Grid Process Scheduling Optimization using the Tabu Search

ANDRÉ M. EBERLE[1]
RODRIGO F. DE MELLO[1]

Universidade de São Paulo
Instituto de Ciências Matemáticas e de Computação
Departamento de Ciências de Computação
São Carlos – SP – Brazil
[1]andre.eberle@gmail.com, [2]mello@icmc.usp.br

Abstract. Process scheduling problems present a large solution space, which exponentially increases according to the number of computers and processes. In this context, exact approaches are, therefore, infeasible. This limitation motivated several works to consider meta-heuristics to optimize the search for good solutions. In that sense, this work proposes a new approach based on the Tabu Search to improve process scheduling by considering application knowledge and the logical partitioning of distributed environments. Such knowledge comprises historical application events (captured during execution) which allow a better parametrization of the optimizer and, consequently, generates better results. Simulation results confirm the contributions of this new approach, which outperforms other techniques when dealing with large and heterogeneous environments, such as Grids.

Keywords: Process scheduling, Meta-heuristics, Tabu Search, Grid computing, Cluster computing.

(Received November 30, 2009 / Accepted September 15, 2010)

1 Introduction

Application-specific knowledge (for example: processing time, memory usage and communication) has been applied to improve process scheduling decisions [14, 22, 9, 20]. Such knowledge is obtained from the description of the computational requirements provided by users, traces of all applications in production environments, or the specific monitoring of single applications. The traces of all applications and specific monitoring have been confirmed to be efficient information sources [19]. Several techniques employ such knowledge to predict parallel application operations as a way to improve scheduling decisions [14, 22, 9, 20, 18, 4, 19].

Feitelson and Rudolph [9] conducted experiments making repeated application executions and observing their resources occupation. They observed that resource consumption presents low variation when executing the same application and, therefore, concluded that resource consumption can be estimated from historical information what avoids the explicit user cooperation in parametrizing scheduling policies.

Harchol-Balter and Downey [14] studied how process lifetime (also called execution time or response time) can be used to improve scheduling decisions. They evaluated execution traces in a workstation-based environment and modeled probability distribution functions to characterize sequential application lifetimes. Based on those models, a load balancing algorithm was proposed, which employs such model to decide on process migrations. Mello *et al.* [5] extended that work by evaluating the processor consumption, what has improved the previous approach.

Pineau *et al.* [17] studied and presented limitations of deterministic scheduling algorithms on heterogeneous environments. They concluded that scheduling optimization approaches highly depend on certain parameters of such environments.

Those works played an important role in process

scheduling, however, with the advent of the Grid computing, new techniques have been proposed to address scheduling decisions by considering heterogeneity and large scale environments. Yarkhan and Dongarra [24] compare a Simulated Annealing (SA) approach to greedy search on a grid computing scenario. They concluded that SA improves scheduling results, due to it helps to avoid local minima. One of the drawbacks of that work is that the authors only consider one parallel application, this is, nothing else is evaluated under the same circumstances. Besides that, the study consider no historical information of applications.

On the other hand, Abraham *et al.*[1] predicted process execution times and proposed nature-inspired heuristics, such as SA and Genetic Algorithms, to schedule applications on grids. Authors only consider Bag-of-Tasks[1] applications, consequently, they do not model communication nor its impact on networks and scheduling.

Besides there are many grid-oriented approaches, most of them only address Bag-of-Tasks applications and, consequently, do not approach inter-process communication. Motivated by this limitation and by the previous good results considering application knowledge [14, 22, 9, 20, 18, 4, 19], this work proposes a new approach to optimize process scheduling in heterogeneous and large scale distributed environments based on the Tabu Search. This approach employs application knowledge (resource occupation), environment capacities (processing, memory, hard disk and network) and current workloads as a way to reduce the total application execution time.

This approach considers a new network logical partitioning technique, which groups the whole computing environment according to network latencies. This logical approach is conducted by a Tabu Search approach, which does not depend on physical partitioning nor environment configurations. Processes are distributed on those computer groups (or logical partitions), what reduces communication costs. Besides that, the cost involved in optimizing scheduling decisions has motivated the proposal of a new fitness function with time complexity of O .

This paper is organized as follows: the process scheduling problem is presented in Section 2; Section 3 describes related optimization approaches; Section 4 presents the Tabu Search and how it is employed in this work; Simulation results and comparisons to other techniques are presented in Section 5; Result analysis and contributions are described under Section 6.

[1]Bag-of-Tasks – this refers to independent tasks with no communication among each other.

2 The Problem

In this paper, the scheduling problem consists in the distribution of processes over a set of interconnected computers in order to reduce the application response time (also called execution time). The response time is the sum of the time consumed in processing, memory and network operations.

According to the formalization by Garey and Johnson [11], we characterize the distributed scheduling problem optimization as follows. Let A be the set of parallel applications $A \quad \{a_0, a_1, \ldots, a_{k-1}\}$ and $\mathbf{size}$. the function which defines the number of processes that compose an application. Thus, each one of the k applications is composed of a different number of processes, for example, $\mathbf{size}\ a_0$, $\mathbf{size}\ a_1$ etc. Consider, then, that the set P contains all processes of all k parallel applications. In this way, the number of elements in P is equal to $|P| \quad \sum_{i=0}^{k-1} \mathbf{size}\ a_i$.

Each process $p_j \in P$, where $j \quad , \ , \ldots, |P| -$, contains particular features, here named behavior, of resource utilization: CPU, memory, input and output. Consequently, every process requires different amounts of resources provided by the set V of computers of the distributed environment (where $|V|$ defines the number of computers). Besides that, each computer $v_w \in V$, where $w \quad , \ , \ldots, |V| -$, has different capacities in terms of CPU, main memory access latency, secondary memory read-and-write throughput, and network interface cards under certain features (bandwidth, latency and overhead to pack and unpack messages).

Computers in V are connected through different communication networks. All this environment can be modeled by a non-directed graph $G \quad V, E$, where each vertex represents a computer $v_w \in V$ and the communication channels in between vertices are edges $\{v_w, v_m\} \in E$. Those channels have associated properties such as bandwidth and latency.

The optimization problem consequently consists in scheduling the set P of processes over the graph vertices in G, attempting to minimize the overall execution time of applications in A. This time is characterized by the sum of all operation costs involved in processing, accessing memory, reading and writing on the hard disk, sending and receiving messages over/from the network.

To simplify the understanding, consider a problem instance with two parallel applications composed of the processes in Table 1 (MI represents the million of instructions executed by processes; MR and MW are, respectively, the number of Kbytes read/write per second from/to the main memory; HDR and HDW are, respectively, the number of Kbytes read/write per sec-

ond from/to the hard disk – secondary memory; NETR and NETS are, respectively, the number of Kbytes received/sent per second from/over the network – in this situation, we also present the sender, for NETR, and the target process, for NETS).

Table 1: Behavior sample of parallel applications

Application 0							
Process	CPU (MI)	MR Kb/s	MW Kb/s	HDR Kb/s	HDW Kb/s	NETR Kb/s	NETS Kb/s
p_0	1,234	123.78	0.00	78.21	0.00	12.50, p_1	532.12, p_1
p_1	1,537	23.54	89.45	0.00	12.30	532.12, p_0	12.50, p_0
Application 1							
Process	CPU (MI)	MR Kb/s	MW Kb/s	HDR Kb/s	HDW Kb/s	NETR Kb/s	NETS Kb/s
p_2	1,221	823.78	70.00	78.21	543.00	10.92, p_3	321.12, p_4
p_3	1,137	223.54	179.45	324.00	212.31	423.12, p_4	10.92, p_2
p_4	2,237	23.54	17.45	12.00	0.00	321.12, p_2	423.12, p_3

Table 2: Sample of computer capacities

Computer	CPU (MIPS)	MR Kb/s	MW Kb/s	HDR Kb/s	HDW Kb/s
v_0	1,200	100,000	40,000	32,000	17,000
v_1	2,100	120,000	50,000	42,000	19,000
v_2	1,800	100,000	30,000	22,000	9,000
v_3	1,700	95,000	20,000	25,000	11,000
v_4	2,500	110,000	60,000	62,000	30,000

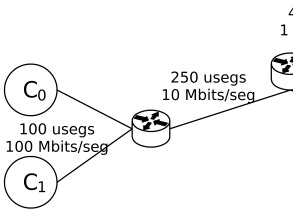

Figure 1: Example of network interconnection

Let the environment be composed of the five computers described in Table 2 (where MIPS represents the processing capacity in million of instructions per second; MR and MW are, respectively, the main memory read-and-write throughput, in Kbytes per second; HDR and HDW are, respectively, the hard disk read-and-write throughput, in Kbytes per second – secondary memory). Such computers, in V, are interconnected according to Figure 1, which also presents the average network bandwidth and latency. Besides that, consider the scheduling operator defined by $\propto$.

The distributed scheduling problem consists in defining on which computer v_w each process p_j will be placed on, considering the resource capacities and workloads. An example of solution for this instance is given by $p_0 \propto v_0$, $p_1 \propto v_1$, $p_2 \propto v_2$, $p_3 \propto v_3$ and $p_4 \propto v_4$. In this way, for each problem instance, we must schedule $|P|$ processes on an environment composed of $|V|$ computers, consequently, the universe of possible solutions is equal to $|V|^{|P|}$. For the previously presented instance, the problem has a solution universe equals to ^{5} , . Real problem instances can consider, for instance, , computers and applications, containing processes each. In this situation, the solution space would be equal to , $^{64 \cdot 512}$, ^{32,768}.

In this context, we may observe that the problem solution space is exponential and, therefore, we must propose alternative approaches capable of providing good solutions in acceptable computational time. There is no polynomial-time algorithm to optimally solve this problem (this is, to find the best solution at all), what allows to characterize it as intractable [11]. Given this fact, we may adopt algorithms that explore part of the solution space and find, by using guess and checking, a good candidate in non-deterministic polynomial time [11].

After characterizing the problem as intractable, it is important to understand how hard it is. In order to understand it, consider the equivalent NP-complete problem approached by Lageweg and Lenstra [16], and presented by Garey and Johnson [11]. This problem defines the process scheduling over m processors as follows:

Instance: *Let the set of tasks T, a number $m \in \mathbb{Z}^+$ of processors, for each task $t \in T$ a length $l\ t\ \in \mathbb{Z}^+$ and a weight $w\ t\ \in \mathbb{Z}^+$, and a positive integer K.*

Question: *Is there a scheduling σ over m processors for T, where the sum, for every $t \in T$, of $\sigma\ t \quad l\ t\ \cdot w\ t$ is not higher than K?*

Adaptation: *In this context, instead of considering m processors, we assume m computers with specific capacities, and T, instead of representing tasks, is the set of processes. The length $l\ t$ defines the process duration and, in our situation, the resource consumption. Besides that, we mention that according to Garey and Johnson [11] the problem stays NP-complete for any instance where $m \geq$. The problem can be solved in polynomial time only if the processes duration are identical, what is not expected in the presented problem, due to there are great variations in application profiles.*

Such cost has motivated the adoption of meta-heuristics which are capable of finding approximate solutions in polynomial time.

3 Optimization Techniques

As previously presented, meta-heuristics have been considered for optimization purposes, due to their ability to find good solutions in polynomial time. This section presents commonly considered meta-heuristics.

3.1 Genetic Algorithm

Genetic Algorithm [13] is an heuristic inspired in animal and plant evolution. It simulates the natural selection and genetic (characteristics) recombination as a way to explore diversity and, therefore, find solutions. Given a certain problem, solutions (codified as chromosomes) are described in terms of their characteristics (genes). Those chromosomes assume the role of individuals in a population, where each one covers part of the solution space. Candidate solutions are evaluated using a fitness function which measures the optimization quality, considering specific problem constraints.

The algorithm periodically considers two genetic adaptations (operators), which change the chromosome characteristics and, consequently, better explores the search space. The considered operators are the mutation and crossover. Mutation randomly selects chromosome genes and changes their values. Crossover recombines chromosomes by considering the random exchange of genes, simulating the mating between living beings. After such adaptations, the modified chromosomes compose new candidate solutions to be evaluated. Populations of candidates are assessed until converging to an acceptable quality or according to other constraints (e.g. time).

3.2 Hopfield Artificial Neural Network

The Hopfield artificial neural network [15] was introduced as an associative memory between input and output data. This network associates inputs and outputs using a Lyapunov energy-based function. This function is applied in dynamical systems theory in order to evaluate stable states. The goal of such systems is to prove that modifications in the system state reduce the energy function results.

The use of an energy function has motivated the adoption of the Hopfield network to solve optimization problems. In that sense, the problem solution attempts to obtain the global minimum of the energy function, which represents the best solution for a certain instance. The connection weights of Hopfield network neurons represent the energy function surface, which is explored to find minima regions.

The output function considered for Hopfield networks [10] is defined in Equation 1, where γ is the constant gain parameter and u_i is the network input for the neuron i.

$$v_i \quad g_i\ \gamma u_i \quad - \quad tanh\ \gamma u_i \tag{1}$$

In order to approach an optimization problem using the Hopfield network, an energy function has to be defined and applied on the matrix that represents the system solution. This function defines constraints to acceptable solutions. The Hopfield artificial neural network applies the energy function on a possible solution, evaluates it and decreases the input values of neurons. Afterwards, it proposes a new solution to be analyzed. This cycle is executed until finding a solution that satisfies all constraints.

3.3 Asynchronous Boltzmann Machine

The Asynchronous Boltzmann Machine is a generalization of the Hopfield Neural Network which adds a stochastic element. While Hopfield considers thresholds to decide whether a neuron is activated, the Boltzmann machine employs a probability function, described by neuron connection weights and threshold parameters, as shown in Equations 2 and 3.

$$E_i \quad \sum_j s_i w_{ij} - \theta_i \tag{2}$$

$$p\ i \qquad \frac{}{e^{-\Delta E_i/T}} \tag{3}$$

The parameter T represents the temperature of the system. This concept is based on the increase and decrease of temperature in real systems. We observe that high values of T increase the probability of neuron activations, what allows fast convergence, however, it is likely to reach a local minima. Lower values are better when looking for a global minimum, although they consume more processing time.

The application of this technique is similar to Hopfield. The solution and the optimization function are similarly described. The temperature is gradually decreased down to a limit when an output is issued. This technique presents a smaller bias toward local minima when compared to Hopfield networks.

3.4 Tabu Search

Tabu Search is a meta-heuristic, proposed by Glover and Laguna [12], which explores search spaces looking

for good solutions. It applies the concept of movement and the Tabu list. In order to illustrate it, consider an optimization problem where there is a finite set S which contains all possible (feasible or infeasible) solutions. Let $s \in S$ be a solution and $M\ s, par_{movement}$ be a function over s with a set of parameters $par_{movement}$. $M\ .$ is called a movement if $M\ s, par_{movement}\ \ s'$ where $s' \in S$. This is, if it is possible to move from a solution s to another one s'.

The set of parameters $par_{movement}$ restricts the movement M from a solution s to a limited number n of new solutions. Therefore, $M\ .$ defines a subset in S with size n which is also called the neighborhood of s, defined as $n\ s\ .$ We may observe that solutions reached after a movement may or may not be feasible. The feasibility and quality of solutions are evaluated by a fitness function. Some works add constraints in the movement function $M\ .$, consequently, only feasible solutions are accepted, what reduces the complexity of fitness functions.

The main idea of the Tabu Search is to navigate from a certain solution s to some $s' \in n\ s$, where, ideally, s' presents a better quality. An usual implementation is to choose the best solution from $n\ s$, or a subset $n'\ s\ \subset n\ s$. It is important that the movement allows any solution to be reached from any other within a finite number of movements, otherwise the meta-heuristic does not cover all possible areas in the search space and potentially optimal solutions may never be visited.

In Tabu Search, there is a list of constraints indicating prohibited movements and parameters. Certain movements, or their parameters, are inserted in this list for a number of iterations, meanwhile, they are forbidden and, consequently, avoided by the search algorithm.

Once the movement function is defined and the Tabu criteria determined, the algorithm starts by choosing a random candidate solution and executing the function $M\ .$ over it, using m parameters in $par_{movement}$. Implementations usually consider some $m < k$, where k is the number of possible parameters to define the movement. The application of $M\ .$ over a candidate solution s, using each one of the m parameters, defines a subset of neighbor solutions $n\ s$. The fittest solution found in neighborhood and its parameters are tabued for a certain number of iterations (i.e. they will not be used for a while). These steps are repeated until a stop criterion is reached, and the best solution visited is chosen. When tabuing a solution, the algorithm attempts to find other better candidates, this is, it better explores the search space.

3.5 Simulated Annealing

Simulated Annealing (SA) is an approach based on the metallurgic process of annealing, which consists of the successive heating and cooling of metals [2, 10]. According to that process, the energy of atoms is continuously increased and decreased, leading to an equilibrium state. SA simulates this process, looking for candidate solutions in a state of high energy, or temperature, which is gradually reduced.

Similarly to the Tabu Search (Section 3.4) the concept of movement and neighborhood are considered. Once the best solution of the neighborhood, s', is found, the algorithm probabilistically moves towards it using a probability function in the form $P\ s, s', T\ .$ P is commonly some function that has a bias in the direction of the best solution, either s or s'. The algorithm allows more random movements when the system energy (parameter T) is high, but increases the bias towards good solutions when the energy is low, trying to escape from local minima. While these iterations are repeated, the temperature T is continuously decreased, until reaching a lower limit and finding a good solution.

3.6 Ant Colony Optimization

Ant Colony Optimization (ACO) is an optimization technique inspired in the behavior of ants in colonies [8]. When searching for food, ants wander randomly while laying down pheromone. Once the food is found, the ant returns home, reinforcing a pheromone path. Ants are attracted by pheromone and tend to prioritize paths with such substance. The more intense the amount of pheromone in a path, the more likely other ants will follow it. After some time, the food paths will tend to be more intensively used and, therefore, have more pheromone. As several food paths may be found, the shortest one will be covered faster and, consequently, it will have more pheromone over it. After some time, there is a high probability to consider only the shortest path. ACO, similarly, attempts to find optimal paths while visiting neighbor solutions. The technique considers virtual ants and pheromone to simulate the whole process.

4 Proposed Scheduling Approach

As previously shown, the problem of process scheduling optimization presents an exponential complexity, therefore, approaches to find the best solution (also called exact approaches) are infeasible due to they cover the whole search space. Motivated by the problem complexity and the good results of other meta-heuristics,

this work considers the Tabu Search technique, parametrized with application knowledge, to find good scheduling solutions.

In order to employ knowledge, we developed online monitors to intercept application operations and store them on an experience database. In that sense, we considered the knowledge acquisition model proposed by Senger *et al.* [19], which supports the estimation of the expected behavior of new applications. This model looks for similar applications in execution traces, using a machine learning algorithm inspired in the Instance-Based Learning paradigm [3]. Execution traces of parallel applications are considered as experience database and parallel applications are submitted to the system, as query points. The database information is, therefore, used by the learning algorithm to estimate the behavior of parallel applications (the behavior includes estimatives of the CPU, main and secondary memory, and network usages).

After employing the knowledge acquisition model, we consider the following application behavior components to parametrize our optimization approach: the processing consumption in million of instructions (MI), the total expected memory usage, and the network load, in terms of the number of messages per second and size (bytes). Then, such behavior components are used in our two Tabu-based optimization stages: 1) Firstly, the distributed environment is logically divided into smaller partitions, according to a first Tabu-based optimization approach. This approach considers the average inter-computer communication costs. This step reduces the full search space to local spaces, what makes the second stage of our approach run faster (consequently, it is more useful in real-world circumstances); 2) Secondly, the incoming applications are distributed on one of the logical environment partitions (this logical partition does not require any configuration nor physical segmentation). In that sense, when an application arrives at the system, the logical partition with the lowest processing load is selected, and it executes another Tabu-based optimization approach to locally schedule application processes. Both stages are detailed next.

4.1 First Stage: A Tabu-Based Approach for Distributed Environment Partitioning

Considering an environment composed of V computers, we intend to schedule applications on a subset V' of it (where $V' \subset V$), attempting to reduce communication costs, and therefore the execution time, for all processes. In order to prevent high synchronization delays, we proposed a first stage of environment partitioning using a Tabu-based optimization approach. By partitioning, this approach reduces the search space, this is, the number of candidate solutions for the second stage (Tabu-based approach for process scheduling), reducing the time consumed to find good solutions.

In this stage, candidate solutions are represented by a set of computers. The movement function considers the swapping of a single computer from a logical partition to another. The fitness is computed by summing the communication costs in between every pair of computers in the same logical partition. The lower is the communication cost among computers in the same partition, the higher the solution quality is. The Tabu list is implemented by forbidding the last used movement for a number of iterations.

Our algorithm builds the initial solution dividing the whole environment into several logical partitions, containing at most z computers each (for evaluation purposes, our experiments considered z). The algorithm builds n random parameters for the movement function, which are executed for the current solution. The resulting solutions are sorted and the best one is considered. As long as the correspondent movement is not in the Tabu list, the solution is selected, otherwise the next is chosen. These steps are repeated until the associated movement is not in the Tabu list, or every solution is examined. Thus, the selected solution becomes the current one. The algorithm is iteratively repeated until the best solution found is not modified for k iterations.

The size of z is set by the administrator. Depending on its value, this first Tabu-based stage divides the environment into smaller or larger logical partitions to received applications. A low z reduces the search space for the second stage, however small partitions may have low computing capacity to fulfill the needs of certain applications. For example, z would create as many logical partitions as the number of computers in the environment, then complete applications (including their processes) would be scheduled on the idler logical network, what, in this situation, is a single node. Otherwise, a very high z would join far away (far meaning high latency and low bandwidth) computers in the same logical partition, then, two communicating processes allocated on different and far computers would present high communication delays and, consequently, the performance would decrease.

4.2 Second Stage: Tabu Search for Process Distribution

Once the whole environment is logically divided according to inter-computer communications costs, the subset or partition with idler computers is selected to

receive new launched applications. Consequently, application processes are scheduled over nodes in that logical partition. This intra-partition scheduling is also based on a Tabu Search approach as presented next.

4.2.1 Description of the Solution

In this second stage we also consider the Tabu Search approach, in that sense we described candidate solutions using a matrix of computers (rows) versus processes (columns) – we may observe that one may choose to represent solutions using other data structures such as vectors, trees, etc. In our case, we use a sparse matrix which contains 's and 's, where a in column i and row j represents process p_i scheduled on computer $v_j \in V$. The value depicts that the underlying process is not located at the respective computer. In this context, we consider a feasible or valid solution when every process is scheduled on only one computer.

As an example consider an environment with computers. An application, containing processes, has to be distributed over this logical partition. A matrix is created containing a possible solution such as the one presented in Table 3. In this solution, each process is assigned to only one computer.

Table 3: Matrix of computers (rows) versus processes (columns)

	p_0	p_1	p_2	p_3
v_0	0	0	1	0
v_1	1	0	0	0
v_2	0	0	0	0
v_3	0	0	0	0
v_4	0	1	0	0
v_5	0	0	0	0
v_6	0	0	0	1

4.2.2 Movement

In this second stage, the movement considers the swap of a process p_i to another computer. This swap is parametrized by a number n, which defines how many changes are executed to the process p_i over the sparse matrix. In that sense, the movement function is represented as $M\ p_i, n\ $.

As an example, consider the solution described in Table 3. Let a movement be executed over it, where $M\ p_2, n$, consequently, the process number or p_2, currently assigned to v_0, will be changed in the matrix. An example of result is shown in Table 4. Now p_2 will be allocated on computer v_4. This movement was chosen since it never leaves the feasibly region of solutions and it can also reach any solution from this last (given an enough number of movements).

Table 4: Computers and Processes, Movement of p_2 with $n = 1$

	p_0	p_1	p_2	p_3
v_0	0	0	**0**	0
v_1	1	0	0	0
v_2	0	0	0	0
v_3	0	0	0	0
v_4	0	1	**1**	0
v_5	0	0	0	0
v_6	0	0	0	1

4.2.3 Tabu List

In this second stage, we defined a Tabu List over the parameters of the movement function $M\ p_i, n\ $, where p_i is the process that will be transferred to another computer, and n is the number of swaps. This Tabu list works as follows: after executing a movement over p_i, given n, that action (or movement itself) is added into the Tabu list and, consequently, forbidden for a number of algorithm iterations.

4.2.4 Fitness Function

As described in Section 3.4, the Tabu Search algorithm requires a fitness function, which evaluates the quality of a solution. The fitness is the target function to be minimized. As proposed, the parameters of this function are based on the application-specific knowledge, which includes the total processing cost, memory and network usage. We also consider knowledge about the environment, such as the current workload of computers as well as their capacities and network overheads.

The purpose of our function is to estimate the total computer workload when considering a new candidate solution. This estimative is made by computing the total time consumed when processing instructions, the processing slowdown caused by memory usage (main and virtual memory usage), and the inter-process communication cost.

In order to estimate the processing time, we consider the local process queue of each computer. In that way, the fitness function assumes that the local computer scheduler uses a Round-Robin policy, where the maximum time a process remains in the processor is second (this is, therefore, called time slice). After defining that, the first approach we considered to compute the processing time in every computer was a simulated

execution, which also calculated the slowdown caused by memory usage (when a process consumes memory, processing time is penalized. Mello and Senger [6] concluded that the more the main and virtual memories are occupied, the higher is the slowdown in process execution), as presented in Algorithm 1, where $T\ v_j$ is the memory slowdown function of computer v_j, proposed by Mello and Senger [6], defined in Equation 4 (where vm_j is the main memory size, vs_j the virtual memory size and x is the total memory currently used in computer v_j – all of them in megabytes).

The parameters for $T\ v_j$ were defined according to an average of experiments executed on heterogeneous computers which (conducted in our laboratory) [6]. Among the evaluated computers, we had: AMD 2600+, AMD 64 bits and Intel Pentium 4 and Intel Core 2 Duo processors.

Therefore, function $T\ v_j$ generates a slowdown factor to be multiplied by the processing time consumed. This factor models delays when processes are accessing main and virtual memories. The slowdown is a function of the memory being currently used (x, in Megabytes, in Equation 4). If only the main memory is being used and no swap is done, the slowdown grows linearly according to the amount of megabytes in memory. When swap starts to be used, the slowdown causes an exponential influence in the process execution.

$$T(v_j) = \begin{cases} 1 \quad \text{when using less than } \frac{0.089}{0.0069} \text{ Megabytes} \\ 0.0068 \times x - 0.089 \quad \text{when using more than} \\ \quad \frac{0.089}{0.0069} \text{Megabytes and less than } vm_j \text{ Megabytes} \\ 1.1273 \times \exp((vm_j + vs_j) \times 0.0045) \quad \text{otherwise} \end{cases} \tag{4}$$

The Algorithm 1 adds up all the slowdown caused my memory usage and the expected processing time. This fitness function was, then considered to make optimizations. However, we observed that its time complexity was too high and simulations took long to execute. Then, we started studying other approaches and end up proposing a polynomial equivalent version of this first. Such approach is presented as follows.

Let E_i be the expected execution time for a single process p_i. Let D_i and I_i be respectively the decimal and integer part of E_i, therefore $E_i \quad D_i \quad I_i$. Let $|Q|$ be total the number of processes currently in the processor queue Q. Assume that the queue is kept ordered by the expected processing time (i.e. the process with the lowest expected time is executed first and so on).

Now consider a queue or circular list where processes were previously scheduled. Assume that every process can reside in the CPU for at most second, which is usually referred as CPU quantum (or

Algorithm 1 Process Time and Memory Simulation

```
1: Define simulationTime = 0.0
2: Define totalTime = 0.0
3: Define totalMemory = 0.0
4: Define the expected memory usage for a computer (i.e.
   the sum of the expected memory usage for all processes)
   as computerMemory
5: Define the vector (or queue) which contains running pro-
   cesses as runningProcess[]
6: while runningProcess is not empty do
7:   for every process p in runningProcess do
8:     totalMemory += T(computerMemory)
9:     if p.expectedTime < 1 then
10:      simulationTime += p.expectedTime
11:      totalTime += simulationTime
12:      remove p from runningProcess
13:      computerMemory -= p.memory
14:    else
15:      simulationTime += 1
16:      p.expectedTime -= 1
17:    end if
18:  end for
19: end while
20: return [totalTime,totalMemory]
```

time slice) [23]. Consider a set of n processes in such queue. Now, let a process p_i need $I_i.D_i$ seconds to finish, where I_i is the integer and D_i the decimal part, respectively. This queue is sorted according to $I_j.D_j\ \forall j$, thus, a process at an index j will finish before j . However, we consider a Round-Robin local scheduling policy (by local we mean the CPU policy and not the distributed environment policy), which will schedule $p_0, p_1, \ldots, p_{j-1}$, in such order, before p_j. In that sense, there is some probability that p_{j-1} finishes before using all the CPU quantum and the scheduler releases the CPU and gives it to the next process, this is p_j.

Let E_i be an estimative of the expected execution time for the process p_i in such queue where other processes were also added, where i is the queue index. We observe that the total execution time of each process is always greater than or equal to the time of the previous one (since the queue is kept sorted according to the expected execution times). Since the maximum time a process p_i can reside in the CPU is , its integer expected time I_i determines how many times it will enter and leave the queue, and the decimal part determines the remaining time to finish execution. When all previous processes have finished executing and since there are $|Q| - i -$ remaining processes after p_i (which have larger or equal processing time), the total time that p_i will wait in the queue between each CPU allocation is $|Q| - i - \quad \times$ (this is the time to make a full cir-

cle in the queue – this happens due to we consider the Round-Robin policy as the local CPU scheduler).

Now consider that every process has already executed I_{i-1} times (this is, the total integer part of p_{i-1}), then process p_i returns to the processor only $I_i - I_{i-1}$ times. So, the remaining time of process p_i, after all previous processes $p_j \ \forall j < i$ have finished, is given by $I_i - I_{i-1} \times |Q| - i$, plus D_i. Hence the total execution time for any process, considering only the processing time of its instructions is given by the total time currently executed increased by its own remaining time, according to Equation 5.

$$E_i \quad E_{i-1} \quad D_i \quad I_i - I_{i-1} \times |Q| - i \tag{5}$$

Therefore, the total expected processing time for a computer is given by Equation 6, assuming E_{-1} .

$$\sum_{i=0}^{n} E_{i-1} \quad D_i \quad I_i - I_{i-1} \times |Q| - i \tag{6}$$

Equivalently, the memory terms may be obtained as follows. Let m_j be the total memory of each process p_j and M_i be the sum of all memory allocated at the computer, in the moment that process p_i finishes executing. The total impact of the memory in a computer is the sum of all the occupied memory. Hence, every time a process is at the processor, its overall time is impacted by the slowdown factor (computed according the main and virtual memory usage), as shown by Mello and Senger [6]. Consequently, the factor represents the slowdown impact being applied until the moment the process p_i leaves the queue, thus $M_i \quad M_{i-1} - m_{i-1}$. To allow a proper mathematical model, let $M_{-1} \quad M_0$ M where M is the total sum of memory beforehand, and m_{-1} . Assume that the impact of the memory on a process when it remains less than one second ($t <$) at the processor is equivalent to the impact caused by the remaining one second. Analog to the processing time, we can define T_i as shown in Equation 7, where $T \ v_j \quad M_i$ is the slowdown impact caused by the memory currently used in computer v_j, after finishing all processes $p_j \ \forall j < i$. The total memory is given by Equation 8.

$$T_i \quad T \ v_j \quad M_i \times I_i - I_{i-1} \times |Q| - i \tag{7}$$

$$\sum_{i=0}^{n} T \ v_j \quad M_i \times I_i - I_{i-1} \times |Q| - i \tag{8}$$

The two previously presented equations can be mapped to an incrementally polynomial time complexity algorithm (as presented in Algorithm 2).

Algorithm 2 Process Time and Memory Simulation

1: Define $totalTime = 0.0$
2: Define $totalMemory = 0.0$
3: Define the expected memory usage for a computer (i.e. the sum of the expected memory usage for all processes) as $computerMemory$
4: Define the vector (or queue) which contains running processes as $runningProcess[]$
5: Define $lastInteger = 0$
6: **for** every process p in $runningProcess$ **do**
7: Define $p.expectedTime$ integer part as I
8: Define $p.expectedTime$ decimal part as D
9: Define p index on the array as i
10: $totalMemory$ += T($computerMemory$) * ((I - $lastInteger$) * ($runningProcess.size$ - i) + 1)
11: $totalTime$ += D + (I - $lastInteger$) * ($runningProcess.size$ - i)
12: $lastInteger$ = I
13: **end for**
14: return [$totalTime$,$totalMemory$]

Further improvements were made to create a fitness function of time complexity O . Using Equation 6, an attempt was made in order to find the difference between the expected processing time before and after inserting a new process. Equation 5 represents the total execution time associated to a certain process in the system. Let a process p_k, with processing time $I_k \quad D_k$, be inserted in the system at the queue index j. Let E_k represent the estimated execution time of the inserted process p_k, which impacts in the execution time of all processes already in the queue. Therefore, E_k is defined in Equation 9. After inserting a process at the queue index j, the execution time of every process p_i, which was previously referred as E_i, is now called E_i. This new nomenclature makes evident that the expected execution time has changed for every process. In Equation 10, the time associated to the process p_k is presented.

$$E_k \quad E_{j-1} \quad D_k \quad I_k - I_{j-1} \times |Q| \quad - j \quad \text{where } k \quad j \tag{9}$$

Equation 9 computes the expected execution time of process p_k (E_k) by considering the estimated execution time E_{j-1} of the previous process p_{j-1}, the decimal part of p_k (D_k) and $I_k - I_{j-1} \times |Q| \quad - j$ deals with a circular list to consume the integer part already processed, similarly to the time calculations presented in Equation 5.

Now that a new process has been inserted in the queue, the process p_j, which was previously in index j, moves to the position j and its expected time now depends on E_k. Equation 10 illustrates this (notice that we still refer to this process as p_j, even though its index has changed).

$$E_j \quad E_k \quad D_j \quad I_j - I_k \times |Q| \quad - \; j \tag{10}$$

We now define the remaining $hatE_i$, for all $i < k$ and $i > k$, where k is the index where process p_k was inserted ($k \quad j$). For processes $p_i \; \forall \; i < j$, Equation 5 can be used to determine the expected times straightforwardly. For processes $p_i \; \forall \; i > j$ the multiplicative factor $n \quad -i$ changes to $n \quad - \; i$ (remember that we are comparing the processes expected times in the initial setup with the ones after the insertion, therefore i is an index from the initial position, and becomes $i \quad \forall \; i > j$ in the final setup). This is demonstrated in Equation 11.

$$E_i \quad \begin{cases} \bar{\hat{E}}_{i-1} + D_i \\ +(I_i - I_{i-1}) \times ((n+1) - i) & i < j \\ \\ \bar{\hat{E}}_{i-1} + D_i \\ +(I_i - I_{i-1}) \times ((n+1) - (i+1)) & i > j \end{cases} \tag{11}$$

By maintaining the initial indices i, it is possible to calculate the difference of the expected times for each process p_i before and after the insertion of process p_k. Using Equations 9, 10 and 11, we calculate $E_i - E_i \; \forall \; p_i$, where E_i is the estimated time for the process p_i before the insertion of p_k. The result is presented in Equation 12.

$$E_i - E_i \quad \begin{cases} (I_i - I_{i-1}) + ((I_{i-1} - I_{i-2}) \\ + \ldots + (I_0 - I_{-1}) & i < j \\ \\ \bar{\hat{E}}_i + D_i + \sum_{a=0}^{j-1}(I_a - I_{a-1}) \\ +D_k + I_k - I_{j-1} & i \geq j \end{cases} \tag{12}$$

These differences can be written in the form of sums, as presented in Equations 13 and 14.

$$\sum_{i=0}^{j-1} E_i \quad \sum_{a=0}^{j-1} \sum_{b=0}^{a} I_b - I_{b-1} \tag{13}$$

$$\sum_{i=j}^{n-1} \bar{\hat{E}}_i = (|Q| - j) \times (\sum_{a=0}^{j-1}(I_a - I_{a-1}) + D_k + I_k - I_{j-1}) \tag{14}$$

Such equations represent the time to be added to the total execution time of each process p_i, initially in the queue, after the insertion of process p_k at the position j. If these sums are added to E_k (the expected time for p_k), the total difference between the current expected time and the time after the insertion is found (obviously the time of p_k must be considered integrally in the difference, since p_k was not in the queue beforehand). So, the total difference is defined in Equation 15, where I_i and D_i are the integer and decimal times expected for each p_i in the original queue (i.e. all the sums iterate over the initial queue, not the one after p_k is inserted, thus i ranges from to $|Q| -$), and $|Q|$ is the number of processes initially at the queue.

$$E_k \quad \sum_{i=0}^{|Q|-1} E_i \quad |Q| \quad \sum_{i=0}^{j-1} I_i - I_{i-1} \; - \sum_{i=0}^{j-1} D_i \quad I_k \; |Q| - j \quad |Q| - j \quad I_{j-1} \quad D_k \quad I_k \tag{15}$$

All the sums considered by the equations only depend on the current values. This behavior allows these sums to be incrementally computed every time the algorithm is executed. Then, in order to evaluate new solutions, we consider the current computer status, thus, taking advantage of previous computations and, therefore, making the algorithm faster.

The calculation of the memory considers the same procedure. Let T_i be the associated memory calculation, which represents the total memory impact for process p_i after the insertion of process p_k, and M_i be the new total memory associated to every process after the insertion. Again the new process p_k is inserted at the position j. The correspondent T_i is defined in Equation 16. When $i < j$, $M_i \quad M_i \quad m_k$, since $M_0 \quad \sum^n m_i$. For $i \geq j$, $M_i \quad M_i$ since $M_i \quad M_{i-1} - m_{i-1}$.

$$Tm_i \quad \begin{cases} T(M_i + m_k) \times (I_i - I_{i-1})(|Q| - i + 1) & i < j \\ T(M_i) \times (I_i - I_{i-1})(|Q| - i + 1) & i > j \\ T(M_i) \times (I_i - I_k)(|Q| - i + 1) & i = j \end{cases} \tag{16}$$

It can be observed that for values between . and the total main memory, the function T . allows that $T \; a \quad b \quad T \; a \quad T \; b \quad .$ if both, a and b, are in this range. This fitness function assumes that every value for memory inside a function T . is in this range in the following calculations.

The differences between the new and old associated memory are defined in Equation 17.

$$Tm_i - Tm_i \quad \begin{cases} T(M_i)(1+(I_i - I_{i-1}))+ \\ (T(m_k) + 0.089) \\ (1+(I_i - I_{i-1})(|Q| - i + 1)) & i < j \\ \\ T(M_i)(I_k - I_{j-1})(|Q| - i) & i = j \\ \\ 0 & i > j \end{cases} \quad (17)$$

The sum of the differences and Tm_k are defined in Equation 18.

$$\sum_{i=0}^{n} Tm_i - Tm_i \quad \sum_{i=0}^{j-1} T\ M_i \quad I_i - I_{i-1} \quad T\ M_j \quad I_k - Ij - \quad T\ m_k \quad . \quad j \quad I_k\ |Q| - j \quad \sum_{i=0}^{j-1} I_i \quad (18)$$

Since the sums are incrementally updated at every algorithm cycle, it allows a O complexity for the calculation.

4.2.5 Tabu Search Algorithm

The algorithm starts by creating a fully random solution, since any solution will work, and setting it as the current solution. Iteratively, a certain number m of random parameters for the movement are generated. For each one of the generated parameters, a movement is executed over the current solution. The resulting solutions are sorted out and the one with the best fitness is chosen (lowest execution time). If the movement that leads to it is in the Tabu list, the next solution is chosen, and so on, until a solution, which is not tabued, is found, or every solution is examined. The chosen solution is set as the current solution, and the correspondent movement is inserted in the Tabu list for a certain number of iterations. At the end of the iteration, the current solution is compared to the best one found and replaces it when better.

The algorithm stops when a certain number of iterations k is elapsed and no improvements were made to the current solution.

5 Simulations

In order to validate our approach, several simulations were conducted using SchedSim, a multicomputer environment simulator written by Mello *et al.* [6]. This simulator allows the execution and comparison of different scheduling policies.

SchedSim was written in Java language, and has an interface, *SchSlowMig*, which allows the implementation of scheduling algorithms. Once implemented, the simulator will request computers to each incoming process, hence allowing the scheduling policy to take place.

SchedSim use probability distribution functions to simulate the application arrival as well as its behavior. It is parametrized using the number of computers in the network and their characteristics, the number of applications, processes per application and the network communication costs. It executes by generating applications according to probability distribution functions and issuing requests to the scheduler to distribute them. The simulation calculates all the time spent in CPU, memory, hard disk and network operations, returning the average application response time (or execution time) and the standard deviation. It also computes the time spent by the scheduling policy.

5.1 Parametrization

Simulations were conducted using the following environments: -node cluster and a -node grid, considering application sizes (number of processes) of and processes. The number of applications ranged from to . The simulations considered inter-message intervals of around second (messages were generated according to an exponential probability distribution function with average of , million of instructions per second – this considers the number of instructions between consecutive message events). These messages are sent and received through the network (by the processes of the application), characterizing interprocess communication.

Applications dynamically arrive at the environments after they start executing. The application allocation cannot be static, since multiple runs are necessary to acquire application specific parameters.

The capacity of every computing resource was generated by using Normal probability distribution functions with the following averages:

1. Processing capacity – , MIPS (million of instructions per second);
2. Main memory capacity – , MBytes;
3. Virtual memory – , MBytes;

In the cluster scenario, all nodes are interconnected through the same switch and, therefore, the bandwidth and latency is the same. In the grid scenario, all computers are physically organized into local

area networks with one to five computers each (what tends to be closer to an opportunistic grid).

The cluster nodes were connected through a Gigabit Ethernet infrastructure with RTT (Round-Trip Time) of . second, obtained from a benchmark by Mello and Senger [6]. On the other hand, experiments were carried out to characterize the latency in between the grid nodes. Local, metropolitan and worldwide networks were analyzed and, according to results, we defined an exponential probability distribution function (with average . second) to model the network latency behavior.

The Tabu-based logical network partitioning algorithm was implemented to group the environment nodes into smaller networks with at most computers. The parameter m, this is the total number of random movements per iteration, was set to . The number of total iterations to the stop criterion k was set to . After k iterations, if no better solution is found, the algorithm stops. Simulations were executed in the SchedSim simulator [6] comparing our Tabu Search approach to other scheduling policies.

5.2 Comparison to other Scheduling Policies

The proposed Tabu Search approach was compared, by using simulations, to the following policies: Random [21], Route [7] and RouteGA [4]. They were evaluated under the same conditions. It is important to make clear that only the approach proposed here makes the logical partitioning. This partitioning does not physically modify the environment (but just reduces the search space for feasible optimization solutions).

The Random algorithm randomly chooses a computer to allocate every process. The Route technique, proposed by Mello and Senger [7], creates neighborhoods by choosing computers with low communication cost. When an application is launched on a certain computer, it is distributed over its neighbors, considering the processing load and application runtime. The neighbor may redistribute processes according to the migration model proposed by Mello and Senger [5]. RouteGA is an extension of Route, which considers the meta-heuristic of Genetic Algorithms to select the best neighborhood to distribute processes.

5.3 Results

Results, in terms of the average execution time of applications and confidence intervals of (in seconds), are presented in Tables 5, 6, 7 and 8. The numbers represent the average response time and confidence intervals for every policy.

We observe that this instance of the Tabu Search, in most of the cases, presents better results than other techniques. The considerably better results in grids are consequence of the network logical partitioning, performed before the process scheduling. This partitioning restricts the search space and improves the overall meta-heuristic performance (it plays the important role of defining a hierarchy for scheduling). In cluster environments, the results were only slightly better than the best available technique, which is RouteGA.

We also observe that the confidence interval was considerably wide for the grid computing environment. This happens to due there are few computers in every local area network (up to) and the communication costs in between computers are high. This simulates the opportunistic-type of grid computing architecture. Besides that, applications have more processes than the number of computers per local area network, what tends to require computers in different networks and, therefore, increases the execution time dispersion. On the other hand, by executing the network logical partitioning, our Tabu Search approach considerably improves such confidence interval due to the allocation of processes in nearby regions (according the inter-computer communication costs). Such allocation, reduces the dispersion of the execution time, ensuring higher performance to applications. Consequently, by using hierarchical scheduling, we obtain application performance improvements.

Table 5: Results for Cluster with 32 computers and applications up to 32 processes

# Apps	Random	Route	RouteGA	Tabu
10	3.2493± 0.1582	2.4423± 0.2502	1.9295± 0.0423	2.1011± 0.0319
20	20.195± 0.2948	15.394± 47.857	8.2093± 0.4221	7.5915± 0.1306
30	19.975± 0.4704	15.042± 40.685	9.4902± 1.0870	7.7338± 0.2707
40	13.565± 0.1997	12.729± 11.872	9.5049± 0.2350	7.7394± 0.0929
50	41.046± 0.8057	20.865± 18.095	14.856± 1.1616	12.064± 0.3185
60	45.616± 2.2791	37.250± 33.261	22.024± 0.9666	21.576± 0.9009
70	55.718± 1.5792	42.751± 42.725	27.567± 0.7663	24.397± 0.7997
80	53.241± 1.7958	46.781± 57.306	29.107± 1.5444	25.953± 2.5727
90	120.04± 3.5824	113.19± 44.081	60.053± 4.5557	58.722± 2.9846
100	172.61± 4.9879	160.44± 48.078	85.808± 4.8041	81.985± 4.1718

The corresponding cost of each algorithm (in seconds) is presented in Figures 2, 3, 4 and 5. We observe that the Tabu Search presents lower costs when compared to other techniques such as RouteGA, while keeping costs comparable to methods such as Random

Table 6: Results for Cluster with 32 computers and applications up to 128 processes

# Apps	Random	Route	RouteGA	Tabu
10	4.5429± 0.4759	3.8921± 0.2230	3.4337± 0.1812	3.7729± 0.2286
20	13.575± 5.6920	12.463± 2.9334	10.806± 0.7079	10.069± 0.9039
30	14.706± 7.8257	13.025± 7.7339	11.331± 1.5323	10.119± 0.8143
40	15.157± 3.2099	15.009± 7.6891	14.575± 1.2135	11.516± 0.6116
50	15.660± 1.7992	14.345± 0.4912	15.953± 0.7219	12.337± 0.3532
60	78.386± 385.25	69.323± 87.872	40.034± 9.4197	42.242± 10.772
70	81.029± 351.45	70.640± 40.463	50.024± 4.6634	44.536± 6.7419
80	83.091± 238.78	75.105± 91.058	51.633± 11.584	46.073± 16.828
90	328.59± 568.13	311.69± 138.08	172.09± 31.060	167.17± 33.258
100	544.14± 2085.7	533.22± 269.40	292.48± 55.243	279.83± 33.044

Table 7: Results for Grid with 512 computers and applications up to 32 processes

# Apps	Random	Route	RouteGA	Tabu
10	95.906± 13208.	104.04± 9734.7	19.668± 111.25	7.5879± 5.1062
20	263.12± 42801.	332.69± 38046.	39.809± 913.85	20.128± 62.788
30	337.78± 54501.	299.05± 24436.	41.720± 740.04	21.288± 124.46
40	239.23± 29344.	236.32± 21958.	27.513± 480.62	15.434± 41.274
50	323.04± 26934.	291.93± 20039.	27.968± 259.21	19.308± 44.102
60	434.90± 24922.	636.46± 153632	83.579± 961.32	53.369± 299.19
70	617.47± 399973	598.82± 383541	178.80± 59228.	62.614± 546.76
80	988.35± 167369	966.52± 170541	488.94± 125555	48.155± 121.96
90	619.89± 63420.	551.21± 82274.	155.03± 5075.9	69.901± 435.07
100	558.69± 134920	475.73± 62375.	148.71± 1082.6	70.304± 76.408

and Round-Robin in most of the situations.

6 Conclusions

Motivated by the exponential complexity and meta-heuristics results, this paper proposes a new process scheduling approach using the Tabu Search. Results confirmed that the proposed approach presents better results than other techniques, considering different environments. The approach also reasonably keeps low costs to compute the scheduling optimization, due to its efficient O fitness function, and the logical network partitioning performed beforehand.

In clusters with computers, the approach slightly outperforms other techniques. It does keep low costs, mostly because of the low complexity fitness function.

Table 8: Results for Grid with 512 computers and applications up to 128 processes

# Apps	Random	Route	RouteGA	Tabu
10	67.42± 2294	74.14± 5077	66.161± 2461	29.29± 215
20	123.7± 6650	137.6± 196	60.475± 1563	34.83± 445
30	153.9± 4346	157.8± 5753	51.146± 1425	31.35± 364
40	136.0± 4213	140.9± 5067	50.963± 1661	31.10± 546
50	125.3± 2991	147.8± 4148	41.94± 908.2	25.51± 380
60	400.0± 16840	367.3± 17652	165± 5570	141± 2859
70	624± 428935	647± 421454	348± 253740	211± 41041
80	1207± 477025	1211± 488172	968± 482697	365± 353137
90	548.6± 60275	530.5± 44366	261± 14388	187± 4033
100	545.4± 23345	509.2± 23741	266± 3348	188± 1271

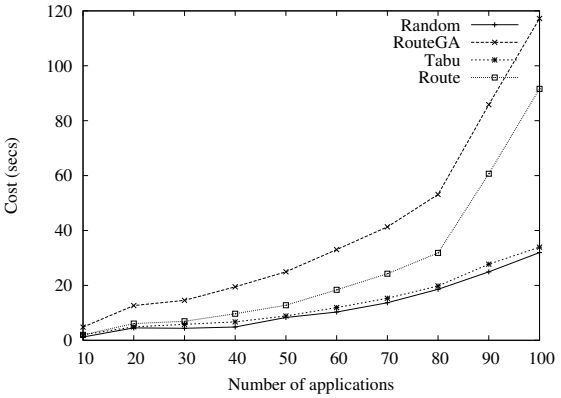

Figure 2: Costs for the 32-node Cluster with applications up to 32 processes

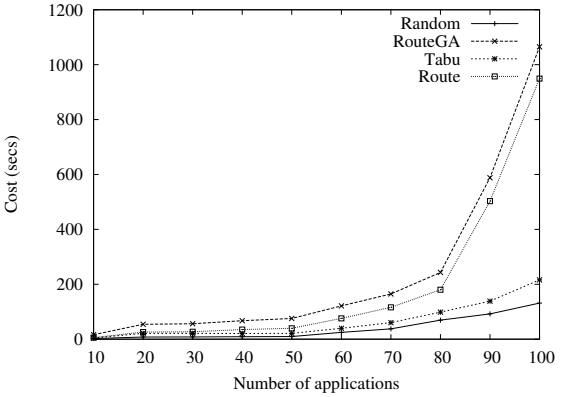

Figure 3: Costs for the 32-node Cluster with applications up to 128 processes

In larger clusters and grids, the Tabu Search presents a considerably better result and carry on keeping low

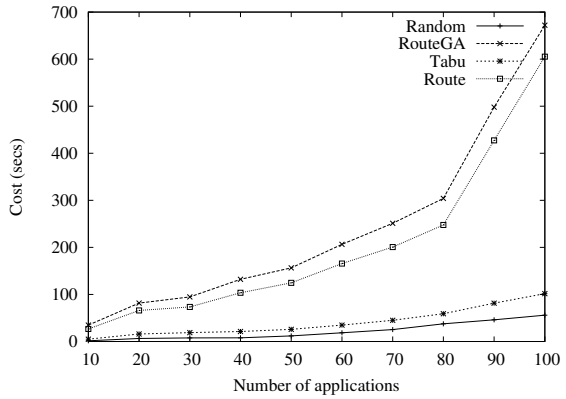

Figure 4: Costs for the 512-node Grid with applications up to 32 processes

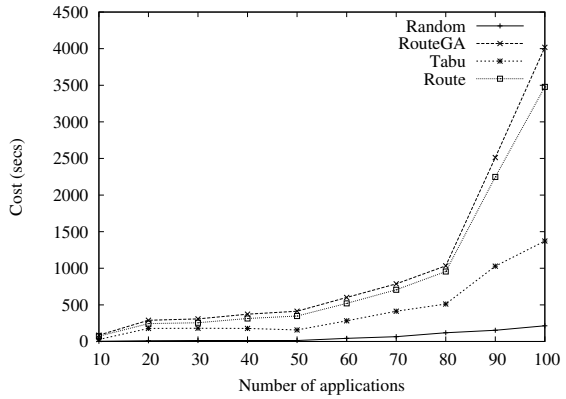

Figure 5: Costs for the 512-node Grid with applications up to 128 processes

costs, due primarily to the logical partitioning of the network in the first stage of the algorithm. Those results show a general superior performance of Tabu Search in the considered environments.

The low costs observed in the simulations are a consequence of the flexibility of the stop criteria combined to a low strictness in the movement, and the low complexity fitness function adopted. The partitioning of the network environment also considerably reduces the search space (as observed in the section 2), therefore allowing the usage of the divide-and-conquer approach which is faster.

The tolerant stop criterion allows the algorithm to stop quickly, choosing a solution without an exact search. The reduced strictness diminishes the number of evaluated candidate solutions what decreases the time consumed by the algorithm, at the expense of wasting potentially good solutions.

Results presented in Table 8 confirm that, for grids, the Tabu Search presents better results, due to the environment partitioning. Other techniques have been unable to explore the search space so efficiently, as they do not employ such method. The exponentially larger unpartitioned search spaces are much harder to explore using meta-heuristics, exponentially increasing the costs to achieve high quality results. Furthermore, the logical partitioning also considerably reduced the confidence interval of execution time for grid environments, ensuring higher performance to applications. This confirms that by using hierarchical scheduling, we obtain application performance improvements.

References

[1] A. Abraham, R. B. and Nath, B. Nature's heuristics for scheduling jobs on computational grids. In *8th IEEE International Conference on Advanced Computing and Communications (ADCOM 2000)*, India, 2000.

[2] Aarts, E. and Korst, J. *Simulated annealing and Boltzmann machines: a stochastic approach to combinatorial optimization and neural computing*. John Wiley & Sons, Inc., New York, NY, USA, 1989.

[3] Aha, D. W., Kibler, D. F., and Albert, M. K. Instance-based learning algorithms. *Machine Learning*, 6:37–66, 1991.

[4] de Mello, R. F., Filho, J. A. A., Senger, L. J., and Yang, L. T. RouteGA: A Grid Load Balancing Algorithm with Genetic Support. In *21st International Conference on Advanced Networking and Applications*, pages 885–892, Aug. 2007.

[5] de Mello, R. F. and Senger, L. J. A new migration model based on the evaluation of processes load and lifetime on heterogeneous computing environments. In *International Symposium on Computer Architecture and High Performance Computing - SBAC-PAD*, page 6, 2004.

[6] de Mello, R. F. and Senger, L. J. Model for simulation of heterogeneous high-performance computing environments. In *7th International Conference on High Performance Computing in Computational Sciences – VECPAR 2006*, page 11, 2006.

[7] de Mello, R. F., Senger, L. J., and Yang, L. T. A Routing Load Balancing Policy for Grid Computing Environments. In *The IEEE 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, pages 1–6. IEEE Computer Society Press, Apr 2006.

[8] Dorigo, M. and Stützle, T. *Ant Colony Optimization*. MIT Press, 2004.

[9] Feitelson and Nitzberg. Job characteristics of a production parallel scientific workload on the NASA ames iPSC/860. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing – IPPS'95 Workshop*, volume 949, pages 337–360. Springer, 1995.

[10] Freeman, J. A. and Skapura, D. M. *Neural networks: algorithms, applications, and programming techniques*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1991.

[11] Garey, M. R. and Johnson, D. S. *Computers and Intractability : A Guide to the Theory of NP-Completeness*. Series of Books in the Mathematical Sciences. W. H. Freeman, January 1979.

[12] Glover, F. and Laguna., M. *Tabu Search*. Kluwer, Norwell, MA, 1997.

[13] Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Kluwer Academic Publishers, Boston, MA., 1989.

[14] Harchol-Balter, M. and Downey, A. B. Exploiting Process Lifetimes Distributions for Dynamic Load Balancing. *ACM Transactions on Computer Systems*, 15(3):253–285, August 1997.

[15] Hopfield, J. J. Neural networks and physical systems with emergent collective computational abilities. *Neurocomputing: foundations of research*, pages 457–464, 1988.

[16] Lageweg, B. J. and Lenstra, J. K. Private communication, 1977.

[17] Pineau, J.-F., Robert, Y., and Vivien, F. The impact of heterogeneity on master-slave scheduling. *Parallel Comput.*, 34(3):158–176, 2008.

[18] Senger, L. J., de Mello, R. F., Santana, M. J., and Santana, R. H. C. An on-line approach for classifying and extracting application behavior on linux. In *High Performance Computing: Paradigm and Infrastructure (to appear)*. John Wiley & Sons, 2005.

[19] Senger, L. J., Mello, R. F., Santana, M. J., and Santana, R. H. C. Aprendizado baseado em instâncias aplicado à predição de características de execução de aplicações paralelas. *Revista de Informática Teórica e Aplicada*, 14:44–68, 2007.

[20] Senger, L. J., Mello, R. F., Santana, M. J., Santana, R. H. C., and Yang, L. T. Improving scheduling decisions by using knowledge about parallel applications resource usage. In *High Performance Computing and Communications: First International Conference (HPCC), LNCS 3726*, volume 3726, Sorrento, Itália, September 2005.

[21] Shivaratri, N. G., Krueger, P., and Singhal, M. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, 1992.

[22] Silva, F. A. B. D. and Scherson, I. D. Improving Parallel Job Scheduling Using Runtime Measurements. In Feitelson, D. G. and Rudolph, L., editors, *Job Scheduling Strategies for Parallel Processing*, pages 18–38. Springer, 2000. Lect. Notes Comput. Sci. vol. 1911.

[23] Tanenbaum, A. S. *Modern Operating Systems*. Prentice Hall, New Jersey, 1992.

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