EINDHOVEN UNIVERSITY OF TECHNOLOGY

Experiences with the KOALA co-allocating scheduler in multiclusters

Citation for published version (APA):
Mohamed, H. H., & Epema, D. H. J. (2005). Experiences with the KOALA co-allocating scheduler in multiclusters. In *Proceedings of the 5th International Symposium on Cluster Computing and the Grid (CCGrid 2005, Cardiff, UK, May 9-12, 2005)* (pp. 784-791). Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/CCGRID.2005.1558642

DOI:
10.1109/CCGRID.2005.1558642

Document status and date:
Published: 01/01/2005

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 17. Nov. 2023

Experiences with the KOALA Co-Allocating Scheduler in Multiclusters

H.H. Mohamed and D.H.J. Epema
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
e-mail: H.H.Mohamed, D.H.J.Epema@ewi.tudelft.nl

Abstract

In multicluster systems, and more generally, in grids, jobs may require co-allocation, i.e., the simultaneous allocation of resources such as processors and input files in multiple clusters. While such jobs may have reduced runtimes because they have access to more resources, waiting for processors in multiple clusters and for the input files to become available in the right locations, may introduce inefficiencies. Moreover, as single jobs now have to rely on multiple resource managers, co-allocation introduces reliability problems. In this paper, we present two additions to the original design of our KOALA co-allocating scheduler (different priority levels of jobs and incrementally claiming processors), and we report on our experiences with KOALA in our multicluster testbed while it was unstable.

1 Introduction

Grids offer the promise of transparent access to large collections of resources for applications demanding many processors and access to huge data sets. In fact, the needs of a single application may exceed the capacity available in each of the subsystems making up a grid, and so *co-allocation*, i.e., the simultaneous access to resources of possibly multiple types in multiple locations, managed by different resource managers [5], may be required.

Even though multiclusters and grids offer very large amounts of resources, to date most applications submitted to such systems run in single subsystems managed by a single scheduler. With this approach, grids are in fact used as big load balancing devices, and the function of a grid scheduler amounts to choosing a suitable subsystem for every application. The real challenge in resource management in grids lies in co-allocation. Indeed, the feasibility of running parallel applications in multicluster systems by employing processor co-allocation has been demonstrated [18, 2]. Co-allocation faces a real challenge in coping with the inherent unreliability of grids: A single failure in any of the sites participating in executing a job with co-allocation may cause the whole job to fail.

In previous work, we have studied our co-allocation policy called Close-to-Files with and without replication at a time when our testbed proved to be very stable [12]. The contributions of this paper are the addition to our KOALA co-allocating scheduler of job priorities, of fault-tolerance mechanisms, and of the Incremental Claiming Policy (ICP), which is used to claim processors in multiple clusters for single jobs in the absence of support for processor reservation by local resource managers. In addition, we report on experiments with KOALA in our DAS testbed (see Section 2.1), which proved to be rather unreliable at the time of the experiments. As far as we know, this is the first co-allocating scheduler for multiclusters that has been thoroughly tested.

Our results show that 1) assigning priorities to the jobs helps to get the important jobs to finish quickly despite the unreliable system (as long as they are not very large), that 2) large jobs are a problem also in multicluster systems even when we allow them to be allocated across the system in many different ways, 3) that many jobs use co-allocation when given the chance, and that 4) even with high failure rates, KOALA succeeds in getting all jobs submitted in our experiments completed successfully.

2 A Model for Co-allocation

In this section, we present our co-allocation model in multiclusters and more generally, in grids.

0-7803-9074-1/05/$20.00 ©2005 IEEE

2.1 The Distributed ASCI Supercomputer

The DAS [19] is a wide-area computer system consisting of five clusters (one at each of five universities in the Netherlands, amongst which Delft) of dual-processor Pentium-based nodes, one with 72, the other four with 32 nodes each. The clusters are interconnected by the Dutch university backbone (100 Mbit/s), while for local communications inside the clusters Myrinet LANs are used (1200 Mbit/s). The system was designed for research on parallel and distributed computing. On single DAS clusters the scheduler is PBS [21]. Each DAS cluster has its own separate file system, and therefore, in principle, files have to be moved explicitly between users' working spaces in different clusters.

2.2 System Model

We assume an environment consisting of geographically distributed and interconnected sites (clusters) like the DAS. Each site contains computational resources (processors), a file server, and a local resource manager. The sites may combine their resources to be scheduled by a *grid scheduler* for executing jobs in a grid. The sites where a job runs are called its *execution sites*, and the site(s) where its input file(s) resides are its *file sites*. In this paper, we assume a single central grid scheduler, and the site where it runs is called the *submission site*. Of course, we are aware of the drawbacks of a single central submission site and currently we are working on extending our model to multiple submission sites.

2.3 Job Model

By a job we mean a *parallel application* requiring files and processors that can be split up into several *job components* which can be scheduled to execute on multiple execution sites simultaneously (*co-allocation*) [3, 1, 5, 12]. This allows the execution of large parallel applications requiring more processors than available on a single site [12]. Job requests are supposed to be *unordered*, meaning that a job only specifies the numbers of processors needed by its components, but not the sites where these components should run. It is the task of the grid scheduler to determine in which cluster each job component should run, to move the executables as well as the input files to those clusters before the job starts, and to start the job components simultaneously. Multiple components of a job can be scheduled on the same cluster.

We assume that the input of a whole job is a single data file. We deal with two models of file distribution to the job components. In the first model, job components work on different chunks of the same data file, which has been partitioned as requested by the components. In the second model, the input to each of the job components is the whole data file. The input data files have unique logical names and are stored and possibly replicated at different sites. We assume that there is a replica manager that maps the logical file names specified by jobs onto their physical location(s).

2.4 Job Priorities

A job priority is used to determine the importance of a job relative to other jobs in the system. The priority levels, which are *high* and *low*, are assigned to the jobs by our scheduler (see Section 2.5). The priority levels play a big part during the placement of a job, i.e., when finding a suitable pairs of execution site and file site for the job components (see Section 3.1). Moreover as we do not claim the processors allocated to a job immediately, when a job of a high priority is about to start executing, it is possible that not enough processors are available anymore. Then, a job of high priority may preempt low-priority jobs until enough idle processors for it to execute are freed (see Section 3.3).

2.5 The KOALA Co-Allocator

We have developed a processor and data co-allocator named KOALA for our DAS system. It employs Grid services such as job execution, file transfer, replica management, and security and authentication. KOALA accepts job requests and uses a placement algorithm (see Section 2.6) to try to place jobs. We use the Globus Resource Specification Language (RSL) [20] as our job description language with the RSL "+" construct to aggregate job components to form multi-requests. On success of job placement, KOALA first initiates the third-party file transfers from the selected file sites to the execution sites of the job components, and only then it tries to claim the processors allocated to the components using our processor claiming policy (see Section 3.3). If the claiming can proceed, the components are sent for execution to their respective execution sites. Synchronization of the start of the job components is achieved through a piece of code which delays the execution of the job components until the job actually starts (see Section 3.3).

2.6 The Close-to-Files Placement Policy

Placing a job in a multicluster means finding a suitable set of execution sites for all of its components and suitable file sites for the input file. (Different components may get the input file from different locations.) The most important consideration here is of course finding execution sites with enough processors. However, when there is a choice among execution sites for a job component, we choose the site such that the (estimated) delay of transferring the input file to the execution site is minimal. We call the placement policy doing just this the Close-to-Files (CF) placement policy. A more extensive description of this policy can be found in [12].

2.7 Processor Reservation

When a job is successfully placed by CF, we do no claim the processors immediately, because its input file will in general not be present at its execution sites. It is possible to estimate the job's start time based on the file transfers that have to take place, and reserve processors in the job's execution sites. However, openPBS [21], which is used in our testbed (see Section 2.1), and many resource managers in grids in general, do not support processor reservations. Of course, we can add a small piece of code to the executable of the job with the sole purpose of delaying the execution of the job barrier and start the job rightaway, but this is wasteful of processor time. Therefore, to achieve co-allocation, we postpone starting the application until a later time, running the risk that processors may not be available anymore. This is our *Incremental Claiming Policy* (ICP), which is discussed in detail in Section 3.3.

3 Scheduling Jobs

In this section, we present our job placement and claiming mechanisms. We first describe the way we place the jobs of the two priority levels. Then we present our *Incremental Claiming Policy* (ICP) for claiming processors for a job. Here, by *claiming* for a job we mean starting its components at their respective execution sites.

3.1 Placing Jobs

When a job is submitted to the system, KOALA tries to place it according to CF. If this *placement try* fails, KOALA adds the job to the tail of either the so-called *low-priority placement queue* if the job's priority is low, or the so-called *high-priority placement queue* if the job is of high priority. These queues hold all jobs that have not yet been successfully placed. KOALA regularly scans the placement queues one at a time from head to tail to see whether any job in them can be placed. In order to give preference to the high-priority queue, we successively scan this queue S times before scanning the low-priority queue once; S, which is an integer greater than 1, is a parameter of KOALA. The time between successive scans of the placement queues is a fixed interval (which is another parameter of the KOALA).

For each job in the queues we maintain its number of placement tries, and when this number exceeds a threshold, the job submission fails. This threshold can be set to ∞, i.e., no job placement fails.

3.2 Claiming Jobs

After the successful placement of a job, we estimate its File Transfer Time (FTT) and its Job Start Time (JST), and add it to the so-called *claiming queue*. This queue holds all jobs which have been placed but for which we still have to claim (part of) the processors. The job's FTT is calculated as the maximum of all of its components' estimated transfer times, and the JST is estimated as the sum of its Job Placement Time (JPT) and its FTT (see Figure 1).

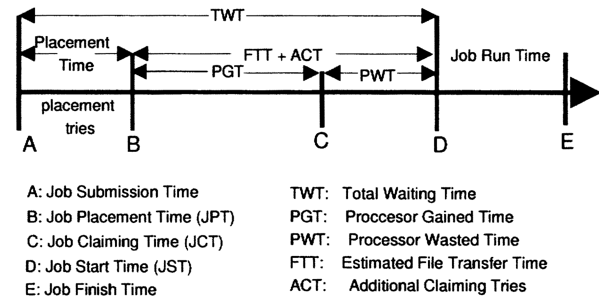


Figure 1. The timeline of a job submission.

A job's Job Claiming Time (JCT, point C in Figure 1), which is the time when we try to claim processors for it, is initially set to the sum of its JPT and the product of L and FTT:

$$JCT_0 = JPT + L * FTT,$$

where L is a parameter assigned to each job by KOALA, with $0 < L < 1$. In the claiming queue, jobs are arranged in increasing order of their JCTs.

We try to claim for a job at the current JCT by using our incremental claiming policy (see Section 3.3). The job is removed from the claiming queue only if this *claiming try* is successful for all of its components. Otherwise, we perform successive claiming tries. For each such try we recalculate the JCT by adding to the current JCT the product of L and the time remaining until the JST (time between points C and D in Figure 1):

$$JCT_{n+1} = JCT_n + L * (JST - JCT_n).$$

If the job's JCT_{n+1} reaches its JST and claiming for some of its components is still not successful, the job is returned to the placement queue (see Section 3.1). Then, its components that were already successfully started in one of the claiming tries are aborted. In addition, so as to increase the chance of successful claiming, its parameter L is decreased by a fixed fraction, unless L has reached its lower bound. If the number of times we have performed claiming tries for the job exceeds some threshold (which can be set to ∞), the job submission fails.

A new JST is estimated each time a job is returned to the placement queue and re-placed with our CF policy. We define the *start delay* of a job to be the difference between the last JST where the job execution succeeds and the original JST.

3.3 The Incremental Claiming Policy

Claiming processors for job components starts at a job's initial JCT and is repeated at subsequent claiming tries. In one claiming try, we may only be able to claim processors for some but not all components. Claiming for a component will only succeed if there are still enough idle processors to run it. Since we want jobs to start with minimal delay, we possibly re-place their components with the CF policy to find new pairs of execution site-file site, and we allow high-priority jobs to preempt low-priority jobs. This is our Incremental Claiming Policy (ICP) which is described below.

When claiming for a job, ICP first calculates F, which is the fraction of its components that have been previously started, or that can be started immediately in the current claiming try. If F is lower than its lower bound T, the job is returned to the claiming queue. Otherwise, for each component that cannot be started, ICP tries to find a new pair of execution site-file site with the CF policy such that it is possible to transfer file between the new pair before the JST. On success, the new pair replaces the component's execution site-file site pair and the file transfer is initiated immediately.

For a job of high priority, if the re-placement of the component failed or if the file cannot be transferred before the JST, we consider preempting low-priority jobs. At the execution site of the job component, the policy checks whether the number of processors the component requests does not exceed the number of idle processors the the number of processors currently being used by low-priority jobs. If so, the policy preempts those low-priority jobs in descending order of their JST until a sufficient number of processors has been freed. The preempted jobs are then returned to the low-priority placement queue.

Finally, those components for which claiming is successful in this claiming try are started; they will run a small piece of code that has been added with the sole purpose of delaying the execution of the job barrier until the job start time. Synchronization is achieved by making each component wait on the barrier until it hears from all the other components.

When T is set to 1 the claiming process becomes atomic, i.e., claiming only proceeds if it is successful for all the job components simultaneously.

3.4 Fault Tolerance

In the course of executing a job, a number of errors may arise at its execution sites. These errors, which may interrupt the execution, may be hardware or software related and our scheduler is notified on their occurrence. The failure of any one of its components causes the whole job to be aborted and to be returned to its respective placement queue. KOALA counts the number of errors of each cluster and if the number of consecutive errors reaches a certain threshold, that cluster is marked unusable. It should be noted that our fault tolerance mechanism focuses only on ensuring that the application is restarted. It is left for the application to employ mechanisms like checkpointing to ensure that it continues with the execution rather than start from the beginning.

4 Experimental Setup

In this section we describe the experiments we have conducted to assess our co-allocation service. In our experiments, we did not impose limits on the numbers of placement and claiming tries. We fixed the interval between successive scans of the placement queues at 4 minutes. The parameter L determining when to start claiming is set at 0.75. At each re-computation of job's JST, the fixed fraction used to decrease L, is set to 0.125; L has 0.25 as a

lower bound. The parameter T of our claiming algorithm (see Section 3.3) is set to 0, so we claim processors for any number components we can. The parameter S which determines how often we scan the high-priority placement queue for every scan of the low-priority placement queue is set to 2.

4.1 The Test Application

In our experiments, we use an artificial application, which consists of a real parallel application to which, because it uses little input data itself, we have added large input file. The file is only being transferred, but not actually used, and the application simply deletes it when it exits. We have previously adapted this application, which implements an iterative algorithm to solve the two-dimensional Poisson equation on the unit square, to co-allocation on the DAS [2]. In the application, the unit square is split up into a two-dimensional pattern of rectangles of equal size among the participating processors. When we execute this application on multiple clusters, this pattern is split up into adjacent vertical strips of equal width, with each cluster using an equal number of processors.

4.2 The Workload

In our experiments, we put a workload of jobs to be co-allocated on the DAS that all run the application of Section 4.1, in addition to the regular workload of the ordinary users. In our experiments, we consider job sizes 36 and 72, and four number of components, which are 3, 4, 6, or 8. The components are of equal size, which is obtained by dividing the job size by the number of components. We restrict the component sizes to be greater than 8, so the jobs of size 72 can have any of the four sizes, while those of size 36 have 3 or 4 components. Each job component requires the same single file of either 4 or 8 GByte. The input files, which are randomly distributed, are replicated in two different sites. For a single job, its priority, its number of components, its job size, and the size of its input file are picked at random and uniformly. The execution time of our test application ranges between 30.0 and 192.0 seconds.

We assume the arrival process of our jobs at the submission site to be Poisson. Based on the above description and the total number of available processors in the DAS, we generate a workload of 500 jobs. However, at the submission site, we limit the sum of the lengths of the placement queues to be 100. If this number is reached, the submission of jobs is in principle, if the system were stable delayed until it drops below 100 again.

4.3 The Testbed State

The experiments were done on the DAS system right after a major upgrade of the operating system and the local resource manager (openPBS). The system, which is homogeneous and centrally managed, was still unstable and hence unreliable during the experiments. This gave us the opportunity to evaluate our co-allocation service on a grid-like environment where the job failure rate is high. The fact that this rate is high in such an environment shows the strong need for good fault tolerance.

5 Results

In this section, we present the results of our experiment with KOALA in the DAS.

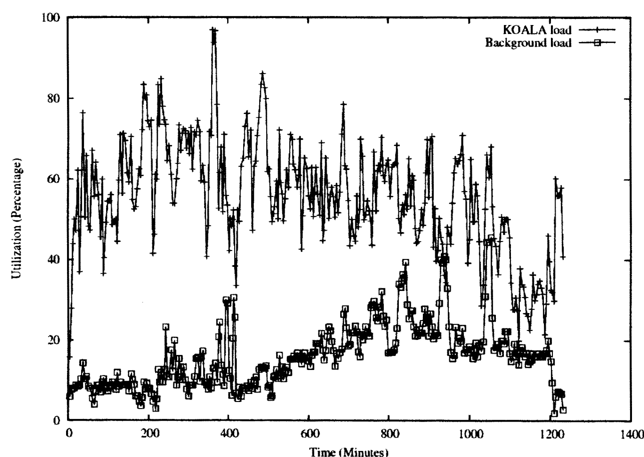


Figure 2. The system utilization during the experiment.

5.1 Utilization

At the start of the experiment, a total of 310 processors in 4 out of the 5 DAS clusters were available to KOALA. During the experiment, one of the clusters reported a much higher consecutive number of failures and was removed for selection by KOALA (Section 3.4). As a result, the number of available processors was reduced to 250. The utilization

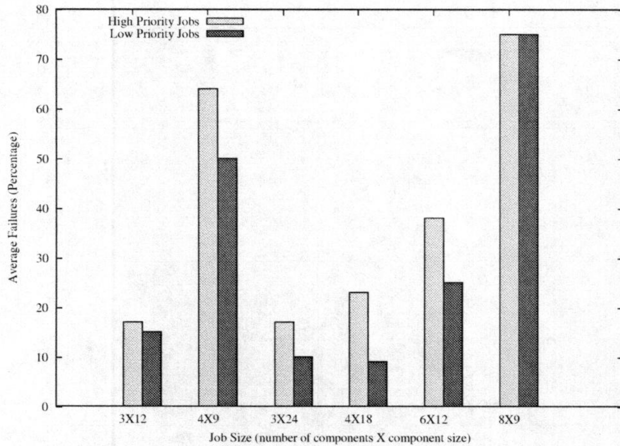


Figure 3. The percentages of failures of jobs of different sizes.

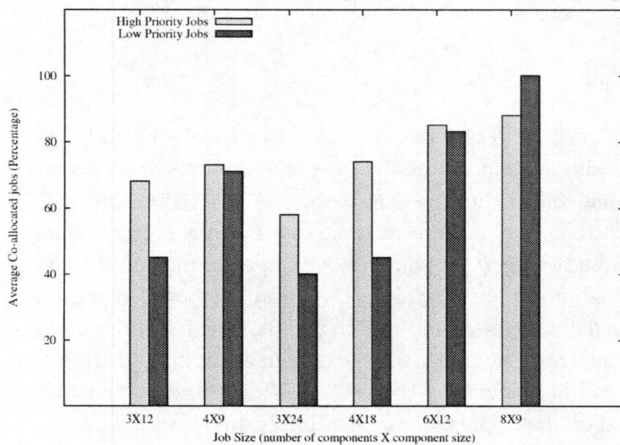


Figure 4. The percentages of jobs that use co allocation.

of these processors by jobs due to other DAS users and to KOALA are shown in Figure 2. In this figure we see that up to 80%– 90% of the system was used during the experiment, which shows that co-allocation can drive the utilization to quite high levels.

5.2 Failures

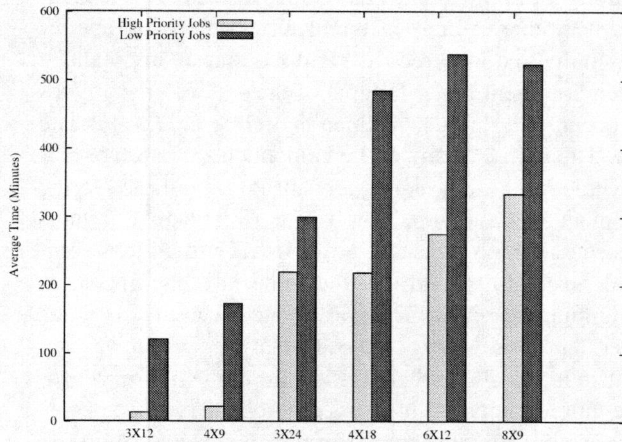
In this paper, by the failure of a job we mean that one the clusters' local resource managers has reported a failure during the execution of the corresponding job component. Such a failure may occur when claiming has not yet succeeded for all job components, or when the system is really executing the application. The failures of the resource managers were due to bugs in them; also some of the nodes were configured incorrectly. We think that in any realistic grid environment many failures of these (and other) types will occur, and that a scheduler should be able to tolerate them. The failure of any of the components of a job causes the whole job to fail, and as a result, to be returned to the placement queue. Figure 3 shows the percentage of failures for each of the job sizes. Note that in the end all jobs completed successfully, and that the same job may appear as failing multiple times. The percentage of failures is much higher than what we experienced when the system was stable; then it was always below 15%. From the figure, we observe more failures for high priority-jobs. This is expected because more attempts are performed to place, to claim for, and therefore to run these jobs. As a result, more jobs are started simultaneously, which results in some components to be given mis-configured nodes. The failures also increase when the number of components increases, because then the chances for components to be placed on different clusters (co-allocated, see Figure 4) and hence on unreliable nodes, increases.

5.3 Placing and Claiming

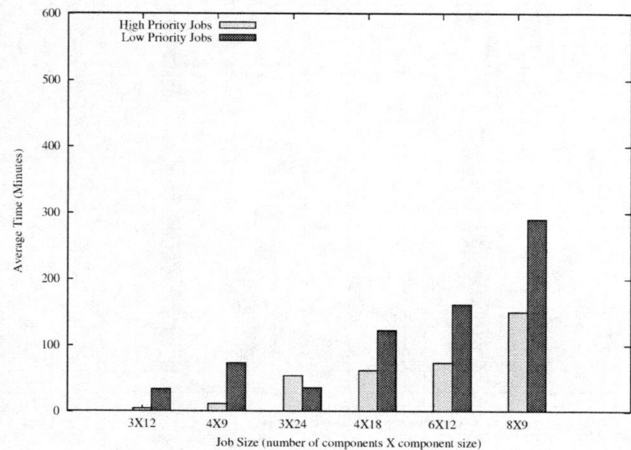
Failed jobs are returned to their respective placement queues in KOALA, which then tries to re-place them until their execution succeeds. As a result of re-placements and of higher demands for processors, the job placement times increase considerably with the increase of the number of components as shown in Figure 5(a). We do emphasize however, that all our jobs eventually ran to completion successfully. Jobs of small total sizes do not suffer from many re-placements or long waiting times for enough free processors to be available. Yet these jobs still require co-allocation as shown in Figure 4, which helps to reduce their placement times (Figure 5(a).

Figure 5(b) shows the start delays of jobs of different sizes, which is also affected by the number of failures. Like in the above observations, the start delay increases with the number of components, with high priority jobs performing better compared to low priority jobs. From Figure 5 we find that only for relatively small high-priority jobs the placement time and the start delay are reasonable.

Overall, splitting jobs into many components does not necessarily guarantee smaller placement times. On the other hand, small jobs with small numbers of components still require co-allocation during placement to guarantee smaller placement times and start delays. Nevertheless, we cannot conclude that jobs of smaller sizes perform much better, but



(a) Placement Times



(b) Start Delays

Figure 5. The Placement Times and Start Delays of jobs.

rather we can conclude that our co-allocation service starves jobs requesting many processors in these conditions. Finally, despite an unreliable system, still jobs of high priority outperform jobs of low priority.

6 Related Work

In [8, 9], co-allocation (called multi-site computing there) is studied also with simulations, with as performance metric the (average weighted) response time. There, jobs only specify a total number of processors, and are split up across the clusters. The slow wide-area communication is accounted for by a factor r by which the total execution times are multiplied. Co-allocation is compared to keeping jobs local and to only sharing load among the clusters, assuming that all jobs fit in a single cluster. One of the most important findings is that for r less than or equal to 1.25, it pays to use co-allocation. In [15] an architecture for a grid superscheduler is proposed, and three job migration algorithms are simulated. However, there is no real implementation of this scheduler, and jobs are confined to run within a single subsystem of a grid, reducing the problem studied to a traditional load-balancing problem.

In [13], the Condor class-ad matchmaking mechanism for matching single jobs with single machines is extended to "gangmatching" for co-allocation. The running example in [13] is the inclusion of a software license in a match of a job and a machine, but it seems that the gangmatching mechanism might be extended to the co-allocation of processors and data.

Thain et al. [17] describe a system that links jobs and data by binding execution and storage sites into I/O communities that reflect the physical reality. A job requesting particular data may be moved to a community where the data are already staged, or data may be staged to the community in which a job has already been placed. Other research on data access has focused on the mechanisms for automating the transfer of and the access to data in grids, e.g., in Globus [20] and in Kangaroo [16], although there less emphasis is placed on the importance of the timely arrival of data.

In [14], the scheduling of sequential jobs that need a single input file is studied in grid environments with simulations of synthetic workloads. Every site has a Local Scheduler, an External Scheduler (ES) that determines where to send locally submitted jobs, and a Data Scheduler (DS) that asynchronously, i.e., independently of the jobs being scheduled, replicates the most popular files stored locally. All combinations of four ES and three DS algorithms are studied, and it turns out that sending jobs to the sites where their input files are already present, and actively replicating popular files, performs best.

In [6], the creation of abstract workflows consisting of application components, their translation into concrete workflows, and the mapping of the latter onto grid resources is considered. These operations have been implemented using the Pegasus [7] planning tool and the Chimera [10] data definition tool. The workflows are represented by DAGs, which are actually assigned to resources using the Condor DAGMan and Condor-G [11]. As DAGs are involved,

no simultaneous resource possession implemented by a co-allocation mechanism is needed.

In the AppLes project [4], each grid application is scheduled according to its own performance model. The general strategy of AppLes is to take into account resource performance estimates to generate a plan for assigning file transfers to network links and tasks (sequential jobs) to hosts.

7 Conclusions

We have addressed the problem of scheduling jobs consisting of multiple components that require both processor and data co-allocation in the environment that exhibits many failures. We have presented the fault tolerance additions made to the KOALA scheduler to enable it to cope with unstable environments. Our results show the correct and reliable operation of KOALA at the time where our testbed was unreliable.

As future work, we are planning to remove the bottleneck of a single global scheduler, and to allow flexible jobs that only specify the total number of processors needed and allow KOALA to fragment jobs into components (the way of dividing the input files across the job components is then not obvious). In addition, more extensive performance study of KOALA in a heterogeneous grid environment is planned.

References

[1] S. Ananad, S. Yoginath, G. von Laszewski, and B. Alunkal. Flow-based Multistage Co-allocation Service. In B. J. d'Auriol, editor, *Proc. of the International Conference on Communications in Computing*, pages 24–30, Las Vegas, 2003. CSREA Press.

[2] S. Banen, A. Bucur, and D. Epema. A Measurement-Based Simulation Study of Processor Co-Allocation in Multicluster Systems. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 105–128. Springer-Verlag, 2003.

[3] A. Bucur and D. Epema. Local versus Global Queues with Processor Co-Allocation in Multicluster Systems. In D. Feitelson, L. Rudolph, and U. Schwiegelshohn, editors, *8th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2537 of *LNCS*, pages 184–204. Springer-Verlag, 2002.

[4] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-Level Middleware for the Grid. pages 75–76, 2000.

[5] K. Czajkowski, I. T. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proc. of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[6] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, and K. Vahi. Mapping Abstract Complex Workflows onto Grid Environments. *J. of Grid Computing*, 1:25–39, 2003.

[7] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. M. K. Vahi, S. Koranda, A. Lazzarini, and M. A. Papa. From Metadata to Execution on the Grid Pegasus and the Pulsar Search. Technical report, 2003.

[8] C. Ernemann, V. Hamscher, U. Schwiegelshohn, R. Yahyapour, and A. Streit. On Advantages of Grid Computing for Parallel Job Scheduling. In *2nd IEEE/ACM Int'l Symposium on Cluster Computing and the GRID (CCGrid2002)*, pages 39–46, 2002.

[9] C. Ernemann, V. Hamscher, A. Streit, and R. Yahyapour. Enhanced Algorithms for Multi-Site Scheduling. In *3rd Int'l Workshop on Grid Computing*, pages 219–231, 2002.

[10] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *14th Int'l Conf. on Scientific and Statistical Database Management (SSDBM 2002)*, 2002.

[11] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, San Francisco, California, August 2001.

[12] H. Mohamed and D. Epema. An Evaluation of the Close-to-Files Processor and Data Co-Allocation Policy in Multiclusters. In *Proc. of CLUSTER 2004, IEEE Int'l Conference Cluster Computing 2004*, September 2004.

[13] R. Raman, M. Livny, and M. Solomon. Policy driven heterogeneous resource co-allocation with gangmatching. In *12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, pages 80–89. IEEE Computer Society Press, 2003.

[14] K. Ranganathan and I. Foster. Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications. In *11 th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 Edinburgh, Scotland*, July 2002.

[15] H. Shan, L. Oliker, and R. Biswas. Job superscheduler architecture and performance in computational grid environments. In *Supercomputing '03*. 2003.

[16] D. Thain, J. Basney, S. son, and M. Livny. The Kangaroo Approach to Data movement on the Grid. In *proc. of the Tenth IEEE Symposium on High Performance Distributed Computing, San Francisco, California*, August 2001.

[17] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Gathering at the Well: Creating Communities for Grid I/O. In *Proc. of Supercomputing, Denver, Colorado*, November 2001. http://www.cs.wisc.edu/condor/nest/papers/community.pdf.

[18] R. van Nieuwpoort, J. Maassen, H. Bal, T. Kielmann, and R. Veldema. Wide-Area Parallel Programming Using the Remote Method Invocation Method. *Concurrency: Practice and Experience*, 12(8):643–666, 2000.

[19] Web-site. The distributed asci supercomputer (das). `http://www.cs.vu.nl/das2`.

[20] Web-site. The globus toolkit. `http://www.globus.org/`.

[21] Web-site. The portable batch system. `www.openpbs.org`.