Parallel Jacobi

Final Project for the SPM course

Francesco Balzano

Master Degree in Computer Science and Networking

A.Y. 2016-2017

Contents

1 Problem

Write a parallel application that finds the solution of a system of linear equations using the Jacobi iterative method.

2 Dependency analysis

The algorithm iterates until either the maximum number of iterations is reached or a stopping condition occurs (*e.g.* a convergence condition).
Basically, at each iteration the algorithm computes the following equations:

$$\begin{cases} \sigma_{\mathrm{i}} = \sum_{j \neq i} A_{i,j} \, X_j^{(k)} \\ X_i^{(k+1)} = \frac{1}{A_{i,i}}(b_i - \sigma_{\mathrm{i}}) \end{cases}$$

$\forall i = 1, ..., n.$

Unfortunately, the two equations cannot be parallelized because they violate the Bernstein conditions. Indeed, if we consider the same iteration of the algorithm, a *READ-AFTER-WRITE* dependency holds for σ_{i}. We cannot even parallelize the second equation at step K and the first equation at step $K+1$, because in that case it holds a *READ-AFTER-WRITE* dependency on X. So $X_i^{(k+1)}$ can be started to be computed only after $X_i^{(k)}$ has been computed.

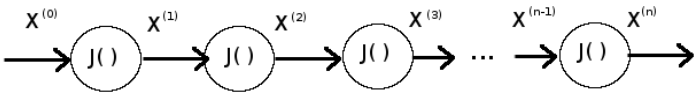

Figure 1: Chain of dependencies

3 Decomposition strategies

I analyze some alternative strategies to turn the problem into a set of concurrent activities. Each alternative is characterized by a different level of granularity for the concurrent activities. In this section the main metric kept into account to conduct the assessment of the alternatives is the *Completion Time* T_C, because the computation is a *data parallel* one. Besides, the completion time refers to the single iteration of the Jacobi iterative method. In the following, I am assuming that the concurrent activities are instantiated inside a loop run on a single processing element, so the time to setup them is fully paid. Instead, once setup they can (possibly) run in parallel, and so when assessing the completion time I consider the slowest path in the concurrency graph that must be serialized. In the following formulas I indicate with T_{aritm} the time neeeded to perform an aritmetic operation. This is a big semplification, introduced to keep more compact and readable the formula. The rationale is that any aritmetic operation takes $O(10^{-9})s$, so I use T_{aritm} and $O(10^{-9})s$ as synonyms. Furthermore I introduce T_{comm} to express the communication time. $T_{comm}(1)$ will be used when the amount of data sent is constant.

3.1 Tree of concurrent activities

We can decompose the concurrent activities in a tree made of n independent subtrees. Based on the kind of computation done, we can recognize three different types of concurrent activities:

- *Leaves:* Each leaf receives $A_{i,j}$ and $X_j^{(k)}$ and computes $A_{i,j} * X_j^{(k)}$;
- *Internal nodes:* Each internal node receives two quantities (one from each son), makes the sum and passes it to its father in the tree.
- *Roots(of the n subtrees):* The i^{th} root receives two quantities from its sons, makes their sum to finish the computation of σ_{i} and then computes $X_i^{(k+1)}$.

Performance model To deduce the performance model I consider for a moment the implementation of the concurrent activities. The details will be discussed later, but for the moment being I notice that the concurrent activities will be eventually mapped into processing elements. If I map each of these concurrent activities into a distinct processing element I have a waste of resources, because at any time a significant fraction of them will be idle. What I can do instead, is to map both internal nodes and roots into the processing elements that were used to run the leaves. Let us call *Workers* the leaves of the tree: then we are mapping the tree of concurrent activities into the array of workers $[W_{1,2}, W_{1,3}, ..., W_{n,n-1}]$.

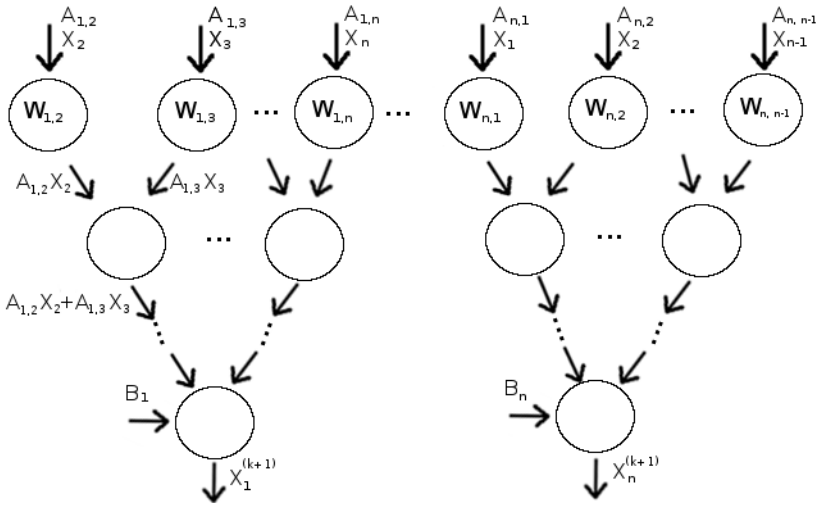

Figure 2: Decomposition strategy number 1: tree of workers

- **Number of Workers:** $O(n^2)$
- **Height(tree):** $O(\log n)$. Indeed the tree is made of n, independent subtrees, without cross edges between different subtrees. Each subtree is a complete binary tree with *n-1* leaves.
- Each level of the tree can be computed in parallel
- **Completion Time:** $T_C = O(n^2)\, T_{setup} + O(\log n)\, (T_{calc} + T_{comm}(1)) = O(n^2)\, T_{setup} + O(\log n)\, (T_{aritm} + T_{comm}(1))$. Since each concurrent activity makes a constant number of aritmetic operations, it's calculation time is modelled by T_{aritm}. The overhead of this decomposition strategy is given by the time to setup the concurrent activities ($O(n^2)\, T_{setup}$) and by the communication of intermediate results through the levels of the tree ($O(\log n)\, T_{comm}(1)$).

3.2 Array of concurrent activities

Each worker W_i receives $A_{i,*}$ and X^k and computes $X_i^{(k+1)}$, *i.e.* it computes the system of section 2 for some $i \in 1, 2, ..., n$ This strategy corresponds to collapse each independent subtree of figure 2 into a single worker.

Performance model

- **Number of Workers:** n
- **Completion Time:** $T_C = O(n)\, T_{setup} + T_{calc} + T_{comm}(1) = O(n)\, T_{setup} + O(n)\, T_{aritm} + T_{comm}(1)$. The calculation time T_{calc} of each concurrent activity is modelled by $O(n)\, T_{aritm}$, because each Worker makes approximately $O(n)$ sums and $O(n)$ multiplications. The overhead of this decomposition strategy is basically given by the time to setup the n concurrent activities, which is $O(n)\, T_{setup}$.

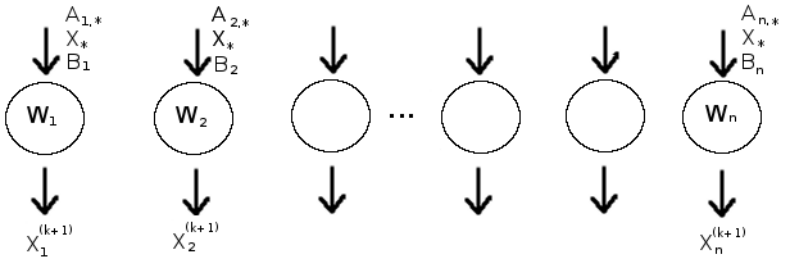

Figure 3: Decomposition strategy number 2: array of workers

3.2.1 Array of concurrent activities with pipeline

Since the computation of the worker in the decomposition number two takes a time linear and not constant, we could try to lower the service time by inserting a 2-stage pipeline. In this new parallelization, the first stage of the pipeline that parallelizes W_i computes σ_i, while the second stage computes $X_i^{(k+1)}$. We are forced to build this exact configuration of the pipeline because we must keep into account the data dependencies already pointed out in section 2.

Performance model This parallelization is not worth it, because stage one takes $O(n)\, T_{aritm}$ while stage 2 takes $O(1)\, T_{aritm}$, so if there is a bottleneck in decomposition number two we are not able to remove it with this parallelization. Furthermore a pipeline may reduce the service time but increases the latency, and since we are more interested in completion time than throughput we don't find it very useful .

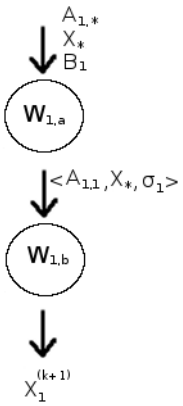

Figure 4: Decomposition strategy number 2, variant with pipeline: only the first worker is considered

3.3 Shortened tree of concurrent activities

We have already said that we can see the array of concurrent activities as a tree of concurrent activities in which each subtree has been collapsed into a single concurrent activity. These are the two extreme situations, but we can also consider an intermediate situation in which each subtree is implemented with $O(1)$ workers and not with $O(n)$. Let us assume that the total number of workers is m, with $n \leq m \leq n^2$.

Performance model

- **Number of Workers:** m
- **Height(tree):** $O(\log \frac{m}{n})$. Indeed the tree is made of n, independent, binary subtrees, each one having $O(\frac{m}{n})$ leaves.

- **Completion Time:** $T_C = O(m)\, T_{setup} + O(\frac{n^2}{m})\, T_{aritm} + O(\log \frac{m}{n})(T_{aritm} + T_{comm}(1)))$. The contribute $O(m)\, T_{setup}$ is paid to setup the concurrent activities, $O(\frac{n^2}{m})\, T_{aritm}$ is due to the leaves (which perform the most of the computations) and finally $O(\log \frac{m}{n})(T_{aritm} + T_{comm}(1)))$ is for the internal nodes and the roots (they perform a constant number of arithmetical operations).
  The nice thing of this decomposition is that it includes as special cases both decompositions one and two: indeed, if $m = O(n^2)$ than we get the completion time of decomposition one, whereas if $m = O(n)$ we get the completion time of decomposition two.

3.4 Single concurrent activitiy

We consider a single worker W that receives A and X and computes $X^{(k+1)}$, *i.e.* it computes the system of section 2 $\forall i \in 1, ..., n.$.

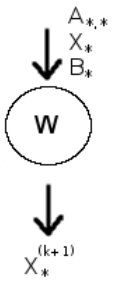

Figure 5: Decomposition strategy number 4: single worker

Performance model

- **Number of Workers:** 1
- **Completion Time:** $T_C = O(n^2)\, T_{aritm}$. If we instantiate the concurrent activity on a processing element different from the one running the main flow of control we should consider in principle also the time $O(1)\, T_{setup} + T_{comm}(n)$. Anyway this quantity can be neglected when compared to the shown completion time, especially if n is big.

4 Evaluation

In this section I carry out a theoretical evaluation of the alternative decomposition strategies based on their performance model. In order to conduct a quantitative evaluation of the alternatives I need some (approximated) value for T_{comm} and T_{setup}. These value are architecture-dependent, and we target a shared-memory architecture. I assume T_{comm} to be the time needed to access main memory, and based on the values found on Peter Norvig's site ([1]) I assume $T_{comm} = O(10^{-7})s$. This is a semplification that does not keep into account that some data is shared among processing elements, and even in the favourable case in which the implementation is lock-free there is some overhead in the management of shared blocks. For instance, if some block is replicated in the caches of different processing elements, if one processing element modifies such block than a cache coherence protocol will be fired to keep the value of such block consistent among caches. In a shared-memory architecture T_{setup} is the time to start a thread. Based on experimental values, I deduced that the time to start a thread should be approximately $O(10^{-5})s$.
Different decomposition strategies may be convenient for different data size. For this reason, I consider three different values for n: 100, 1000 and 10000.

Tree of concurrent activities: $T_C = O(n^2)\, T_{setup} + O(\log n)\, (T_{aritm} + T_{comm}(1))$	
n = 100	$T_C = (10^4 * 10^{-5} + 6,64 * (10^{-9} + 10^{-7}))s = O(10^{-1})s$
n = 1000	$T_C = (10^6 * 10^{-5} + O(10^{-7}))s = O(10)s$
n = 10000	$T_C = (10^8 * 10^{-5} + O(10^{-7}))s = O(10^3)s$

Array of concurrent activities: $T_C = O(n)\, T_{setup} + O(n)\, T_{aritm}$	
n = 100	$T_C = (10^2 * 10^{-5} + 10^2 * 10^{-9})s = O(10^{-3})s$
n = 1000	$T_C = (10^3 * 10^{-5} + 10^3 * 10^{-9})s = O(10^{-2})s$
n = 10000	$T_C = (10^4 * 10^{-5} + 10^4 * 10^{-9})s = O(10^{-1})s$

Single concurrent activity: $T_C = O(n^2)\, T_{aritm}$	
n = 100	$T_C = (10^4 * 10^{-9})s = O(10^{-5})s$
n = 1000	$T_C = (10^6 * 10^{-9})s = O(10^{-3})s$
n = 10000	$T_C = (10^8 * 10^{-9})s = O(10^{-1})s$

Looking at the tables above, we see that the solution with a single concurrent activity has the best performances. This strange result arises because we are considering too many concurrent activities. Indeed, if we look at the formulas in the first and second table, we can easily recognize that the bottleneck is the time to setup the concurrent activities (*i.e.* the threads). The solution is thus instantiating a smaller number of concurrent activities, which will do coarser grain computations. Because I want to make coarser grain computations in order to achieve better performances, I abandon the decomposition with the tree (also the shortened version). I only consider the decomposition with the array of workers for the implementation. In order to make a check that this could be the right way, let's consider 100 concurrent activities computing one iteration of the Jacobi iterative method with n = 10000. We get:

$$T_C = 100\, T_{setup} + O(\frac{n^2}{100})\, T_{aritm} = (10^2 * 10^{-5} + 10^6 * 10^{-9})s = O(10^{-3})s$$

that is the solution with the array made of 100 workers could be 100 times faster in computing the single iteration of the Jacobi method with respect to the sequential implementation. This is a result that comes from the assumption that the workers can do their job independently from one another. This could not be the case, for instance, if they need to read/write the data from/to the same place, their accesses could be serialized.

5 Implementation

I have implemented the Jacobi iterative method both in a sequential and in a parallel way. The sequential implementation is a simple single-threaded program. The parallel implementaion has been done both with pthreads and with fastflow. In the implemenation with pthreads I provide two alternatives: one using mutexes and condition variables, the other lock-free with busy waiting. In the implementation with fastflow, I have used the *parallelFor*

construct to run the concurrent activities in parallel.

I decided to implement the second decomposition strategy, because is the one that according to me ffers the best tradeoff "Parallelism Degree VS Overhead". In this section I describe the main points concerning implementation decisions. While in the previous section I considered a single iteration of the Jacobi method, now I take into account that it is an iterative method.
The problem is a **data parallel** one: so I can consider a **map** decomposition. I must take into account the following features:

- **Split** of the data to be sent to the workers. This is tipically done by a **scatter** process, that sends the appropriate partition to each Worker, but since:
  - W_i reads only $A_{i,*}$ and X^k
  - W_i writes $X_i^{(k+1)}$, but it is the *only one* that makes such write
  - I target a *shared-memory* architecture.

  I avoid using a scatter process that sends to each Worker the partition of data on which it has to do the computation. Instead, I will merge the scatter and the gatherer process into a single processing element and will assign the appropriate partitions to each worker only once at creation time.
- **Computation** done by each Worker W_i: at initialization time each worker is simply provided a reference to A, X and B and the index i. After the computation, it updates $X_i^{(k+1)}$ in memory and tells the scatter that it has completed the current iteration.
- **Gathering** of the results: the **gatherer** process waits that all the W_i have concluded the computation for the current iteration and then:
  - either stop the Workers and output the final X;
  - or notify all the Workers to go on with another iteration.

  As already pointed out, I merge the scatter and the gatherer into a single process. This allows me to save one processing element and to have a centralized "Locus of Control" to control the execution of the program.

So the structure of the chosen parallelization is basically a map without the gatherer.

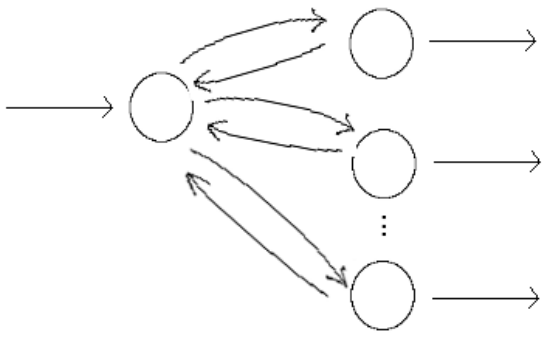

Figure 6: Structure of the implemented decomposition

References

[1] Peter Norvig. *Teach Yourself Programming in Ten Years*,http://norvig.com/21-days.html#answers.