Project of Distributed Systems: Paradigms and Models
Iterative Jacobi Method

Student: Diego Arcelli
Matricola: 647979
Professor: Marco Danelutto

May 31, 2022

UNIVERSITÀ DI PISA

Contents

1 Description of the problem

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 idea of Jacobi algorithm is to decompose A as a sum of three matrices:

$$A = D + U + L$$

where D is a diagonal matrix, U is an upper-triangular matrix and L is a lower-triangular matrix. Then we compute iteratively:

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

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 array using the equation:

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

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 suppose that at iteration $k+1$, we compute the updated value for the i-th component $x_i^{(k+1)}$ and we modify x in-place, then when we compute the update for the next component $x_{i+1}^{(k+1)}$ we won't use x_i^k, but we will use x_i^{k+1}, which is not mathematically correct. The algorithm is composed by an three nested loops, where the first iterates l times, the other two iterate n times, and all the other operations can be done in constant time, so the total time complexity is $O(ln^2)$. Actually we can optimize the code, by avoiding to do the copy loop, if we see x and $x\prime$ as pointers (which is ok since in C++ arrays are pointers), we can simply swap the addresses that they're pointing instead of copying every element of $x\prime$ in x. In this way we substitute the $O(l)$ copy loop, with a $O(1)$ swap operation, even thou the overall cost of the iteration is still $O(n^2)$.

Algorithm 1 Sequential code for Jacobi method

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

```
for k ← 1 to l do
    for i ← 1 to n do
        val ← 0
        for j ← 1 to n do
            val ← val + A[i, j] ∗ x[j]
        end for
        x'[i] ← 1/A[i,i] (b[i] − val)
    end for
    for i ← i to n do
        x[i] ← x'[i]
    end for
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 to update in the auxiliary vector x'.
Inside this map operation we have a sum of $n-1$ elements, which is the classical case of a reduce pattern. So inside each map we can call a reduce in the following way:

$$map(reduce(+, nw_2), nw_1)$$

where nw_1 is the number of workers used for the map and w_2 is the number of workers used inside each map for the reduce, so the number of total workers nw will be:

$$nw = nw_1 \cdot nw_2$$

Therefore, we have the problem of determine the best way to use the workers of the machine.
For the copy of $x\prime$ we can simply Basically we simply need to put a barrier between the update phase and the copy phase. Since before starting the copy, the previous map must be ended, for this second map we can use all the $nw_1 \cdot nw_2$ workers that we have at disposal.
After the map is computed the swap of $x\prime$ and x can be arbitrarily assigned to one of the workers. To do this, before the worker starts to swap the vectors we

have to assure that all the other workers finished computing they're portion of $x\prime$. This can be achieved by simply putting a barrier between the map-reduce and the arrays swap. Actually another barrier is needed to assure that no workers starts the next iterations before the worker designated to the swap ended the computation, otherwise it might happen that some workers start the next iteration working with not swapped $x\prime$ and x.
The last thing we need to discuss is how to stop the the execution of the workers after l iteration. This can be easily implemented by using the worker assigned to the swap to also increase the iterations counter. This solution does not require mutual exclusion, since the counter is modified by just one worker, moreover it guarantees the correctness of the program since, all the workers will start the next iteration with the same value of the iteration counter.