A Multiagent-Based Approach to the Grid-Scheduling Problem

Mauricio Solar, Jorge Rojas, Marcelo Mendoza, Raúl Monge
Departamento de Informática
Universidad Técnica Federico Santa María,
Santiago, Chile, 7660251
{*msolar, jorge.rojas, mmendoza, rmonge*}*@inf.utfsm.cl*

Víctor Parada
Departamento de Ingeniería Informática
Universidad de Santiago de Chile,
Santiago, Chile, 9170124
victor.parada@usach.cl

Abstract

Computer grids are systems containing heterogeneous, autonomous and geographically distributed nodes. The proper functioning of a grid depends mainly on the efficient management of grid resources to carry out the various jobs that users send to the grid. This paper proposes an algorithm that uses intelligent agents in each node to perform global scheduling in a collaborative and coordinated way. The algorithm was implemented in a grid simulation environment that allows the incorporation of intelligent agents. This simulation environment was designed and developed to run and analyze the behavior of the proposed algorithm, which outperforms the numerical performance of two well-known algorithms in terms of balancing the load and making use of the grid's capacity without giving preference to any node.

Keywords: Grid computing, Multiagents, Scheduling.

1. Introduction

A distributed system is a set of independent machines that are presented to the user as a single computer that provides different services [1]. Examples of these systems are computer clusters, peer-to-peer networks and computer grids, among others. These systems, which share hardware and software resources, are managed by a software layer (*middleware*) that presents them as a unified resource to the user, who should not need to worry about the system's architecture, how to locate the provided services, or where to carry out specific tasks.

In a cluster, resources are available to the scheduler at all times and do not vary their availability dynamically, as is done by a grid's resources. In this sense, the prediction of the performance of a grid becomes more complex due to the heterogeneity and autonomy of its resources [2].

A grid is a complex system in which resources are distributed geographically and are not present under the same administrative domain, which hinders the control of the connected resources and forces the scheduler to make use of the resources in the best possible way in the available time.

Grid scheduling requires a dynamic solution that constantly evaluates the global condition of the grid (to balance the load), the state of the grid's resources (such as its occupation and availability), and the communication channels (data transfer times between nodes) because the grid operates over the Internet and its resources are distributed geographically.

In summary, the grid-scheduling problem deals with assigning resources to a set of tasks that enter the grid through different nodes at any instant of time, considering availability (dynamic and autonomous) and computing capacity (heterogeneous), among other things. The different parameters and requirements relevant to the grid's clients and its resources must also be considered to ensure the quality of the services for the different actors in the grid.

This paper focuses on proposing a solution that is adapted to the characteristics of a grid, considering the problem from the perspective of the grid's resources as well as the perspective of the users who consume its resources. A decentralized algorithm is proposed based on collaborative agents capable of dynamically generating fault-tolerant task scheduling within the expected deadlines while remaining scalable.

The following section presents a description of the problem together with the concepts required to contextualize the developed approach. In section 3, the information necessary to understand the modeling, design and

implementation of the solution is provided, and the assumptions made and considerations that must be taken into account in the work are explained. Section 4 presents the environment of the test and the workload used in the test cases. The results obtained from the different test cases are also presented with their corresponding analyses. The last section presents the conclusions.

2. The Grid Scheduling Problem

2.1. Grid architecture

A grid is heterogeneous, dynamic and geographically distributed, and its computing capacity and quality of service are associated with and limited to the availability, capacity and performance of its resources. A grid provides services that depend on the purpose for which the grid was constructed [2], such as Computational Grids, which provide computing capacity to execute applications and processes in parallel to reduce total execution time; Data Grids, which provide large-capacity distributed storage services; and Service Grids, which provide services that are possible due to the aggregation of the different resources of the grid, i.e., collaborative work applications, distributed multimedia services, among others.

Regardless of the type of service provided by a grid, its architecture is described in terms of layers, which fulfill such functions as the following:

- **Application layer:** The layer visible to the user that provides the services available from the applications to which it has access.
- **Middleware layer**: Software between the operating system and the services provided by a grid resource.
- **Resource layer:** Computers, supercomputers, storage systems, data catalog, databases, sensors, etc.
- **Network layer:** The stable connection and communication among the grid's resources.

The middleware handles authentication, authorization and safety in the grid, and provides access to data, applications, and services. The scheduling algorithms are implemented in this layer to administer the resources in an efficient and coordinated manner: for example, finding the resources to execute a task requested by a user, optimizing the use of dispersed resources, organizing efficient access to the data, monitoring the progress of the work that is being executed, managing recovery after faults, and reporting the end of a task.

The middlewares most widely used in grid projects are the following:

- Alchemi: Open code framework to generate a grid with the capacity to develop applications [3] from different machines in a network.
- Unicore (Uniform Interface to Computing Resources): Unicore allows distributed computation and data resources to be available in intranets and on the Internet safely, without problems for the users [4].
- Globus: A set of open source tools used to build grid systems and applications [5]. The services run on Unix platforms.
- GridBus: A set of open source tools oriented to services and technologies for *eScience* and *eBusiness* applications [6].
- Other less frequently used middlewares are Aneka [7], gLite [8], OGSA-DAI [9], Grid Datafarm [10], Legion [11], NorduGrid middleware [12], Storage Resource Broker [13], ProActive [14], Vishwa [15], GrelC [16], Fura [17], etc.

2.2. Grid simulation environments

An environment that simulates a real grid offers the tools to implement a scheduling algorithm that generates information related to the performance of the algorithm and its results. The most widely used simulation environments are the following:

- SimGrid: A set of tools that provides functionalities to simulate applications distributed in heterogeneous and distributed environments [18]. Its objective is to facilitate research in the field of scheduling parallel and distributed applications over distributed computational platforms that go from simple job networks to computational grids.
- GridSim: GridSim allows modeling and simulation of entities such as distributed user systems, applications, resources, and resource schedulers for the design and evaluation of scheduling algorithms. It provides the

capacity to create different types of heterogeneous resources that can be added using schedulers [19].

- Bricks: System for evaluation of the performance of grid scheduling algorithms that allows analysis and comparison of various types of scheduling in high-performance computational arrangements [20].
- MicroGrid: Online simulation for grids and distributed systems that provides online simulation of large-scale networks and resource grids [21].
- Alea 2: Simulator for studying different scheduling approaches in a grid [22]. It considers characteristics such as resource heterogeneity, dynamism, and task arrival in processing time, but it does not allow the incorporation of agents into algorithms.

2.3. Existing approaches to the grid scheduling problem

A simple case of scheduling in a grid is one in which the resources are always available and the number of tasks is always the same; however, even in such a case, the problem is NP-complete [23].

Several approaches have been proposed in the literature to tackle the grid scheduling problem, such as classical algorithms that use known scheduling policies (*First Come First Served FCFS*, *Greedy algorithm*, *Tabu search*, and *Simulated Annealing*, among others). Other proposals have involved distributed solutions that use intelligent agents, ant colonies, genetic algorithms, etc. For example, [24] proposes a scheduling algorithm using ant colonies based on a node reputation fuzzy model to make better decisions. The algorithm proposed in [25] is focused on monitoring resources to distribute tasks by balancing and adjusting the load at the nodes. In [26], agents are used that represent resources and tasks and can learn to negotiate adaptively, reducing the search space by means of a modified learning algorithm. The agents' state value is calculated using an approximation of a numerical function, and the efficiency and complexity balance are calculated by *simulated annealing*. In [27], a genetic algorithm is presented to propose an improvement that searches for patterns in the individuals that generate the best solutions to decrease the randomness of the mutations and find better solutions.

2.4. The problem of grid scheduling by means of multiagents

The agents are capable of forming agent societies in which interaction and organization depend on the objectives of each of the agents and the global objective to be achieved, i.e., agents can perform competitively or collaboratively in a shared environment.

When agents interact competitively, they pursue their own objectives and can impact the environment, affecting the objectives of the other agents. On the other hand, the agents can interact collaboratively to achieve a common global objective (Multiagent System, MAS). In this case, the agents do not handle all the existing information (decentralized information) or do not have the ability to perform all the actions. That is, an agent of the system is unable to solve the problem by himself. Therefore, the agents develop social skills to communicate information and intentions with the aim of organizing with other agents [28].

Several behaviors found in the agent societies of a MAS have common characteristics, in spite of having been constructed to solve different problems. When an abstraction is made in a communication between agents [29], communication patterns are found that resemble the social organizational structures [30]. The patterns used depend on the characteristics of the problem to be solved. These patterns can be standardized and described with the purpose of facilitating the development of future MASs because they are generic solutions that facilitate reuse and can be applied to known problems.

In this paper a grid-scheduling algorithm is proposed using a MAS with the purpose of solving the grid-scheduling problem.

The proposed algorithm approaches the problem globally, i.e., it performs a distribution of tasks that go through the entry points to the available nodes that execute those tasks. Once the tasks have been assigned to the nodes, they are distributed over the node's available machines. The local scheduling of these machines to the processors is beyond the reach of the proposed algorithm because in a real environment, these policies belong to the local operating system.

In a grid, resources are not necessarily found in the same administrative domain and are autonomous; therefore, they enter and exit the grid according to their availability, hindering resource control and management. For that reason, a decentralized and cooperative algorithm is proposed in which each node of the grid contributes to the global scheduling with a part of the solution. For this purpose, there are autonomous agents at each node that

process the tasks sent to the node, where they exist and collaborate with agents from other nodes to balance the load and execute the tasks with the purpose of achieving a global solution.

The solution is based mainly on agent collaboration, agents' constant evaluation of the condition of the grid, and the decisions that agents make based on their assignment policies and beliefs. On this basis, the objective is to achieve the execution of the tasks in the shortest possible time, assigning tasks intelligently to the different nodes of the grid so that the workload is distributed homogeneously between the nodes. Each node should have the largest possible workload to decrease the waiting and delivery time of the tasks and reduce the total delay time.

A grid can be considered as a finite number of N nodes. Each node is a cluster that can have a different computing capacity from the other nodes. A node $n \in \mathbb{N}$ with $n \leq \mathbb{N}$ is composed of a finite number M_n of machines, of which each has a finite number of P_{mn} processors. It is also assumed that the machines are equal within each node; therefore, the different machines' processors can process the same task in the same period of time.

All the nodes are found at the same hierarchic level and are autonomous, that is, no node is governed by another node, so all the nodes can communicate on equal terms with the others. Processing of the tasks is performed in a nonpreemptive way.

It is assumed that the nodes have no limitations to execute the tasks, i.e., all the nodes can process all the arriving tasks without distinguishing between the types of service provided by the nodes. Only the computation time required and the number of processors needed to execute the tasks, are considered.

3. Design of the Agent-Based Scheduling Algorithm

The design of the algorithm considers each agent as part of a society of agents that live in the nodes and collaborate to achieve a global schedule adequate to handle the work load in a given time period.

The agent carries out the scheduling through two behaviors that define its actions and decisions. The first is the global scheduling behavior, in which decisions about where to execute the incoming tasks are made in negotiation with the other agents. The other behavior is the local scheduling that is in charge of prioritizing the tasks that will be executed in the node and assigning resources to them so that they can be delivered in the time requested by the clients. This scheduling process is supported strongly by other agent behaviors that are in charge of keeping all the nodes' information updated in the grid through their sensors and social skills and by monitoring the tasks.

When a task is sent to the grid by a user, it can go through different intermediate processes before being executed; what is done with a task at a given time depends on the agent's behavior and its decisions. To organize the work and the flow of tasks each node handles several queues of tasks and the time the tasks spend in the queues. The task's passage between queues depends on which behavior is taking place, the evaluation of the condition of the grid and the characteristics of the tasks.

Figure 1 depicts a task's flow through the queues in a node. The various queues are described below:

Incoming Job Queue: A task enters this queue when it arrives at a node; it contains tasks that have not been processed by any behavior of the agent.

Local Job Queue: Tasks enter this queue when the global scheduling process indicates that they will be processed locally. These tasks are revised by the local scheduling behavior.

Local Monitoring Queue: A task starting its execution enters this queue.

Negotiations Queue: The tasks that do not enter the local job queue enter this queue to negotiate their execution in another node. When a node accepts remote execution, the task is sent to this node (it enters the Local Jobs of that node).

Remote Monitoring Queue: Tasks that will be executed in another node enter this queue with the purpose of following up on the progress and state of the task.

Critical Jobs Queue: The tasks that have been waiting too long in the local job queue enter here; it is necessary to perform a rescheduling to avoid generating delays in their deliveries. These tasks arrive at this queue through the behavior of the local monitoring.

Finished Tasks Queue: This queue contains the tasks for which execution has ended.

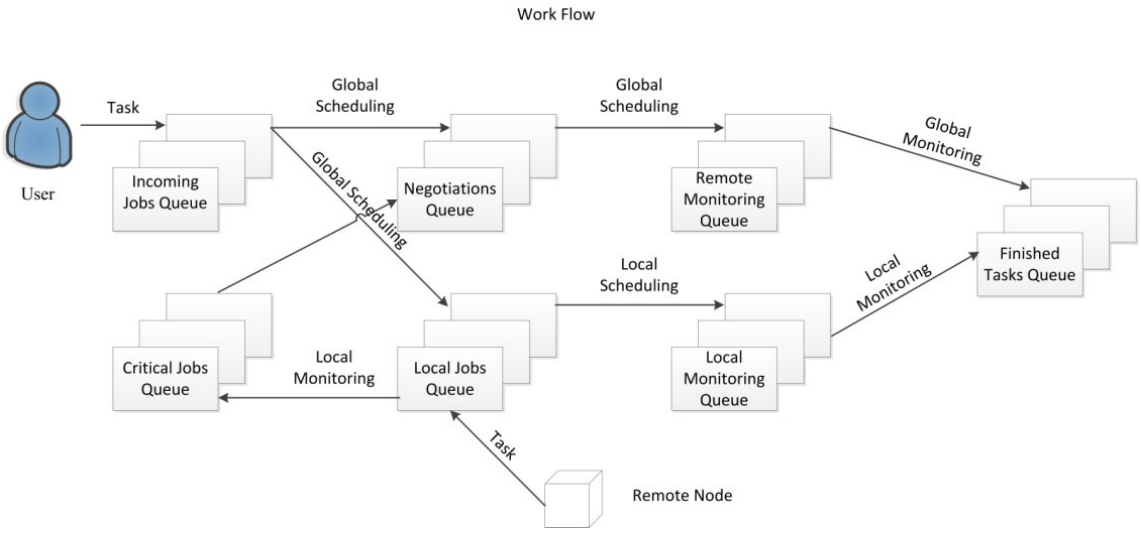

Figure 1: Flow of a task that enters through different work queues.

3.1. Social Skills

It is necessary for agents to be able to communicate with the agents of other nodes to manage and coordinate the global scheduling of the grid. The agents must also be capable of requesting and sharing information related to the different nodes, keeping the global condition of the grid updated so that the agent's decisions decisions will be as accurate as possible. Based on that requirement, two forms of interaction between the agents are recognized.

The first form of communication between the agents is the delivery of information from one agent to another without the need to maintain a conversation. This type of communication is asynchronous; an agent that needs to send information to another agent decides when to send it. This form of communication is used to report the condition of a node to the other components of the grid through a *broadcast*-type message. Another application occurs when a node executes a task requested by another agent and reports constantly on the condition of the processing of the task.

The second form of communication is more complex because its aim is to maintain a conversation between a pair of agents of the grid to negotiate and coordinate the execution of the grid's various tasks. To achieve this purpose, the negotiation agents communication pattern is used; i.e., in the grid, no agent has power over another, and to carry out tasks, the agents collaborate and agree on the execution.

To summarize, an agent that needs help starts a conversation with an agent that it determines can help. The receiving agent evaluates the petition of the initiating agent and delivers a reply according to its beliefs and its internal condition; i.e., the receiving agent can reject the request, accept it, or ask the starting agent to wait for its answer until its evaluation is processed.

If the agents agree to execute the task, the negotiation conversation between the two agents ends and the executing agent starts processing the task, going into a state of constant delivery of information to the starting agent.

Once the task has ended, the executing agent tells the starting agent that its task is ready. The starting agent then requests the results of the task; once they arrive, it confirms their receipt. If the results do not arrive in the proper form, their delivery is requested again.

The steps described above comprise the communication process between two agents of the grid that converse to perform a task. This process is depicted in the communication diagram of Figure 2.

The negotiation process between the agents goes through different states that depend on the roles played by the agents in the conversation and the events that take place during the negotiation.

When an agent is performing task assignment and decides that the best location to perform a task is an external node, it starts a conversation with the agent belonging to the node that it determines has the best conditions for the task's execution. Figure 3 shows the states through which the conversation passes for the agent that starts the conversation and the different events that trigger the transition between one state and another.

The agent starts the conversation by sending a message requesting the execution of a task to the agent of the node chosen for the job. The conversation immediately begins in the **Asking** state and waits for a message confirming the arrival of the request. Once the confirmation message is received, the conversation goes to the **Waiting for Evaluation** state. In this state, the agent again waits for a reply from the other agent. The starting agent constantly evaluates the current state of the task attempting to be executed at the other node because if the reply takes too long, it will be necessary to decide whether to cancel the request and reassign it to another node or put it in the local queue. If the request is canceled, a *TimeOut* is generated and the conversation goes into the **Canceled** state, sending an information message to the agent evaluating the execution to cancel the process. Next, the agent carries out the local process of reassignment and the conversation enters the **Finished** state. On the other hand, the agent can receive three different replies. The request can be rejected by the counterpart agent, causing the agent to start the process of task reassignment and changing the state of the conversation to **Finished**. Another possible reply is the counterpart's request that the assigning agent wait, indicating that the agent receiving the request is too busy to decide at that moment. While it waits, the first agent keeps the conversation in the **Waiting for Evaluation** state. Finally, the last possible reply is a confirmation that the request was accepted, in which case the agent passes the task to a monitoring state and ends the conversation by entering the **Finished** state.

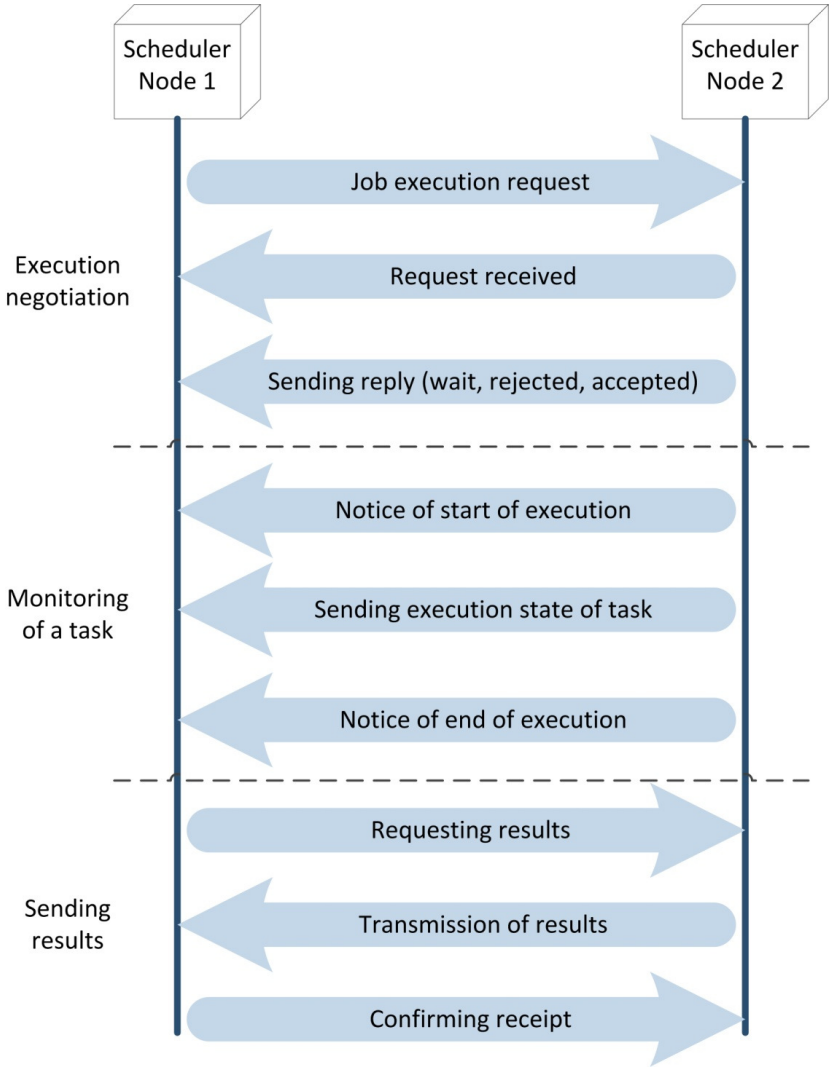

Figure 2: Communication between two agents: negotiation, execution and task monitoring.

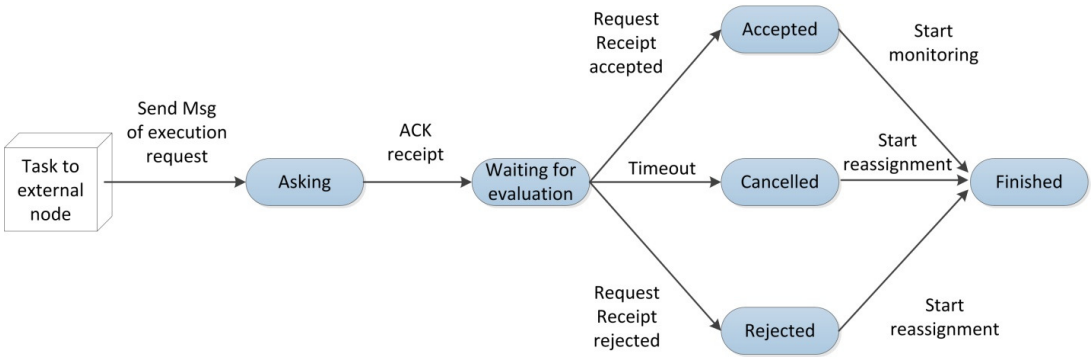

Figure 3: States of the conversation from the viewpoint of the starting agent.

When an agent notices the arrival of a request message, it starts a conversation with the agent that sent the message. Figure 4 shows the different states through which an agent passes from the time it receives a request until it makes a decision and puts an end to the conversation.

When a conversation is created, it starts in the **Requested** state. When the agent decides to carry out the negotiation process and becomes aware of the request, it sends a message to the agent that started the conversation, notifying the starting agent that the request message has been received. The agent immediately starts the evaluation process of the request, changing its state to **Evaluating**. Once the agent enters this state, it can deliver three different replies to the node that made the request. The first option is to notify the starting agent that the task cannot be carried out yet but that it will wait until the queue of incoming tasks and the queue of tasks of the local node are empty to decide whether to execute the task. While the agent completes the pending local assignments, the state of the conversation remains at **Evaluating** and the state of the task queues is revised constantly to identify the moment at which the conversation can be continued.

Once the agent is ready to make a decision, it can decide to accept or reject the requested task depending on the internal state of the node in comparison with the grid. To do so, the agent sends the reply (Accepted or Rejected) to the agent that made the request. In both cases the conversation is **Finished**, but if the job was accepted, the task is placed at the end of the node's local queue to be processed as soon as possible.

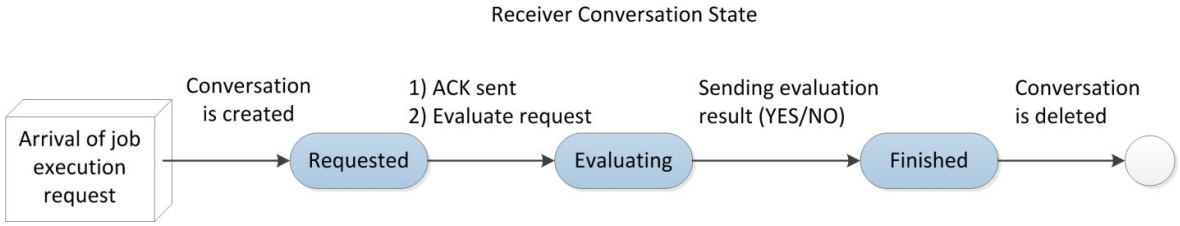

Figure 4: States of the conversation from the viewpoint of the receiving agent.

3.2. Collaborative Global Scheduling

The information presented by a task in this model is as follows:

- ***Release Time*:** When the task is ready to be processed.
- ***Processing Time*:** Time taken to execute the task.
- ***Requested Time*:** Amount of time requested by the user for the task to be executed.

The scheduling process begins with the decision of where the task should be executed, considering the local state of the node and the global state of the grid. For this purpose, it is necessary to have a procedure in place for the prioritization of the grid's resources, depending on the state of each node and of the task that it is asked to execute.

Each node handles information that allows it to be distinguished from the others in different dimensions. The size of the node and its computing capacity have a direct relationship to the contribution to occupation generated by the

use of a node's processor to the global state of the node to which it belongs and to the grid. Based on this information, the estimator referred to as Cost of Occupation of a Processor (*COP*), which indicates the percentage of contribution to occupying a processor of a node *i* respective of the total occupation of the node to which it belongs is defined as follows (Eq. 1):

$$COP_i = \left(\frac{1}{P_{ni}} \right) * 100 \quad \text{(Eq. 1)}$$

Additionally, there is information available on the state of the node during the time of execution, depending on the workload that is being processed at that time. From this information, we can obtain the current occupation of the node (Eq. 2), the number of tasks in the job queue, the number of tasks that have arrived at the node and are waiting to be processed, and the estimated time for the node to obtain the available computing capacity. According to the servers of the node and their processors, the waiting time for some of the processors to become free is estimated as follows:

$$Oc = \left(\frac{Occupied\ \text{Pr}\ ocessors}{Total\ \text{Pr}ocessors} \right) * 100 \quad \text{(Eq. 2)}$$

One node records the number of tasks that have arrived in the period of time that it has been online, which allows the calculation of the rate of arrival of the tasks to the node, as shown in Eq. 3:

$$\lambda = \left(\frac{Number\ of\ Tasks\ \text{R}eceived}{T} \right) \quad \text{(Eq. 3)}$$

The algorithm is adjusted as the rate of arrival of tasks varies over time. Use of *Poisson's* distribution allows the algorithm to know the probability that a task will reach one of the nodes in time *t*, depending on the rate of arrival of tasks recorded in the node (Eq. 4):

$$P(t, \lambda) = \frac{e^{-\lambda} \lambda^t}{t!} \quad \text{(Eq. 4)}$$

The process by which the nodes determine priority compares the states of the pairs of nodes to decide where specifically to execute a task, as follows:

Nodes with equal occupation less than 100%

If two remote nodes have computing capacity available, the node with the least processor occupation cost, i.e., with greater computing capacity, receives priority. If one of them is local, it receives priority regardless of its occupation, reducing costs such as negotiation time, transmission time, traffic load between nodes, etc.

Nodes with occupation equal to 100%

When both nodes are at maximum capacity, it is necessary to ensure that the task begins processing as soon as possible. The first discriminator is to prioritize the node that requires the least time to become available.

It is unlikely that the time remaining to become available will be equal in two nodes, but if this occurs, the node with the fewest incoming jobs in its queue is prioritized. The rates of the arrival of tasks are compared and the node with the lowest rate is chosen.

Nodes with different occupations

If the nodes have different occupations, the future occupation is calculated. In other words, a calculation is made of the effect that assigning the task would have on the current occupation of the node; the probability that a different task arrives at the node during the task execution time; and the occupation generated by the task, considering the number of processors it uses. Based on these criteria, priority is given to the node with the lowest expected occupation (Eq. 5):

$$ExpectedOccupation = Oc + P_j * COP_j * P(t_i, \lambda) \quad \text{(Eq. 5)}$$

where t_i, is the duration of task *i* and P_i is the number of processors used by that task.

The *ExpectedOccupation* indicator allows a finer load balance to be made; i.e., if the difference between the occupations is large, priority will be given to the node with the lowest occupation. As the similarity of the occupations increases, the estimator starts to be affected by the size of the node (occupation generated by a

processor), prioritizing the node that will have the lowest occupation if the task is processed in it. If this similarity continues increasing, the indicator starts to be affected by the probability of more tasks arriving at that node in the time taken for the execution of the task. For this reason, a node that has a low rate of task arrival has a lower probability of continuing to increase its occupation as a task is executed. In this way, the more passive nodes tend to help the nodes that have a more active arrival rate.

This comparison logic allows decisions to be made considering the grid's current characteristics, the learning about the activity of the nodes that has taken place, and the estimation of the future state of the grid, allowing the generation of an algorithm adaptable to the specific situations of the grid's load.

The main objective of the global scheduling behavior is to consider the tasks in the incoming jobs queue at a given time and choose the node in which each will be executed. To accomplish this objective, the agent decides in this behavior whether to put the task in the local queue or start a conversation with one of the nodes to negotiate its execution remotely (see Figure 1).

From the time the agent is created, it can execute this behavior again at any time; therefore, between one execution and another, pending tasks may remain in a state of negotiation. This occurs because communication between agents is asynchronous; before processing new incoming tasks, it is necessary that the agent finish negotiating the waiting tasks.

The first step of the global scheduling behavior is to review whether there are pending negotiations. If so, it must continue with each pending negotiation. This process reviews the conversations started by the agent and the negotiations started by a remote agent. Once progress has been made in the conversation of each pending negotiation, a review is performed to determine whether there are critical or pending tasks to be processed. If there are no critical tasks or incoming jobs at that time, the behavior ends and the agent decides to perform another behavior.

Once an iteration of the pending negotiations has been made, a review is performed to determine whether there are critical tasks. If critical tasks are found, they are arranged in decreasing critical order and reassigned to a node that is not the local one because if they wait in that node, they will not be processed on time. Once an iteration of the critical tasks has been made, the assignment of the tasks entering the node is performed. The nodes for each task (in order of arrival) are arranged by assignment priority, and the negotiation with the highest-priority node is begun. The list of priorities also includes the local node. When its turn comes, the task goes to the local queue. Once the task enters that queue, it is processed when the agent carries out its local scheduling behavior.

3.3. Local Scheduling Algorithm

Once tasks have been assigned to the nodes or the execution of a task belonging to another node has been negotiated, it is necessary to execute them efficiently to comply with the delivery times requested by the users.

Because a dynamic algorithm is being used, there is no information on the number of tasks that will arrive or on the state in which the node will remain over a given period. The task estimation and assignment processes must be performed during the execution time; i.e., assignment of a task, information about all the tasks existing in the waiting queue and the current state of the node must be considered.

To estimate the order of the tasks' completion, they must be evaluated and compared with those in the waiting queue. The information presented by a task in this model includes a ***Due Date***, which is the maximum delivery time in relation to the arrival time and the time requested by the user (Eq. 6).

$$Due\ Date = Release\ Time + Requested\ Time \quad \text{(Eq. 6)}$$

Considering this information and the time that has passed since the task entered the grid, the state of the task after it has ended can be estimated if it started at a specific instant. This estimation is generated dynamically over time, and it is updated so that tasks can be compared with other tasks in the local queue:

- ***Waiting Time*:** Time from the arrival to the present time without processing the task. This variable is updated dynamically over time (Eq. 7).

$$Waiting\ Time = Present\ Time - Release\ Time \quad \text{(Eq. 7)}$$

- ***Estimated End Time*:** Estimated ending time of a task if its processing begins at the present time (Eq. 8).

$$\begin{aligned} Estimated\ End\ Time &= Release\ Time + Waiting\ Time + Processing\ Time \\ &= Present\ Time + Processing\ Time \end{aligned} \quad \text{(Eq. 8)}$$

- ***Estimated Earliness:*** Anticipated delivery time in relation to the task's end time if the task started at the present time (Eq. 9).

$$Estimated\ Earliness = Due\ Date - Estimated\ End\ Time \quad \text{(Eq. 9)}$$

- ***Estimated Tardiness***: End time of the task after the due date if the task begins at the present time (Eq. 10).

$$Estimated\ Tardiness = Estimated\ End\ Time - Due\ Date \quad \text{(Eq. 10)}$$

Figure 5 shows the timeline that a task can follow, which is related to the initial information of the task plus the estimated information generated at a given time.

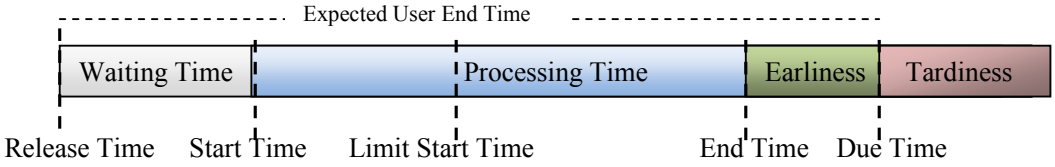

Figure 5: Timeline followed by a task in a scheduling context.

The objective of the scheduling process within the local node is to maximize the *total earliness* (or minimize the *total tardiness*). The estimated information is used for this purpose to arrange the tasks according to priority. The tasks that have the least *earliness*, i.e., which have a greater probability of exceeding their due dates, come first. The consequence of prioritizing the tasks in this way is that the tardiness is reduced (*tardiness*) and the waiting time for the tasks is reduced. However, the tasks are required to wait as necessary to prioritize those that are at risk of becoming tardy, regardless of their order of arrival.

4. Test Cases and Results

The test environment is based primarily on the characteristics of the grid used for the simulation, i.e., the number of nodes, the number of machines per node, the number of processors per machine, the number of clients, the points of entry, etc. Once the topology of the grid and the existing clients have been considered, it is necessary to provide a job load in which each task is sent to an entry point [31]. The test cases are generated from an actual grid environment to achieve tests that are close to reality. The grid used in this paper is the **Auvergrid**, described below.

4.1. The Auvergrid

The **Auvergrid** is a production platform composed of five clusters. Its machines have Pentium dual-IV Xeon 3 GHz processors and Linux operating systems (Table 1), and it uses the *middleware* LGC (*Large hadron collider computing grid project*) as part of its infrastructure.

A log file of the job load generated by the *resource manager* of the **Auvergrid** is used to identify 404 clients that send 34,7611 jobs to the grid over a period of one year. Figure 6 shows the tasks per day that arrive at the grid in one year. Four task arrival peaks can be observed that are outside the average job load, as it is uncommon for the grid to exceed 2000 tasks per day. These differences in the rates of task arrival serve to evaluate different aspects of the algorithm with a job load extracted from a real environment.

Table 1: Number of servers and processors of the specific nodes in **Auvergrid**

Cluster (node)	Number of Servers	Number of processors per server	Total number of processors
clrlcgce01	56	2	112
clrlcgce02	42	2	84
clrlcgce03	93	2	186
iut15	19	2	38
opgc	28	2	56

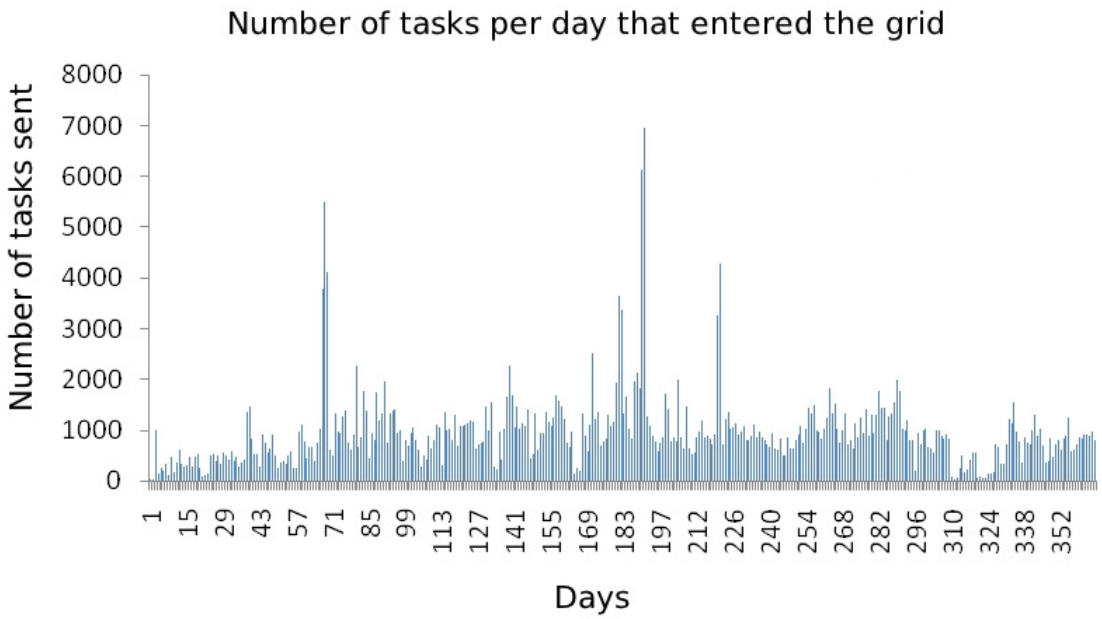

Figure 6: Number of tasks per day that enter the grid in one year.

4.2. Test Cases

Test case 1
Test case 1 considers the entry of tasks in the first 15 days of the job load file of the **Auvergrid**. The objective of this test is to analyze the behavior of the algorithm in a normal time period and to analyze the load balance generated by the algorithm because the tasks in this period enter in an unbalanced way, as shown in Table 2. The tasks enter in the first 31 days with an average of 348 tasks per day and a peak of 995 tasks on the third day.

Test case 2
The first peak of Figure 6, which occurs specifically on days 67, 68 and 69, is chosen for the purpose of analyzing the behavior of the algorithm with uncommon task income rates. Table 2 shows the number of tasks sent to the different nodes and their estimated total processing time, showing that this case enters in a more balanced manner than test case 1.

Test case 3
The largest task peak of Figure 6 appears on days 190 and 191, when 6127 and 6951 tasks arrive, respectively. This number of tasks is outside the normal arrival range of tasks. Table 2 shows that in this test case, the tasks also enter the grid in an unbalanced form.

Table 2: Job load of entry to the nodes of the grid for each test case.

	Test case 1		**Test case 2**		**Test case 3**	
Entry node	**Tasks sent**	**Processing time**	**Tasks sent**	**Processing time [days]**	**Tasks sent**	**Processing time**
clrlcgce01	2445	678 days	1912	95.06	3871	301.59 days
clrlcgce02	2433	557 days	1246	77.27	42	28.42 days
clrlcgce03	0	0 min	577	40.05	2397	70.24 days
iut15	15	3.85 min	0	0	4	4.6 min
opgc	30	5.16 min	1783	24.17	101	18.88 days
Total	**4923**	**1236 days**	**5518**	**236.57**	**6415**	**419.15 days**

The **Alea 2** grid simulation environment was used to perform the comparisons. The job load and the configuration of the grid are entered with the same files used to execute the test cases in the grid simulation environment, based on agents developed in the present paper. **Alea 2** implements several algorithms that were executed in the three test cases introduced here. Some algorithms stopped functioning due to the lack of information on the job load, and others behaved similarly to one another due to the characteristics of the grid and the tasks. This paper presents the two algorithms that showed the best behavior for the purpose of comparison with the proposed algorithm. The selected algorithms are FCFS (*First Come First Served*) and ESG-LS (*Earliest Suitable Gap – Local Search*).

4.3. Results

Figure 7 shows the occupation of the nodes in the first 18 days. It is demonstrated that the load at the nodes behaves in a balanced way at the time of execution of the tasks.

However, the assignment of the tasks respects the time windows requested by the users who send the tasks, and most of the tasks are processed and ended early, before the *due date*.

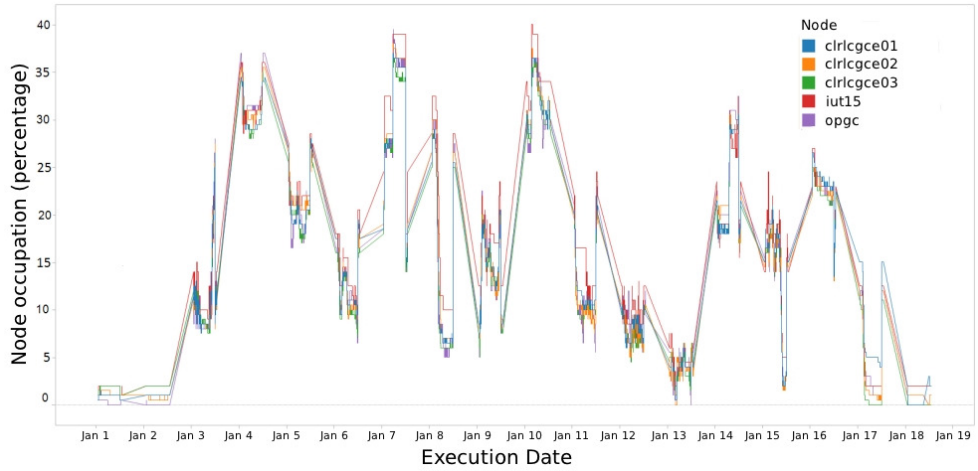

Figure 7: Occupation of the nodes for test case 1 with the proposed algorithm.

The trend in the occupation of the nodes is to behave the same way as long as the arrival rate is within a normal range; however, when the arrival rate increases, the tasks tend to be executed in the node with the largest computing capacity. This phenomenon is depicted in Figure 8. It may occur because assigning a task in that *cluster* has a lower impact on the occupation of the node, making the agents more likely to choose it because from the beginning of the increase in the arrival rate of tasks until the receiving agent updates its local state, the other agents continue trying to negotiate the execution of tasks with the node. However, in spite of this increase, the node is not saturated at 100% occupation because there is a period in which it starts executing the previously negotiated tasks and begins to update its local state to send to the other agents.

The other nodes continue in concordance with the job load because they all collaborate to carry out the same tasks.

In test case 3, one real second is equivalent to 80 msec of simulation, i.e., the simulation time is equivalent to 8% of real time, reducing the number of executions of the different agent behaviors in the same period of time. Figure 9 shows that the nodes' job load behaves in an consistent equivalent way over the entire period, and the algorithm maintains its behavior over all the test cases. Although the arrival of tasks increases the node with highest capacity is not overloaded, compared with that in test case 2. This is primarily due to the calibration made before the execution. The job load was unbalanced, but the proposed algorithm succeeded in distributing the tasks in an equitable manner.

When comparing the results of the proposed algorithm with those of the FCFS and ESG-LS algorithms, Table 3 demonstrates that the proposed algorithm balances the load better because the load distribution is more balanced and proportional to the capacity of the nodes. Table 3 shows the results of the two algorithms executed in **Alea 2** (FCFS and ESG-LS) for the three test cases. The "**MAS"** column represents the algorithm proposed in this paper.

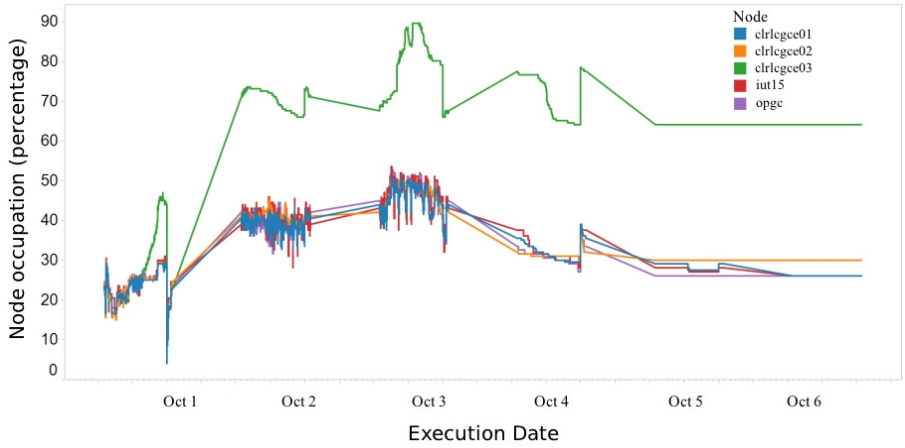

Figure 8: Occupation of the nodes for test case 2 with the proposed algorithm.

Test case 1 showed that with the FCFS algorithm, the job load is carried by the node that receives the most tasks. There are 3 unoccupied nodes (0%), which wastes computing capacity. The ESG-LS algorithm distributes the load among the nodes more efficiently than does the FCFS algorithm, making use of all of the existing capacity. ESG-LS tends to distribute the load, but in spite of being an improvement over FCFS, the difference with ESG-LS between the node with highest occupation (39.8%) and the node with lowest occupation (9.1%) is large. Although ESG-LS succeeds in preventing the nodes from becoming saturated, they have a high occupation in contrast with the proposed algorithm, which keeps the occupation values balanced at 15%.

In test case 2 using FCFS, the same phenomenon encountered in test case 1 is seen: there are wasted nodes. The problem with FCFS is that the nodes that receive fewer tasks are less occupied, causing an imbalance of the global load of the grid.

In test case 3 using FCFS, one node reaches 83.5% occupation, in contrast with another node that is not occupied (0%). With the proposed algorithm, this imbalance does not occur because the agents negotiate and collaborate with each other in order to avoid saturating the nodes of the infrastructure.

The occupation generated with ESG-LS in test cases 2 and 3 has higher values with higher task arrival rates. With the proposed algorithm, the nodes have a balanced occupation level, even in the extreme situations in test cases 2 and 3.

In brief, the proposed algorithm exceeds FCFS and ESG-LS, balancing the load efficiently and making use of the grid's capacity without giving preference to any node.

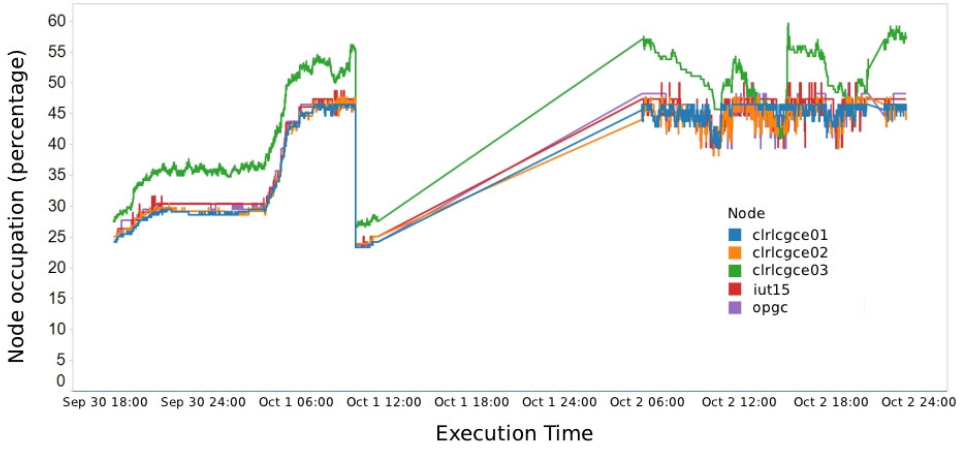

Figure 9: Occupation of the nodes generated in test case 3.

Table 3: Comparison of occupation generated by the three algorithms.

	Test case 1			Test case 2			Test case 3		
	FCFS	ESG-LS	MAS	FCFS	ESG-LS	MAS	FCFS	ESG-LS	MAS
Node	%	%	%	%	%	%	%	%	%
clrlcgce01	40.9	13.9	14.9	16.7	24.3	32.9	67.6	48.9	33.7
clrlcgce02	2.9	19.0	14.9	5.8	35.8	34.0	29.1	68.6	36.8
clrlcgce03	0.0	9.1	14.3	62.2	17.3	62.3	83.5	38.0	42.9
iut15	0.0	39.8	16.4	0.0	61.7	33.4	0.0	83.7	37.7
Opgc	0.0	28.2	14.8	0.0	47.0	32.6	4.5	77.3	37.6

5. Conclusions

This paper presents a solution to the grid-scheduling problem through the design and implementation of an algorithm using intelligent agents that collaborate with each other to find a global solution to each problem. A grid simulation *framework* was developed to incorporate intelligent agents; tests of the proposed algorithm were performed on this framework. The results show that an approach to scheduling with intelligent agents is feasible as a solution for use in computer grids.

The proposed algorithm adapts to the architecture of the infrastructure, generating a distributed, dynamic and scalable algorithm because each of the grid's nodes contributes one more agent to the solution of the problem. This distributed algorithm increases the reliability of the system and its high availability because it does not depend on a centralized scheduler to carry out the task distribution, and it diminishes the congestion in communication routes because task transfer takes place only between negotiating nodes and not through intermediaries.

The proposed algorithm performs efficient load distribution among the nodes, making use of the grid's entire capacity without leaving idle nodes. The results that it generates are acceptable within the normal functioning of a grid like **Auvergrid**.

The policy of maximizing earliness allows a reduction of the number of tardy tasks, executing critical tasks first, allowing the waiting time to be reduced, and generating a compact assignment that avoids downtime at the nodes. Combined with the balancing performed by the agents, this policy ensures to have always an occupied server when there are tasks to be executed.

Although the algorithm performs well under the assumptions made in this study, it will be necessary to carry out tests that are affected by the latency times between nodes, unexpected server freezing, and the increased computing capacity of the grid at execution time. In addition, the agents have greater development potential; they can combine with more sophisticated learning algorithms and other AI techniques to foresee the future states of the grid.

This paper assumes that all the nodes have the capacity to execute all the tasks that enter the grid, but there may be cases in which some of the nodes are incapable of carrying out all the tasks. This can impact the load balance because the intelligent agents may find the negotiation set limited depending on the tasks that they have to execute.

More intelligence can be added to the agents' decision-making method. Although the agents consider the rate of arrival and make probability calculations, learning and more complex decision-making processes can be added to combine the intelligent agents with other AI techniques and improve the quality of the solutions.

Acknowledgements

This work was partially funded by DGIP-UTFSM, Project No. 241142. The fifth author thanks to the Complex Engineering Systems Institute (ICM: P-05-004-F, CONICYT: FBO16).

Bibliography

[1] Tanenbaum, A. and van Steen, M. *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002.

[2] Krauter, K., Buyya, R. and Maheswaran, M. A taxonomy and survey of grid resource management system for distributed computing. John Wiley & Sons. *Software: Practice and Experience*, Vol. 32, pp. 135-164, 2001.

[3] Alchemi .NET Grid Computing Framework. <http://sourceforge.net/projects/alchemi/>

[4] UNICORE (Uniform Interface to Computing Resources). <http://www.unicore.eu/>

[5] Globus. <http://www.globus.org/>

[6] The GridBus Middleware. <http://www.cloudbus.org/middleware/>

[7] Aneka: Enabling .NET-based Enterprise Grid and Cloud Computing. <http://www.manjrasoft.com/products.html>

[8] gLite - Lightweight Middleware for Grid Computing. <http://glite.cern.ch/>

[9] OGSA-DAI. <http://www.ogsadai.org.uk/>

[10] Grid Datafarm, Gfarm File System. <http://datafarm.apgrid.org/>

[11] Legion, Worldwide virtual computer. <http://legion.virginia.edu/>

[12] Nordugrid Advanced Resource Connector. <http://www.nordugrid.org/arc/>

[13] SRB – The DICE Storage Resource Broker. <http://www.sdsc.edu/srb/index.php/Main_Page>

[14] ProActive - Professional Open Source Middleware for Parallel, Distributed, Multi-core Programming. <http://proactive.inria.fr/>

[15] Vishwa: Peer-to-Peer Middleware for Grid Computing. <http://dos.iitm.ac.in/Vishwa/>

[16] Grid Relational Catalog Project (GRelC), an easy way to manage grid databases. <http://grelc.unile.it/home.php>

[17] Fura, a self-contained grid middleware. <http://fura.sourceforge.net/>

[18] SimGrid project. <http://simgrid.gforge.inria.fr/>

[19] GridSim: A Grid Simulation Toolkit for Resource Modelling and Application Scheduling for Parallel and Distributed Computing. <http://www.cloudbus.org/gridsim/>

[20] Bricks: A Performance Evaluation System for Grid Computing Scheduling Algorithms. <http://ninf.apgrid.org/bricks/>

[21] MicroGrid: Online Simulation Tools for Grids, Distributed Systems and the Internet. <http://www-csag.ucsd.edu/projects/grid/microgrid.html>

[22] ALEA 2, GridSim based Grid Scheduling Simulator. <http://www.fi.muni.cz/~xklusac/alea/>

[23] Mary Saira Bhanu, S. and Gopalan, N.P.: A Hyper-Heuristic Approach for Efficient Resource Scheduling in Grid. *Int. J. of Computers, Communications & Control*, Vol. III, No. 3, pp. 249-258. 2008.

[24] Zhou, Z., Deng, W. and Lu, L. A Fuzzy Reputation Based Ant Algorithm for Grid Scheduling. *Int. Joint Conf. on Computational Sciences and Optimization*. pp.102-104. 2009.

[25] Zhendong, C. and Xicheng, W. A Grid Scheduling Algorithm Based on Resources and load adjusting. Knowledge Acquisition and Modeling Workshop. KAM Workshop 2008.

[26] Zeng, B., Wei, J. and Liu, H. Dynamic Grid Resource Scheduling Model Using Learning Agent. *IEEE Int. Conf on Networking, Architecture, and Storage*, 2009.

[27] Li, W. and Yuan, C. Research on Grid Scheduling based on Modified Genetic Algorithm. *Third Int. Conf. on Pervasive Computing and Applications*. ICPCA 2008. pp. 633-638. 2008.

[28] Wooldridge, Michael J. *An Introduction to MultiAgent Systems*. John Wiley and Sons, 2009.

[29] Deugo, D., Weiss, M. and Kendall, E. Reusable patterns for agent coordination. A. Omicini, F. Zambonelli and M. Klusch (Eds.). Coordination of Internet agents. Springer, pp. 347-368. 2001.

[30] Schelfthout, K. and otros. Agent Implementation Patterns. In Proc. Workshop on Agent-Oriented Methodologies, 17th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, 2002.

[31] Iosup, A., Li, H. and Dumitrescu, C. The Grid Workload Format. http://gwa.ewi.tudelft.nl/TheGridWorkloadFormat_v001.pdf