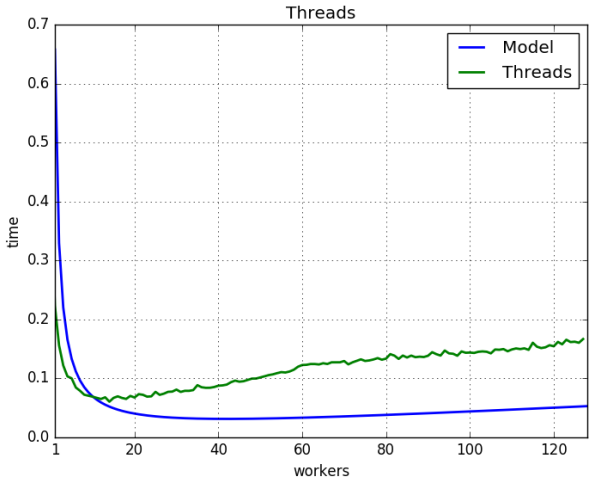
(b) Matrix size 16384



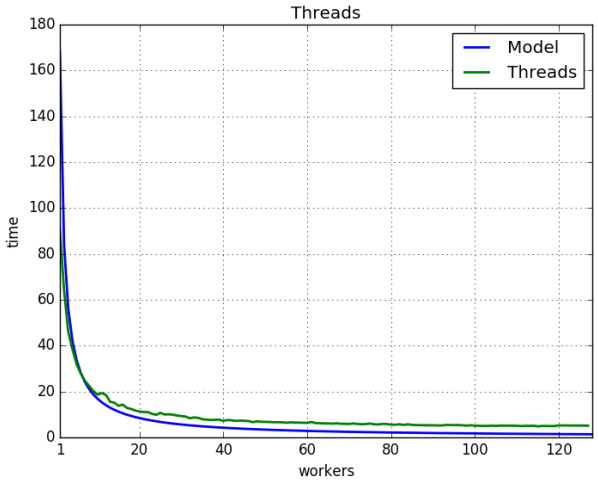
(a) Matrix size 1024



(b) Matrix size 16384



(a) Matrix size 1024



(b) Matrix size 16384

6 Conclusions

In conclusion, the measured performances are close to the one calculated from the model if the matrix is "big enough". Using small matrices the performances are not very good because of the synchronization overhead. Differently using big matrices leads to a smaller gap between the ideal performance and the real one. The manual thread implementation archives very good performance in the computation time but around 45 workers the initialization time starts dominating, impacting the performance and the scalability. Fastflow shows a good behavior with a very large number of workers.