Scheduling Strategies for Cycle Scavenging in Multicluster Grid Systems

Ozan Sonmez Bart Grundeken Hashim Mohamed Alexandru Iosup
Dick Epema
Delft University of Technology, NL
{O.O.Sonmez, H.H.Mohamed, D.H.J.Epema}@tudelft.nl, {Bart.Grundeken, A.Iosup}@gmail.com

Abstract

The use of today's multicluster grids exhibits periods of submission bursts with periods of normal use and even of idleness. To avoid resource contention, many users employ observational scheduling, that is, they postpone the submission of relatively low-priority jobs until a cluster becomes (largely) idle. However, observational scheduling leads to resource contention when several such users crowd the same idle cluster. Moreover, this job execution model either delays the execution of more important jobs, or requires extensive administrative support for job and user priorities. Instead, in this work we investigate the use of cycle scavenging to run jobs on grid resources politely yet efficiently, and with an acceptable administrative cost. We design a two-level cycle scavenging scheduling architecture that runs unobtrusively alongside regular grid scheduling. We equip this scheduler with two novel cycle scavenging scheduling policies that enforce fair resource sharing among competing cycle scavenging users. We show through experiments with real and synthetic applications in a real multi-cluster grid that the proposed architecture can execute jobs politely yet efficiently.

1. Introduction

Cycle scavenging is the underlying technology of desktop grids and volunteer computing projects (such as Seti@Home [8]), which enables harnessing idle CPU cycles of desktop workstations to solve large-scale scientific problems in a variety of research areas. The same concept can be applied to multicluster grid environments to give users the opportunity of executing such large-scale computation-intensive applications at a low priority without being in the way of regular grid users or local users. In this paper, we present scheduling strategies for the support of cycle scavenging in multicluster grid systems, and evaluate the performance of the implemented solutions both from the user and the system perspectives.

Today, many grids exhibit significant job submission bursts between periods of relative idleness [18]. Many users perform observational scheduling, that is, they postpone the submission of relatively low-priority jobs until a cluster becomes (largely) idle. This attitude, however, may lead to resource contention when several such users crowd the same idle cluster, and may delay the execution of more important jobs unless some form of administrative support for job and user priorities is deployed. Cycle scavenging, on the other hand, obviates the need for establishing priority classes, which can be a time-consuming and an error-prone administrative operation. Supporting cycle scavenging in a grid system would enable users to execute long-running applications (e.g., 3-D rendering, molecular docking, and game solving applications) at the lowest priority without violating the resource usage rules enforced in the system.

In this paper, we extend our multicluster grid scheduler, KOALA [26], with cycle scavenging support. The implemented cycle scavenging mechanism runs alongside the regular grid scheduling, being unobtrusive to the jobs of higher priority (both local and grid jobs). Although cycle scavenging infrastructures do exist [11, 15, 22], our mechanism obviates the need for additional software installations on the compute nodes or any modifications to the resource managers of the clusters (e.g., SGE [9]), both of which would be administrative obstacles in multicluster grid systems. We exclusively target large-scale applications that can be modeled as Parameter Sweep Applications (PSAs) to run as cycle scavenging (CS) jobs. We enable PSAs to run across multiple clusters simultaneously, that is, in a co-allocated fashion.

The scheduling architecture for cycle scavenging that we have incorporated into KOALA comprises two levels. At the grid level, scheduling policies run to ensure fair distribution of idle resources among CS users in a dynamic fashion, and at the application level, CS users can customize the scheduling policies in order to improve the performance of their applications. We have designed two best-effort cycle scavenging scheduling policies that enforce fair resource sharing between CS users in a dynamic fashion. The policies do not need to keep track of the past usages of the CS users. The first policy distributes or reclaims the idle nodes evenly

among CS users, regardless of the site these idle nodes belong to. The second policy, on the other hand, partitions or reclaims the idle nodes evenly such that each CS user is assigned an equal share of idle nodes on each site. We show with experiments conducted in a real multicluster environment that the latter policy outperforms the former in terms of fairness. We also perform experiments to demonstrate the efficiency of the implemented system in terms of scheduling overhead.

In summary, the contributions of this paper are:

- The integrated design of cycle scavenging support into a scheduling architecture for multicluster grids.
- Two best-effort cycle scavenging policies that try to achieve fair-share resource allocation among CS users, and their analysis.

2. Problem Statement

The problem we address in this paper is the efficient allocation of idle resources that may be spread across multiple grid sites to cycle scavenging (CS) applications. In this section, we state the requirements of cycle scavenging that should be taken into account by a grid resource manager or a grid scheduler to ensure the efficient allocation of idle resources. These requirements are a proper notion of idleness of processors, unobtrusiveness, and fair-share scheduling, respectively.

2.1 Notion of Idleness

A grid scheduler would possibly schedule CS jobs whenever it is aware of idle nodes on clusters. However, the existence of idle resources may not be the only prerequisite for acquiring those resources to run jobs of a CS application. When to consider a resource as idle may be subject to the additional administrative policies of each site. For instance, a site may only be willing to run such jobs when a certain percentage of its resources are idle and simultaneously there are no local jobs waiting in the queue, or site administrators may want to set usage limits due to reasons such as cooling costs, which would increase by running long-lasting CS jobs at a high utilization.

2.2 Unobtrusiveness

From the system perspective, we need to ensure that placing and running CS jobs are unobtrusive to the jobs of higher priority; a grid scheduler has to make immediate preemption possible without causing significant delays whenever non-CS jobs demand the nodes allocated to CS jobs. These high priority jobs in a multicluster grid can be defined as the non-CS jobs that are directly submitted to local resource managers by local users, and the non-CS jobs that are submitted to a grid scheduler by grid users.

When a CS job is canceled due to a high priority job by the grid scheduler, it has to return back to a job pool so that it can be re-scheduled. A checkpointing mechanism would be useful not to lose many computations; however, we do not consider checkpointing as it lies outside the focus of this paper. Instead, we leave users to implement their own application level checkpointing solutions.

2.3 Fair-Share Scheduling

As we know from desktop grids or volunteer computing environments, CS applications are high throughput computing applications that consist of independent sequential jobs, which scale to many thousands of nodes, and require large amounts of computation. Considering the size of these applications, the scheduling mechanism should try to achieve fair-share resource allocation among the users submitting CS jobs, so that it prevents some users monopolizing all free resources for a considerable amount of time, leaving space for users having relatively light workloads.

Fair-Share scheduling, in fact, was originally proposed for managing resource allocation of processes on time-sharing uniprocessor systems [20]. The application of the fair-share resource allocation to a distributed system is very much dependent on how the system administrators define the fair share. A study investigating this issue in cluster systems [21] shows that unless the jobs on a cluster are flexible in terms of space (number of nodes) and time (checkpointable), fair-share is not able to achieve real-time fairness as it can on a uniprocessor; rather, it becomes a best-effort service.

Due to the fact that jobs of a CS application may be preempted at any time, giving a hard completion time guarantee for a CS application is almost impossible; nevertheless, users submitting CS applications may be more interested in the rate of receiving partial results, that is, the throughput, since their jobs consist of independent sequential tasks. To improve such performance, scheduling at the application level should be considered as a separate layer under the fair-share resource allocation scheme. Such a layered architecture would provide modularity and flexibility.

3. Related Work

There are various platforms that make desktop grids or volunteer computing possible as well as some that enable cycle scavenging at the organization level to put idle cycles to good use.

The BOINC [11] platform facilitates volunteer computing projects (e.g., Folding@home [3], Rosetta@home [7],

and Seti@home [8]). BOINC provides tools that allow participants to remotely install a client software on a large numbers of desktops, and to attach the client software to accounts on multiple projects. With BOINC, desktop owners are able to specify how their resources can be allocated among the projects. Another similar platform is Entropia [14], which can be distinguished from its counterparts by its binary sandboxing technology for ensuring security and unobtrusiveness, and its architecture which incorporates physical node management, resource scheduling, and job management layers.

OurGrid [15] is an open platform that enables different research labs to share their idle computational resources when needed. OurGrid relies on a peer-to-peer incentive mechanism, called *Network of Favors*, which aims to make it in each participant's best interest to donate idle cycles, along with preventing free riding. With OurGrid, each user runs a broker-agent which competes with other agents to schedule the jobs over the resources on behalf of the user. Therefore, it does not provide fair-share resource allocation among users. The Condor [22] platform was initially designed to scavenge compute cycles on large collections of idle desktop machines, but it has also been extended to operate as a batch scheduler on top of a cluster system and as a resource broker on top of Globus [5] based grids [17]. In addition to desktop computers, it is also possible with Condor to scavenge idle nodes in a cluster by configuring each node such that they can execute Condor jobs when no job is running which has been submitted by the local resource manager of the cluster. Condor does not adopt traditional scheduling, rather it uses a central matchmaking mechanism in which jobs and resources are matched according to their requirements specified with so-called ClassAds. As the fair-share policy, Condor runs the *Up-Down* [27] algorithm to protect the rights of light users when a few heavy users try to monopolize all resources. The algorithm relies on the information of past resource usage rates of users.

In this paper, we do not consider past usage, since our notion of fair-share partitions the idle resources evenly among users dynamically in real time. In addition, to deploy any of the platforms mentioned above in a multicluster grid system requires per-node installations or configurations in the local resource managers of clusters, which could be impossible due to administrative restrictions. In contrast, the system we present in this paper does not require any such installations or modifications, rather, it seamlessly integrates the notion of cycle scavenging into grid-level scheduling.

4. The KOALA Grid Scheduler

In this section, we summarize our previous work on the KOALA grid scheduler [26], which is the basis of the work presented in this paper.

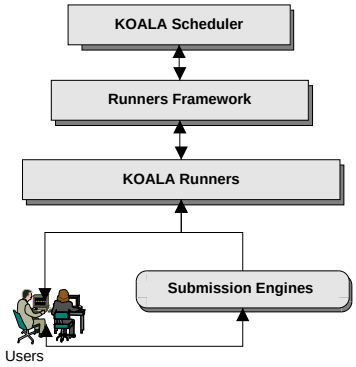

Figure 1. The KOALA layered architecture.

4.1 Architecture

The KOALA grid scheduler has been designed for multicluster systems such as the DAS [10] (see Section 6.1), with the aim of implementing mechanisms and policies for scheduling various grid application types. As distinctive features, KOALA includes support for data and processor coallocation [25] and malleability management [12] for parallel applications.

As shown in Figure 1, KOALA has a layered architecture that allows us to develop distinct layers independently, which can then work together. The *scheduler* is responsible for scheduling jobs received from the *runners* with its placement policies [24] [29] that are used to place jobs on suitable execution sites. The scheduler is supported by an information service, which monitors the status of resources by processor and network information providers and a replica location service. Providers connect KOALA with the multicluster monitoring infrastructure. The *runners framework* hides the heterogeneity of the grid by providing to the runners a runtime system and its corresponding set of APIs for commonly used job submission operations. The runners are specialized components for submitting and monitoring different application types. The last layer consists of the *submission engines*, which are third-party tools that use the runners to submit jobs to KOALA.

4.2 Job Management for non-CS Jobs

In the KOALA job model for non-CS jobs, a job comprises either one or multiple components that each can run on a separate execution site. Each job component specifies its requirements and preferences such as the program it wants to run, the number of processors it needs, and the names of its input files. Upon receiving a job request from a runner, KOALA uses one of its placement policies, to try to place job components on execution sites. If a placement try fails, KOALA places the job at the tail of a placement queue which holds all the jobs that have not yet been successfully placed.

5. Designing Support for Cycle Scavenging in KOALA

In this section, we present our design for supporting cycle scavenging (CS) in KOALA. First, we explain the application model, then we describe the system architecture, and finally, we present the fair-share policies and scheduling at the application level, respectively.

5.1 The Application Model

In this paper, we consider in particular high-throughput applications that conform to the Parameter Sweep Application (PSA) model. A PSA can be defined as a single executable that is run for a range of parameters for a large number of times [13]. The PSA model is well suited for our problem since on the one hand many large-scale scientific applications are structured in this way, and on the other hand PSAs are very flexible to be run as CS jobs in a multicluster grid environment. We support OGF's standardized Job Description Language (JSDL 1.0 [6]) to which we have added an extension for parameter sweeps such that users can submit PSAs as single entities by specifying input files for parameter extraction and representing the ranges of parameters as loops or lists of comma-separated values.

We treat PSAs as malleable applications that can grow or shrink dynamically in terms of the number of compute nodes they execute on. Moreover, we allow PSAs to run in multiple clusters simultaneously, i.e., in a co-allocated fashion. The KOALA scheduler has been modified to handle PSAs in a different way than regular grid jobs (we explain this in the next section in more detail). A job component, in the cycle scavenging context, represents the set of tasks of a PSA that are running in the same cluster. The job components are dynamically created at run time according to the interactions with the scheduler rather than being statically specified at submission time.

5.2 System Architecture

In order to support cycle scavenging in KOALA, we have implemented two additional components, a specific KOALA runner called the *CS-Runner* and a glide-in mechanism [17] called the *Launcher*. Figure 2 illustrates the interaction between these components and the existing KOALA components that together achieve cycle scavenging. The fair-share scheduling policies have been incorporated in the existing scheduler component. The KOALA Component Manager (KCM) is our job submission daemon that runs as a separate copy per job component on the head node of the corresponding cluster. The KCM interfaces to a local resource manager (e.g., SGE [9]) through the standardized Distributed Resource Management Application API (DRMAA) [1]. We have added to the KCM the functionality of notifying the KOALA Information Service (KIS) about local (non-CS) job submissions.

Along with scheduling regular grid jobs, the scheduler is responsible for allocating and reclaiming idle nodes among active CS-Runners, based on the fair-share policy in use. The fair-share policies decide on which CS-Runners are going to be offered additional nodes (i.e., *grow*), or are going to be forced to release nodes (i.e, *shrink*), from which clusters, and how many (see Section 5.3). A CS-Runner may receive a grow request whenever the scheduler becomes aware of idle nodes in one or more clusters through the KOALA Information Service (KIS). On the other hand, a CS-Runner may receive a shrink request for one of two reasons. First, a CS-Runner may be forced to release nodes that it occupies whenever a non-CS job (at the local or grid level) demands those nodes. Secondly, depending on the fair-share policy deployed, it may also be asked to release nodes to open up space for other CS-Runners.

The CS-Runner is entitled to manage the scheduling and monitoring of CS jobs (each job refers to a parameter) on behalf of a user on the allocated idle resources. It initiates Launchers on the idle nodes to delegate the execution of the parameters. For simplicity, in this paper we restrict users to have a single CS-Runner at a time; hence, each CS-Runner process in the system corresponds to a different user. When it accepts a grow offer, a CS-Runner submits a request to the KCM to initiate Launchers on the idle nodes allocated by the scheduler, and then the parameter values are submitted to the Launchers, based on the application level policy in use (see Section 5.4). Upon successful execution of a parameter, a Launcher sends back the result to the associated CS-Runner. Upon receiving a shrink message, the Launchers are preempted according to the Launcher preemption policy deployed in the CS-Runner (see Section 5.4).

The Launcher runs an executable for the given set of parameters in sequential order that otherwise would be submitted one by one to the scheduler. The motivation behind the Launcher mechanism is to reduce the overhead by decreasing the number of job submissions which would put a considerable burden on the head nodes in a parameter study, and to have more control on application level scheduling in the CS-Runner (see Section 6.2 for the performance results). If compute nodes have multiple processors, a CS-Runner can optionally initiate a separate Launcher per processor in order to improve the performance (i.e., the throughput).

In fact, the idea of enabling the rapid execution of many jobs on clusters, creating a virtual pool of resources and bypassing the local resource manager, has previously been realized with several implementations such as the Condor Glide-In mechanism [17], MyGrid's virtual cluster ap-

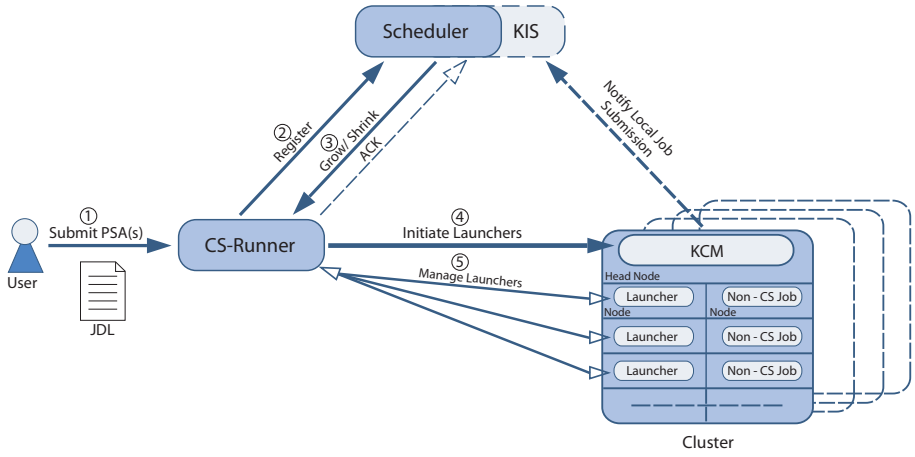

Figure 2. The system architecture to support cycle scavenging in KOALA.

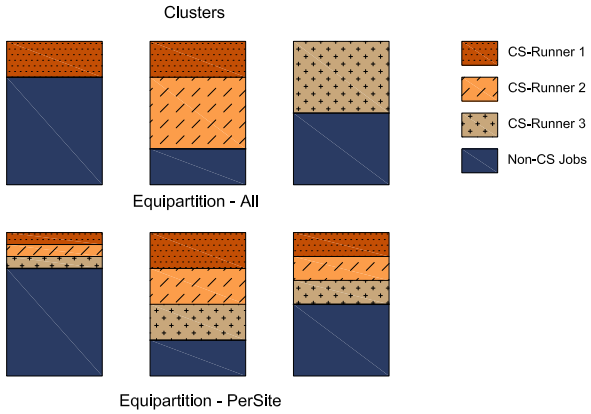

Figure 3. The distribution of the idle nodes among the CS-Runners with the Equipartition-All and the Equipartition-PerSite policies.

proach [16], and the Falkon framework [28]. The Launcher adopts the same idea; however, it differs from these mechanisms in terms of functionality as it is specialized to run complete PSAs as single entities.

5.3 Fair-Share Policies

Our solution to fair-share resource allocation is to partition the idle resource space equally among the active CS-Runners. We have designed two best-effort fair-share policies that are based on the well-known *Equipartition* policy [23], which has originally been proposed as a dynamic processor allocation scheme for malleable parallel applications running in a single cluster.

The first policy, *Equipartition-All*, tries to distribute the idle nodes (or reclaims the required number of nodes) evenly among the active CS-Runners on a grid-wide basis (see Figure 3). Whenever some of the nodes are reclaimed for non-CS jobs, the policy does not repartition the idle nodes allocated to the CS-Runners to equalize the numbers of idle nodes they occupy. Instead, the policy gives back the reclaimed nodes to the corresponding CS-Runners after the non-CS jobs are finished.

The second policy, *Equipartition-PerSite*, partitions the idle nodes on a per-cluster basis. It tries to allocate or reclaim nodes evenly in each cluster to or from the active CS-Runners. There is no need for repartitioning with the Equipartition-PerSite policy, since each CS-Runner has the same number of nodes before and after some of the nodes are reclaimed for higher priority jobs.

In comparison to the Equipartition-PerSite policy, the Equipartition-All policy may not treat users equally due to the heterogeneity of the node capabilities and the possibly different background loads on the clusters because of local or grid-level jobs. We investigate this issue and its effects in Section 6.3.

In case there are too many CS users competing for the limited numbers of idle resources, the overall throughput can substantially decrease with our dynamic fair-share policies. Therefore, we apply admission control to limit the number of active CS-Runners to improve the overall service quality. The admission control is administrated manually, yet, as a future work, we consider to implement a dynamic solution that will incorporate predictions of future availability of idle resources.

Table 1. The structure of DAS3.

Cluster Location	Nodes	Type	Speed
Vrije University	85	dual-core	2.4 GHz
U. of Amsterdam	41	dual-core	2.2 GHz
Delft University	68	single-core	2.4 GHz
MultimediaN	46	single-core	2.4 GHz
Leiden University	32	single-core	2.6 GHz

5.4 Scheduling at the Application Level

The default scheduling solution that we have implemented at the application level is based on a pure pull model. Each Launcher requests from its CS-Runner a new parameter to execute when it becomes idle. When the scheduler sends a shrink message, the desired numbers of Launchers are preempted, and the uncompleted parameters are placed back into the parameter pool of the CS-Runner. The Launcher preemption policy preempts the Launchers starting from the one that has pulled a parameter most recently, to the one that has pulled a parameter earliest, with the intention to lose less computation. The reason we prefer the pull model instead of the push model is that the latter necessitates a CS-Runner to frequently poll its Launchers for idleness.

We provide an API with which users can customize the CS-Runner, and the mentioned scheduling solutions according to their applications' characteristics and requirements. One such example might be resubmitting the preempted jobs to the sites where input files already exist (in case the parameters are input files), in order to reduce unnecessary file transfers.

6. The Performance of the CS System

In this section, we evaluate the performance of the implemented system and the scheduling strategies.

6.1 The Testbed

Our testbed is the Distributed ASCI Supercomputer (DAS3) [10], which is a wide-area computer system in the Netherlands that is used for research on parallel, distributed, and grid computing. It consists of five clusters of 272 dual AMD Opteron compute nodes. The distribution of the nodes over the clusters is given in Table 1. On each of the DAS clusters, the Sun Grid Engine (SGE) [9] is used as the local resource manager. SGE has been configured to run applications on the nodes in an exclusive fashion, i.e., in space-shared mode.

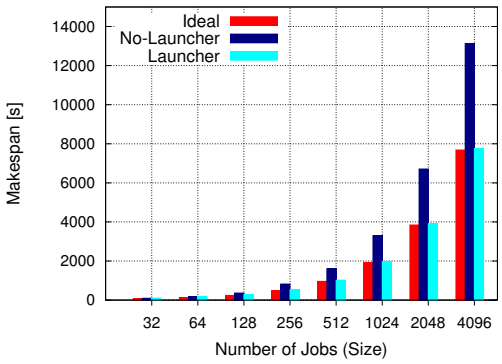

Figure 4. The makespan of PSAs for different submission mechanisms.

6.2 The Impact of the Job Submission Mechanism

In our first experiment, we demonstrate the performance gain of using the Launcher mechanism over submitting the parameters of a PSA as separate jobs to a grid system. We use a synthetic application that takes the same time, 60 seconds, to execute each of its parameters. We consider a single cluster with 32 nodes, and vary the application size (number of parameters) as powers of 2 between 32 and 4096. We submit the application using the CS-Runner with and without the Launchers for each size. For the former case, the Launchers are initiated once and pull equal numbers of parameters to execute. For the latter, the parameters are submitted as separate jobs to the grid middleware whenever the CS-Runner is notified of idle nodes by the scheduler. During the experiments, we ensure that no other jobs run in the cluster.

Figure 4 shows the results in terms of the *makespan* of the application. The makespan of a PSA can be defined as the difference between the time of the earliest submission of one of its jobs, and the time of the latest completion of one of its jobs. The ideal case assumes no overhead in the system, that is, sets of 32 jobs are placed on the nodes and started immediately. With using Launchers, the performance is close to the ideal case, irrespective of the job size. On the other hand, as the size of the application increases, the difference becomes much more visible between the regular submission of jobs and the ideal case. It leads to a difference of approximately one and a half hour when running the application of size 4096. Provided that the executable and the input files reside in the execution site, there are two sources of this difference: the job startup overhead, which is 5 seconds per job on average, and the information delay due to the polling nature of monitoring resources. The monitoring period is 60 seconds in the experiment, which can be

considered as a realistic grid monitoring setting (see, e.g., Ganglia [4]).

6.3 Performance of the Fair-Share Policies

In our second experiment, we compare the performance of the Equipartition-All and the Equipartition-PerSite policies. We assume that three CS users submit the same application with the same parameter range. The parameter sweep application we use is a program that we have implemented to solve a rewarding puzzle (2M.US$), Eternity-II [2], which is played on a square board with 16X16 spaces. The goal is to place all of the 256 pieces on the board in such a way that the patterns on adjacent sides match. Finding a solution is computationally hard; a brute-force technique requires millions of CPU years. Our solver performs random walks to yield the best solution that it can, based on the parameters given.

Although local job submissions force CS-Runners to release nodes, in this experiment, we mainly attributed preemptions to grid job submissions. We use two synthetic workloads, with different arrival patterns, in order to represent the jobs of regular grid users. The first workload, *WBlock*, periodically imposes for ten minutes a steady load of 40% on the system with a period of 20 minutes; the load is distributed non-uniformly across the clusters. The second workload, *WBurst*, imposes a 40% load (again non-uniform across the clusters) with burst submissions of 1 minute repeated every 10 minutes. The motivation behind using such workloads is to observe the performance of the policies under dynamic loads, which is a typical case for grids.

Each experiment is terminated after 1 hour. We monitor the load due to local jobs in each cluster, and ensure that this background load is steady and does not disturb the experiments. We use the tools provided within the GrenchMark project [19] to create the workloads and to ensure the correct submission of them to the system.

Figure 5 shows the performance results of the Equipartition-All and the Equipartition-PerSite policies in terms of the number of completed jobs (parameters) during the experiments, and Table 2 presents the throughput, the number of preemptions, and the average number of nodes allocated to each CS user (only for the experiments with WBlock, for brevity). With the Equipartition-All policy, we observe that the number of jobs completed (as well as the metrics in Table 2) per CS user varies considerably. With both workloads, the load is distributed in an unbalanced way across the clusters. As a consequence, with the Equipartition-All policy, CS users who happen to occupy more processors than other users in the heavily loaded clusters, suffer more due to frequent preemptions. This phenomenon, on the other hand, does not affect the CS users when the Equipartition-PerSite policy is used. That policy

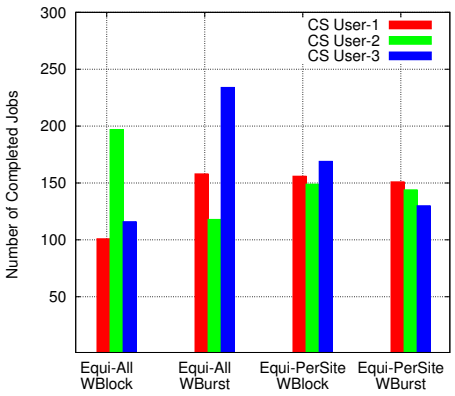

Figure 5. Number of jobs completed per CS user

always allocates an equal number of nodes in each cluster to each CS user, and therefore, each CS user would suffer or benefit equally from the behavior of non-CS jobs in a particular cluster. In this experiment, we have also verified that all the CS users receive almost the same amounts of CPU time with the Equipartition-PerSite policy. The reason for the small differences in the number of completed jobs are due to Launcher failures (the failed Launchers are restarted immediately when noticed by the CS-Runners) and to the execution time variation of the solver application due to the randomness it includes.

6.4 Unobtrusiveness of the CS System

In order to assess the unobtrusiveness of our CS system, we have performed extensive test runs to quantify the additional delay that local jobs and non-CS grid jobs experience before they start execution due to reclaiming of the nodes occupied by CS-Runners. For non-CS jobs submitted to KOALA, we observe an additional delay between 2 and 8 seconds before they start execution. Local jobs, however, experience an additional delay between 8 and 15 seconds. KCM polls the local resource manager with a period of 10 seconds to be aware of recent local job submissions. This contributes most to the additional delay that local jobs experience. To decrease the polling period would of course decrease the delay, but we have observed that periods lower than 10 seconds increase the processor load on the head node of the cluster in question considerably.

7. Conclusion

In this paper, we have presented the design and the analysis of the support for cycle scavenging in multicluster grids. We have incorporated the proposed scheduling strategies for

Table 2. Performance of the CS Policies under the WBlock workload

	Equipartition-All			Equipartition-PerSite		
	Throughput [jobs/min]	Num. of Preemptions	Avg. Num. of Nodes	Throughput [jobs/min]	Num. of Preemptions	Avg. Num. of Nodes
CS User-1	1.68	346	23	2.55	323	37
CS User-2	3.3	304	45	2.4	321	37
CS User-3	1.92	333	28	2.7	323	37

cycle scavenging in our KOALA grid scheduler. We have implemented two best-effort fair-share polices that dynamically partition the idle nodes among the active CS users. We have compared these policies with experiments conducted in a real multicluster grid system. The results show that a dynamic cycle scavenging policy should distribute the idle nodes from each cluster in equal amounts to active CS users in order to ensure fairness. In addition, we have performed experiments to show that the implemented system is efficient in terms of scheduling overhead and unobtrusiveness.

Acknowledgment

This work was carried out in the context of the Virtual Laboratory for e-Science project (www.vl-e.nl), which is supported by a BSIK grant from the Dutch Ministry of Education, Culture and Science (OC&W), and which is the part of the ICT innovation program of the Dutch Ministry of Economic Affairs.

References

[1] Distributed Resource Management Application API. http://www.drmaa.net/w/.
[2] The Eternity Puzzle. http://www.eternityii.com.
[3] Folding@home. http://folding.stanford.edu/.
[4] Ganglia monitoring system. http://ganglia.info/.
[5] Globus monitoring and discovery system. http://www.globus.org/mds/.
[6] Open Grid Forum: JSDL specification 1.0. www.ggf.org/documents/GFD.56.pdf.
[7] Rosetta@home. http://boinc.bakerlab.org/rosetta/.
[8] SETI@home. http://setiathome.ssl.berkeley.edu/.
[9] Sun Grid Engine. http://gridengine.sunsource.net.
[10] The Distributed ASCI Supercomputer. http://www.cs.vu.nl/das3/.
[11] D. P. Anderson. BOINC: A system for public-resource computing and storage. In *GRID'04*, pages 4–10, 2004.
[12] J. Buisson, O. Sonmez, H. Mohamed, W. Lammers, and D. Epema. Scheduling malleable applications in multicluster systems. In *IEEE Cluster*, Austin, USA, 17-20 September 2007.
[13] H. Casanova, A. Legrand, D. Zagorodnov, and F. Berman. Heuristics for scheduling parameter sweep applications in grid environments. In *HCW'00*, pages 349–363, Cancun, Mexico, May 2000.
[14] A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: architecture and performance of an enterprise desktop grid system. *J. of Parallel and Distrib. Comput.*, 63(5):597–610, 2003.
[15] W. Cirne, F. Brasileiro, N. Andrade, L. Costa, A. Andrade, R. Novaes, and M. Mowbray. Labs of the world, unite!!! *Journal of Grid Computing*, 4(3):225–246, 2006.
[16] W. C. et.al. J. Sauve, F. A. B. Silva, C. O. Barros, and C. Silveira. Running bag-of-tasks applications on computational grids: The mygrid approach. *ICPP'00*, 00:407, 2003.
[17] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5:237–246, 2002.
[18] A. Iosup, C. Dumitrescu, D. H. J. Epema, H. Li, and L. Wolters. How are real grids used? the analysis of four grid traces and its implications. In *GRID'06*, pages 262–269, 2006.
[19] A. Iosup and D. Epema. GRENCHMARK: A framework for analyzing, testing, and comparing grids. In *CCGRID'06*, pages 313–320, Washington, DC, USA, 2006.
[20] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, 1988.
[21] S. D. Kleban and S. H. Clearwater. Fair share on high performance computing systems: What does fair really mean? *CCGRID'03*, 0:146, 2003.
[22] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.
[23] C. McCann and J. Zahorjan. Processor allocation policies for message-passing parallel computers. In *SIGMETRICS'94*, pages 19–32, New York, NY, USA, 1994. ACM.
[24] H. Mohamed and D. Epema. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. In *IEEE Cluster*, pages 287–298. IEEE Society Press, 2004.
[25] H. Mohamed and D. Epema. The design and implementation of the Koala co-allocating grid scheduler. In *European Grid Conference*, volume 3470 of *LNCS*, pages 640–650. Springer-Verlag, 2005.
[26] H. Mohamed and D. Epema. Koala: A co-allocating grid scheduler. *Concurrency and Computation: Practice and Experience*, 20:1851–1876, 2008.
[27] M. Mutka and M. Livny. Scheduling remote processing capacity in a workstation-processing bank computing system. In *7th International Conference on Distributed Computing Systems*, pages 2–9, Berlin, Germany, September 1987.
[28] I. Raicu, Y. Zhao, C. Dumitrescu, I. Foster, and M. Wilde. Falkon: a fast and light-weight task execution framework. In *SC'07*, 2007.
[29] O. Sonmez, H. Mohamed, and D. Epema. Communication-aware job placement policies for the Koala grid scheduler. In *E-SCIENCE'06*, pages 79–87, Washington, DC, USA, 2006.