Master Degree in Computer Science
Curriculum Artificial Intelligence

Parallel and distributed systems: Paradigms and Models
A.Y. 2022/2023

Jacobi method: A comparison of Parallel implementations

Alessandro Capurso (638273)

Abstract

The aim of this project is to implement the Jacobi method in a parallel way. To understand what time-optimization benefits are obtained by parallelization, both sequential and different parallel versions (C++ threads, OpenMP and FastFlow) were implemented. The comparison of the implementations has been performed under three different aspect: speedup, efficiency and scalability. To have a clear view of the performances, also a study on overhead time has been carried out.

1 Introduction

The Jacobi method is an iterative algorithm for determining the solutions of a strictly diagonally dominant system of linear equations. Let $\boldsymbol{Ax} = \boldsymbol{b}$ be a square system of $\boldsymbol{n}$ linear equations, where:

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

At each iteration the following formula is performed:

$$x_{k+1} = \frac{1}{a_{ii}}(b_i - \sum_{i \neq j} a_{ij} x_j^{(k)}) \qquad \forall i = 1...n$$

The full pseudo-code of the algorithm is defined in the Appendix. A sufficient (but not necessary) condition for the method to converge is that the matrix $\boldsymbol{A}$ is strictly or irreducibly diagonally dominant. In this work the generation of the systems of equations satisfy this condition and for a simpler testing phase, all the system have as solution a vector of all one. This approach is taken by the work of Cross et al. [1]

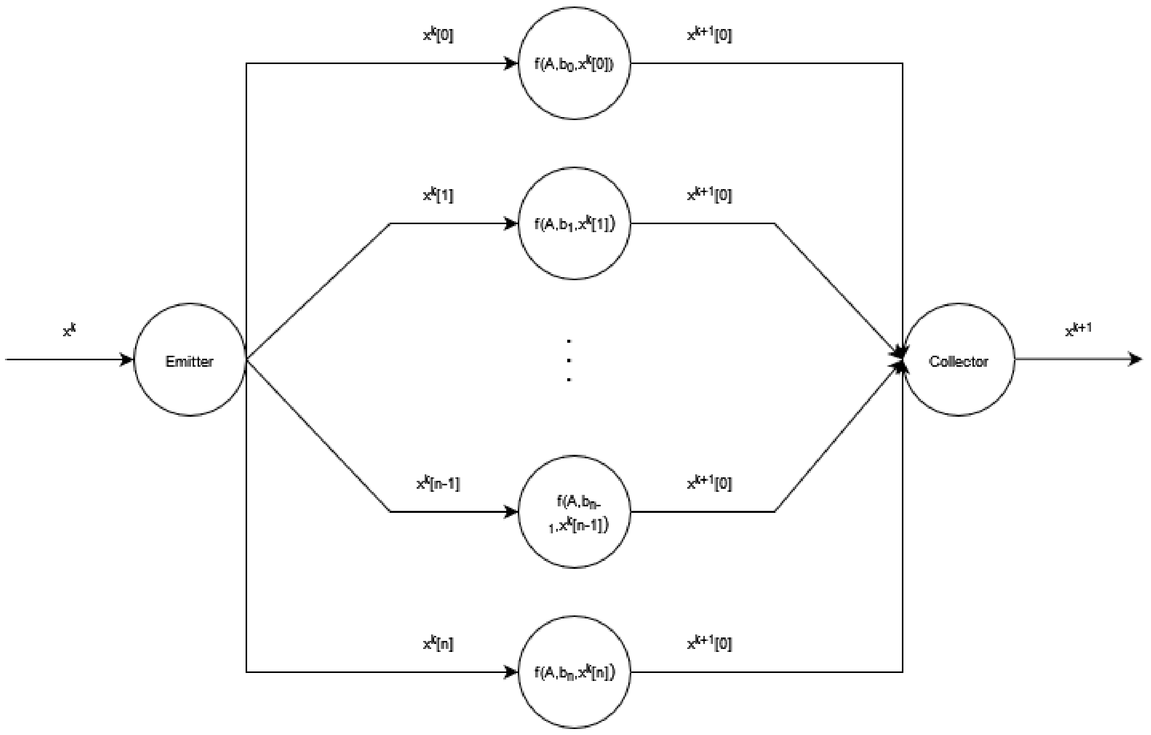
The report contains an analysis of performances of various parallel implementations: a C++ native threads implementation, an OpenMP implementation and a FastFlow implementation.

The report is divided as follow: in **Section 2** an analysis of the algorithm is performed, in **Section 3** the methods used to parallelize the algorithm are descripted, in **Section 4** the results experiments and the results of the work are described and in **Section 5** conclusions are drawn.

2 Analysis

The algorithm (defined in the Appendix) is composed by two principal nested loops. The external one iterates on the rows of the matrix instead the internal one is used to perform a vector product on each row. The external loop can be easily parallelized using a *Map* parallel pattern thanks to the fact that there are no dependencies between iterations.

Unlike the external loop, the internal one has some dependencies, the *sigma* value is computed basing on the previous solution and so the new solution vector could not be updated in parallel. Despite this, it can still be parallelized using a *Reduce* parallel pattern but considering that it performs just a vector product, it is not worth to do. Given this considerations, here is the schema of the parallelization:



Clearly the bottleneck of the solution is the necessity to have a barrier in order to synchronize the update of the solution vector x^{k+1}. Without considering the barrier overhead and the other sources of overhead the expected speedup is

$$sp(nw) = nw \tag{1}$$

In order to make a comparison between different implementations two aspects are needed: $T_{seq}(n)$ that is the sequential time of the computation of a linear system of dimension n and $T_{par}(n, nw)$ that is the the parallel time of the computation of a linear system of dimension n using nw processing resources.

$$T_{seq}(n) = T_{init} + (n\_iter * (n * T_{iter} + T_{stop})) \tag{2}$$

where T_{init} is the time needed to initialize variables, n_iter is the number of iterations, T_{iter} is the time to compute the scalar product of a single vector $x^k[i]$ and T_{stop} is the time needed to verify the convergence at each iteration.

$$T_{par}(n, nw) = T_{init} + T_e + n\_iter * (\frac{n}{nw} * T_{iter} + T_{sync}) + T_c \tag{3}$$

where T_e and T_c are respectively the overhead to split data into chunks and the overhead to merge results coming from the different threads. Notice that T_{stop} has been substituted by T_{sync} because in the synchronization point (barrier) it is necessary to consider not only the time to verify the stop conditions but also the overhead given by the synchronization process.

3 Methods

The implementations proposed in this work are four: the sequential one, a solution with C++ native threads, a solution with OpenMP library and a solution with FastFlow library.

Sequential The sequential implementation is the same as the one in the Appendix.

C++ Native threads The map is implemented dividing data into chunks of size n/nw. Then these chunks are distributed in the nw threads. At each step the threads are synchronized using a barrier. In the synchronization point the solution vector is updated and the stopping criteria is verified.

OpenMP The OpenMP implementation consists in the addition of the directive #pragma omp parallel for num_threads(nw) over the outer for loop to the code of the sequential implementation.

FastFlow The FastFlow version is made by starting from the sequential implementation, the outer cycle is replaced by a parallel for, that is instantiated once at the beginning of the computation. The framework handle all the synchronization issues.

Two different types of stopping criteria were implemented. The first verifies that a maximum number of iterations is not exceeded, and the second terminates the process when the solution found is equal to the actual solution. These criteria have been used in AND as condition for all the implementations.
In order to evaluate the performances of the four implementations, three different metrics have been taken into account: **speedup**, **efficiency** and **scalability**.

4 Experiments and Results

All the experiments have been performed on the Intel(R) Xeon(R) Gold 5120 CPU @ 2.20GHz with 32 cores. The matrices used in this phase have been generated using the method described in [1] which generates a system of linear equations with a strictly diagonally dominant matrix. The solution for all the generated system is a vector of ones and helps to simplify the testing phase (see *utils.hpp*). The equality between the solution found and the expected solution is computed using a threshold equals to $1e-8$.
At compilation time the files have been compiled with the -O3 flag in order to optimize where possible the code such that the execution time could be the best possible.
The performances of the solutions were explored by varying different values of `number of threads` (from 1 to 32) and `size of matrix A` (2^7,2^8,2^9,2^{10},2^{11},2^{12},2^{13}). The experiments have been executed 5 times and the average time has been taken into account to improve the

reliability of the results. Due to the time constraint, for all the matrices with a dimension greater or equal than 2^{10} have been tested with 1000 as the maximum number of iterations. The metrics represented in the graphs have been computed w.r.t. the time taken by the algorithm to perform an iteration.

4.1 Overhead estimation

Before to test the different implementations, an overhead analysis has been performed. Being a simple problem there are only two principal source of overhead. The first overhead taken in account is the one for the thread initialization (T_{init}) and the second one is the time of thread synchronization at each iteration (T_{sync}).
Exploiting the Amdhal law an upper bound for the speedup for each matrix dimension has been found. Amdhal say that, fixing a problem dimension, the speedup that the application can achieve is

$$sp(n) = \frac{T_s}{fT_s + (1-f)\frac{T_s}{n}} \tag{4}$$

where T_s is the unit time needed by a program to complete the task in sequential, f is the serial fraction, $(1-f)$ the parallel fraction and n is the number of processing resources. Assuming to have infinite processing resources the speedup can be written as

$$\lim_{n \to +\infty} \frac{T_s}{fT_s} = \frac{1}{f} \tag{5}$$

The $(1-f)$ term is excluded because with a growing n it goes to 0.

Considering also the parallel overhead, the formula become:

$$sp(n) = \frac{T_s}{fT_s + (1-f)(\frac{T_s}{n} + T_{ov}(n))} = ... = \frac{1}{f + (1-f)(\frac{1}{n} + \frac{T_{ov}(n)}{T_s})} \tag{6}$$

The Equation 6 is the formula applied to define the upper bounds in the speedup plots. The overhead times used are the ones listed above. The serial fraction considered is composed by the update of the solution vector, the check of the stopping criteria and the increment of the iteration counter. In order to understand the impact of the different overheads two different bound have been computed, one considering only the thread initialization overhead and one considering both the thread initialization and the barrier time. The barrier time has been taken considering the minimum time passed in the barrier.

4.2 Metrics evaluation

Speedup

In Figure 1 it is possible to see that the upper bound defined as described in the Paragraph 4.1 is a good approximation of the implementation performances. Increasing the problem size the upper bounds become close to the ideal speedup. For *ub. thr* is clearly caused by the fact that the barrier time is not considered, instead for the *ub. over.* is caused by the fact that the barrier time considered is an optimistic approximation. More generally, the results achieved are quite satisfactory reaching a maximum speedup of **16.68** with n =

8192 and nw = 32 with the C++ Thread implementation.

Efficiency

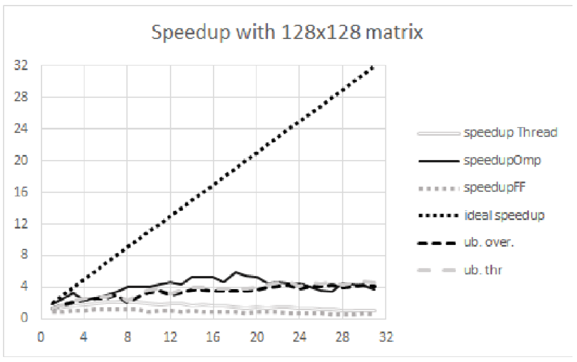
For what concern efficiency, the results obtained in the experiments for the small matrices (2(a), 2(b), 2(c)) decrease as the processing resources increases. This happens also for the others matrices dimensions but at a certain point it is possible to notice that increasing the number of processing resources a sort of asymptote is reached. OpenMP performs better than other implementations in terms of efficiency with the small matrices. For bigger dimensions, OpenMP and C++ Threads perform in a similar way.

Scalability

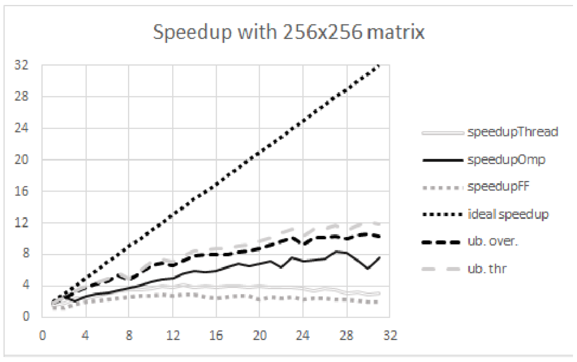
Performances scales well increasing the number of processing resources. For the small matrices (3(a), 3(b), 3(c)), also in term of scalability, OpenMP performs better than the other implementations.

5 Conclusions

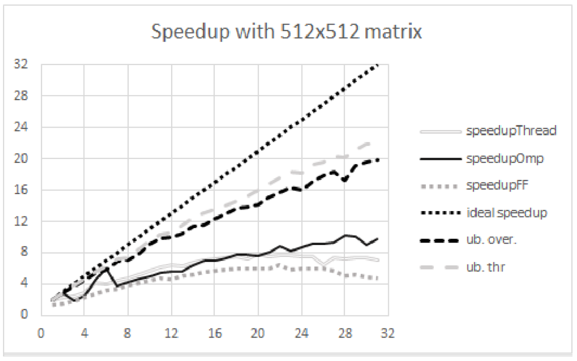
In this project, an analysis was presented for parallelization of Jacobi's method algorithm on strictly diagonally dominant matrices with different types of implementations (C++ Threads, OpenMP and FastFlow). For system of equations greater or equals to 1024, the various implementations perform in a similar way, instead, with small problems the most performing implementation is OpenMP. Considering very small matrices the parallelization is not worth because the overheads can make parallel execution slower than the sequential one. It was intresting to try to predict an upper bound for the speedup, analysing the sequential code and the overhead involved in the parallelization setup.



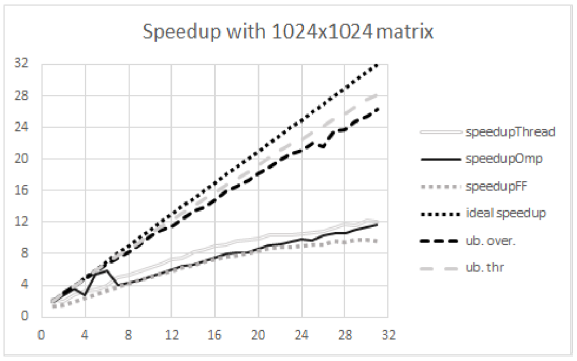
(a) 128x128



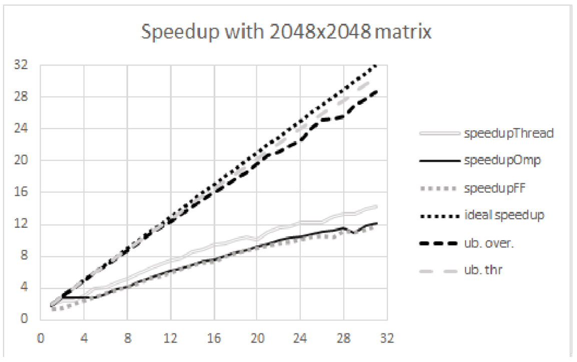
(b) 256x256



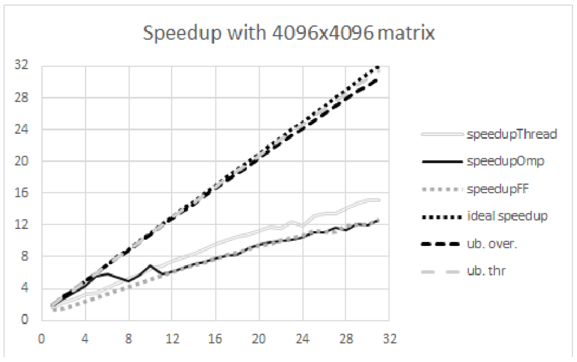
(c) 512x512



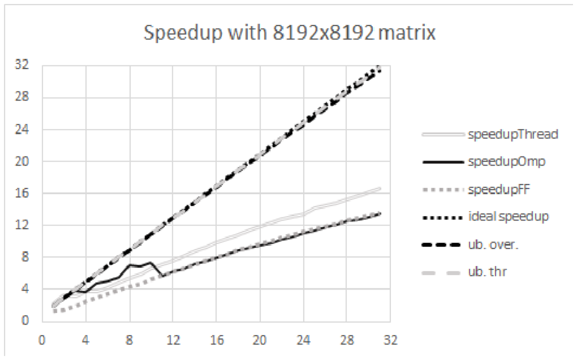
(d) 1024x1024



(e) 2048x2048

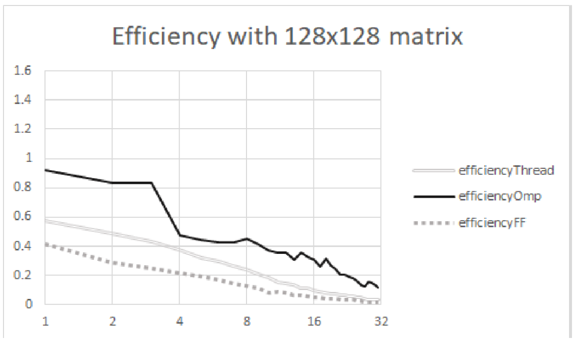


(f) 4096x4096

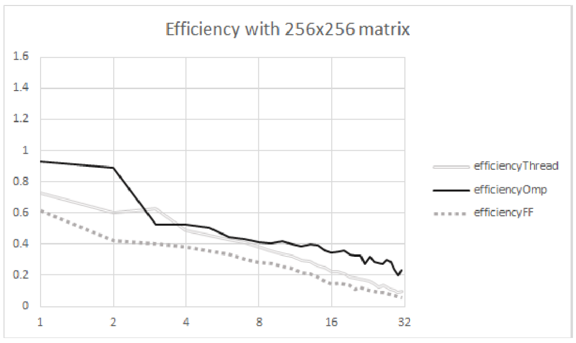


(g) 8192x8192

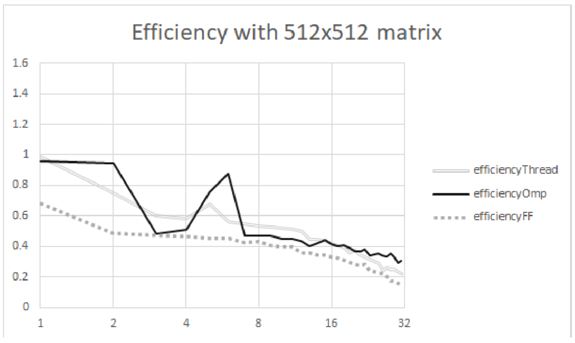
Figure 1: Speedup plots of the different parallel implementations



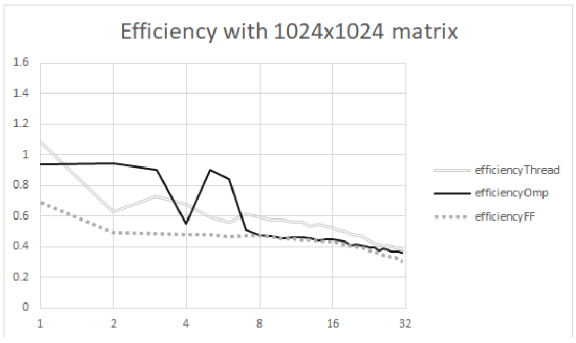
(a) 128x128



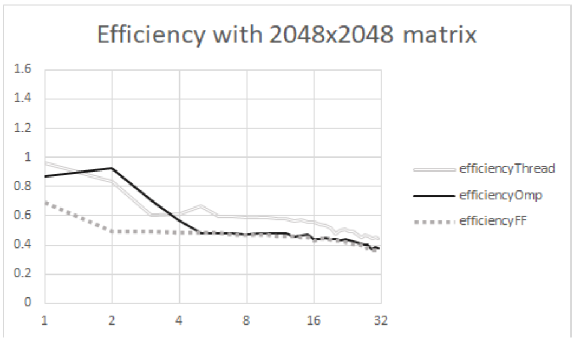
(b) 256x256



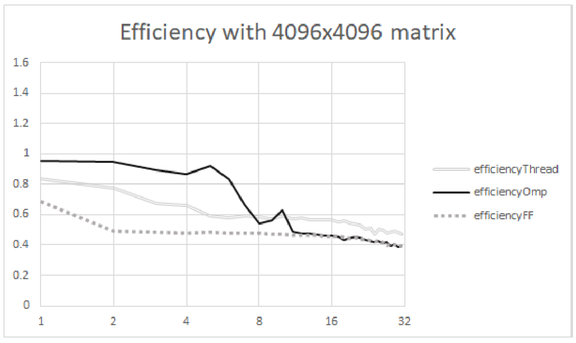
(c) 512x512



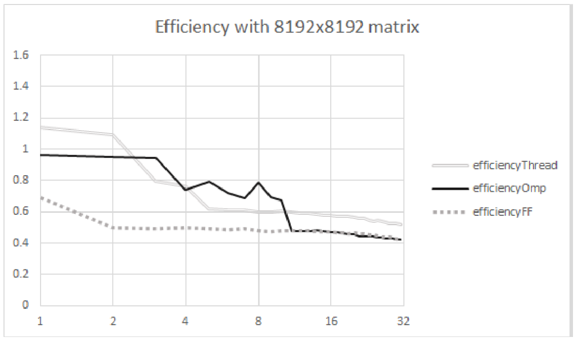
(d) 1024x1024



(e) 2048x2048

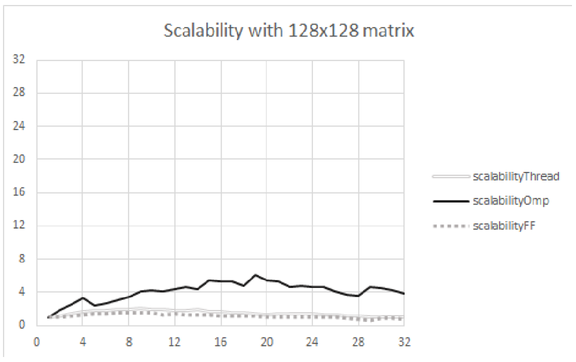


(f) 4096x4096

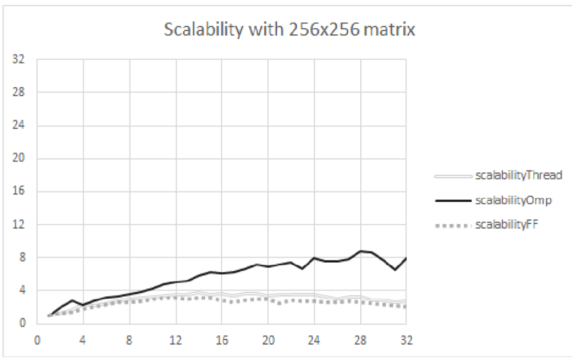


(g) 8192x8192

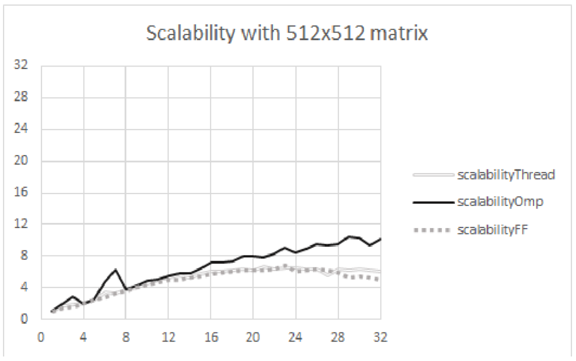
Figure 2: Efficiency plots of the different parallel implementations



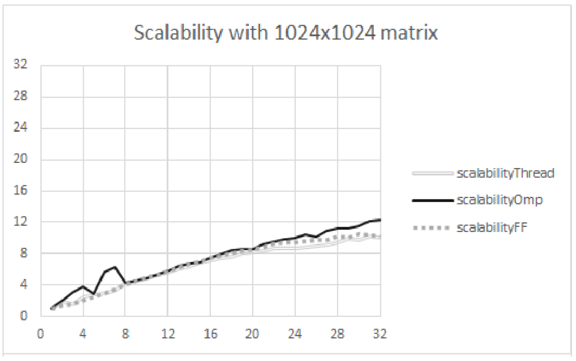
(a) 128x128



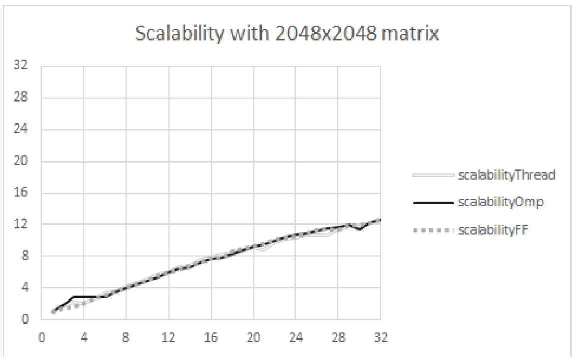
(b) 256x256



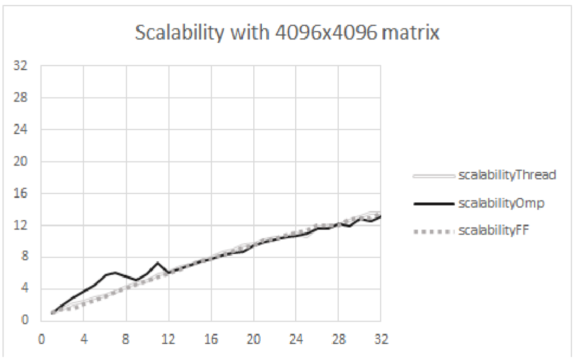
(c) 512x512



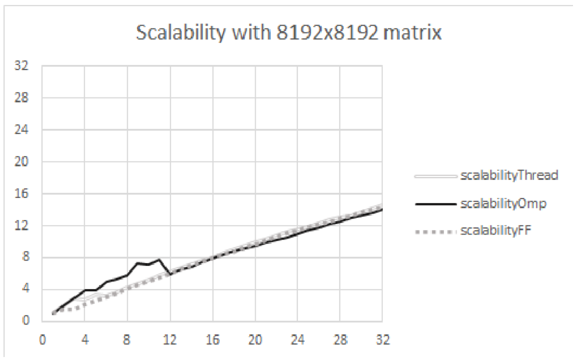
(d) 1024x1024



(e) 2048x2048



(f) 4096x4096



(g) 8192x8192

Figure 3: Scalability plots of the different parallel implementations

References

[1] Andreea-Ingrid Cross, Liucheng Guo, Wayne Luk, and Mark Salmon. Cjs: Custom jacobi solver. In *Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies*, pages 1–6, 2018.

A Sequential Code

```
//A is a matrix NxN
//b, x, x_new are vectors of size N
//StoppingCriteria is a function that choose if the method must terminate.
//tol is the threshold for the stopping criteria
//nIter is the max number of iterations that should be performed (in case
    that the methods converge very slowly)

while (!stoppingCriteria(x, tol) && (iter < nIter)) {

    for (int i = 0; i < dim; i++) {
        double sigma = 0;
        for (int j = 0; j < dim; j++) {
            if (i != j) {
                sigma = sigma + A[i][j] * x[j];
            }
        }
        x_new[i] = (1.0 / A[i][i]) * (b[i] - sigma);
    }

    x = x_new;
    iter++;
}

return res;
```

B Building and running code

B.1 Build

To build the project use the following commands

```
mkdir build
cd build
cmake ..
make
```

B.2 Run

The program offers two different interfaces. The first one is formatted as

```
./JacobiParallel 2 1024 10000 5 1 1 1 1
```

Where 2 is the number of workers to use, 1024 is the matrix dimension, 10000 is the maximum number of iterations and the ones are boolean flag to execute respectively the sequential code, C++ threads, OpenMP and FastFlow. It is useful to run experiment on a single configuration with one or more methods.
The second one is formatted as

```
./JacobiParallel 0
```

Where 0 is an integer value. If it is 0, it runs an exhaustive test with different configurations over all the implementations. If 1, it performs an exhaustive computation of the overhead time for different configurations of the C++ threads implementation. If 2, it performs a serial fraction time evaluation. Otherwise it performs a thread initialization time evaluation.

In addition to the parameters, it is necessary to indicate which type of times you want to record. This can be done by commenting and/or uncommenting the variables `PERFORMANCE`, `OVERHEAD` and `PRINT` in the *include/constants.hpp* file. When `PERFORMANCE` is declared the execution times will be recorded, when `OVERHEAD` is declared the overhead times of the parallel C++ threads implementation will be recorded. Finally when `PRINT` is declared the times recorded will be shown in the output.

All the requested times will be recorded in the project *results* folder (it will be automatically generated).