HPCLatAm 2013, pp. 49-60 (full paper)
Session: Evolutionary Computation & Scheduling
C. Garcia Garino and M. Printista (Eds.)
Mendoza, Argentina, July 29-30, 2013

Heterogeneous Resource Allocation in the OurGrid Middleware: A Greedy Approach

Miguel Da Silva and Sergio Nesmachnow

Centro de Cálculo, Facultad de Ingeniería,
Universidad de la República, Montevideo, Uruguay
{mdasilva,sergion}@fing.edu.uy

Abstract. OurGrid is an open source grid middleware that enables the creation of peer-to-peer computational grids to speed up the execution of bag-of-tasks applications. This article addresses the scheduling problem arising when the participants of the grid contribute with heterogeneous resources having different computing power, by studying the application of a greedy approach for selecting and assigning resources to jobs submitted for execution in a cooperative grid. The proposed method has been incorporated to the OurGrid code. The experimental analysis performed over a set of **90** realistic problem instances following both the related and unrelated machines model demonstrates that significant execution time improvements over the standard scheduling policy are obtained: about **30-35%** overall, and **25-30%** for large grid scenarios.

1 Introduction

Grid computing is a paradigm for parallel/distributed computing that allows the integration (federation) of many computer resources from diverse locations worldwide, in order to provide a powerful integrated computing platform that allows solving applications with high computing demands. This paradigm has been increasingly employed to solve complex problems (i.e. e-Science, optimization, simulation, etc.) in the last ten years [11].

Grid infrastructures are conceived as a large loosely-coupled virtual supercomputer formed by many heterogeneous platforms of different characteristics, usually working with batch (i.e. non-interactive) workloads with a large number of files. Grid infrastructures have made it feasible to provide pervasive and cost-effective access to distributed computing resources for solving hard problems [10]. Starting from studies on small grids in the earlier 2000's, nowadays grid computing is a consolidated field of research in Computer Science and many grid infrastructures are widely available. As of 2012, more than 12 PFLOPS (i.e. 10^{15} floating point operations per second) are available in the current more powerful grid system, from the Folding@home project.

Many kinds of computer systems can be used in order to create a grid; from super-computers to low-cost personal computers. As a consequence of this feature, an important problem appears: finding an appropriate allocation of resources in a grid environment composed by the aggregation of heterogeneous resources with different computing power [9,16].

Allocation or *scheduling* is a key problem to achieve efficiency when using grid systems. The goal is to assign tasks to the computing resources by reflecting some user- and system-centric objectives and satisfying some efficiency criteria, usually related to the execution time, resource utilization, economic cost, etc.

In this work, we focus on a specific case of the independent tasks model, which is relevant for grid infrastructures: the case of Bag-of-Task (BoT) applications (jobs). In the independent tasks model, tasks within a job are assumed to have no dependencies between them, and they arrive to the scheduler from the different grid users. The scheduler collects and maps the tasks into the grid resources with a given frequency, to optimize some criteria. The BoT model typically arises in grid and volunteer-based computing infrastructures –such as OurGrid, Teragrid, Berkeley's BOINC, Xgrid, etc. [5]–, where applications using Single-Program Multiple-Data domain decomposition are very often submitted for execution to solve problems such as multimedia processing, parameter sweeps, data mining, parallel numerical models for physical phenomena, simulations, and computational biology. Thus, the relevance of the scheduling problem faced in this work is justified due to its significance in realistic grid environments.

We study the resource assignment in the OurGrid open source grid middleware, a modern tool for developing peer-to-peer grid and volunteer-computing platforms [8] (see a description of OurGrid in Section 2). The standard resource allocation policy used by OurGrid does not support resource heterogeneity. It works following a round-robin approach, which allows achieving good performance allocation results for homogeneous systems [8]. The main motivation of the research reported in our article is to enhance the OurGrid middleware with a more efficient resource policy, capable of handling heterogeneity. The proposed scheduling method is based on a greedy algorithm that searches for the most suitable computing resources for each new task available to execute.

The main contributions of the research reported in this article are: i) a new greedy approach for resource allocation in OurGrid is developed; ii) the proposed method is implemented within the OurSim simulator for OurGrid computing infrastructures, as well as all the features needed to make a site aware of the local resources available and their computing power, and iii) an experimental evaluation is performed in order to evaluate the results of the proposed scheduling method, by using realistic workloads and scenarios generated following a specific methodology. All the contributions to the OurSim code have been sent to be included in the official release of the OurGrid middleware.

The rest of the manuscript is organized as follows. Section 2 introduces the OurGrid middleware and explains the components comprising a regular OurGrid site. Section 3 accounts for the main concepts about heterogeneous computing (HC) environments and the problem scenarios used in this work. A review of relevant related works in the area is included in Section 4. The new greedy scheduling method for OurGrid is formally described in Section 5. After that, Section 6 presents the experimental analysis to validate the proposed approach, and the main results are reported and discussed. Finally, Section 7 presents the conclusions of the research and formulates the main lines for future work.

2 The OurGrid Middleware

OurGrid is an open source grid middleware based on a peer-to-peer architecture, developed by researchers of *Universidade Federal de Campina Grande* (UFCG), Brazil [8]. This middleware enables the creation of peer-to-peer computational grids, and it is intended to speed up the execution of BoT applications.

The OurGrid architecture is built by aggregating several participants in a grid environment, allowing them to use remote and local resources to run their applications. OurGrid uses the eXtensible Messaging and Presence Protocol (XMPP), an open technology for real-time communication which powers a wide range of applications, including instant messaging, presence, multi-party chat, voice and video calls, collaboration, lightweight middleware, content syndication, and generalized routing of XML data. XMPP allows federation, it is Internet-friendly, and efficient, since several services can use the same XMPP server.

2.1 OurGrid Components

The main components of the OurGrid architecture are:

- *The broker*, which implements the user interface to the grid. By using the broker, the users can submit jobs to the grid and also track their execution. All the interaction between the user and the grid infrastructure is performed through the broker.
- *The workers*, which are the component which are responsible for processing the jobs submitted to the grid. Each worker represents a real computing resource. OurGrid workers support virtualization, and so they offer an isolated platform for executing jobs comprising no risks to the local system running the component.
- *The peers*, which has a twofold role. From the point of view of the user, it is responsible to search and allocate corresponding computing resources for the execution of his jobs. From the point of view of the infrastructure (implicitly, for the administrator of the site) the peer is responsible for determining which workers can be used to execute an application, and also how they will be used. When the scheduling algorithm is executed, the resources to be assigned to execute a given application will be selected among those available in the grid. Normally, it is enough to have one peer per site. Communication between peers makes possible to execute jobs remotely; in case that the local resources are not enough for satisfying the requirements of a job, the peer seeks for additional resources available in remote sites.
- *The discovery service*, which keeps updated information about the sites comprising the grid, and it is used to find out the end points that peers should use to directly communicate with each other.

All this components are integrated in a transparent way to the user, allowing the grid to provide a single-image of an infrastructure with a large computing power. A description of the Ourgrid architecture is shown for a sample grid environment in Figure 1.

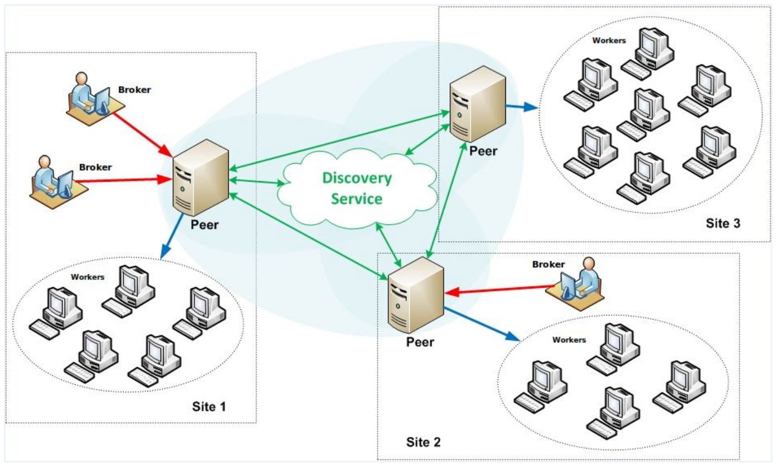

Fig. 1. Ourgrid architecture example

2.2 OurSim and Synthetic Applications

OurSim [6] is a discrete-events simulator that allows creating a virtual grid infrastructure, including all the components of a typical OurGrid installation:

To use OurSim, a *simulation*, i.e. a set of files describing the site and a set of jobs to be run is mandatory. The job files can be generated automatically by OurSim. Jobs generated using this procedure are called *synthetic applications*, Three essential features have been chosen for the tests proposed in this work: (a) number of sites (peers), (b) number of users (workers) per site and (c) whole duration (period) for the set of jobs generated. Once the files describing the grid and the set of applications are available, a simulation can be executed.

The experimental evaluation of the new scheduling policy proposed in this article is performed using OurSim and realistic grid workload and scenarios. The OurSim simulations are used to avoid deploying many Ourgrid sites for the experiments. Since OurSim implements exactly the same scheduling functions than the Ourgrid middleware, all results are valid also for the Ourgrid middleware.

3 Heterogeneous Grid Computing

Nowadays, building distributed computing platforms by gathering heterogeneous resources is very common. When using this kind of collaborative grid infrastructures, the computing power of the grid workers may vary significantly.

Our proposal in this article is to account for heterogeneity in collaborative grids using the OurGrid middleware. The greedy scheduling approach is tested using simulated OurGrid infrastructures which considers realistic computing element with diverse computing power. The execution time results of the greedy resource allocation policy are then compared to the ones obtained when the standard allocation resource method is used by OurGrid.

We use as a reference baseline the heterogeneity model for the Heterogeneous Computing Scheduling Problem (HCSP) [14,15] and the methodology for creating HCSP instances described in [13]. To model grid heterogeneity, two relevant features are taken into account for machines: the number of cores (#C) and the number of operations from the SSJ SPEC benchmark results (SSJ_ops) [19]. SSJ_ops is the average value of operations per second reported when solving the standard benchmark SSJ, and it is used in order to take into account a realistic value for the computing power of a given computer. We avoided using the maximum theoretical GFLOPS reported for each vendor, since this is a peak value that usually does not reflect the real computing capacities of the machine. The data used to generate the grid scenarios are presented in Table 1.

In order to define a specific methodology to select the computing elements within a specific grid scenario, we model the resource heterogeneity regarding the standard deviation in the SSJ operations per core (σ_C). According to σ_C, three heterogeneity categories for scenarios are defined: *low* ($\sigma_C < 36\%$), *medium* ($36 < \sigma_C < 44\%$), and *high* ($\sigma_C > 44\%$). For each experiment performed using the OurSim simulator, each site can be classified regarding these categories.

Two models widely used in literature to represent HC scenarios are considered: the *related machines* model [2] and the *unrelated machines* model [12]. For the related machines model, each task t_i demands a fixed number of operations $TO(t_i)$, and in a grid scenario the processing time of task i in machine j is $ET(t_i, m_j) = TO(t_i)/SSJ\_ops(m_j)$. Unrelated machines scenarios are built by defining different heterogeneity levels for each site. Using OurSim it is easy to define the heterogeneity level for a site: when creating the virtual grid, the characteristics for each component can be configured by selecting the appropriate computing resources to match the corresponding intra-site heterogeneity level.

Table 1. Details of the processors used in the experimental evaluation

#	**processor**	**#C**	**SSJ_ops**	#	**processor**	**#C**	**SSJ_ops**	#	**processor**	**#C**	**SSJ_ops**
1	AMD Opteron 6238	48	1898162	25	Intel Xeon E5-2670	16	1394534	49	Intel Xeon E5-2660	16	1204440
2	AMD Opteron 6238	24	987946	26	Intel Xeon E5-2450L	16	1006482	50	Intel Xeon E5-2470	16	1496064
3	Intel Xeon E5-2670	16	1236124	27	Intel Xeon E5-2470	16	1328249	51	Intel Xeon E5-2470	16	1511097
4	Intel Xeon E5-2660	16	1280523	28	Intel Xeon E5-2450	16	1254329	52	Intel Xeon E5-2640	12	1028960
5	Intel Xeon E5-2650L	16	959239	29	Intel Xeon E5-2440	12	986986	53	Intel Xeon E5-2630	12	961609
6	Intel Xeon E5-2670	16	1336479	30	Intel Xeon E5-4640	32	2395181	54	Intel Xeon E5-2665	16	1365171
7	Intel Xeon E5-2650L	16	960074	31	Intel Xeon E5-4640	32	2402711	55	Intel Xeon E5-2640	12	990555
8	Intel Xeon E5-2680	16	1426130	32	Intel Xeon E5-2640	12	1034209	56	Intel Xeon E5-2665	16	1347230
9	AMD Opteron 6276	64	2220373	33	Intel Xeon E5-2665	16	1337393	57	Intel Xeon E5-2660	16	1446931
10	AMD Opteron 6278	64	2296333	34	AMD Opteron 6278	32	1233423	58	Intel Xeon E5-2660	16	1447399
11	Intel Xeon E5-2660	16	1304143	35	Intel Xeon E5-2640	12	1007444	59	Intel Xeon E5-4650L	32	2790966
12	Intel Xeon E5-2660	16	1308474	36	Intel Xeon E5-2665	16	1347230	60	Intel Xeon E5-2660	16	1432829
13	Intel Xeon E5-2660	16	1316689	37	AMD Opteron 6278	64	2331362	61	AMD Opteron 6380	32	1660274
14	Intel Xeon E5-4650L	32	2540179	38	Intel Xeon E5-4640	32	2347591	62	Intel Xeon E5-2660	16	1450305
15	Intel Xeon E5-2470	16	1187945	39	Intel Xeon E5-2470	16	1367462	63	Intel Xeon E5-2660	16	1459934
16	Intel Xeon E5-2660	16	1290357	40	Intel Xeon E3-1240V	24	467481	64	Intel Xeon E5-2470	16	1482687
17	Intel Xeon E5-2660	16	1338554	41	Intel Xeon E5-2470	16	1516025	65	Intel Xeon E5-2470	16	1463219
18	Intel Xeon E3-1265LV	24	416999	42	Intel Xeon E5-4640	32	2619038	66	Intel Xeon E5-2470	16	1288201
19	Intel Xeon E3-1265LV	24	420255	43	Intel Xeon E5-2470	16	1329468	67	Intel Xeon E5-2670	16	1448093
20	Intel Xeon E5-2660	16	1309437	44	Intel Xeon E5-2670	16	1401473	68	AMD Opteron 4376 HE	16	843336
21	Intel Xeon E5-2660	16	1306867	45	Intel Xeon E5-2470	16	1327737	69	AMD Opteron 4386	16	971064
22	Intel Xeon E5-2660	16	1246966	46	Intel Xeon E5-2470	8	670220	70	AMD Opteron 4310 EE	8	373385
23	Intel Xeon E5-2660	16	1297229	47	Intel Xeon E5-2470	8	671815				
24	Intel Xeon E5-2650L	16	1007084	48	Intel Xeon E5-2470	16	1384613				

4 Related Work

Several works have addressed different variants of the grid scheduling problem using greedy algorithms in the last years.

Beaumont et al. [4] tackled the problem of assigning a set of clients with demands to a set of servers with capacities and degree constraints. Two versions of the problem are considered: *static* (clients are known beforehand), and *dynamic* (clients can join or leave the grid it at any time). Several methods are introduced: SEQ, OSEQ, and the three greedy schedulers: LCLS, LCBC, and OBC. The evaluation showed that *SEQ* had the better performance for the static version, and for the dynamic case OSEQ outperformed both LCBC and LCLS.

Singh and Kant [18] addressed the problem of efficient use of resources and minimization of turnaround time using a greedy method that chooses resources on the basis of computing power. The authors deal with the approach where jobs are submitted on resources where the data must be transferred to. The experimental analysis compares the greedy algorithm against a random selection of resources and a sequential assignment method, using the GrimSim simulator. According to the reported results, the sequential assignment reduces the average turnaround time by 53% over the random selection, and the greedy method reduces the average turnaround time by 14% over the sequential assignment.

Buss et al. [7] proposed three greedy methods to allocate and exchange grid services for efficiently trading markets (defined as the space where resource owners and resources consumers can agree the allocation of the resources based on consumers needs) and resource allocation. The experimental evaluation performed using stochastic simulation for a set of 50 instances with both numbers of owners and consumers chosen between 20 and 120 allowed to conclude that the greedy solutions presented are less expensive in terms of computational effort.

Zhu et al. [20] defined the Greedy-Search Based Service Location Problem and provided a theoretical model to analyze the influence of the network topology. A greedy method is introduced for service location and analysis, and the authors show that under reasonable conditions, it can achieve good result even in a practical large community, specially when comparing the QoS and fault tolerance against DHT Gnutella-like, and Napster-like systems. The experimental analysis shows that if the node degree is about 10∼20, under some assumptions a node with relative high QoS level is to be find in short hops.

Besides the previous publications, which tackle specific features of realistic grid infrastructures, many articles have proposed heuristic and metaheuristics methods for solving the grid scheduling problem from the point of view of optimization (see a review on our articles tackling the HCSP [14, 15]). However, these works often do not address for realistic features of grid systems and/or they do not use any grid or simulated grid in their experimental evaluation.

In addition, the review of the related work allows to conclude that there are few articles dealing with real grid systems and no implementations over current grid middlewares. Our work here provides a real contribution in this topic, as the proposed method is implemented in a real middleware and the experimental analysis is performed on realistic grid instances.

5 The Greedy Scheduling Approach for OurGrid

This section describes both the traditional and the new greedy approach proposed for scheduling in heterogeneous grid infrastructures using OurGrid.

5.1 Traditional Resource Allocation in OurGrid

OurGrid uses a traditional round-robin-like allocation policy to assign computing resources to newly arrived tasks. Each time a user submit a job, he will have to access the grid infrastructure through a broker. The user must describe the set of tasks in the submitted job by using a *job description file* (JDF).

Once a job is submitted, the broker contacts the local peer to ask for a set of resources that satisfy the user requirements for executing the job. The peer uses the JDF created by the user to search for an appropriate set of workers to fulfill the request. The peer first searches in the local site and, if no local worker fulfills the computing requirements, the peer will contact another peer (if any) to ask for resources. If no remote sites can satisfy the request, the job is scheduled such that its tasks are inserted in the local resources execution queue.

The scheduling policy in Ourgrid is "first coming task to first discovered resource". Implicitly, the computing resources within each site are ordered according to the time the peer registers in the grid. This ordering is then used to search for appropriate workers each time that a new request for task execution is received. By looking sequentially into the list of workers, the first one found that fulfills the task requirements is selected to be allocated to a given task.

The resource allocation policy in OurGrid uses the *Network of Favours* [3], which encourages the resource contribution to the network. The reputation of a given peer increases when it collaborates to execute a task from another peer. In that case, the peer that requested the execution keeps a local record that it owes a *favour* to the former one. Favours are taken into account when a peer has idle resources that are requested to be used by other peers. The favour count is used to prioritize the peers requests, and workers will be assigned to tasks of the peer that has contributed the most with the local peer.

Different mechanisms are used by others grid middlewares to prioritize nodes requesting resources. For example, the Condor middleware implements a C-like language called *control expressions* that allows a user to set the constraints and preferences of his jobs. Therefore, a user can chose machines according to its computing power; Condor middleware is responsible to rank the machines and assign the most suitable one to the each user job.

5.2 Changing the Way OurGrid Allocates Resources

Using the traditional Ourgrid scheduler, once a suitable resource is found for a given task, peers do not check any further. The greedy allocation policy provides OurGrid with a more efficient method for scheduling. By taking into account the computing capability of workers, the greedy scheduler allows executing incoming tasks faster, thus resulting in a better response time for the users.

The processing power was chosen as the main feature to rank the workers, because most grid and volunteer computing systems are mainly used to execute computing-intensive tasks (simulation, optimization, data mining) [10], and users are always interested in executing their tasks in the shortest time possible.

The proposed greedy scheduler can be classified into the category of *dynamic priority scheduling* algorithms [17]. This class comprises many deterministic scheduling methods that work by assigning priorities on-line, based on a particular criterion, during the execution of the job. In our case, the criterion used to determine the priorities is the expected finishing time of a given task, which is inversely proportional to the computing power of each worker.

The greedy approach proposed for task allocation (see Algorithm 1) searches for available resources and creates a list with the resources found in descending order according to the computing power. Once the list of available resources is created, the scheduler will assign as many resources as possible to the brokers with pending tasks to be processed (tasks are ordered according to the arrival time of the job they belong to). The assignment stops if no more resources are available or the requirements are fulfilled. The scheduling algorithm is executed each time that a new task arrives to the system.

When using the greedy allocation resource policy, a site providing workers with higher computing power has more chances to get the resources assigned if the request is done to a site it have given favours before.

```
declare $response$, $posWksToAllocateOrd$: list of objects of type $Allocation$;
declare $tmpWorker$, $allocWorker$: type $Worker$;
declare $allocTmp$: type $Allocation$;
$\{response, posWksToAllocateOrd\} \leftarrow$ empty list;
$\{tmpWorker, allocWorker, allocTmp\} \leftarrow$ null;
forall the $W$ in $possibleWorkersToAllocate$ do
    if $posWksToAllocateOrd.Empty()$ then
        add $W$ to $posWksToAllocateOrd$;
    else
        $allocWorker \leftarrow W$.load_data();
        boolean $ordering \leftarrow true$;
        int $indexOrd \leftarrow 0$;
        while $posWksToAllocateOrd.notEmpty() \wedge ordering$ do
            $allocTmp \leftarrow$ posWksToAllocateOrd.next();
            $tmpWorker \leftarrow allocTmp$.load_data();
            if $allocWorker.cpuPower() \geq tmpWorker.cpuPower()$ then
                $ordering \leftarrow false$;
            else
                $indexOrd \leftarrow indexOrd + 1$;
            end
        end
        add $alloc$ to posWksToAllocateOrd in position $indexOrd$;
    end
end
while $i < allocationsLeft \wedge posWksToAllocateOrd.notEmpty()$ do
    add $posWksToAllocateOrd.next()$ to $response$;
end
return $response$
```

Algorithm 1: Greedy scheduling algorithm in OurGrid

Both OurGrid and OurSim are implemented using Java. The greedy method was implemented over the OurSim code. When a job is submitted, the peer execute a sequence of Java methods in order to select workers. At the end of this sequence, a method called *takeNeededWorkers* is executed and a resource allocation policy is applied. This method belongs to the Java class *AllocationHelper*.

To implement the greedy resource allocation policy, the method *takeNeededWorkers* was completely rewriten. Its signature had to be changed in such a way that when it is invoked, OurSim can select the workers based on its computing power. Other Java classes which invoke *takeNeededWorkers* were also changed (*SamePriorityAllocationHelper* and *LowerPriorityAllocationHelper*).

6 Experimental Analysis

This section presents the experimental evaluation of the greedy scheduler, by comparing its results with against those computed using the regular Ourgrid policy over a wide range of problem instances.

6.1 Grid Scenarios

The greedy approach was tested using OurGrid simulated infrastructures, considering workers with different computing power. The analysis was carried out on unrelated machines scenarios (with low/medium/high heterogeneity) and dimension 1, 10, and 100 sites, and using the unrelated machine model with sites having different heterogeneity levels, and dimension 10 and 100 sites. A total number of 30 instances per scenario were used. The number of workers per site is 8, 16, 32 or 64, chosen using an uniform distribution, and computing power chosen according to Table 1, such that the three heterogeneity levels are satisfied.

Workloads are based on synthetic applications to represent jobs submitted to the grid. Three main features were configured to stress the infrastructure, thus few jobs would be treated in the same way by both scheduling algorithms: (a) the number of peers, (b) the number of brokers and (c) the duration of the simulation (3, 6, or 12 months, used to set the number of jobs). All problem instances were generated following the related and unrelated machine models and the Expected Time to Compute (ETC) estimation model by Ali et al. [1], following the methodology described in our previous work [13]. The problem instances are available to download at `http://www.fing.edu.uy/inco/grupos/cecal/hpc/GSOS`.

6.2 Numerical Results

Execution time. Table 2 summarizes the execution time results, reporting the average improvements of the greedy scheduler for each model/heterogeneity level.

The results in Table 2 demonstrate that the greedy scheduler outperformed the traditional one for all grid instances and workloads. Although in a few specific cases the standard policy obtained shorter (less than 2.5%) times, in average, the greedy approach significantly outperformed the traditional method.

Table 2. Improvements of the grid scheduler over the traditional scheduler

		# peers	1			10			100		
data		*months*	3	6	12	3	6	12	3	6	12
		avg. # jobs	1995	5323	9485	24310	44778	89281	231089	439339	904624
		avg. #workers	27.20	45.35	18.40	23.33	25.60	18.12	29.77	33.17	30.70
improv. (%)	*related*	*low*	25.65	21.73	25.23	16.16	18.14	16.36	24.41	26.50	29.65
		medium	13.08	15.19	14.50	49.33	56.22	47.01	23.71	25.38	27.14
		high	117.65	84.63	105.57	35.58	35.18	38.23	27.11	24.02	24.49
	unrelated		—	—	—	23.89	24.41	23.84	24.76	25.47	28.68

The traditional scheduling mechanism do not take advantage of the computing capabilities of the workers, and those resources better suited to execute a given task may be idle instead of processing. On the other hand, the greedy approach focuses on keeping these resources busy as much as possible by trying to select them each time the scheduling algorithm is executed.

As the number of the peers increases, the execution time improvement stabilizes near **25-30%**, mainly due to the necessary network communications between peers to transfer data to remote workers. The results show that even in such situation, the greedy scheduling algorithm is a more effective method for resource allocation once the data is available at the remote peer.

The high improvements for high heterogeneity scenarios with only one peer are due to long tasks, assigned by the traditional scheduling mechanism to a worker with small computing power. As there are no more peers to look for idle resources, tasks have to wait in the workers queues. Meanwhile, the greedy scheduler rationally uses those workers with high computing capabilities.

Figure 2 graphically summarizes the average improvements results obtained by the grid scheduler for each tackled problem model and dimension.

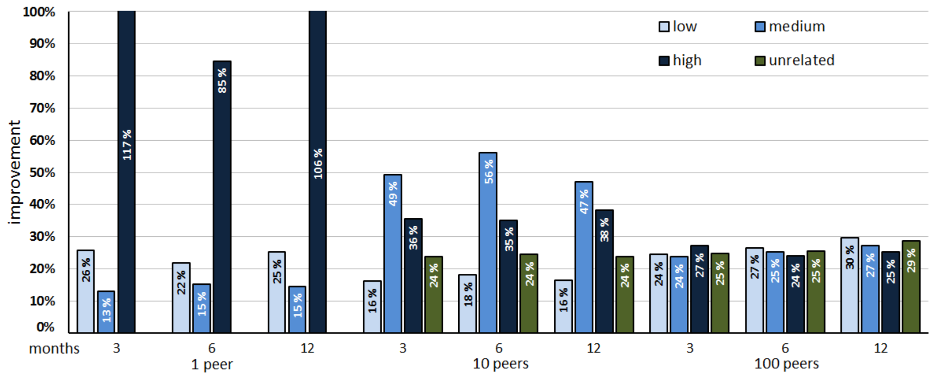

Fig. 2. Average improvements of the greedy scheduler over the traditional policy

Load balancing. We also studied the load balancing for both scheduling methods. Figure 3 reports the loads for a representative execution of both scheduling algorithms, showing the improvement of the greedy scheduler (normalized load st.dev.=26.9%) when compared with the traditional scheduler (normalized load st.dev.=71.2%). The greedy scheduler applies a kind of Optimistic Load Balancing strategy [2] that allows a better load distribution over the available workers.

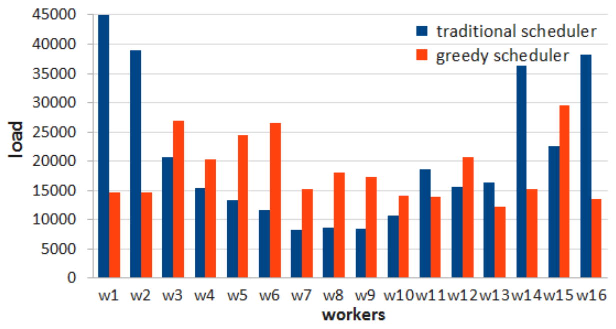

Fig. 3. Load balancing study for a representative execution

7 Conclusions and Future Work

This article presented an experimental study of applying a greedy resource allocation policy in the OurGrid middleware. After reviewing OurGrid and related works on the topic, a specific greedy scheduler that takes into account the computing power of workers was designed and implemented into OurGrid/OurSim.

The experimental analysis of the proposed method was carried out using OurSim and grid instances of different dimensions, and following both unrelated and related machines model. The experimental results demonstrate that the greedy scheduler is an effective method for reducing the overall execution time of BoT jobs. In average, the execution time improvements of the greedy scheduler over the traditional policy in OurGrid is about **30-35%**, and it tends to stabilize about **25-30%** for large scenarios and simulation periods. Speedup and heterogeneity level are not related, but the number of peers do impact in the execution time improvements. Nevertheless, the greedy scheduling allocation policy select more suitable workers than the standard policy in most cases.

The proposed greedy scheduling method is now available within the official distribution of the OurGrid/OurSim code.

The main lines for future work include improving the methods for selecting remote workers and studying new scheduling policies, by taking into account other resource selection criteria (i.e. node reliability and energy consumption). Additional scalability studies regarding the numbers of brokers per site for both *static* (known number of brokers) and *dynamic* (brokers can register and unregister from a peer) scenarios should be performed.

References

1. S. Ali, H. J. Siegel, M. Maheswaran, S. Ali, and D. Hensgen. Task execution time modeling for heterogeneous computing systems. In *Proc. of the 9th Heterogeneous Computing Workshop*, page 185, Washington DC, USA, 2000.
2. P. Ambrosio and V. Auletta. Deterministic monotone algorithms for scheduling on related machines. *Theoretical Computer Science*, 406:173–186, 2008.
3. N. Andrade, F. Brasileiro, W. Cirne, and M. Mowbray. Automatic grid assembly by promoting collaboration in peer-to-peer grids. *Journal of Parallel and Distributed Computing*, 67(8):957–966, August 2007.
4. O. Beaumont, L. Eyraud-Dubois, C. Caro, and H. Rejeb. Heterogeneous resource allocation under degree constraints. *IEEE Transactions on Parallel and Distributed Systems*, 24(5):926–937, 2013.
5. F. Berman, G. Fox, and A. Hey. *Grid Computing: Making the Global Infrastructure a Reality.* Wiley, New York, NY, USA, 2003.
6. F .Brasileiro and M.Carvalho. A user-based model of grid computing workloads. In *ACM/IEEE 13th Int. Conf. on Grid Computing*, pages 40–48, 2012.
7. G. Buss, K. Lee, and D. Veit. Scalable grid resource trading with greedy heuristics. In *Int. Conf. on Complex, Intelligent and Software Intensive Systems*, pages 427–432, 2010.
8. W. Cirne, F. Brasileiro, N. Andrade, L. Costa, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4:225–246, 2006.
9. H. Dail, O. Sievert, F. Berman, H. Casanova, A. YarKhan, S. Vadhiyar, J. Dongarra, C. Liu, L. Yang, D. Angulo, and I. Foster. *Grid resource management*, chapter Scheduling in the Grid application development software project, pages 73–98. Kluwer Academic Publishers, 2004.
10. I. Foster and C. Kesselman. *The Grid 2: Blueprint for a Future Computing Infrastructure.* Morgan Kaufmann Publishers, 2003.
11. J. Joseph and C. Fellenstein. *Grid Computing.* Pearson Education, 2004.
12. J. Lenstra, D. Shmoys, and É. Tardos. Approximation algorithms for scheduling unrelated parallel machines. *Mathematical Programming*, 46(3):259–271, 1990.
13. S. Nesmachnow, B.Dorronsoro, J. Pecero, and P.Bouvry. Energy-aware scheduling on multicore heterogeneous grid computing systems. *Journal of Grid Computing*, 2013. Online first, May 2013. DOI: 10.1007/s10723-013-9258-3.
14. S. Nesmachnow, H. Cancela, and E. Alba. Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, 15(4):685–701, 2010.
15. S. Nesmachnow, H. Cancela, and E. Alba. A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing*, 12(2):626–639, 2012.
16. A. Pugliese, D. Talia, and R. Yahyapour. Modeling and supporting grid scheduling. *Journal of Grid Computing*, 6:195–213, 2008.
17. K. Ramamritham and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proc. of the IEEE*, 82(1):55–67, 1994.
18. S. Singh and K. Kant. Greedy grid scheduling algorithm in dynamic job submission environment. In *Int. Conf. on Emerging Trends in Electrical and Computer Technology*, pages 933–936, 2011.
19. SPEC. Standard performance evaluation corporation, SPECpower_ssj2008, 2011. Available online at http://www.spec.org/power_ssj2008. Accessed May 2013.
20. C. Zhu, Z. Liu, W. Zhang, W. Xiao, and D. Yang. Analysis on greedy-search based service location in P2P service grid. In 3rd *Int. Conf. on Peer-to-Peer Computing*, pages 110–117, 2003.