Parallel and distributed systems: paradigms and models

The Jacobi method

Silvio Martinico

July 03, 2022

Abstract

In this report we deal with the Jacobi method, an iterative algorithm for determining the solution of a square and strictly diagonally dominant system of linear equations.

The focus of this work is a parallel implementation of the Jacobi method. We will see the classical sequential implementation of the algorithm and then we will compare it with two parallel versions: the first one relies explicitly on the classical thread approach, while the second one uses `FastFlow`, a `C++` library for high-performance parallel patterns.

Contents

1 Preliminaries

Let's briefly describe under what assumptions the Jacobi method works and introduce the basic notions and notations.

Let $Ax = b$ be a linear system, where A is a square matrix and b is the vector of the known terms. Let D be the diagonal matrix whose diagonal coincides with that of A, then we can write $A = D - R$ for an opportune matrix R. If the diagonal elements of A are different from 0, then we can define:

$$J := D^{-1}R\,.$$

So, we can write the generic iteration of the Jacobi method as follows:

$$x^{(k+1)} = Jx^{(k)} + D^{-1}b\ ,$$

which, componentwise, becomes:

$$x_i^{(k+1)} \;=\; \frac{1}{a_{ii}}\left(b_i \;-\; \sum_{\substack{j=1\\ j\neq i}}^{n} a_{ij}x_j^{(k)}\right) \tag{1}$$

Now, if A is strictly diagonally dominant (i.e. the sum of the modules of the non-diagonal elements in each row is smaller than the diagonal element of that row), then $\rho(J) < 1$ (where $\rho(J)$ is the spectral radius of J) and, under these assumptions, the following facts hold:

- the system admits a unique solution (because strictly diagonally dominant $\implies$ invertible);
- the sequence $\{x^{(k)}\}_{k=0,1,\ldots}$ converges to the solution.

1.1 Data generation

The matrix A and the vector b were generated randomly, with a uniform distribution in the interval $[-10, 10)$ and precision up to three decimal digits.
Let now d_i be the sum of the non-diagonal elements of the i-th row. In order to make A strictly diagonally dominant, the diagonal elements are not generated randomly, but they are taken as $\frac{11}{10}d_i + 1$.
Since the convergence of the method does not depend on the initial vector $x^{(0)}$, it is also randomly generated.

So, sure we are under the right assumptions, we can proceed with the implementations.

2 Sequential implementation

Let's see how to implement the sequential version of the Jacobi method.

The idea is to use three nested `for` loops, the first one iterates over the desired number of iterations of the method, while the other two iterate over the dimension of the space (the outermost one fixes one component of the vector $x^{(k)}$, the innermost one is the index j in the summation (1)).

One substantial improvement in the algorithm was done at the 10-th row in the code in (1): it is way cheaper to don't check if $i \neq j$ at each iteration and just add $A_{ii}x_i^{(k)}$ after the `for` loop. An alternative way of avoiding this check would be that of splitting the loop in two parts, the first one which goes from 0 to $i - 1$ and the second one which goes from $i + 1$ to $n - 1$.

In the pseudocode (1), at the 18-th row, a control on the 1-norm (renormalized) of the difference of $x^{(k+1)}$ and $x^{(k)}$ is made in order to avoid a waste of time. The tolerance *tol* is fixed to 10^{-12}. During the experiments we observed that, in some cases, the tolerance is too small and it is never reached, leading us to an high number of useless iterations. If, instead, we increase the tolerance too much, the opposite problem occurs, i.e. the execution stops too quickly. In order to avoid these cases, a further stopping criterion was added: if $\delta_k := \frac{\sum |x_i^{(k+1)} - x_i^{(k)}|}{n} < \delta_{k+1}$ for a certain number of times (6 in our case), i.e. if the convergence is not monotone anymore, the execution is stopped. This is useful because, often, the delta begins to go up and down without decreasing anymore, so the algorithm does not stop due to the tolerance, but anyways these iterations are useless.

Algorithm 1 Sequential Jacobi method

1: Input:
2: **A** matrix of the linear system
3: **b** known term of the system
4: **n** dimension of the space
5: **N** number of iterations of Jacobi method
6: **tol** is the tolerance
7:
8: **for** $k = 0, 1, \ldots, N-1$ **do**
9: **for** $i = 0, 1, \ldots, n-1$ **do**
10: **for** $j = 0, 1, \ldots, n-1, \ \ j \neq i$ **do**
11: $x_i^{(k+1)} -= A_{ij} x_j^{(k)}$
12: **end for**
13: $x_i^{(k+1)} += b_i$
14: $x_i^{(k+1)} /= A_{ii}$
15: $\delta^{(k+1)} = \frac{\sum |x_i^{(k+1)} - x_i^{(k)}|}{n}$
16: **if** $\delta^{(k+1)} > \delta^{(k)}$ **then** $cont++$
17: **end if**
18: **if** $\delta^{(k+1)} < tol$ **then return** $x^{(k+1)}$
19: **elseif** $cont > 6$ **then return** $x^{(k+1)}$
20: **end if**
21: **end for**
22: **end for**
23: **return** $x^{(N)}$

2.1 Parallelizability analysis

Given the pseudocode above, we are now going to analyze what operations we can parallelize in order to improve the performances of the algorithm.

The algorithm is basically made up of three loops:

- the outermost loop is not parallelizzable, because each iteration of the method strictly depends on the previous one, so we have to fully compute $x^{(k)}$ in order to compute $x^{(k+1)}$;
- the middle loop fixes the component of the vector $x^{(k+1)}$ that we have to compute. We can act on this loop by distributing the components of the vector among various workers: each worker can compute a part of $x^{(k+1)}$ independently of the other workers;

- the innermost loop is also parallelizzable: each worker could subtract a fraction of the total number of the $A_{ij}x_j^{(k)}$s, then we have to sum b and divide each $x_i^{(k+1)}$ by A_{ii}. However, this approach clearly gives us a lower level of parallelization since it leads us to some sequential operations.

So, after this analysis, we come to the conclusion that the best way to parallelize the algorithm is to act on the intermediate loop. Given this parallelization, another thing we can improve is the computation at the 15-th row: since we are giving each worker a range of components, each of them can compute a partial summation and then, we have to do sequentially just nw sums and one quotient in order to get the total delta (nw indicates the number of workers).

3 Parallel implementation

The parallel pattern we are going to use is a *Data parallel pattern*: each worker performs the same task but with different data, and this is done independently of the other workers. In particular, the i-th worker takes as input the vector $x^{(k)}$ and a range of rows of the matrix A and produces as output the corresponding components of $x^{(k+1)}$. More specifically, $\forall\ i = 1, \ldots, nw$ the i-th worker takes as input the rows from $(i-1)\frac{n}{nw}$ to $i\frac{n}{nw} - 1$.

The partitioning of the vector $x^{(k+1)}$ is immediate if n is a multiple of nw. If this is not the case, we have to take a good partition of this vector.
Let $n = q \cdot nw + r$ be the Euclidean division of n by nw. Each worker computes a range of q components of $x^{(k+1)}$ and then there are r components left. In order to make the work as balanced as possible, we distribute this r components one to each of the first r workers (from the Euclidean division it follows that $r < nw$). Instead of doing this kind of "circular" division, we give $q + 1$ (consecutive) components to the first r workers and q to the other $nw - r$. In the code, this work is done by the function `factoriz()`, in the file `libr.h`.

Once we know how to divide the work among the various workers, we can proceed with the actual parallel implementation. We did two kinds of parallel implementations: the first one uses explicitly the threads, while the second one exploits the library `FastFlow`.

3.1 Standard thread approach

The parallel version of the Jacobi method is implemented in the function `jacobi_par`.
The starting point is a loop that iterates over the number of workers and creates each time a new thread, giving to it, as argument, the function `task`, which contains what the worker is going to do.

We report the pseudocode for the function `task` in (2). We can notice that the general structure is similar to that of the sequential version but, the intermediate loop takes only a portion of the total set of indexes ($fact[i], \ldots, fact[i+1] - 1$ instead of $0, \ldots, n-1$). The other big difference is that, after the two innermost loops, inside the outermost loop, we call the method `arrive_and_wait` of the class `Barrier`. We need the `Barrier` in order to synchronize the various threads, because, before starting a new iteration of the method, every thread must have finished the previous iteration (we need the whole $x^{(k)}$ in order to compute $x^{(k+1)}$).

In (3) we can see the pseudocode of the function of the `Barrier`. In this function we decrease the iterations counter, compute the total delta and check for the stopping conditions.

Algorithm 2 task (body function of a thread)

Input:
  i number of the current thread
  n dimension of the space
  N number of iterations of Jacobi method
  fact vector which tells to each thread how many components it have to compute

while $N > 0$ **do**
  for $s = fact[i], \ldots, fact[i+1] - 1$ **do**
    for $j = 0, 1, \ldots, n-1, \;\; j \neq i$ **do**
      $x_s^{(k+1)} -= A_{sj} x_j^{(k)}$
    end for
    $x_i^{(k+1)} += b_s$
    $x_i^{(k+1)} /= A_{ss}$
    $\delta_i^{(k+1)} += |x_s^{(k+1)} - x_s^{(k)}|$
  end for
  Barr.arrive_and_wait();
end while
return $x^{(N)}$

Algorithm 3 Barrier object function

Input:
  i number of the current thread
  n dimension of the space
  nw number of threads
  N number of iterations of Jacobi method
  tol is the tolerance
  fact vector which tells to each thread how many components it have to compute

$N--$
$\delta^{(k+1)} = \frac{1}{n} \sum \delta_i^{(k+1)}$

if $\delta^{(k+1)} > \delta^{(k)}$ **then** $cont++$
end if
if $\delta^{(k+1)} < tol$ **then** $N = 0$
elseif $cont > 6$ **then return** $N = 0$
end if
if $\delta < tol$ **then** $N = 0$
end if
return

3.2 FastFlow

For the parallel version, besides the standard threads, we also used the library FastFlow.

This version of the Jacobi method is similar to the sequential one, but the intermediate loop is a `parallel_for` instead of a classical `for` loop. A `parallel_for` is a method of the class `ParallelFor` from the `FastFlow` library. This method allows us to parallelize a loop, specifying the starting point, the arrival point and a lambda function which is the body of the loop; inside this function we put the innermost loop of the method.

4 Tests

Finally, we can see how the code works.

We want to analyze both the behaviour of the algorithm with various number of threads (in order to understand if it works) and some technical aspects. In particular, in the following tests, as the number of threads (nw) increases, we are going to compute the following parameters:

- T_{seq}: the sequential service time in microseconds (μs);
- $T_{par}(nw)$: the parallel service time with nw threads, in microseconds (μs);
- $Speedup(nw)$;
- $Eff(nw)$: the efficiency
- $Scal(nw)$: the scalability.

For each number of threads, we ran the code a certain number of times and then we removed the outliers and took the average for each parameter.

Each test had run with an input of 100 iterations, but each of them stopped earlier due to the stop conditions. The number of iterations effectively done by the various executions are all between 20 and 30.

We tested matrices of two different dimensions: 10000×10000, and 1000×1000. In this way we can see how the algorithm scales with the dimension of the space.

In the pages below, we report two tables (threads and FastFlow) for the higher space dimension, together with three plots (speedup, efficiency and scalability) and two tables for the smaller space dimension with the same three kind of plots.

nw	T_{seq} (μs)	$T_{par}(nw)$ (μs)	$T_{par}(1)$ (μs)	$Speedup(nw)$	$Eff(nw)$	$Scal(nw)$
2	3229543	1630957	3235306	1,9806	0,9902	1,9512
3	3579783	1205252	3583274	2,9701	0,9768	2,9734
4	3652002	940289	3656361	3,8862	0,9715	3,8921
5	3943593	810826	3942242	4,8638	0,9727	4,8622
6	3432191	591419	3395883	5,8025	0,9670	5,7446
7	3926821	596557	3930506	6,5846	0,9406	6,5909
8	3936376	511385	3967568	7,6992	0,9623	7,7613
9	3632287	414272	3627853	8,7694	0,9743	8,7645
10	3827689	412762	3791465	9,3061	0,9306	9,2192
11	3658565	357149	3636761	10,2226	0,9293	10,1645
12	3502327	309003	3475729	11,3229	0,9435	11,2404
13	4147571	348309	4156958	11,9116	0,9161	11,9384
14	3780188	294956	3672572	12,8297	0,9137	12,4722
15	3580807	264602	3583933	13,5543	0,9035	13,5677
16	3708272	257809	3733701	14,3819	0,8988	14,4854
17	3795125	243809	3825044	15,5597	0,9152	15,6827
18	3872447	242698	3878902	15,9466	0,8858	15,9752
19	3542000	209034	3571967	16,9453	0,8918	17,0922
20	3921932	240092	3934694	16.3351	0,8167	16,3883
21	3479681	197729	3470169	17.5982	0,8380	17,5501
22	3966012	218122	3944356	18,1825	0,8264	18,0833
23	3468084	190705	3463320	18,1992	0,7912	18,1743
24	3599952	194116	3618907	18,5454	0,7727	18,6431
25	3892051	200320	3883022	19,4266	0,7771	19,3820
26	3145224	189051	3147552	16,6369	0,6398	16,6492
27	3929780	222691	3923682	17,6468	0,6535	17,6194
28	3912841	229400	3929194	17,0568	0,6091	17,1281
29	3287751	181568	3290059	18,1075	0,6244	18,1203
30	3556058	203777	3556574	17,4324	0,5810	17,4363
31	3366070	194765	3368129	17,27785	0,5747	17,2859
32	3854727	299704	3880793	12,8607	0,4018	12,9824

Table 1: Standard threads tests with $n = 10000$ and $N = 100$

nw	T_{seq} (μs)	$T_{par}(nw)$ (μs)	$T_{par}(1)$ (μs)	$Speedup(nw)$	$Eff(nw)$	$Scal(nw)$
2	3229543	1660377	3245748	1,9448	0,9724	1,9672
3	3579783	1231066	3562753	2,9076	0,9692	2,8936
4	3652002	946340	3641310	3,8528	0,9631	3,8428
5	3943593	811130	3937225	4,8622	0,9724	4,8542
6	3432191	601377	3395999	5,7100	0,9516	5,6546
7	3926821	579729	3910293	6,7762	0,9680	6,7473
8	3936376	522815	3941852	7,5912	0,9417	7,5454
9	3632287	426449	3653013	8,5126	0,9458	8,5606
10	3827689	428261	3862574	8,9646	0,8964	9,0270
11	3658565	362028	3627651	10,1232	0,9202	10,0479
12	3502327	307690	3479049	11,3774	0,9481	11,3049
13	4147571	349744	4219786	11,8622	0,9124	12,0717
14	3780188	294021	3687459	12,8531	0,9180	12,5502
15	3580807	264980	3579304	13,5291	0,9019	13,5249
16	3708272	257809	3733701	14,3819	0,8988	14,4854
17	3795125	244922	3804620	15,5017	0,9118	15,5409
18	3872447	241220	3857813	16,0423	0,8912	15,9796
19	3542000	214102	3541912	16,5690	0,8720	16,5678
20	3921932	237947	3940349	16,4824	0,8241	16,5598
21	3479681	206783	3499807	16,8277	0,8013	16,9250
22	3966012	224108	3948608	17,6969	0,8044	17,6192
23	3468084	187938	3477244	18,4445	0,8019	18,4919
24	3599952	190275	3606426	18,9197	0,7883	18,9538
25	3892051	202136	3855630	19,2655	0,7706	19,0879
26	3145224	181656	3182591	17,3142	0,6659	17,5199
27	3929780	225472	3913192	17,4291	0,6455	17,3556
28	3912841	219280	3931008	17,8440	0,6372	17,9269
29	3287751	172406	3349577	19,0698	0,6575	19,4284
30	3556058	183541	3582026	19,4300	0,6451	19,4974
31	3366070	184100	3365470	18,2839	0,5897	18,2810
32	3854727	297474	3925826	12,9578	0,4049	13,1967

Table 2: FastFlow tests with $n = 10000$ and $N = 100$

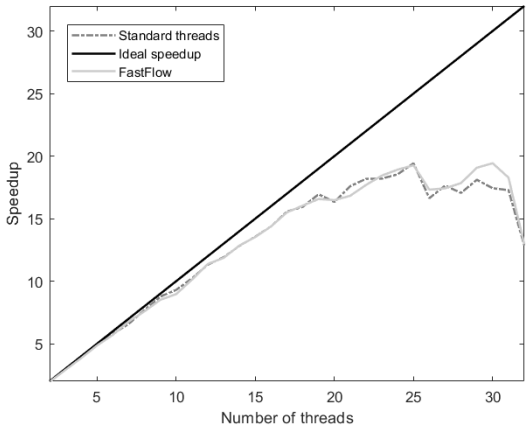

Figure 1: Speedup

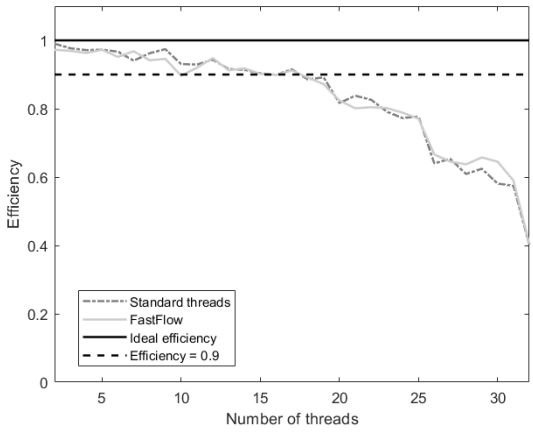

Figure 2: Efficiency

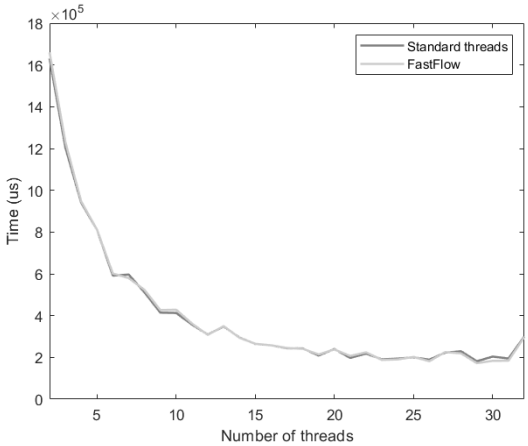

Figure 3: Service time

nw	T_{seq} (μs)	$T_{par}(nw)$ (μs)	$T_{par}(1)$ (μs)	$Speedup(nw)$	$Eff(nw)$	$Scal(nw)$
2	39859	21347	39978	1,8656	0,9328	1,8715
3	39144	14830	38965	2,6372	0,87905	2,6221
4	34262	9772	33825	3,4994	0,8753	3,4601
5	38468	9424	37800	4,0819	0,8163	4,0110
6	43893	8648	40601	5,0755	0,8459	4,6948
7	38940	7295	39262	5,3379	0,7625	5,3820
8	38560	6703	38377	5,8816	0,7185	5,7336
9	34925	5558	34415	6,2737	0,6970	6,1886
10	41786	6069	37935	6,8851	0,6885	6,2506
11	39068	5918	40247	6,6001	0,5999	6,8003
12	40641	5742	39324	7,0778	0,5898	6,8484
14	43762	5801	40619	7,5438	0,5388	7,0020
16	35278	5091	35489	6,9294	0,4330	6,9709
18	36312	5212	37239	6,9705	0,3872	7,1507
20	37813	5485	38258	6,8938	0,3446	6,9750
22	37501	5783	37644	6,4847	0,2947	6,5094
24	32385	5142	32442	6,3097	0,2628	6,3168
26	36540	6269	36581	5,8286	0,2241	5,8352
28	39549	7018	39625	5,6353	0,2012	5,6462
30	43064	8051	42571	5,3385	0,1779	5,2812
32	40012	9225	43586	4,33734	0,1355	4,7247

Table 3: Standard threads tests with $n = 1000$ and $N = 100$

nw	T_{seq} (μs)	$T_{par}(nw)$ (μs)	$T_{par}(1)$ (μs)	$Speedup(nw)$	$Eff(nw)$	$Scal(nw)$
2	39859	21474	40030	1,8187	0,9301	1,8671
3	39144	14816	37928	2,6353	0,8817	2,56695
4	34262	9888	34812	3,4623	0,8655	3,5151
5	38468	8895	37769	4,3246	0,8649	4,2460
6	43893	8481	41064	5,1754	0,8625	4,8418
7	38940	7181	39139	5,4226	0,7746	5,4503
8	38560	6427	38821	5,9950	0,7493	6,0362
9	34925	5475	34148	6,3931	0,7103	6,2474
10	41786	5643	39122	7,4049	0,7404	6,9328
11	39068	5807	39184	6,7257	0,6114	6,7455
12	40641	5676	39152	7,1601	0,5966	6,8978
14	43762	5588	40560	7,8314	0,5593	7,2584
16	35278	4442	35737	7,9717	0,4981	8,0699
18	36312	4993	37650	7,2856	0,4047	7,5694
20	37813	5295	37922	7,1412	0,3570	7,1618
22	37501	6036	39461	6,2128	0,2824	6,5376
24	32385	4515	31590	7,2471	0,3019	7,0747
26	36540	5583	36536	6,5448	0,2517	6,5441
28	39549	5899	39361	6,7043	0,2394	6,6724
30	43064	7794	42527	6,0157	0,2004	5,9829
32	40012	5751	37862	6,9574	0,2174	6,5835

Table 4: FastFlow tests with $n = 1000$ and $N = 100$

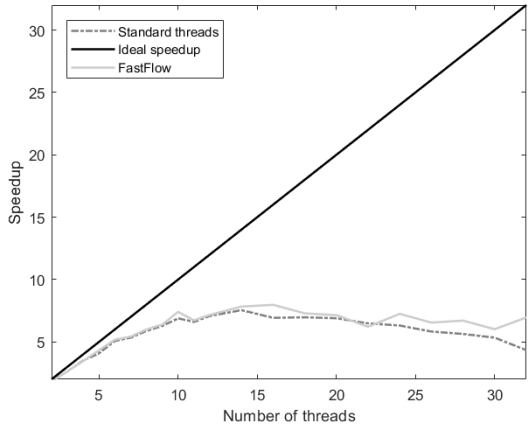

Figure 4: Speedup

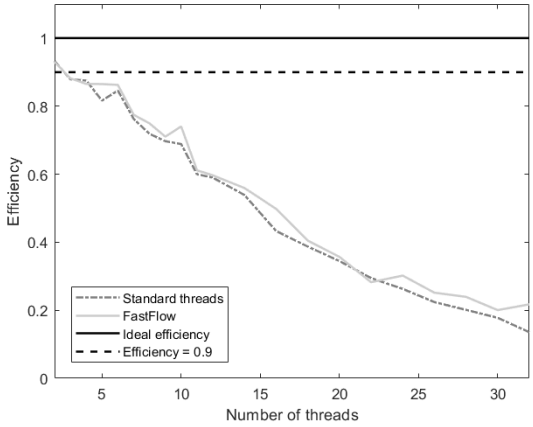

Figure 5: Efficiency

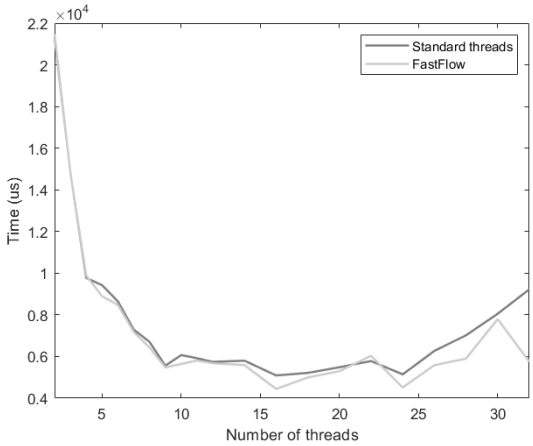

Figure 6: Service time

4.1 Considerations on the results

Regarding the bigger matrix, in Table 1 we reported the results of the tests with the standard thread approach, while, in Table 2, we can see the results of the FastFlow tests. In the plots 1, 2 and 3 we can see graphically the comparison between threads and FastFlow, both in terms of speedup, efficiency and service time.
As we can see, the two parallel versions have very similar behavior in every respect. Regarding the speedup, we see that, as the number of threads grows, it deviates from the line of the ideal speedup (as expected from the theory) and, starting from $nw = 25$, it begins to decrease. We can notice that the efficiency decreases (as expected) and, from 2 to 17 threads, it is greater then $0,9$ (except that with 16 threads, where it is slightly under $0,9$), so, we can say that 17 threads is the number of thread which gives us the best overall performances. On the other hand, the greatest speedup was reached for $nw = 25$ by the standard threads and for $nw = 30$ by FastFlow. We observe that, the FastFlow works better with an higher number of threds, while with less threads the distance between it and standard threads is negligible.

Finally, passing to service time, we see that the difference between threads and FastFlow is very small.

For the smaller matrix, we reported the results of the tests in Table (3) for standard threads, and in Table (4) for FastFlow. The plots are in the figures 4, 5 and 6. With a smaller matrix, we can have a better visualization of the behaviour of the speedup, which, for $nw > 16$, starts to decrease. Regarding the efficiency, it is greater than $0,9$ only for $nw = 2$; this is due to the small dimension of the matrix. In fact, the smaller dimension of the space implies a smaller service time, which results in a greater weight of the overhead since the overhead does not depend on the space's dimension but only on the number of threads we have to create, synchronize and join.

The maximum speedup is reached for $nw = 14$ in the case of standard threads and for $nw = 16$ by the FastFlow.

The differences between the two approaches are more clear with these smaller matrices and we can see how FastFlow works better overall.

In both cases the scalability is very similar to the speedup, so we decided to not plot it.

Comparing the tests on big matrices with those on small matrices we can notice that the algorithm scales well with the space dimension. In fact, the sequential times are proportional, while the parallel times are clearly not so proportional due to the big overhead caused by a big number of threads (which has a bigger impact on the smaller matrices).