A Decentralized Utility-based Grid Scheduling Algorithm

João Vasques
Inesc-id/IST
Av. Prof. Doutor Aníbal Cavaco Silva
2744-016 Porto Salvo, Portugal
joao.vasques@ist.utl.pt

ABSTRACT

Grid systems have gained tremendous importance in past years since application requirements increased drastically. The heterogeneity and geographic dispersion of grid resources and applications place some difficult challenges such as job scheduling. A scheduling algorithm tries to find a resource for a job that fulfills the job's requirements while optimizing a given objective function. Utility is a measure of a user's satisfaction that can be seen as an objective function that a scheduler tries to maximize. Many utility functions have been proposed as an objective for scheduling algorithms. However, the proposed algorithms do not consider partial requirement satisfaction by awarding an utility based on the total fulfillment of the requirement. Most of them follow centralized or hierarchical approaches, suffering from scalability and fault tolerance problems. Our solution proposes a decentralized scheduling architecture with utility based scheduling algorithm that considers partial requirements satisfaction to overcome the shortcomings of actual solutions. Performance results show that user utility, submission and execution times are improved and a slightly more balanced system is achieved.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Operating Systems—*scheduling*
O; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Simulation

Keywords

grids, utility scheduling, partial utility, resource discovery

1. INTRODUCTION

The computational power required nowadays is huge. Genetic studies, large macroeconomic simulations and physicists trying to find the origin of the universe are examples of investigation areas that need access to a lot of computational power, i.e. computational resources. Due to the continuous growing need of science for computational resources it is important to have mechanisms that assure that shared resources are used in an efficient and fair way. For this reason, grid scheduling is a very important problem that has been widely studied by the computer science community. The purpose of grid scheduling is to allocate a job to a resource, fulfilling the job's requirements while optimizing an objective function.

Many solutions to the grid scheduling problem have been proposed. However, most of them use a centralized or hierarchical approach, lacking for scalability and reliability. Several decentralized approaches have also been proposed [5], but they do not consider partial requirement fulfillment, which might have a major impact on the user satisfaction. To address this shortcomings we propose a grid scheduling algorithm that considers partial requirement fulfillment based on information provided by the users. With this new scheduling approach, named as Partial Utility (PU) scheduling, we intend o maximize user's success rate, utility and system load balancing.

The rest of the paper is organized as follows: in section 2 we present the related work; the scheduler architecture is described in section 3; in 4 we present our simulation results; finally, in section 5 we present some concluding remarks and our plans for future work.

2. RELATED WORK

This section surveys the most relevant classical and utility-based scheduling algorithms.

2.1 Classic Scheduling Algorithms

In this section we present some of the classical scheduling algorithms in Grids and distributed systems.

In First Come First Served (FCFS) algorithm jobs are executed according to the arriving time order [11]. This algorithm has major disadvantage the fact that when a large job is on the waiting queue, the jobs behind it must wait a long time for the large job to finish.

Round Robin (RR) algorithm solves this issue by assigning to each job a time interval, called quantum, during which it is allowed to run. If a job cannot be completed in a quantum it will return to the queue and wait for the next round [11]. The only challenging issue with this algorithm is to find a

suitable length for the quantum.

The Minimum Execution Time (MET) algorithm assigns each task to the resource that performs it with the minimum execution time [12]. MET does not consider whether the resource is available or not at the time (ready time) and can cause severe imbalance in load across resources [7, 12, 14]. The main advantage of the algorithm is that it gives to a task the resource that performs it in the smallest amount of time.

The Minimum Completion Time (MCT) algorithm assigns a task to the resource that obtains the earliest completion time for that task [7, 12, 14]. MCT has the following disadvantage: the selected resource may not be one that has the minimum execution time for that task.

To deal with the completion time problem several algorithms divides the scheduling process in two phases. This is the case of Min-min algorithm [7], the Min-Max algorithms [8, 14] and the Max-min [7]. During the first phase they calculate the minimum completion time and in the second one they use the information from the first phase and select the best resource using different heuristics.

The sufferage of a task is the difference between its second minimum completion time and its first minimum completion time [8, 14]. These completion times are calculated considering different resources [11]. In the Sufferage algorithm the criteria to assign a task to a resource is the following: assign a resource to a task that would suffer the most if that resource was not assigned to it [11, 12]. The sufferage value of a task is the difference between its second earliest completion time and its earliest completion time

Different classical scheduling algorithms have been presented. Apart from the most simple (FCFS and RR), all the others take into account job completion time in their heuristics. However they do not consider Quality of Service (QoS) or utility demands restraining the user satisfaction regarding the way their jobs are treated.

2.2 Utility-based Scheduling Algorithms

Next, we present some QoS and utility scheduling algorithms for grid environments that aim to solve the open issues of classical approaches.

In [1] Amuda et al. propose a QoS priority-based scheduling algorithm. The algorithm assumes that all the necessary information about resources, jobs and priority values is available and is designed for batch mode independent tasks. The task partition divides the tasks into two groups (high and low) using the priority value as a QoS parameter. Tasks with higher priority are scheduled first on all the machines.

In [3] Chauhan et al. propose two algorithms for QoS-based task scheduling: QoS Guided Weighted Mean Time-min (QGWMT) and QoS Guided Weighted Mean Time Min-Min Max-Min Selective (QGWMTMMS).

The QGWMT is a modification of the Weighted Mean Time-min [9] algorithm that considers network bandwidth as a QoS parameter. First, the algorithm divides the tasks in two groups: high and low QoS. Tasks from the high QoS group are scheduled first. Next, for each task on a group, the algorithm calculates the task's weighted mean time (WMT). The task with the higher WMT is selected. Then, the algorithm choses the resource that gives the earliest completion time to the selected task and maps the task to it.

The QGWMTMMS is a modification of the Weighted Mean Time Min-Min Max-Min Selective [2] algorithm using network bandwidth as a QoS parameter. First, the algorithm creates n priority groups and assign tasks to them according based on their QoS demands. Tasks from higher priority groups are scheduled first.

In [4] Chen proposes an economic grid resource scheduling based on utility optimization that uses a universal flexible utility function that addresses QoS requirements of deadline and budget. The paper assumes that a grid is hierarchical and that the user submits the assignment to a Grid Resource Manager (GRM). The GRM is at the top of the hierarchy, on the second level there are the Domain Resource Managers (DRM) that are responsible for Computing Nodes (CN) or other DRM. The algorithm starts by the GRM getting utility information from all DRM and by calculating the rate of throughput and average response delay. Then, the algorithm finds out which of DRM has the maximum utility value (MUV) and selects it to be the scheduling node. If MUV is not unique, them the DRM which has the greatest variance is chosen. If the nodes on the next level of the chosen DRM are not CN them the process is repeated. Otherwise, the algorithm finds the node which has the maximum utility value and give it the user assignment.

In [5] Chunlin et al. propose an optimization approach for decentralized QoS-based scheduling based on utility and pricing. Grid resources can be divided into computational resources (CPU speed, memory size, storage capacity) and network resources (bandwidth, loss rate, delay and jitter). Task agents specify their resource requirements using a simple and declarative utility model. The Grid is seen as a market where task agents act as consumers and resources as providers that compete with each other to maximize their profit. Due to the fact that the its not realistic that the Grid knows all the utility functions of the task agents, although it is mathematical tractable, and it requires global coordination of all users, the authors propose a decomposition of the problem in two problems (task agent optimization and resource agent optimization) by adopting a computational economy framework. The proposed solution allows multi-dimensional QoS requirements that can be formulated as a utility function that is weighted sum of each dimension's QoS utility function.

Utility based scheduling algorithms have been proposed to solve the lack of QoS support and utility demand of the classical approaches. Although all of them use different approaches and provide support to several requirements, none of them support partial utility mechanisms. Hence, if the resource that matches the user's requirements is not available, the job will be delayed or rejected. Our proposal solve this by providing partial utility.

3. PROPOSED SOLUTION

3.1 Architectural design options

The scheduling responsibility can be delegated on one centralized scheduler or be shared by multiple distributed schedulers. In this section we present the main reasons that lead us to the use of a decentralized solution.

The centralized approach is very simple, as there is only one scheduler for the Grid. However, it has several important drawbacks that overcomes its use, such as having a single point of failure [15], lack of scalability and lack of fault-tolerance [6, 10].

Hierarchical solutions minimize the aforementioned prob-

lems by organizing the schedulers hierarchically. Keeping track of the hierarchy introduces additional complexity without completely solving the problems of centralized approach: it is more scalable and more reliable but in case of failure of a scheduler all the associated resources become unavailable.

In decentralized scheduling algorithms there is no central scheduler to control the resources [5]. Instead, there are local schedulers to which the scheduling requests are sent to. They take into consideration important issues such as fault-tolerance, scalability and multi-policy scheduling and so we used a decentralized scheduling architecture.

The grid network is divided into Virtual Organizations (VOs), as depicted in Figure 1. Each one of the VOs comprise three different types of entities: local resources that are used for job's submission; Grid Information Service (GIS) that is used to maintain the information of the VO resources and of the other existing VOs; and Local Scheduler (LS) that participates in local and remote scheduling.

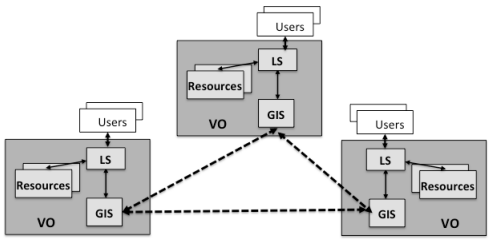

Figure 1: Grid organization

3.2 Local scheduler

This sections describes the LS architecture and its components: the Resource Manager and the Job Scheduler. It also contains a detailed description of our utility function.

3.2.1 Local scheduler architecture

Figure 2 represents the high level architecture of a LS, comprising two different entities: a Resource Manager and a Job Scheduler.

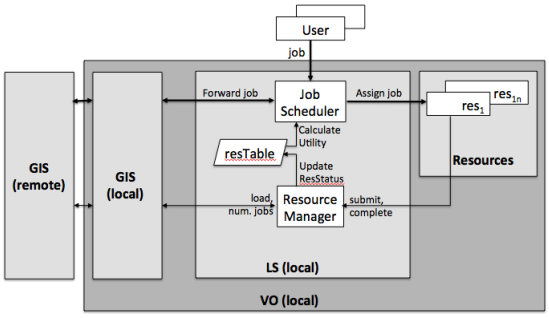

Figure 2: High Level Architecture

The Resource Manager is responsible for maintaining a global view of the network, by periodically transferring resource state with remote VOs so that each one of them maintains a snapshot of others LS resources. The Job Scheduler is responsible for processing both local and remote jobs according to the scheduling algorithm in use, either by assigning them to a local resource or by forwarding it to a remote LS.

3.2.2 Resource Manager Entity

Resources' information can be classified in two categories: static and dynamic. Static information has a low rate of change, as nodes may be added to, or removed from, clusters. Some examples of static information are: operating system, processor, number of cores, disk memory, RAM, etc. Dynamic information is likely to change over time. Some examples of dynamic information are: CPU occupation (per core), number of allocated jobs, free disk memory, free RAM, etc.

Resources send information about their current state to their LS when a new job is submitted or completed. In order to have a snapshot of the entire set of the grid resources, VOs maintain up-to-date resources status information by sending information of their resources' status to the others VOs and updating the information of the remote resources status. The exchanged information is a resume of the VO's resources state that includes average resource utilization, resource with the less CPU occupation and resource that has more memory (disk and RAM) available.

The update of the remote resource status is performed in three ways: periodically, when the updated load of the snapshot exceeds a limit or when the number of jobs that have been submitted to the resource exceeds a limit. Using these three mechanisms a more accurate view of the remote resources status is achieved although inconsistent views might still happened between two consecutive updates.

The entity responsible for performing such tasks is the Resource Manager. The pseudo-code is depicted in Algorithm 1.

Algorithm 1 Resource Manager algorithm

```
WaitFor(evt)
if evt==(Submit(job, res) || Complete( job, res)) then
   UpdateResTable (resTable[Local, res])
end if
if evt==(timeout  || load>lMax  ||  nJobs>nMax) then
   SendResStatusMsg(resTable[Local, All])
end if
if evt == RcvResStatusMsg(resTable[remoteVO, All]) then
   UpdateResTable (resTable[remoteVO, All])
end if
```

3.2.3 Job Scheduler Entity

Every incoming job is submitted to the LS either from a local or a remote user. In case of local jobs, the LS finds the resource that best matches the user requirements by calculating the utility function of all available resources. If the chosen resource is local then the job is submitted otherwise it is forwarded to the remote LS.

As the LS remote resources' status might be inconsistent and the required resource might not be really available. In case of reception of a remote job, the LS must checks if the required local resources are available before realizing the submission. If this is not the case, a new utility calculation is performed; if no local resource is found the job is forwarded again to a remote LS, unless the Time-To-Live (TTL) expires.

The entity responsible for performing such tasks is the Job Scheduler. The pseudo-code for the described procedure is presented in Algorithm 2.

Algorithm 2 Job Scheduler algorithm

```
while TRUE do
  jobProcessed=FALSE
  TTL=MAX_TTL
  WaitForJobSubmission(job)
  GetJobInfo(job,user,TTL,jobReq,reqRes)
  if user ∈ remoteVO then
    TTL − −
    if AvailableRes(reqRes) then
      AssignJob(job,reqRes)
      jobProcessed=TRUE
    end if
  end if
  if NOT jobProcessed then
    bestRes = CalcUtility(job,jobReq)
    if bestRes ∈ res[Local] then
      AssignJob(job,bestRes)
    else
      if TTL == 0 then
        SubmissionFail(user,job)
      else
        PutJobInfo(job,user,TTL,bestRes)
        ForwardJob(job)
      end if
    end if
  end if
end while
```

3.2.4 Utility function

Next, we will describe how we calculate our utility function. Unlike other schedulers that select a resource that is able to satisfy all the user requirements (Matchmaking mechanism [13]) we propose a more flexible approach on requirement fulfillment by introducing the notion of partial utility or partial requirement fulfillment.

Jobs have different requirements such as operating system (Linux, MacOs, Solaris, Windows), architecture (Intel, Motorola...), number of cores, memory (RAM and disk), as well as a maximum time for job completion. Each of those requirements has an utility value associated, ranging between [0,1]. However, unlike other solutions, the user can specify more than one option per requirement with the corresponding utility value. The different utility values that are assigned to each requirement's option defines the partial utility. The fact that each requirement can have more than one option will allow a more flexible resource selection. For example, a MacOS operating system requirement option has utility value 1 and another option (Linux) has utility value 0.5. Although Linux is not the user's preferred operating system our solution allows the user to determine the penalty, in terms of utility, whilst other solutions simply set utility to zero.

The global utility value weights the different requirements equally and the final result is a combined aggregation of the combined satisfaction of requirements. For each resource, either local or remote, its utility is calculated according to the resource status information and the job's requirements. The best resource is the one that maximizes the utility function.

Algorithm 3 depicts the pseudo-code of this procedure.

Algorithm 3 Job Scheduler Utility algorithm

```
for all v ∈ numberVO do
  for all r ∈ maxRes(resTable[v, All]) do
    globalUtil[v,r]=Utility([resTable[v,r], jobReq)
  end for
end for
return (resMatchMaxUtil(globalUtil))
```

Let us now define how the *globalUtil* of each resource is calculated. Considering:

- A list of job requirements: $jobReq = \{req_1, ...req_N\}$;
- For a given requirement, req_i, the list of possible options: $opt_i = \{opt_{(i,1)}, ..., opt_{(i,K)}\}$;
- The set of utility values of a requirement's (req_i) options (opt_i): $\alpha(i)$
- The weight $\beta_{(i)} \in [0, 1]$ assigned to each requirement req_i.

The $globalUtlity[v, r]$ of resource r belonging to cluster v is given by Eq. 1:

$$globalUtility[v, r] = \frac{\sum_{i=1}^{N} max(\alpha_{(i)} * \beta_{(i)})}{N} \quad (1)$$

If all the requirements have a similar importance β must be set to one. However, if one wants to prioritize them, a different value of β must be assigned to each one of them.

4. SIMULATION STUDIES

4.1 Performance metrics

The following metrics will be used to validate the performance of our scheduling algorithm.

- User success ratio – is the ratio between the number of successfully completed jobs and the number of submitted jobs of each user. It is calculated as follows:

$$user\_suc\_ratio_{(u)} = \frac{\sum_{j=1}^{n\_jobs_{(u)}} complete[j]}{\sum_{j=1}^{n\_jobs_{(u)}} submit[j]} \quad (2)$$

- User utility – average utility value of the resources used by the user's jobs. It is given by:

$$user\_util_{(u)} = 1 - \frac{\sqrt{\sum_{j=1}^{n\_jobs_{(u)}} utility\_value[j]}}{n\_jobs_{(u)}} \quad (3)$$

- Submission time – average time since the user's jobs are submitted till they are assigned. The User submission time is given by:

$$user\_submit\_time(u) = \frac{\sum_{j=1}^{n\_jobs_{(u)}} (t\_assign[j] - t\_submit[j])}{n\_jobs_{(u)}} \quad (4)$$

  The system submission time ($system\_submit\_time$) is given by the average value of all users.
- Execution time – average time since the user's jobs are submitted till they are completely executed. It is given by:

$$user\_exec\_time(u) = \frac{\sum_{j=1}^{n\_jobs_{(u)}} (t\_executed[j] - t\_assign[j])}{n\_jobs_{(u)}} \quad (5)$$

  The system execution time ($system\_exec\_time$) is given by the average value of all users.

- System load balancing level– The load variation of the system's resources. It is measured as follows:

$$system\_load\_balance = 1 - \frac{std\_dev\_load}{avg\_load} \quad (6)$$

Where *avg_load* represents the mean resource utilization and *std_dev_load* the respective mean square deviation.

4.2 Simulation environment

The validation of our proposal was done by simulation, using the GridSim toolkit. In order to assess its performance the following scheduling algorithms have been tested:

- Partial Utility (PU) - Our scheduling algorithm without any priority assigned to the requirements, having β set to one for each one of the requirements.
- Binary Utility (BU) - The generic scheduling mechanism that defines the utility of a resource based on a binary decision that just states wether or not a resource is able to fulfill the user requirement. In this algorithm each requirement has only one possible option.
- MM - the Matchmaking algorithm defined in [13]. In this scheduling approach, the utility of a resource is based on its ability to satisfy all the user requirements. For each resource the user has to define a single requirement.

We simulate four cluster with 20 resources each and 70 users randomly distributed among them. Each user submits 20 jobs that were generated using a Poisson distribution with mean 2. This makes a total of 1400 jobs for 80 grid resources. There are four types of requirements: operating system, architecture, job execution time and processor speed. Apart from architecture, whose options are 32 and 64 bits, all the others have a maximum of four options. Each option is selected in a random way. It is important to mention that all the options have the same probability of being selected. The utility value assigned to each option is also randomly generated. There are four utility intervals: [0;0.25[, [0.25; 0.50[, [0.50; 0.75[, [0.75,1]. First, one interval is randomly selected. After the interval is selected, a random number, bounded by the interval, is generated. This number corresponds to the utility value of the requirement's option. Resource's characteristics such as operating system, architecture, number of processing elements (cores) and their capacity are generated using the job's requirements method.

For each experiment 10 simulation runs have been executed and statistically processed with a confidence level of 90%.

4.3 Results and analysis

Figure 3 depicts the user success ratio of the different schedulers, which are identified as defined in the previous section. Apart from the MM scheduler, all the others were able to serve all the submit jobs. The particular case of the MM scheduler is caused by the rejection of jobs that do not fulfill all the user requirements. In all the other cases, a job is accepted whenever, at least, one of its requirement is satisfied.

When comparing the average user utility, depicted in figure 4, we can conclude that our scheduler (PU) outperforms

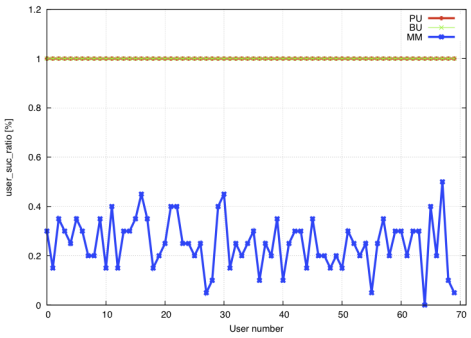

Figure 3: Users success ratio

the others, followed by the BU scheduler. The higher utility value achieved by PU schedule is caused by the partial utility police that allows for the selection of resources that partially satisfy the user requirements and the possibility of job forwarding between different clusters.

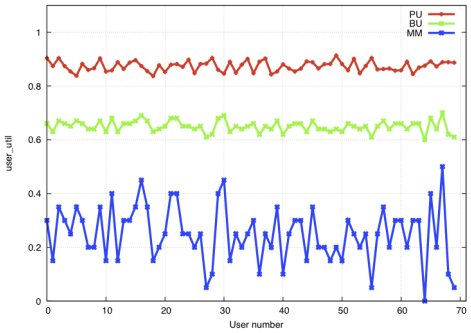

Figure 4: User utility

Figure 5 depicts the user submission time of the three scheduling algorithms and Figure 6 the execution time. Our solution has a higher submission time due to the fact that jobs may be forwarded between clusters. However, this additional delay does not decreases user's satisfaction as figure 4 presents.

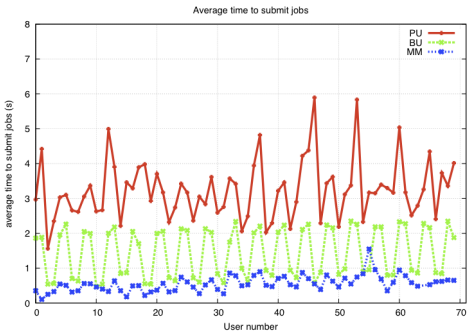

Figure 5: User submission time

Regarding the execution time, our solution performance is way better than Matchmaking and slightly better than the Binary Utility solution.

The results achieved showed that PU scheduling algorithm

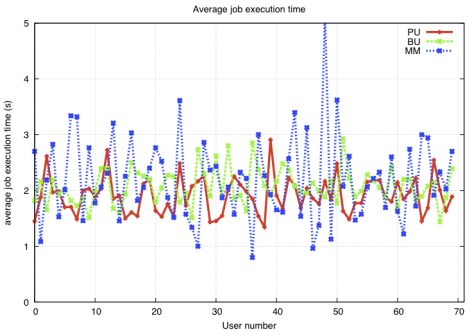

Figure 6: User execution time

outperforms all the others in terms of user's satisfaction, job's success and execution time. It is important to mention that although there was an additional delay is introduced by the mechanism of job forwarding between clusters our solution's performance was the best.

5. CONCLUSIONS

We proposed a solution to maximize user's satisfaction and system load balancing. To do so, we proposed a decentralized scheduling architecture were each VO maintain the information of resources through the use of a GIS. Our architecture also comprises a LS that is responsible for all scheduling mechanisms. We use a method to calculate utility that considers partial requirement fulfillment/ partial utility. The simulations studies performed in GridSim have shown that our solution achieves better results than the Binary utility or Matchmaking algorithms in all the metrics studied. Hence, using partial utility enables better user satisfaction, smaller job submission and execution times and a more well-balanced grid. We plan to expand our proposal to allow the user to prioritize the requirements and to test it in a real grid scenario.

6. ACKNOWLEDGMENTS

This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds.

7. REFERENCES

[1] T. Amudha and T. Dhivyaprabha. Qos priority based scheduling algorithm and proposed framework for task scheduling in a grid environment. In *Recent Trends in Information Technology (ICRTIT), 2011 International Conference on*, pages 650 –655, June 2011.

[2] S. Chauhan and R. Joshi. A weighted mean time min-min max-min selective scheduling strategy for independent tasks on grid. In *Advance Computing Conference (IACC), 2010 IEEE 2nd International*, pages 4 –9, feb. 2010.

[3] S. S. Chauhan and R. C. Joshi. Qos guided heuristic algorithms for grid task scheduling. *International Journal of Computer Applications*, 2(9):24–31, 2010.

[4] J. Chen. Economic grid resource scheduling based on utility optimization. In *Intelligent Information Technology and Security Informatics (IITSI), 2010 Third International Symposium on*, pages 522 –525, April 2010.

[5] L. Chunlin and L. Layuan. An optimization approach for decentralized qos-based scheduling based on utility and pricing in grid computing. *Concurrency Computation Practice And Experience*, 19(1):107–128, 2007.

[6] F. Dong and S. G. Akl. Scheduling Algorithms for Grid Computing : State of the Art and Open Problems. *Components*, pages 1–55, 2006.

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

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

[9] Z. Jinquan, N. Lina, and J. Changjun. A heuristic scheduling strategy for independent tasks on grid. In *High-Performance Computing in Asia-Pacific Region, 2005. Proceedings. Eighth International Conference on*, pages 6 pp. –593, July 2005.

[10] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software: Practice and Experience*, 32(2):135–164, Feb. 2002.

[11] Y.-H. Lee, S. Leu, and R.-S. Chang. Improving job scheduling algorithms in a grid environment. *Future Generation Computer Systems*, 27(8):991–998, Oct. 2011.

[12] M. Maheswaran, S. Ali, H. Siegal, D. Hensgen, and R. Freund. Dynamic matching and scheduling of a class of independent tasks onto heterogeneous computing systems. In *Heterogeneous Computing Workshop, 1999. (HCW '99) Proceedings. Eighth*, pages 30 –44, 1999.

[13] R. Raman, M. Livny, and M. Solomon. Resource management through multilateral matchmaking. *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*, pages 290–291.

[14] R. Sahu. Many-Objective Comparison of Twelve Grid Scheduling Heuristics. *International Journal*, 13(6):9–17, 2011.

[15] F. Xhafa and A. Abraham. Computational models and heuristic methods for Grid scheduling problems. *Future Generation Computer Systems*, 26(4):608–621, Apr. 2010.