Project of Distributed Systems: Paradigms and Models
Parallel Iterative Jacobi Method

Student: Diego Arcelli
Matricola: 647979
Professor: Marco Danelutto

June 25, 2022

UNIVERSITÀ DI PISA

Contents

1 Analysis of the problem **2**

2 Time analysis **4**

3 Implementations **4**
3.1 C++ thread 5
3.2 FastFlow 5
3.3 OpenMP 5

4 Manual **5**

5 Experiments **5**

6 Conclusions **6**

1 Analysis of the problem

Notation premises, with $x_i^{(k)}$ we indicate the i-th entry of vector x at iteration k. The pseudo-code to execute the Jacobi method sequentially, for a fixed amount of iterations, is shown in 1. Note that we are 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'll 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 *limit* times, and the other two iterate n times, so the time complexity is $O(limit \cdot n^2)$. The *swap* function used to copy the elements of x' in x, can be implemented in $O(1)$ time complexity, by simply swapping the pointers of the main and auxiliary arrays.

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 problem 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 need to be strictly sequential.
A relevant property of the algorithm that we can notice from the pseudo-code, is that each element $x_i^{(k)}$ is computed independently from the others elements of the vector x, since to compute $x_i^{(k)}$ we just need the old vector $x^{(k-1)}$ and the matrix A. Therefore the iteration of the loop at line 2 of the pseudo-code, can be executed in parallel, which is the classical application of a map pattern. So, if we dispose of nw workers, we can assign to each worker a partition of indices of x to compute.
The innermost loop at line 4 is used to compute a sum of $n-1$ elements, which is the classical application of a reduce pattern. This is done inside the loop that we already said we can parallelize using a map pattern, so, if we also want to implement also the reduce, for each one of the nw workers of the map, we have to arrange other nw' additional workers that the worker which is computing $x_i^{(k)}$ can use apply the reduce to the computation of the summation.
The introduction of the reduce inside each map would increase the overhead, since each worker assigned to the map would have to coordinate its computations with the workers that it uses for the reduce. Moreover, if we decide to use nw workers for the map, and nw' workers for the reduce, in total we would have $nw_{tot} = nw + nw \cdot nw'$ workers. But we could have simply assigned all the nw_{tot} workers to the map, and with the same parallel degree we would have the same theoretical speedup but with less overhead. Therefore there is no advantage in implementing the reduce.

Even if we only implement the map, there some problem that we need to care about:

1. Start iteration $k+1$ only after all the workers finished their computations for iteration k, since as we said the 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 some workers are still computing their portion of $x^{(k)}$

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 $k = limit$ 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 ← start to end do
3:          val ← 0
4:          for j ← 1 to n do
5:              if j ≠ i then
6:                  val ← val + A[i, j] ∗ x[j]
7:              end if
8:          end for
9:          x'[i] ← 1/A[i,i] (b[i] − val)
10:     end for
11:     Vectors swap barrier
12:     if start = 0 then
13:         swap(x, x')
14:         k ← k + 1
15:     end if
16:     Next iteration barrier
17: end while
```

The variables *start* and *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.

2 Time analysis

In the sequential case, the time required to execute the algorithm is:

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

where k is the number of iterations, n is the number of row and columns of the matrix, T_{init} is the time required to initialize the computations, $T_{\oplus}$ is the time required to compute the expression at line 6 of 1 and T_{swap} is the time required to perform the swap of the vectors. As we explained before, the swap is just a swap of pointer, so T_{swap} is negligible. T_{init} includes the creation of the auxiliary array, an operation whose cost depend by n.
In the parallel case, the second innermost loop of 1 will be parallelized, so, with nw workers, the program will execute at the same time nw parallel loops, each of which will iterate more or less $\frac{n}{nw}$ times. The overhead introduced is given by the forking and joining of the threads, and by the synchronization with the two barriers, so the total time is:

$$T_{par}(nw) = T_{init} + T_{fork} + k(\frac{n}{nw} n T_{\oplus} + T_{synch} + T_{swap} + T_{synch}) + T_{join}$$

where T_{fork} and T_{join} are the times for the fork-join mechanism, and T_{synch} is the time required to handle the barrier. So the speedup is:

$$Speedup(nw) = \frac{T_{init} + k(n^2 t_{\oplus} + t_{swap})}{T_{init} + T_{fork} + k(\frac{n}{nw} n t_{\oplus} + T_{synch} + T_{swap} + T_{synch}) + T_{join}}$$

If we consider negligible everything that happen outside the main loop of the program, then the speedup is:

$$Speedup(nw) \approx \frac{k n^2 t_{\oplus}}{k \frac{n}{nw} n t_{\oplus}} = nw$$

which is the ideal speedup which we can theoretically obtain with a map pattern. Of course to have a better estimate of the speedup we should consider the serial fraction of the program. We can make a rough estimation by considering that the total number of operations done by the algorithm is more or less in the order of kn^2, since we have to do some fixed amount of operations inside the three loops. Ideally, with n workers, every worker could compute its own $x_i^{(k)}$ in more or less n operations, reducing the number of operation to kn, so a rough estimation of the serial fraction f is:

$$f = \frac{kn}{kn^2} = \frac{1}{n}$$

if we now consider the speedup given by the Amdahl's law, we get that in the best case the maximum speedup is:

$$Speedup < \frac{1}{f} = \frac{1}{\frac{1}{n}} = n$$

So the speedup is upper-bounded by the number of rows/columns of the matrix. This makes sense, since in an ideal scenario, we would have a worker for each component of the vector x, reducing the cost of a single iteration of the algorithm from $O(n^2)$ to $O(n)$. So we should expect a better speedup as n increases.

3 Implementations

The parallel program has bee implemented in three versions:

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

3.1 C++ thread

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

3.2 FastFlow

For the FastFlow implementation we use thed `ParalleFor` class of FastFlow

3.3 OpenMP

For the OpenMP implementation we simply parallelized the second innermost loop at line 2 of algorithm 1, using the `#pragama omp paralle for` directive. Since,

4 Manual

The source code, the scripts written for the experiments and a detailed manual of the program are available at this repository.
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 launched to execute the program, providing the following arguments:

- `n`: 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
- `iters`: the number of iterations of the algorithm
- `mode`: an integer number which species which implementation of the algorithms will be executed

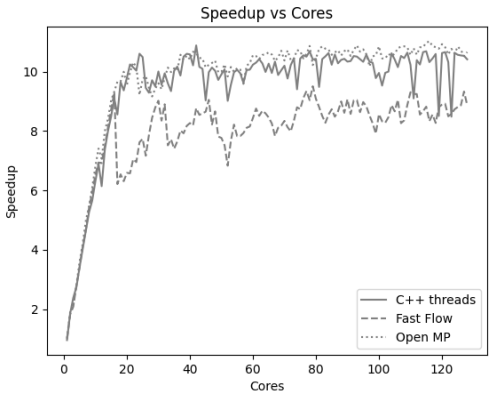
The possible values of `mode`, with the corresponding effects, can be showed by launching the program with no argument. Once executed the program will run the versions of the algorithm required (according to the value of `mode`), and it will print their execution times.

5 Experiments

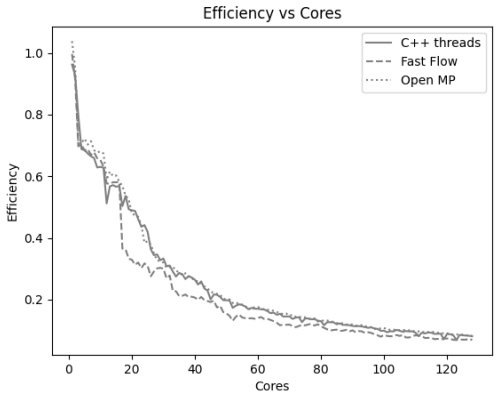
The goals of the experiments that we will execute are the following:

- 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 impact 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 obtain by one implementation with a certÈ nsain number of worker, 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.

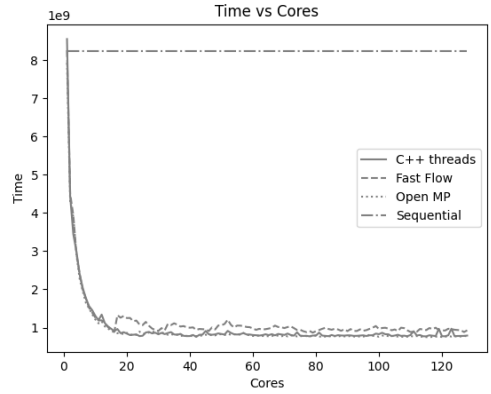


(a) label 1

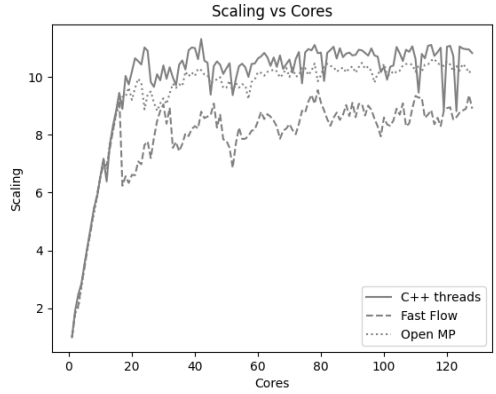


(b) label 2

Figure 1: 2 Figures side by side

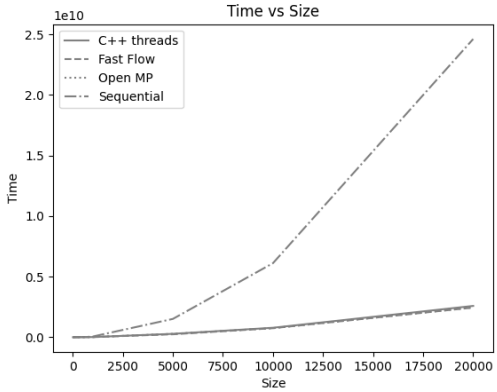


(a) label 1

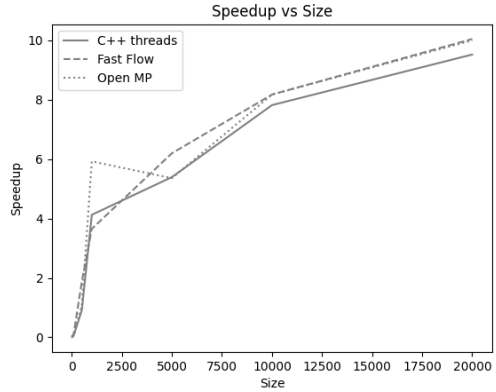


(b) label 2

Figure 2: 2 Figures side by side



(a) label 1



(b) label 2

Figure 3: 2 Figures side by side

6 Conclusions