

Project of Distributed Systems: Paradigms and Models Iterative Jacobi Method

Student: Diego Arcelli

Matricola: 647979

Professor: Marco Danelutto

June 19, 2022



UNIVERSITÀ DI PISA

Contents

1	Problem description	2
2	Analysis of the problem	2
3	Time analysis	4
4	Implementation	5
4.1	C++ thread	5
4.2	FastFlow	5
4.3	Open MP	5
5	Experiments	5
6	Manual	6

1 Problem description

The Jacobi method is an iterative algorithm which can be applied to solve systems of linear equations of the form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is the coefficient matrix, $b \in \mathbb{R}^n$ is the constant terms vector and $x \in \mathbb{R}^n$ is the unknown vector that we want to compute. The algorithm compute iteratively x as follows:

$$x^{(k+1)} = D^{-1}[b - (U + L)x^{(k)}]$$

where $x^{(k)}$ indicates the value of x at the k -th iteration. Actually at each iteration k we can compute independently every component i of the vector using the equation:

$$x_i^{(k+1)} = \frac{1}{a_{i,i}} \left(b_i - \sum_{j=1}^n a_{i,j} x_j^{(k)} \right) \quad (1)$$

where x_i indicates the i -th component of vector x . Discussing further details of the algorithm is beyond the scope of this project.

2 Analysis of the problem

The pseudo-code to execute the algorithm sequentially for a fixed amount of iteration l , is shown in 1. Note that at each iteration we're forced to use an auxiliary vector x' to compute the updated elements of x for the current iteration, and then, before starting the following iteration, we copy the elements of x' in x . This is necessary because if at iteration k , we compute $x_1^{(k)}$ modifying x in-place, then when we compute $x_2^{(k+1)}$ instead of using $x_1^{(k)}$, we'll use $x_1^{(k+1)}$. The algorithm is composed by three nested loops, where the first iterates l times, and the other two iterate n times, so the total time complexity is $O(ln^2)$. The swap function used in the pseudo-code to copy the elements of x' in x , can be implemented with $O(1)$ cost, by simply swapping the pointers of the two array (this works in C++ where we see as vectors).

Algorithm 1 Sequential code for Jacobi method

Require: A matrix, b and x vectors, l positive integer

```
1: for  $k \leftarrow 1$  to  $limit$  do
2:   for  $i \leftarrow 1$  to  $n$  do
3:      $val \leftarrow 0$ 
4:     for  $j \leftarrow 1$  to  $n$  do
5:       if  $j \neq i$  then
6:          $val \leftarrow val + A[i, j] * x[j]$ 
7:       end if
8:     end for
9:      $x'[i] \leftarrow \frac{1}{A[i, i]}(b[i] - val)$ 
10:  end for
11:   $swap(x, x')$ 
12: end for
```

Let's now analyze how we can exploit the properties of the problems in order to parallelize the algorithm. The first important thing we notice is that in order to compute the updated values for the vector x at iteration k , we need the full updated vector x at iteration $k - 1$, therefore the iterations of the algorithm needs to be strictly sequential. So let's focus on understanding what we can parallelize inside a single iteration. As we said in the description of the problem, at each iteration k every component of the new vector $x^{(k+1)}$ can be computed independently from the others components using the formula in equation 1. So this is a data parallel computation where (for the reasons described before) we cannot modify the vector x in place. So we can parallelize the code by assigning to each worker a different partition of components of x . The usage of multiple workers introduces the following synchronization problems:

1. Start iteration $k+1$ only after all the workers finished their computations for iteration k , since as we said at the beginning of the analysis iterations must be strictly sequential
2. Executing the swap of the main and auxiliary vectors only after all the workers finished their computations, otherwise it might happen that the vectors are swapped while other vectors have still to finish their computation
3. Keep track of the number of iteration so that all the workers will stop after $limit$ iterations.

The first problem can be solved simply by putting a barrier after the execution of the map, so that all the worker will be forced to wait the others before starting the next iteration. In this way we can also solve the third problem by selecting one of the worker to count the number of iterations of the algorithm executed so far. The designated worker can simply use a shared variable

among the workers to increase by one at the end of every iteration, before the barrier (since the variable is modified by just one worker it doesn't need synchronization mechanism to be accessed). This solution guarantees that before starting iteration k the shared variable will have value k for all the workers, so every worker can individually check if $k = \textit{limit}$ and stop its execution. The second problem can be solved simply by adding a second barrier, which will be placed between the end of the map and the next iteration barrier, and designate one of the worker to execute the swap of the vectors after the barrier (it can be the same worker which updates the iterations counter). In this way the swap will happen only after every worker finished its computations for the current iteration.

Algorithm 2 Worker pseudo-code

Require: A matrix, b and x vectors, l positive integer

```

1: while  $k < l$  do
2:   for  $i \leftarrow \textit{start}$  to  $\textit{end}$  do
3:      $\textit{val} \leftarrow 0$ 
4:     for  $j \leftarrow 1$  to  $n$  do
5:       if  $j \neq i$  then
6:          $\textit{val} \leftarrow \textit{val} + A[i, j] * x[j]$ 
7:       end if
8:     end for
9:      $x'[i] \leftarrow \frac{1}{A[i, i]}(b[i] - \textit{val})$ 
10:  end for
11:  Vectors swap barrier
12:  if  $\textit{start} = 0$  then
13:     $\text{swap}(x, x')$ 
14:     $k \leftarrow k + 1$ 
15:  end if
16:  Next iteration barrier
17: end while

```

The variables \textit{start} and \textit{end} represent respectively the first and the last index of x assigned to the worker to compute. Note that both of the barrier are needed, because if we only use the copy barrier, it might happen that one of the worker might start the next iteration before the designated worker could swap the vectors and update the iteration counter, so both barriers are needed.

3 Time analysis

In the sequential case as we said the complexity of the algorithm is:

$$T_{seq} = T_{init} + k(n^2 t_{\oplus} + t_{swap})$$

as we already said the swap can be implemented just by swapping the pointer and the, so the time is approximately:

$$T_{seq} \approx kn^2t_{\oplus}$$

4 Implementation

The parallel program has been implemented in three versions:

- The first using C++ threads
- The second using FastFlow
- The third using Open MP

4.1 C++ thread

In this implementation we use the fork-join mechanism to spawn nw threads, each of them is assigned to a chunk of x of size $\lfloor \frac{n}{nw} \rfloor$, and each of them executes code in 2.

4.2 FastFlow

For the fast flow implementation we simply use the `ParallelFor` class of FastFlow to parallelize the for loop which iterates over

4.3 Open MP

5 Experiments

The goals of the experiments that we will execute are the following:

- Comparing the performances obtained by the three implementations
- Check how the performances change with respect to the size of the matrix

To compare the performances of the three implementations we will compute the speedup obtained when executed with nw workers, varying nw from 1 to 128, and showing the results in a plot. To see if the size of the matrix impacts on the speedup, we'll repeat the procedure for matrices of different sizes (100, 500, 1000). Note: in order to avoid good and bad case, to take the time obtained by one implementation with a certain number of workers, we execute the program 5 times and take the average time.

After that we'll select the number of workers for which we obtain the best speedup and we'll execute the algorithms with that number of workers for matrices of different sizes (10, 20, 100, 200, 500, 1000, 2000, 5000, 10000) to better understand the impact that the size of the matrix has on the performances. Even in this case each time is the mean of five executions.

6 Manual

To compile the code is sufficient to execute the command `make` in the `src` directory of the project. The compilation will produce the executable `main` which can be called to launch the program providing the following arguments:

- **n:** and integer which represent the number of row and columns of the matrix
- **nw:** the number of workers that will be used in the parallel versions of the algorithm
- **iterations:** the number of iterations of the algorithm

Once execute the program will execute the sequential algorithm and the three parallel implementations and report the execution times in nanoseconds. For the experiment