University of Pisa
Computer Science
Master Degree Artificial Intelligence

Parallel and distributed systems:
Paradigms and Models

Academic Year: 2021/2022
Date: 03/06/2022

Parallelization process of the Jacobi algorithm (AJ)

Author
Christian Peluso
c.peluso5@studenti.unipi.it

Abstract

In this report we will analyze benefits and drawbacks in the exploitation of parallelism on the iterative Jacobi algorithm for searching the vector x that multiplied to the matrix A, of the known terms, produces the solution vector b. As we will see this method is not trivially parallelizable for the dependency of each iteration with the precedent one, but still some approaches can offer significant time gains.

Contents

1 Introduction

The Jacobi method is born with the purpose of solving systems of linear equations, using the simple inverse formula, so replacing iteratively at each step $x^{(k)}$ an "increasingly precise" approximation $x^{(k+1)}$ using the following formula:

$$x_i^{(k+1)} = \frac{(b_i - \sum_{j(i \neq j)}^{n} A_{ij} \cdot x_j^{(k)})}{a_{ii}}, \quad i = 1, ..., n; \quad k \geq 0$$

Instead below we can find a snippet of pseudo code, in which we can see the behavior of the variables involved and the presence of the grafted cycles:

Algorithm 1 Pseudo-code of the Jacobi method

```
1: procedure JACOBI(n, A, b, xold, MaxIt)
2:     repeat
3:     for i=1, n do
4:         x(i) := b(i)
5:         for j=1, n do
6:             if i ≠ j then
7:                 x(i) := x(i) − A(i,j) · xold(j)
8:         x(i) := x(i)/A(i,i)
9:     return x
```

The algorithm starts with the x vector initialized with 0, than in the first **for**, the one scanning the rows, we save the b_i value in x_i, in the second **for** loop, of columns, there is an **if** that make sure to avoid the diagonal values and takes off from x_i all the others multiplied for the corresponding value A_{ij}, finally we can divide by the diagonal value A_{ii}. It's easy to notice the simplicity of the algorithm and in fact the method is not a safe bet, but a strongly dominant diagonal in the A matrix is a sufficient reason to let $\lim_{k \to +\infty} x^{(k)}$ converge to the solution vector [1].

2 Parallel logic

2.1 Measures

To evaluate our parallel implementation we used the typical metrics:

Speedup $s(n)$ the ratio between the best known sequential execution time (on the target architecture at hand) and the parallel execution time.

$$s(n) = \frac{T_{seq}}{T_{par}(n)}$$

Scalability $scalab(n)$ the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to n.

$$scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}$$

Efficiency (ϵ) the ratio between the ideal execution time and actual execution time.

$$\epsilon(n) = \frac{T_{id}}{T_{par}(n)} = \frac{T_{seq}}{n \cdot T_{par}(n)} = \frac{s(n)}{n}$$

2.2 Preliminary planning

The initial development has been on the sequential code to obtain values on which to make estimates and, in the case, speculate a possible performance sport. As we can acknowledge from the snippet algorithm 1, once computed all the sub-results we have to join them, as the threads that have handled them, so the parallel activity will conclude at each iteration.
From this starting point we start to assume some projections: the sequential code with a $n\_dim = 5000$ and $MaxIt = 1$ takes 25000 μsec, each thread takes around 20 μsec to start and to join, so with a $nw = 100$ we can grossly calculate an overhead:

$$\phi = (T_{start} + T_{join}) \cdot nw = 20\mu sec \cdot 100 = 2000\mu sec$$

this value in conjunction with the ideal speedup formula:

$$T_{id} = \frac{T_{seq}}{n} = \frac{25000}{100} = 250$$

permit us to state an approximation of the speedup's upper bound $\Omega(s(n))$; the T_{id} summed with our overhead (ϕ) permit us to esteem the T_{par}, that inside the speedup formula gives:

$$\Omega(s(n)) = \frac{T_{seq}}{T_{id} + \phi} = \frac{25000}{2000 + 250} \approx 11$$

This is considered an upper bound because it involves only the management of the threads the cache hits, synchronization points, scheduling and more overheads have been omitted to keep an high level computation.

We would like to emphasize that the discussed implementation has the only purpose of showing the application of parallelism in commonly used algorithms, therefore the *MaxIt* (maximum iterations) have been left as input from the CLI, but usually the methodology converge in not so many iterations reaching a *cosine similarity* greater then 0.995 between the $x^{(k)}$ vector and the solution vector, so the main parallelization concern should involve the computation of the next x^{k+1} step itself [2].

2.3 Building Blocks

The code has some points from which we should take care of. These points suggest us to create a **pipeline(farm(nw), seq, seq)** with a 1_{st} stage in which we can do a **reduce** of the summation in the row vectors, then send it to the 2_{nd} stage in which the value is subtracted by b and then divided by the a_{ii} diagonal, finally take the result and save it in the proper position of the temporary vector that can be swapped as soon as the master has received back the **EOR** (End Of Rows).
Eventually the 2_{nd} and 3_{rd} phases can be collapsed because the most expensive computation is the reduce one, as well as the emitter and collector can be merged, using a **wrap around**, shrinking as low as possible the communication. The schema would look like:

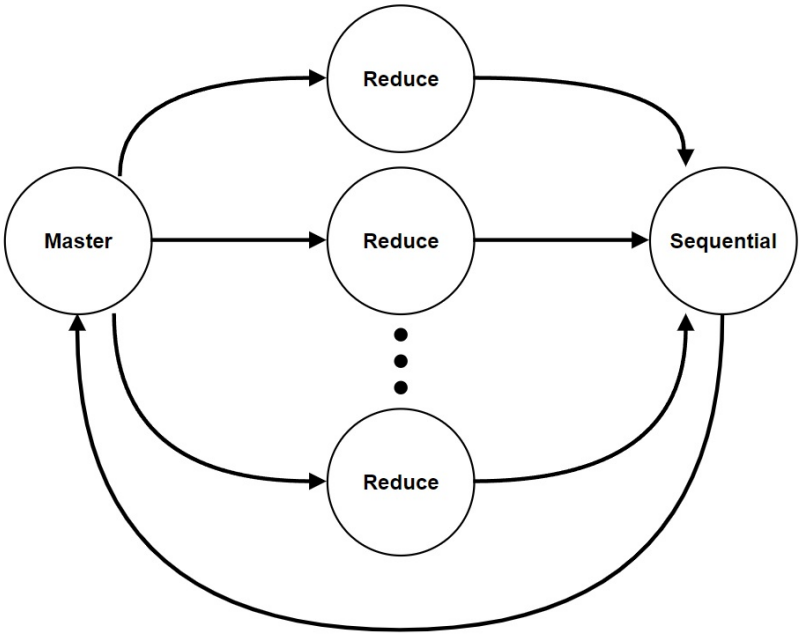

Figure 1: First parallel schema Pipeline(Map(Reduce), Seq)

This implementation, in our opinion, is not efficient for different reasons: first of all the data are all available since the beginning, we are dealing with a proper **Data Parallel Pattern**, so there is no benefit in dividing the method in phases searching to overlap their *latency*; the stages are not balanced so we should shrink more and more the heaviest stage to improve the *service time*, therefor the *completion time*, finishing to waste resources; finally when the EOR signal has been sent all the stages should finish their work and send it back to the master, increasing the synchronization bottleneck.

2.4 Refactoring

So from the precedent building blocks we opted for a solution in which all the phases are computed in a **stencil** of **composition** with the division of data among resources in order to synchronize the results to iterate again. In this way we can use all the *workers* to shrink the heaviest part of the methodology without useless consumption. The schema is the following one:

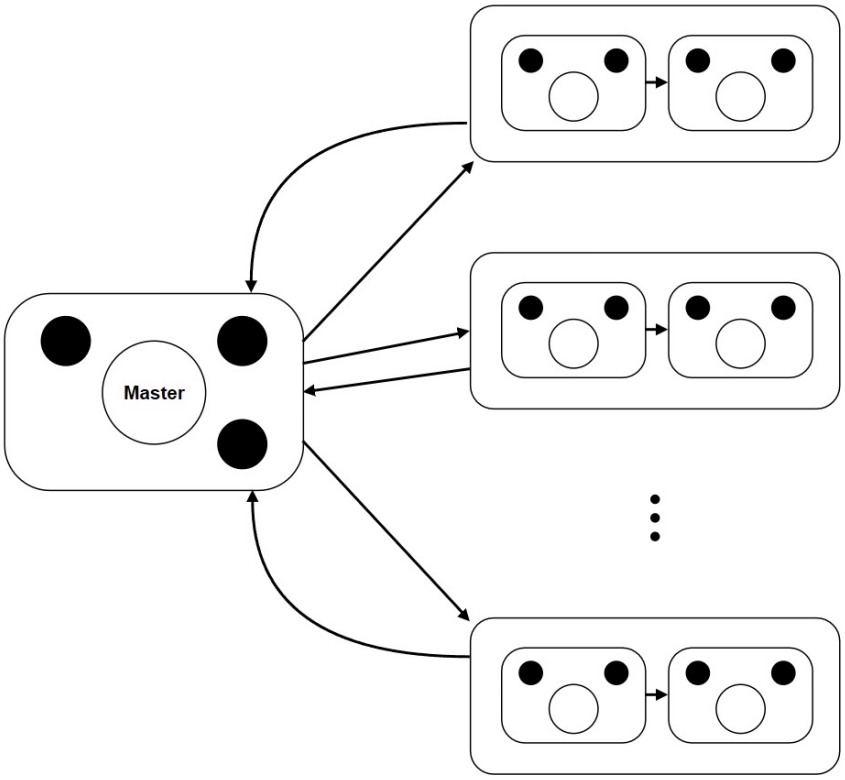

Figure 2: The final schema adopted, Stencil(Composition(Seq, Seq))

In this way, as soon as the *master* receives all the rows of the matrix it can provide the update of the temporal vector. This phase is the crucial one, any parallel implementation we can think of will suffer from this synchronization. So our strategy is to start all the threads together in such a way that they would end to compute alongside, this could prove to be a success given that the load of work is balanced, as in our case.

3 Implementation

The project is divided in 2 main phases for each of the `.cpp` files, first of all we build the linear system that has to be solved, than we adapt the schema illustrated in the Parallel Implementation section to the `std::library` and `ff::library` respectively.

3.1 Linear System

The initial step is the creation of a linear system solvable by the Jacobi method, so with a strongly dominant diagonal. The following code is a representation of what the `linear_system.hpp` file does in the constructor of the class:

Algorithm 2 Creation of the linear system

```
1:  procedure LINEAR SYSTEM(n_size, print)
2:      for i=1, n_size do
3:          b(i) := 0
4:          xold(i) := 0
5:          x(i) := 0
6:          sol(i) := rand(HI, LO)
7:          for j=1, n_size do
8:              if i = j then
9:                  A(ij) := (LO · n_size) + rand
10:             if i ≠ j then
11:                 A(ij) := rand(HI, LO)
12:     for i=1, n_size do
13:         for j=1, n_size do
14:             b(i)+ = A(ij) · sol(j)
```

A solution vector is created randomly to fill out, in the bottom loop, the solution vector b. The HI and LO values are *const* that are used to limit the dimension of the values, the diagonal ones are summed by $LO \cdot n\_size$ to keep the $||a_{ii}|| > \sum_{i \neq j} a_{ij}$.

3.2 Standard Library

The `std::library` implementation takes advantage from the `std::barrier` class, giving as inputs in the constructor the *nw* (number of workers) and the `update()` function that takes care of the synchronization. Then the `std::thread`s are started in a loop that terminates when the *MaxIt* (maximum iterations) is reached. In that loop is present the normal Jacobi function and in the end the `std::barrier.arrive_and_wait()` call. This implementation has been preferred because with a substantial number of items of vectors to deal with, this synchronization point is an overhead almost negligible. As the measurements have proved it takes about 500 μsec to synchronize and update at the `std::barrier`, this is an average taken from 100 iterations, the threads have been waited each other and we realized that the waiting time is uniformly, another confirm of the balance of the computation.
The split policy adopted is the **cycle** one in order to avoid the arranging of the blocks, also we don't deem necessary a dynamic scheduling for the precedent affirmation.

This method has been compared to a thread pool system in which periodically each thread is assigned with a `std::future` binded with an index, then it is called with a `get()`, to ensure that all the functions have been performed to do the synchronization step. We thought that this method would sport better since the functions are only "checked" by the master thread, but at the end of the day the master is standstill waiting all the threads and the overhead is focused only to him, concluding with a waste of time.

3.3 FastFlow

This implementation was a little bit different, taking advantage of the well done handling of the threads by the library itself. We took chance to build a simple `ff::ParallelFor` to divide the loop equally to the available processes and then `update()`. This kind of implementation is possible only because the processes are not joined as it would occur with the standard library, but are setted in a wait state, ready to perform new tasks.

4 Results and Discussion

4.1 Experimentation protocol

The experiments have been conducted in the machine arranged in the department of Information Technology at the UNIPI with an Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz 32 cores - 4 hyper threading, therefore with 128 available threads. The executable are: `jacobi_std`, `jacobi_fastflow`, `jacobi_sequential`. To build them just type `make all` and to remove `make clean`. To reproduce the experiments run `bash ./experiment/gustaffson.sh`, `bash ./experiment/amdahl.sh` and `bash ./experiment/average.sh`, instead to run the executable type (for the sequential one omit "nw"):

```
./build/"exec_name.cpp" n_dim max_it nw print_results[0/1]
```

4.2 Results

Following is reported the summary of the experiments conducted that shows how much the dimension of the matrix is important for the usability of the parallelism in this problem, notice that the number of iteration has been left the same for all the different dimensions ($MaxIt = 100$), of course with a different number of iterations the gains of the speedup would have been different, because more iterations means more synchronization overheads, so the balance between the number of threads and the work to each of them would have changed as well.

Size	Sequential	Best	
		Barrier / nw	FastFlow / nw
1280	261'063 μsec	58'700 μsec / 12	51'171 μsec / 16
2560	1'060'564 μsec	201'090 μsec / 19	164'338 μsec / 59
5120	4'288'950 μsec	591'512 μsec / 27	489'808 μsec / 59

Table 1: As we can see the FastFlow has a better optimization of the threads life in order to avoid as much as possible the overheads.

The following graphs show the curve of the measures mentioned in Section 2.1 when the number of workers are increasing:

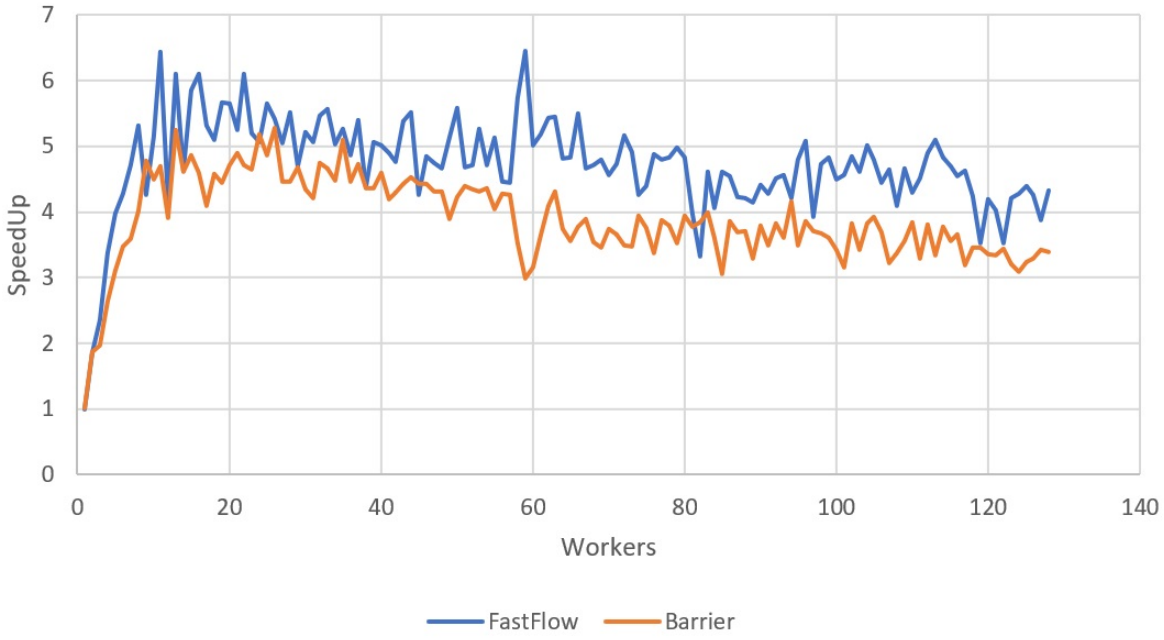

Figure 3: Speedup N = 2560

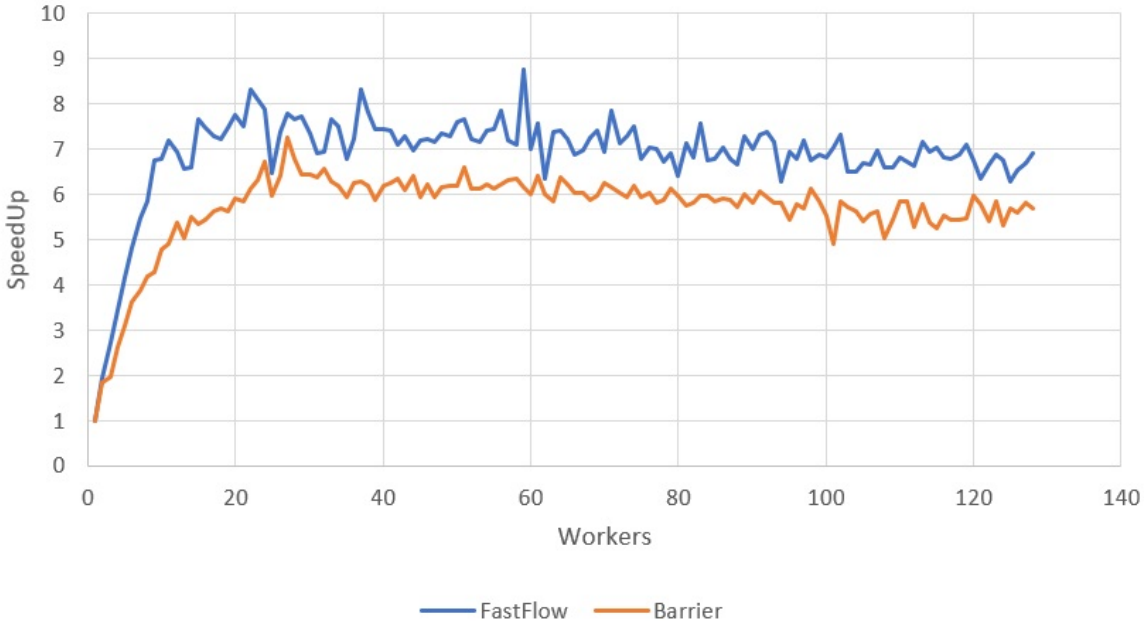

Figure 4: Speedup N = 5120

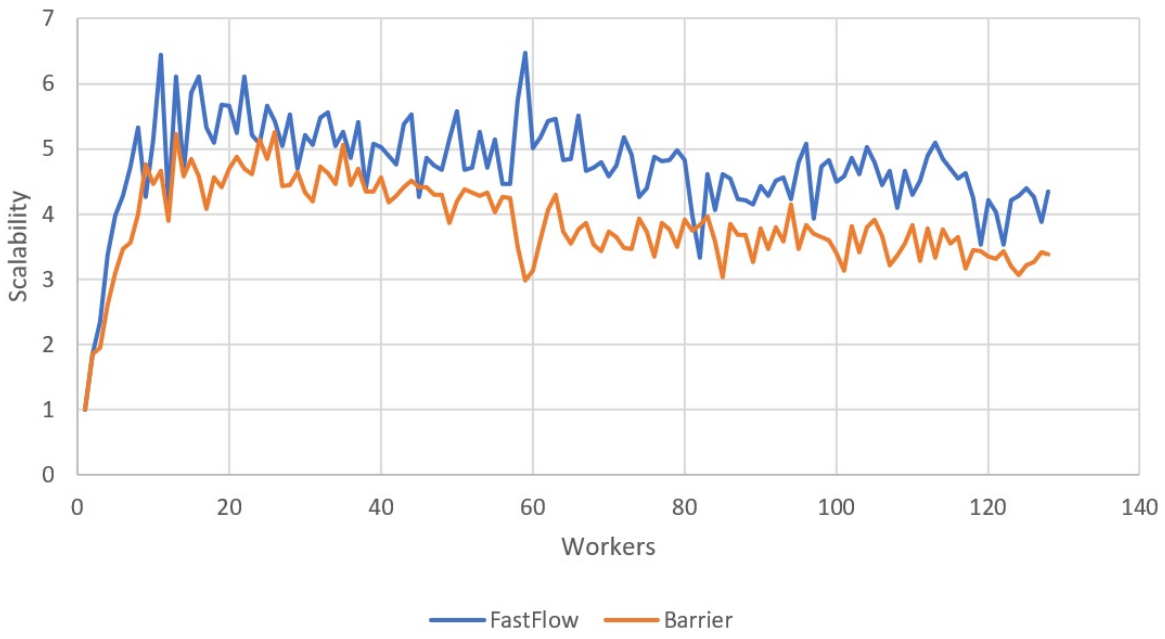

Figure 5: Scalability N = 2560

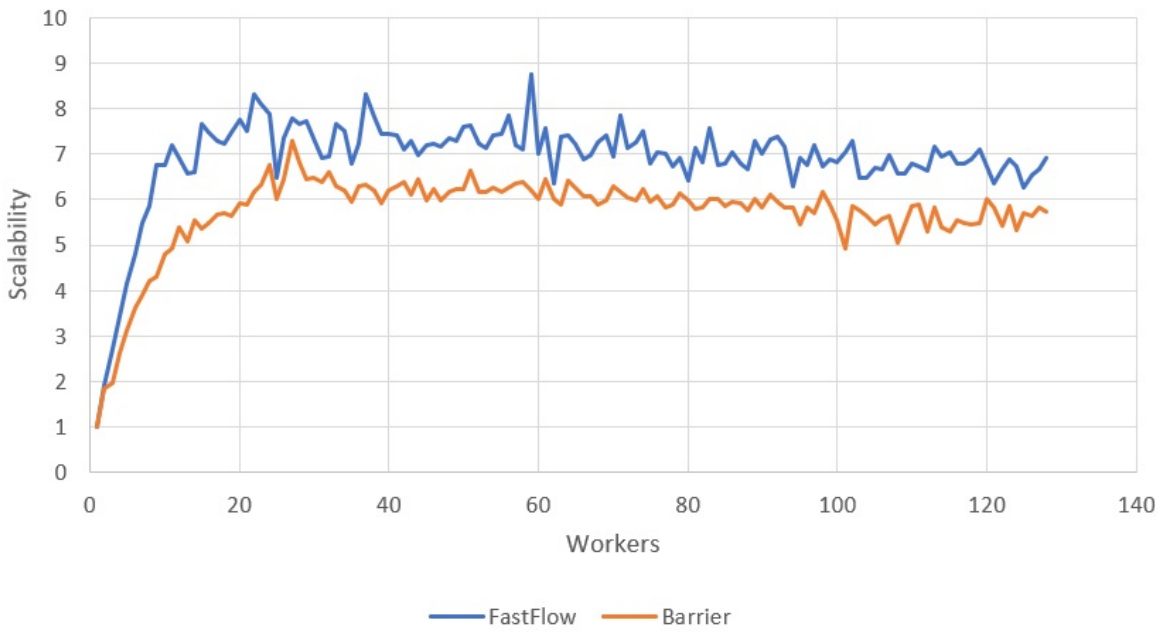

Figure 6: Scalability N = 5120

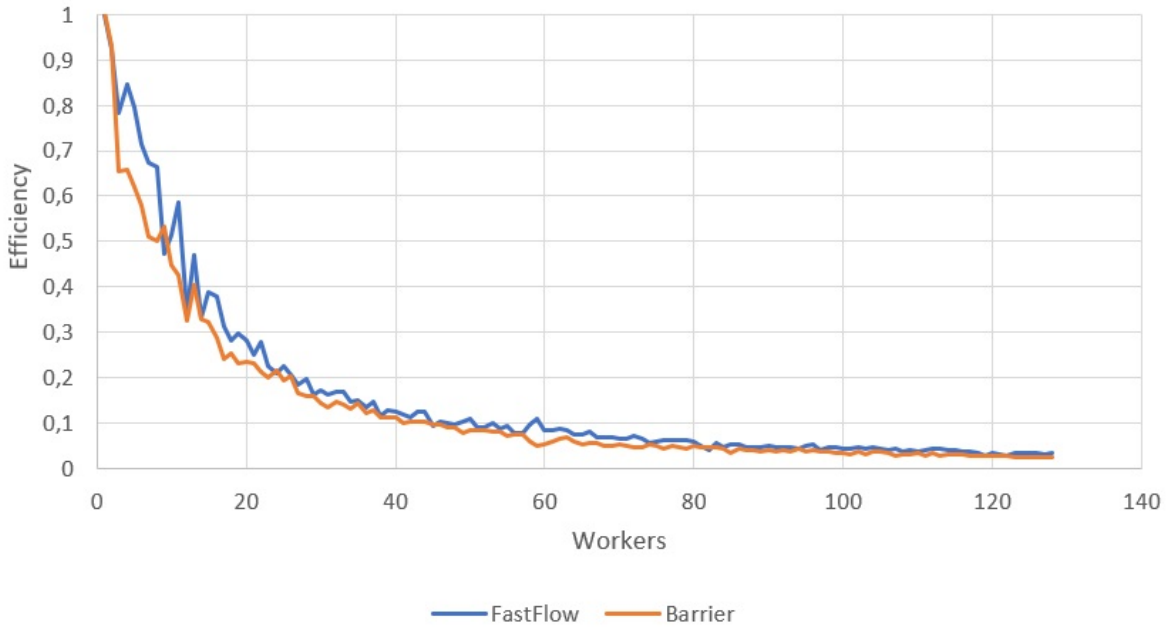

Figure 7: Efficiency N = 2560

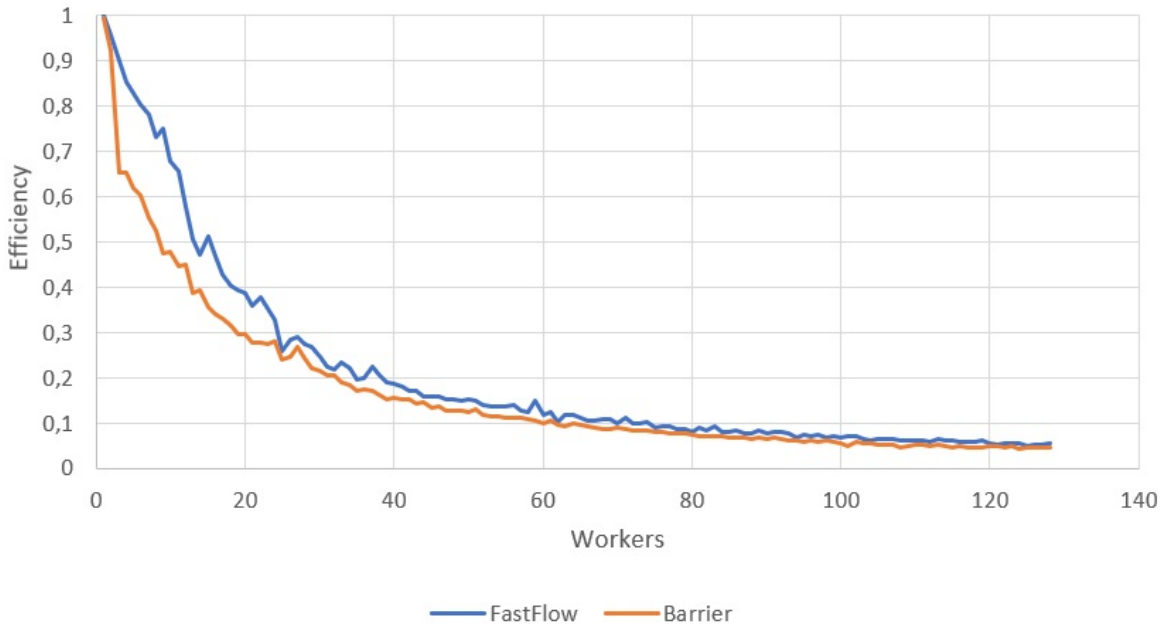

Figure 8: Efficiency N = 5120

Size	Implementation	Speedup	Scalability	Efficiency
1280	Barrier	4,4474	4,5842	0,9701
	FastFlow	5,1017	5,0510	1,0100
2560	Barrier	5,2740	5,2521	1,0041
	FastFlow	6,4535	6,4666	0,9979
5120	Barrier	7,2508	7,2890	0,9947
	FastFlow	8,7563	8,7605	0,9995

Table 2: As we can see the FastFlow implementation sports better, maybe having a better threads life management.

4.3 Conclusion

As we can see from the graphs illustrated the FastFlow implementation is performing above the implementation with the Standard library with a maximum speedup of about 8,75 against 7,25. We can state that the reasons are different: the PIN of the thread on specifics input data taking advantage of already present cache lines, the lock-free interaction to swap the support vector and the fact that it avoids hits of *cache coherency protocol*. Of course with the standard approach the barrier synchronization point is an overhead that can be significant for small matrices with great number of iterations, but as we stated in Section 2 the goal is to find the point of convergence, whom will need a significant number of iteration only with a very huge matrix, which will, in any case, lead to an effective gain in parallelization.

We would like to underline that a problem like this without an efficient lock-free method that can parallelize and maybe even hide the swap of the vectors at the end of each cycle it's impossible to reach the ideal speedup $T_{ideal} = \frac{T_{seq}}{nw}$, but all said we can be satisfied of the results obtained reaching almost our optimal predictions.

References

[1] Roberto Bagnara. A unified proof for the convergence of jacobi and gauss–seidel methods. *SIAM review*, 37(1):93–97, 1995.

[2] Marco Aldinucci, Marco Danelutto, Maurizio Drocco, Peter Kilpatrick, Claudia Misale, Guilherme Peretti Pezzi, and Massimo Torquati. A parallel pattern for iterative stencil+ reduce. *The Journal of Supercomputing*, 2018.

Appendix

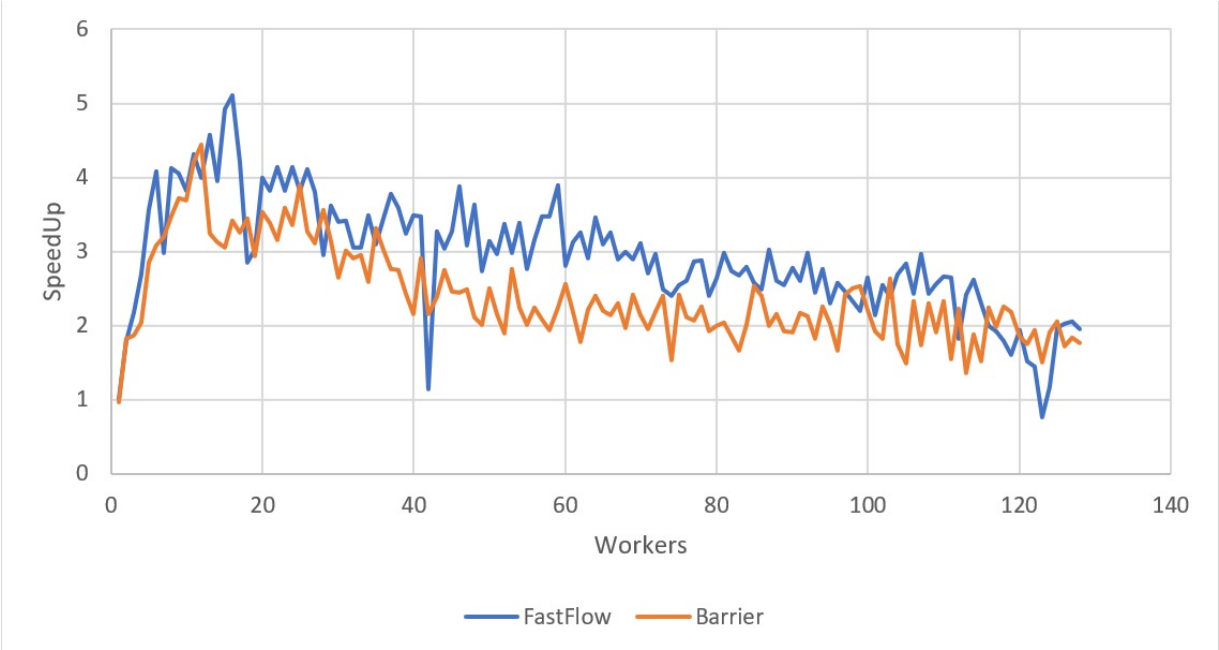

Figure 9: Speedup N = 1280, it's important to acknowledge the weight of the cardinality of the matrix with the number of iterations, we can see that rising the number of workers to a level in which each thread computes only 10 rows and then has to synchronize 100 times, is useless and almost vanish the speedup

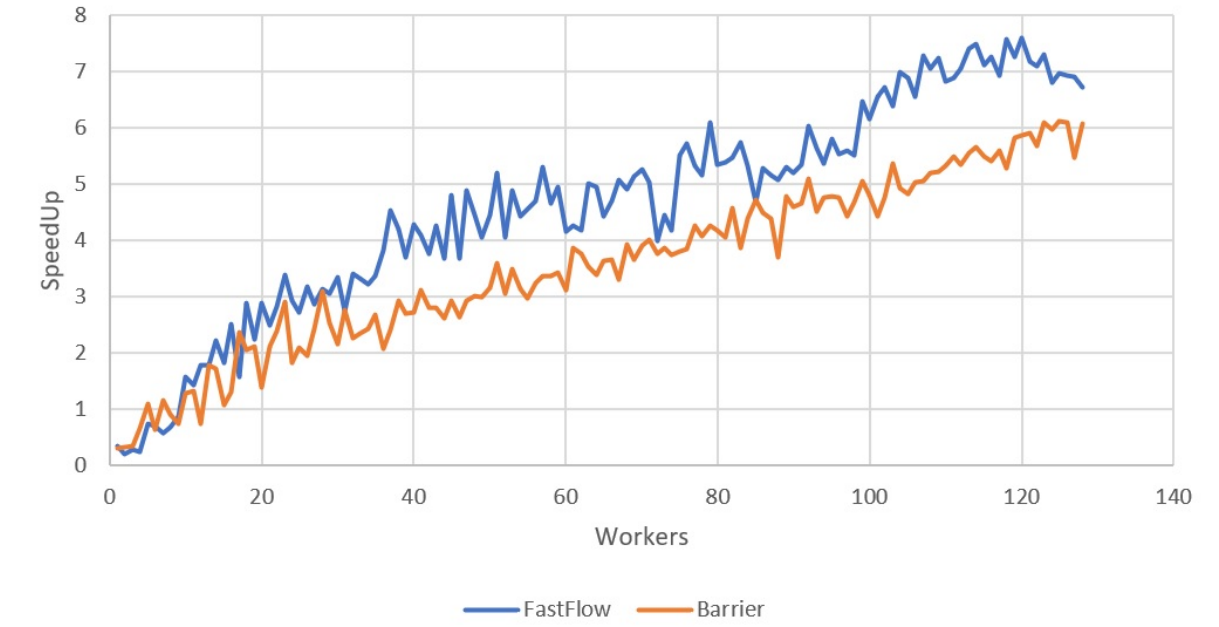

Figure 10: Speedup compared with the Gustafsson Law (n=nw · 40), in which the dimension of the matrix grows linearly with the number of workers, the picture is trivial and it's normal that the speedup would continue to rise with a sequential time that can only go slower