Master degree in Computer Science
University of Pisa

Parallel and Distributed Systems: Paradigms and Models

A.Y. 2017/2018

Genetic Algorithm

Maria Chiara Cecconi
528025

Repository: https://github.com/MCC04/SPM_Proj_Cecconi

1. Introduction

The aim of this project was to implement a Genetic Algorithm. Two different approaches have been followed:

- sequential approach
- parallel approach, the algorithm has been parallelized using:
  - *C++ threads*,
  - *FastFlow* framework (using the parallel for pattern)

All the experiments have been executed on an *Intel Xeon PHI Knights Landing (KNL)* 64 cores (4 contexts per core).
For each experiment (i.e. for each approach, listed above, and for different population sizes) the completion time have been collected. Furthermore, for parallel versions, speedup, scalability and efficiency have been computed.

The following section discusses details of the algorithm, including the expected performance both for sequential and parallel implementations.
Section 3 discusses details about the project structure and the implementation, listing the classes design and main features.
Section 4 reports and briefly discuss the experimental results.
Section 5 concludes with some final remarks.
Appendix A gives a user manual and some additional informations.

2. Algorithm design

In this section it will be introduced the sequential version of the Genetic algorithm (pseudo-code) with an expected computational time.
Then, we will also describe the parallel model; also, in this case a pseudo-code has been listed, the one that has to be executed for each concurrent activity.

2.1 Sequential implementation

The Genetic Algorithm simulates the evolution of a population of trees, the goal is to find the best tree that approximates a given function, below the pseudocode of the algorithm.

Alg.1: *Pseudo-code for sequential Genetic Algorithm*

```
Input: population, points, tolerance, maxIterations

1. fitnessValues = BuildAllFitness(population, points)
2. while (i < maxIterations)
3.     fitnessValues = UpdatelFitness(population, points);

4.     if (bestFitness<=tolerance) break;

5.     updatedTreesIndexes = nextGeneration( fitnessValues);
   end

Output: newPopulation, newFitnesses
```

Suppose that the computation of the Genetics takes $T_{genetics}(n)$, then $T_{data}(n)$ is the time needed to allocate and process input data (e.g. generate a random population given the tree number), the completion time (or latency) will be:

$$T_c \approx T_{data}(n) + T_{Genetics}(n)$$

We'll focus on the time to compute the result $T_{genetics}(n)$, obviously this involves other times in itself:

$$T_{Genetics}(n) \approx m \cdot T_{fit}(n) + k\,(r \cdot T_{fit}(n) + T_{next}(n)))$$

Assuming *k* as the maximum iteration number of the algorithm, here we have:

- $m \cdot T_{fit}(n)$ is the time given by row 1 in *Alg.1,* where *m* is the size of population (number of trees),
- $r \cdot T_{fit}(n)$ is the row 3 in *Alg.1,* where *r* is the size of the only fitness that need to be updated,
- $T_{next}(n)$ is the row 5 in *Alg. 1,*

This is also the completion time, used to analyze the performances in Section 4 (the time needed to set up data and inputs was not considered).

2.2 Parallel approach

Our goal is to minimize $T_{genetics}(n)$ by parallelizing the algorithm described in the previous section.
For parallel approach only the part of fitness computation can be considered.
That's because of the strong dependencies between the parts forming `nextGeneration` task (i.e. Genetic Operators depend on Selection Phase, that in turn depends on Weights computation).
On the other hand the Fitness computations are independent between them (i.e. every fitness is computed on a different tree).
So the parallelization will be achieved by splitting the load between workers, more equally possible (`#fitnessToUpdate / #workers`).
So, essentially, we based our parallel implementations on a *Map* (*Parallel For*) pattern.

Alg.2*: pseudo-code for single worker*

```
Input: population, points, start, end

1. while ( i = start : end )
2.     sum = SumOfDifferences(population [i], points);

3.     if (sum is valid number)
4.             n = sqrt (sum)
5.     else
6.             n = symbolicBigValue
7.     <fitnessValues.push( n )> or UpdateFitnessValue(n)
    end

Output: fitnessValues
```

In this case, the performance model must take into account of other factors:

$$T_{Genetics}(n, w) \approx m \cdot (\frac{T_{fit}(n,w)}{w} + T_{pool}(n, w)) + k \cdot (r \cdot \frac{T_{fit}(n,w)}{w} + T_{pool}(n, w) + T_{next}(n))$$

New factors we're considering are:

- number of workers w,
- $T_{pool}(n, w)$ the time required to setup the w workers and to assign them the work, wait for all tasks completion.

$\frac{T_{fit}(n,w)}{w}$: is the ideal time needed by w workers to compute the Fitness values.

3. Implementation

The implementation consists in three variants of the Genetics Algorithm (sequential, threads, FastFlow), following the ideas explained in *Section 2*.
The source code is organized as follows:

- main function, in file `main.cpp`, parses the command line input parameters and initialization of data (i.e. create a random population, read points set from text file, and so on),
- class `Node`, represent a binary or unary operation, or a leaf,
- class `Tree`, works as a "list" of nodes and allows operations on trees,
- class `Genetic`, implements the sequential version of the algorithm
- class `ThreadGenetic`, reimplement Fitness computation and contains a simple Threadpool,
- class `FFGenetic`, use some function from Genetic and re-implement the Fitness computation.

3.1 C++ threads implementation

Class `ThreadGenetic` implements C++ thread version for fitness computation, this is done by splitting the loop on trees as said in *Section 2.2*.
In particular, this is almost a *Map* pattern, but submitting each task to a *Threadpool*. So we have a queue of tasks, from which workers will pop and execute them; before going on, to `nextGeneration`, main thread will wait for completion of all tasks.
The choice to use a threadpool was given by the fact that, create *w* workers at each iteration, would have introduced non-negligible overhead.

Code 1. *Part (of C++ Threads code) where we submit tasks, in this case are fitness updates*

```
int chunk=indexes.size() / nExecutors;
int rem=indexes.size()%nExecutors;
for(int k=0 ; k<indexes.size(); k+=chunk){
auto w = std::bind(&ThreadGenetic::updater,this, k, k+chunk);
submit(w);
}
if(rem>0){
auto w = std::bind(&ThreadGenetic::updater,this, indexes.size()-rem, indexes.size()-1);
submit(w);
}
waitFinished();
```

3.1.1 Threadpool implementation

The queue management is positioned in the ThreadGenetic constructor.
We in that class three fundamental functions:

1. `ThreadGenetic::waitFinished()` waits for work completion, that is computation or update of all needed fitness values,
2. `ThreadGenetic::submit(std::function<void()> action)` pushes a task in the tasks queue,
3. `ThreadGenetic::joinAll()` when algorithm finishes we join all workers.

3.2 Fastflow implementation

Class `FFGenetic` implements Fitness computation using the *ParallelFor* pattern from FastFlow library.
More precisely it has been used the following method:
`parallel_for (first, last, step, grain, bodyfunction, nworkers).`
The body function is a C++ lambda-function which implements the loop computing function on all given x points, subtracting them from given y values and summing up all (this is hidden in `computeFit(i)`).

`Parallel_for` automatically parallelizes the given loop upon trees for which we have to compute Fitness.

Code 2. *Parallel For using FastFlow to do fitness updates*

```
pf->parallel_for(0, this->indexes.size(), 1,0,
       [&](const long i){
       double sum=this->computeFit(i);
       double n=0.0;
       if(sum!=DBL_MAX)
              n=sqrt(sum);
       else
              n=DBL_MAX;

       fitValues[indexes[i]].second= n;
}, this->nw);
```

4. Analysis and results

Experiments were performed using the scripts int test folder. Those runs the sequential, C++ threads and FastFlow version both on the *Xeon KNL*.

The range of the number of threads, for parallel versions, goes

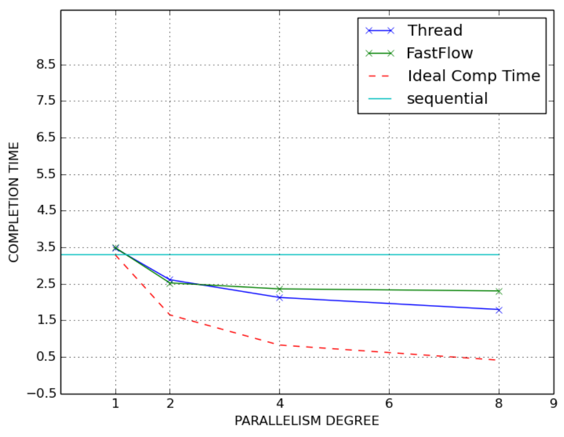
- from 1 to 16 for bigger sets of trees and/or points,
- from 1 to 8 for smaller sets of trees and points

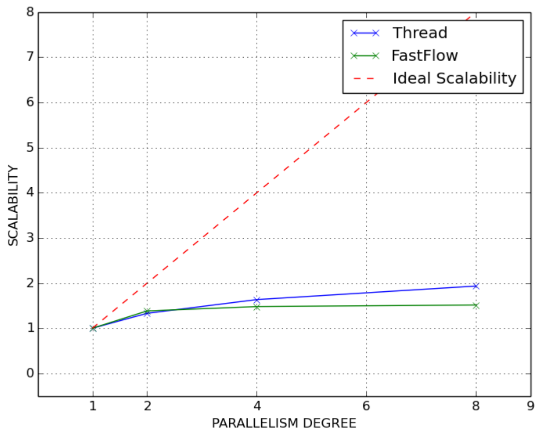
The tests were executed on different sizes of population:

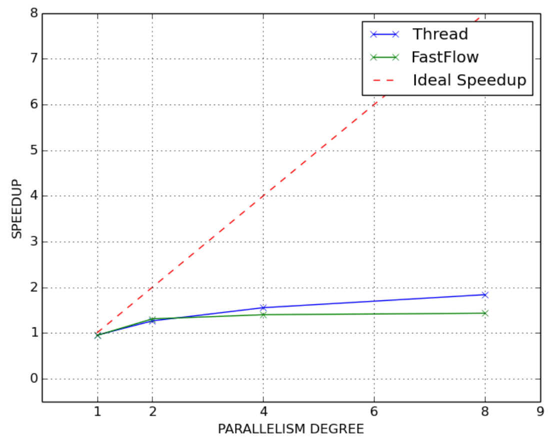
- 200 trees and 100 points
- 400 trees, with both 100 and 200 points.

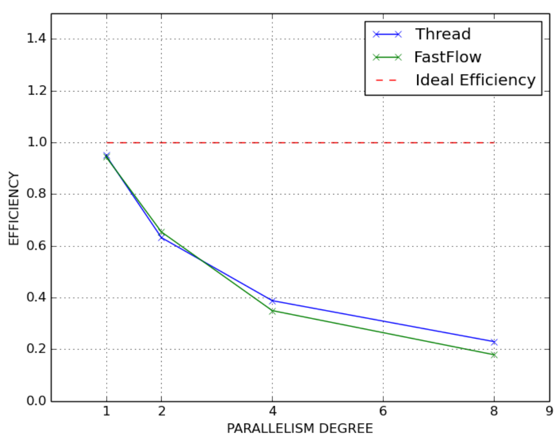
Each group reports experimental results in terms of completion time, efficiency, scalability, and speedup. Tests have been repeated seven times for each dataset size and, in parallel versions, seven times for each number of threads in range.
The completion time has been obtained by making the average on the measured values, previously eliminating outlayers values collected.
Figures below show a comparison between FastFlow and C++11 threads implementation on Xeon KNL.

Graph 1. *200 trees, 100 points, 8 workers*

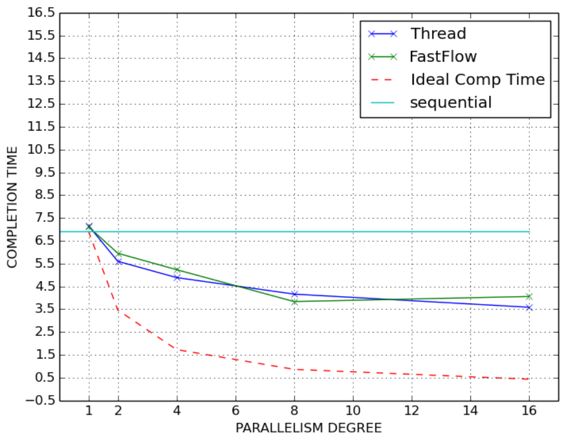


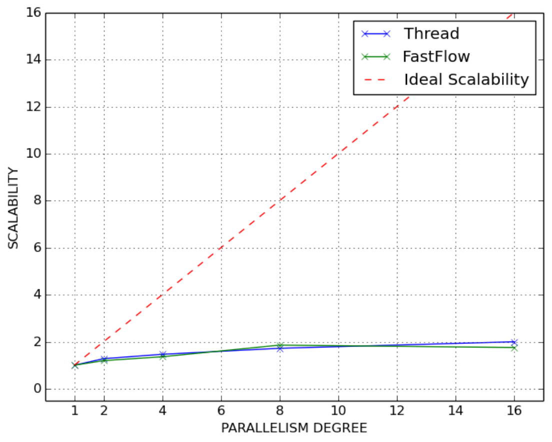


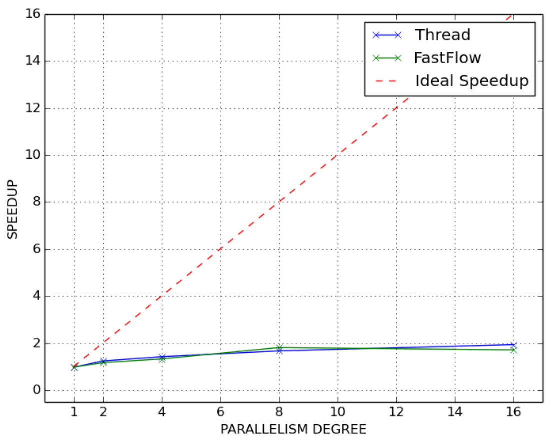


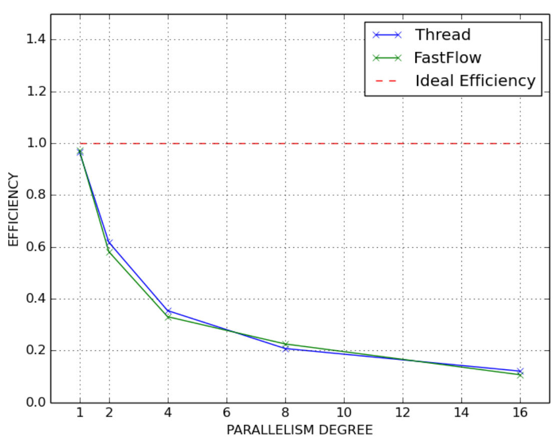


Graph 2. *400 trees, 100 points, 16 workers*

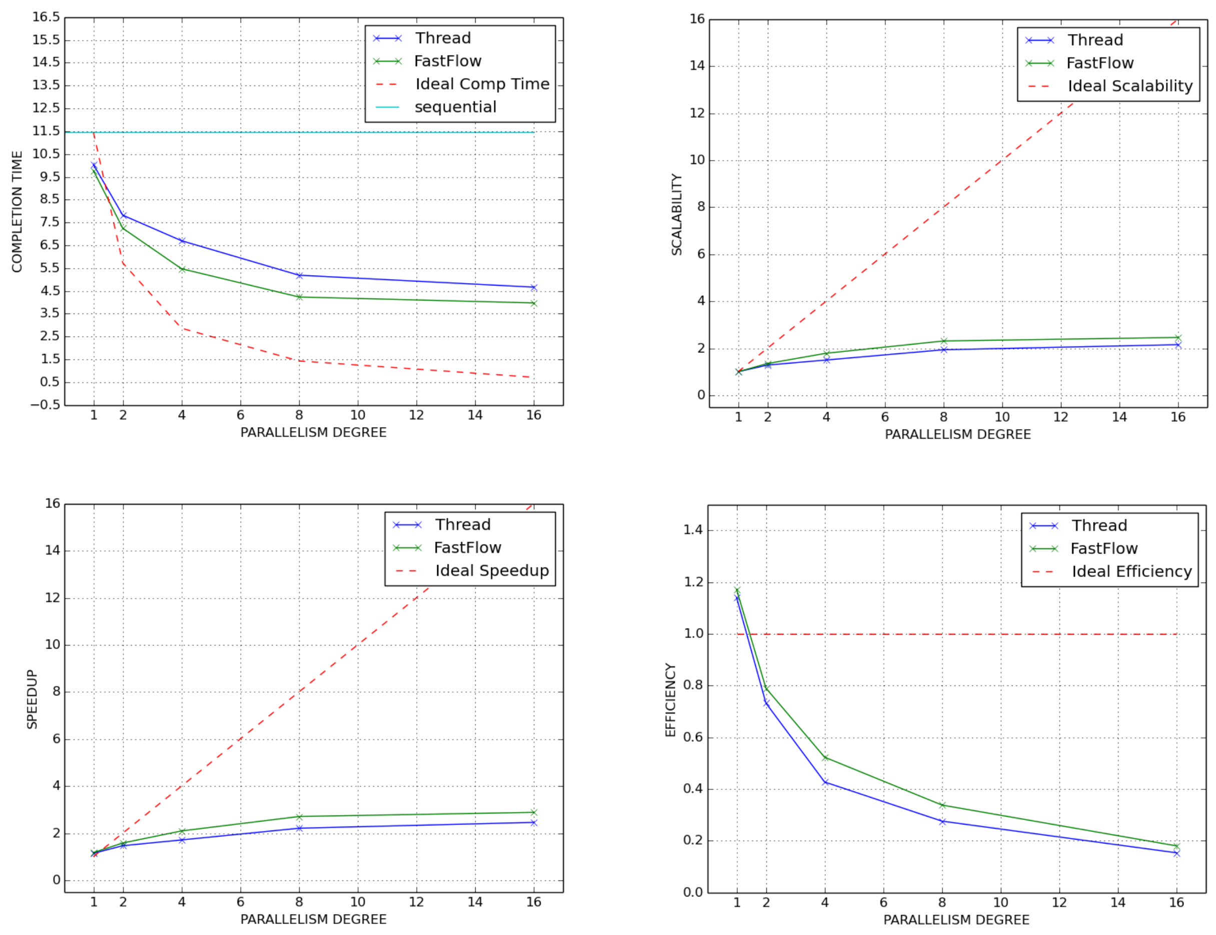








Graph 3. *400 trees, 200 points, 16 workers*



It is interesting to see that none of the measured metrics adhere to the ideal curve.
Furthermore for more trees in input we can see that behavior doesn't significantly changes (compare *Graph 1* and *Graph 2* to 8 workers), while for the same number of trees (400) and different number of points we can observe a slightly different behavior. In fact the number of points affects strictly the fitness computation, so it influence directly the workload of a single worker.
Another interesting fact is that in *Graph 3* we can observe an efficiency > 1, in the case of 1 worker. As expected speedup is over the ideal value for 1 worker too, this means that here we have a superscalar speedup; maybe this is due to the fact that, in sequential execution, data don't fit in cache while, with 1 worker, the cache hit is bigger than the overhead to setup parallel activities. Indeed overhead grows as the workers number grows, making disappear the mentioned behavior.

Table 1. *All times and measures for 200 trees, 100 points, 8 workers*

	Workers	***1***	***2***	***4***	***8***
Thread	***Completion***	3.4766	2.6126	2.1272	1.7974
	Scalability	1.0	1.3307	1.6343	1.9342
	SpeedUp	0.9498	1.2639	1.5523	1.8371
	Efficiency	0.9498	0.6319	0.3880	0.2296
FastFlow	***Completion***	3.4946	2.5251	2.3614	2.3067
	Scalability	1.0	1.3839	1.4798	1.5149
	SpeedUp	0.9449	1.3077	1.3983	1.4315
	Efficiency	0.9449	0.6538	0.3495	0.1789
	Ideal Comp	3.3021	1.6510	0.8255	0.4127
	Sequential	3.3021			

Table 2. *All times and measures for 400 trees, 100 points, 16 workers*

	Workers	***1***	***2***	***4***	***8***	***16***
Thread	***Completion***	7.1652	5.6010	4.8871	4.1625	3.5865
	Scalability	1.0	1.2792	1.4661	1.7213	1.9978
	SpeedUp	0.9665	1.236	1.417	1.6637	1.9309
	Efficiency	0.9665	0.6182	0.3542	0.2079	0.1206
FastFlow	***Completion***	7.1211	5.9529	5.2340	3.8365	4.0589
	Scalability	1.0	1.1962	1.3605	1.8561	1.7544
	SpeedUp	0.9725	1.1633	1.3231	1.8051	1.7062
	Efficiency	0.9725	0.5816	0.3307	0.2256	0.1066
	Ideal Comp	6.9256	3.4628	1.7314	0.8657	0.4328
	Sequential	6.9256				

Table 3. *All times and measures for 400 trees, 200 points, 16 workers*

	Workers	***1***	***2***	***4***	***8***	***16***
Thread	***Completion***	10.0401	7.8192	6.7006	5.1887	4.6652
	Scalability	1.0	1.2840	1.4983	1.9349	2.1521
	SpeedUp	1.1410	1.4651	1.7097	2.2079	2.4557
	Efficiency	1.1410	0.7325	0.427	0.2759	0.1534
FastFlow	***Completion***	9.7741	7.2471	5.4729	4.2342	3.9741
	Scalability	1.0	1.3486	1.7859	2.3083	2.4594
	SpeedUp	1.1721	1.5808	2.0933	2.7057	2.8827
	Efficiency	1.1721	0.7904	0.5233	0.3382	0.1801
	Ideal Comp	11.45	5.7283	2.8641	1.4320	0.7160
	Sequential	11.4566				

A. User guide

A1. Workspace organization

The project workspace is organized as follows:

- `src` folder contains all the source code of the program;
- `include` folder contains all the .h libraries of the program;
- `test` folder contains the bash script test.sh which contains the code to run experiments;
- `results` folder contains all the csv files of the different machines where the code is executed, generated by tests;
- `plots` folder with graphs generated by a Python scripts, that are in the same folder;
- `Makefile` compiles the project as explained in *A.2*.

In the following we always consider the current working directory the project root folder.

A2. Compilation

- To compile the project a `Makefile` with three rules is provided:
- `make gen_threads`: produces the executable for C++ thread and sequential both (note sequential has zero workers as input parameter),
- `make gen_ff`: produces the executable for FastFlow version,
- `make cleanall`: eliminate all compilation files.

A3. Execution and test

- To execute C++ threads and sequential code write:

  ```
  make ARGS="treesNum treesHeigth nExec iterNum tol perc” < “functName.txt" run_gen_threads
  [>> gen_threads.txt]
  ```

  Example:
  ```
  make ARGS="400 5 4 50 1.0 40.0” < “3x+sin2x_100.txt" run_gen_threads >> threads.txt
  ```

- To execute Fast Flow code write:

  ```
  make ARGS="treesNum treesHeigth nExec iterNum tol perc” < “functName.txt" run_gen_ff [>>
  gen_ff.txt]
  ```

  Example:
  ```
  make ARGS="400 5 8 50 1.0 40.0” < “3x+sin2x_100.txt" run_gen_ff >> ff.txt
  ```

The meaning of arguments is the following:

1. `treesNum` = number of trees the program will randomly generates,
2. `treesHeigth` = max allowed trees height (it will be a random in [*3, treesHeight*] range),
3. `nExec` = number of workers (zero to run sequential version),
4. `iterNum` = max number of iteration that the algorithm will run,
5. `tol` = max Fitness value tolerated (if fitness < tol computation will early stop),
6. `perc` = percentage of population for which genetic operator will be applied,
7. `functName.txt` = file from which the main will extract all points set.

In `test` folder there are .sh scripts to run test (each one for type -seq, threads, FF- and each data size). To run one write, for example:

```
./test/test_gen_ff.sh
./test/test_gen_threads.sh
./test/test_gen_sequentialf.sh
```

A4. Graphs

The graphs in this report are generated from a python script GeneticGraphs.py.
It simply reads result files obtained from the execution of tests and plots.

```
python GeneticGraphs1.py
python GeneticGraphs2.py
python GeneticGraphs3.py
```