Università di Pisa

Department of Computer Science

SPM Final Project Report

The Jacobi Iterative Method

Matteo Busi

Student ID. 494087

June 11, 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 sequential implementation of the Jacobi method,

Thread implementation is an implementation of the algorithm using `threads` from C++11,

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). For each test (i.e. for each different implementation and for different linear system sizes) we collected the latency and for parallel versions also computed speedup, the scalability, the efficiency, and the "best configuration" to minimize latency. We also produced some graphs to compare expected results against empirical ones.

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 6 includes the user manual for the program, and indications on how to reproduce results reported here. Finally Section 5 compares obtained results against the expected ones.

2 Design

Here we first introduce the sequential version of the Jacobi iterative algorithm. We then explain how to adapt the algorithm for vectorization and parallelization. We also provide an performance model, to justify parallelization and design choices.

2.1 Sequential algorithm and performance

In the numerical computing literature the Jacobi iterative method is presented as particular iterative method and the corresponding pseudo-code is usually similar to the one in Algorithm 1.

Supposing that a program implementing the algorithm takes $T_{jacobi}(n)$ time to compute the result (i.e. the latency), then the completion time T_C of an hypothetical program is

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

where $T_{alloc}(n)$ and $T_{fill}(n)$ are, respectively, the time needed do allocate and fill the memory to store the input data.

Further expanding T_{jacobi} we get:

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

where k is the number of iterations, $T_{conv}(n)$ 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.

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

Data: A linear system in the form of $Ax = b$ with $N \times N$ the size of A, a maximum accepted error ε, 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

2.2 Parallel algorithm and performance

Given the performance model of the previous sub-section to better parallelize the algorithm we may consider to minimize those $T_{conv}(n)$, $T_{comp}(n)$, and $T_{upd}(n)$ but:

- $T_{conv}(n)$ and $T_{upd}(n)$ corresponds to computations with linear time complexity
- $T_{comp}(n)$ corresponds to a computation with quadratic time complexity

Hence we will consider only consider $T_{conv}(n)$ and $T_{upd}(n)$ as negligible, and proceed by parallelizing only of the last computation (always assuring that vectorization of each of the three computations takes place, see Section 3).

More specifically given the system matrix A with size $N \times N$ and having w workers:

1. First, an emitter should divide A into sub-matrices each with about N/w rows, delivering it to each worker,
2. then, each worker should compute its part of the solution using the provided sub-matrix as described in the pseudo-code of Algorithm 2,
3. finally a collector should recompose the solution

Algorithm 2: Pseudo-code for the worker

Data: A sub-matrix A_s of A with M rows and N columns, the previous approximation $x^{(k)}$.
Result: The next approximation of the solution.

```
for i ← 0 to M do
    σ ← 0
    for j ← 0 to N do
        if j ≠ i then
            σ ← σ + a_ij x_j^(k)
        end
    end
    x_i^(k+1) ← (b_i − σ) / a_ii
end
```

As evident from Algorithm 2 the performance model now also depends on w, so the latency, $T_{jacobi}(n, w)$, can be expanded — ignoring $T_{conv}(n)$ and $T_{upd}(n)$ — as:

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

where $T_{setup}(n, w)$ is the time required to setup the w workers and assign them the work (i.e. the time needed to the emitter), $T_{barrier}(n, w)$ is the total time spent by the collector 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

The actual implementation consists in three different variants of the Jacobi iterative method, following the ideas in Section 2.

The source code is organized as follows:

- The main function, in file `main.cpp`, implements the parameter parsing, I/O and initialization.
- The class `JacobiReport` collects statistics about the execution of the algorithm.
- The class `JacobiSolver` that implements, together with class `JacobiSequentialSolver`, `JacobiFFSolver`, and `JacobiThreadSolver` a template method pattern. Each of the concrete sub-classes implement a `deltax` method specifying how the new approximation of the solution x to the system is computed.
- Class `JacobiSequentialSolver` implements `deltax` exactly as in Algorithm 1
- Class `JacobiFFSolver` implements `deltax` using the `parallel_for` building block from FastFlow library; `parallel_for` automatically parallelizes the given loop.
- Class `JacobiThreadSolver` implements `deltax` using C++ threads. The implementation is naïve, meaning that the threads are re-created at each method invocation (i.e. T_{setup} may become non negligible) and the collector is a simple barrier implemented using `join` method from `thread` class.

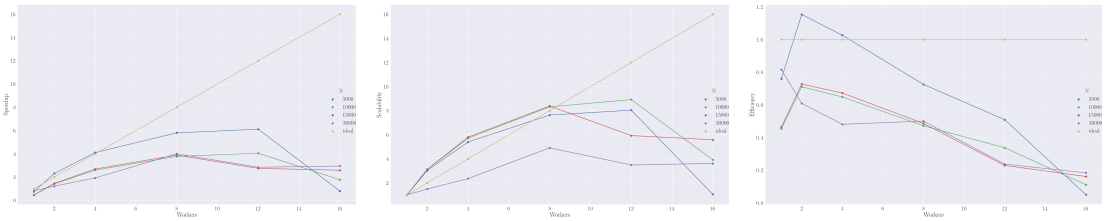
Please note that, the code of each `deltax` (and also other methods) have been made vectorizable using the feedback provided by the compiler. Specifically the `if` construct present in the inner-loop in the algorithms above have been removed and the loop have been split in two parts, i.e. the first part in range $[0, i)$ and the second $[i + 1, N)$ (or $[i + 1, M)$ in case of parallel implementation).

4 Experiments

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

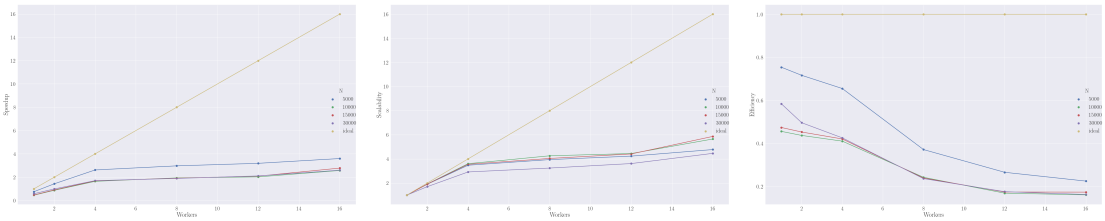
4.1 Methodology

Experiments are performed using the script `jacobirun.sh`, as better explained in subsection 6.4. The script runs sequential, thread and FastFlow versions both on the Xeon CPU and Xeon Phi co-processor. If on the Xeon CPU The number of workers for parallel versions goes from 1 to 16, otherwise it ranges from 1 to 240 (to match the number of threads of each processor). N, instead, assumes values of 5000, 10000, 15000, and 30000.



(a) Speedup graph. (b) Scalability graph (c) Efficiency graph

Figure 1: Graphs of different performance measures for $N \in \{5000, 10000, 15000, 30000\}$ on the Xeon CPU using FastFlow



(a) Speedup graph. (b) Scalability graph (c) Efficiency graph

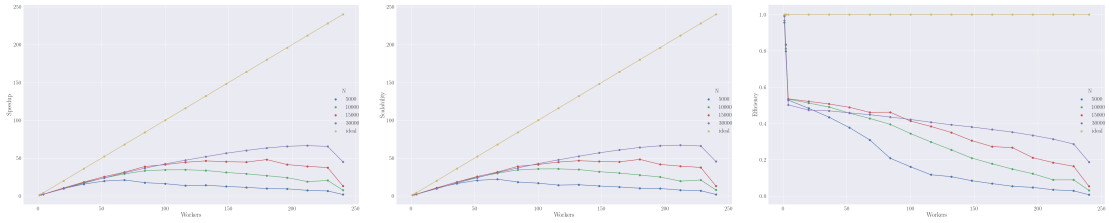
Figure 2: Graphs of different performance measures for $N \in \{5000, 10000, 15000, 30000\}$ on the Xeon CPU using C++11 threads

Sadly bigger values of N quickly filled up the memory of the co-processor, hence they were not included in the analysis. All the tests of the FastFlow implementation use a grain size of 10.

4.2 Results and remarks

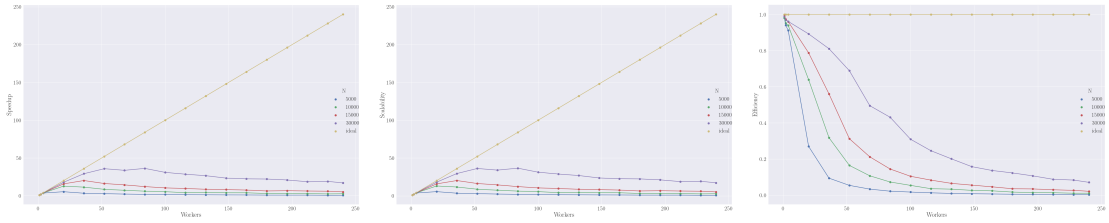
Each of the graphs in this section reports experimental results in terms of efficiency, scalability, and speedup of each experiment described above. Figure 1 and Figure 2 respectively report the results of the execution of the FastFlow and C++11 thread implementation on the Xeon CPU. It is interesting to see that none of the measured metrics adhere to the ideal curve, especially for bigger Ns. Smaller Ns lead to better performance metrics, probably this is partly due to the time spent in barrier and in the setup of the workers and partly due to the time spent by the processor in performing cache coherence.

Figure 3 and Figure 4 respectively report the results of the execution of the FastFlow and C++11 thread implementation on the Xeon Phi co-processor. As above none of the measured metrics adhere to the ideal curve. In this case, on the other hand, bigger Ns lead to better performance metrics since the time spent in barrier and in the setup of the workers becomes smaller w.r.t. effective time of computation. In this case (up to $N = 30000$) cache coherence probably is not a bottleneck, since the cache on the Xeon



(a) Speedup graph. (b) Scalability graph (c) Efficiency graph

Figure 3: Graphs of different performance measures for $N \in \{5000, 10000, 15000, 30000\}$ on the Xeon Phi co-processor using FastFlow



(a) Speedup graph. (b) Scalability graph (c) Efficiency graph

Figure 4: Graphs of different performance measures for $N \in \{5000, 10000, 15000, 30000\}$ on the Xeon Phi co-processor using C++11 threads

Item		
Animal	Description	Price ($)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99

Table 1: Best configurations in terms of minimal latency for FastFlow implementation at different values of N

Item		
Animal	Description	Price ($)
Gnat	per gram	13.65
	each	0.01
Gnu	stuffed	92.50
Emu	stuffed	33.33
Armadillo	frozen	8.99

Table 2: Best configurations in terms of minimal latency for C++11 thread implementation at different values of N

Phi is bigger than the one on the Xeon CPU. Table 1 shows, for each size N of the system, the best configuration in terms of latency for the FastFlow implementation. Table 2 shows the same data as Table 1, but reports also the ratio between setup/barrier time and computing time (i.e. $(T_{barrier} + T_{setup})/T_{comp}$) (full data is available in folder `results` as `csv` file, as explained in Section 6).

Please note that (almost) none of the best configurations include a number of workers equal to the number of contexts, this is due to the fact that usually at least one context is used for orchestration.

5 Conclusions

Summing up the data let us to conclude that:

- The time spent in setting up the workers and in the barrier are not always negligible, especially for bigger Ns

- On the other hand the ratio of time spent in setup increases as w increases with small Ns
- Huge systems may not perform well since cache coherence procedures may spend non negligible time in moving data (i.e. incrementing the time for updating)
- Moreover small workloads in combination with *vectorization* — which dramatically improves the computation time increasing — increase, again, the setup/barrier time fraction.

So, decent performances were achieved but to have better metrics one should, for example:

- Optimize the code to work better on a specific architecture
- removing unnecessary memory allocations or use better allocators
- reduce idle time of each worker by giving a more evenly distributed workload

6 User guide

This section provides a short guide on how the project is organized, how the code can be compiled, and how to use compiled programs.

6.1 Workspace

The project workspace is organized as follows:

- The folder `bin` contains the results of the compilation (including vectorization reports),
- the folder `graphs` contains the full size 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 sub-section 6.2.

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

6.2 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.

6.3 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
```

6.4 Experiments and analysis

After compilation, to execute the experiments and analyse the results to obtain the above graphs and metrics one must:

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