Università di Pisa

Parallel and distributed systems: Paradigms and Models

# A parallelized version of the Jacobi Method

Gennaro Daniele **Acciaro** (635009)

github.com/gdacciaro/SPM22_Acciaro_Project

August 20, 2022

**Abstract**

The Jacobi Method is an algorithm for finding the solution of a system of linear equations $Ax = b$. This C++ project aims to implement a sequential version of this algorithm, a parallelized version using native threads, and finally a parallel version with Fast Flow.

## Contents

## 1 Introduction

The **Jacobi Method** is an algorithm for finding the solution of a system of linear equations $Ax = b$, where $A \in R^n \times R^n$ is the matrix of coefficients of the system of linear equations, $b \in R^n$ is a vector of known values and $x \in R^n$ is the vector we want to approximate. At each iteration $k$, we calculate the new values contained in the vector $x$ using the following formula:

$$x_i^{k+1} = \frac{1}{a_{(i,i)}} b_i - \sum_{i \neq j} a_{(i,j)} x_j^k \qquad \forall i \in 1 \dots n \tag{1}$$

The pseudo-code of the algorithm, which is based on this equation, is defined in **Alg.1**.

It has been demonstrated that if the matrix $A$ is *strictly diagonally dominant*, Jacobi method is guaranteed to converge. For this reason, I assumed that all the matrices examined respect that property.

The purpose of this project is the C++ implementation of this algorithm in parallel version, in order to achieve the same results with less time.

This report is organized as follows: In section **2** I described the theoretical analysis of the parallelization of the Jacobi method. In section **3** I described the methodologies used to solve the task. In section **4** I showed the results. In section **5** I concluded the work.

## 2 Analysis

The algorithm uses two vectors $x$ and $tmp\_x$, referring to **Eq.1**, the first vector represents $x_i^k$ while the second vector represents $x_i^{k+1}$.

The values of the vector $tmp\_x$ are computed at each iteration and only at the end of an iteration do we perform the swap to $x$. Thus, in order to compute each value of $tmp\_x[i]$, we need to know the corresponding previous value $x[i]$. This means that it is not possible to parallelize the cycle `while` (referring to **Alg.1**). However, the remaining `for` loops can be parallelized.

This algorithm is a perfect instance of **data parallel pattern** since for each iteration of the first `for` loop (lines $10 - 18$) we have already everything we need to compute the cycle (that is, the data contained in the vector x). In particular we can recognize *map* pattern, since we can apply the Jacobi method to each item of the vector independently.

Moreover, the second `for` loop (lines $12-16$) can be seen as a *reduce* parallel pattern, since we have to perform an operation, the sum, that is commutative and associative. We could then perform partial sums in parallel and accumulate them with each other, as reduce pattern dictates. In practice, for this kind of problem, it is not very useful because adding this parallelization also means adding a lot of overhead.
For the same reason, I also did not parallelize the norm calculation, which is necessary for early stopping, which in theory, it is possible to parallelize. In parallel implementations, in light of the fact that each worker works (more or less) on a portion of similar size compared to all other workers, there was no need to implement load balancing techniques.

As described in more detail in the 3.2 section, the experiments on the different implementations were performed using different metrics. All of these metrics, however, are based on two values: $T_{seq}(n)$, $T_{par}(n, nw)$ which are the times required to perform the Jacobi method sequentially and in a parallel way, respectively, on a matrix of dimensions $n$ (and $nw$ refers to the number of workers). We now examine these two values in more detail.

Let $M$ be the number of iterations.
Due to the way the algorithm was designed, the sequential time $T_{seq}(n)$ can be seen in this way:

$$T_{seq}(n) = T_{init} + M \cdot (n \cdot T_{single\_iter} \cdot T_{ES}) \tag{2}$$

where $T_{init}$ is the time for initializing the variables, $T_{single\_iter}$ is the time for a single iteration and $T_{ES}$ is the time to compute the norm for early stopping.

Instead, the time to run the algorithm in parallel, $T_{par}(n, nw)$, can be seen as:

$$\begin{aligned} T_{par}(n, nw) = T_{init} + T_{split} + T_{merge} \\ + M \cdot (\frac{n}{nw} \cdot T_{single\_iter} + T_{ES}) \end{aligned} \tag{3}$$

where $T_{split}$ and $T_{merge}$ are the overheads for splitting and merging the whole iteration among the workers.

# 3 Methods

## 3.1 Setup

The software starts by parsing the arguments [`utils/argument_parser.cpp`], if they are specified. If the program is compiled without making the arguments explicit, the default values for each argument are obtained using the constants defined in `setup/constants.h`. For more information on compilation options, see appendix B.

At this point, the software runs unit tests; these were very useful for me to make sure, while writing the project, that the functions tested were correct in their required behavior. Unit tests are managed by the `UnitTestManager` class.

Once the unit tests have been run, the software generates a strictly diagonally dominant matrix $A$ and a vector $b$, both with random values distributed in a range [`RAND_LOW_VALUE`, `RAND_HIGH_VALUE`], where these values are constants in `setup/constants.h`.

Once the matrix $A$ and vector $b$ have been generated, the software executes the Jacobi method on these values, in particular, it first executes the sequential version `solutions/sequential.cpp`, then the parallel version with native threads `solutions/parallel_threads.cpp`, and finally the parallel version with Fast Flow `solutions/fast_flow.cpp`

In the end, if required from the user, the software performs the experiments that are described in section 4.

## 3.2 Metrics

In order to evaluate the quality of the different parallel versions of the sequential algorithm of Jacobi Method, I selected three performance indicators: **Speedup**, **Scalability** and **Efficiency**.

Each of these metrics was evaluated with respect to completion time; actually, the completion time is defined as the time required by an algorithm from when it starts until it finishes. In the case of this project, however, I thought it appropriate to exclude from this time all the calculations to perform the unit tests, the initialization of the $A$ matrix and the $b$ vector, and in general, anything that was not strictly related to the resolution of the algorithm itself.
Let us briefly look at these metrics:

**Speedup** the ratio between the best sequential execution time and the parallel execution time. Speedup gives a measure of how good is our parallelization with respect to the "best" sequential computation.

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

**Scalability** the ratio between the parallel execution time with parallelism degree equal to 1 and the parallel execution time with parallelism degree equal to $nw$.

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

**Efficiency** the ratio between the ideal execution time and actual execution time.

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

where $n$ is the parallelism degree.

## 3.3 Sequential

The first version of Jacobi Method that I implemented was the sequential version. It is, in essence, an exact translation of the pseudo code found in appendix A. The implementation of this version can be found in `solutions/sequential.cpp`.

## 3.4 Native C++ Threads

The first way I thought of in order to parallelize the algorithm was to use a `barrier` in order to synchronize several threads, which work simultaneously to compute the values of $tmp\_x$ (each thread works on different portions of the vector, independently). When all the threads finish, using the barrier, I updated the vector $x$. The barrier was implemented using the class `std:barrier`. The use of barriers introduced an overhead that was described in 4. The implementation of this version can be found in `solutions/parallel_threads.cpp`.

## 3.5 Fast-Flow

After developing the parallel version of the algorithm with threads, I implemented a parallel version of Jacobi's method using the library `Fast-Flow`[1]. In particular, I used the method `ff::ParallelFor`, designed to parallelize a *map* pattern, specifically it is able to parallelize a lambda function passed as input on a worker number, also specified as input. In our case. the function passed to the method is responsible for computing the individual values of $tmp\_x$. The implementation of this version can be found in `solutions/fast_flow.cpp`.

## 4 Experiments and Results

**Experiments** The experiments I performed were comparisons of the different implementations against the three selected metrics (Speedup, Scalability, Efficiency), and they were performed on servers provided by the University of Pisa that use an Intel Xeon Gold 5120 CPU @ 2.20GHz 32 cores CPU.
Tests were performed using matrices of size: $1024 \times 1024$, $4096 \times 4096$, $16384 \times 16384$.

Each experiment was performed 10 times and then the results were averaged: this was done to obtain more reliable results. Moreover, for a proper comparison, the seed remained constant while running the tests: thus the three matrices (of the three different dimensions) are exactly equal to each other.
To conclude, all experiments were performed with the early stopping flag active. Experiments run in batch mode at the end of main if the argument `results=1` is set when the program is executed.

**Results** Figures 1,2,3 show us, respectively, the values of Speedup, Scalability and Efficicency, obtained on the three dimensions of the matrices.

We can immediately see that the size of the matrices plays a key role in choosing the best implementation, indeed in Fig. 1(a) we can see that the speedup calculated on the matrix $1024 \times 1024$ is very far from an ideal speedup (which is linear with respect to the number of workers). Instead, the matrix $16384 \times 16384$ (Fig. 1(b) ) is closer to an ideal speedup, especially for the first few iterations.

Table 1 shows a comparison of times between native threads and Fast

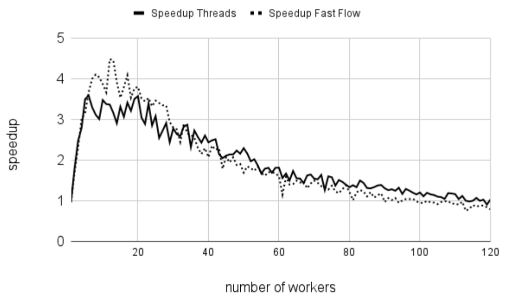
Flow, while Table 2 shows a comparison of metrics between native threads and Fast Flow.
The introduction of the barrier introduces a clear overhead within the implementation with native C++ threads. In particular, we can see in figure 4 that the more we increase the number of workers, the more the overhead due to the barrier increases. Also, the larger the size of the matrix $A$, the greater the overhead. In the worst case, we have an overhead of more than **5.32sec**(!).
In addition, I also calculated the overhead required to perform thread creation and join, in the case of the implementation with native C++ threads. Figure 5 shows that the overhead is linear to the number of workers used, regardless of the size of the $A$ matrix.

| Matrix size | Implementation | Time ($\mu sec$) | $nw$ |
|---|---|---|---|
| 1024 | Native Thread | 2326.6 | 20 |
| | Fast Flow | **1843.6** | 12 |
| | Sequential | 8216 | |
| 4096 | Native Thread | **9579.8** | 56 |
| | Fast Flow | 9594.6 | 30 |
| | Sequential | 115233 | |
| 16384 | Native Thread | 72703.7 | 28 |
| | Fast Flow | **71328.6** | 58 |
| | Sequential | 1360302 | |

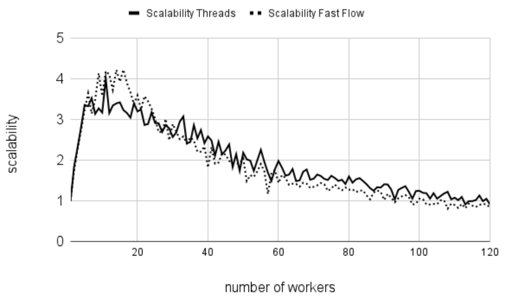
Table 1: Best time

| Matrix size | Implementation | Speedup | Scalability | Efficiency |
|---|---|---|---|---|
| 1024 | Native Thread | 3.58838 | 4.00557 | 0.939837 |
| | Fast Flow | 4.48102 | 4.22221 | 0.939749 |
| 4096 | Native Thread | 12.0923 | 12.2677 | 0.937034 |
| | Fast Flow | 12.2503 | 13.6484 | 0.975051 |
| 16384 | Native Thread | 18.7684 | 17.9763 | 0.98332 |
| | Fast Flow | 19.1659 | 18.7347 | 0.990929 |

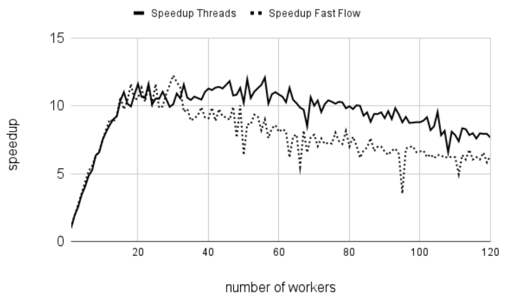
Table 2: Best results



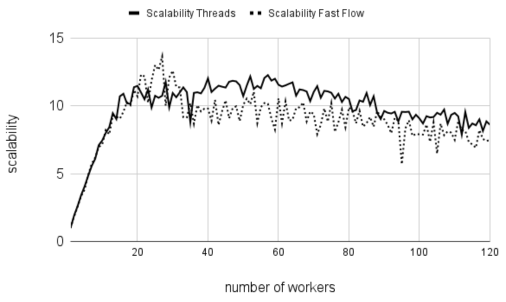
(a) 1024 ×1024



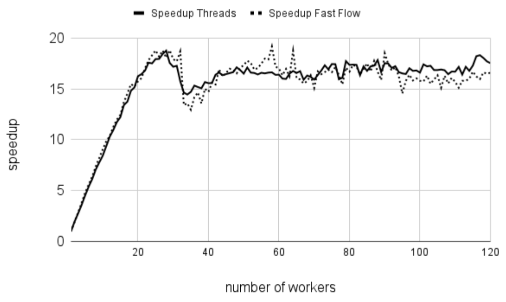
(a) 1024 ×1024



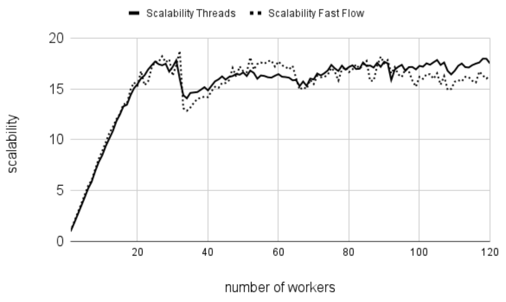
(b) 4096 ×4096



(b) 4096 ×4096



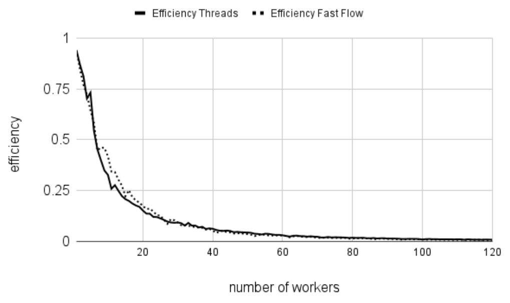
(c) 16384 ×16384



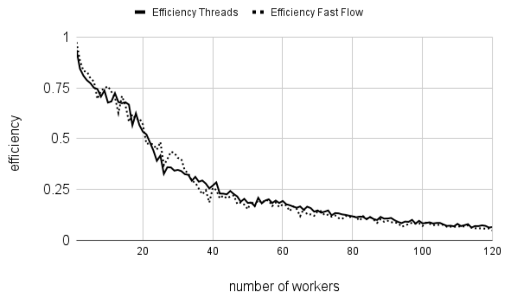
(c) 16384 ×16384

Figure 1: Speedup

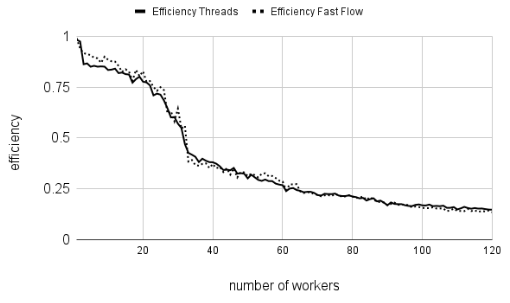
Figure 2: Scalability



(a) 1024 ×1024

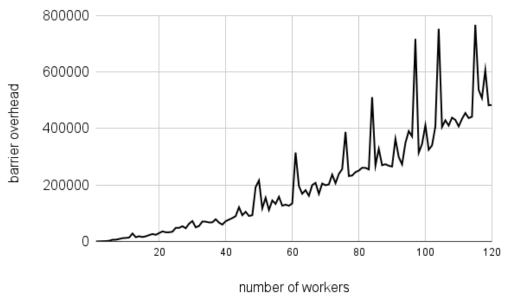


(b) 4096 ×4096

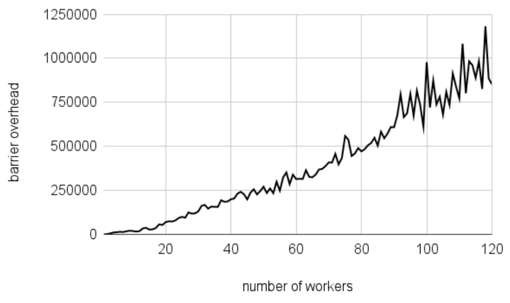


(c) 16384 ×16384

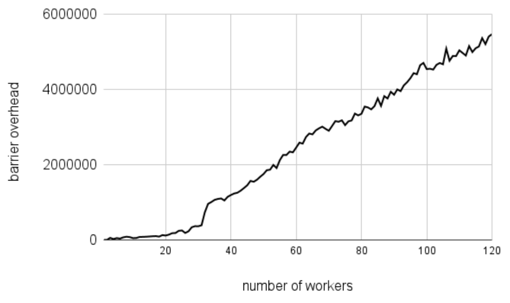
Figure 3: Efficiency



(a) 1024 ×1024

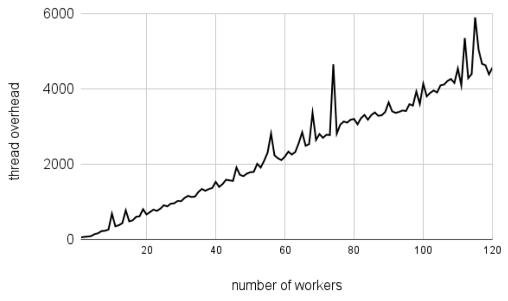


(b) 4096 ×4096

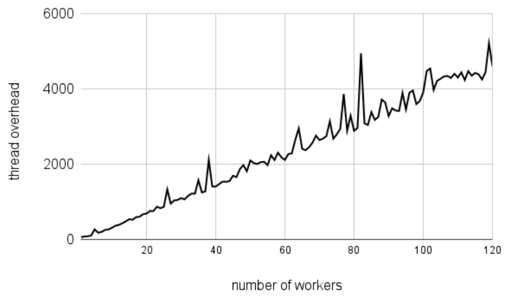


(c) 16384 ×16384

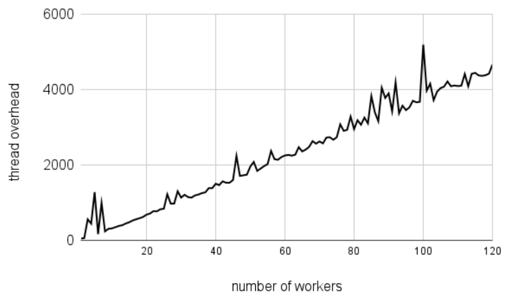
Figure 4: Barrier overhead ($\mu sec$)



(a) 1024 ×1024



(b) 4096 ×4096



(c) 16384 ×16384

Figure 5: Thread overhead ($\mu sec$)

## 5 Conclusions

In this project, an analysis was presented for parallelization of Jacobi's method algorithm on strictly diagonally dominant matrices, through a comparison of two different types of parallel implementations (Fast Flow and native threads). From this comparison we can see that Fast Flow behaves similarly to the implementation with native threads.

If the matrix is very small, it is not worth the effort to parallelize, since the overheads involved in setting up parallelization can make parallel execution slower than sequential execution.

Finally, personally, working on this project made me realize how overheads cannot be taken for granted if we really want to use all available computing power to solve our tasks.

## References

[1] Marco Aldinucci et al. "Fastflow: High-Level and Efficient Streaming on Multicore". In: *Programming multi-core and many-core computing systems*. John Wiley  Sons, Ltd, 2017. Chap. 13, pp. 261–280. ISBN: 9781119332015. DOI: https://doi.org/10.1002/9781119332015.ch13. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119332015.ch13. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119332015.ch13.

## A Jacobi pseudocode

**Algorithm 1** Jacobi Method

```
Require: $\epsilon$                                    ▷ Convergence threshold (default 0.0001)
 1: function JACOBI( max_iterations, $A \in R^n \times R^n$, $b \in R^n$)
 2:     $n = A.size()$
 3:     Initialize $x$ as a vector of $n$ zeros
 4:     while true do
 5:         if iteration = 0 then
 6:             break
 7:         end if
 8:
 9:         $iteration \leftarrow iteration - 1$
10:         for $i \leftarrow 1$ to $n$ do
11:             $sum = 0$
12:             for $j \leftarrow 1$ to $n$ do
13:                 if $i \neq j$ then
14:                     $sum \leftarrow sum + A_{(i,j)} * x_j$
15:                 end if
16:             end for
17:             $tmp\_x_i = \frac{1}{A_{(i,i)}} \cdot (b_i - sum)$
18:         end for
19:
20:         if $distance(x, tmp\_x) < \epsilon$ then                  ▷ Eucl. distance
21:             break
22:         end if
23:         $x \leftarrow tmp\_x$
24:     end while
25:     return $a$
26: end function
```

## B How to compile and execute

To make compilation and execution easier and faster, you can run the bash script:

```
./run.sh
```

If necessary, you can still change any parameter by first compiling the program by running the command:

```
make
```

and then:

```
./main.out <params>
```

where parameters are the following:

| parameter | domain | default value |
|---|---|---|
| **n** | integer $\geq 0$ | 5000 |
| **num_of_worker** | integer $\geq 0$ | 100 |
| **num_of_iterations** | integer $\geq 0$ | 10 |
| **calculate_results** | boolean ($\in \{0, 1\}$) | 0 |
| **early_stopping** | boolean ($\in \{0, 1\}$) | 1 |
| **execute_unit_test** | boolean ($\in \{0, 1\}$) | 1 |

Table 3: Parameters