A parallel approach for the Jacobi method

Michele Morisco 505252

A.Y. 21/22

Abstract

This report summarizes the main choices made for the Parallel and Distributed Systems Paradigms and Models project. It consists of implementing the Jacobi resolution method for a linear system using threads and FastFlow. We will focus on the main coding aspects and on the results achieved in the experiments. In particular, we will analyze the effects of different choices of parameters, i.e. the size of the matrix and the parallel degree.

1 Introduction

The report describes the development of a parallel version of the Jacobi method that is used to solve systems of linear equations $Ax = b$ where $A \in \mathbb{R}^{nxn}$ is a strictly diagonally dominant square matrix, $b \in \mathbb{R}^n$ is the vector of known terms and $x \in \mathbb{R}^n$ is the vector of variables. The Jacobi method is an iterative method that at each step computes a more accurate approximation of the solution vector with respect to the previous step.
The formula of the method is:

$$x_i^{k+1} = \frac{1}{(A_{ii})}(bi - \sum_{j=1, j\neq i}^{n} A_{ij}x_j^k)$$

Assuming that matrix A is a strictly diagonally dominant matrix, the method will converge to the solution of the linear system starting from every point $x_0 \in \mathbb{R}^n$.
We will implement a function that generates a random matrix with this hypothesis for the experiments.

2 Implementation

In this section, we describe various versions of the Jacobi method. Firstly, we will see the sequential version to compute costs for experiments and finally some parallel versions to compare with each other.

2.1 Sequential version

The sequential version of the Jacobi method in algorithm 1 consists of three nested loops, one with a number of iterations, and one with the components of

a vector that represents the results of each iteration. The last one represents the horizontal components of the matrix.
The innermost loop splits into two distinct loops, which scan the matrix to sum the elements from 0 to n, but jumping the A[i][i] element. Because checking the condition $i \neq j$ in each iteration has been expensive. In the experiments, it is possible to notice that after a number of iterations the algorithm from a certain point, produces very small variations in the results. To avoid these useless operations we can add a stopping criterion, where if the difference between x_{old} and x_{new} is less or equal to threshold ϵ then we could stop the execution early, and produce the final result.

Algorithm 1: Sequential version

```
Data: A the n x n matrix, b the right hand side vector n x 1, N the
      number of iterations, $x_{old}$ previous iteration on vector x
Result: $x_{new}$ last iteration on vector x
for $k \leftarrow 0$ to $N$ do
    for $i \leftarrow 0$ to $n$ do
        $sum \leftarrow 0$
        for $j \leftarrow 0$ to $i$ do
            $sum \leftarrow sum + A[i][j] * x_{old}[j]$;
        end
        for $j \leftarrow i + 1$ to $n$ do
            $sum \leftarrow sum + A[i][j] * x_{old}[j]$;
        end
        $x_{new}[i] \leftarrow (b[i] - sum)/A[i][i]$;
    end
    if $\frac{\|x_{old} - x_{new}\|_1}{n} \leq \epsilon$ then
        break ;          /* Check criterion to stop execution */
    else
        $x_{old} \leftarrow x_{new}$ ;          /* otherwise update old value */
    end
end
```

2.2 Parallel version

In order to implement the parallel version with the native threads library, in the sequential algorithm there is a loop that could have been done in parallel, i.e. the loop on the single horizontal components of x_{new}.
Indeed, the parallel pattern used on this loop is a map data-parallel computation where partitioned the data in chunks, according to the parallel degree.
In order to do a balance distribution of data so each thread would do almost the same amount of work we compute the chunk dimension:

$$dimension = \lfloor \frac{n}{(num\_threads)} \rfloor + 1$$

In the average case, each thread works with more variables, but in the worst case, a thread that computes $n - (dimension * (num\_threads - 1))$ of them, remains unused. Fortunately, this case is uncommon and happen usually when

n is small and $n \approx num\_threads$.
Each worker is assigned a lambda expression and computes the assigned chunk of the matrix A since we are computing K iterations so each thread must wait until all the others have completed the step k before they go to the next step. An approach to avoid much overhead is to use the synchronization methods to force all threads to wait for the termination of a step. Using the barrier object, we can achieve better results with the function `arrive_and_wait` at end of each iteration.
In this case, the stopping criterion is into the barrier callback, and to improve the speed up of computations the loop is divided among the threads, and in each one, we computed the norm 1 of the difference between x_{old} and x_{new}. In this way, in the barrier callback, there is the sum of the various norms computed and divided the results by the number of variables.
Alg.2 and Alg. 3 report the codes about the parallel version implemented for the experiments.

Algorithm 2: Parallel version

```
Data: A the n x n matrix, b the right hand side vector n x 1, N the
      number of iterations, $x_{old}$ previous iteration on vector x, m
      index of the first variable to compute, M index of the first
      variable of the next chunk
Result: $x_{new}[m]...x_{new}[M-1]$
while $n > 0$ do
    for $i \leftarrow m$ to $M$ do
        same loops as the sequential version and update $x_{new}$;
    end
    Barr.arrive_and_wait() ; /* Block thread until all ones reach
     this line */
end
```

Algorithm 3: Barrier callback

```
Data: N the number of iterations, n the number of thread, $x_{old}$
      previous iteration on vector x, $x_{old}$ last iteration on vector x
for $i \leftarrow 0$ to $n$ do
    $sum \leftarrow sum + norm[i]$ ;                /* sum partial norms */
    $norm[i] \leftarrow 0$;
end
$sum \leftarrow sum/matrix\_size$;
if $sum \leq \epsilon$ then
    $N \leftarrow 0$ ;           /* Check criterion to stop execution */
else
    $N \leftarrow N - 1$ ;                    /* decrease iterations */
    $x_{old} \leftarrow x_{new}$ ;                      /* update old value */
    $sum \leftarrow 0$;
end
```

In addition, there is another parallel implementation where it uses *thread* *pinning* on the cores.

2.3 FastFlow version

FastFlow is a C++ API that allows using high-level parallel patterns simplifying the development of parallel applications. In order to parallelize the algorithm we used the `ff::ParallelFor` that requires a lambda function corresponding to the iterations of the associated cycle that should be parallelized.
In this implementation, it is used the *spinwaits* to speed up the computation, especially in cases where the grain is small and with very short waits, spinning may be preferable to blocking as it avoids context switch overhead.
In our case, the lambda function corresponds to the internal cycle of sequential code.

3 Experiments

In the experiments, we will consider some metrics such as scalability, speedup, and efficiency to evaluate the performances of algorithms. These parameters are computed using T_{seq}, the sequential time, $T_{par}(n)$, the parallel time with n workers, and $T_{par}(1)$, the parallel time with 1 worker using the provided class utimer. The completion time does not take into account the time spent generating the matrices and vectors. The tests are performed by setting the number of iterations to **150** and using matrices with different sizes and repeating the experiments **5** times by averaging the results to get a more reliable them.
The experiments use matrices with sizes **1000**, **5000**, and **20000**. In addition, the stopping criterion used is $\epsilon = 10^{-11}$.
Below there is the information on how to run the tests:

```
./jacobi_seq n_iterations dim_matrix

./jacobi_par n_iterations dim_matrix n_threads show_result

./jacobi_pinned n_iterations dim_matrix n_threads show_result

./jacobi_ff n_iterations dim_matrix n_threads show_result
```

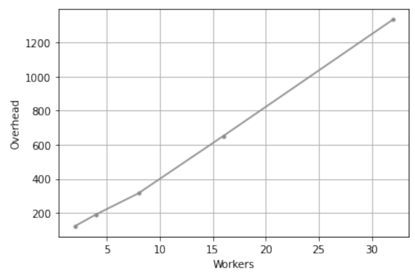
It is possible to digit *help* command to visualize a guide to use the scripts. For example:

```
./jacobi_seq help
```

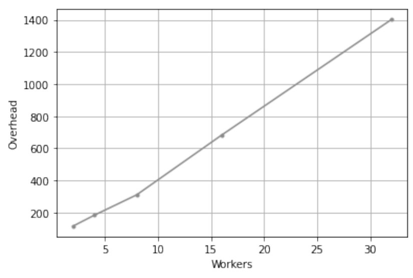
In addition, there is a *makefile* that provides to compile all scripts by using the command `make all` and there is a command to delete all files compiled thanks to `make clean`.
Finally, the experiments were conducted on a remote machine provided by the University of Pisa.

4 Results

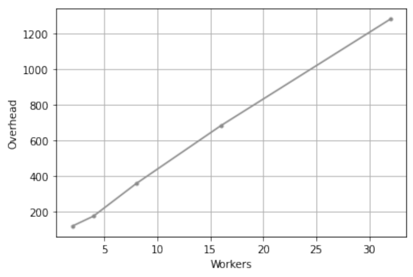
The following section shows the results obtained by running the proposed approaches with different parameters.



(a) $N = 1000$



(b) $N = 5000$



(c) $N = 20000$

Figure 1: Thread overheads for different sizes of matrix

Firstly, we can define the ideal completion time by running the method sequentially and computing it for each number of workers, then in Table 1 we can see the ideal completion time by different sizes of the matrix.

N° worker	1000	5000	20000
1	40641	973872	14675933
2	20321	486936	7337967
4	10160	243468	3668983
8	5080	121734	1834491
16	2540	60867	917245
32	1270	30434	458623

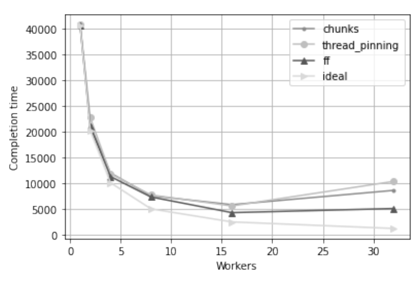
Table 1: Ideal completion time expressed in microseconds for each matrix size

In Figure 1, we can see the trend of thread overheads depending on the number of workers used on parallel execution.

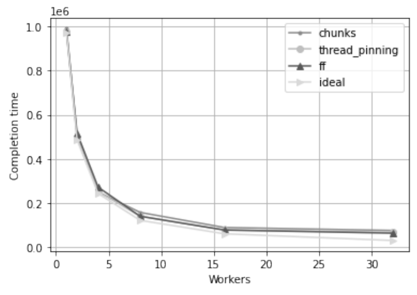
As expected, increasing the number of workers the overhead increases linearly.

Figure 2 is shown the plots of the completion time of each version with respect to the ideal time, we notice that with a small size matrix, all parallel executions have a decline using more workers due to overhead; this fact leads to an increase in the completion time. On the contrary, with a larger matrix, the resources bring benefits for each version. Generally, the FastFlow version runs better with respect to other ones.

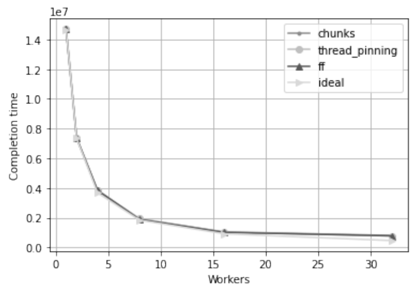
Figure 3 shows the speedup trend and confirms as we mentioned before, a good speedup is achieved more or less up to the use of 16 workers in larger matrices, but with 32 threads the overhead is present which causes a decrease in the performances. In addition to the time spent on thread setup, there are other



(a) $N = 1000$



(b) $N = 5000$



(c) $N = 20000$

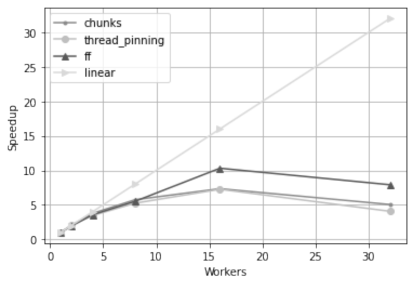
Figure 2: Completion time for different sizes of matrix

possible overheads given by the cache coherency protocol which is activated by the system to make the vector x_{new} coherent between the caches. When a thread writes to the vector, the x_{new} vector is made consistent with respect to all other threads by the cache coherence protocol by copying the vector to the caches of all cores with the result that they are generating overheads.

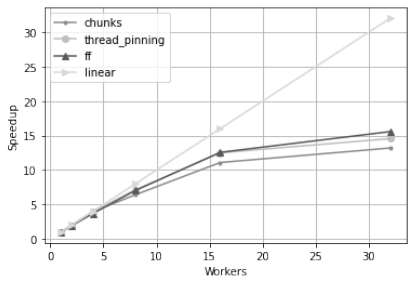
In Figure 4 we can see how the efficiency decreases with a higher number of workers, this behavior is predictable, as shown by speedup. In particular, it confirms that FastFlow and thread pinning versions get the best results.

5 Conclusions

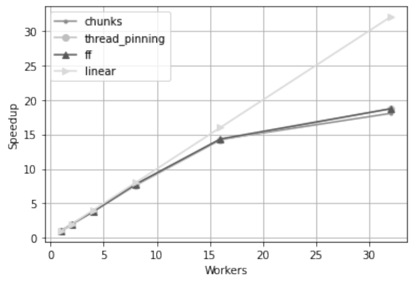
From the experiments' results, we have seen different approaches to parallelize the Jacobi method like as splitting the matrix into chunks, using thread pinning, and, finally, the FastFlow implementation. The tests were conducted using matrices of different sizes so they have shown different behaviors due to overheads. After that, the larger matrices make better use of the parallelization, indeed the linear speedup can be more or less maintained by using up to a maximum of 16 threads as the efficiency remains good up to 16 workers. Regardless, using 32 threads lead to benefits on completion time despite the overheads. Generally, the FastFlow implementation achieves better results compared to the other ones. We conclude that the parallelization of Jacobi's method brings benefits to execution time.



(a) $N = 1000$



(b) $N = 5000$

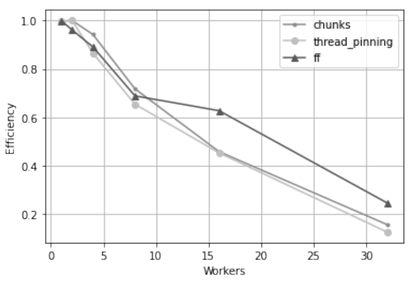


(c) $N = 20000$

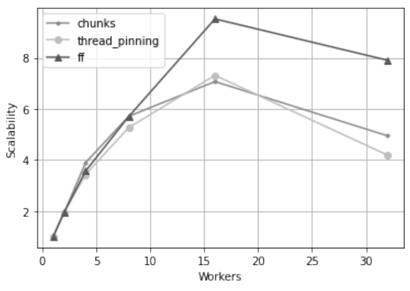
Figure 3: Speedup for different sizes of matrix

$N = 1000$						
N° worker	$T_{par}(n)$	$T_{pinning}(n)$	$T_{ff}(n)$	$Speedup_{par}$	$Speedup_{pinning}$	$Speedup_{ff}$
1	40768	40847	40693	0.9969	0.9950	0.9987
2	21300	22780	21002	2.0343	2.0335	1.9222
4	12043	11901	11355	3.7672	3.4596	3.5620
8	7616	7777	7388	5.7414	5.2348	5.5096
16	5888	5641	4345	7.3402	7.2577	10.3079
32	8669	10416	5138	5.0497	4.0682	7.9134
$N = 5000$						
N° worker	$T_{par}(n)$	$T_{pinning}(n)$	$T_{ff}(n)$	$Speedup_{par}$	$Speedup_{pinning}$	$Speedup_{ff}$
1	990145	974513	976888	0.9836	0.9993	0.9969
2	501945	497153	517277	1.9925	1.9801	1.9179
4	250613	251736	272143	3.9341	3.9856	3.6936
8	157593	140441	140102	6.4130	7.1296	7.0273
16	89198	78100	77741	11.0906	12.4996	12.5643
32	75672	67316	63597	13.1942	14.5545	15.5698
$N = 20000$						
N° worker	$T_{par}(n)$	$T_{pinning}(n)$	$T_{ff}(n)$	$Speedup_{par}$	$Speedup_{pinning}$	$Speedup_{ff}$
1	14685321	14688154	14696567	0.9994	0.9992	0.9986
2	7383517	7381032	7376303	1.9662	1.9955	2.0028
4	3797270	3822855	3841610	3.8729	3.8058	3.7833
8	1907168	1940161	1883098	7.6714	7.5700	7.7664
16	1039539	1025081	1013914	14.2707	14.1792	14.3326
32	805475	775927	775018	18.0381	18.7192	18.7039

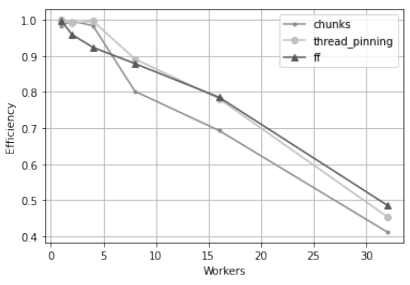
Table 2: Real completion time is expressed in microseconds and speedup for each matrix size, divided into chunks, thread pinning, and FastFlow.



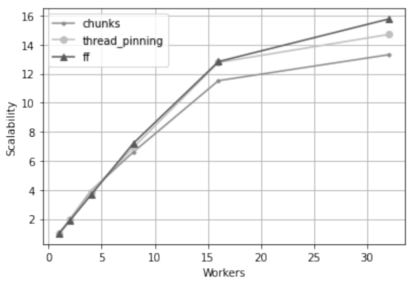
(a) Efficiency with $N = 1000$



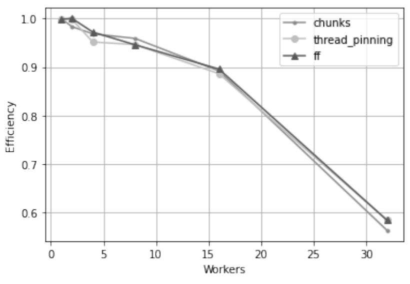
(b) Scalability with$N = 1000$



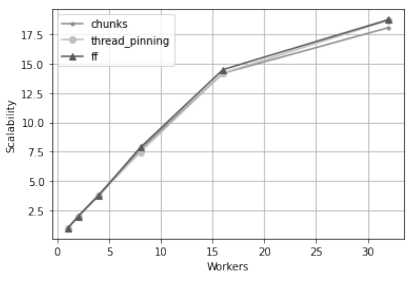
(c) Efficiency with $N = 5000$



(d) Scalability with $N = 5000$



(e) Efficiency with $N = 20000$



(f) Scalability with $N = 20000$

Figure 4: Efficiency and scalability with different matrix sizes

$N = 1000$						
N° worker	$Scalab_{par}(n)$	$Scalab_{pinning}(n)$	$Scalab_{ff}(n)$	Eff_{par}	$Eff_{pinning}$	Eff_{ff}
1	1.0000	1.0000	1.0000	0.9969	0.9950	0.9987
2	1.9105	1.9327	1.9453	1.0000	1.0000	0.9611
4	3.8927	3.4291	3.5635	0.9418	0.8649	0.8905
8	5.7108	5.2675	5.6991	0.7177	0.6544	0.6887
16	7.0713	7.3160	9.5272	0.4588	0.4536	0.6276
32	4.9472	4.1907	7.9070	0.1578	0.1271	0.2473
$N = 5000$						
N° worker	$Scalab_{par}(n)$	$Scalab_{pinning}(n)$	$Scalab_{ff}(n)$	Eff_{par}	$Eff_{pinning}$	Eff_{ff}
1	1.0000	1.0000	1.0000	0.98357	0.9993	0.9969
2	1.9578	1.9578	1.8838	0.9963	0.9901	0.9590
4	3.9416	3.8636	3.6536	0.9835	0.9964	0.9233
8	6.6192	6.9326	7.2179	0.8016	0.8912	0.8784
16	11.5234	12.7794	12.8351	0.6932	0.7812	0.7853
32	13.3051	14.6996	15.7629	0.4123	0.4548	0.4866
$N = 20000$						
N° worker	$Scalab_{par}(n)$	$Scalab_{pinning}(n)$	$Scalab_{ff}(n)$	Eff_{par}	$Eff_{pinning}$	Eff_{ff}
1	1.0000	1.0000	1.0000	0.9994	0.9992	0.9986
2	2.0062	1.9657	2.0201	0.9831	0.9976	1.0000
4	3.8790	3.8081	3.7946	0.9682	0.9515	0.9717
8	7.6954	7.4701	7.8609	0.9589	0.9463	0.9458
16	14.1819	14.1913	14.4897	0.8919	0.8862	0.8958
32	18.0582	18.6861	18.7365	0.5637	0.5850	0.5845

Table 3: Scalability and efficiency for each matrix size, divided into chunks, thread pinning, and FastFlow.