Assessing the Use of Genetic Algorithms to Schedule Independent Tasks Under Power Constraints

Ayham Kassab, Jean Nicod, Laurent Philippe, Veronika Sonigo

Assessing the Use of Genetic Algorithms to Schedule Independent Tasks Under Power Constraints

Ayham KASSAB, Jean-Marc NICOD, Laurent PHILIPPE, Veronika REHN-SONIGO
FEMTO-ST Institute, Université Bourgogne Franche-Comté / CNRS / ENSMM
F-25000 Besançon, France
[ayham.kassab—jean-marc.nicod—laurent.philippe—veronika.sonigo]@femto-st.fr

***Abstract*—Green data and computing centers, centers using renewable energy sources, can be a valid solution to the over growing energy consumption of data or computing centers and their corresponding carbon foot print. Powering these centers with energy solely provided by renewable energy sources is however a challenge because renewable sources (like solar panels and wind turbines) cannot guarantee a continuous feeding due to their intermittent energy production. The high computation demand of HPC applications requires high power levels to be provided from the power supply. On the other hand, one advantage is that unlike online applications, HPC applications can tolerate delaying the execution of some tasks. Since the users however want their results as early as possible, minimum makespan is usually the main objective when scheduling this kind of jobs. The optimization problem of scheduling a set of tasks under power constraints is however proven to be NP-Complete. Designing and assessing heuristics is hence the only way to propose efficient solutions. In this paper, we present genetic algorithms for scheduling sets of independent tasks in parallel, with the objective of minimizing the makespan under power availability constraints. Extensive simulations show that genetic algorithms can compute good schedules for this problem.**

***Index Terms*—Energy efficiency, computing center, scheduling, genetic algorithm**

I. INTRODUCTION

Big data center operators are cutting the costs of the energy consumed by their centers by locating them in cold areas, taking advantage of cold air to lower the cooling expenses and thereby achieving a better PUE[1]. This is, for instance, the case of HP's Wynyard data center in the United Kingdom or Facebook's in Lulea, Sweden. By building green data centers solely powered by green energy sources, data center operators do not only reduce the electricity bill, but also provide a lower carbon footprint. Verne Global's datacenter, located in Keflavik, Iceland operates with 100% sustainable green power using geothermal and hydro-electric sources. Such structures make an ecological platform for green computing, which has become an inevitable necessity in High Performance Computing (HPC) as it heads towards exascale computing. Low cost renewable energy sources, as wind turbines and solar panels, however produce intermittent energy and this implies to adapt the computation schedule to the available power.

[1]PUE: Power Usage Effectiveness = total facility power delivered/IT equipment power usage

HPC applications are usually tolerant to delay, unlike online applications such as web services, which demand instant response. HPC applications are generally batch applications, submitted via a script and put in a job queue until the scheduler finds available computing resources to execute them. This makes running HPC applications on a platform powered by green energy sources possible, as the users do not expect instant response, but not necessary easy. The challenge here is not to reduce the energy consumption but rather to schedule a set of jobs under the constraint of the power availability, as the power varies over time, and to finish their execution as early as possible. The scheduling algorithm must hence take the best of the available power to decide which task to run when the power is low, and which tasks to delay until more power is available, in a way to minimize the makespan.

Finding the optimal makespan, the shortest possible time to execute a set of tasks under power constraints, is proven to be NP-Complete in [12] and, by assessing classic list algorithms on this problem, it is shown that the algorithm performance depends on the problem characteristics as mean task duration or power consumption. In this article, we assess the use of genetic algorithms on the problem of scheduling a set of tasks under power constraints to minimize the makespan. The contributions of the paper are the following:

- The proposition of a genetic representation of a schedule, mutation and crossover operators that allow to explore the space of possible solutions;
- An extensive performance study of several genetic algorithms;

The remainder of the paper is organized as follows. In Section II we present research works related to this problem. In Section III we formally define the model of the problem. In Section IV we propose a genetic based scheduling algorithm, mutation and crossover operators. In Section V we detail the performance results of the different versions of the genetic algorithm before concluding in Section VI.

II. RELATED WORK

a) Energy efficient infrastructures: Energy efficiency has risen as a global concern for HPC applications so that many works tackle this problem, most of them focus on reducing the energy consumption of the facility's components (IT [11],

cooling system [15], power transportation or delivery [2], [8]), either by addressing the energy consumption of its IT components, or by considering the whole facility [15].

Another approach is to add renewable energy sources to the power supply. Goiri et al. [8] propose GreenSlot, a scheduler that matches the workload with a predicted green energy level by scheduling more jobs at times where the green energy production level is high. They still have recourse to brown energy in case of need. The Datazero research project [3] aims at proposing robust solutions for datacenters that are solely powered by renewable energy sources. They focus on a negotiation tool that links the IT parts of a datacenter to the electrical parts and which allows to adapt the power supply to the computing demand and vice versa.

Since green energy production highly depends on weather conditions, a possible solution to cope with the intermittent production is virtual machine migration between geographically distributed platforms. Zhang et al. [20] propose such heuristic algorithms for multi-site datacenters. The idea is to relocate computations to sites where the actual energy production is high. The authors show that despite of migration costs, their approach allows to save up to 31% of brown energy consumption.

b) Energy Aware Scheduling: A lot of effort has been made in the domain of energy aware scheduling, where people try to find solutions to reduce the energy consumption while computing [17]. Dorronsoro et al. [4] propose energy and completion time estimators used in their two-level strategy for scheduling large workloads in multicore distributed systems. A higher level scheduler dispatches requests among available clusters where a local scheduler schedules them on the multicore servers. They achieve up to 46.8% improvements regarding the makespan of their solution and up to 29% regarding energy consumption.

Commonly used on the IT level, Dynamic Voltage and Frequency Scaling (DVFS) slows down the processors in order to reduce the power consumption for the price of longer execution times [19]. A variant of DVFS, Dynamic Voltage Scaling (DVS) is used by Garg et al. [6]. Their scheduling solution for HPC applications on Cloud oriented data centers allows to reduce the energy consumption up to 25% in comparison to profit based approaches. The trade-off between task execution time and energy consumption in DVFS systems is evaluated in the work of Wang et al. [18]. The authors take advantage of slack times in order to increase task execution times with lower energy consumption. An increase of task execution times of 30% allows to achieve energy savings up to 70%. In recent CPU generations, DVFS is integrated in the design of the processor (e.g., Intel HASWELL) [10], [11] and thus less easily usable by external schedulers.

In [16] Tchernykh et al. evaluates online scheduling algorithms for IaaS clouds with the objectives of increasing provider income and reducing power consumption, taking the quality of service into account. The authors conclude that the strategy which allocates jobs to processors with minimum total power consumption is a stable solution which outperforms other strategies in almost all test cases.

Similarly to our time interval decomposition of the available energy Dutot et al. [5] reduce energy consumption by scheduling parallel jobs of a HPC cluster queue under energy budget constraints in a given time duration.

All these works however differ from our approach in that they try to reduce the energy consumption whereas we aim at optimizing the amount of computations within a power envelope, composed of time intervals where the available power is capped. Hence we are concerned by instant power while works on energy are interested in power consummed over time.

c) Genetic Algorithms: Genetic algorithms (GA) are widely used in NP-complete optimization problems, as they allow to generate high quality-solutions. In these algorithms, a population of candidate solutions evolves through nature inspired operations of crossover, mutation and selection towards better solutions. The crucial point for the success of GA is the modeling of problem properties via chromosomes. [1] is an example of early work on using GA for parallel scheduling. The authors combine GA with a list scheduling approach in order to schedule DAGs on multi-processor systems. Each chromosome represents a different configuration of the priority assignment for each node of the DAG. Therefore, the real numbers in the i-th gene represent the priority of the i-th node.

A parallel bi-objective hybrid genetic algorithm for reducing energy consumption and makespan in computing systems is presented in [13]. Each chromosome represents a possible solution, and each gene assigns one task to a processor and sets its voltage. The GA shuffles the tasks through the genes of a solution by a series of mutations and crossovers, while the processor and voltage parts of the genes are formed by an energy-conscious scheduling heuristic (ECS). The algorithm is evaluated by running a DAG parallel application on a grid of three clusters. Their results show that using this hybrid approach reduces the energy consumption by 47.49% and the makespan by 12.05% comparing to using only ECS. This work aims at reducing the power consumption using DVS scheduling DAG-like applications, whereas we focus on scheduling independent tasks under power constraints.

Lei et al. [14] propose a multi-objective co-evolutionary algorithm for scheduling tasks on data centers partially powered by renewable energy. The chromosome encoding is similar to the one in [13]. The processor and voltage parts of genes are the ones who change through a chromosome by mutation and crossover, while the order of tasks remains fixed according to their index. They evaluate scheduling tasks with different lengths and deadline on homogeneous computing nodes.

Both works ([13] and [14]) use DVFS to set the voltage of a processor, while the model considered here assumes that different tasks have different fixed power consumptions. Also they only partially supply the data center with renewable energy and have recourse to brown energy sources.

III. Model

As previously stated, we tackle the problem of statically scheduling a set of independent sequential tasks in parallel on a set of machines under power constraints. In this optimization problem we seek finishing the tasks as soon as possible, the objective function is hence the makespan or $C_{\max}$, the completion time of the last finishing task. Tackling this problem

requires to first formally define how the set of tasks, the machines and the power are considered. In this section we thus give a formal definition of the model used in the remaining of the paper. Note that we use the same model as in [12]. Table I summarizes the notations.

Table I: Summary of the notations.

var.	definition	var.	definition
$\mathcal{T}$	set of tasks	$\mathcal{M}$	set of machines
n	number of tasks	m	number of machines
T_i	task i	M_j	jth machine of $\mathcal{M}$
p_i	processing time of T_i		
φ_i	power needed by T_i		
Δ_x	interval with constant power	δ_x	length of Δ_x
X	number of intervals Δ_x	Φ_x	useful power of Δ_x

We consider a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of m execution units M_j on which to execute the tasks. The energy consumed by a computer is usually decomposed into static and dynamic energy, where the static energy is the energy needed to power the machine when its cores are idle and the dynamic energy is the extra energy used to run the tasks. The general model thus includes these values plus switch on/off times for the machines. In the current work, to reduce the complexity of the problem, we consider execution units as cores. The platform used to asses our algorithms rather models a multi-core machine than independent machines. In that case m, the number of execution units, corresponds to the number of cores of the machine.

The energy used to power these execution units comes from renewable sources, such as wind turbines, photo-voltaic panels (PVs) and so on. We are therefore not sure to be able to always run the machine at full speed. On a cloudy day, the energy produced by PVs could drop and become too low to run all the cores of the machine. On the other hand, the energy has no cost and unused energy is lost which means that our objective is to use as much energy as possible to finish the tasks the soonest. As the energy varies over time, we discretize it in a set of Δ_x intervals of length δ_x and of available power Φ_x.

Given this platform and energy we want to schedule a set $\mathcal{T}$ of tasks, composed of n sequential independent tasks T_i, $\mathcal{T} = \{T_1, T_2, \ldots, T_n\}$, whose processing time is p_i. In this paper we consider the power consumed by a task, its instant consumption, rather than its energy need. In [7] Glesser et al. show that running a task results in an energy consumption that depends on the task type, either intensive or not. Based on that we define φ_i as the power consumption φ_i of task T_i. This value can be the peak consumption of the task during its execution duration. Once started the tasks are uninterruptible.

As illustrated on Figure 1, our problem consists in scheduling the set of tasks, gray rectangles, in the time intervals, red lines, provided that the number of tasks scheduled at the same time does not exceed the number of cores of the machine. This optimization problem can be formally defined, using the Graham notation for scheduling problems [9] completed by [12], as $P|\varphi_i \leq \Phi_x|C_{\max}$ which means that we have identical processors, that the power consumed by a tasks (φ_i) is lower than the power provided to the machines (Φ_x) and our objective function is the makespan.

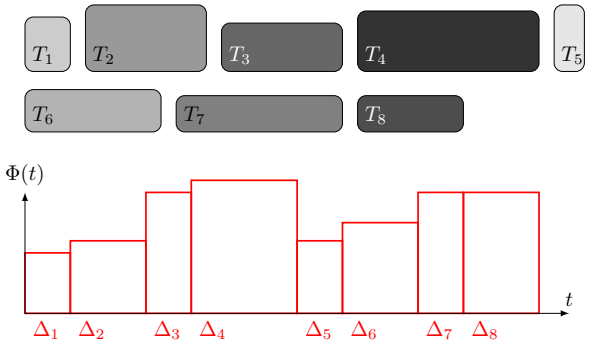


Figure 1: Illustrating example for the optimization problem

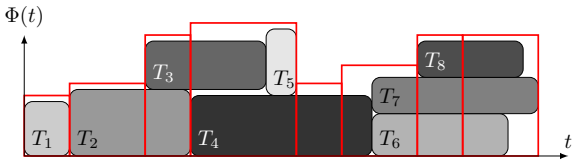


Figure 2: Example of a schedule

Note that using the power approach, instead of the energy one often used in the literature, allows to turn the problem into a one criterion classical optimization problem, the makespan, rather than a multi-criteria problem which only leads to compromises and no optimal solution.

IV. Genetic algorithms for power constraint

In a previous work [12] we have tested several classical list algorithms to solve this problem. We have shown that the best algorithm depends on parameters as mean task size or mean task power need. As shown in Section II, genetic algorithms often give interesting results in scheduling problems provided that a correct genetic representation of the schedule is found. In this paper we assess the use of genetic algorithms on this power constraint scheduling problem.

a) Chromosome representation of a schedule: The first issues to be solved when designing a genetic algorithm are to code the solutions under gene shapes and to measure to corresponding fitness. The fitness choice is rather easy as it usually corresponds to the optimization objective, the makespan here. There are however numerous ways to code a solution and the representation choices highly impact the results so that several solutions must be explored.

As a first try, each chromosome represents a possible schedule. A chromosome is made of n genes with the i-th gene being the time interval where task T_i is scheduled for execution. The time interval selected for each task is chosen from a list of all possible time intervals in the time interval list with enough available power. This list is calculated for all tasks once before generating the initial population. Since tasks are scheduled in parallel, several genes can have the same time interval, i.e., several tasks can be scheduled in the same time interval. The algorithm tries to schedule the task list according to the schedule presented by a chromosome, i.e., each task is executed at the time interval defined in its corresponding gene.

The fitness of the chromosome is then set to the makespan of the schedule if the latter is feasible or to infinity if not.

This approach has several drawbacks. The process of calculating the time interval possibility lists for each task is time consuming and the generation of non-feasible schedules when applying the mutation and cross-over operators leads to time losses when generating and evaluating the chromosome.

In [12] we established, on the one hand, that the order in which a list of tasks is passed to a list scheduling algorithm makes a big difference in the resulting makespan, and on the other hand, that, depending on the characteristics of a set of tasks, using the priority criteria of the task list impacts the performance of the resulting schedule. Therefore, an other GA was developed where each chromosome represents an ordered list of chromosomes that is then passed to a list algorithm that computes the corresponding schedule. The first obvious advantage is that all combinations of the task list order are valid if the available power constrains of the time slot list allows it. The coding and implementation is also simpler and faster than in the preceding proposition and thus solutions can be found in reasonable computation time.

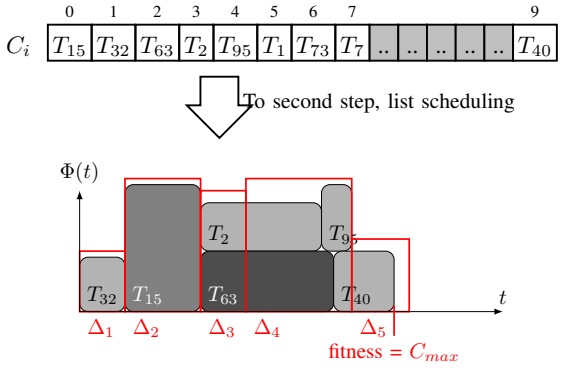


Figure 3: Illustrating example of a chromosome and the corresponding schedule

Figure 3 illustrates this approach. The chromosome representation is a simple ordered list of the tasks. To be valid it must contain all the tasks of the list and not twice the same task. The chromosome list is then passed to a list scheduling algorithm which schedules the tasks depending on the time interval constraints to compute the fitness, i.e. the makespan. Here T_{15} is the first task to be scheduled, the second task is T_{32}, and T_{40} is the last task to be scheduled. Due to power constraints task T_{15} cannot be scheduled first although it is the first in the list. Then, with respect of the power availability constraint, task T_{32} is executed in the first interval Δ_1 while task T_{15} is delayed until the second interval Δ_2 as there is enough power available at Δ_2.

This chromosome representation makes generating a new individual faster than in the previous one. The GA only needs to shuffle a list of integers $[1 \rightarrow n]$ that represents the indexes of all n tasks. This does not limit us to small population sizes.

b) Genetic algorithm: Algorithm 1 gives the main genetic algorithm used to manage the chromosomes and generate the schedules. In this GA the population size is set to 50. And by setting the number of intervals X high enough, it is valid to assume that all initial chromosomes give a feasible order of the task list.

Algorithm 1: `geneticAlgorithm(`$\mathcal{T}$`, `Δ`, nbI)`

```
Data: T, Δ: set of tasks, set of intervals
      nbI: number of iterations without enhancement
Result: task list order
1  stopCounter ← 0
2  currentGeneration[0] ← LPT(T, Δ)
3  currentGeneration[1] ← LPTPN(T, Δ)
4  currentGeneration[2] ← 2Qs(T, Δ)
5  currentGeneration[3] ← LPN(T, Δ)
6  currentGeneration[4:50] ← 46 random solutions
7  calculateFitnessOfPopulation(currentGeneration)
8  currentGeneration.sort()/* fitness: in increasing order
     */
9  while stopCounter ≤ nbI do
10     oldBest ← currentGeneration[0]
11     nextGeneration ← []
12     nextGeneration[0:10] ← currentGeneration[0:10]
13     for i=1 to 15 do
14         mutant ← mutation(selection(currentGeneration))
15         nextGeneration.append(mutant)
16     for i=1 to 15 do
17         mutant ← chunckMutation(selection(currentGeneration))
18         nextGeneration.append(mutant)
19     for i=1 to 10 do
20         C1, C2 ← selection(currentGeneration)
21         newC1, newC2 ← crossOver(C1, C2)
22         nextGeneration.append(newC1, newC2)
23     calculateFitnessOfPopulation(nextGeneration)
24     nextGeneration.sort()
25     currentBest ← nextGeneration[0]
26     currentGeneration ← nextGeneration
27     if oldBest − currentBest = 0 then
28         stopCounter++
29     else
30         stopCounter← 0
31 return currentBest
```

Four individuals of the initial population are the priority queues of four list scheduling algorithms: LPT, LPN, LPTPN and 2Qs. The LPT algorithm is the classical Largest Processing Time list algorithm that sorts the tasks with the largest p_i first. The LPN algorithm does the same but using the power need φ_i of the tasks as priority. The LPTPN algorithm uses the product of processing time and power need, $p_i \times \varphi_i$, to take both properties into account. Finally the 2Qs algorithm creates two lists, one sorted by p_i and the other by φ_i, and takes in turn one task in each list. Once a task is scheduled, it is removed from both lists. Note that the three scheduling algorithms LPT, LPN and LPTPN have already been assessed in a previous work. We however do not know their distance to the optimal, so we do not know the optimization potential that they leave to other algorithms. The other 46 individuals are randomly generated by shuffling a list of integers $[1 \rightarrow n]$ that represents the indexes of all n tasks as mentioned above.

Another important point in a genetic algorithm is the individual selection. We have implemented both wheel selection and random selection when choosing which individuals to apply the genetic operators on.

In Algorithm 1, first, the best ten individuals are copied to the next generation (elitism). Then, based on the used selection (wheel or random), 15 chromosomes are selected for 1-gene mutation (see $[l13 \rightarrow l15]$ in Algorithm 1) and another 15 chromosomes are selected for chunk mutation ($[l16 \rightarrow l18]$).

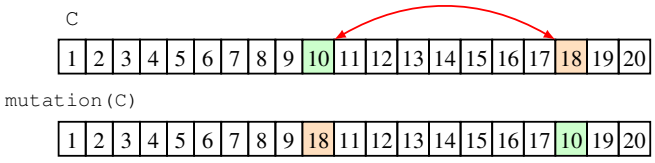


Figure 4: `mutation(C)`

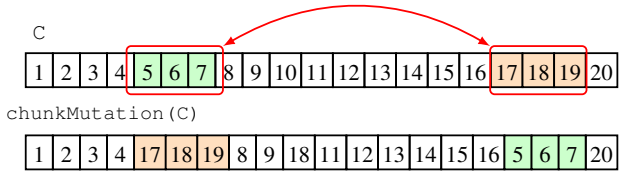


Figure 5: `chunkMutation(C)`

As for the rest of the evaluated GAs, 20 additional chromosomes are selected to perform 10 crossovers. In this study, we evaluate three crossover techniques. For each evaluation, one of these three crossovers is applied at $l21$ in Algorithm 1. Finally, the fitness of the new off-springs is calculated, and the entire process is repeated for the next generation while we run nbI steps without improvement.

The mutation and crossover operators are explained in the following.

c) Mutation: As presented previously two different mutation operators are used in the GA algorithm. In the 1-gene mutation operator, two points on the chromosome are randomly picked, and the values of the two corresponding genes are interchanged as illustrated in Figure 4. The chunk mutation operator uses the same concept as 1-gene mutation, only except swapping two "1-gene"s, it swaps 2 "chunk"s of random size between $1 \rightarrow 10$ of the chromosome at two randomly selected points (Figure 5).

Due to the characteristics of our problem, it is interesting to study the effect of small modifications on a given solution. Applying too many modifications might eventually be as arbitrary as a random solution. Hence, the first GA we evaluate applies only 1-gene or chunk mutation genetic operators (it skips lines $[l19 \rightarrow l22]$] in Algorithm 1). This algorithm is named noX, for no crossover, algorithm. With the two different selections, random (R) and wheel based (W), we define the algorithms: two noX-R and noX-W.

d) One point crossover: Crossover is one of the fundamental operators of GA and many crossovers mechanisms are described in the literature. In this paper we evaluate the affect of using three different crossovers. First, we evaluate the most common one point crossover. A randomly selected crossover point splits both parents into two parts. We cannot however directly exchange the chromosome parts as it is usually done because, doing this, the same task could appear twice in one chromosome, which would be a non-sense for a schedule. In our crossover we generate two new chromosomes, each one keeping its parent's head, while the genes of its tail (the remaining tasks) are ordered according to the order of the other parent. Algorithm 2 details the 1-point crossover operator.

Figure 6 illustrates this crossover operator. The crossover

Algorithm 2: `onePointCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks                    */
1        C2 /* chromosome 2 with n tasks                    */
   Result: newC1, newC2: 2 new chromosomes each with n tasks
2  n ← length(C1)
3  p ← intRand(0, n)/* integer random value: 0 ≤p< n       */
4  newC1 ← C1[0:p]/* p values between 0 and p-1            */
5  newC2 ← C2[0:p]
6  newC1 ← newC1 + C2 \ newC1
7  newC2 ← newC2 + C1 \ newC2
8  return newC1 , newC2
```

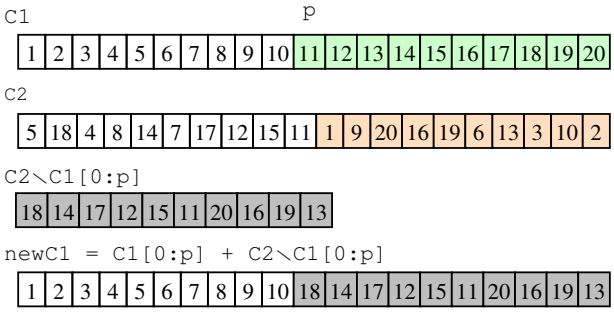


Figure 6: `OnePointCrossOver(C1, C2)`

point is set to p. Chromosome $newC1$ takes the head of $C1$ and orders the remaining tasks according to the task order of $C2$. For instance, task 5 that is already present in the head of $C1$ is not duplicated in the tail of $newC1$ and thus task 18 appears first in the second part of the chromosome.

The 1-point crossover algorithms are named 1pX. With the two selection operators we have 1pX-R, for random selection, and 1pX-W, for wheel selection.

e) Two point crossovers: Considering that the 1-point crossover generates large changes in the chromosomes we have also implemented two other crossover operators that use two points.

The first operator is a two point crossover called Order Crossover (OX). Two crossover points are randomly drawn.Each parent keeps its middle part $C_i[p_1 : p_2]$. Then, the genes of its edges starting from $C_i[p_2 + 1]$, circling back to $C_i[p_1 - 1]$ are ordered according to their order in the other parent starting from $p_2 + 1$. The Order Crossover operator is given by Algorithm 3.

For example in Figure 7, the first new off-spring cC_1 is composed of the middle part of the first parent $C_1[p_1 : p_2]$. The subset of the rest of the genes of C_1 starting from $C_1[p_2 + 1]$: [18,19,20,1,2,3] is reordered as these genes appear in $C_2[p_2 + 1] \rightarrow C_2[p_2]$: [3,2,18,1,20,19]. The ordered subset is then added to the first off-spring cC_1 in the same circular manner.

The algorithms that use this crossover operator are named OX. We thus have OX-R and OX-W depending on the associated selection operator.

The second 2-point crossover operator is a classical two point crossover. Each off-spring has the same edges as its parent $C_i[0 : p_1 - 1]$ and $C_i[p_2 + 1 : n]$. The genes in the middle of each parent $C_i[p_1 : p_2]$ are reordered in the off-spring according to their order in the other parent. This operator is detailed in Algorithm 4. Figure 8 illustrates an example of this operator. The operator is named Middle Cross Over and the

Algorithm 3: `orderCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks                    */
1        C2 /* chromosome 2 with n tasks                    */
   Result: newC1, newC2: 2 new chromosomes each with n tasks
2  n ← length(C1)
3  p1 ← ⌊n × rand(0, 1) × 0.15⌋
4  p2 ← ⌊n × (rand(0, 1) × 0.15 + 0.85)⌋
5  newC1 ← C1[p1:p2]
6  newC2 ← C2[p1:p2]
7  temp1, temp2 ← [], []
8  for i = 0 to n − 1 do
9      if C2[(i + p2)%n] ∉ newC1 then
10         temp1.append(C2[(i + p2)%n])
11     if C1[(i + p2)%n] ∉ newC2 then
12         temp2.append(C1[(i + p2)%n])
13 newC1 ← temp1[n−p2:n−p2+p1] + newC1 + temp1[0:n−p2]
14 newC2 ← temp2[n−p2:n−p2+p1] + newC2 + temp2[0:n−p2]
15 return newC1 , newC2
```

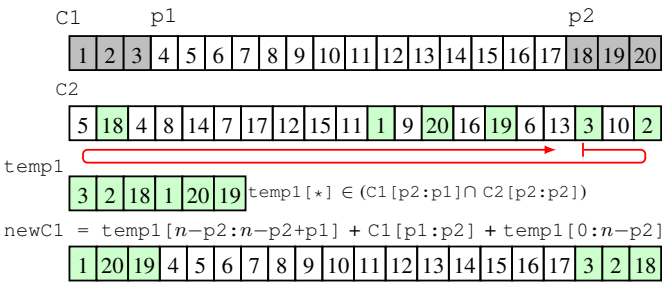


Figure 7: `orderCrossOver(C1, C2)`

corresponding algorithm identified as MX (MX-R and MX-W with their selection operators) in the following.

V. Experiment and Results

To assess the performance of the presented algorithms we have implemented them and we have run simulations with varying parameters. The algorithms and the simulation testbed have been developed in python[2].

The experimental settings are the following. The number of available cores is set to 8, which means that we cannot run more than 8 tasks in parallel even if there is still unused power. The performance of the algorithms is computed with different values of mean task length, from 10 to 100 with a step of 10, and mean task power need, from 4 to 40 with a

[2]The source code is available on GitHub at http://github.com/laurentphilippe/greenpower

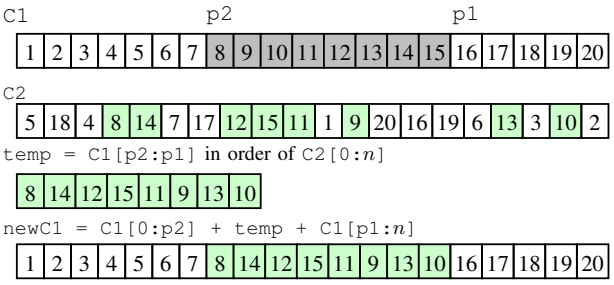


Figure 8: `middleCrossOver(C1, C2)`

Algorithm 4: `middleCrossOver(C1, C2)`

```
   Data: C1 /* chromosome 1 with n tasks                    */
1        C2 /* chromosome 2 with n tasks                    */
   Result: newC1, newC2: 2 new chromosomes each with n tasks
2  n ← length(C1)
3  p1 ← intRand(0, n)/* integer random value: 0 ≤p1< n     */
4  p2 ← intRand(0, n)
5  if p1 < p2 then
6      newC1, newC2 ← orderCrossOver(C1, C2)
7  else
8      if p1 = p2 then
9          newC1 ← mutation(C1)
10         newC2 ← mutation(C2)
11     else
12         newC1 ← C1[0:p2] /* p2 task indices           */
13         newC2 ← C2[0:p2]
14         for i = 0 to n − 1 do
15             if C2[i] ∈ C1[p2:p1] then
16                 newC1.append(C2[i])
17             if C1[i] ∈ C2[p2:p1] then
18                 newC2.append(C1[i])
19         newC1 ← newC1+C1[p1:n]
20         newC2 ← newC2+C2[p1:n]
21 return newChromo1 , newChromo2
```

step of 4. For each couple of values we run 200 simulations with different sets of 100 tasks and sets of 1000 intervals where $\Phi_{max} = 80$. Note that our tests show that the results become stable starting from 150 executions, thus, we chose 200 to be sure of having reliable results. For the tasks, the p_i and ϕ values are randomly chosen with, respectively, an exponential law and an uniform law. The performance of the algorithms depends on their obtained makespans. An algorithm may however not always get the best result, depending on the experimental parameters. Algorithms are hence rather compared based on their mean makespan. However a simulation with larger task is hardly comparable with another using small tasks. For these reasons we measure the algorithm's performance with their *Permake*, where *Permake* = $(makespan - useless)/\sum p_i$, which normalizes the raw makespan value, where $useless$ is the sum of intervals with $\Phi_x < \min_i(\varphi_i)$, the available power is less than the minimum task power.

We evaluate the GAs with different types of crossovers, without cross over, and evaluate the effect of wheel vs random selection. For each GA the stopping condition (nbI in Algorithm 1, number of generations without any improvement) is set to 50. Note that other computations have shown that a value of 10 gives poorer results.

Figure 9 shows the best algorithm, the one with the lowest mean *Permake*, with each $p_i max$ and $\phi_i max$ values presented as a square on the heat map. From the figure we can say that in general, wheel selection gives better results than random selection. This probably means that better solutions are rather found by slightly modifying initially good solutions than searching at a wider distance from these initial solutions. The only cases where random selection gives better results is when the power demand of tasks is high, 36 or higher. From this figure we can also notice that the 1-point crossover does not show any best result in the heat map. This is because changing a big chunk (90%) of a solution, has almost the same probability of producing a good solution as generating a completely random solution, even if the parent is a good solution. In the same way

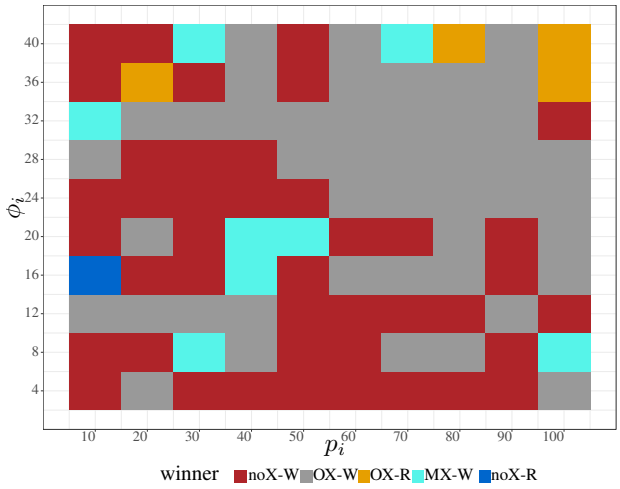


Figure 9: Best algorithm, Permake, 200 executions

Table II: Computation time

Algo	noX-W	OX-W	MX-W	1pX-W	LPT
Time (s)	239.34	301.83	300.11	128.6725	0.02
Algo	noX-R	OX-R	MX-R	1pX-R	2Qs
Time (s)	287.28	356.08	339.58	334.93	0.04

we can state that limiting the size of the chunk to be changed in the crossover leads to better solutions.

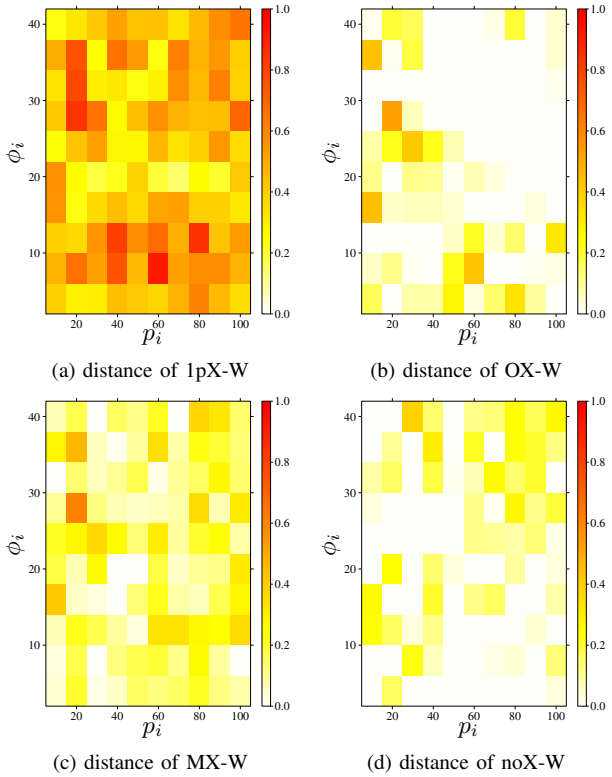
Figure 9 also shows that the characteristics of the tasks play a rule in determining which GA configuration is the best. Since the heat map is relatively split diagonally between two GA configurations, we can conclude that GA with no crossover performs better for small tasks (the rectangle area = $p_i \times \phi_i$ is small), while it is better to use OX for big tasks.

Figure 10 shows the distance in percent of the performance of the four algorithms using wheel selection to the best algorithm for each square in the heat map. We can see that the algorithms are never more than 2% worse than the best solution and that, except for the 1-point crossover, the algorithms are generally less than 0.5% worse than the best one, even for the noX-W algorithm. The same analysis on random selection shows that it does not provide as good results as the wheel one. Note that, even in the cases where random selection provides the best result, OX-W is not farther than 0.3%. We conclude that OX-W may be used in almost all cases.

Figure 11 presents the distance of the list algorithms, used as initial population of the different GAs, to the best *Permake* value. The figure shows how far the GAs are able to improve these initial schedules. Preceding results showed that 2Qs and LPT were the best solutions of the list algorithms which is confirmed here as we observe less red squares on their heat maps. Note that the black squares on the LPN heat map mean that the distance exceed the upper value of 20 %, reaching up to 40 % in some cases. Generally the best improvement of the GAs from those two solutions is around 5% and never better than 10%. This means that all GAs improve the good initial schedules but the improvement is not that significant.

Table II answers the question of the cost of the improvement. It gives mean computation times for each algorithm. Note that the computation times stay almost constant whatever the value of p_i and φ_i. There is a huge difference in the computation times between list algorithms and GAs. If nbI, the number of iterations without improvement is set only to 10, the mean computation time falls to around 25%. The results also show that using random selection leads to bigger computation times, especially for 1-point crossover, where big changes are applied on the chromosomes, and randomly selecting bad individuals for crossover would delay the convergence comparing to selecting good individuals by the wheel selection, and improving them through next generations. Given these results we can see that the list algorithms provide pretty good results for the time used and the improvements are costly. So the question is how many times can we wait for a schedule? Note that the algorithms are implemented in python and run faster in a compiled language.

Further analysis of the results also shows that all tested GAs have a relatively stable standard Permake deviation to each other, around 0.31, with ranging between a minimum around 0.18 and maximum around 0.45. This means that the Permake measure is stable for the GAs. On the other hand list based algorithms have bigger variations and higher standard deviation up to around 0.36.



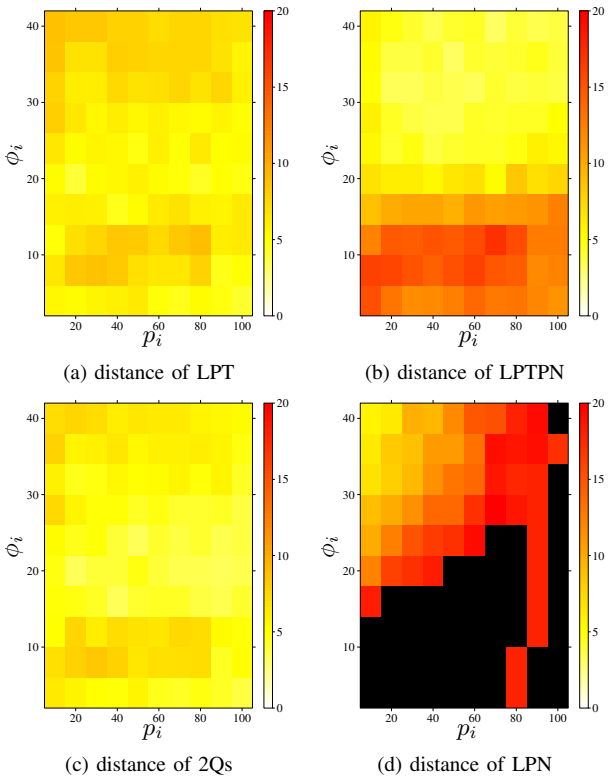
(a) distance of 1pX-W (b) distance of OX-W

(c) distance of MX-W (d) distance of noX-W

Figure 10: Distance of 1pX-W, OX-W, MX-W, and noX-W from the best algorithm

VI. Conclusion

In this work we evaluated the interest of using a genetic algorithm (GA) to find a solution to the optimization problem of scheduling a set of independent tasks on a parallel platform



(a) distance of LPT (b) distance of LPTPN

(c) distance of 2Qs (d) distance of LPN

Figure 11: Distance of list algorithms from the best algorithm

powered solely by renewable energy sources. We examine the effect of applying different types of crossover operators on the performance of GA. We compare the performance of these four GA configurations to previously tested list algorithms. We also investigate the impact of applying the wheel selection in comparison with random selection. Extensive simulations show that implementing any tested GA configuration outperforms all tested list algorithms, by improving known good solutions through small size genetic modifications. Even though the superiority in the performance of GA is not proportional to the time loss compared to list scheduling algorithms, the computation time of GA is still within acceptably limits.

In future works we intend to explore GA configurations, to reduce the computation time cost and to investigate other strategies than GA. Meanwhile, we are developing experimental settings that will allow us to measure the distance of our solutions to the optimal solution. We also plan to assess the GA for other optimization objectives. For instance the flowtime, i.e., the job slowdown time, is an important user oriented objective that we will study. It however requires the design of specific genetic operators to match the objective.

ACKNOWLEDGMENT

This work was supported in part by the ANR DATAZERO (contract "ANR-15-CE25-0012") project and by the Labex ACTION project (contract "ANR-11-LA BX-01-01"). Computations have been performed on the supercomputer facilities of the Mésocentre de calcul de Franche-Comté – Besançon.

REFERENCES

[1] Imtiaz Ahmad and Muhammad K Dhodhi. Multiprocessor scheduling in a genetic paradigm. *Parallel Computing*, 22(3):395–406, 1996.

[2] Martin Arlitt, Cullen Bash, Sergey Blagodurov, Yuan Chen, Tom Christian, Daniel Gmach, Chris Hyser, Niru Kumari, Zhenhua Liu, Manish Marwah, et al. Towards the design and operation of net-zero energy data centers. In *Thermal and Thermomechanical Phenomena in Electronic Systems (ITherm), 2012 13th IEEE Intersociety Conference on*, pages 552–561. IEEE, 2012.

[3] ANR Datazero. http://datazero.org.

[4] Bernabé Dorronsoro, Sergio Nesmachnow, Javid Taheri, Albert Y Zomaya, El-Ghazali Talbi, and Pascal Bouvry. A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. *Sust. Computing: Informatics and Systems*, 4(4):252–261, 2014.

[5] Pierre-François Dutot, Yiannis Georgiou, David Glesser, Laurent Lefevre, Millian Poquet, and Issam Rais. Towards energy budget control in hpc. In *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 381–390. IEEE Press, 2017.

[6] Saurabh Kumar Garg, Chee Shin Yeo, Arun Anandasivam, and Rajkumar Buyya. Environment-conscious scheduling of HPC applications on distributed cloud-oriented data centers. *Journal of Parallel and Distributed Computing*, 71(6):732 – 749, 2011.

[7] Y. Georgiou, D. Glesser, and D. Trystram. Adaptive resource and job management for limited power consumption. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pages 863–870, May 2015.

[8] Íñigo Goiri, Md E Haque, Kien Le, Ryan Beauchea, Thu D Nguyen, Jordi Guitart, Jordi Torres, and Ricardo Bianchini. Matching renewable energy supply and demand in green datacenters. *Ad Hoc Networks*, 25:520–534, 2015.

[9] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5(2):287–326, 1979.

[10] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *International Parallel and Distributed Processing Symposium Workshop (IPDPSW)*, pages 896–904. IEEE, 2015.

[11] Johannes Hofmann, Dietmar Fey, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Analysis of intel's haswell microarchitecture using the ecm model and microbenchmarks. In *International Conference on Architecture of Computing Systems*, pages 210–222. Springer, 2016.

[12] A. Kassab, J. M. Nicod, L. Philippe, and V. Rehn-Sonigo. Scheduling independent tasks in parallel under power constraints. In *46th International Conference on Parallel Processing (ICPP)*, pages 543–552, 2017.

[13] Yacine Kessaci, Mohand Mezmaz, Nouredine Melab, El-Ghazali Talbi, and Daniel Tuyttens. Parallel evolutionary algorithms for energy aware scheduling. In *Intelligent Decision Systems in Large-Scale Distributed Environments*, pages 75–100. Springer, 2011.

[14] Hongtao Lei, Rui Wang, Tao Zhang, Yajie Liu, and Yabing Zha. A multi-objective co-evolutionary algorithm for energy-efficient scheduling on a green data center. *Computers & Op. Research*, 75:103–117, 2016.

[15] Chandrakant Patel, Ratnesh Sharma, Cullen Bash, and Sven Graupner. Energy aware grid: Global workload placement based on energy efficiency. In *ASME 2003 International Mechanical Engineering Congress and Exposition*, pages 267–275, 2003.

[16] Andrei Tchernykh, Luz Lozano, Pascal Bouvry, Johnatan E Pecero, Uwe Schwiegelshohn, and Sergio Nesmachnow. Energy-aware online scheduling: Ensuring quality of service for iaas clouds. In *High Performance Computing & Simulation (HPCS), 2014 International Conference on*, pages 911–918. IEEE, 2014.

[17] Silvana Teodoro, Andriele Busatto do Carmo, Daniel Couto Adornes, and Luiz Gustavo Fernandes. A comparative study of energy-aware scheduling algorithms for computational grids. *Journal of Systems and Software*, 117:153–165, 2016.

[18] Lizhe Wang, Samee U Khan, Dan Chen, Joanna KołOdziej, Rajiv Ranjan, Cheng-Zhong Xu, and Albert Zomaya. Energy-aware parallel task scheduling in a cluster. *Future Generation Computer Systems*, 29(7):1661–1670, 2013.

[19] Lizhe Wang, Gregor Von Laszewski, Jay Dayal, and Fugang Wang. Towards energy aware scheduling for precedence constrained parallel tasks in a cluster with dvfs. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pages 368–377. IEEE Computer Society, 2010.

[20] Liang Zhang, Tao Han, and Nirwan Ansari. Renewable energy-aware inter-datacenter virtual machine migration over elastic optical networks.

In *Cloud Computing Technology and Science (CloudCom), 2015 IEEE 7th International Conference on*, pages 440–443. IEEE, 2015.