Computer Science Department
University of Pisa

Parallel Jacobi Method

Luca de Martino

Parallel and Distributed System Project

List of Contents

1 Introduction

This report aims to build several implementations of the Jacobi Method to fully understand the benefits of using parallel patterns. The report is divided into the following seven sections:

- **Introduction** which summarizes the purpose of each section;
- **Problem Analysis** in which the sequential pseudo-code of the problem is analyzed to identify the parallel patterns which best suited for the parallelization, given the critical issues the algorithm may have;
- **Time Analysis** in which is described the time analysis in the sequential and parallel cases;
- **Implementation** in which the implementation choices are described;
- **Build & Run** in which it is shown how to build the program, launch it and reproduce the *"same"* results;
- **Results** in which the results of the experiments are shown;
- **Conclusions** in which the results obtained are analyzed and conclusions are drawn accordingly;

2 Problem Analysis

In this section, are made some considerations given the sequential pseudo-code of the Jacobi Method in algorithm 1. Before looking at the pseudo-code in detail, the following two premises are made:

- $x_i^{(k)}$ indicates the i-th entry of vector $\boldsymbol{x}$ of the unknowns at the iteration k of the algorithm.
- As *"convergence criteria"*, it was planned to use only an *a priori* fixed number of iterations K, since using the norm:

  would lead to the algorithm having a varying number of iterations when employing different linear system dimensions, thus preventing us from achieving an optimal time comparison.

Algorithm 1 Sequential Jacobi Method pseudo-code

Require: $\boldsymbol{A} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{n}}$ (strictly diagonally dominant), $\boldsymbol{b} \in \mathbb{R}^{\boldsymbol{n}}$, $K \in N$
Ensure: $\boldsymbol{x}$ *s.t.* $\boldsymbol{Ax} = \boldsymbol{b}$

```
1:  x^(k) ← 0
2:  x^(k−1) ← 0
3:  for k ← 0 to K do
4:      for i ← 0 to n do
5:          sum ← 0
6:          for j ← 0 to n do
7:              if j ≠ i then
8:                  sum ← sum + A[i, j] ∗ x_j^(k−1)
9:              end if
10:         end for
11:         x_i^(k) ← (b_i − sum)/A[i, i]
12:     end for
13:     swap(x^(k−1), x^(k))
14: end for
```

Let us analyze the pseudo-code 1 line by line, to fulfill the requirements, the algorithm needs a function to generate a strictly diagonally dominant matrix and a function to generate vectors.

Next, we see two vectors must be initialized and not one, this is because the Jacobi Method needs an auxiliary vector since if at iteration k it computes $x_1^{(k)}$ modifying $\boldsymbol{x}$ in-place, then when it computes $x_2^{(k)}$ it would use the updated $x_1^{(k)}$ instead of using $x_1^{(k-1)}$. This issue costs memory but is not a big deal from a time complexity point of view as we simply need to use a $swap$ function which swaps the pointer of the two vectors in $O(1)$ time complexity at the end of the algorithm.
Having noticed several for we try to exploit their characteristics to parallelize the code.
From the first for (from lines 3 to 14) is clear that the algorithm needs the entire $x^{(k-1)}$ to produce the $x^{(k)}$ at the iteration k consequently it has to be computed sequentially to be consistent.
From the second for (from lines 4 to 12) we can notice that to compute each $x_i^{(k)}$ of the vector $x^{(k)}$ we only need the vector of unknowns at the previous iteration $x^{(k-1)}$ then each $x_i^{(k)}$ can be calculated independently in parallel, this is nothing more than an application of the *map pattern*. However, we must ensure we have calculated all the elements $x_i^{(k)}$ of the vector $x^{(k)}$ before moving on to the next $k+1$ iteration.
The last for (from lines 6 to 10) could be parallelized with a *reduce pattern* since it is the summatory of n elements, nevertheless, I decided not to parallelize it, since it would increase the overhead due to synchronization with the *map pattern* workers and since theoretically if we used the same number workers of the *map plus reduce* to just increase the parallel degree of the *map* we would have better performance as demonstrated in section 3.

3 Time Analysis

In this section, we will theoretically analyze the time to execute the algorithm in both sequential and parallel cases.
In the sequential case:

$$T_{seq}(n,K) = T_{init} + K\Big(n^2 T_{ps} + nT_{xi} + T_{swap}\Big)$$

where T_{init} is the time required to initialize some *Primitive Data variables* in *C++*, to generate the strictly diagonally dominant matrix $\boldsymbol{A}$, the vector of known terms $\boldsymbol{b}$ and the vector of the unknown and it is auxiliary one $x^{(k)}$, $x^{(k-1)}$, K is the number of iterations of the Jacobi Method, n is the linear system dimension corresponding to the number of iterations of the last two nested for loop (the reason why also the second for is executed exactly n time is explained at the end of the subsection 4.1), T_{ps} is the time required to compute the partial summation at line 8 of algorithm 1, T_{xi} is the time required to compute the line 11 of algorithm 1 used to obtain $x_i^{(k)}$ and T_{swap} is the time required to swap the vectors of the unknown.
Considering negligible the initialization and the swap times, we have:

$$T_{seq}(n,K) \approx K\Big(n^2 T_{ps} + nT_{xi}\Big)$$

In the parallel case, having parallelized the for at line 4 of the algorithm 1 by using nw workers, the algorithm will execute at the same time nw parallel loops, and each of these iterates roughly $\frac{n}{nw}$ times (*chunk size*). However, we have to consider the *overhead* $T_{overhead}$ introduced by the parallelization.
$T_{overhead}$ is given by the summation of:

- T_{start} which is the time to spawn a workers;
- T_{stop} which is the time to terminate a workers;
- T_{split} which is the time to compute a *chunk*;
- T_{sync} required to manage each worker e.g., using a *barrier* (as illustrated in section 4) which is used to ensure we have computed each $x_i^{(k)}$ before moving to the next iteration $k+1$;

Note that since we are in a *shared memory multicore* and each worker works on its own *chunk*, their task is embarrassingly parallel and they work directly in place on the vector $x^{(k)}$ consequently we do not have a T_{merge} time.

Summarizing, we can write the parallel time as:

$$T_{par}(n, K, nw) = T_{init} + T_{start} + T_{split} + K\Big(\frac{n}{nw} nT_{ps} + \frac{n}{nw} T_{xi} + T_{synch} + T_{swap}\Big) + T_{stop}$$

As before, we can consider negligible the times required for the initialization and the swap. Therefore, considering just for a while $T_{overhead}$ negligible, the parallel time could be roughly approximated by:

$$T_{par}(n, K, nw) \approx K\frac{n}{nw} nT_{ps} + K\frac{n}{nw} T_{xi} = K\Big(\frac{n}{nw} nT_{ps} + \frac{n}{nw} T_{xi}\Big)$$

From this result, let us assume that we had also implemented the *reduce pattern* within the *map*. If we assign nw_1 workers to the *map* and nw_2 workers to each worker of the *map* for the *reduce*, then the parallel time would be:

$$T'_{par}(n, K, nw1, nw2) \approx K\Big[\frac{n}{nw_1}\Big(\frac{n}{nw_2} T_{ps} + (nw_2 - 1)T_{sum}\Big) + \frac{n}{nw_1} Txi\Big] =$$

$$= K\frac{n}{nw_1}\frac{n}{nw_2} T_{ps} + K\frac{n}{nw_1}(nw_2 - 1)T_{sum} + K\frac{n}{nw_1} Txi$$

where T_{sum} is the time to perform a sum between two numbers.
This way, we have computed for each map nw_2 partial sums of $\frac{n}{nw_2}$ elements each, and at the end, we sum up these partial sums.
At this point, it can easily be seen that if we increase the parallelism degree of the *map-only* version to $nw = nw_1 \cdot nw_2$, to have the same number of workers as the *map plus reduce pattern*, we will have worse results in the *map plus reduce pattern* because comparing T_{par} with T'_{par}:

- $K\frac{n}{nw} nT_{ps} = K\frac{n}{nw_1}\frac{n}{nw_2} T_{ps}$
- $K\frac{n}{nw} T_{xi} < K\frac{n}{nw_1} T_{xi}$
- $K\frac{n}{nw_1}(nw_2 - 1)T_{sum}$ which is greater than zero and is not present in T_{par}

Furthermore, the introduction of the *reduce* would increase the overhead due to synchronization between the workers of the two different patterns (as mentioned in the section 2). Consequently for sure we can claim $T'_{par} > T_{par}$.

4 Implementation

In this section are described the different implementations of the Jacobi Method. The *core* of the program is arranged in:

- ***utility.cpp*** which contains the method to generate a strictly diagonally dominant matrix given a seed and the matrix dimension and a method to generate a random vector given a seed and its dimension;
- ***jacobi.cpp*** which contains the three implementations of the Jacobi Method (sequential, parallel using the standard thread library, and parallel using *FastFlow* [Aldinucci et al. (2017)]);
- ***overhead.cpp*** which computes an approximation of $T_{overhead}$ for both the implementation using the standard library and *FastFlow* for an increasing number of workers;
- ***utimer.cpp*** which uses the *RAII* mechanism to get the creation and the destruction time of its instances;
- ***main.cpp*** which implements the interface that reads parameters from the command line, reports errors, builds the result *csv files*, and launches methods implemented in the *.cpp* mentioned above;

4.1 Sequential Implementation

This subsection refers to the *sequential_jacobi* method contained in *jacobi.cpp*. It is nothing more than the *C++* implementation of pseudo-code 1 with an expedient to promote vectorization. Practically, the if in line 7 with its body in line 8, is replaced by this single following *C++* statement:
$sum = ((i == j) * sum) + ((i! = j) * (sum + A[i][j] * prev_x[j]));$
This takes advantage of the properties of the *comparison operator* to compare indices and assign the correct value to the summatory, allowing us to vectorize this part of the code, eliminating the if statement.
Furthermore, note this way exactly n statements are executed in the body of the for at line 6 of algorithm 1.

4.2 Standard Parallel Implementation

This subsection refers to the *standard_parallel_jacobi* method contained in *jacobi.cpp*. The standard *thread* library was used to implement the following code. Unlike the sequential version, here the fork-join mechanism is used to spawn a certain number nw of threads and each of them computes its corresponding *chunk* which has a size of approximately $\frac{n}{nw}$ elements of the vector $\boldsymbol{x}$. As mentioned earlier, when we are at a certain step k to move on to the next step $k+1$ we must be sure that the whole vector has been updated, i.e., each thread has finished updating all the elements of its chunk and to ensure this we use the *barrier* (available from C++20). Within the body of each thread there is a synchronization point called the method *arrive_and_wait* so that when all threads have finished their work, one of them decreases the number of iterations and executes the pointers' swap.

4.3 *FastFlow* Implementation

This subsection refers to the *ff_jacobi* method contained in *jacobi.cpp*. For the implementation of the following code, it was used the *FastFlow* library. As with the standard parallel version, the number of workers nw must be specified. We then instantiate an object of the *ff::ParallelFor* class and subsequently call its *parallel_for* method instead of the classical *for* of line 4 of algorithm 1. The chunksize specified in the method *parallel_for* is equal to 0 to have a static scheduling where each worker gets a contiguous chunk as in the standard parallel implementation.

4.4 Overhead & Expected Parallel Time

In this subsection, we try to estimate the $T_{overhead}$ for both the parallel versions using the *thread* library and the *FastFlow* one. In the first case, the fork-join mechanism is used to launch a number of threads nw, in which body we compute its corresponding chunk and is added a *barrier* to try to simulate its handling time as well. In the second case, instead, we calculate the instantiation time of the class *ff::ParallelFor* for handling nw number of workers to perform its *parallel_for* method with a static chunksize. Furthermore, to compute the *expected parallel time* for both the parallel versions it was decided to use the formula: $T_{expected} = \frac{T_{seq}}{nw} + T_{overhead}$ so the ideal time plus the overhead time.

5 Build & Run

The *core* of the program has already been explained in section 4. To build it, all we need to do is to move to the *src* folder of the program and run the ***make*** (or ***make all***) command, which will execute the *Makefile* (which compiles using the *-O3* flag), making the ***main*** and ***overhead*** executable available.

You can check the usage of the ***main*** executable running: *./main -help* command and it has to be launched with the following parameters in the following order:

- N: linear system dimension;

- SEED: seed number for the matrix and array generation;
- K: number of iterations of the Jacobi Method;
- TRIALS: Number of trials of the Jacobi Method for the time analysis;
- MODE: implementation mode types: *seq* (for sequential), *std_par* (for parallel mode using the standard thread library), *ff* (for *FastFlow*);
- NW: number of workers for the parallel code (NOT use it if you have decided to use the *seq* mode);
- FILENAME: filename without the *.csv* extension;
- DEBUG: insert any number to print the time of the algorithm for each trial, 0 to not print it;
- FIXED_DIM: Used to standardize the name of the *csv* file, insert any number to run the method with a fixed dimension of the linear system, 0 to run the method with a fixed number of workers;

Depending on the *FIXED_DIM* parameter, the resulting *csv* file will have a name like *fixed_dim_mode-n.csv* or *fixed_nw_mode-nw.csv*, this is useful for plots creation.

You can check the usage of the ***overhead*** executable running: *./overhead -help* command. It computes the $T_{overhead}$ from 1 worker up to *max_nw* workers for both the parallel versions of the algorithm by inserting the results in two different *csv files*: *standard_overhead.csv* and *ff_overhead.csv*. The *overhead* executable has to be launched with the following parameters in the following order:

- TRIALS: Number of trials of the algorithm for the time analysis;
- MAX_NW: The maximum number of workers for which we want to calculate $T_{overhead}$;
- N: linear system dimension;

All the *csv* results will automatically be placed in a *results* folder.
In case, you want to generate the *"same"* results of section 6, simply run the two scripts:

- ***fixed_dim.sh*** which will compute the results for three fixed linear system dimensions with increasing numbers of workers from 1 to 32;
- ***fixed_nw.sh*** which will compute the results with a fixed number of workers by increasing the linear system dimension;

Finally, to generate the plots, you have to move the results from the virtual machine to the local one (as this one does not have *pip3* installed therefore it is not possible to install *matplotlib* [Hunter (2007)] and *pandas* [pandas development team (2020)] modules). After that, from the project folder, you can install and then create a virtual environment, this way:

- *sudo apt install python3-venv* to install the module;
- *python3 -m venv .venv* to create the virtual environment;
- *source .venv/bin/activate* to activate the virtual environment;

Finally, you can use the *"pip3 install -r requirements.txt"* command to install all the modules needed to run the following two scripts:

- ***fixed_dim_plot.py*** to generate the T_{seq}, the T_{par}, the $T_{expected}$ (for both the parallel versions), the speedup, the efficiency and the scalability plots for fixed linear system dimensions and increasing number of workers. They will automatically be placed in the *fixed_dim* folder of the *plots* directory contained in the *results* folder. You should specify the size of the linear system with the *-n* parameter or it will use the default ones. If you want

to run it with multiple arguments you can use the comma, between each number.
For example: *python3 fixed_dim_plot.py -n 2000,5600,11000*

- ***fixed_nw_plot.py*** to generate the T_{seq}, the T_{par} and the speedup plots for increasing linear system dimensions and a fixed number of workers. They will automatically be placed in the *fixed_nw* folder of the *plots* directory contained in the *results* folder. You should specify with the *-nw* parameter the number of workers or it will use the default one.
For example: *python3 fixed_nw_plot.py -nw 20*

6 Results

This section shows what results were obtained. As stated in section 5, the *"same"* results can be obtained by launching the existing "*.sh*" and then "*.py*" scripts. It can be divided into the following two subsections:

- ***Fixed Linear System Dimension***: in which three linear system dimensions (with increasing size) are fixed to understand if it affects performance and changing the number of workers nw from 1 to 32 (maximum number of cores within the machine we are using) with a step equal to 1 to show: the T_{seq}, the T_{par}, the $T_{expected}$ (for both the parallel versions), the speedup, the efficiency and the scalability;
- ***Fixed Number of Workers***: in which the number of workers nw is fixed according to the previous subsection and the dimension of the linear system is increased by a specific step from an initial linear system dimension up to a maximum one;

All tests were performed with a number of trials equal to 6 to try to compensate for any outliers in the calculation of the times.

6.1 Fixed Linear System Dimension

The results in this subsection are obtained with the parameters in Table 1 and by sequentially launching first *fixed_dim.sh* and then *fixed_dim_plot.py*.

Parameter	Value
K	100
N	2000 - 5600 - 11000
NW	[1 - 32]
SEED	10

Table 1: Fixed Linear System Dimension - Main Parameters

Figure 1: **2000 - Linear System Dimension**

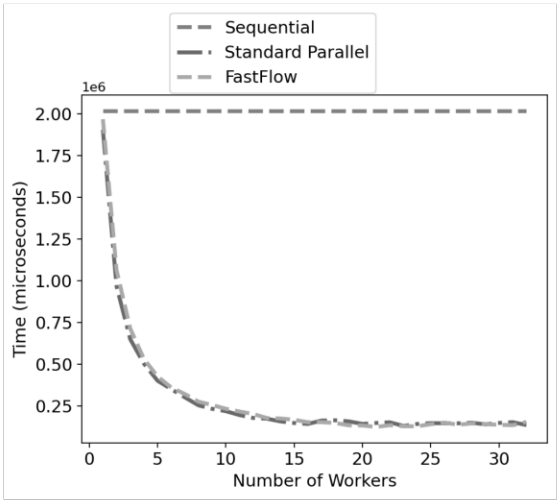

Figure 2: Completion Time

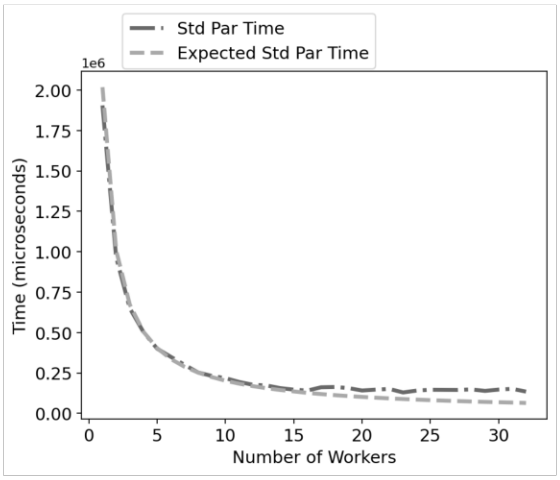

Figure 3: Expected Standard Parallel Time

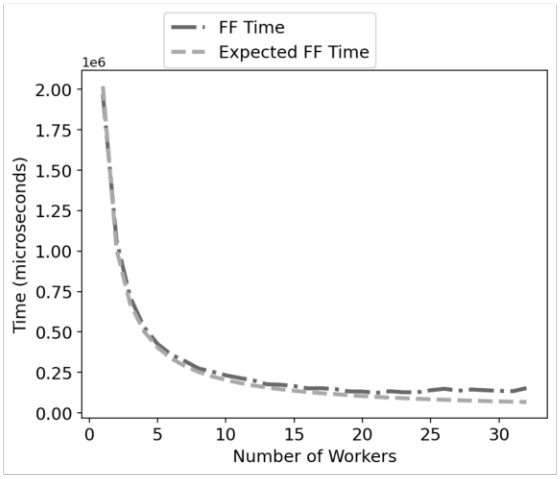

Figure 4: Expected *FastFlow* Parallel Time

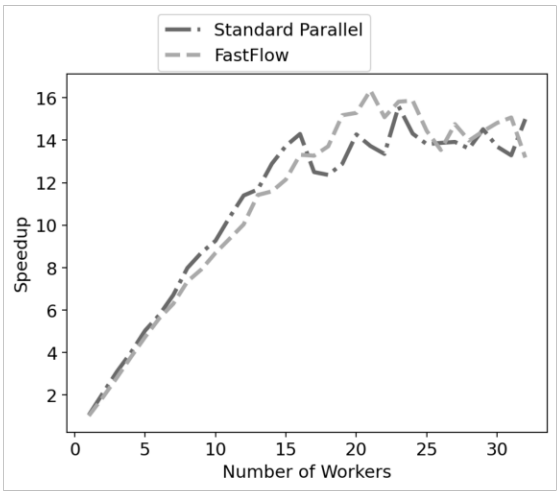

Figure 5: Speedup

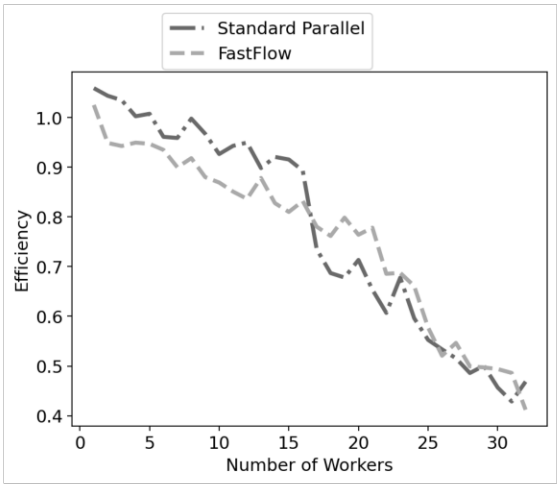

Figure 6: Efficiency

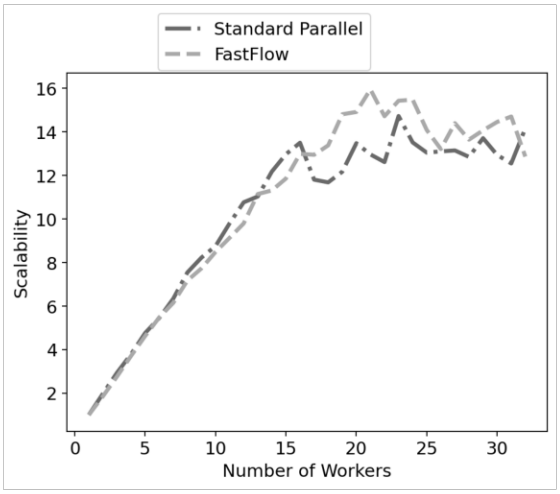

Figure 7: Scalability

Figure 8: **5600 - Linear System Dimension**

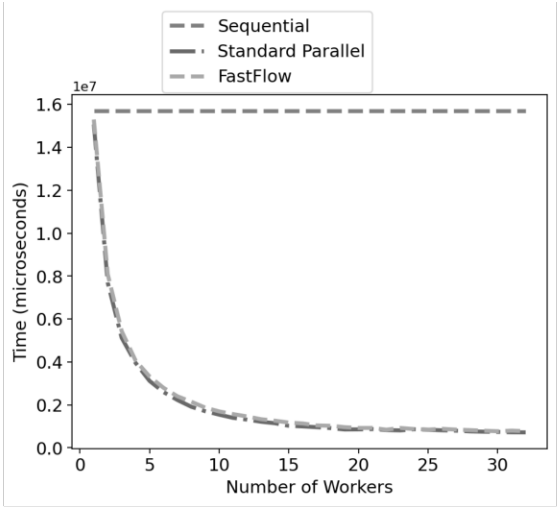

Figure 9: Completion Time

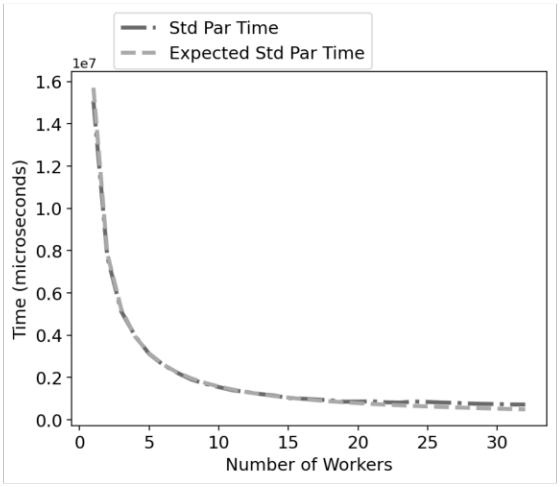

Figure 10: Expected Standard Parallel Time

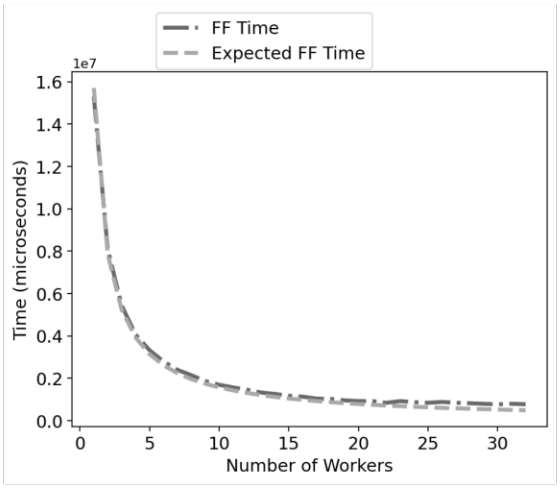

Figure 11: Expected *FastFlow* Parallel Time

Figure 12: Speedup

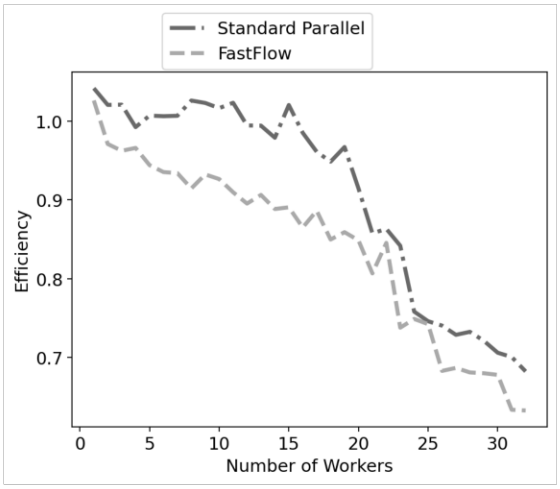

Figure 13: Efficiency

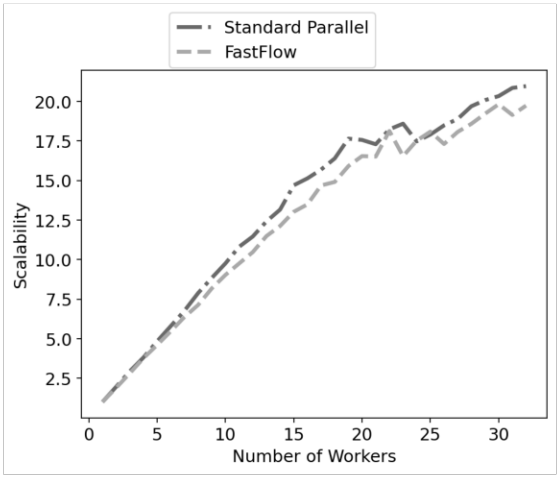

Figure 14: Scalability

Figure 15: **11000 - Linear System Dimension**

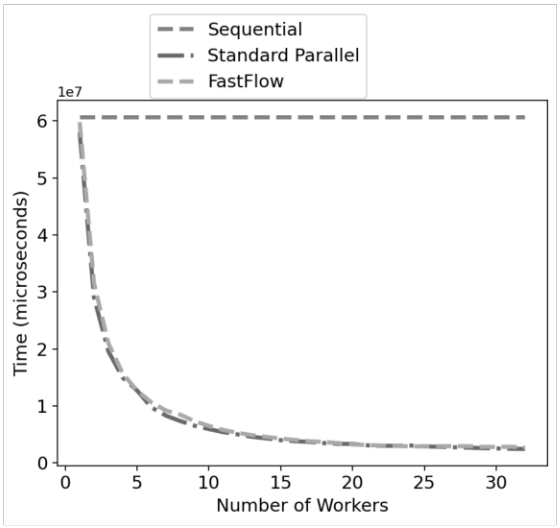

Figure 16: Completion Time

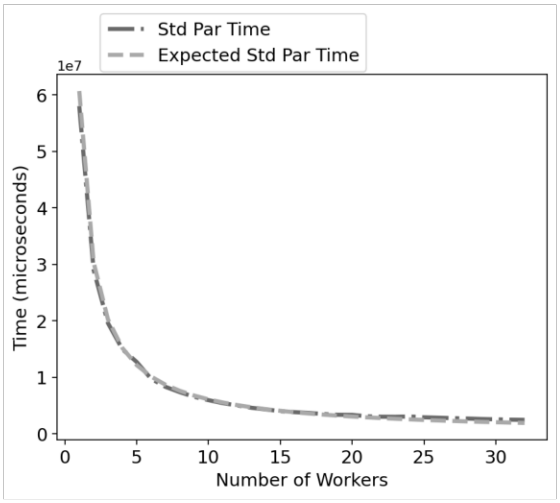

Figure 17: Expected Standard Parallel Time

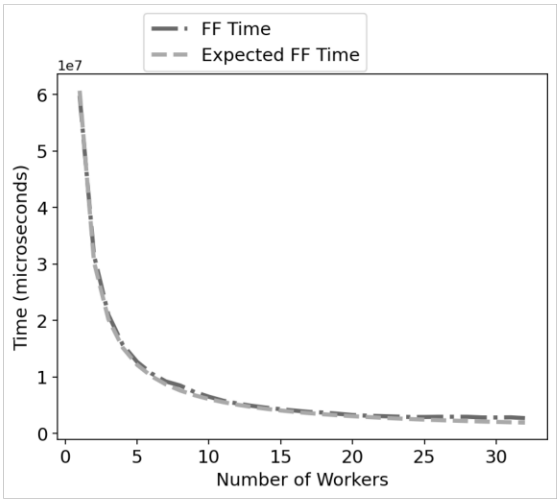

Figure 18: Expected *FastFlow* Parallel Time

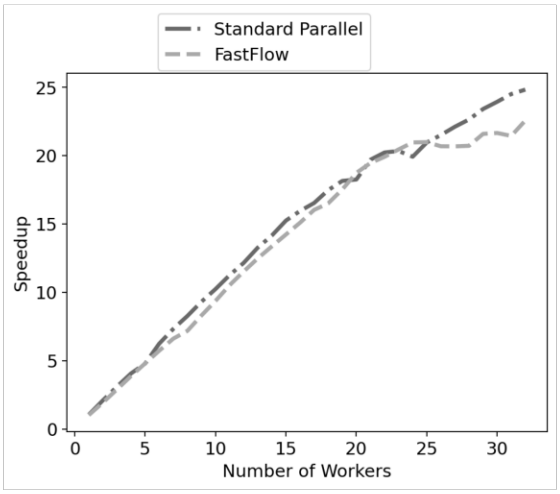

Figure 19: Speedup

Figure 20: Efficiency

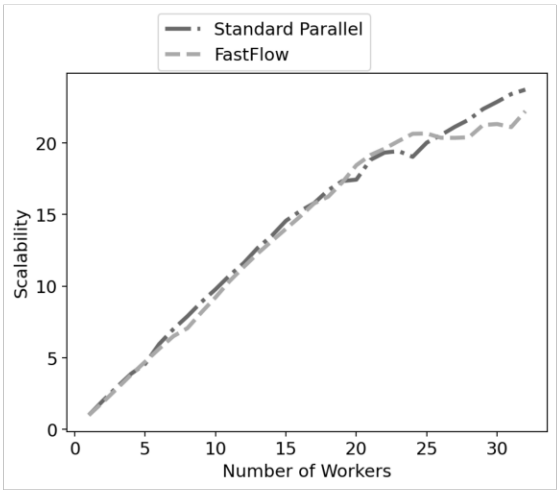

Figure 21: Scalability

6.2 Fixed Number of Workers

The results in this subsection are obtained with the parameters in Table 2 and by sequentially launching first *fixed_nw.sh* and then *fixed_nw_plot.py*.

Parameter	Value
K	100
Maximum N	16000
Starting N	2000
Step of N	800
NW	20 - 22
SEED	10

Table 2: Fixed Number of Workers - Main Parameters

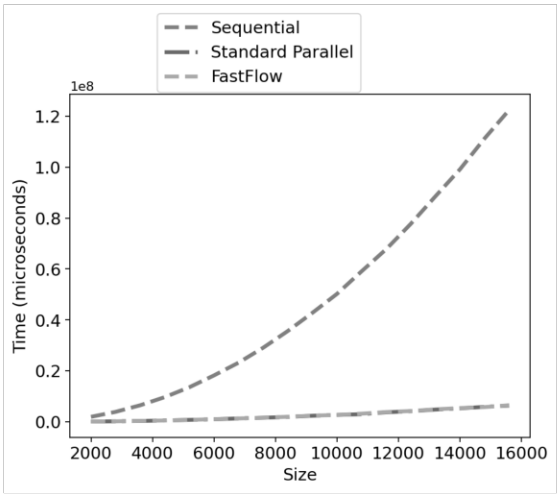

Figure 22: Completion Time with nw equal to 20

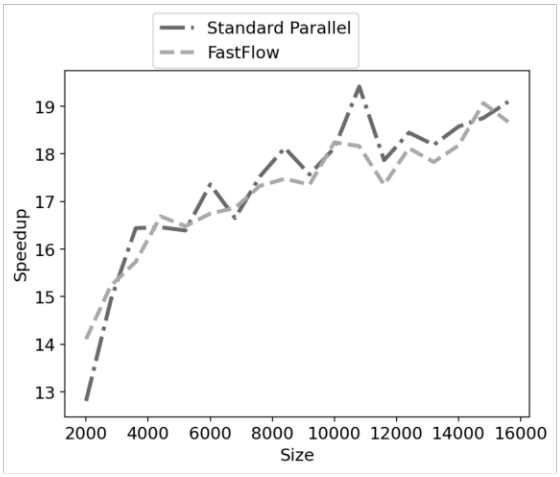

Figure 23: Speedup with nw equal to 20

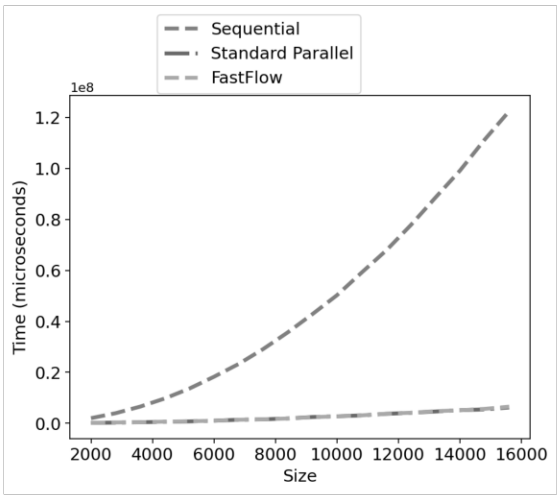

Figure 24: Completion Time with nw equal to 22

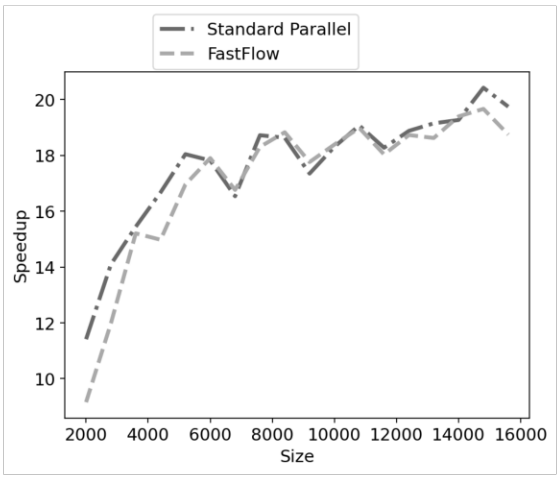

Figure 25: Speedup with nw equal to 22

7 Conclusions

Let us analyze the results obtained in section 6 with the three different linear system dimensions with the increasing number of workers. From the completion time graphs 2, 9, 16, we can see that the times of both the *FastFlow* version and the standard parallel version are almost completely overlapping. However, going to analyze the graphs of the speedup figures 5, 12, 19 and of the scalability figures 7, 14, 21, we can see that the standard version performs slightly better than *FastFlow* one. This result is also confirmed by the efficiency plots 6,13, 20. Furthermore, from all these plots is clear that the ***trend*** is that the more the size n increases, the better speedup, efficiency, and scalability we obtain. The graphs in the figures 22, 23, 24, 25, also confirm what was found previously.
In general, this *trend* happens because the size of the chunks to be assigned to each worker will also be greater, and consequently the computation time will be greater, while the time for synchronization management and communication will carry less and less weight on the completion time, following Gustafson's law. Conversely, when n is *"small"*, then the overhead time has greater valence as it is "closer" to the computation time for each chunk.
The just claimed *trend* is further confirmed by the graphs 3,10,17,4,11 and 18, which show us that the more the dimension n increases, the closer the values of $T_{expected}$ are to those of the real T_{par}.
In conclusion, with these data, the results between the two parallel versions are comparable with the standard version performing slightly better. Nevertheless, it has to be said that the level of detail for development in *FastFlow* is negligible compared to that required by implementation with the standard thread library, consequently the programmability is better in *FastFlow*. Probably with an even more complex problem, we would appreciate this feature more, always having comparable performance.

References

Aldinucci, M., Danelutto, M., Kilpatrick, P. and Torquati, M., 2017. *Fastflow: High-level and efficient streaming on multicore*. John Wiley Sons, Ltd, chap. 13, pp.261–280. https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13, Available from: https://doi.org/https://doi.org/10.1002/9781119332015.ch13.

Hunter, J.D., 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3), pp.90–95. Available from: https://doi.org/10.1109/MCSE.2007.55.

team, T. pandas development, 2020. *pandas-dev/pandas: Pandas* (v.latest), February. Zenodo. Available from: https://doi.org/10.5281/zenodo.3509134.