Parallel and Distributed Systems: Paradigms and Models

Parallel implementation of the Jacobi iterative method

Federico Ramacciotti
f.ramacciotti4@studenti.unipi.it
582646

July 2022

Contents

1 Problem analysis

The Jacobi iterative method is used to solve a system of equations in multiple iterations, getting more and more precise results.
The pseudocode for the algorithm is the following:

Algorithm 1 Jacobi iterative method

```
Require: $A \in \mathbf{R}^{n \times n}$, $B \in \mathbf{R}^n$, $max\_iter$ maximum iterations, $tol$ tolerance
 1: $x^{old} \leftarrow \{0\}^n$
 2: $x^k \leftarrow \{0\}^n$
 3: $k \leftarrow 0$
 4: while $k < max\_iter$ do
 5:     for $i \leftarrow 0, \dots, n$ do
 6:         $sum\_j \leftarrow 0$
 7:         for $j \leftarrow 0, \dots, n$ do
 8:             if $i \neq j$ then
 9:                 $sum\_j \leftarrow sum\_j + (A_{i,j} * x_j^{old})$
10:             end if
11:         end for
12:         $x_i^k \leftarrow \frac{1}{A_{i,i}} * (B_i - sum\_j)$
13:     end for
14:     if $\|x^k - x^{old}\| < tol$ then
15:         return $x^k$
16:     end if
17:     $x^{old} \leftarrow x^k$
18:     $k \leftarrow k + 1$
19: end while
20: return $x^k$
```

In my code I purposely removed the part related to the tolerance (lines 14-15), in order to get a consistent behaviour during the tests, and I only used a maximum number of iterations of 128.

From the pseudocode of the algorithm we can see that the external loop (line 4) has to be done sequentially, since it need previous results (x^{old}) for the next computation. We can instead parallelize the internal loop at line 5 with a *map* pattern: in fact, the rows of the matrix are computed singularly and there are no dependencies among them. Furthermore, we could also parallelize the most internal loop (line 7) with a *reduce* pattern, but I chose not to do it because it would introduce a lot of overhead and would not improve the performance of the method.

1.1 Metrics and cost analysis

For the tests I measured the *speedup*, the *scalability* and the *efficiency*. In this section I analyze the expected values and the bounds for these metrics.
Recalling the formulas:

- **Speedup**: $s(n) = \frac{T_{seq}}{T_{par}(n)}$
- **Scalability**: $scalab(n) = \frac{T_{par}(1)}{T_{par}(n)}$
- **Efficiency**: $\varepsilon(n) = \frac{s(n)}{n}$

The cost of the parallel time, given n size of the matrix, k iterations and nw workers is

$$T_{par}(n, k, nw) = T_{split}(nw) + k * \left(\frac{n * T_{jacobi\_pass}(n)}{nw} + T_{sync}(nw) \right) + T_{merge}(nw)$$

We can see from this function that k and n have the major role on the time consumption on the algorithm, while with bigger matrices the parallel overhead is negligible.

Now, let's make some assumptions on the time of the execution on a matrix of size 1,024 x 1,024 with 32 workers. Assuming that the total overhead is

$$T_{overhead} = T_{split}(32) + T_{merge}(32) + T_{sync}(32) = 25,000 \mu sec^{1}$$

and $T_{seq} = 200,000\mu sec$ [2], we can see that, given $n = 1,024$, $k = 128$ and $nw = 32$:

$$T_{ideal} = \frac{T_{seq}}{nw} = \frac{200,000}{32} \mu sec = 6,250 \mu sec$$

Considering the speedup and the fact that $T_{par} \geq T_{ideal} + T_{overhead}$, its upper bound using 32 workers is

$$s(n) = \frac{T_{seq}}{T_{ideal} + T_{overhead}} = \frac{200,000}{6,250 + 25,000} \mu sec \cong 6$$

[1]These values are calculated with the test `1024-overhead.cpp`, measuring the total time spent to fork, sync and join a growing number of threads. See Chapter 3.2.1 for more details about the test.

[2]Computed with the sequential algorithm on a 1,024 matrix.

2 Implementation

I made a single program that executes all three versions for the sequential, standard and FastFlow method, managed with some `#define` and `#ifdef` directives[3].

There are no external libraries other than FastFlow, and the code is structured in three files: `main.cpp` contains the main function that reads from input and calls the Jacobi method, in `jacobi.cpp` there are the three versions of Jacobi and in `utils.cpp` there are auxiliary functions used in the main, like the matrix generation.

The program works as following: the main reads from the arguments of the execution the size of the matrix, the seed and the number of workers; then, it generates, with elements in $[-128, 128]$, a random matrix A and a random vector B with the seed passed by parameter[4]. There is no need to check if the matrices are diagonally dominant, since the generation functions guarantee that. After that, the program simply calls the three versions of the Jacobi method and measures the time they take.

In order to copy the vector between the implementations, I used the `swap` function of the standard vector library. As we can see in the cppreference, it has a constant complexity of $O(1)$. An alternative to this function could have been, only in the standard threads implementation, to create another barrier and let threads modify singularly their value of the vector; I chose not to implement it this way since it would have the same complexity of the `swap` while adding some overhead due to the barrier.

2.1 Sequential implementation

In the sequential implementation I simply implemented the pseudocode reported in the first Section, and it corresponds to the function `jacobi_seq` in the file `jacobi.cpp`.

2.2 Standard threads

For the parallel version with the standard C++ threads you can refer to the function `jacobi_par_std`.

I implemented this version of the Jacobi method using a *barrier* as a synchronization point, to wait for all the threads before proceeding to the next iteration. With the barrier, as you can see in the following code, I synchronized the threads, decremented the iteration variable and copied the new vector `x` into the old one.

```
std::barrier sync_point(nw, [&]() {
    iter--;
    x_old.swap(x_new);
});
```

Then, I created a vector of ranges representing the start and stop indices for each thread to work on. After that, I launched all the threads with a lambda function and joined them afterwards:

```
for (size_t i = 0; i < (size_t)nw; i++) {
    threads[i] = std::thread(
        [&](std::pair<size_t, size_t> range) {
            while (iter > 0) {
                for (size_t j = range.first; j <= range.second; j++) {
                    double sum_j = 0;
                    for (size_t l = 0; l < n; l++) {
                        if (l != j) {
                            sum_j += A[j][l] * x_old[l];
                        }
                    }
                    x_new[j] = (1 / A[j][j]) * (B[j] - sum_j);
```

[3]See Section 4 for more details on the build and run cycle.

[4]I used the seed in order to have consistent tests with the same matrices. For general testing, if set to 0 the program generates 'really' random matrices with `srand(time(0))`.

```
                }
                sync_point.arrive_and_wait();
            }
        },
        ranges[i]);
}
```

2.3 FastFlow

The FastFlow version (`jacobi_par_ff`) is straight-forward and only needs the `ParallelFor` class and the sequential code, thanks to the main purpose of the library for the code re-use. For the parameter of the `parallel_for` function, I defined the first and last index of the matrix with a step size of 1, *nw* workers, and a chunk size of 0.

```
ff::ParallelFor par_for(nw);
for (size_t k = 0; k < (size_t)max_iter; k++) {
    par_for.parallel_for(
        0, n, 1, 0,
        [&](size_t i) {
            // Jacobi pass
            double sum_j = 0;
            for (size_t j = 0; j < (size_t)n; j++) {
                if (i != j) {
                    sum_j += A[i][j] * x_old[j];
                }
            }
            x_new[i] = (1 / A[i][i]) * (B[i] - sum_j);
        },
        nw);

    x_old.swap(x_new);
}
```

3 Test results

The test made are two different types: measuring time and speedup with fixed workers and growing matrices, and measuring overhead, time, speedup, scalability and efficiency with fixed matrices and growing workers.

3.1 First test: fixed workers, growing matrix

For the first test I used the maximum cores of the CPU (in the virtual machine is 32) and I measured the time and the speedup for a matrix of growing dimension.

3.1.1 32 workers, matrix from 64x64 to 5120x5120

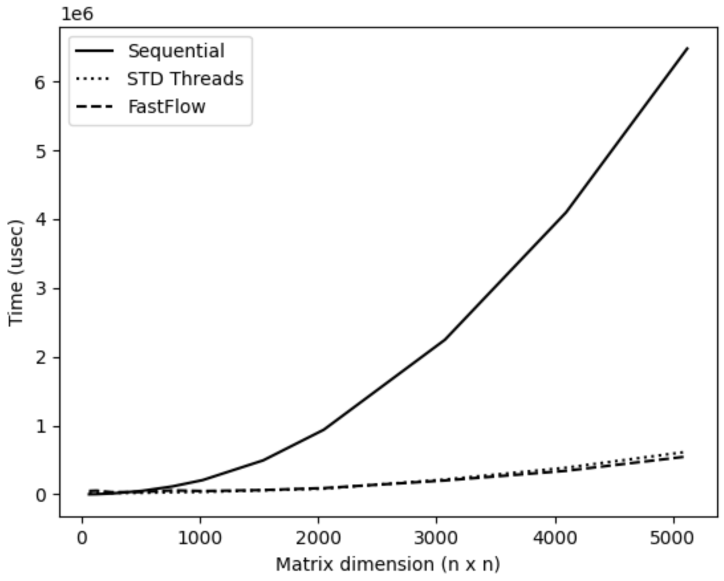


Figure 1: Time

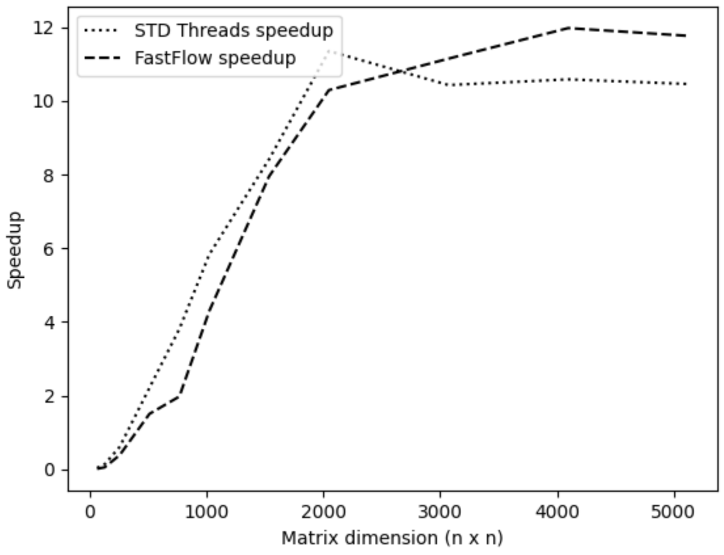


Figure 2: Speedup

We can see that both parallel implementations are better (i.e. have a higher speedup) when the matrix is bigger; this is because, given that the threads introduce some overhead, when the matrix is small the overhead is noticeable, while with bigger matrices it becomes almost negligible.

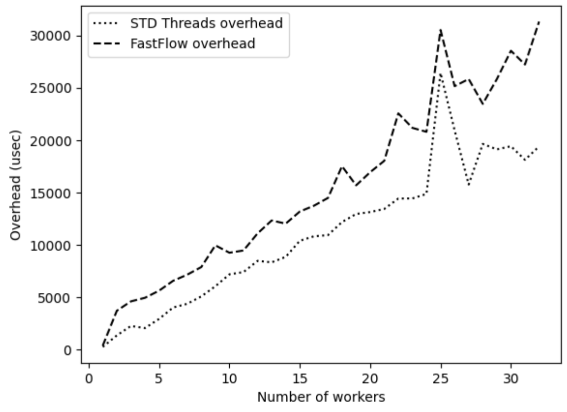
I also measured the efficiency of this tests but I did not include them, since the trend of the results is exactly the same as the speedup (using always 32 workers, $s(32)/32$ is the same function scaled down).

3.2 Second test: fixed matrices, growing workers

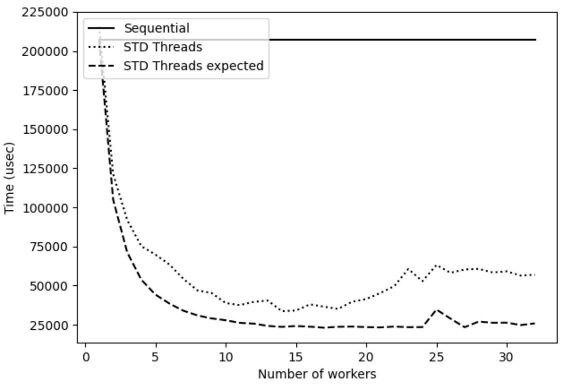
For the second test I found the maximum cores of the CPU (in the virtual machine is 32) and I measured time, speedup, scalability and efficiency with a growing number of workers on fixed matrices of dimensions 1024, 5120 and 10240.

3.2.1 Matrix 1024x1024, workers from 1 to 32

I used this test also to measure the overhead of the parallel implementations and compare the results with the real times I got in the tests. To measure the overhead I used a `define` directive that does not execute the Jacobi passes and only executes the code to setup the threads and the internal (now empty) cycles. The following graphs compare the overhead of both implementations and the expected times (computed with $(T_{seq} + T_{overhead})/nw$) versus the real ones and show the other metrics.

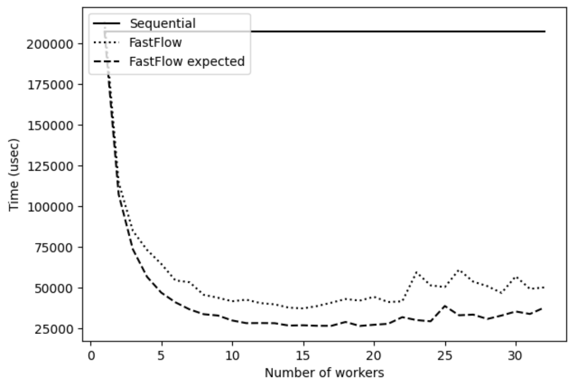


(a) Overhead

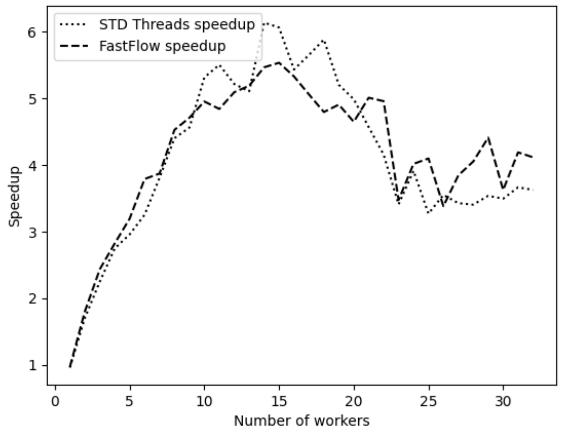


(b) STD version - Expected vs Real

Figure 3: Overhead and STD version - 1024x1024

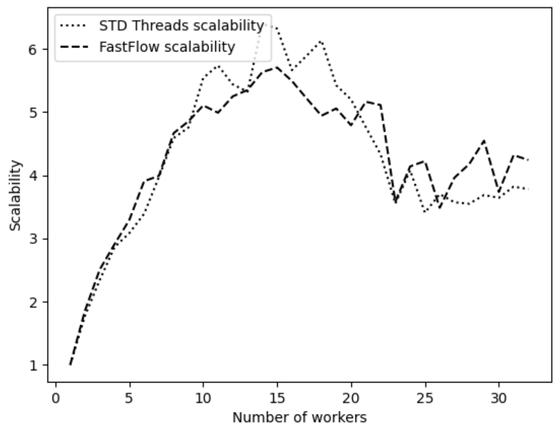


(a) FF version - Expected vs Real

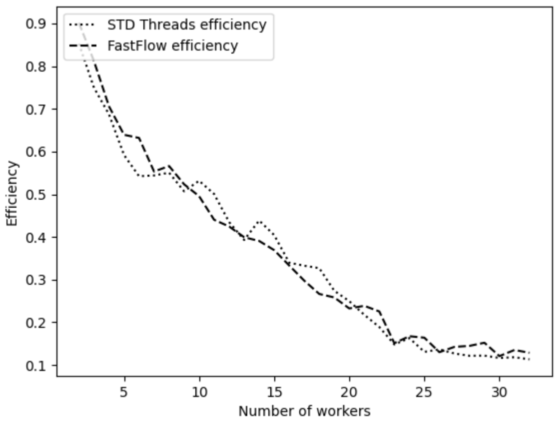


(b) Speedup

Figure 4: FastFlow and Speedup - 1024x1024



(a) Scalability



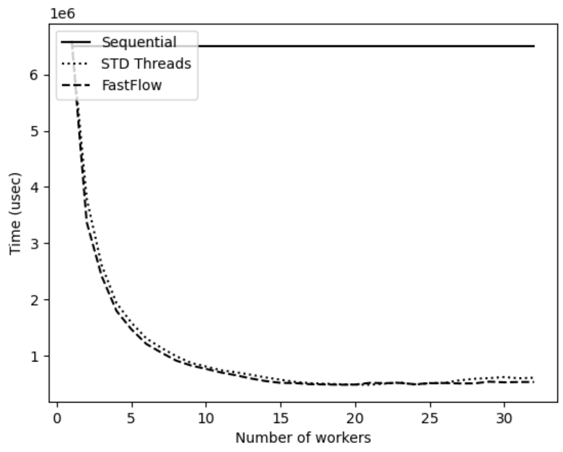
(b) Efficiency

Figure 5: Scalability and Efficiency - 1024x1024

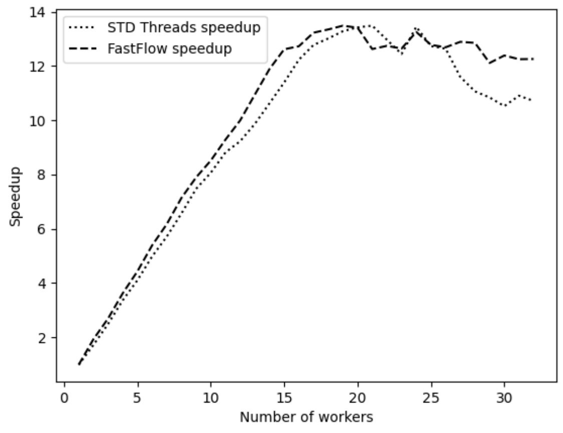
The first graph shows that the overhead introduced by the parallel versions of the algorithm grows quite linearly with the number of workers introduced and that, as we expected, FastFlow introduces more overhead. Other than that, the second and third graphs show how FastFlow is closer to the expected values than the standard implementation when working with a high number of workers.

With a matrix this small we can see that the parallel versions of the algorithm perform quite well and, in relation to the upper bound on the speedup with 32 workers found in Section 1.1, this implementation achieves almost the maximum possible. We can also see that with more than circa 20 workers the speedup drops a little bit, due to the higher overhead of the larger number of threads.

3.2.2 Matrix 5120x5120, workers from 1 to 32



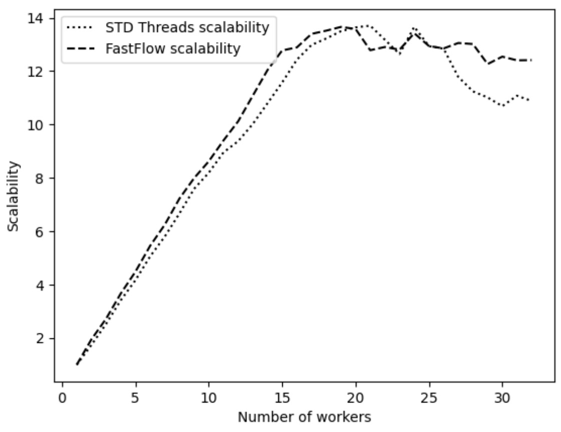
(a) Time



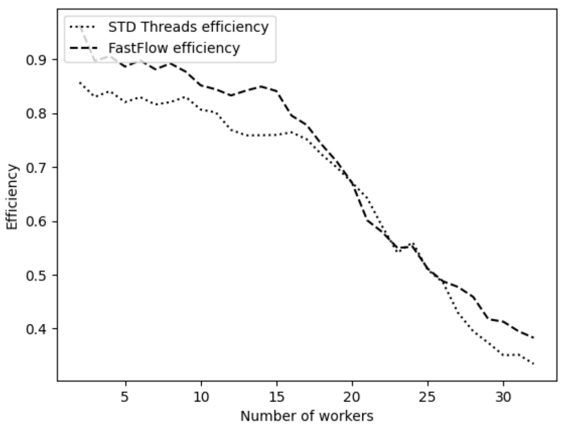
(b) Speedup

Figure 6: Time and Speedup - 5120x5120

Contrary to what has been found in the previous test, with bigger matrices we start to see an inversion of the trend: the speedup grows but the overall algorithm has a speedup further from the optimal than before. Anyway, a matrix of 5120x5120 is still too small for the parallel versions of the algorithm to have higher efficiency with more workers (in the previous test the speedup stopped to grow with 12 workers, now with about 16).



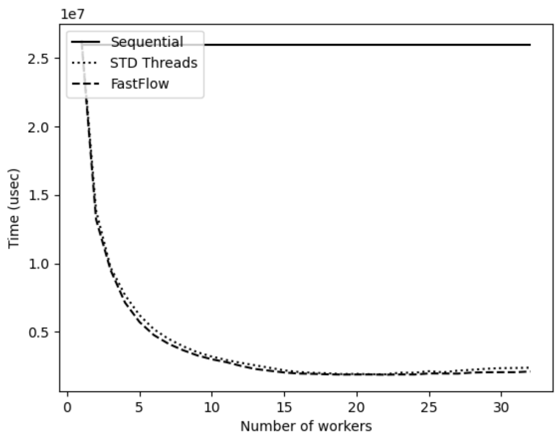
(a) Scalability



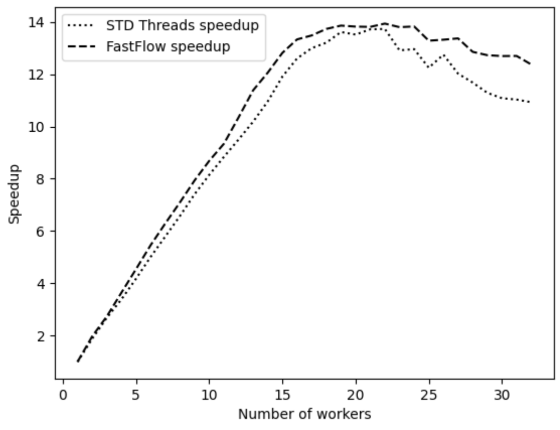
(b) Efficiency

Figure 7: Scalability and Efficiency - 5120x5120

3.2.3 Matrix 10240x10240, workers from 1 to 32

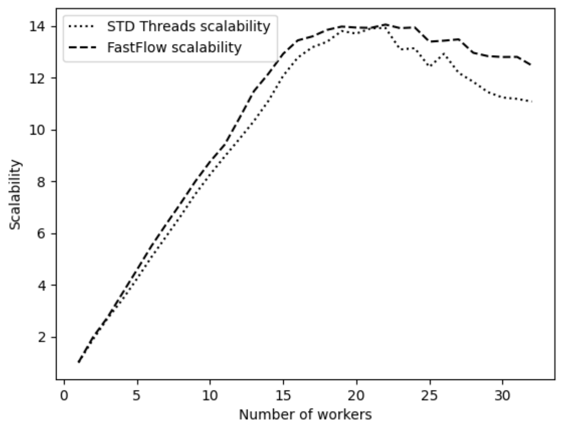


(a) Time

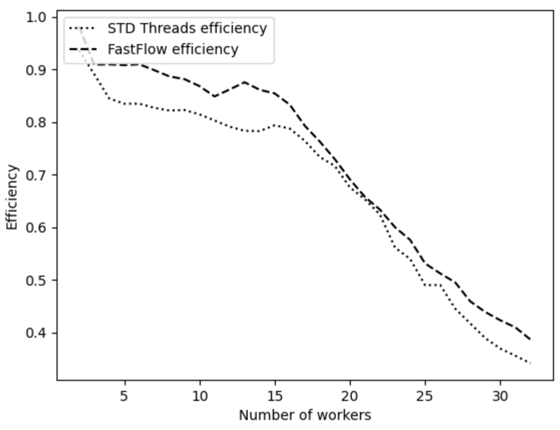


(b) Speedup

Figure 8: Time and Speedup - 10240x10240



(a) Scalability



(b) Efficiency

Figure 9: Scalability and Efficiency - 10240x10240

Similar to the trend of the previous test, with a bigger matrix of 10240x10240 we experience better efficiency with more workers and consequently the speedup continues to grow until about 20 workers.

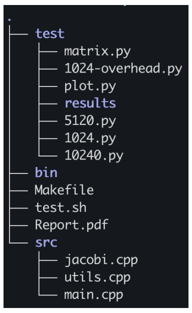
3.3 Final thoughts

From all these tests we see that the speedup asymptote for the 1024 matrix that we found in Section 1.1 is respected and we even get quite close to it. We also found that, in order to have higher speedup and efficiency, the matrix needs to be quite large in size, even if this means also being further from the asymptote.

Furthermore, we also see that in general the low-level barrier performs almost the same as FastFlow, due to the fact that FastFlow is a general-purpose library created with code-reuse in mind. Otherwise, if the low-level parallel code is well-written and optimized, it can usually perform very well and even better than the libraries.

4 Build-run instructions

The repository with the entire codebase and the Python tests is available here and is structured as below:



```
.
├── test
│   ├── matrix.py
│   ├── 1024-overhead.py
│   ├── plot.py
│   ├── results
│   ├── 5120.py
│   ├── 1024.py
│   └── 10240.py
├── bin
├── Makefile
├── test.sh
├── Report.pdf
└── src
    ├── jacobi.cpp
    ├── utils.cpp
    └── main.cpp
```

Figure 10: Directory tree for the codebase

In order to build the code I made a simple `Makefile` with the following directives:

- `make (all)`: build every single version of the algorithm;
- `make prod`: build the code for the "production" environment with default parameters (its executable is simply called `jacobi` and is used to test every version of the code with minimal debugging info);
- `make debug`: build the code with debug purposes, so that it prints the matrices created and their solutions;
- `make overhead`: build the code to find the overhead of the algorithms;
- `make seq`: build the code to run only the sequential version;
- `make std`: build the code to run only the standard threads version;
- `make ff`: build the code to run only the FastFlow version;
- `make par`: build the code to run both parallel versions;
- `make clean`: remove the executables, empty the `bin` directory.

To run the executable use `./bin/<filename> <matrix-size> <seed> <nw>`, with the following parameters[5]:

- `matrix-size`: size of the square matrix;
- `seed`: seed for the randomly generated numbers in the matrix, used to always build the same matrices in the test environment. If set to 0, it creates a random matrix using `srand(time(0))`;
- `nw`: number of workers.

Other than manual testing, to execute the Python tests you just need to compile the code and then launch them singularly with `python3 ./test/<filename>` or execute all tests together with `./test.sh`[6] and they will print the results both to screen and in the `results` directory. If you want to plot them, you can copy the results in the `plot.py` file and run it in your local machine, since the remote one does not have the package `matplotlib` used to create the plots.

[5]The parameters are mandatory; in the sequential version the number of workers will be discarded

[6]The bash script also compiles the code, so there's no need to compile it manually