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 hods a *READ-AFTER-WRITE* dependency on X. So the $X_i^{(k+1)}$ can be started to be computed only after $X_i^{(k+1)}$ has been computed.

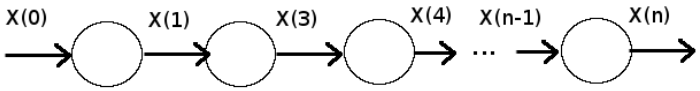

Figure 1: Chain of dependencies

3 Decomposition strategies

I analyze some alternative strategies to turn the problem into a set of concurrent activities. Each alternatice is characterized by a different level of granularity for the concurrent activities. In the followingI am assuming that the concurrent activities are instantiated inside a loop of a single processing elements, so the time to setup them is fully paid. Instead, once setup they can go in parallel, and so when assessing the communication time I have considered the slowest of the communications that must be serialized.

3.1 Tree of concurrent activities

Each worker $W_{i,j}$ receives $A_{i,j}$ and X_j and computes $A_{i,j} * X_j^{(k)}$

Performance model In order to save some Workers, I map the computations onto the array $[W_{1,1}, ..., W_{n,n}]$.

- **Number of Workers:** n^2
- **Height(tree):** $O(\log n)$. Indeed the tree is made of n, independent subtrees, without cross edges between different subtrees.
- Each level of the tree can be computed in parallel

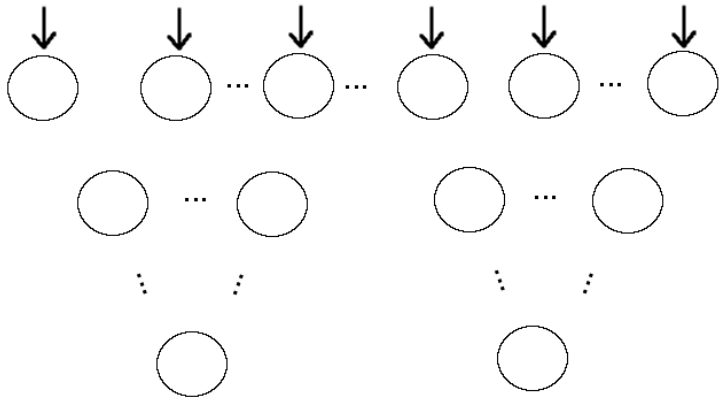

Figure 2: Decomposition strategy number 1: tree of workers

- **Calculation:** $T_{calc} = O(1)T_{aritm}$. Indeed the single Worker does a constant number of arithmetic operations.
- **Overhead:** $O(n^2)\ T_{setup} + O(\log n)\ T_{comm}$. Indeed we have to setup $O(n^2)$ concurrent activities and the communications that are seriaized are proportional to the height of a tree with n leaves. The communication cost is given by the path in the subtree (from one leave to the root) that is slowest to be computed.

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 $i \in 1, 2, ..., n$.

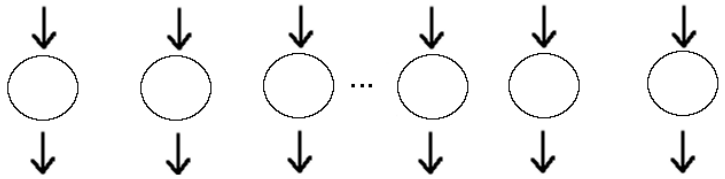

Figure 3: Decomposition strategy number 2: array of workers

Performance model

- **Number of Workers:** n
- **Calculation:** $T_{calc} = O(n)\ T_{aritm}$. Indeed the single Worker makes approximately $O(n)$ sums and $O(n)$ multiplications.
- **Overhead:** $O(n)\ T_{setup} + O(1)\ T_{comm}$. Indeed we have to setup $O(n)$ concurrent activities and the communications of the n Workers can go in parallel.

3.3 Array of concurrent activities with pipeline

Since the computation of the worker now 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 It is very likely that this parallelization is not worth it, because stage one takes $O(n)T_{aritm}$ while stage 2 takes $O(1)\,T_{aritm}$.

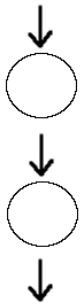

Figure 4: Decomposition strategy number 2, variant with pipeline: only one worker is considered

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

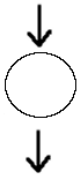

Figure 5: Decomposition strategy number 3: single worker

Performance model

- **Number of Workers:** 1
- **Calculation:** $T_{calc} = O(n^2)\,T_{aritm}$.
- **Overhead:** $O(1)\,T_{setup} + O(1)\,T_{comm}$, if we instantiate the concurrent activity on a processing element different from the one running the main flow of control. If instead we instantiate the worker on the very same processing element, then the overhead is zero.

4 Implementation

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.

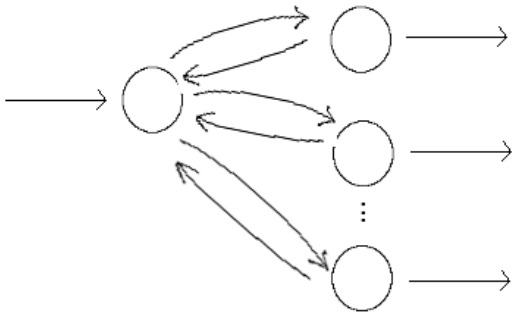

Figure 6: Structure of the implemented decomposition