CentraleSupélec

GENETIC ALGORITHMS FOR TASKS SCHEDULING PROBLEM LONG REPORT

Charles Jestin-Scanvion, Mathieu Peel, Paul Hainsselin, Bilal Manssouri, Somia Soudi

charles.jestin-scanvion@student-cs.fr
mathieu.peel@student-cs.fr
paul.hainsselin@student-cs.fr
bilal.manssouri@student-cs.fr
somia.soudi@student-cs.fr

April 2021

Contents

1 Introduction

1.1 Context

Scheduling problems are one of the most crucial NP-complete problem in parallel and distributed computing systems, they are optimization problems, in which tasks are assigned to resources at particular times. For example, we have n tasks: $t_1, t_2, ...t_n$ that should be scheduled on m processors: $P_1, P_2...P_m$.

The goal is to minimize the make span, which is the total length of the schedule, such that precedence constraints are satisfied. We modeled this problem using the Directed Acyclic Graph (**DAG**), which we will present later in this report, and to solve it we chose a Genetic approach using the well known Genetic Algorithm.

1.2 Genetic Algorithms

Genetic Algorithm (GA) is a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms, they are optimization techniques used to solve nonlinear or non-differentiable optimization problems, they use concepts of evolutionary biology to search for a global minimum of an optimization problem.

Genetic Algorithms start with an initial generation of candidate solutions that are tested against the objective function, then subsequent generations evolve from the first through **selection**, **crossover**, and **mutation**.

In genetic algorithms, each solution is generally represented as a string of binary numbers, known as a **chromosome**. We must test these solutions and come up with the best set of solutions to solve the problem. Each solution, therefore, needs to be awarded a score, to indicate how close it came to meeting the overall specification of the desired solution. This score is generated by applying the fitness function to the test, or results obtained from the tested solution.

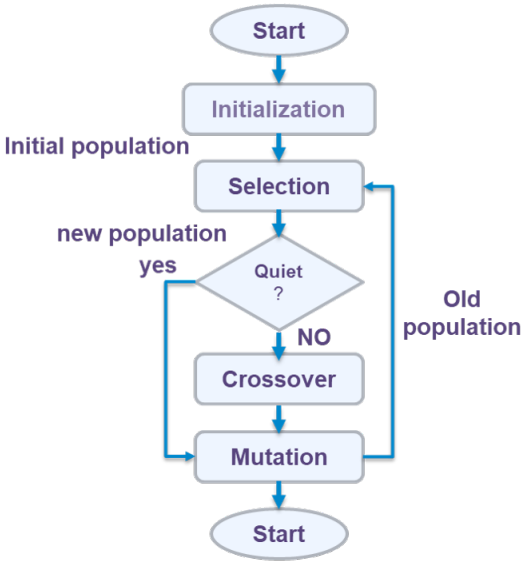


Figure 1: Genetic algorithm

2 Architecture

2.1 Directed Acyclic Graphs and topological orders

We call a directed acyclic graph any partial mathematical order on a finite set S. This means we have an operator $\prec$ on S with the property:

$$\nexists x_1, \ldots, x_k \in S, x_1 \prec x_2 \prec ... \prec x_k \prec x_1$$

There is an obvious graph representation of $(S, \prec)$ which we call a **DAG**, as the rule given above prevents any cycle from occurring into the representation.

We call a linear extension of $(S, \prec)$ any total order $<$ on S, where

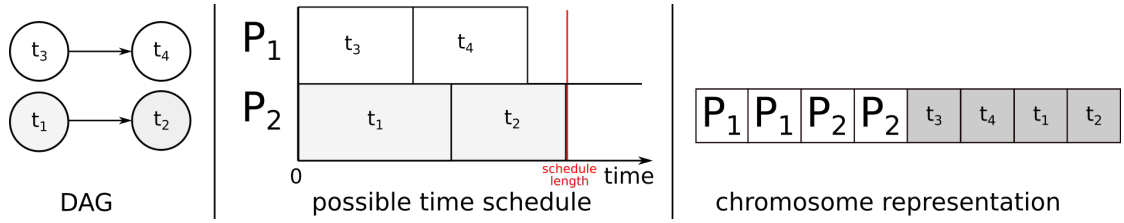
$$\forall x, y \in S, x \prec y \implies x < y$$

There is an obvious representation of this linear extension called a topological order of the **DAG**, which is a list containing every element of S once, and where: x *is on the left of* $y \Leftrightarrow x < y$.

2.2 Data representation

The problem data is nothing more than a partial order on the set of tasks. What we aim to find is a linear extension of the **DAG** joined with an allocation of processor for each task. Then, we schedule on each processor the task in the order given by our linear extension, as soon as the processor is free and all of its dependencies have been computed. So as to reduce the number of unused time on the processors, the "as soon as the processor is free" also include any slot of time situated between the finish time of all the dependence of the task and the time the processor is free (no more work to do). We will develop this in the implementation section.

To represent the data, we create two lists, one list of processors, the second a linear extension of the **DAG**, and the i-th element of the task list is done on the i-th processor of the processor list. This is the basis for our chromosome representation.



2.3 Combinatorial ideas

On a given **DAG**, the number of topological sort can range from 1 to $n!$, depending on the structure of the graph. Add up on this issue the repartition of the processors which gives M^n possibilities. One can still consider the quotient of all the processors allocation by the group of permutations $\sigma(M)$ (all the processors are identical so any permutation of the processors list will essentially give the same solution). In the worst case, we have to try $n!M^n/M!$ possible solutions.

On average, the expected number of topological sorts on a random **DAG** behaves asymptotically as $n!$ [1]. It is therefore impossible to choose any naive approach. Any graph that has more than a dozen of nodes will crush our computational power due to a combinatorial explosion of the number of solutions. This is why the use of a meta-heuristic is essential in finding a local minimum which improves significantly our solution.

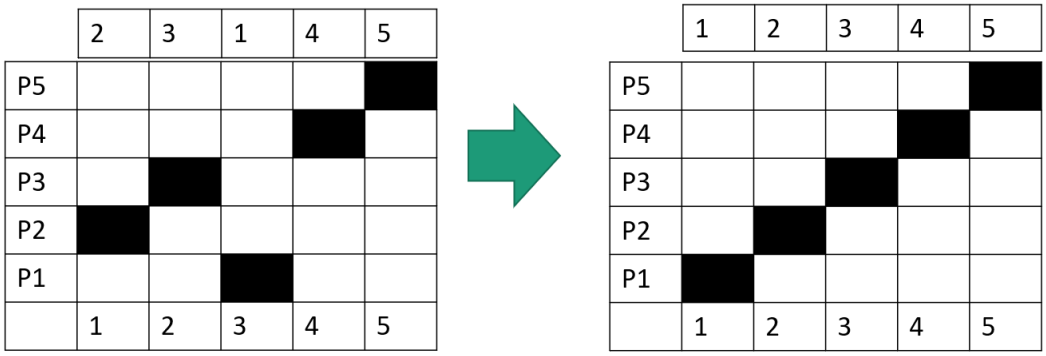
2.4 Reduction of the repartition of the processors

As we said, we absolutely want to quotient the processors allocation by the group of permutations $\sigma(M)$. To do that, what we implement is a reduction function applied to the list of processors to force one representation. The algorithm is :

- Scan the list from the beginning to the end, and save $P_1, \ldots, P_j$, where P_i is the $i-th$ **different** processor seen in the list
  We end up with $(P_1, \ldots, P_j) \in [\![1, M]\!]^j$, with $j \leqslant M$
- Re index the list, by mapping $\forall i, P_i \mapsto i$

By applying the cyclic reduction algorithm, we end up with lists that are equal should they be equal up to a permutation.

We call this function in the code `cyclic_reduction`.
Example : `[2,3,1,2]` $\mapsto$ `[1,2,3,1]`



3 Evolution

We took inspiration from [2] in large part for this algorithm, with

- The data representation (except the cyclic reduction we added)
- The fitness function (that we also improved)
- Crossovers and mutations (except the topological mutation that we added)

3.1 Initialisation

Given a size of our population P_{size}, we can create P_{size} amount of individuals randomly. In order to do this, we both need a linear extension of our nodes, and a processor allocation. We can simply take a list of random processors with the same size as the number of nodes. However we can't apply a deterministic algorithm that would create a linear extension. Quite nicely, there is an easy way to create a random topological order.

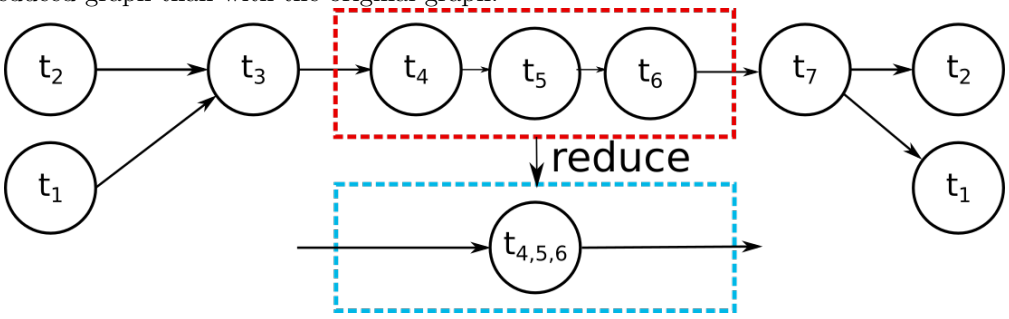
- Fetch for every node the amount of dependencies it has. We call that the "in-degree" of a node.
- Pick a random node t_i with in-degree 0 and append it to the linear extension of our graph.
- Reduce the in-degree of every node with with t_i as its dependency by one.
- Repeat the last two steps until we have visited every node of the graph.

It is easily proven that any topological order of a **DAG** can be obtained through this algorithm, hence the expected diversity of our initial population.

This algorithm can be parallelised, in 2 different ways. Either by allocating to each processor a sample of the graph and run the genetic algorithm to only a subset of the graph or run one algorithm on the whole graph but on multiple processors (cf the parallelisation section for more details)

3.2 Reduction

We introduced a function to reduce the number of nodes and vertices in the graph by reducing long chains of tasks into a single one. This approach may cause suboptimal results. But in some cases, with the same number of iterations, the genetic algorithm would find better solutions with a reduced graph than with the original graph.



We did some test with and without this algorithm. On 6 iterations of the algorithm (3 with reduce, 3 without), run on 4 processors, on a laptop, with a population size of 96, on 20 generations, we had the following results :

Time

- Average time with reduce : 30 seconds
- Average time without reduce : 96 seconds

Quality of result

- Average of the best solution with reduce : $2.79 \times 10^6 s$
- Average of the best solution without reduce : $2.83 \times 10^6 s$

We see that, for 20 generations, the result is unchanged (we cannot say anything other than that with a so little number of tests, and cannot say anything for a different number of iterations greater than 20).
What we can say with certainty is that, in this case, the gain in time is much greater than the time spent reducing the graph as the total time is 3 times lower

3.3 Evaluation

The evaluation function is a refined version of the one found in [2]. There are two version of the evaluation function:

- the first that only calculate, with a deterministic behavior the time required to complete the graph task from a chromosome representation

- The second also returns the timetable (gives, for each processor the starting time and end time of the tasks programmed on the processor), the total time of the tasks of the graph (what time it would take on a single processor), what theoretical best time we would obtain (time on one processor divided by the number of processors) and the theoretical time on an infinite number of resources

We initialize ST,FT two dictionary of the starting times and finish times of tasks.
We initialize RT a list of ready time to 0 for all the processors.
We initialize a dictionary empty_ slots in which we will add any empty slot that we create along the way

For each couple of processor/task (P_i, t_i), $i = 1$ to n

- Calculate $DAT(t_i) = \max_{t_j; t_j \to t_i} \{FT(t_j)\}$ (correspond to the time when all the task dependencies of t_i are finished)
- If there exist a empty slot between $DAT(t_i)$ and $RT(P_i)$ such that we can fit the task t_i, put it in, define $ST(t_i), FT(t_i)$ and update `empty_slots`
- Else, $ST(t_i) = \max\{DAT(t_i), RT(P_i)\}$. and $FT(t_i) = ST(t_i) + T(t_i)$ with $T(t_i)$ the time requirement to complete the task t_i. If $ST(t_i) > RT(P_i)$ we have created an empty slot, we update `empty_slots` in consequence

For the time on one processor, we only have to add up all the time requirements of all the tasks of the graph.
For the time requirement on infinite resources, we change the algorithm to only put $ST(t_i) = \max\{DAT(t_i)\}$

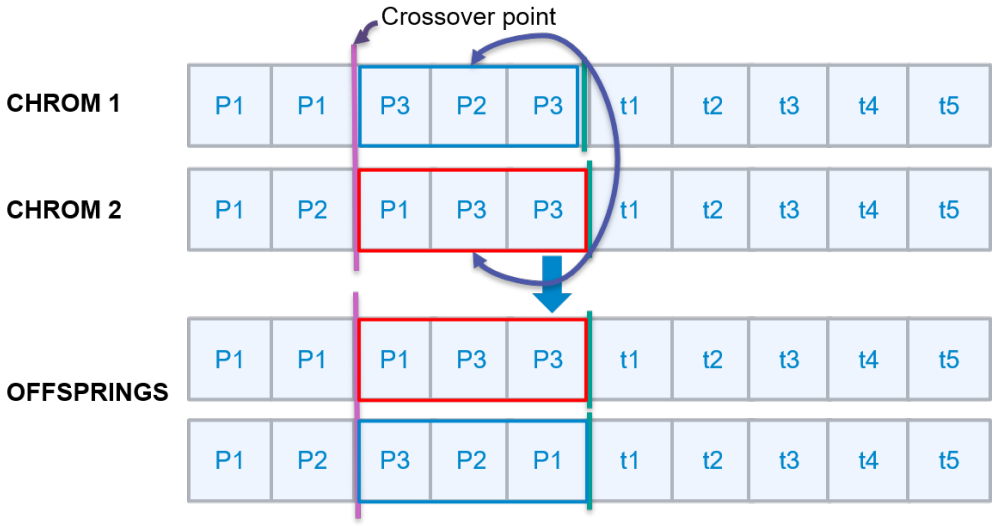
Considering that this function is very costly, we keep track of the individuals which fitness has been changed, in order to only evaluate them and not the whole population regardless of whether the genome of the individual has been changed on not.

3.4 Crossovers

Each chromosome within the populace is subjected to crossover with probability µc, there are two types of crossovers : The map crossover, and the Order crossover.

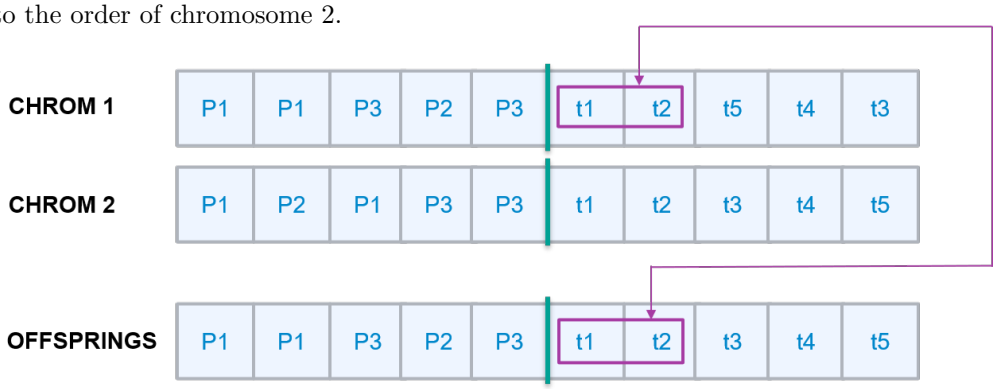
3.4.1 CMAP

This crossover concerns the mapping part of the chromosome. We generate a random number called crossover point between 1 and Number of tasks and we exchange the parts located to the right of the crossover point.



3.4.2 CPGA

This crossover concerns the scheduling part of the chromosome. As in the previous crossover, we generate a random number between 1 and Number of tasks. We pass the left segment from the chromosome 1 to the offspring, then we construct the right fragment of the offspring according to the order of chromosome 2.



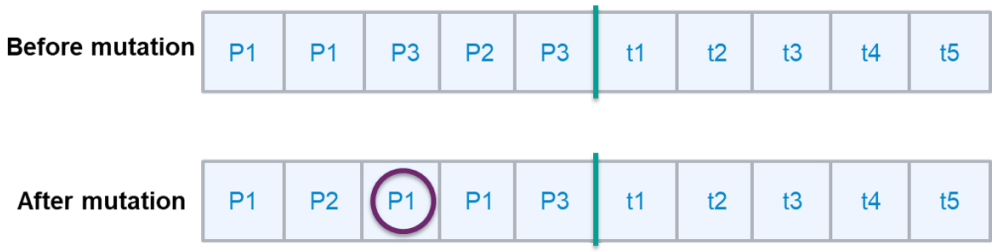
It is very important to realize that we still have a topological order at the end. Indeed, the right part and the left part of the offspring are partial linear extensions. If we call L_o and R_o the left and right part of our offspring (in the example, $L_o = [t_1, t_2]$ and $R_o = [t_3, t_4, t_5]$) then $\forall (x, y) \in L_o^2, x \prec y \implies x$ is on the left of y in L_o. This is true because L_o respects this property in the first chromosome. We have the same thing for R_o. For x, y in $L_o \times R_o$, since the first chromosome is a topological order, we have that $x \prec y$.

This idea is a very fast way of applying complex transformations to our population, without changing the feasibility of an individual.

3.5 Mutation

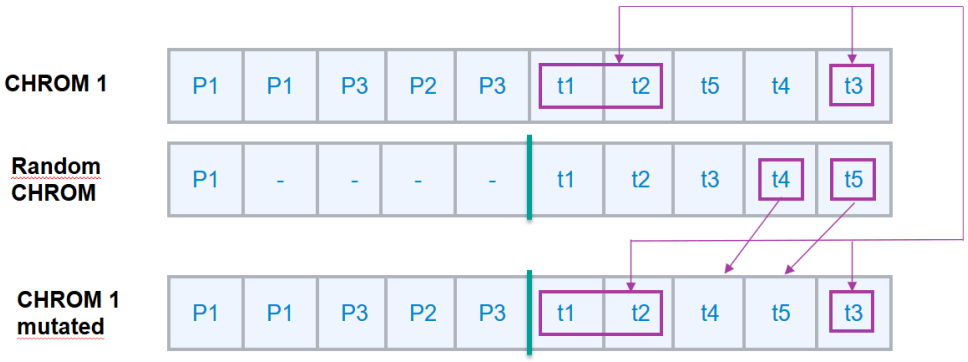
3.5.1 Processors mutation

Each position in the first part of the chromosome is subjected to mutation with a probability µm. Mutation involves changing the assignment of a task from one processor to another.



3.5.2 Topological order mutation

The main issue with having only processors mutation is that our exploration space is inherently limited to all the crossovers of our population at any given generation. This is a problem if we want to run our algorithm for a while: the population will eventually be filled with individuals having the same tasks order. To solve this problem, one can introduce a new mutation inspired from the **CPGA** mutation. The idea is to create a random individual and apply a 2 points crossover. We keep the individual before the first point and after the second point, and fill the gap with the missing tasks while keeping the order of the second random individual.



The percentage of the original chromosome that has to be kept can be changed in order to obtain some big mutations or small nudges.

3.6 Cyclic reduction

We talked about the cyclic reduction in the last section. This reduction function is applied after each operation done on a processor list (be it a processors mutation of a CMAP crossover) but also after the initialisation of the population .
After careful testing, the effect of the cyclic reduction is much smaller than previously thought, as the variability of the results of the algorithm is important. One can have bad luck on a long

number of iterations with no cyclic reduction and good luck with cyclic reduction (that was what happened, see the two results at 5.2.2, the first was done with cyclic reduction, the second without and, in this case, the second one gave worse result in 500 generations than the other one in 150 generations).
Testing with and without cycling reduction, with a fixed number of generations gives results too close to one another to be conclusive.

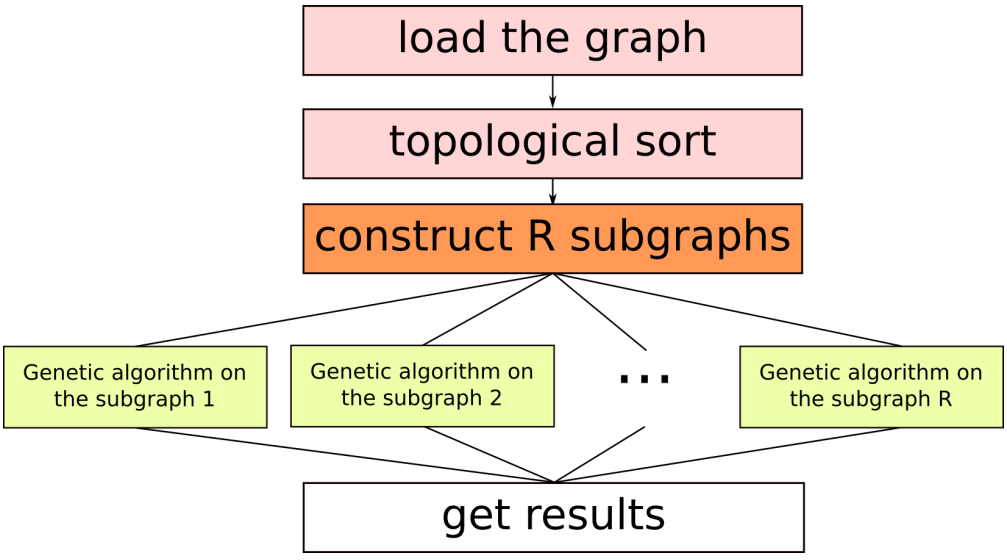
4 Parallelization

4.1 One genetic algorithm/multiple processors

The first approach we took was the one of a unique genetic algorithm but with operations (initialisation, evaluation, crossover, mutations) done on multiple processors. In this case, every chromosome represent a solution to the whole problem on the whole graph. More specifically, the individuals on which an operation has to be done are broadcast on the available nodes, each node calculates the result and sends it to the root.

This approach is particularly efficient on small problems, but becomes unpractical for bigger graphs. The major issue comes with the size of the data. Consider a 1Gb graph, which information are stored into 100 individuals. It is more than 100Gb that has to be broadcast several times each generation. That is why we chose to explore another approach as the size of the graphs grew.

4.2 Multiple genetic algorithms/multiple processors



The second approach was the one of cutting the large problem into smaller problems, each resolved on its own processor with a separate genetic algorithm. We denote R the number of processors thread we have at our disposal with MPI.

On the figure, we can see the algorithm implemented. It was done as a proof of concept but need further refinement to be used. The colors mean :

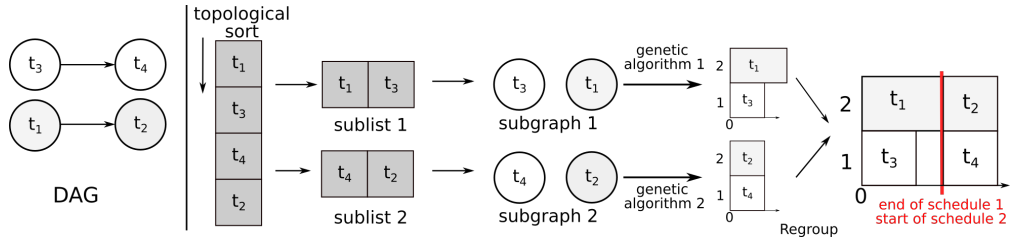
- Red : not currently parallelized
- Orange: not currently parallelized but can be done with minimal modification (here : construct the R subgraphs on each of the R processors instead of the $MPI\_RANK = 0$ one)
- Green : parallelized (here each subgraph i is processed on the processor i)

4.2.1 Creating smaller problems

Constructing the smaller graphs is done using the whole graph and a topological sort on the whole graph. With this what we do is

- Separate the topological sort list into R sublists of approximately the same size
- For each sublist i (i is the rank of the subgraph) of the topological sort, create a new graph only containing the nodes present in the sublist and vertices between two nodes of the sublist
- (for future work) : Store the inbound nodes and outbound nodes with their associated subgraph rank

Given these subgraphs we can run a genetic algorithm on each of them. At the moment, the approach is a naive one : find the minimum time on each subgraph then concatenate the execution schedule into one. The final time is the sum of the individual ones, and there is no need for exchanges between processors during the algorithm.
To improve upon this idea, we should communicate between the subgraphs and adapt the evaluation function to take into account the state of previous subgraphs. This has not been tested but some of the function (inbound nodes, outbound nodes and updating the graph) are implemented.



4.3 Benefits

This allowed us to run on 24 processors an optimisation on the `largeComplex` graph with 42241 nodes and 223952 vertices in total. In the current state of the algorithm, the red and orange tasks were the bottleneck for speed but the parallelized part run smoothly.
This was not possible with the precedent approach (as in not possible, running on Azure with 5 nodes crashed at the first iteration, and on Paul's desktop computer on 24 cores, his computer was on the verge of collapse so we stopped)

5 Results

Remark : the algorithms were run on different machines, on a laptop for some (4 cores i7) and these present a much larger time spent on initialization than the others that are run, either on Azure or on a desktop (this over prevalence of the initialization phase is much less pronounced on these two)

5.1 One genetic algorithm/multiple processors

5.1.1 smallComplex

This first result was obtained with conservatives settings and a small number of cores (cf fig 2). For good measure we add a result obtained with 24 cores on a desktop computer (fig 3)

5.1.2 Exponential graph

One major difficulty to assess performance comes from the fact that the optimal solution to the problem is unknown. We thus created a graph generator of variable size, with a known minima. With $P = 4$, $L = 5$ we created the $P4L5$ graph, with a known minimum of $342s$ each task having the same time requirement of 1 second.

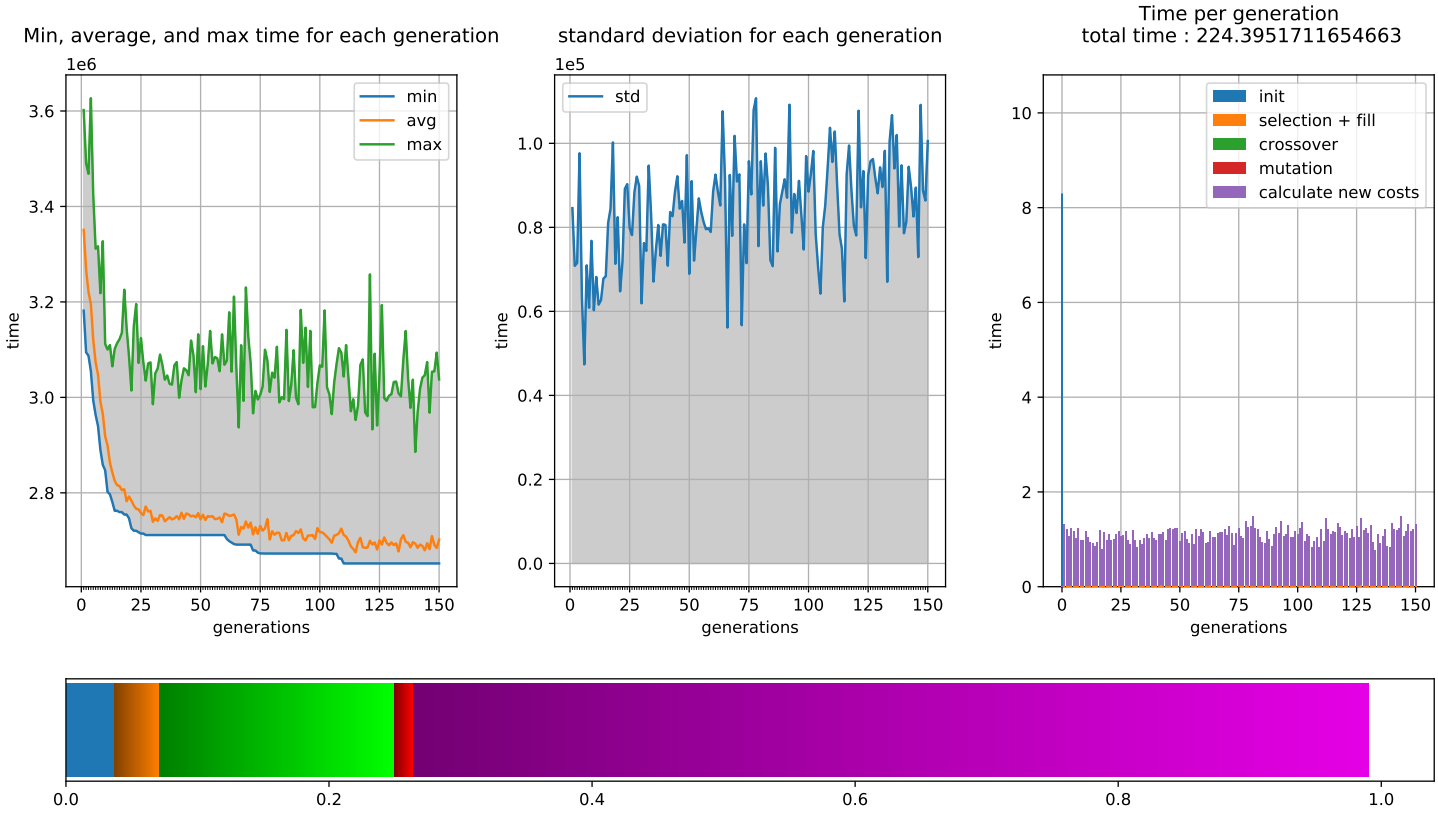
The minima obtained is $350s$ close to the absolute minimum of $342s$. If we observe the timetable, we still have some empty slots (but it was an improvement over the results we had without the updated evaluation function taking into account the empty slots of the processors)

5.2 Multiple genetic algorithms/multiple processors

5.2.1 largeComplex

As explained we have been able to use this algorithm to compute a solution for the largeComplex graph. We obtain, in 100 generations a minimum of 1.54×10^6 seconds

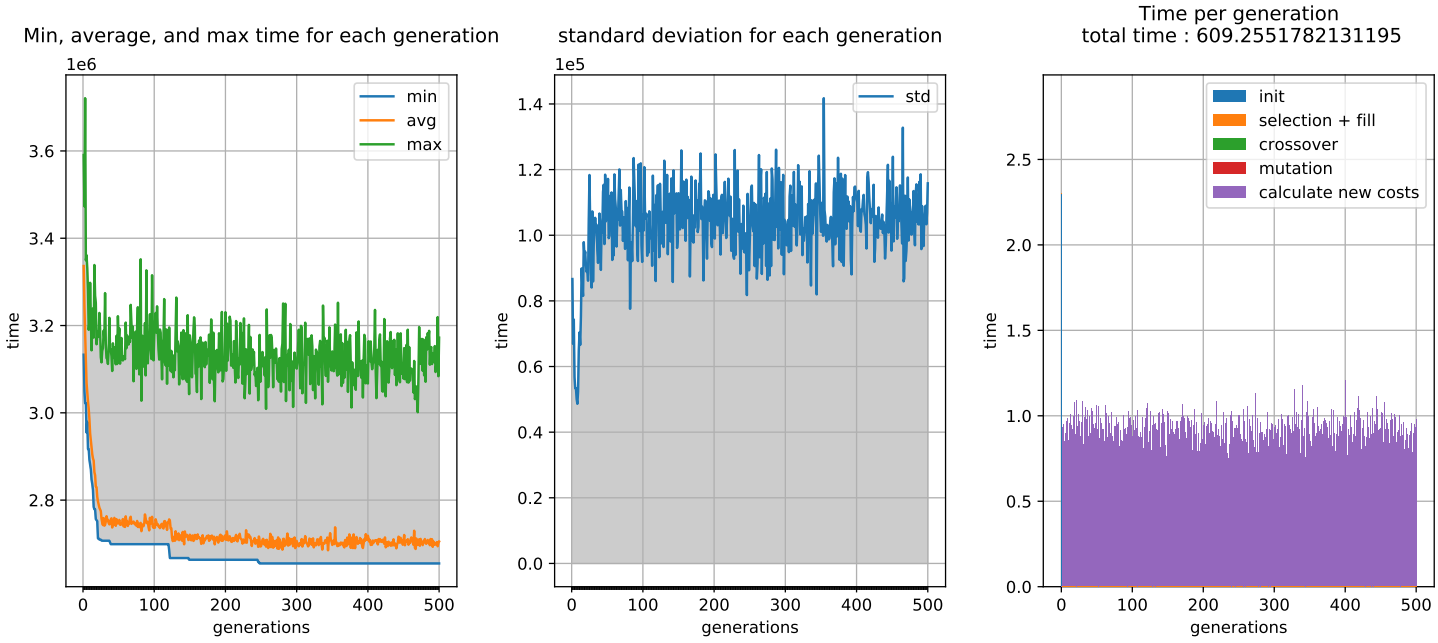
MPI parameter : 3 processors used to compute
Parameters of the genetic algorithm : Nb Processors : 4 Graph used : GraphssmallComplex.json Pop size : 96 number of generations : 150 selection size : 96
cpga probability : 0.8 crossover probability : 0.5 mutation probability : 0.2 reduce: 1 mutate_order : 0.0



Best time found :	30 days, 16:41:34.050687	so 2652094.050687 s
Theoretical best time on 4 processors	20 days, 16:57:56.998778	so 1789076.998778 s
Theoretical best time on ∞ processors	1 day, 20:06:57.989646	so 158817.989646 s
Time on one processor	82 days, 19:51:47.995112	so 7156307.995112 s

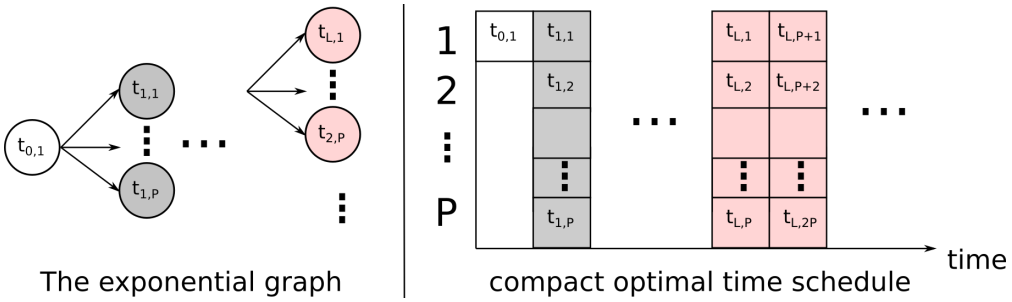
Figure 2: results for the smallComplex graph (result on a laptop)

MPI parameter : 24 processors used to compute
Parameters of the genetic algorithm : Nb Processors : 4 Graph used : GraphssmallComplex.json Pop size : 192 number of generations : 500 selection size : 192
cpga probability : 0.8 crossover probability : 0.4 mutation probability : 0.2 reduce: 1 mutate_order : 0.0

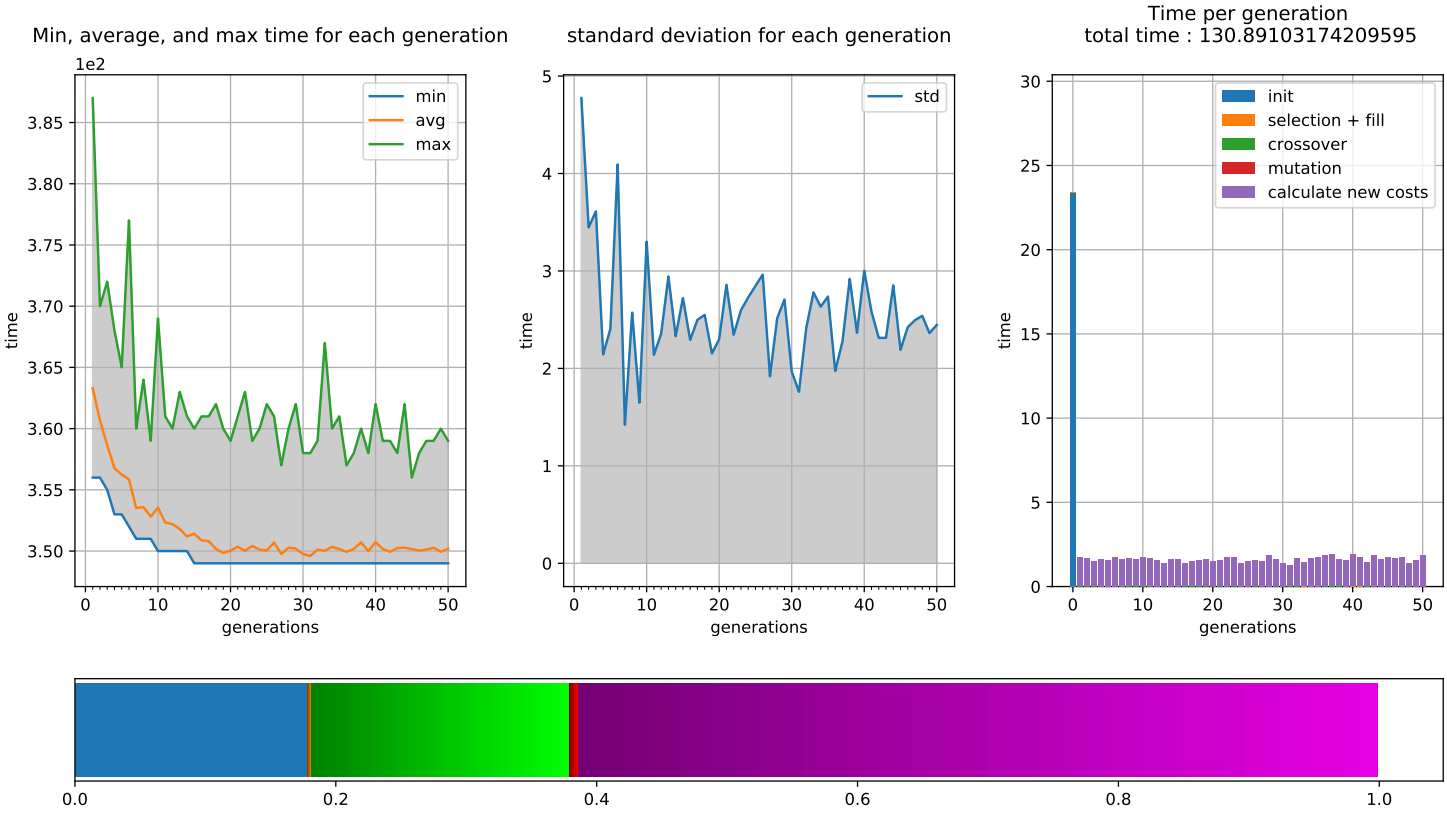


Best time found :	30 days, 17:28:36.459348	so 2654916.459348 s
Theoretical best time on 4 processors	20 days, 16:57:56.998778	so 1789076.998778 s
Theoretical best time on ∞ processors	1 day, 20:06:57.989646	so 158817.989646 s
Time on one processor	82 days, 19:51:47.995112	so 7156307.995112 s

Figure 3: results for the smallComplex graph (result on a desktop)



MPI parameter : 3 processors used to compute
Parameters of the genetic algorithm : Nb Processors : 4 Graph used : GraphsP4L5.json Pop size : 96 number of generations : 50 selection size : 96
cpga probability : 0.8 crossover probability : 0.5 mutation probability : 0.2 reduce: 1 mutate_order : 0.0mutate order each 75 generations



Best time found :	0:05:49	so 349.0 s
Theoretical best time on 4 processors	0:05:41.250000	so 341.25 s
Theoretical best time on ∞ processors	0:00:06	so 6.0 s
Time on one processor	0:22:45	so 1365.0 s

Figure 4: The exponential graph and results for P4L5 (result on a laptop)

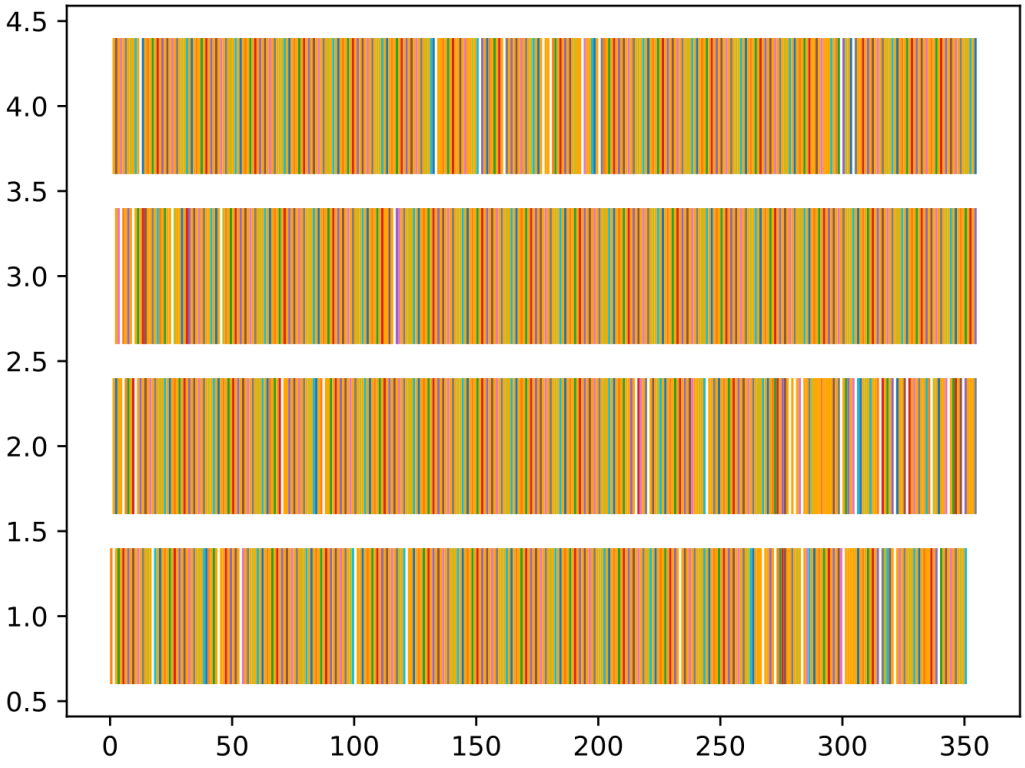


Figure 5: Timetable for the P4L5 graph : close to the optimal one but with some empty slots

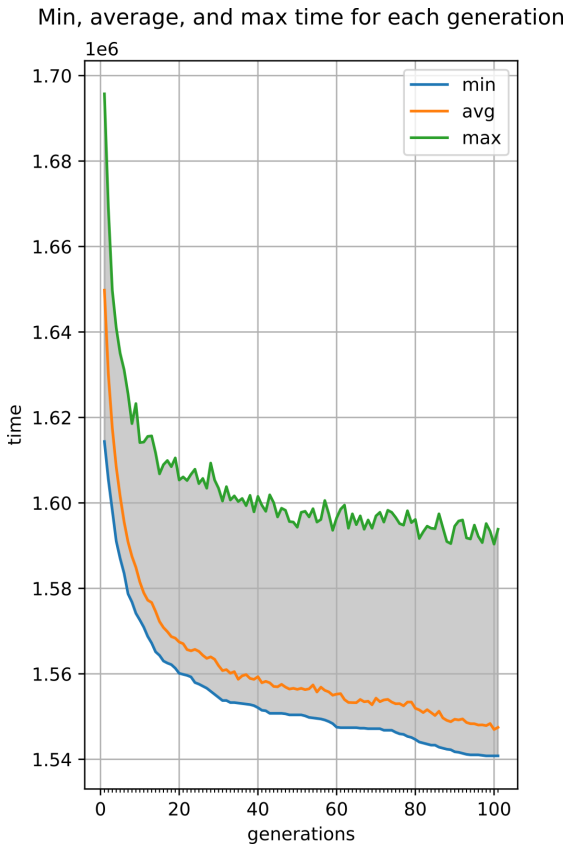


Figure 6: Results for LargeComplex (on 20 processors, desktop)

References

[1] Graham Brightwell, Hans Jürgen Prömel, and Angelika Steger. "The average number of linear extensions of a partial order". In: *Journal of Combinatorial Theory, Series A* 73.2 (1996), pp. 193–206. ISSN: 0097-3165. DOI: https://doi.org/10.1016/S0097-3165(96)80001-X. URL: https://www.sciencedirect.com/science/article/pii/S009731659680001X.

[2] Fatma A. Omara and Mona M. Arafa. "Genetic algorithms for task scheduling problem". In: *Journal of Parallel and Distributed Computing* 70.1 (2010), pp. 13–22. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2009.09.009. URL: https://www.sciencedirect.com/science/article/pii/S0743731509001804.