Scalable, Low Complexity, and Fast Greedy Scheduling Heuristics for Highly Heterogeneous Distributed Computing Systems

Cesar O. Diaz*, Johnatan E. Pecero*, Samee U. Khan† and Pascal Bouvry*
*Computer Science and Communication Research Unit
CSC, University of Luxembourg, Luxembourg, LU
Email: {firstname.lastname}@uni.lu
†Department of Electrical and Computer Engineering
North Dakota State University, Fargo, ND, USA
Email: samee.khan@ndsu.edu

***Abstract*—For heterogeneous distributed computing systems, important design issues are scalability and system optimization. Given such systems, it is crucial to develop low computational complexity algorithms to schedule tasks in a manner that exploits the heterogeneity of the resources and applications. In this paper, we report three scalable, and fast scheduling heuristics for highly heterogeneous distributed computing systems. The heuristics are thoroughly compared with three other heuristics from the literature. The benchmarking outlines the performance of the schedulers, representing scalability, schedule length, computational complexity, and memory utilization. The set of experimental results shows that our heuristics perform as efficiently as the related approaches while featuring lower complexity, lower running time, and lower used memory. The experimental results also detail the various scenarios under which certain algorithms excel and fail.**

***Keywords*-Scalable scheduling heuristics, heterogenous computing systems, energy-aware systems, performance of systems, optimization.**

I. Introduction

Heterogeneous Computing Systems (HCS) are widely used as an inexpensive but powerful parallel and distributed systems for solving large-scale complex problems. In HCS, the heterogeneity is characterized by different hardware architectures, operating systems, computing power, and different application requirements and constraints. The heterogeneity is exploited to maximize the cost-effectiveness and performance of the system. Moreover, many HCS components (e.g., naming, authentication, authorization, accounting, communication and scheduling) are affected by scale. Scalability is one of the most important design goals for developers of distributed systems [1]. The scalability of a system can be measured by three different components [2], by size, by geographical position and by administrative scalability [1], [3]. Generally speaking, the scalability of a system indicates its ability to handle growing amount of work by increasing the number of tasks and resources.

In the HCS system, the mapping or scheduling is responsible for optimally allocating the applications onto the machines. It is closely related to the performance of the HCS system. Due to its key importance on performance, the scheduling problem has been extensively investigated and numerous methods have been reported in the literature. Several heuristics [4], [5], [6], [7], [8] and meta-heuristics [9], [10], [11] have been developed. However, most of them have a high overhead. This implies that the time needed to run these algorithms on large-scale systems can be a disadvantage when such algorithms are used frequently. Some efforts have been developed to cope with the aforementioned issues, such as parallelization of scheduling heuristics [12] and efficient use of graphics processing units (GPUs) [13].

Low overhead is a key issue for large-scale problems in which $O(t^2)$ algorithms (where t is the number of tasks into the system to be executed) may be cost-prohibitive. With the new exascale computing paradigm, scalability issues in scheduling algorithms are a major concern. The scheduler deals with thousands of resources, users and jobs [14]. Moreover, there are some situations in which only low-cost overhead scheduling heuristics can be used, where the scheduling process is performed during the execution of the allocated tasks. For example, when the HCS system is used as an application service provider to respond to online computational requests. The tasks waiting for its execution can be scheduled in a batch mode to increase the system utilization ratio. Although time-consuming heuristic can achieve a shorter schedule length, the scheduling process would delay the actual task completion, which must be prohibitive if tasks have real-time constraints [6]. Therefore, low-cost and scalable scheduling heuristics are the best choice in this situation.

In this paper, we investigate and compare six static heuristics that address the problem of scheduling independent tasks in a HCS system. We are interested in performance and scalability issues. In terms of performance we consider the optimization of schedule length or makespan. We evaluate scalability of the heuristics by increasing the number of tasks and machines. The six heuristics that we evaluate in this work are: the well known Min-Min [5], two Task Priority Diagram (TPD) based heuristics [6], and

our three developed *low-complexity* heuristics [15], [2]. The low computational complexity heuristics are based on list scheduling approaches and considered as batch mode scheduling heuristics[1]. The heuristics take advantage of the resource capabilities and task requirements to optimize their objective. Their main characteristics are low-cost, and good performance behavior [15]. We evaluate the performance of the algorithms by analyzing the results of numerous simulations that feature high heterogeneity of resources, and high heterogeneity of applications. Simulation studies are performed based in the experimental framework proposed in [6] comparing their best heuristics and the Min-Min algorithm [4] against the *low complexity* heuristics. The large set of experimental results shows that the promoted *low complexity* heuristics perform better than the related approaches in most of the studied scenarios Moreover, the low execution time depicts the applicability for the considered scheduling problem and the scalability.

The paper is organized as follows. Section II describes the scheduling problem. After presenting related work in Section III, the proposed heuristics are described in Section IV. The computational results of applying the proposed heuristics to the scheduling problem are provided in Section V, as well as comparisons of the performance and scheduling costs against stat-of-the-art heuristics from the relevant literature. Section VI concludes the paper.

II. Background

There exists several literature covering many job scheduling and heterogeneous computing models. In this paper, we consider the job scheduling problem with the following characteristics: a set of $M = \{m_1, ..., m_m\}$ machines and a set of independent tasks $T = \{t_1, ..., t_t\}$ to be executed on the system. Each task is considered as an indivisible unit of workload and must be processed completely in a single machine without interruptions. The computational model that we consider is the *expected time to compute* (ETC) model [4]. This model is assumed that we know the estimate of the computational load of each task, the computing capacity of each resource, and an estimation of the prior load of the resources. The ETC matrix, of size $t \times m$ is known. Each element $ETC[t_i][m_j]$ of the matrix indicates the expected time to compute task t_i on machine m_j. The model allows for a representation of the heterogeneity among tasks and machines. Machine heterogeneity evaluates the variation of execution times for a given task across the computing resources. Task heterogeneity represent the degree of variation among the execution time of tasks for a given machine. The model is implemented by the *coefficient-of-variation* (COV) based method [16] using *task heterogeneity* as V_{tasks} parameter, and *machine heterogeneity* as $V_{machines}$ parameter. These two parameters are used to characterize different heterogeneous computing environments and computational tasks. For instance, *low* values of $V_{machines}$ parameter represent computing systems composed of similar, almost homogeneous, computing resources. On the contrary, computing systems integrated by resources of different type and capacity are represented by *high* values of $V_{machines}$.

Low values of V_{tasks} parameter represents cases when tasks are almost homogeneous (i.e., when the computational requirement of tasks is quite similar), with nearly the same execution times for a given machine. *High* values of V_{tasks} parameter describes scenarios in which different types of applications are submitted to execute in HCS ranging from simple applications to complex programs that require large computational time. Moreover, the ETC model also tries to reflect the characteristics of different scenarios using different ETC matrix consistencies defined by the relation between a task and how it is executed in the machines according to heterogeneity of each one [4], [16]. The scenarios are *full-consistent*, *partial-consistent*, and *original-consistent* [6]. The full-consistent scenario models the *Single Process Multiple Data* (SPMD) applications that use local input data. It implies that if a given machine m_j executes task t_i faster than machine m_k, then machine m_j executes all tasks faster than machine m_k. The original-consistent scenario represents a case where machine m_j may be faster than machine m_k for some tasks and slower for others. Finally, the partial-consistent scenario consists of both aforementioned scenarios. Furthermore, the ETC matrix considers an overhead implied by moving the executable code and data associated with each task.

The scheduling problem is formulated as follows. Given an HCS system composed of the set of M machines and the set of n tasks, any task is scheduled without preemption from time $\sigma(t_i)$ on machine $\pi(t_i)$, with an execution time $ETC[t_i][m_j]$. The task t_i completes at time $C_i = \sigma(t_i) + ETC[t_i][m_j]$. The objective is to minimize the maximum completion time ($C_{max} = max(C_i)$) or makespan.

III. Related Work

One of the most widely used batch mode scheduling heuristic for the scheduling problem in HCS systems is the Min-Min algorithm [4], [6], [5]. The Min-Min algorithm, which is referred to as *Heuristic 3* in this paper, starts with a set of all unmapped tasks, then it works in two phases. In the first phase, Min-Min establishes the minimum completion time for every unassigned task. In the second phase, the task with the overall minimum expected completion time is selected and allocated on the corresponding machine. The task is removed from the set and the process is repeated until all tasks are mapped.

The *max-min* heuristic follows the same procedure as Min-Min. The main difference is in the second phase. The

[1]In the batch mode, meta-tasks or applications are mapped and scheduled after predefined intervals [4].

task with the maximum completion time is selected instead of minimum, as Min-Min, and assigned to the corresponding machine. The sufferage heuristic [17] computes for each task the difference between the second earliest completion time (on a machine m_j) and the earliest completion time (on a machine m_i) in the first step. This difference is called the sufferage value. In the second step, the heuristic selects the task with the maximum sufferage value. Then, the task is assigned to the corresponding machine with minimum completion time. The heuristic gives precedence to those tasks with high sufferage value. *Sufferage II* and *Sufferage X* extend the original Sufferage heuristic [18]. *High Standard Deviation First* [7] considers the standard deviation of the expected execution time of a task as a selection criterion. The task with the highest standard deviation must be assigned first for scheduling.

The heuristics proposed in [6] take advantage of the task heterogeneity. These heuristics are founded on the idea of defining an order of task execution. For that purpose, the authors proposed the Task Priority Diagram (TPD) technique. TPD is a precedence task graph, based in the Hasse diagram, that defines a partial order between tasks based on their ETC values. TPD contains the information about the mapping sequence of tasks. For the Min-Min algorithm, the element in the lower level of the diagram always has higher priority for mapping, and an element cannot be mapped until all its predecessors have been mapped. Algorithm 1 shows the generic framework of the scheduling algorithm with TPD in the decreasing order of task effort. In the algorithm the cycle (from line 3 to line 8) repeats until all the tasks have been assigned. In each round of the iteration, T is the set of all the maximal elements of the unassigned tasks in the current graph G. The First-Mapping Step algorithm (line 4) can then be used to select a task t_i' from T and map it onto m_j'. After t_i' is mapped, node t_i' and its edges are removed from G, and the set T, containing the maximal elements of the new G, is also updated. In First-Mapping-Step two metrics are used. For each task the first metric is to select the best machine onto which the task can be assigned. Based on this result, the second metric is used to select the best task with its corresponding machine. The authors conclude that TPD with Min-Min and TPD with min-minSD (minimum completion time on the first phase and minimum standard deviation in the second phase) outperform the related approaches. In this paper, we use these heuristics and Min-Min as a basis of comparison.

In the following, we refer to TPD_minCT_minCT algorithm as, *Heuristic* 1. It is one of the *greedy* heuristics proposed in [6] with the support of a *task priority graph* and Min-Min heuristic. TPD_minCT_minSD algorithm as *Heuristic* 2 following the same procedure of *Heuristic* 1, but in the second phase this heuristic uses minimum standard deviation.

The complexity of all these algorithms is at least $O(t^2m)$ [6].

Algorithm 1 TPD heuristic: Mapping Algorithm with TPD

1: G = Hasse-Diagram-Generator($E_{p\times q}$) {E is the input ETC matrix};
2: $T = \{t | t$ is a maximal element of $G\}$;
3: **while** $T \neq \emptyset$ **do**
4: $F_{i'j'}$ =First-Mapping-Step($\{F_{ij} | t_i \in T, 1 \leq j \leq q\}$, heuristic);
5: map $t_{i'}$ to $m_{j'}$ and update the load on $m_{j'}$;
6: delete $t_{i'}$ and the edges of $t_{i'}$ from G;
7: $T = \{t | t$ is a maximal element of the new $G'\}$;
8: **end while**

IV. Heuristics Descriptions

In this section, we describe three low computational complexity heuristics with good quality schedules proposed in [15]. These heuristics are based on the Min-Min algorithm. However, we took special care to decrease computational complexity. The main idea is to avoid the loop on all the pairs of machines and tasks in the Min-Min algorithm corresponding to the first phase of the heuristic [19]. One alternative is to consider one task at a time to determine the task that should be scheduled next. The heuristics sort the tasks by a predefined priority, so that they must be selected in constant time. Once the order of tasks is determined in the second phase, each task is assigned to the machine that minimizes two objectives (a) the expected completion time and (b) the execution time of tasks.
This paper uses the weighted function called *score function* $SF(t_i, m_j)$ (see Eq. 1) [15], that tries to balance both aforementioned objectives. The goal is to minimize makespan and balance the load of the system. The main difference among the *low-complexity* heuristics is the priority used to construct the list.

The score of each mapping event is calculated as in Eq. (1). For each machine m_j,

$$SF(t_i) = \lambda \cdot \frac{C_i}{\sum_{k=1}^{m} C_{ik}} + (1-\lambda) \cdot \frac{ETC[t_i][m_j]}{\sum_{k=1}^{m} ETC[t_i][m_k]}, \quad (1)$$

where $\sum_{k=1}^{m} C_{ik}$ is the sum of the completion time of the task t_i over all machines and $\sum_{k=1}^{m} ETC[t_i][m_k]$ is the sum of the expected time to complete a task t_i over all machines. The first term of Eq. (1) aims to minimize the completion time of the tasks t_i, while the second term aims to assign the task to the fastest machine or the machine on which the task takes the minimum expected time to complete.

Algorithm 2 depicts the general structure of the heuristics. It is based on classical list scheduling algorithms for which well founded theoretical performance guarantees exist [5]. The heuristics start by computing the *Priority* of each task according to some objective (line 1). Thereafter, the sorted

list of tasks is computed (line 2). The order of the list is not modified during the execution of the heuristics. Next, the heuristics proceed to allocate the tasks to the machines and determine the starting date for them (main loop line 3). One task is scheduled at a time. The heuristics always consider the task t_i at the top of the list (highest priority) and remove it from that (line 4). The score function $SF(t_i, m_j)$ Eq. (1), for the selected task is evaluated for all the machines (lines 5 and 6). Each heuristic then selects the machine for which the value of the score function is optimized for task t_i and it is scheduled on that machine (line 8). The task is removed from the list (line 9) and it restarts the main loop.

Algorithm 2 Pseudo-code for the low complexity heuristics

```
1: Compute Priority of each task t_i ∈ T according to some
   predefined objective;
2: Build the list L of the tasks sorted in decreasing order of
   Priority;
3: while L ≠ Ø do
4:    Remove the first task t_i from L;
5:    for each machine m_j do
6:       Evaluate Score Function SF(t_i);
7:    end for
8:    Assign t_i to the machine m_j that optimize the Score
      Function;
9:    Remove task t_i from the list L;
10: end while
```

The heuristics differ in the way used to compute the priorities. *Minimum*, *maximum*, and *average* expected time to complete the task is used. The names of the heuristics are respectively: (1) *MinMax Min* or *Heuristic 4*. The algorithm uses minimum completion time of tasks to determine the priority, thereafter, the tasks are sorted in decreasing order of the maximum completion time and scheduled based on the minimum completion time. (2) *MaxMax Min* or *Heuristic 5*, the algorithm uses maximum completion time of tasks as a priority. The tasks are sorted in decreasing order of their maximum completion time and scheduled based on the minimum completion time. And (3) *MeanMax Min* or *Heuristic 6*, considers the average execution time of tasks as a priority. The tasks are sorted in decreasing order and scheduled based on the minimum completion time.

V. Experiments

In this section, we compare the heuristics described in the previous section by simulation. We follow the same experimental framework proposed in [6] for fair comparison of heuristics. A simulation software tool has been developed in this study to compare the *low complexity* heuristics and the related approaches. ETCs described in Section II are used for performance assessment. The experiments were performed using a MacPro1.1, Dual-Core Intel Xeon©2.66GHz, 4MB L2 cache (per processor), 8GB in RAM.

A. Parameter tunning

Considering the ETC model generated in [6], following parameters are used: V_{task} is [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 1.1] and $V_{machine}$ is [0.1, 0.2, 0.3, 0.4, 0.5, 0.6], the mean of task execution time μ_{task} is 1000. The heterogeneous ranges were chosen to reflect that, in real situations, there is more variability across the execution time for different tasks on a given machine than that across the execution time for a single task on different machines. Table I shows the 16 heterogeneities tasks and machines combinations.

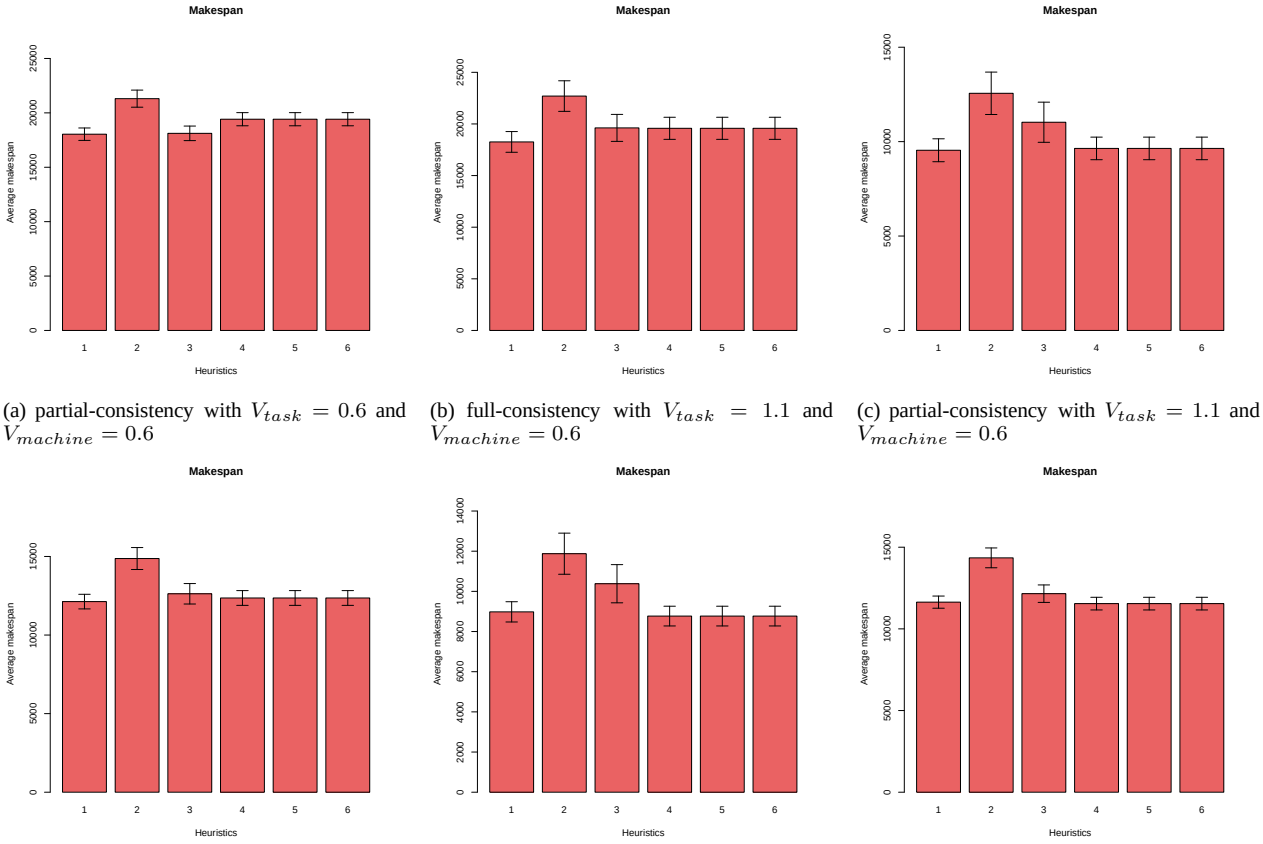
Table I: Total of 16 Heterogeneity Combinations.

Combinations					
V_{task}	$V_{machine}$	V_{task}	$V_{machine}$	V_{task}	$V_{machine}$
0.1	0.1	0.6	0.6	-	-
0.2	0.1	0.7	0.6	0.6	0.2
0.3	0.1	0.8	0.6	0.6	0.3
0.4	0.1	0.9	0.6	0.6	0.4
0.5	0.1	1	0.6	0.6	0.5
0.6	0.1	1.1	0.6	-	-

We assume that the tasks arrive at the system before the scheduling event. Furthermore, we consider that all the machines are idle or available at time zero, which is possible by considering an advance reservation. We have generated 144,000 instances (1000 for each parameter combination, in total 48 combinations with three different sizes) as explained below. We have generated instances with 512 tasks in size to be scheduled on 16 machines, and, for scalability reasons, we increase the number of tasks as well as the number of machines such that 1024 tasks to be scheduled on 32 machines and 2048 tasks to be scheduled on 64 machines. Moreover, instances with 4096 tasks in size to be scheduled on 128 machines and 8192 tasks in size to be scheduled on 256 machines were generated. As it can see, in Figure 4, for these two set of instances, the scheduling calculations for the related heuristics are expensive.

All of the experiments are performed as follows: V_{task} increases with a step pace 0.1 from 0.1 to 0.6 while $V_{machine}$ is fixed at 0.1 then $V_{machine}$ is fixed at 0.6 while V_{task} increases from 0.6 to 1.1. Finally, the last experiment part V_{task} is fixed at 0.6 while $V_{machine}$ increases from 0.1 to 0.6 [6]. According to the chosen lambda values of the *Score Function* defined in Section IV, it took the best lambda for each *tasks* $\times$ *machines* combination $(\lambda = 0.8)$ as computed from early work [2].

We are interested in performance and scalability issues. Therefore, the performance of the simulated heuristics is measured by the average makespan of the ETCs with the specified type. In terms of scalability, we increase the size of the ETC and the number of machines in the HCS system and we evaluate performance, time to compute the schedule and the memory used by each heuristic. In the next section,



(a) partial-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.6$

(b) full-consistency with $V_{task} = 1.1$ and $V_{machine} = 0.6$

(c) partial-consistency with $V_{task} = 1.1$ and $V_{machine} = 0.6$

(d) partial-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.5$

(e) original-consistency with $V_{task} = 1.1$ and $V_{machine} = 0.6$

(f) original-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.5$

Figure 1: Average makespan of the heuristics for 512×16, Figures (a) and (b) when Heuristic 1 is better than ours, Figures (c) and (d) when Heuristics 3, 4, 5 and 6 are almost the same among them and better than Heuristic 3, Figures (e) and (f) when *low-complexity heuristics* are better.

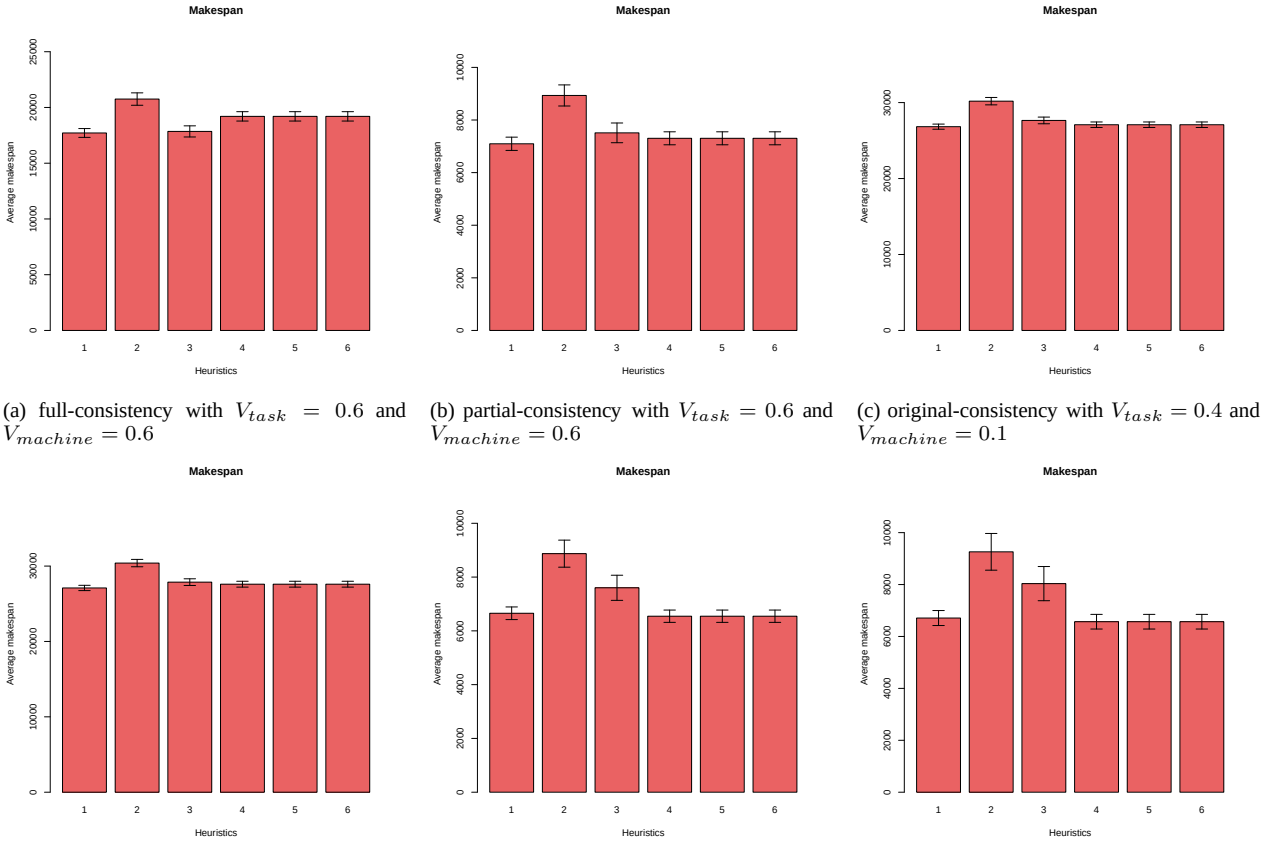
we present the simulation results.

B. Results

1) Makespan: The most significative results are depicted in Figure 1, Figure 2, and Figure 3 for the average makespan. There are three sets of results. For the first set, Figure 1 shows the results for 512 task to be scheduled on 16 machines. Approximately 75% of the results *low-complexity heuristics* are similar to or better than results referred in [6]. The six images depicted in Figure 1 show the three common comparison states among them: Figure 1(a) and Figure 1(b) when at least one of the related heuristic outperform *low-complexity heuristics*; Figure 1(c) and Figure 1(d) when they have almost the same average makespan; finally Figure 1(e) and Figure 1(f) *low-complexity* heuristics are better than related heuristics. Notice that is the average makespan of *low-complexity heuristics* bigger than *Heuristic* 1 in most of the full-consistency cases, and similar behavior on almost partial-consistency simulations. The best behavior of *low-complexity heuristics* is in original-consistency cases, which refers to the most generic and real scenario [4]. We take *Heuristic* 1 as a basis of comparison because outperform *Heuristic* 2 and *Heuristic* 3 in most of cases, and the results are consistent as showed in [6]

For the second set, Figure 2 shows the results for 1024 tasks to be scheduled on 32 machines. In this combination, approximately 62% of the results of *low-complexity heuristics* are similar to or better than the results obtained in [6]. As 512×16 combinations, there are six images depicted in Figure 2 with the same arrangement as Figure 1. The measure of 62% instead of 75% show the best scalability behavior of the *low cost heuristics* against *Heuristics* 1.

For the third set, Figure 3 show the results for 2048 task to be scheduled on 64 machines. In this combination, approximately 60% of the results of *low-complexity heuristics* are similar to or better than the results obtained in [6]. As 512×16 and 1024×32 combinations, there are six images depicted in Figure 3 with the same arrangement as Figures 1 and 2. The measure of 60% instead of 75% and 62% as two aforementioned measures, show the best scalability behavior



(a) full-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.6$

(b) partial-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.6$

(c) original-consistency with $V_{task} = 0.4$ and $V_{machine} = 0.1$

(d) partial-consistency with $V_{task} = 0.4$ and $V_{machine} = 0.1$

(e) original-consistency with $V_{task} = 0.9$ and $V_{machine} = 0.6$

(f) original-consistency with $V_{task} = 1.1$ and $V_{machine} = 0.6$

Figure 2: Average makespan of the heuristics for 1024×32, Figures (a) and (b) when Heuristic 1 is better than ours, Figures (c) and (d) when Heuristics 3, 4, 5 and 6 are almost the same among them and better than Heuristic 3, Figures (e) and (f) when *low-complexity heuristics* are better.

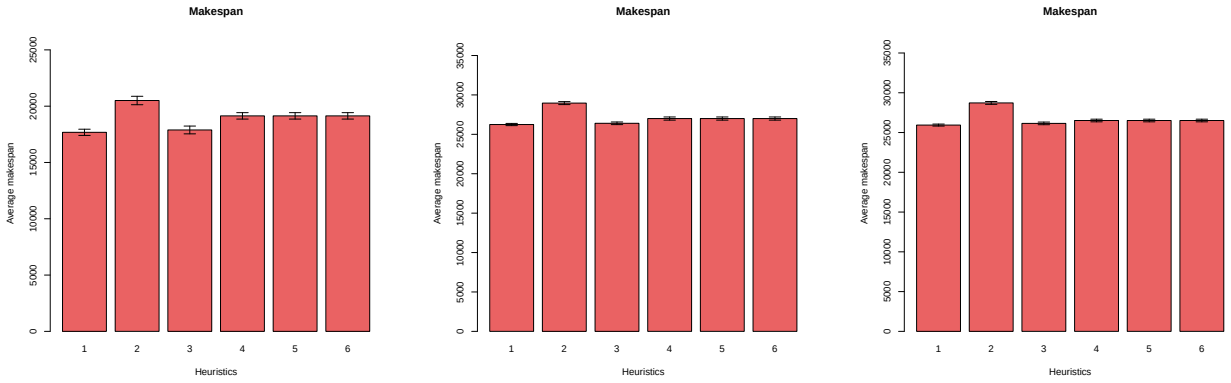
of the *low-complexity heuristics* against *Heuristic* 1. Moreover, the memory and time consumption are exponential increasing more than *low-complexity* heuristics, as we can see in the next section.

2) Time and Memory: Due to the low complexity of *low-complexity heuristics*, the results of the scheduling time calculation is efficient as well as the memory that each algorithm uses.

Time and memory used for all heuristics were measured for each size instance. Each algorithm has been executed separately and the experiments are independent. Figure 4(a) shows the time results in milliseconds of these executions. The logarithmic scale was used to emphasize the results. As can be observed, *low-complexity* heuristics have better behavior. The algorithms use less time to compute a schedule over all the set of instances. Figure 4(b) shows the memory used in MB to calculate each heuristic as well. We can observe in this figure, that *low complexity* heuristics outperform the related algorithms for all the sizes of the instances and the gain in memory space is more important when the size of the instances scales. The evaluated heuristics use much more memory when the instances scale because the algorithms evaluate the completion time for the remaining tasks to be scheduled at each step of the loop, furthermore, two of these evaluated heuristics, use TPD graph. On the contrary, low-complexity heuristics have a good scalability and low overhead, memory use is reduced, because each step of the algorithm only considers one task to be scheduled, the task with the highest priority.

VI. Conclusion

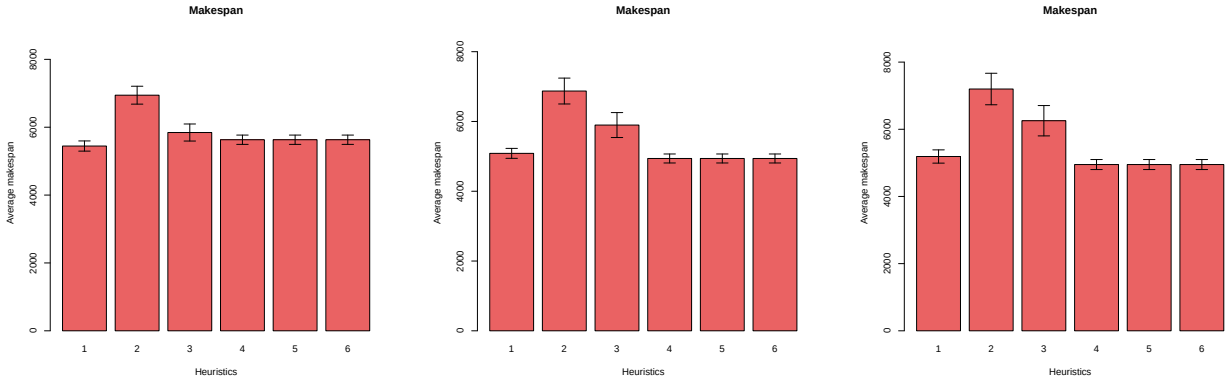
In this paper, we evaluate three batch mode scheduling algorithms in the context of their performance and scalability in heterogeneous computing systems. The algorithms are based on list scheduling approaches and they were evaluated and compared with the best heuristics reported in literature. The set of experimental results shows that the evaluated heuristics perform as efficiently as the related approaches while featuring lower complexity, lower running time, and lower used memory. Moreover, these show their applicability



(a) full-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.6$

(b) partial-consistency with $V_{task} = 0.2$ and $V_{machine} = 0.1$

(c) original-consistency with $V_{task} = 0.2$ and $V_{machine} = 0.1$

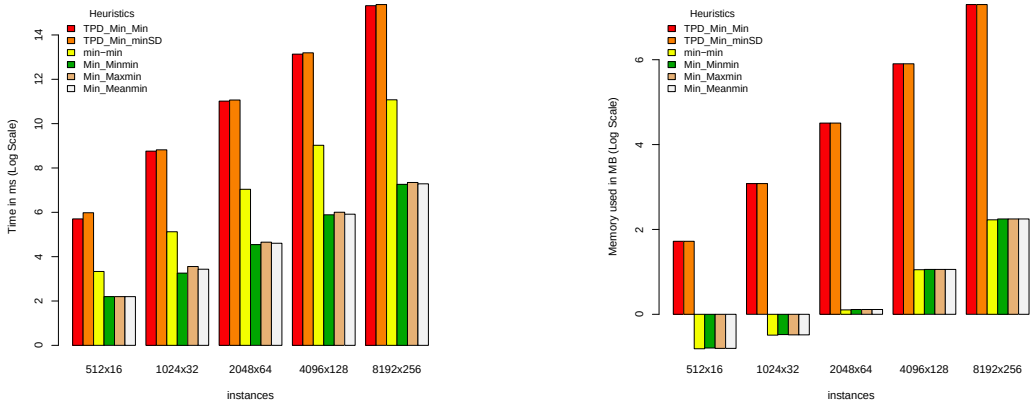


(d) partial-consistency with $V_{task} = 0.6$ and $V_{machine} = 0.6$

(e) original-consistency with $V_{task} = 0.9$ and $V_{machine} = 0.6$

(f) original-consistency with $V_{task} = 1.1$ and $V_{machine} = 0.6$

Figure 3: Average makespan of the heuristics for 2048×64, Figures (a) and (b) when Heuristic 1 is better than ours, Figures (c) and (d) when Heuristics 3, 4, 5 and 6 are almost the same among them and better than Heuristic 3, Figures (e) and (f) when *low-complexity heuristics* are better.



(a) Time consumed for each heuristic

(b) Memory used for each heuristic

Figure 4: Time and Memory used to calculate scheduling for each heuristic

for the considered scheduling problem, scalability, and good efficiency. We are implementing the proposed heuristics in *GreenCloud* simulator [20] and in an Opportunistic Computing environment [21], [22].

ACKNOWLEDGMENT

The authors would like to thank Dzmitry Kliazovich and Daniel Di Nardo for their comments during the preparation

of this paper. This work was partially supported by the National Research Fund (FNR) of Luxembourg through project CORE Green-IT no. C09/IS/05. Samee U. Khan's work was partly supported by the Young International Scientist Fellowship of the Chinese Academy of Sciences, (Grant No. 2011Y2GA01).

REFERENCES

[1] A. S. Tanenbaum and M. V. Steen, *Distributed Systems: Principles and Paradigms*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001.

[2] C. O. Diaz, M. Guzek, J. E. Pecero, P. Bouvry, and S. U. Khan, "Scalable and energy-efficient scheduling techniques for large-scale systems," in *IEEE 11th Int CIT Conference*, 31 2011-sept. 2 2011, pp. 641 –647.

[3] B. C. Neuman, "Scale in distributed systems," in *Readings in Distributed Computing Systems*. Los Alamitos, CA, USA: IEEE CS Press, 1994, pp. 463–489.

[4] T. D. Braun, H. J. Siegel, N. Beck, L. L. Bölöni, M. Maheswaran, A. I. Reuther, J. P. Robertson, M. D. Theys, B. Yao, D. Hensgen, and R. F. Freund, "A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems," *J. Parallel Distrib. Comput.*, vol. 61, pp. 810–837, 2001.

[5] O. H. Ibarra and C. E. Kim, "Heuristic algorithms for scheduling independent tasks on nonidentical processors," *J. ACM*, vol. 24, pp. 280–289, 1977.

[6] P. Luo, K. Lü, and Z. Shi, "A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems," *J. Parallel Distrib. Comput.*, vol. 67, pp. 695–714, 2007.

[7] E. U. Munir, J. Li, S. Shi, Z. Zou, and Q. Rasool, "A performance study of task scheduling heuristics in hc environment," in *Modelling, Computation and Optimization in Information Systems and Management Sciences*, ser. Communications in Computer and Information Science, H. A. Le Thi, P. Bouvry, and T. Pham Dinh, Eds. Springer Berlin Heidelberg, 2008, vol. 14, pp. 214–223.

[8] J. Kolodziej and S. U. Khan, "Multi-level hierarchic genetic-based scheduling of independent jobs in dynamic heterogeneous grid environment," *Inf. Sci.*, vol. 214, pp. 1–19, Dec. 2012. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2012.05.016

[9] F. Xhafa and A. Abraham, "Computational models and heuristic methods for grid scheduling problems," *Future Gener. Comput. Syst.*, vol. 26, pp. 608–621, April 2010.

[10] A. YarKhan and J. Dongarra, "Experiments with scheduling using simulated annealing in a grid environment," in *Proceedings of the Third Int Workshop on Grid Computing*, ser. GRID '02. London, UK: Springer-Verlag, 2002, pp. 232–242.

[11] P. Lindberg, J. Leingang, D. Lysaker, K. Bilal, S. U. Khan, P. Bouvry, N. Ghani, N. Min-Allah, and J. Li, "Comparison and analysis of greedy energy-efficient scheduling algorithms for computational grids," in *Energy Aware Distributed Computing Systems*, A. Y. Zomaya and Y.-C. Lee, Eds. John Wiley Sons, 2012, pp. 246–253.

[12] F. Pinel, B. Dorronsoro, J. E. Pecero, P. Bouvry, and S. U. Khan, "A two-phase heuristic for the energy-efficient scheduling of independent tasks on computational grids," *Cluster Computing*, pp. 1–13, 2012.

[13] P. Frederic, D. Bernabe, and B. Pascal, "Solving very large instances of the scheduling of independent tasks problem on the gpu," *Journal of Parallel and Distributed Computing*, pp. 1–8, 2012, available online 9 March 2012.

[14] I. Raicu, I. Foster, M. Wilde, Z. Zhang, K. Iskra, P. Beckman, Y. Zhao, A. Szalay, A. Choudhary, P. Little, C. Moretti, A. Chaudhary, and D. Thain, "Middleware support for many-task computing," *Cluster Computing*, vol. 13, no. 3, pp. 291–314, Sep. 2010.

[15] C. O. Diaz, M. Guzek, J. E. Pecero, G. Danoy, P. Bouvry, and S. U. Khan, "Energy-aware fast scheduling heuristics in heterogeneous computing systems," in *HPCS, 2011 Int. Conference*, july 2011, pp. 478 –484.

[16] S. Ali, H. J. Siegel, M. Maheswaran, and D. Hensgen, "Representing task and machine heterogeneities for heterogeneous computing systems," *Journal of Science and Engineering*, vol. 3, no. 3, pp. 195–207, 2000.

[17] M. Maheswaran, S. Ali, H. J. Siegel, D. Hensgen, and R. F. Freund, "Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems," in *Proceedings of the Eighth Heterogeneous Computing Workshop*, ser. HCW '99, Washington, DC, USA, 1999.

[18] H. Casanova, D. Zagorodnov, F. Berman, and A. Legrand, "Heuristics for scheduling parameter sweep applications in grid environments," in *Proc. of the 9th Heterogeneous Computing Workshop*, USA, 2000, pp. 349–363.

[19] A. Giersch, Y. Robert, and F. Vivien, "Scheduling tasks sharing files from distributed repositories," in *Euro-Par 2004 Parallel Processing*, ser. Lecture Notes in Computer Science, M. Danelutto, M. Vanneschi, and D. Laforenza, Eds. Springer Berlin Heidelberg, 2004, vol. 3149, pp. 246–253. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27866-5_32

[20] D. Kliazovich, P. Bouvry, and S. Khan, "Greencloud: a packet-level simulator of energy-aware cloud computing data centers," *The Journal of Supercomputing*, pp. 1–21, 2010, 10.1007/s11227-010-0504-1. [Online]. Available: http://dx.doi.org/10.1007/s11227-010-0504-1

[21] H. Castro, M. Villamizar, G. Sotelo, C. Diaz, J. Pecero, and P. Bouvry, "Green flexible opportunistic computing with task consolidation and virtualization," *Cluster Computing*, pp. 1–13, 2012. [Online]. Available: http://dx.doi.org/10.1007/s10586-012-0222-y

[22] H. Castro, M. Villamizar, G. Sotelo, C. O. Diaz, J. E. Pecero, P. Bouvry, and S. U. Khan, "Gfog: Green and flexible opportunistic grids," in *Scalable Computing and Communications, Theory and Practice*, S. U. Khan, L. Wang, and A. Y. Zomaya, Eds. Wiley&Sons, Forthcomming.