Università di Pisa

Department of Computer Science

SPM Final Project Report

The Jacobi Iterative Method

Matteo Busi

Student ID. 494087

June 10, 2017

1 Introduction

The aim of this project was to produce a program to solve linear systems using the Jacobi method.

Three different implementations are proposed here:

Sequential implementation the sequential implementation provides a vanilla implementation of the Jacobi method,

Thread implementation is a naïve implementation of the algorithm using `threads` from `C++11` standard,

FastFlow implementation is an implementation using the `parallelFor` from FastFlow library.

Tests were conducted on a machine using a *Intel Xeon E2650* CPU (8 cores clocked at 2 GHz each with 2 contexts) and a *Intel Xeon Phi* co-processor (60 cores clocked at 1 GHz each with 4 contexts).

Summary. The next section discusses the details of program design, including an analysis of the expected performance of the sequential and parallel implementations. Section 3 reports some details about the implementation, discussing the classes design, their methods, and some optimization. Section 4 is divided in two sub-sections. The first sub-section discusses the methodology for the experiments and chosen parameters, while the second sub-section reports the experimental results in the form of tables and graphs. Section 5 includes the user manual for the program, and indications on how to reproduce results reported here. Finally Section **??** compares obtained results against the expected ones.

2 Design

In this section the basic Jacobi algorithm is introduced. Then a brief account on design choices, driven by the performance model, is given along with a sketch of the parallel algorithm.

2.1 Sequential algorithm and performance

Figure 1 shows the pseudo-code for the Jacobi iterative method as presented in the numerical computing literature [].

It is pretty evident that the completion time T_C for a program using this procedure could be computed as:

$$T_C = T_{alloc}(n) + T_{fill}(n) + T_{jacobi}(n)$$

```
Data: A linear system in the form of $Ax = b$ with $N \times N$ the size of
      $A$, a maximum accepted error $\varepsilon$, and a maximum number of
      iterations $K$.
Result: An approximated solution $x^{(k)}$ s.t. $||b - Ax^{(k)}||_2^2 \leq \varepsilon$ or
        $k \geq K$.
$x^{(0)} \leftarrow$ initial guess for the solution
$k \leftarrow 0$
while $||b - Ax^{(k)}||_2^2 \geq \varepsilon$ and $k \leq K$ do
    for $i \leftarrow 0$ to $N$ do
        $\sigma \leftarrow 0$
        for $j \leftarrow 0$ to $N$ do
            if $j \neq i$ then
                $\sigma \leftarrow \sigma + a_{ij} x_j^{(k)}$
            end
        end
        $x_i^{(k+1)} \leftarrow \frac{b_i - \sigma}{a_{ii}}$
    end
    $k \leftarrow k + 1$
end
```

Figure 1: Pseudo-code for the sequential Jacobi iterative method.

where $T_{alloc}(n)$ and $T_{fill}(n)$ are, respectively, the time needed do allocate and fill the memory to store the input data, and $T_{jacobi}(n)$ is the time to solve the system using the algorithm in Figure 1 (i.e. the latency).

Further expanding T_{jacobi} we get:

$$T_{jacobi}(n) = k \cdot (T_{conv}(n) + T_{comp}(n) + T_{upd}(n))$$

where $T_{conv}(n)$ is the time needed to check convergence, $T_{comp}(n)$ is the time needed to compute the new approximation of the solution, and $T_{upd}(n)$ is the time needed to update the solution vector.

Basic complexity theory allow us to conclude that $T_{conv}(n)$ and $T_{upd}(n)$ have linear complexity and $T_{comp}(n)$'s complexity is quadratic. It is worth, then, to parallelize the computation corresponding to time $T_{comp}(n)$ (i.e. the computation of the new value of x).

2.2 Parallel algorithm and performance

Observations of the previous sub-section let us to design a worker. The idea is to split the matrix A in rows of the same size and to feed said rows to the worker.

The performance model depends also on the number of workers, w, and $T_{jacobi}(n, w)$ can be expanded — ignoring $T_{conv}(n)$ and $T_{upd}(n)$ — as:

$$T_{jacobi}(n, w) = T_{setup}(n, w) + T_{barrier}(n, w) + \frac{k}{w} \cdot T_{comp}(n)$$

where $T_{setup}(n, w)$ is the time required to setup the w workers, $T_{barrier}(n, w)$ is the total time spent by threads waiting each other and $\frac{k}{w} \cdot T_{comp}(n)$ is the ideal time needed to w workers to compute the new value of the approximation of the solution.

3 Implementation

As already told in Section 1 the actual implementation of the project consists in three different variants of the Jacobi iterative method, following the ideas in Section 2. The first variant is the sequential one, a direct rewriting of the algorithm in Figure 1, the second variant implements the algorithm sketched in sub-section 2.2 using the `ParallelFor` provided by `FastFlow` library, and the third variant implements the same parallel algorithm using `C++11` threads. More precisely:

- The main function in file `main.cpp`, that implements the parameter parsing, I/O and initialization.
- The class `JacobiReport` that collects statistics about the execution of the algorithm.
- The class `JacobiSolver` that implements, together with class `JacobiSequentialSolver`, `JacobiFFSolver`, and `JacobiThreadSolver`, a template method pattern []. The class provides a method `solve` that, given a linear system, produces a `JacobiReport`.
- Classes `JacobiSequentialSolver`, `JacobiFFSolver`, and `JacobiThreadSolver` implement said algorithms by redefining the method `deltax` that computes the new approximation of the solution.

During the development we paid particular attention to vectorization, ensuring that the compiler could vectorize most of the code, and to minimization of overhead in method `deltax` trying to remove unneeded memory allocations and initializations (with the exception of `JacobiThreadSolver::deltax` which has been kept as simple as possible).

4 Experiments

This section summarizes how the experiments were conducted and their results.

4.1 Methodology

Experiments were performed using the script `jacobirun.sh`, as better explained in sub-section 5.4. Specifically each variant of the program (i.e. sequential, thread and FastFlow versions) were run both on Xeon CPU (with number of workers between 1 and 16) and Xeon Phi co-processor (with number of workers between 1 and 240) for four different system sizes N, i.e. 5000, 10000, 15000, and 30000. Bigger Ns filled up the memory of the co-processor, hence were not included in the analysis. All the tests of the `FastFlow` implementation use a grain size of 10.

4.2 Results and remarks

Figure **??** and Figure **??**, available in bigger format in the appendix, report the experimental results of the two parallel implementations. In particular Figure **??** reports the results of the `ParallelFor` implementation: on the left the results refer to an execution on the Xeon CPU while on the right the results are for the Xeon Phi. While Figure **??** shows the results of the implementation using `C++11` threads both for the Xeon CPU and the Xeon Phi co-processor. inserire qui i grafici

In both parallel implementations it is evident that curves of efficiency, speedup, and scalability are less than optimal — expected performances have been outlined in Section 2.

The explanation resides in the following facts (extracted from the measures produced for `C++11` implementation):

- The time spent in setting up the workers and in the barrier are not negligible, as assumed above.
- Smaller system size N leads to a bigger fraction of time spent in setting up threads.
- The same happens for bigger number of workers, as the fraction of time spent in setting up and orchestrating threads increases w.r.t. to the fraction of time spent in computation.
- Moreover small workloads in combination with *vectorization* — which effectively improves the computation time increasing — increase, again, the setup/barrier time fraction.

The above affirmations are supported by Table **??**, which shows results regarding the best configuration in terms of latency for each size N, processor, and implementation. For each "best configuration" of `C++11` thread implementation an extra column shows the ratio between setup/barrier time and computing time (i.e. $\frac{T_{barrier}+T_{setup}}{T_{comp}}$) (full data is available in folder `results` as `csv` file, as explained in Section 5).

Finally can be observed that:

- None of the best configurations are for very small or very big number of workers.
- Increasing the N leads to better results, but huge systems may not perform well (i.e. $N > 30000$) since cache coherence procedures may spend non negligible time in moving data (i.e. incrementing the time for updating).
- The extra columns shows that the ration of time spent in setup/barrier decreases as N increases.

5 User guide

This section provides a short guide on how to use the program, how to conduct the experiments, and how to gather the results.

5.1 Workspace

The workspace content is organized as follows:

- The folder `bin` contains the results of the compilation (including vectorization reports),
- the folder `graphs` contains the graphs generated by `reportgen.py`,
- the folder `results` contains the collection of `csv` files generated by `jacobirun.sh`,
- the folder `src` contains the source code of the program,
- the bash script `jacobirun.sh` contains the code to run experiments,
- the Python program `reportgen.py` that generates graphs starting from data in `results` folder,
- the make file `Makefile` compiles the project as explained in subsection 5.2.

In the following we assume that the current working directory is the root of the workspace.

5.2 Compilation

To compile the project a `Makefile` with four rules is provided:

1. Executing `make jacobix` the executable for the Xeon CPU is produced and placed in `bin/jacobix`,

2. executing `make jacobim` the executable for the Xeon Phi is produced and placed in `bin/jacobim`,

3. executing `make offload` the executable for the Xeon Phi is produced and placed in both `bin/jacobim` and in the home directory on `mic1`,

4. executing `make clean` the files produced by compilation, testing, and analysis are deleted.

5.3 Program usage

To run a single resolution of a random system one of the compiled executables located in `bin` must be run. Executable `jacobix` runs on the Xeon CPU, while executable `jacobim` must be offloaded to the Xeon Phi.

Executing one of the executable without arguments produces as output a guide that should be self-explaining:

```
Usage: bin/jacobix N ITER ERR METHOD [NWORKERS] [GRAIN]
Where:
N : is the size of the matrix A
ITER : is the maximum number of iterations
ERR : is the maximum norm of an acceptable error
METHOD: is either
    s : indicating that the sequential implementation
       must be used
    f : indicating that the FastFlow implementation must
       be used
    t : indicating that the Thread implementation must be
        used
NWORKERS : the number of workers that should be used (
   ignored if METHOD is 's')
GRAIN : the grain of the computation (only if METHOD is '
   f')

Produces a CSV line, in the form:
    N_WORKERS, N_ITERATIONS, COMP_TIME, UPD_TIME,
       CONV_TIME, SETUP_AND_BARRIER_TIME, LATENCY, ERROR
Where:
N_WORKERS : is the number of workers used
N_ITERATIONS : is the effective number of iterations
   performed
COMP_TIME : is the total time spent in computation of the
    new approximation
UPD_TIME : is the total time spent in updating the
   solution vector
CONV_TIME : is the total time spent in convergence checks
SETUP_AND_BARRIER_TIME : is the total time spent setting
   up threads and in barrier (computed only if METHOD is
    't')
```

```
LATENCY : is the latency
ERROR : is norm of the error
```

5.4 Experiments and analysis

After compilation, to execute the experiments and analyse the results one must:

1. run `./jacobirun.sh` or `./jacobirun.sh MIC` (if Xeon Phi must be used),
2. run `reportgen.py` to produce graphs.