ISSN: 1803-3814 (Printed), 2571-3701 (Online)
https://doi.org/10.13164/mendel.2019.1.179

MENDEL Soft Computing Journal

INTERRC: AN INTER-RESOURCES COLLABORATION HEURISTIC FOR SCHEDULING INDEPENDENT TASKS ON HETEROGENEOUS DISTRIBUTED ENVIRONMENTS

Abdelhamid Khiat[1,2,✉], **Abdelkamel Tari**[2]

[1]Networks and Distributed Systems Division, Research Center on Scientific and Technical Information, Algeria
[2]Faculty of Exact Sciences, University of Bejaia, Algeria
a.khiat@dtri.cerist.dz[✉], abdelkamel.tari@univ-bejaia.dz

Abstract
The independent task scheduling problem in distributed computing environments with $makespan$ optimization as an objective is an NP-Hard problem. Consequently, an important number of approaches looking to approximate the optimal $makespan$ in reasonable time have been proposed in the literature. In this paper, a new independent task scheduling heuristic called InterRC *is presented. The proposed* InterRC *solution is an evolutionary approach, which starts with an initial solution, then executes a set of iterations, for the purpose of improving the initial solution and close the optimal $makespan$ as soon as possible. Experiments show that* InterRC *obtains a better $makespan$ compared to the other efficient algorithms.*

Keywords: *distributed computing, scheduling, makespan, evolutionary algorithms.*

Received: 29 April 2019
Accepted: 17 June 2019
Published: 24 June 2019

1 Introduction

Nowadays, most computing architectures are distributed, like Cloud, Grid and High-Performance Computing (HPC) environment [13]. This kind of architectures can be used to achieve a hard computing, which takes longer time when executed on only one computer. This is why it is important to execute this work on a parallel architecture sustained by a significant number of computing resources.

A work is composed of several tasks, each one of them representing a program unites that cannot be divided. In a given work, the necessary time to finish the last task is called *makespan*. In order to ensure the execution of a work in a distributed environment, a set of resources must be allocated, then, each task of the work must be assigned to one of these allocated resources. The latter are often of heterogeneous nature, which means that the execution time of a task changes from a resource to another. Subsequently, the *makespan* changes according to the used mapping. Therefore and in order to ensure a better mapping of the set of tasks to the set of allocated resources, a power scheduler must be used.

The problem of scheduling remains one of the more important challenges in distributed computing environments. In general, the scheduling problems are classified as NP-Hard [6], which means that there is no known general solution for which can get the optimal value of the *makespan* in a time that is polynomial in the problem size. As a Consequence, an important number of approaches have been proposed in the literature, these approaches envisages to find a mapping and come close to the optimal *makespan* in reasonable time.

Evolutionary approaches are considered as an important way to solve this problem of scheduling. Usually, in an evolutionary approach, the value of *makespan* improved over time, by starting from a solution called *initial solution* which is then improved through iterations until reaching one or some conditions called *end conditions*.

In this article, a new evolutionary heuristic is exposed. It tries to find a best mapping of a set of tasks to a set of heterogeneous resources in a distributed environment with the objective of *makespan* minimization. The proposed approach uses a new concept called *InterRC*: a given solution can evolve to a better solution using some operators which will be presented in details in Section 3. We assume that the tasks of a given work are independent, non preemptive, and with same static priorities.

This article is organized as follows. Section 2 composed of two subsections: the first one presents the related work based on fast deterministic heuristics, and the second one presents the related work based on evolutionary heuristics, including the presentation of some scheduling algorithms that has the *makespan* optimization as objective. Section 3 presents the used model and the details of the proposed heuristic named *InterRC*. Then, Section 4 presents the evaluation of *InterRC*, including a comparison of *InterRC* with some others heuristics. Finally, Section 5 gives the conclusion and the future work.

2 Related Work

The complexity of task scheduling problem in distributed environments when it comes to find the optimal *makespan* is NP-Hard in general [6], consequently, there is a lack of solutions that can find the optimal *makespan* in a reasonable time, especially when the problem size increases. An important number of approximate approaches that address the problem have been proposed in the literature, these approaches envisage to find in reasonable time a solution near as possible to the optimal solution. A set of non exhaustive works will be presented in the remainder of this section. The presentation will be done in two subsections. In the first one, a set of fast deterministic heuristics will be presented, then, in the second one, some evolutionary approach will be introduced.

2.1 Related Work Based on Fast Deterministic Heuristics

Max-Min [5] algorithm consists to execute a set of iteration. The iteration process consists of selecting the task that has the biggest completion time, and then affects it to the resource that gives the minimum execution time. It subsequently repeats this process until the end of scheduling all tasks. After each iteration, the completion time is updated for each task that is not yet executed.

In the *Min-Min* [5] scheduling process, an iteration consists of selecting the resource that has the minimum value of completion time, then, the task that has the minimum execution time on this resource is selected, as *Max-Min*, after each iteration the completion time is updated for all tasks not yet mapped.

Min-Max [9] heuristic works as *Max-Min* and *Min-Min* approach, that schedules one task in each iteration, until the scheduling of all tasks. At each iteration, the minimum completion times of all unassigned tasks over all available resources are computed. Then, for each unassigned task, the ratio of its minimum execution time on all resources to the execution time on the processor that resulted to the minimum completion time is computed. The task that has the highest value of this ratio is removed from the list of unassigned tasks and scheduled to the resource that gives the minimum completion time.

Sufferage algorithm [10] executed in iterations where each iteration is composed of two processes. The first one consists to compute for each task a value called $sufer$, which represents the difference between the first and the second minimum execution time of the concerned task. While the second one allows to affect the task with maximum $suffer$ to the resource that gives the minimum completion time. These two processes are repeated until the end of the assignment of all tasks.

LSufferage proposed in [7] is inspired from *Sufferage*. In *LSufferage* algorithm, a static descending ordered list is generated for each possessor p, each element of the generated lists contains the task identifier and an associated value obtained by computing the ratio between the maximum execution time of the concerned tasks T and its execution time on p if the execution time of T on p is not the maximum execution time, otherwise, the value is calculated by the division of the execution time on the second fastest possessor on the maximum execution time (p). Finally, the scheduler bases on these values to schedule each task to the processor according to their priority (computed ratios).

An algorithm called Relative Cost algorithm (RC) was proposed in [15]. RC utilizes an indicator called Relative Cost (RC). According to authors, RC retains the advantages of the *Min-Min* algorithm regarding *makespan*, and balances the load very well. The task and resource that will be selected in each iteration is based on two quantities: the static relative cost and the dynamic relative cost. The static relative cost is computed once at the start of the algorithm as rate between the execution time of this task on this resource to the average of its execution time on all available resources. The dynamic relative cost is computed before each task is scheduled as rate of the completion time of task on the resource to the average of its completion time on all available resources.

Round Robin (RR) algorithm is a simple heuristic with low complexity, which remains largely used in a significant number of algorithms, particularly, in the real deployed algorithms. RR algorithm affects the first task in the set of not affected tasks to the first available found resource, until affectation of all tasks.

2.2 Related Work Based on Evolutionnar Approach

Genetic Algorithm (GA) [8] is a popular meta-heuristic, considered as an evolutionary approach works in polynomial time, GA is inspired by the biologic process of the natural selection, in a standard GA, the algorithm starts with an initial population, where this latter is composed of a set of solutions called individuals (chromosomes), the initial population known as first generation as inspired from the biologic language, the GA looks to improve the initial population by applying two main operators $Crossover$ and $Mutation$, the new population called new generation. The passage from a generation to another is called iteration, $Crossover$ operator consists to exchange a parts (gens) of two selected individuals, while $Mutation$ operator consists to alter one or more gens with a given probability called crossover probability, usually the value of the latter is low.

A fitness function is used in the selection processes, by giving to best individual a high probability to continue its existence in the next generation, while the individuals with lower fitness values will have a high probability to be dropped out. Then, a stop criteria is used to make an end to the algorithm, that can be a fix number of generations, non evolution in the result after a number of generation, or a fixed time of execution of the algorithm.

Ant Colony Optimization (ACO) and Particle Swarm Optimization (PSO) are two other evolutionary heuristics inspired by living world. The first one [3] is inspired by the foraging behavior of ants, while the second one [14] is inspired by the behaviour of the particles working in collaboration in the swarm.

CHC [4] is a specialization of the traditional *GA* that uses an elitist selection strategy that tends to keep the best individuals in the population. CHC algorithm proposed originally in [4] is an evolutionary algorithm, CHC algorithm uses a special recombination called Half Uniform Crossover (HUX), which randomly swaps exactly half of the bits that differ between two selected solutions (parents). These latter are usually the individuals with a difference in 1/4 bits of the chromosome length for the first generation, and this number reduced by one 1 each time that no offspring is inserted into the new population, the new offspring must compete with their parents for survival. Note that the mutation operator is not used in CHC algorithm.

In [16], a Cellular Memetic Algorithm (*CMA*) was applied for solving the rescheduling problem in the case of batch jobs. The proposed approach shows generated solutions of good quality and a short execution time of the rescheduling procedure.

An other implementation of *GA* is proposed in [12]. The authors use the *GA* in a conventional cluster, in which a millicomputers was added to reduce power consumption. This algorithm is called *PA-CGA* (Parallel Asynchronous - Cellular Genetic Algorithm) and was designed to schedule independent tasks on a distributed system.

3 Proposed Solution

In this section we introduce the proposed heuristic, by first presenting the Heterogeneous Computing Scheduling Problem (HCSP) formulation with some considerations about the used execution time estimation (ETC) mode, and then describing the details of the proposed *InterRC* heuristics.

3.1 Problem Description

The problem addressed in this paper can be described as follows: a set of tasks $T = \{T_0, ..., T_{n-1}\}$ must be mapped to a set of resources $R = \{R_0, ..., R_{m-1}\}$, where n represents the total number of tasks to be scheduled and m represents the total number of available resources allocated to execute the whole set of tasks T. Assuming that a matrix ET of $n * m$ elements is defined, where each entry $ET[i][j]$ of the matrix ET represents the execution time of the task T_i on the resource R_j whatever $T_i \in T$ *and* $R_j \in R$.

It is assumed that all tasks to be scheduled are independents and non-preemptive, i.e. There is no dependency relationship between the tasks and whatever T_i, T_i cannot stop once started. In addition to these two conditions (independence and non-preemptively), all tasks are of the same priority, that is, for each couple (T_i, T_j) in T, T_i and T_j have the same chance to be scheduled first.

The model used to estimate the execution time of each task T_i in T on each resource R_j in R is the one defined by Ali et al [1]. This model is considered as one of the most widely used models for HCSP. When generating the matrix ET this model takes into account three properties: machine heterogeneity, task heterogeneity, and consistency.

Machine heterogeneity represents the relation between resources in term of computing power, which results in a variation of execution time. In a high machine heterogeneity HCSP systems, the difference between the execution time of a task T_i from a resource to another is high. On the other hand, in low machine heterogeneity, the difference between execution times is low. The task heterogeneity represents the difference of computing power needs from a task to other. Subsequently, in a high task heterogeneity HCSP, there is a high difference in term of execution time from a task to another on a given resource R_j. In contrast, for a low task heterogeneity HCSP this difference remains low. The third classification used in HCSP is the consistence: in a consistent ETC, if a task T_i is slower than a task T_j on R_j, then T_i is slower than a T_j on all other machines. An ETC is inconsisrent if it is possible to find two tasks T_i and T_j such that T_i is slower than T_j on some machines and T_j is slower than T_i on other machines. Moreover, a semi-consistent ETC can be used to model those inconsistent systems that include a consistent subsystem.

3.2 InterRC Heuristic

In this sub-section, the different elements of understanding the *InterRC* heuristic, as well as the process flow, will be presented in detail in five sub-sections.

3.2.1 Objective

The main goal of *InterRC* heuristic consists of optimizing the *makespan* of such HCSP described above, where *makespan* represents the total time needed to complete the execution of all the tasks, which can be computed using Formula 1. In the Formula, $ET[i][j]$ represents the execution time of T_i on R_j as already described in the previous subsection and b_{ij} is a boolean value which is equal to 1 if T_i is affected to R_j, otherwise b_{ij} equals to 0.

$$\begin{aligned} makespan &= max_{j=1...m} \sum_{i=0}^{n} ET[i][j] * b_{ij} \\ With, b_{ij} &= \begin{cases} 1 & if\ T_i\ executed\ on\ R_j \\ 0 & else \end{cases} \end{aligned} \tag{1}$$

3.2.2 Operators

The following two operators are defined, and used by InterRC heuristic.

move: consists to test if the value of *makespan* obtained after a move of a task T_i from a resource R_{j1} to another resource R_{j2} will not exceed the actual *makespan*. If that is the case, the move is applied, otherwise, it will not be applied.

permutation: consists in checking if the *makespan* after a permutation between two distinct tasks T_{i1} and T_{i2} situated on two different resources R_{j1} and R_{j2} will not exceed the actual *makespan*, then the permutation is applied in case the test proves to be true, if not, it is not applied.

3.2.3 End Conditions

The *InterRC* heuristic comes to an end if one of the two following conditions is reached: The time $MaxResTime$ which represents the maximum time dedicated to the execution of *InterRC* algorithm is reached, or the loop $L1$ presented in both algorithms (Algorithm 3 and Algorithm 4) ends with any improvement of the actual *makespan* in the case of Algorithm 3, or with any redistribution possible in the case of Algorithm 4.

3.2.4 Global Process

As described in Algorithm 1, the global process of *InterRC* heuristic consists to three phases: The first one aims to generate an initial solution, while the second one looks to improve the actual *makespan*. Finally, the third phase allows to redistribute the set of tasks on the set of available resources without exceeding the actual value of *makespan*. Note that the second and third phases are executed alternatively with a fixed number of times ($NbrIterations$). The alternation allows a maximum redistribution, and avoid the system stability as much as possible. The stability means the difficulty to find any moving/permutation that can improve the actual *makespan*. The second and third phases can be achieved using *permutation*/*move* operators.

```
Call Algorithm 2
i=1;
while NotEndExecTime AND improve = true AND i ≤ NbrIteration do
    i++;
    Call Algorithm 3;
    Call Algorithm 4;
end
Return Task affectation;
```

Algorithm 1: Global *InterRC* Process

3.2.5 Detailed Process

The details of the three phases of *InterRC* heuristic are:

A. Initial affectation: This first phase described by Algorithm 2 consists of generating an initial solution, which starts by choosing randomly a resource R_{init} and a task T_{init}. Then, the loop $L1$ of Algorithm 2 is called to assign all tasks to R_{init} starting by T_{init}. Note that the choice of R_{init} and T_{init} has an impact on the final *makepan*.

Choose randomly a resources R_{j_init} AND a task T_{i_init}
for *($ii = 0; ii < n; ii + +$)* **do**
  $i = (ii + i\_init)\%n$;
  Assign T_i to R_{j_init} ;
end
Return Task affectation;

Algorithm 2: Initial affectation

B. *makespan* improvement: In this second phase of *InterRC* heuristic, which is described by Algorithm 3, a loop of the set of tasks affected to R_{jMS} is done (R_{jMS} represents the resource that gives the *makespan*). Then, for each found task T_{ims}, the algorithm loops the set of other resources (other than R_{jMS}) and for each found resource R_j with $R_j != R_{jMS}$, the algorithm tests in the first time if the *move* of T_{ims} from R_{jMS} to R_j improves the actual *makespan*; if the test is positive, the *move* operation is realized, the new *makespan* is computed, and loop $L1$ broken with the update of *improve* value to true. Otherwise, the loop $L3$ are triggered with the aim of finding a task T_{i1} that can improve the actual *makespan* if it is permuted with T_{ims}, as soon as a task $Ti1$ is found, the *permutation* is realized, the new *makespan* is calculated, and the loop $L3$ is broken with the update of *improve* value to true.

$improve = true$
L0: **while** *($ResTime < Max\_ResTime\ AND\ improve = true$)* **do**
  $improve = false$
  L1: **forall** $T_{ims} \in R_{jMS}$ **do**
    L2: **forall** ($R_j \in R\ AND\ R_j \neq R_{ms}$) **do**
      if ($Move(T_{ims}\ to\ R_j)\ improves\ MS$) **then**
        $Move(T_{ims}, R_j)$; $MS = Get\_New\_MS$; $improve = true$;
        $break\ L1$;
      else
        L3: **forall** *($T_i \in R_j\ AND\ R_j != R_{jMS}$)* **do**
          if $Permut(T_i, T_{ims})\ improves\ MS$ **then**
            $Permute(T_i, T_{ims})$; $MS = Get\_New\_MS$; $improve = true$;
            $break\ L1$;
          else
          end
        end
      end
    end
  end
end
Return Task affectation;

Algorithm 3: *makespan* improvement

C. Task redistribution: Algorithm 4 describes this third phase of *InterRC* heuristic. Unlike Algorithm 3, Algorithm 4 will not try the *move* of a task from R_{jMS}, or the *permutation* of a task situated on R_{jMS} with another task located on another resource other than R_{jMS}. But rather, the tests will be larger. In more details, the loop $L1$ allows to loop the set of resources R, then, for each found resource R_{j1}, a loop of the tasks affected to R_{j1} is done. Afterward, the algorithm re-loop the set of resource and for each found resource R_{j2}. The algorithm test in the first time if the *move* of T_{i1} to R_{j2} will not penalize the actual value of *makespan*, then the *move* is applied in case the test proves to be true, if not, the loop $L3$ are triggered in order to find a task T_{i2} that does not penalize actual *makespan* if it is permuted with T_{i1}, as soon as a permutation is possible, the *permutation* is realized, the new *makespan* is calculated, and the loop $L3$ is broken with the update of *improve* value to true.

In this third phase, each task can be moved/permuted only once, to ensure this uniqueness of *move/permutation*, a set called Ts_Tested is created, then, the tasks candidate to moving/permutation operators are added to this set. Thereafter, the tasks of the set Ts_Tested cannot re-participate to another *move/permutation* operations. The use of Ts_Tested set allow to finish the phase fast as possible, with maximum *move/permutation* possible.

```
improve = true
L0: while (ResTime < Max_ResTime AND improve = true) do
    forall T_j1 ∈ R do
        L1: forall T_i1 ∈ R_j1 do
            if (! T_i1 ∉ Ts_Tested) then
                Ts_Tested.add(T_i1);
                L2: forall (R_j2 ∈ R AND R_j2 ≠ R_i1) do
                    if (Move(T_i1 to R_j2) improvesMS) then
                        Move(T_i1, R_j2); MS = Get_New_MS; improve = true;
                        break L1;
                    else
                        L3: forall (T_i2 ∈ R_j2) do
                            if Permut(T_i1, T_i2) improves MS then
                                Permute(T_i1, T_i2);  MS = Get_New_MS; improve = true;
                                break L1;
                            else

                            end
                        end
                    end
                end
            else

            end
        end
    end
end
Return Task affectation;
```

Algorithm 4: Task redistribution

4 Experiments

In order to evaluate *InterRC* approach, a simulator was developed using Java, R and shell. Then, the proposed *InterRC* algorithm was implemented and integrated to the developed simulator using the *Java* language. The phase of evaluation was realized on a PC with Processor $i7$ and $8GO$ of RAM, on which, a set of experiments was done, then the results of *InterRC* algorithm are compared with a set of algorithms, already presented in the Section 2.

The proposed *InterRC* was compared with two kinds of heuristics: the first one is the fast deterministic heuristics, which are characterized by a low execution time and give the same result even if we repeat the execution of the algorithm many times. While the second kind is the evolutionary approaches characterized by good results. The chosen fast deterministic heuristics are *RC* [15], *Sufferage* [10], *Min-Max* [9], and *Min-Min* [9]. While the used evolutionary heuristics are *cMA* [16], *GA* [8], *PA-CGA* [12] and *CHC* [4].

This evaluation was made on the 12 classic problem instances proposed by Braun et al in [2], each instance has 512 tasks and 16 machines.

The execution of *InterRC* algorithm has been redone several times, then, the best obtained results are presented in Table 2 and Table 1. Table 2 shows the comparison of the *makespan* obtained by *InterRC* algorithm with the evolutionary algorithms, while Table 1 compares the obtained *makespan* with the best known and fast deterministic heuristics.

The last column (LP Bound) of both tables (Table 1 and Table 2) corresponds to the lower bound for the *makespan* value, which can be computed by solving the linear relaxation for the preemptive case using a linear programming solver [11]. And the grey fields in these both tables Indicate that the corresponding value is less than the value obtained by our algorithm, that means, the makespan is not improved using our proposed algorithm.

Three parameters can change the resulting *makespan*. The algorithm's execution time, the values of R_{j_init} and T_{i_init}, and the value $NbrIterations$. In our implementation, the fixed time for *InterRC* execution is $20s$ for all tests, R_{j_init} and T_{i_init} values are randomly selected in each test and the $NbrIterations$ was fixed to 4 each one (redistribution/improvement) executed for $5s$ to get $4*5 = 20s$ that the total execution time dedicated to *InterRC*.

In order to study the speed of the *makespan* evolution as a function of time when running the *InterRC* heuristic, the following process is followed:

Dataset	Min-Min	Min-Max	RC	Sufferage	LSufferage	InterRC	LP Bound
A.u˙c˙hihi.0	8460675.0	8205561.3	9576839.0	10249172.9	8092234.8	7434522,4	7346524.2
A.u˙c˙hilo.0	161805.4	161686.8	163200.2	168982.6	160100.3	154111,9	152700.4
A.u˙c˙lohi.0	275837.4	279907.7	309192.7	337121.5	255070.3	241582,7	238138.1
A.u˙c˙lolo.0	5441.4	5485.4	5542.6	5658.5	5487.4	5174,3	5132.8
A.u˙i˙hihi.0	3513919.3	3066454.8	3447651.4	3306818.9	3436518.1	2985279,8	2909326.6
A.u˙i˙hilo.0	80755.7	75711.6	76471.5	77589.1	77998.5	74194,2	73057.9
A.u˙i˙lohi.0	120517.7	108533.3	126002.4	114578.9	112400.9	104378,5	101063.4
A.u˙i˙lolo.0	2785.6	2613.5	2677.0	2639.3	2735.7	2569,4	2529.0
A.u˙s˙hihi.0	5160342.8	4627988.8	5068011.5	5121953.6	4394021.6	4216710,5	4063563.7
A.u˙s˙hilo.0	104375.2	100128.4	101739.6	102499.9	100813.8	97782,8	95419.0
A.u˙s˙lohi.0	140284.5	133039.3	143491.2	150297.1	134568.5	123327,7	120452.3
A.u˙s˙lolo.0	3806.8	3555.2	3679.6	3846.5	3695.8	3486,6	3414.8

Table 1: Makespan comparaison with deterministic heuristics

Instance	cMA	GA	PA-CGA	CHC	MA + TS	InterRC	LP Bound
A.u˙c˙hihi0	7700930	7659879	7437591	7599288	7530020	7434522,4	7346524.2
A.u˙c˙hilo0	155335	155092	154393	154947	153917	154111,9	152700.4
A.u˙c˙lohi0	251360	250512	242062	251194	245289	241582,7	238138.1
A.u˙c˙lolo0	5218	5239	5248	5226	5174	5174,3	5132.8
A.u˙i˙hihi0	3186665	3019844	3011581	3015049	3058475	2985279,8	2909326.6
A.u˙i˙hilo0	75857	74143	74477	74241	75109	74194,2	73057.9
A.u˙i˙lohi0	110621	104688	104490	104546	105809	104378,5	101063.4
A.u˙i˙lolo0	2624	2577	2603	2577	2597	2569,4	2529.0
A.u˙s˙hihi0	4424541	4332248	4229018	4299146	4321015	4216710,5	4063563.7
A.u˙s˙hilo0	98284	97630	97425	97888	97177	97782,8	95419.0
A.u˙s˙lohi0	130015	126438	125579	126238	127633	123327,7	120452.3
A.u˙s˙lolo0	3522	3510	3526	3492	3484	3486,6	3414.8

Table 2: Makespan comparaison with evolutionary heuristics

The value of the best *makespan* (*best_ms*) given by the *Min-Min*, *Max-Min*, *RC* and *Sufferage* is calculated before launching the *InterRC* algorithm. Then, at each detection of improvement of the *makespan* value during *InterRC* execution, a ratio *ratio* between this *makespan* (*ms_evol*) and *best_ms* is calculated using the Formula 2.

$$ratio = \begin{cases} \text{(best\_ms/ms\_evol)-1} & if \text{ best\_ms} < \text{ms\_evol} \\ \\ \text{1-(ms\_evol/best\_ms)} & else \end{cases} \tag{2}$$

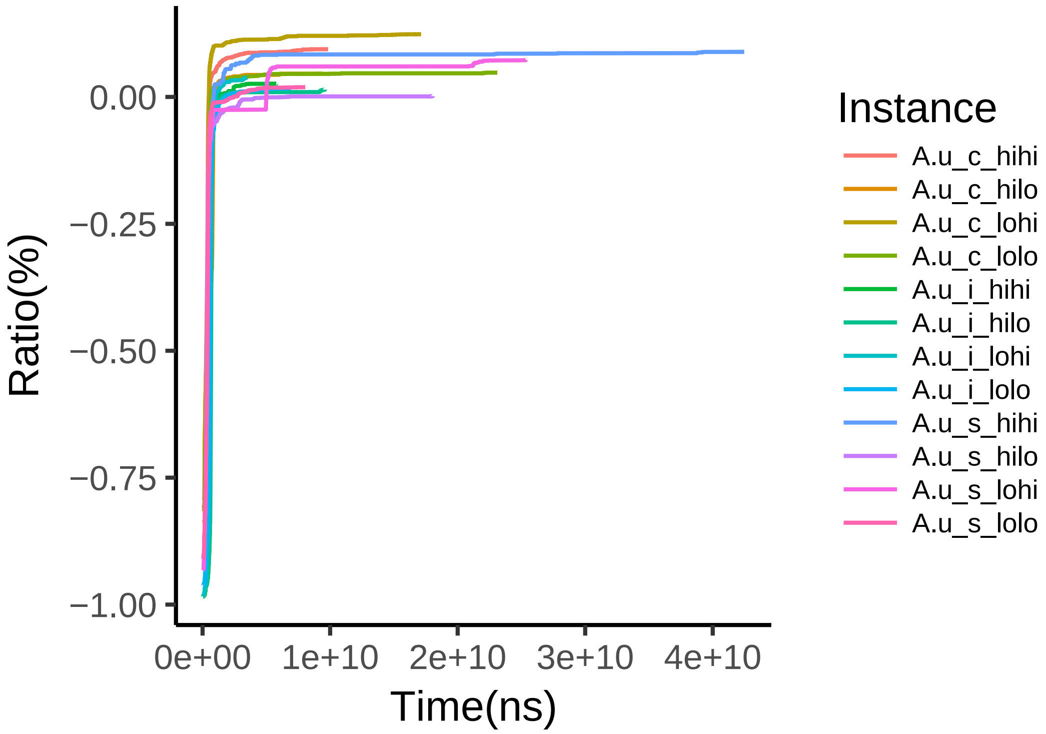
A negative value of *ratio* means that *best_ms* is not yet reached, whilst a positive value of *ratio* means that *ms_evol* is better than *best_ms*. The convergence of the *raio* value to 0 means that *ms_evol* value converges to *best_ms* value. On the contrary, when the value of *ratio* Keep away from 0 to one of the other peak values (1 or -1), that means, the value of *ms_evol*, Keep away from the value of *best_ms*. This move away is to a better result if the peak value is 1, but in other case, the move away is to a bad result.

Fig. 1a, allows to visualize the evolution speed of the *ratio* value obtained for each instance. It shows that the *makespan* improvement through *InterRC* is done quickly at first time, and the *ms_evol* value converges quickly towards the $BestMS$ value, but after some time, the improvement speed starts to become relatively heavy; ultimately, the *makespan* improvement can stop, which makes the continuation of the algorithm execution unnecessary.

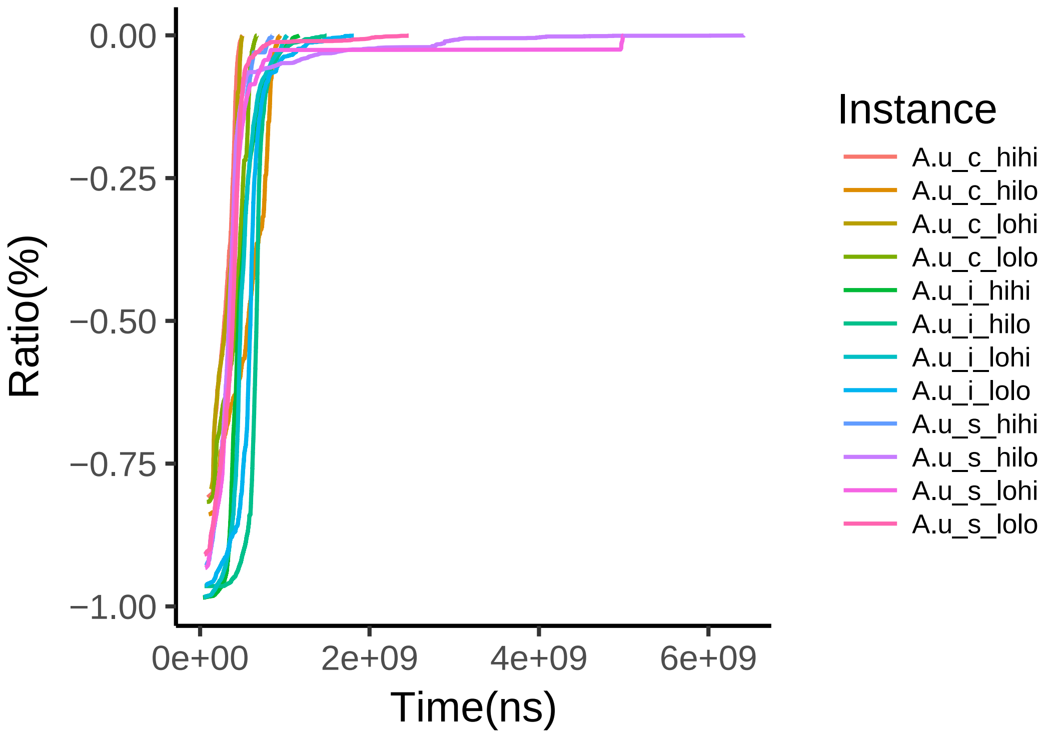
Fig. 1b is zoomed to view in more detail the evolution speed of *ms_evol* before reaching the value of *best_ms*, it is clear that the nature of this evolution can varied from an instance to other, which allows to say that the evolution speed depends on the nature of tasks, as well as resources.

The *gap* value is another parameter used in our evaluation, that represents the relative gap value of any algorithm with respect to the corresponding lower bound, *gap* value is calculated using Formula 3. When, MS_LP_Bound is the *makespan* obtained by lp bound and MS_Algo presents the *makespan* of the algorithm on which we look to calculate the *gap* value.

Fig. 2 shows that average of *gap* value of the evaluated algorithms, the *InterRC* gap value is the best one comparing with all other algorithms with a value equal to 2.013. Fig. 2 shows also that the gap value of evolutionary approaches are the best comparing with all other fast deterministic heuristics.



(a) Evolution of the *ration* value as function of time



(b) Zoom on Fig. 1a

Figure 1: Evolution of the *ration* value as function of time

$$gap = \frac{MS\_LP\_Bound - MS\_Algo}{MS\_LP\_Bound} \quad (3)$$

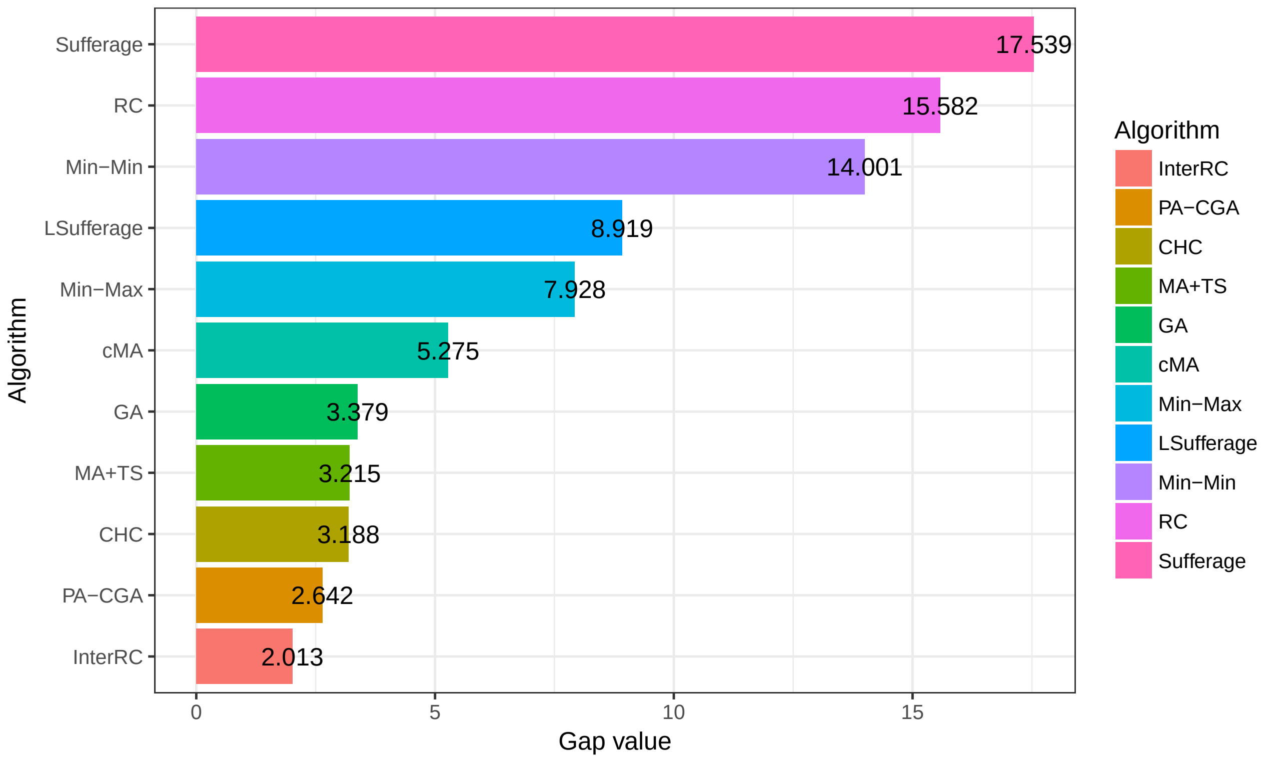


Figure 2: Gap value to the lower bound

5 Conclusion

The Heterogeneous Computing Scheduling Problem (HCSP) that we have addressed in this paper is known as an NP-Hard problem when it comes to optimize the *makespan*. The number of research works examining this problem keeps increasing, especially, with the increased need for computing power. The latter can be achieved through powerful computing architectures like Cloud, HPS, FOG and Grid computing.

In this paper we have proposed *InterRC*, a new evolutionary heuristic that looks to evolve towards the better final *makespan* starting from an initial solution. Then, switch alternatively between two phase (redistribution/improvement) until reaching one of two stop conditions already discussed above. Our experiments phase are achieved by simulation, then different comparisons of the proposed *InterRC* heuristic with others heuristics shows that the proposed approach gives a better *makespan* in about 90 % of cases comparing with evolutionary approaches, and in 100 % of cases comparing with fast deterministic heuristics.

Some directions exist on extending this work: the proposed *InterRC* can be adapted to schedule the dependent tasks, as can be interesting to think about the integration of fault tolerance management aspect where one or more resources failed to continue its work. Another direction could be the incorporation of multi-objective optimization.

References

[1] Ali, S., Siegel, H. J., Maheswaran, M., Hensgen, D., and Ali, S. 2000. Task execution time modeling for heterogeneous computing systems. In *Proceedings 9th Heterogeneous Computing Workshop (HCW 2000)*. Cat. No.PR00556, pp. 185–199. DOI: 10.1109/HCW.2000.843743

[2] Braun, T. D., Siegel, H. J., Beck, N., Boloni, L. L., Maheswaran, M., Reuther, A. I., Robertson, J. P., Theys, M. D., Yao, B., Hensgen, D., and Freund, R. F. 2001. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing* 61, 1, pp. 810–837.

[3] Dorigo, M. and Di Caro, G. 1999. Ant colony optimization: a new meta-heuristic. In *Proceedings of the 1999 congress on evolutionary computation-CEC99*. Cat. No. 99TH8406, IEEE, pp. 1470–1477.

[4] Eshelman, L. J. 1991. The CHC Adaptive Search Algorithm: How to Have Safe Search When Engaging in Nontraditional Genetic Recombination. In *Foundations of Genetic Algorithms*. Vol 1, Elsevier, pp. 265–283. DOI: 10.1016/B978-0-08-050684-5.50020-3

[5] Etminani, K. and Naghibzadeh, M. 2007. A min-min max-min selective algorithm for grid task scheduling. In *2007 3rd IEEE/IFIP International Conference in Central Asia on Internet*. IEEE, pp. 1–7.

[6] Gary, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman and Company, New York, USA.

[7] Gogos, C., Valouxis, C., Alefragis, P., Goulas, G., Voros, N., and Housos E. 2016. Scheduling independent tasks on heterogeneous processors using heuristics and Column Pricing. *Future Generation Computer Systems* 60, pp. 48–66.

[8] Golberg, D. E. 1989. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

[9] Izakian, H., Abraham, A., and Snasel, V. 2009. Comparison of heuristics for scheduling independent tasks on heterogeneous distributed environments. In *2009 International Joint Conference on Computational Sciences and Optimization*. Vol 1, IEEE, pp. 8–12.

[10] Maheswaran, M., Ali, S., Siegel, H. J., Hensgen, D., and Freund, R. F. 1999. Dynamic mapping of a class of independent tasks onto heterogeneous computing systems. *Journal of parallel and distributed computing* 59, 2, 107–131.

[11] Nesmachnow, S., Cancela, H., and Alba, E. 2012. A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing* 12, 2, 626–639.

[12] Pinel, F., Dorronsoro, B., and Bouvry, P. 2010. A new parallel asynchronous cellular genetic algorithm for scheduling in grids. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW)*. IEEE, pp. 1–8.

[13] Sadashiv, N. and Kumar, S. M. D. 2011. Cluster, grid and cloud computing: A detailed comparison. In *2011 6th International Conference on Computer Science Education (ICCSE)*. IEEE, pp. 477–482.

[14] Shi, Y. et al. 2001. Particle swarm optimization: developments, applications and resources. In *Proceedings of the 2001 Congress on Evolutionary Computation*. Cat. No. 01TH8546, IEEE, Vol 1, pp. 81–86.

[15] Wu, M.-Y. and Shu, W. 2001. A high-performance mapping algorithm for heterogeneous computing systems. In *Proceedings 15th International Parallel and Distributed Processing Symposium (IPDPS 2001)*. IEEE, No. 6964466. DOI: 10.1109/IPDPS.2001.925020

[16] Xhafa, F., Alba, E., Dorronsoro, B., Duran, B., and Abraham, A. 2008. Efficient batch job scheduling in grids using cellular memetic algorithms. In *Metaheuristics for Scheduling in Distributed Computing Environments*. Springer, 273–299.