Università di Pisa

Department of Computer Science

SPM Final Project Report

The Jacobi Iterative Method

Matteo Busi

Student ID. 494087

June 12, 2017

1 Introduction

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

Three different variants of the algorithms are provided. The first variant is the straightforward implementation of the sequential algorithm. The second and the third are parallel implementations. One uses C++11 threads to parallelize the algorithm and distribute the work among different threads, the other employs FastFlow and the `parallel_for` construct.

All the experiments 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 experiment (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" that minimizes the latency. We also produced some graphs to compare expected results against empirical ones.

Summary. The next section discusses the details of the 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 first introduces the methodology used in the experiments then reports and discuss the experimental results. Section 5 concludes the report with some final remarks. Finally, a brief user manual is reported in Section 6.

2 Design

In this section we first introduce the sequential version of the Jacobi iterative algorithm, then we 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, 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} and assuming an implementation of Algorithm 1 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 $T_{conv}(n)$, $T_{comp}(n)$, and $T_{upd}(n)$ but:

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

Hence we will 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 with $\approx N/w$ rows each, and deliver each of them to a 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:

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

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 $\frac{T_{comp}(n)}{w}$ 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 command line parameter parsing, I/O and initialization.
- The class `JacobiReport` collects statistics about the execution of the algorithm.
- The class `JacobiSolver` implements, together with class `JacobiSequentialSolver`, `JacobiFFSolver`, and `JacobiThreadSolver` a template method pattern. Each of the concrete sub-classes implements 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 using a grain size provided as argument to the object's constructor.
- Class `JacobiThreadSolver` implements `deltax` using C++11 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 implementations).

4 Experiments

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

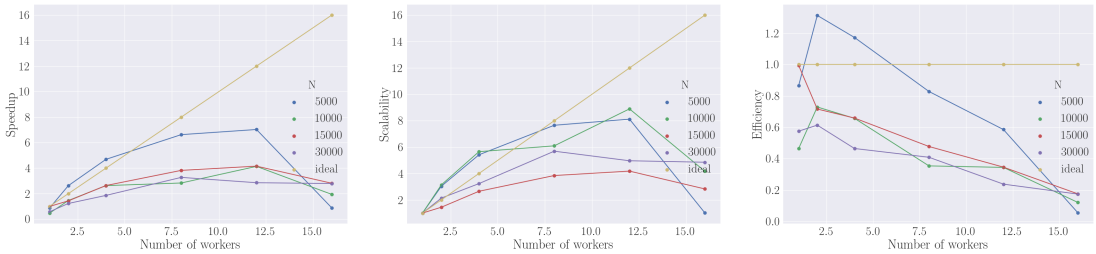
4.1 Methodology

Experiments were performed using the script `jacobirun.sh`, as better explained in subsection 6.4. The script ran sequential, C++11 thread and FastFlow versions both on the Xeon CPU and Xeon Phi co-processor. If executed on the Xeon CPU the number of workers for parallel versions went from 1 to 16, otherwise their number ranged from 1

to 240 (to match the number of contexts of the processor). N, instead, assumed values of 5000, 10000, 15000, and 30000. Sadly bigger values of N filled the memory of the co-processor, hence they were not included in the analysis. All the tests of the FastFlow implementation used a fixed grain size of 10.

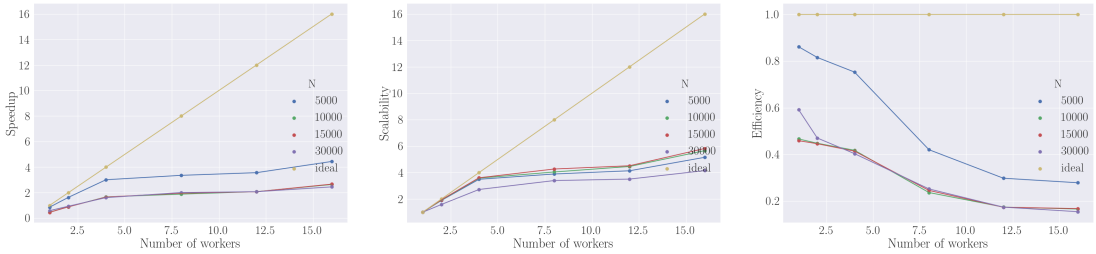
4.2 Results and remarks

Each of the graphs in this section reports experimental results in terms of efficiency, scalability, and speedup for each of the experiments described above.



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

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

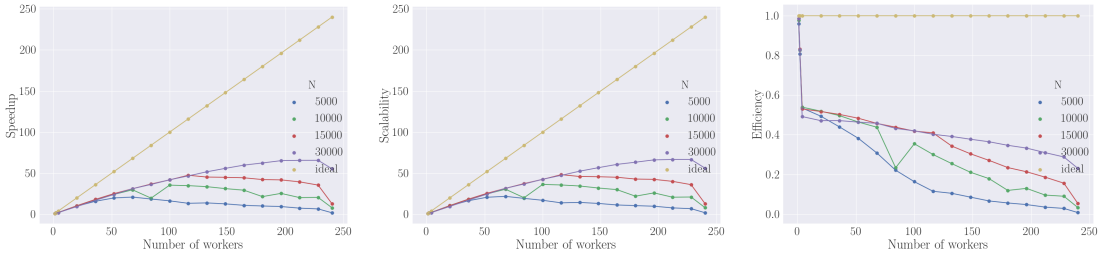


(a) Speedup graph. (b) Scalability graph (c) Efficiency graph

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

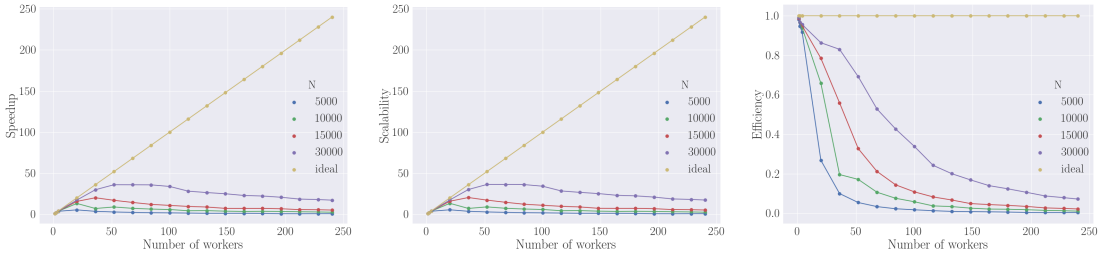
Figure 1 and Figure 2 respectively report the results of the execution of the FastFlow and C++11 thread implementation on the Xeon CPU. It is interesting to see that none of the measured metrics adhere to the ideal curve, especially for bigger values of N.

Smaller values of N had slightly to better w.r.t. bigger values of N, especially as the number of workers increased. This is probably due to the fact non-negligible time is spent in cache coherence procedures for bigger values of N.



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

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



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

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

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 values of N led to better performance metrics since the the time spent in setting up workers and in the barrier became smaller w.r.t. effective time of computation. Also, in this case (up to $N = 30000$) cache coherence probably were not a bottleneck, since the cache size of the Xeon Phi is bigger than the one on the Xeon CPU.

Table 1 and Table 2 show, for each size N of the system, the best configuration in term of minimal latency for the FastFlow and C++11 thread implementations, along with latency for the sequential implementation.

Note also that (most) of the best configurations use a number of workers smaller than the number of contexts, this is due to the fact that usually one context is used for orchestration.

		$N = 5000$		$N = 10000$	
		Xeon CPU	**Xeon Phi**	**Xeon CPU**	**Xeon Phi**
Sequential	L_{seq} (s)	0.055928	1.054620	0.118950	0.782633
FastFlow	L_1 (s)	0.064616	0.209478	0.255854	0.800839
	$L_{w_{best}}$ (s)	0.007958	0.009615	0.028755	0.022076
	w_{best}	12	68	12	100
Thread	L_1 (s)	0.064908	0.205056	0.255311	0.791644
	$L_{w_{best}}$ (s)	0.012583	0.037514	0.045068	0.059418
	w_{best}	16	20	16	20

Table 1: Best configurations in terms of minimal latency for FastFlow and C++11 thread implementations. Here the number of workers that minimizes the latency is denoted by w_{best}, latency with w workers as L_w, sequential latency as L_{seq}. ($N = 5000$ and $N = 10000$)

5 Concluding remarks

Summing up, we can conclude that:

- The ratio between the time spent in setup/barrier and the latency increases as w increases, especially for smaller values of N.
- Even if the experiments seem to imply that the algorithm performs better for bigger values of N, 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 (i.e. small values of N/w) in combination with *vectorization* — which dramatically improves the computation time — do not play well together since the the setup/barrier time fraction increases.

So, decent performances as Table 1 and Table 2 were achieved. To have a better programs in term of measured metrics one could, for example:

- Optimize the code to work even better on a specific architecture,
- removing unnecessary memory allocations (e.g. by re-using workers) or use better allocators,
- reduce idle time of workers by giving a more evenly distributed workload.

		$N = 15000$		$N = 30000$	
		Xeon CPU	**Xeon Phi**	**Xeon CPU**	**Xeon Phi**
Sequential	L_{seq} (s)	0.261870	1.716810	1.054620	6.797150
FastFlow	L_1 (s)	0.574178	1.741540	1.325130	6.887530
	$L_{w_{best}}$ (s)	0.063033	0.036150	0.322207	0.103486
	w_{best}	12	116	8	228
Thread	L_1 (s)	0.571200	1.740340	1.783050	6.861650
	$L_{w_{best}}$ (s)	0.098026	0.085337	0.429399	0.189059
	w_{best}	16	36	16	52

Table 2: Best configurations in terms of minimal latency for FastFlow and C++11 thread implementations. Here the number of workers that minimizes the latency is denoted by w_{best}, latency with w workers as L_w, sequential latency as L_{seq}. ($N = 15000$ and $N = 30000$)

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 executables 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, UPD_TIME, CONV_TIME, LATENCY, ERROR
Where:
    N_WORKERS : is the number of workers used
```

```
N_ITERATIONS : is the effective number of iterations
   performed
UPD_TIME : is the total time spent in updating the solution
   vector
CONV_TIME : is the total time spent in convergence checks
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 and tables.