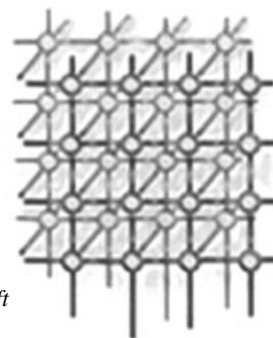

KOALA: a co-allocating grid scheduler

Hashim Mohamed^{*,†} and Dick Epema

Faculty of Electrical Engineering, Mathematics, and Computer Science, Delft University of Technology, P.O. Box 5031, 2600 GA Delft, The Netherlands



SUMMARY

In multicluster systems, and more generally in grids, jobs may require *co-allocation*, that is, the simultaneous or coordinated access of single applications to resources of possibly multiple types in multiple locations managed by different resource managers. Co-allocation presents new challenges to resource management in grids, such as locating sufficient resources in geographically distributed sites, allocating and managing resources in multiple, possibly heterogeneous sites for single applications, and coordinating the execution of single jobs at multiple sites. Moreover, as single jobs now may have to rely on multiple resource managers, co-allocation introduces reliability problems. In this paper, we present the design and implementation of a co-allocating grid scheduler named KOALA that meets these co-allocation challenges. In addition, we report on the results of an analysis of the performance in our multicluster testbed of the co-allocation policies built into KOALA. We also include the results of a performance and reliability test of KOALA while our testbed was unstable. Copyright © 2007 John Wiley & Sons, Ltd.

Received 19 July 2006; Revised 27 July 2007; Accepted 2 August 2007

KEY WORDS: co-allocation; grid computing; scheduling; performance

1. INTRODUCTION

Grid computing has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale, multi-organizational resource sharing, and innovative applications requiring many resources of different types. It is common for the resource needs of grid applications to go beyond what is available in any of the sites making up a grid. For example, a parallel application may require more processors than that are present at any site, and a simulation may require processors for computation in one site and visualization equipment in another site. To run such applications, a technique called *co-allocation*, that is, the simultaneous or coordinated

*Correspondence to: Hashim Mohamed, 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@tudelft.nl



access of single applications to resources of possibly multiple types in multiple locations managed by different resource managers [1], is required.

Co-allocation presents the following challenges in resource management in grids:

1. Allocating resources in multiple sites, which may be heterogeneous in terms of hardware, software, and access and usage policies, to single applications.
2. Guaranteeing the simultaneous availability of the allocated resources at the start of the execution of an application.
3. Managing highly dynamic grid resources, which may come and go, either by being disconnected or by failing, at any time.

In order to address the problem of co-allocation and to meet these three challenges, we have designed, implemented, and deployed a co-allocating grid scheduler called KOALA[‡], which features co-allocation of processors and files. For co-allocating these resources to single applications, KOALA has two built-in so-called placement policies, namely, the Close-to-Files (CF) policy and the Worst Fit (WF) policy; new placement policies can be added without affecting the overall operation of KOALA. CF addresses the problem of long delays when starting a job because of long input file transfers by selecting the execution sites of its components (i.e. the parts of a job that can run on different sites) close to sites where their input files are located. On the other hand, the WF policy simply places job components on the sites with the largest numbers of idle processors. In doing so, WF balances the numbers of idle processors across the grid. The placement policies extensively use the KOALA Information Service (IS) to locate and monitor resource availability.

Due to potentially long input file transfer times (FTTs), the actual start of a job's execution (its job start time (JST)) may be much later than the time when the job was allocated processors (its job placement time (JPT)). This means that when the allocated processors are claimed immediately at the JPT, much processor time is wasted. In order to prevent this and still guarantee the simultaneous availability of processors at the JST in the absence of support for advance processor reservation by local resource managers, KOALA implements the Incremental Claiming Policy (ICP). If needed because some of the allocated processors have been taken by other jobs, in an effort not to delay the JST, ICP tries to make processors available for job components by finding processors at other sites or, if permitted, by forcing processor availability through pre-emption of running jobs.

In order to deal with the dynamicity of the grid resources and reliability problems of some grid components, KOALA incorporates fault-tolerance mechanisms to ensure that grid jobs are completed successfully.

The major contributions of this paper are (1) the design, the implementation, and the deployment of a reliable co-allocating grid scheduler, (2) the design and analysis of two co-allocation policies, and (3) an alternative to advance processor reservation when such a mechanism is absent in local resource managers for achieving the simultaneous availability of allocated processors.

This paper is organized as follows. In Section 2 we further elaborate on the problem of co-allocation, and in Section 3 we present a model for co-allocation in grids. In Section 4 we describe the design of KOALA, and in Sections 5 and 6 we present its placement and processor claiming

[‡]The name KOALA was solely chosen for its similarity in sound with the word co-allocation.



policies, respectively. The results of experiments with the placement policies on our Distributed ASCI Supercomputer (DAS) testbed (see Section 7.1) are the subject of Section 7. We have also done experiments with KOALA while our testbed was unreliable, and the results of these experiments are presented in Section 8. In Section 9 we discuss related work, and finally conclusions and plans for future work are presented in Section 10.

2. THE PROBLEM OF CO-ALLOCATION IN GRIDS

In this section, we elaborate on some aspects of the problem of co-allocation in grids as defined in the Introduction. In particular, we discuss the motivation from applications for co-allocation and the issue of (the lack of) advance processor reservation.

2.1. Motivation from applications for co-allocation

In this paper, co-allocation means the simultaneous or coordinated possession by applications of resources in separate, possibly independently administered sites in multicluster systems or grids. Co-allocation can be motivated by the desire to have applications run faster than is possible in any site of a grid, or by the desire to run applications that need sets of resources that are simply not available in any single site. For example, with co-allocation, it is possible to run parallel applications that require, or that can make efficient use of, more processors than are available at any site in a grid. Co-allocation is also useful for specific types of applications, e.g. applications that access and/or process geographically spread data that are not allowed to be moved, and applications that perform simulations in one site and visualization in another site. The access to multiple types of resources in multiple sites can be either simultaneous or in a coordinated fashion. Simultaneous processor access is, for instance, needed for parallel applications, while coordinated access to resources is useful for application types that require their components to obey precedence constraints, e.g. workflows. Currently, we only deal with simultaneous access of resources while coordinated access to resources at multiple sites is work in progress. We further restrict resources to processors and files.

2.2. Processor reservations

The challenge with simultaneous access of an application to resources at multiple sites of a grid lies in guaranteeing their availability at the application's start time. The most straightforward strategy to do so is to reserve processors at each of the selected sites. If the local schedulers do support reservations, this strategy can be implemented by having a grid scheduler obtain a list of available time slots from each local scheduler, reserve a common time slot for all components of an application, and notify the local schedulers of this reservation. Unfortunately, a reservation-based strategy in grids is currently limited due to the fact that only few local resource managers support reservations (for instance, PBS-pro [2] and Maui [3] do). Even for those resource managers, only privileged users or specially designated user groups are allowed to perform processor reservations, in order to prevent users from abusing the reservation mechanism. In the absence of, or in the presence of only limited processor-reservation mechanisms, good alternatives are required in order to achieve co-allocation. We address this issue in KOALA in Section 6.



3. A MODEL FOR CO-ALLOCATION IN GRIDS

In this section we present a model for the version of the co-allocation problem in multiclusters and grids addressed in this paper with the design and implementation of our KOALA scheduler. This version has restrictions in terms of the type of co-allocation that we deal with (only simultaneous resource possession), the types of resources that we take into account (only processors and files), and the types of applications that we consider (only parallel high-performance applications).

3.1. The system model

In our system model, we assume a multicluster environment with sites that each contain computational resources (processors), a file server, and a local resource manager. The resources of the individual sites can be used by either local jobs or co-allocated jobs. The sites where the components of a job run are called its *execution sites*, and the site(s) where its input file(s) reside are its *file sites*. We assume a grid scheduler through which all grid jobs are submitted. The sites where jobs are submitted from are called the *submission sites*. A submission site can be any site in a grid or a desktop computer. The grid scheduler allows us to perform resource brokering and scheduling across its authorized domain in the grid.

3.2. The job model

In this paper, a job consists of one or more *job components* that collectively perform a useful task for a user. The job components contain information such as their numbers of processors, the sizes and locations of their input files, and the required runtime libraries, needed for scheduling and executing an application across different sites in a grid. For a parallel job, the number of processors it requires can be split up to form several job components, which can then be scheduled to execute on multiple sites simultaneously (co-allocation) [1,4,5]. By doing so we are able to run large parallel applications that require more processors than are available on a single site [5,6]. Jobs that require co-allocation in grids may or may not specify the execution sites of their components. Based on this, we consider two cases for the structure of the job requests, which are already supported by our scheduler:

1. *Fixed request*: The job request specifies the numbers of processors it needs in all the clusters from which processors must be allocated for its components.
2. *Non-fixed request*: The job request only specifies the numbers of processors its components need, allowing the scheduler to choose the execution sites. The scheduler can place different job components on the same or on different sites depending on the availability of processors.

In both cases, the scheduler has the task of moving the executables as well as the input files to the execution sites before the job starts, and of starting the job components simultaneously.

3.3. The file distribution model

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, the 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. We assume the data files to be read-only and, therefore, they can be shared by other jobs. This is a reasonable assumption as is discussed in several data grid scenarios [7]. 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).

3.4. Job priorities

In real systems, the need for priorities may arise to give some jobs preferential treatment over others. For instance, some jobs may have (soft) deadlines associated, or may need interaction from the user. Therefore, we have introduced the priority of a job, which is used to determine its importance relative to other jobs in the system. There are four priority levels, which are *super-high*, *high*, *low*, and *super-low*, and which are assigned to the jobs by our scheduler based on system policies. An example of these policies is to assign priority levels to jobs based on their estimated job runtimes. Moreover, a specific group or type of applications can be assigned a specific priority level. For instance, the super-high priority and the super-low priority can be assigned to interactive jobs and to the so-called 'screensaver applications', respectively.

The priority level plays a part during the placement of a job, i.e. when finding suitable pairs of execution sites and file sites for the job components (see Section 5.2), and when submitting the components for execution (see Section 6.2).

4. The Design of KOALA

This section describes the design of the KOALA grid scheduler. KOALA, which started as a prototype named the Processor and Data Co-Allocator (CO) [5,8], is fully functional in the DAS system [9]. Below, we describe the components of KOALA and how they work together to achieve co-allocation.

4.1. The KOALA components

The KOALA scheduler consists of the following five components: the CO, the IS, the *Data Manager* (DM), the *Processor Claimer* (PC), and the *Runners*. The structure of KOALA is depicted in Figure 1. The CO is responsible for placing jobs, i.e. for finding the execution sites with enough idle processors for the job components. The CO chooses jobs to place based on their priorities from one of the KOALA placement queues. If the components require input files, the CO also selects the file sites for the components such that the estimated FTTs to the execution sites are minimal. To decide on the execution sites and file sites for the job components, the CO uses one of the placement policies discussed in Section 5.1. Finding execution sites for the job components is only done for non-fixed job requests.

The IS comprises the Globus Toolkit's Metacomputing Directory Service (MDS) [10] and Replica Location Service (RLS) [10], and *Iperf* [11], a tool to measure network bandwidth. A repository containing the bandwidths measured with *Iperf* is maintained and updated periodically.

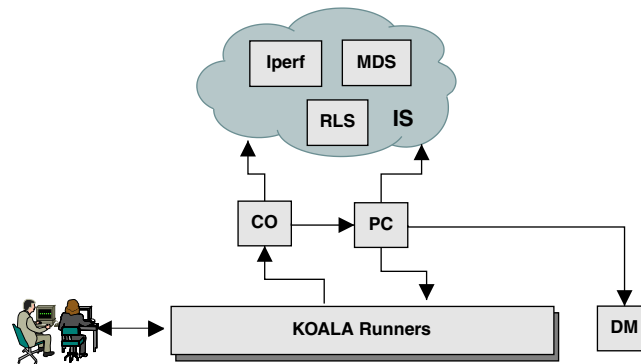


Figure 1. Components of the KOALA scheduler.

The MDS provides on request information about the numbers of processors currently used and the RLS provides the mapping information from the logical names of files to their physical locations. Requests to the MDS and the bandwidth repository impose delays on placing jobs. Therefore, the IS caches information obtained from the MDS and the bandwidth repository with a fixed cache expiry time (a parameter of KOALA). Furthermore, the IS can be configured to do periodic cache updates from frequently used clusters before their cache expiry time.

The DM is used to manage file transfers, for which it uses both Globus GridFTP [12] and Globus Global Access to Secondary Storage (GASS) [10]. The DM is responsible for ensuring that input files arrive at their destinations before the job starts to run. The DM only removes the input files from the execution sites after jobs using them have been completed and they are no longer required by other jobs.

After a job has been placed, it is the task of the PC to ensure that processors will still be available when the job starts to run. If processor reservation is supported by local resource managers, the PC can reserve processors immediately after the placement of the components. Otherwise, the PC uses KOALA's claiming policy to postpone claiming of processors to a time close to the estimated JST; this policy is discussed in detail in Section 6.2.

The Runners are used to submit job components to their respective execution sites; they allow us to extend the support for different application models in KOALA.

4.2. KOALA operational phases

Four operations are performed to any job submitted to KOALA, which include placing its components, transferring its input files, claiming processors for its components and launching and monitoring its execution. The four operations, which form the four phases that the job undergoes, are shown in Figure 2. A failure of the placement procedure in phase 1 results in the job getting appended to one of the KOALA placement queues based on its priority. While in the placement queue, the job's placement is tried for a fixed number of times. Phase 2 is composed of starting the file transfers; while the file transfers are in progress, the job is immediately added to the claiming queue in phase 3. It should be noted that jobs are moved immediately to phase 3 in

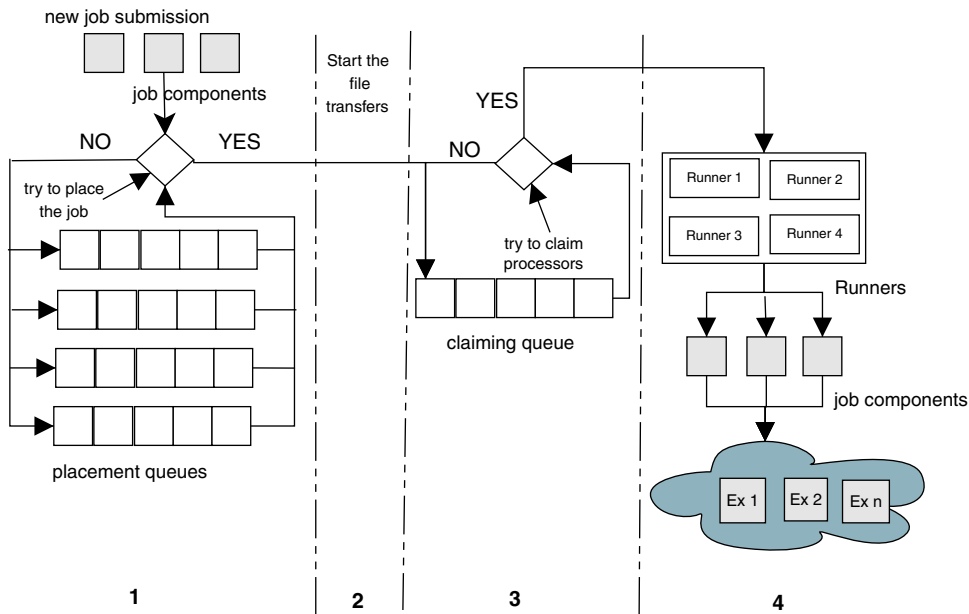


Figure 2. The four phases of job scheduling in KOALA.

case no file transfers are required. In the claiming queue, attempts to claim processors for the job components are made at the designated times. The job is in phase 4 if all of its components have been submitted to their respective execution sites after success of their claiming attempts in phase 3.

4.3. KOALA components interaction

A number of interactions between KOALA components occur in each of the phases mentioned in Section 4.2. Figure 3 shows these interactions as a job moves from one phase to another. The arrows in this figure correspond to the description given below of the interactions happening in each phase.

In phase 1, a new job request arrives at one of the Runners (arrow 1 in Figure 3) in the form of a Job Description File (JDF). We use the Globus Resource Specification Language (RSL) [10] for JDFs, with the RSL '+' construct to aggregate the components' requests into a single multi-request. After authenticating the user, the Runner submits the JDF to the CO (arrow 2), which in turn will append the job to the tail of one of the KOALA placement queues. The CO then retrieves the job from this queue and tries to place the job components based on information (number of idle processors and bandwidth) obtained from the IS (arrow 3). If the job placement fails, the job is returned to its respective placement queue. The placement procedure will be tried for the jobs in the placement queues at fixed intervals for a fixed number of times. The placement queues are discussed further in Section 5.2.

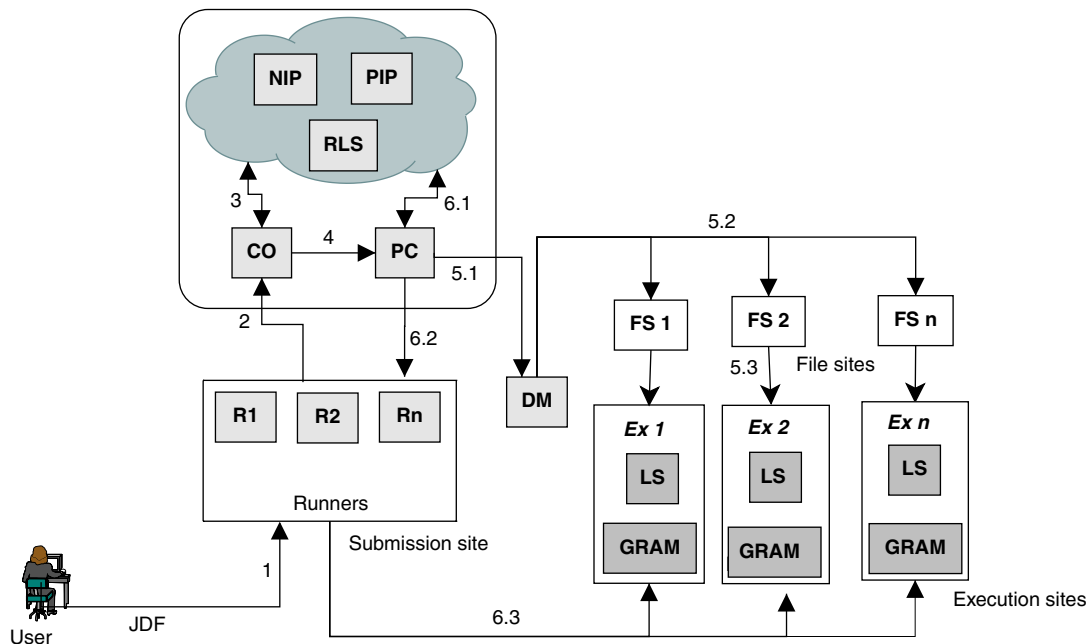


Figure 3. Interaction between the KOALA components. The arrows correspond to the description in Section 4.3.

Phase 2 starts by the CO forwarding the successfully placed job to the PC (arrow 4). On receipt of the job, the PC instructs the DM (arrow 5.1) to initiate the third-party file transfers (arrows 5.2) from the file sites to the execution sites of the job components (arrows 5.3).

In phase 3 the PC estimates the JST and the appropriate time that the processors required by a job can be claimed (Job Claiming Time (JCT)). At this time and if processor reservation is not supported by the local resource managers, the PC uses a claiming policy to determine the components that can be started based on the information from the IS (arrow 6.1). It is possible at the JCT for processors not to be available anymore, e.g. they can then be in use by local jobs. If this occurs, the claiming procedure fails, the job is put into the claiming queue, and the claiming is tried again at a later time.

In phase 4, the Runner used to submit the job for scheduling in phase 1 receives the list of components that can be started (arrow 6.2) and forwards those components to their respective execution sites (arrows 6.3). At the execution sites, the job components are received by the Globus Resource Allocation Manager (GRAM), which is responsible for authenticating the owner of the job and sending the job to the local resource manager for execution. The Runners are discussed in Section 4.4.

4.4. The runners

In computational grids, different application types need different ways to synchronize the start of the job components, different ways to handle communication between the job components, and different



mechanisms for fault tolerance. Examples of these application types include parallel applications, parameter sweep applications and Ibis applications, which are Java programs written for the Java-based grid programming environment called *Ibis* [13,14]. To support different application models, KOALA uses different Runners for different application models. New Runners to support additional application models can be added or old ones can be modified without affecting the operation of other Runners or the scheduler.

Currently, there are three Runners in KOALA, the Default Runner, the DUROC Runner and the Ibis Runner. The Default Runner is used for applications that can handle wide-area communication themselves. The DUROC Runner, which is for parallel Message Passing Interface (MPI) applications, uses Globus's DUROC library [10] to handle the wide-area communication between job components running at different execution sites. The Ibis Runner is a submission tool for Ibis jobs.

4.5. Fault tolerance

The Runners monitor the execution of the job components for any errors that may interrupt their execution. These errors are caused by hardware or software faults of the sites. KOALA responds to these errors by allowing the runners to deal with the failed components. At the same time, KOALA counts the number of consecutive errors of each cluster and if this number reaches a certain threshold, the cluster is marked unusable. A notification message containing the details of the error is then sent to the system administrator of the cluster. In case a Runner does not deal with these errors, the default operation enforced by KOALA simply aborts the whole job, and puts it back to its respective placement queue. When re-placing this job, KOALA uses other available execution sites for the failed job component(s). It should be noted that KOALA only focuses on ensuring that the application is restarted. It is left for the Runner to employ application-specific-fault-tolerance mechanisms like aborting only failed components, checkpointing to ensure that a job continues with the execution and does not start from the beginning.

5. PLACING JOBS

In this section, we present the placement policies used by KOALA to find execution sites as well as file sites for the job components. The placement queues within KOALA that hold jobs that cannot be placed immediately when they are submitted are also discussed.

5.1. The Close-to-Files policy

Placing a non-fixed job in a multicluster allows finding a suitable set of execution sites for all of its components and suitable file sites for the input file. (Different components may receive 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 that site is minimal. The placement algorithm performing this is termed CF policy [5]. It uses the



following parameters for its decisions:

- *The numbers of idle processors in the sites of a grid:* A job component can be placed only on an execution site which will have enough idle processors at the JST.
- *The file size:* The size of the input file, which enters in the estimates of the FTTs.
- *The network bandwidths:* The bandwidth between a file site and an execution site gives the opportunity to estimate the transfer time of a file given its size.

When given a job to place, CF operates as follows (the line numbers mentioned below refer to Algorithm 1). CF first orders the components of a job according to its decreasing size (line 1), and then tries to place the job components in that order (loop starting on line 2). The decreasing order is used to increase the chance of success for large components.

Algorithm 1. Pseudo-code of the Close-to-Files job-placement algorithm.

1. order job components according to decreasing size
 2. **for all** job component j **do**
 3. $S_j \leftarrow$ set of potential execution sites
 4. **if** $S_j \neq \emptyset$ **then**
 5. select an $E \in S_j$
 6. **else**
 7. $P_j \leftarrow$ set of potential pairs of execution site, file site
 8. **if** $P_j \neq \emptyset$ **then**
 9. **for all** $(E, F) \in P_j$ **do**
 10. estimate the file transfer time $T_{(E,F)}$
 11. select the pair $(E, F) \in P_j$ with minimal $T_{(E,F)}$
 12. **for all** file site F' of the job **do**
 13. insert (E, F') into the history table H
 14. **else**
 15. job placement fails
-

For a single job component j , CF first determines the set S_j of *potential execution sites* (line 3); these are the file sites of the job that have enough idle processors to accommodate the job component. If S_j is not empty, CF picks an element from it as the execution site of the component (line 5). (We currently have a function that returns the names of the file sites in alphabetical order, and CF picks the first.)

If the set S_j of potential execution sites is empty (line 6), we might consider all pairs of execution sites with sufficient idle processors and files sites of the job, and try to find the pair with the minimal FTT. This is not efficient in large grids with many sites; therefore, CF maintains a *history table* H with a subset of pairs of execution sites and file sites to consider. From H , CF selects all *potential pairs* (E, F) of execution site, file site, with E having a sufficient number of idle processors for the job component and F being a file site of the job (line 7). If no such pair exists in H , the job component, and hence the whole job, currently cannot be placed (line 15). Otherwise, CF estimates for each selected pair the FTT from the file site to the execution site (line 10), and picks the pair with the lowest estimate (line 11). If (E, F) is the pair selected, CF inserts into H all pairs (E, F')



with F' a file site of the job (lines 12, 13). Note that if the history table is initially empty, it will remain empty. Therefore, it has to be initialized with some set of suitable pairs of execution and file sites.

Built into KOALA is also the WF placement policy. WF places the job components in decreasing order of the number of processors they require on the execution sites with the largest (remaining) number of idle processors. In case the files are replicated, WF selects for each component the replica with the minimum estimated FTT to that component's execution site.

Note that both CF and WF may place multiple job components on the same cluster. We also remark that both CF and WF make perfect sense in the absence of co-allocation, where WF in particular balances the load across the multicluster system.

5.2. The placement queues

A new job arriving to KOALA is appended to the tail of the KOALA placement queue corresponding to its priority. Currently, KOALA maintains four placement queues, one for each of the priorities. These queues hold all jobs that have not yet been placed. KOALA regularly scans the placement queues from head to tail to see whether any job can be placed. To select a queue to scan, KOALA first groups the super-high and high placement queues to form the *higher placement queues*, and the low and super-low placement queues to form the *lower placement queues* as shown in Figure 4. The higher placement queues are scanned first N_h times before scanning the lower placement queues N_l times, with $N_h \geq N_l \geq 1$. In each scan of the higher placement queues, the super-high placement queue is scanned n_1 times before scanning the high placement queue n_2 times, where $n_1 \geq n_2 \geq 1$. This means that after $N_h(n_1 + n_2)$ scans we begin scanning the lower placement queues. Likewise, in each scan of the lower placement queues, the low placement queue is scanned n_3 times before scanning the super-low placement queue n_4 times, where $n_3 \geq n_4 \geq 1$. This also means we scan again the higher placement queues after $N_l(n_3 + n_4)$ scans of the lower placement queues.

The time between successive scans of the placement queues is a fixed interval (this time, N_h , N_l , and the n_i , $i = 1, \dots, 4$, are parameters of KOALA). The time when the placement of a job succeeds is called its JPT, depicted in Figure 5, which shows the timeline of a job submission. The figure also shows the *placement time*, which is the difference between JPT and the time the job enters the placement queue. More of this figure is discussed in Section 6.

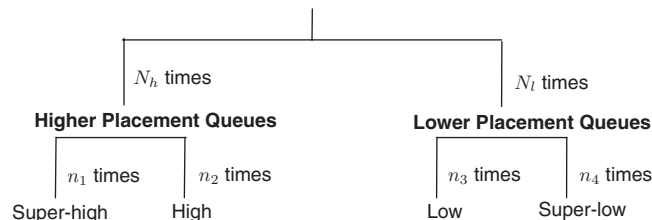


Figure 4. Grouping of priority levels and the number of times KOALA scans the corresponding placement queues.

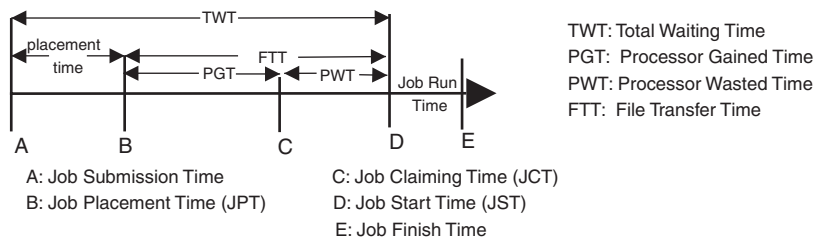


Figure 5. Timeline of a job submission.

For each job in a placement queue 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.

With our current setup, starvation is possible for any job in the KOALA placement queues due to a high load of higher-priority jobs or local jobs. To minimize starvation as much as possible in KOALA, jobs in the low placement queue or in the high placement queue move one priority level up after every P placement tries until they reach the super-high placement queue; P is another KOALA parameter, which can be set to ∞ to prevent jobs from changing their priorities. Jobs in the super-low placement queue are not allowed to change their priority level. It should be noted that the parameters N_h , N_l , n_i , $i = 1, \dots, 4$, and P are assigned static values by the administrator before running the KOALA scheduler. Studying the effects of these parameters as well as dynamically determining their ideal values is left for future work.

6. CLAIMING PROCESSORS

A job is considered started when all the processors requested by all of its components have been claimed. KOALA maintains a claiming queue for the placed jobs that are waiting to be completely started. This section describes the claiming queue and a policy called the ICP that manages claiming of processors for the job components.

6.1. The claiming queue

After the successful placement of a job, its FTT and its JST are estimated before the job is added to the so-called *claiming queue*. The job's FTT is calculated as the maximum of all of its components' estimated FTTs, and the JST is estimated as the sum of its JPT and its FTT (see Figure 5). Our challenge here is to guarantee processor availability at the JST. In the absence of processor reservation (see Section 2.2), the KOALA scheduler can immediately claim processors for a job at its JPT and allow the job to hold the processors until its JST. However, this is a waste of the processor time. It is also possible for KOALA to claim processors only at JST but then there is the risk of processors not being available anymore. Therefore, to minimize the processor wasted time (PWT), which is the time when the processors are held but not used for useful work, and at the same time increase the chance of claiming success, an attempt to claim processors for a job is done



at the job's so-called JCT (point C in Figure 5). A job's JCT is initially set to the sum of its JPT and the product of L and FTT:

$$JCT_0 = JPT + L \cdot FTT$$

where L , which is a real number between 0 and 1, is a parameter assigned to each job by KOALA. The initial value of L assigned to jobs by KOALA is decided by an administrator, e.g. 0.75, and this value is updated dynamically during the claiming attempts. It should be noted that if 0 is assigned to L then the claiming procedure will be attempted at JPT, and if 1 is assigned to L then the claiming procedure will be attempted at JST. More on how L is updated is described below. In the claiming queue, jobs are arranged in increasing order of their JCT.

KOALA tries to claim for a job (*claiming try*) at the current JCT by using our ICP (see Section 6.2). Claiming for a component at the current job claiming try succeeds if all processors it has requested can be claimed, otherwise claiming fails. The success of claiming for all components of the job results in the success of the claiming try. The job is removed from the claiming queue if the claiming try is successful. Otherwise, we perform successive claiming tries. For each such try we recalculate a new 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 5):

$$JCT_{n+1} = JCT_n + L \cdot (JST - JCT_n)$$

If the job's JCT_{n+1} reaches its JST and still claiming for some components fails, the job is returned to the placement queue. Before doing so, its parameter L is decreased by a fixed fraction, e.g. 0.25, and its components that were successfully started in the previous claiming tries are aborted. The parameter L is decreased each time the JST is reached until it hits its lower bound, e.g. 0, so as to increase the chance of success of claiming. If the number of claiming tries for the job exceeds some threshold (which can be set to ∞), the job submission fails.

For a job, a new JST is estimated each time the job is returned to the placement queue and re-placed with a placement policy. We define the *start delay* of a job to be the difference between the final JST where the job execution succeeds and the original JST. It should be noted that the re-placements result in multiple placement times (see Section 5.2). Therefore, we define the *total placement time* of a job as the sum of all its placement times.

We call the time between the JPT of a job and the time of successfully claiming processors for it, the processor gained time (PGT) of the job (see Figure 5). During the PGT, jobs submitted through other schedulers than our grid scheduler can use the processors. The time from the submission of the job until its actual start time is called the total waiting time (TWT) of the job.

6.2. The Incremental Claiming Policy

Claiming processors for a job starts at a job's initial JCT, and if not successful, is repeated at subsequent claiming tries. For components for which claiming has failed, it is possible to increase their chance of claiming success in subsequent claiming tries by finding other sites with enough idle processors to execute them, or by preempting jobs of lower priorities to make processors available.

We call the policy doing exactly this the ICP, which operates as follows (the line numbers mentioned below refer to Algorithm 2). For a job, ICP first determines the sets C_{prev} , C_{now} and



C_{not} of components that have been previously started, of components that can be started now based on the current numbers of idle processors, and of components that cannot be started based on these numbers, respectively. It further calculates F , which is the sum of the fractions of the job components that have previously been started and components that can be started in the current claiming try (line 1). We define T as the required lower bound of F ; the job is returned to the claiming queue if its F is lower than T (line 2).

Algorithm 2. Pseudo-code of the Incremental Claiming Policy

Require: set C_{prev} of previously started components of J (initially $C_{prev} = \emptyset$)
Require: set C_{now} of components of J that can be started now
Require: set C_{not} of components of J that cannot be started now

1. $F \leftarrow (|C_{prev}| + |C_{now}|) / |J|$
2. **if** $F \geq T$ **then**
3. **if** $C_{not} \neq \emptyset$ **then**
4. **for all** $j \in C_{not}$ **do**
5. $(E_j, F_j, ftt_j) \leftarrow Place(j)$
6. **if** $JCT + ftt_j < JST$ **then**
7. $C_{now} \leftarrow C_{now} \cup \{j\}$
8. **else if** $priority(j) \neq \text{super-low}$ **then**
9. $P_j \leftarrow count(processors)$ /* used by jobs of lower priorities than j and idle at E_j */
10. **if** $P_j \geq \text{size of } j$ **then**
11. **repeat**
12. Preempt lower priority jobs at E_j
13. **until** $count(idle\ processors) \geq \text{size of } j$ /* at E_j */
14. $C_{now} \leftarrow C_{now} \cup \{j\}$
15. start components in C_{now}

For each component j that cannot be started on the cluster selected when placing the job, ICP first tries to find a new pair of execution site-file site with the placement policy originally used to place the job (line 5). On success, the new execution site E_j , file site F_j and the new estimated transfer time between them, ftt_j , are returned. If it is possible to transfer the file between these sites before JST (line 6), the component j is moved from the set C_{not} to the set C_{now} (line 7).

For a job of priority other than super-low, if the re-placement of the component fails or the file cannot be transferred before JST (line 8), ICP performs the following. At the execution site E_j of component j , it checks whether the sum of the number of idle processors and the numbers of processors currently being used by jobs of lower priorities is at least equal to the number of processors the component requests (lines 9 and 10). If so, the policy preempts lower priority jobs in descending order of their JST (newest-job-first) until a sufficient number of processors have been freed (lines 11–13). The preempted jobs are then returned to the placement queue.

Finally, those components for which processors can be claimed at this claiming try are started (line 15). Synchronization of the start of the components at the JST depends on the application type and, therefore, it is specific to each Runner. For example, with the DUROC Runner synchronization



is achieved by making each component wait on the job barrier until it hears from all the other components.

When T is set to 1 the claiming process becomes *atomic*, i.e. claiming only succeeds if for all the job components' processors can be claimed.

7. PERFORMANCE RESULTS

In this section we first describe our testbed called the Distributed ASCI Supercomputer (DAS). We then present the results of the experiments we have conducted on the DAS to assess the performance of the KOALA placement policies.

7.1. The Distributed ASCI Supercomputer

The DAS [15] is an experimental computer testbed in the Netherlands that is exclusively used for research on parallel, distributed and grid computing. This research includes work on the efficient execution of parallel applications in wide-area systems [16,17], on communication libraries optimized for wide-area use [13,18], on programming environments [13,18] and on resource management and scheduling [19–21]. The current, second-generation DAS system is a homogeneous multicluster system consisting of five clusters of identical processors (in total 400), which are interconnected by the Dutch University backbone. For local communication within the single clusters, Myrinet LAN is used. Each of the DAS clusters is an autonomous, independent system. Until mid-2005, all the DAS clusters used openPBS [22] as the local resource manager. However, due to reliability problems after the latest upgrade of openPBS, the decision was made to change the local resource manager of the clusters to the Sun N1 Grid Engine [23]. All DAS clusters have their own file system. Therefore, in principle files (including executables) have to be moved explicitly between users' working spaces in different clusters.

7.2. KOALA setup

In the experiments to assess the performance of the placement policies, KOALA is set up as follows. No limits are imposed on the number of job placement and claiming tries to avoid forced job failures when the limits are reached. The interval between successive scans of the placement queues is fixed at 1 min, which as observed from KOALA logs, is a trade-off between too much CPU load due to excessive scans and too long waiting times of jobs in the placement queues. From the KOALA logs, it is also observed that for most jobs, claiming is successful at a value of L between 0.5 and 0.75. Hence, the initial value of the parameter L determining when to start claiming is set at 0.75. As there are only five clusters in our testbed, we initialize the history table H to contain all possible pairs of execution sites and file sites. The parameter T of our claiming algorithm (see Section 6.2) is set to 1; hence, claiming is atomic.

7.3. The test application

In our experiments, we use a parallel application which, because it uses little input data itself, has been modified to request a large input file. The file is only 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 employ co-allocation on the DAS [19]. In the application, the unit square is split up into a two-dimensional pattern of rectangles of equal size among the participating processors. During the execution of 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.

7.4. The workload

We put two workloads of jobs to be co-allocated on the DAS that all run the application of Section 7.3, in addition to the regular workload of the ordinary users. In the workloads, a high priority is assigned to all jobs to give them equal placement opportunities. Both workloads have 200 jobs, with the first workload W_{30} utilizing 30% of the system and the second, W_{50} , utilizing 50% of the DAS. In the workloads jobs have 1, 2 or 4 components requesting the same number of processors, which can be 8 or 16. Each job component requires the same single file of either 2, 4 or 6 Gbyte. For the input files we consider two cases, one without file replication and another with each file replicated in three different sites. The execution time of our test application ranges between 90.0 and 192.0 s. We assume the arrival process of our jobs at the submission site to be Poisson.

7.5. Background load

One of the problems we have to deal with is that we do not have control over the background load imposed on the DAS by other users. These users submit their (non-grid) jobs straight to the local resource managers, bypassing KOALA. During the experiments, we monitor this background load and we try to maintain it at 30–40%. When this utilization falls below 30% in some cluster, we inject dummy jobs just to keep the processors busy. When this utilization rises above 40% and there are dummy jobs, we kill the dummy jobs to lower the utilization to the required range. In case the background load rises above 40% and stays there for more than a minute without any of our dummy jobs, the experiment is declared void and is repeated.

7.6. Results for the workload of 30%

Figure 6 shows the different utilizations in the system for the CF and WF policies for workload W_{30} . During these experiments, the background load due to other DAS users was about 30–40%, yielding a total utilization of about 70%. Because the experiments finished shortly after the submission of the last job and the actual co-allocation load is about 30% throughout the experiments, we conclude that the system is stable with workload W_{30} . It is also shown in this figure that the utilization due to PWT is very low compared with the utilization gained (PGT). This shows that our claiming mechanism (see Section 6.1) works well in a stable system.

Figure 7 shows that the average job FTT with the CF policy is smaller than with the WF policy, both with and without replication. Furthermore, with replication, CF is more successful in finding execution sites ‘closer’ to the file sites, which results in a smaller average job FTT. As a result, the TWT of the jobs is also reduced. The decrease in the average job FTT when the files are replicated

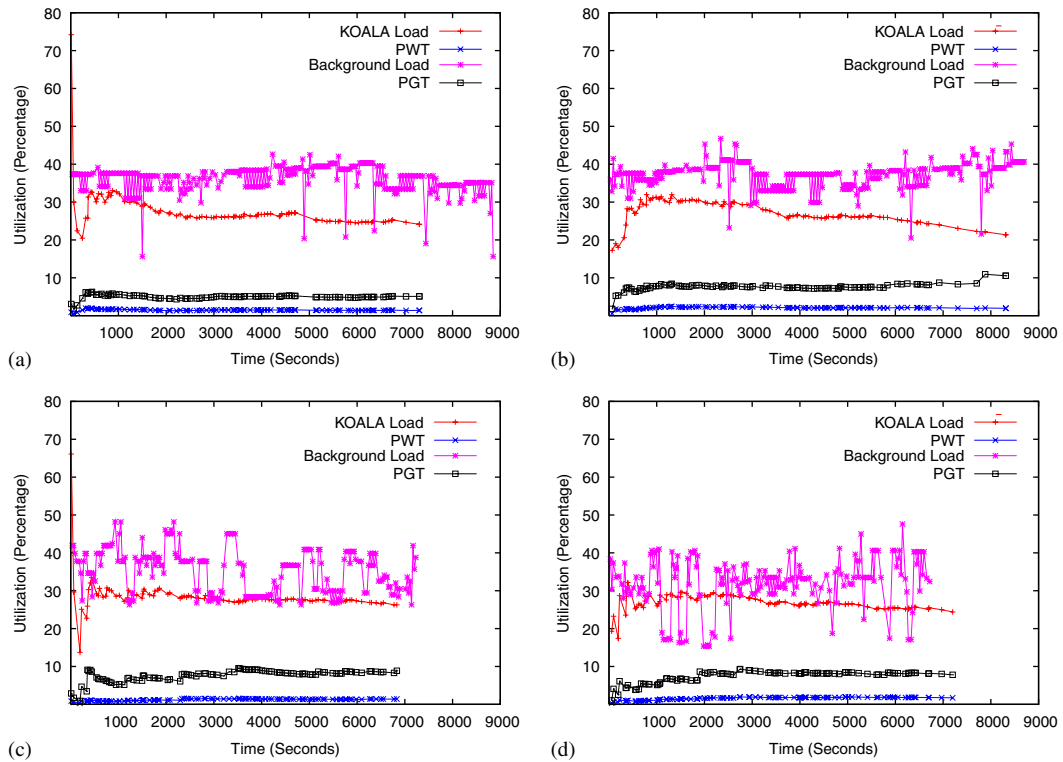


Figure 6. Utilizations for CF and WF with workload W_{30} : (a) CF without replication; (b) WF without replication; (c) CF with replication; and (d) WF with replication.

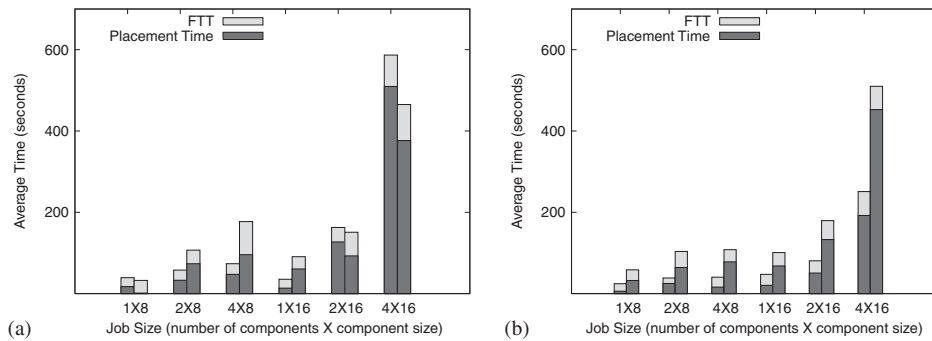


Figure 7. The average file transfer time and placement time for CF (left bars) and WF (right bars) with workload W_{30} : (a) without replication and (b) with replication.



for CF is expected because the number of potential execution sites increases. For both placement policies, the average placement time increases as the number or the size of the job components increases, both with and without replication. The explanation for this is that more time is likely to be spent waiting for clusters to have enough processors available simultaneously.

7.7. Results for the workload of 50%

Figure 8 shows the utilizations of our experiments with workload W_{50} for CF and WF with and without replication. Our main purpose with this workload is to see to at what utilization we can drive the system. From the figure we find that the total utilization during our experiments is 70–80%. However, the actual co-allocation load is well below 40%, the experiments are only finished long after the last job arrival, and the length of the placement queue goes up to 30, which shows that the system is saturated. Hence, we conclude that we can drive the total utilization not higher than what we have achieved here.

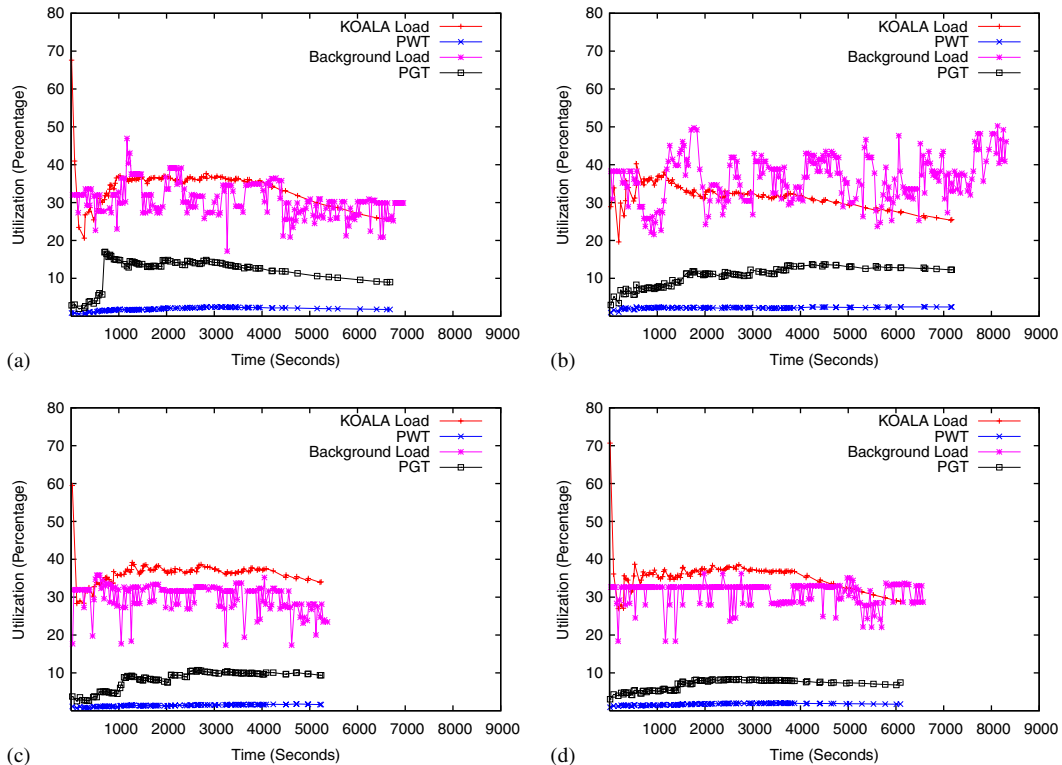


Figure 8. Utilizations for CF and WF with workload W_{50} : (a) CF without replication; (b) WF without replication; (c) CF with replication; and (d) WF with replication.

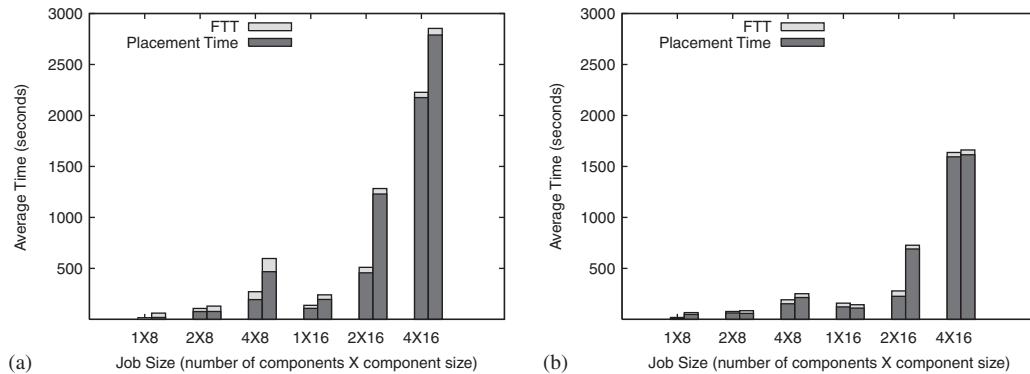


Figure 9. The average file transfer time and placement time for CF (left bars) and WF (right bars) with workload W_{50} : (a) without replication and (b) with replication.

With this workload and with the CF policy, clusters ‘close’ to files will often be occupied, forcing more long file transfers. As a result, the average FTT for CF and WF are relatively close to each other both with and without replication as shown in Figure 9 (note the different scale from Figure 7). Since the system is saturated with this workload, more time is spent waiting for clusters to have enough processors available simultaneously. This explains the increase in the average placement times.

7.8. Assessing the level of co-allocation used

Our placement policies use co-allocation to achieve their primary goals: to minimize the FTTs (CF), and to balance the numbers of idle processors (WF). Recall that both CF and WF are allowed to place different components of the same job on the same cluster. In order to assess the level of co-allocation actually used by a policy, we introduce a metric called the *job spread*, which for a job is defined as the ratio of the number of its execution sites and the number of its components (we will express it as a percentage).

We have carried out experiments to study the average job spread and also the percentage of jobs that actually use co-allocation. For these experiments, we have created a new workload that utilizes 30% of the system and that consists of jobs with two or four components of equal sizes (number of processors), which can be 4, 8, 16 or 24. In this workload, each job component requires the same single file of size 2 Gbyte. Again, we conducted two experiments, one without file replication and the other with files replicated in three different sites. The results of these experiments are shown in Figure 10.

We first observe that with replication, CF places at least half of the components of jobs on separate clusters, which is expected with CF because the number of potential execution sites increases with replication. Without replication, the average job spread of CF is slightly less compared with CF with replication because of the decrease in the number of potential execution sites. Second, the average job spread of WF is not affected by file replication, which can be explained by the fact that finding execution sites with WF depends on the size and the distribution across the sites of

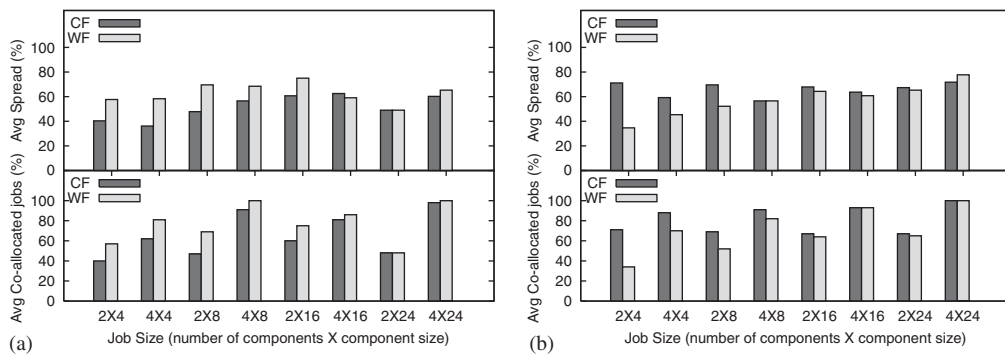


Figure 10. The average job spread (upper graphs) and the percentages of jobs that use co-allocation (lower graphs): (a) without replication and (b) with replication.

the background load. As a result of these two observations and with our background load, CF with replication uses co-allocation more compared with WF, while without replication, it is the opposite (Figure 10). Lastly, for each job size, the percentage of jobs using co-allocation increases as the number of job components increases.

8. EXPERIMENTS IN AN UNRELIABLE TESTBED

In this section we describe the experiments we have conducted to assess our co-allocation service in an unreliable environment. The experiments were done on the DAS system immediately after a major upgrade of the operating system and the local resource manager (openPBS). Even though the system is homogeneous and centrally managed, it was very unstable, and hence unreliable during the experiments. This gave us the opportunity to evaluate co-allocation with KOALA in 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 mechanisms.

8.1. KOALA setup

In these experiments, again we did not impose limits on the number of placement and claiming tries. Similar to the experiments in Section 7, the parameter L is at 0.75 but the intervals between successive scans of the placement queues is increased to 4 min in order to decrease the number or excessive scans due to the unreliability of the testbed. The parameter T of our claiming algorithm (see Section 6.2) is set to 0; hence, we claim processors for any number components we can. Only jobs of high priority and low priority are used in these experiments and we do not allow them to change their priorities by setting parameter P to ∞ (see Section 5.2). The parameters N_h and N_l are set to 1, the parameters n_1 and n_2 are set to 1 and 2, respectively, and the parameters n_3 and n_4 are set to 1. This setup allows us to scan the high placement queue twice before we scan the low placement queue once (see Section 5.2).



8.2. The workload

In these experiments, we put a workload of 500 jobs to be co-allocated on the DAS that all run the application of Section 7.3, in addition to the regular workload of the ordinary users. In our experiments, we consider job sizes of 36 and 72, and four numbers of components, which are 3, 4, 6 and 8. The components request the same number of processors, 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 component sizes, while those of size 36 have three or four 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. We assume the arrival process of our jobs at the submission site to be Poisson.

8.3. Utilization

At the start of the experiment, a total of 310 processors in four out of five clusters were available to KOALA for placing jobs. During the experiment, one of the clusters reported a much higher consecutive number of failures and was taken out of selection by KOALA (see Section 4.5). As a result, the number of processors available for selection was reduced to 250. The utilization of these processors by jobs due to other DAS users and to KOALA is shown in Figure 11. 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.

8.4. Failures

In this paper, by a failure of a job we mean that one of the clusters' local resource managers has failed to execute its assigned job component. The failures of the resource managers were due to bugs in them and to the incorrect configuration of some of the nodes. The failure of any of the

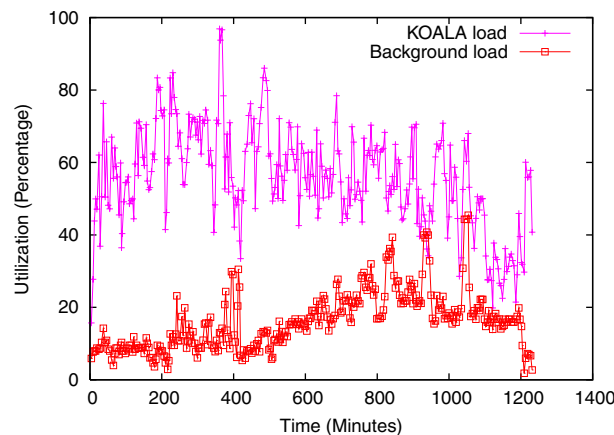


Figure 11. System utilization during the experiment on an unreliable testbed.

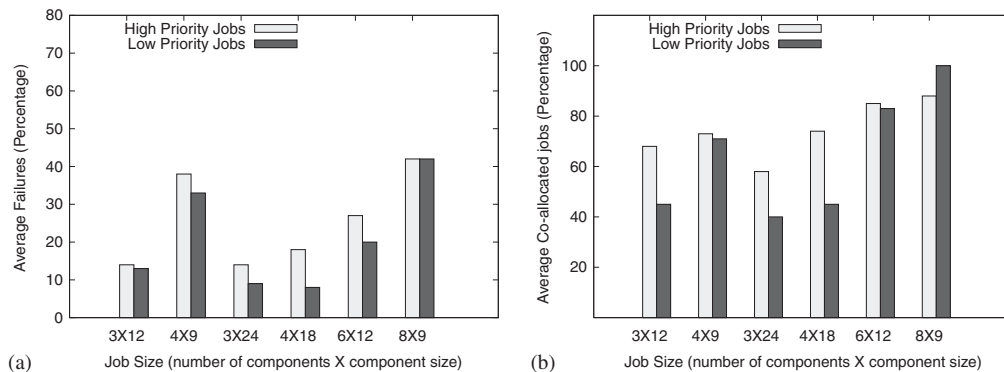


Figure 12. Percentages of failures of jobs of different sizes, and of jobs that use co-allocation: (a) percentages of failures of jobs and (b) percentages of jobs that use co-allocation.

components of a job causes the whole job to fail and, as a result, to be returned to the placement queue. Figure 12(a) shows the percentage of failures for each of the job sizes, as a percentage over all successful placements. By a successful placement we mean a placement that resulted in the job being submitted for execution. Since in the end, all jobs ran successfully, the number of failures of a job is equal to the number of successful placements minus one. The number of failures is much higher compared with the stable system, where 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, and therefore to run these jobs. As a result, more jobs are started simultaneously, which results in some components being given mis-configured nodes because most of the time these nodes are idle. The number of failures also increases with the increase in the number of components, because then the chance for components to be placed on multiple clusters (co-allocated, see Figure 12(b)) and hence the chance of failures increases. In Figure 12(b) we show the percentage of jobs that actually were co-allocated, i.e. of jobs whose components were placed on multiple clusters.

8.5. Placement times and start delays

Failed jobs are returned to their respective placement queues in KOALA, which then tries to re-place these jobs until their execution succeeds. As a result of re-placements, the total placement times of jobs as defined in Section 6.1 increase considerably with the increase in the number of components as shown in Figure 13(a). We do emphasize, however, that all our jobs eventually ran to completion successfully.

Jobs of small sizes do not suffer from many re-placements or a long waiting time for enough idle processors to be available. Yet these jobs still require co-allocation as shown in Figure 12(b), which helps to lower their total placement times (Figure 13(a)).

Figure 13(b) shows the start delays of jobs of different sizes, which is also affected by the number of failures and re-placements. Like the above observations, the start delay increases with the number of components, with high-priority jobs performing better than low-priority jobs.

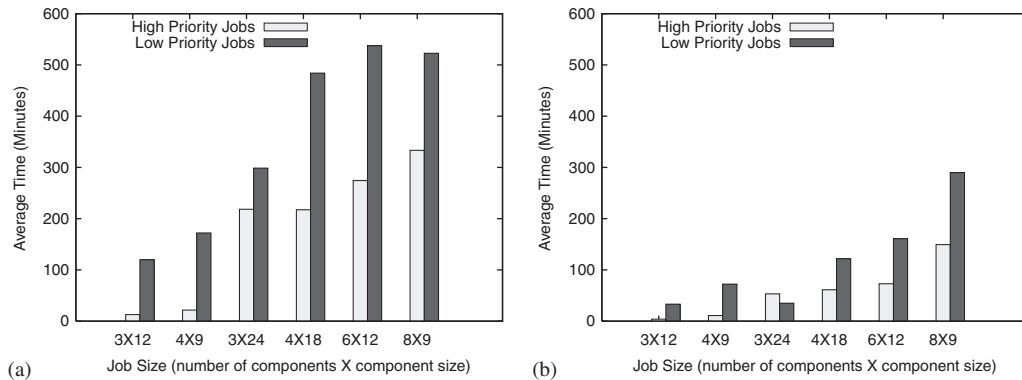


Figure 13. The placement times (a) and start delays (b) of jobs.

Overall, splitting jobs into many components does not necessarily lead to smaller total placement times. On the other hand, small jobs with few components still require co-allocation to guarantee smaller total placement times and start delays. Nevertheless, we cannot conclude that jobs of smaller sizes perform much better but rather we can conclude that our co-allocation service cannot avoid delaying considerably jobs requesting many processors in an unreliable testbed. Finally, despite an unreliable system, we find that still jobs of high-priority out-perform jobs of low priority.

9. RELATED WORK

In our previous work [19–21] we have studied with simulations processor co-allocation in multiclusters with space sharing of rigid jobs for a wide range of such parameters as the number and sizes of the job components, the number of clusters, the service-time distribution, and the number of queues in the system. Our main results were that co-allocation is beneficial as long as the number and sizes of job components (i.e. the parts of a job running at multiple sites), and the slowdown of applications due to the wide-area communication are limited. The impact of wide-area communication is also studied with simulations by Ernemann *et al.* [24,25]. In their work, co-allocation is compared with keeping jobs local and to only sharing load among the clusters. One of the most important findings is that when the application slowdown does not exceed 1.25, it pays to use co-allocation. The scheme Synchronous Queuing for co-allocation that does not require advance reservations and that is also studied with simulations [26] ensures that the subtasks of a job remain synchronized by minimizing the co-allocation skew, which is the time difference between the fastest running and the slowest running subtasks of an application. In KOALA, only the component (subtask) start times are synchronized, and further synchronization during the runtime, if required, is left to the application.

The Globus Architecture for Reservation and Allocation (GARA) [27] enables the construction of application-level co-reservation and co-allocation libraries that are used to dynamically assemble a collection of resources for an application. In GARA, the problem of co-allocation is simplified by the use of advance reservations. Support for co-allocation through the use of advance reservations is also included in the grid resource broker presented in [28].



Czajkowski *et al.* [1] propose a layered co-allocation architecture that addresses the challenges of grid environments by providing basic co-allocation mechanisms for the dynamic management of separately administered and controlled resources. These mechanisms are implemented in a CO called the Dynamically Updated Resource Online Co-allocator (DUROC), which is part of the Globus project [10]. DUROC implements co-allocation specifically for the grid-enabled implementation of MPI applications. However, DUROC, which is implemented as a set of libraries to be linked with application codes and job submission tools, does not provide resource brokering or fault tolerance, and requires jobs to specify exactly where their components should run.

Nimrod-G [29] is an economy-driven grid resource broker that supports soft-deadline and budget-based scheduling of applications on the grid. Like KOALA, Nimrod-G performs resource discovery, scheduling, dispatching jobs to remote grid nodes, starting and managing job execution, and gathering results back to the submission site. However, Nimrod-G uses user-defined deadline and budget constraints to make and optimize its scheduling decisions, and focuses only on parameter sweep applications.

Venugopal *et al.* [30] present a scheduling strategy that has been implemented within the Gridbus broker [31]. For each job, their algorithm first selects the file site that contains the file required for the job, and then selects a compute resource that has the highest available bandwidth to that file site. This work focuses on data-intensive applications that do not require co-allocation.

The GridWay framework [32], which allows the execution of jobs in dynamic grid environments, incorporates similar job scheduling steps as KOALA does, such as resource discovery and selection, job submission, job monitoring and termination, but then at the application level. Two drawbacks of GridWay are that the application source code first has to be instrumented, and that the scheduling process is not aware of other jobs currently being scheduled, rescheduled or submitted, with as a consequence that the throughput of the grid is degraded because of the failure to balance the usage of grid resources.

The Grid Application Development Software (GrADS) [33] enables co-allocation of grid resources for parallel applications that may have significant inter-process communication. For a given application, during resource selection, GrADS first tries to reduce the number of workstations to be considered according to their availabilities, local computational and memory capacities, network bandwidth and latency information. Then, the scheduling solution that gives the minimum estimated execution time is chosen for the application. Like GridWay, GrADS performs only application-level scheduling.

10. CONCLUSIONS

In this paper, we have addressed the problem of scheduling jobs consisting of multiple components that require both processor and data co-allocation in multicluster systems, and in grids in general. We have developed KOALA, a co-allocating grid scheduler that implements placement and claiming policies for multi-component jobs. Fault-tolerance mechanisms that allow KOALA to cope with unstable environments have also been presented. The results of the experiments performed with KOALA show that the combination of the CF placement policy and file replication is very beneficial, and give evidence of the correct and reliable operation of KOALA in an unreliable testbed. In the



absence of processor reservations, our Incremental Claiming Policy can be used without wasting much processor time.

As future work, we are intending to remove the restriction 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. In addition, a performance study of KOALA in a heterogeneous grid environment is planned.

REFERENCES

1. Czajkowski K, Foster IT, Kesselman C. Resource co-allocation in computational grids. *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*. IEEE Computer Society Press: Silver Spring, MD, 1999; 219–228.
2. Web-site. The Portable Batch System. <http://www.pbspro.com/> [July 2007].
3. Web-site. Maui Scheduler. <http://supercluster.org/maui/> [July 2007].
4. Ananad S, Yoginath S, von Laszewski G, Alunkal B. Flow-based multistage co-allocation service. *Proceedings of the International Conference on Communications in Computing*. CSREA Press: Las Vegas, 2003; 24–30.
5. Mohamed HH, Epema DHJ. An evaluation of the close-to-files processor and data co-allocation policy in multiclusters. *Proceedings of IEEE International Conference on Cluster Computing 2004*. IEEE Computer Society Press: Silver Spring, MD, 2004; 287–298.
6. Mohamed HH, Epema DJH. Experiences with the KOALA co-allocating scheduler in multiclusters. *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2005)*, Cardiff, Wales, 2005; 784–791.
7. Park S, Kim J. Chameleon: A resource scheduler in a data grid environment. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2003)*, Tokyo, Japan, 2003; 256–263.
8. Mohamed HH, Epema DJH. The design and implementation of the KOALA co-allocating grid scheduler. *European Grid Conference (Lecture Notes in Computer Science, vol. 3470)*. Springer: Berlin, 2005; 640–650.
9. Web-site. KOALA Co-Allocating Grid Scheduler. <http://www.st.ewi.tudelft.nl/koala> [July 2007].
10. Web-site. The Globus Toolkit. <http://www.globus.org/> [July 2007].
11. Web-site. Iperf Version 1.7.0. <http://dast.nlanr.net/Projects/Iperf/> [July 2007].
12. Allcock W, Bresnahan J, Foster I, Liming L, Link J, Plaszczac J. GridFTP Update. *Technical Report*, January 2002.
13. van Nieuwpoort RV, Maassen J, Hofman R, Kielmann T, Bal HE. Ibis: An efficient Java-based grid programming environment. *ACM JavaGrande ISCOPE 2002 Conference*, Seattle, U.S.A., November 2002; 18–27.
14. van Nieuwpoort RV, Maassen J, Bal HE, Kielmann T, Veldema R. Wide-area parallel programming using the remote method invocation model. *Concurrency: Practice and Experience* 2000; **12**(8):643–666.
15. Web-site. The Distributed ASCI Supercomputer (DAS). <http://www.cs.vu.nl/das2> [July 2007].
16. Weissman JB, Grimshaw AS. A federated model for scheduling in wide-area systems. *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing (HPDC-5)*, New York, U.S.A. IEEE Computer Society Press: Silver Springs, MD, 1996; 542–550.
17. Weissman JB, Zhao X. Scheduling parallel applications in distributed networks. *Cluster Computing* 1998; **1**:109–118.
18. Web-site. Mpich-g2. <http://www3.niu.edu/mpi/> [July 2007].
19. Banen S, Bucur AID, Epema DHJ. A measurement-based simulation study of processor co-allocation in multiclusters. Feitelson DG, Rudolph L, Schwiegelshohn U (eds.). *Ninth Workshop on Job Scheduling Strategies for Parallel Processing (Lecture Notes in Computer Science, vol. 2862)*. Springer: Berlin, 2003; 105–128.
20. Bucur AID, Epema DHJ. The performance of processor co-allocation in multiclusters. *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2003)*. IEEE Computer Society Press: Silver Spring, MD, 2003; 302–309.
21. Bucur AID, Epema DHJ. Trace-based simulations of processor co-allocation policies in multiclusters. *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12)*. IEEE Computer Society Press: Silver Spring, MD, 2003; 70–79.
22. Web-site. The Portable Batch System. www.openpbs.org [July 2007].
23. Web-site. The Sun Grid Engine (SGE). <http://gridengine.sunsource.net/> [July 2007].
24. Ernemann C, Hamscher V, Schwiegelshohn U, Yahyapour R, Streit A. On advantages of grid computing for parallel job scheduling. *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the GRID (CCGrid2002)*. IEEE Computer Society Press: Silver Spring, MD, 2002; 39–46.
25. Ernemann C, Hamscher V, Streit A, Yahyapour R. Enhanced algorithms for multi-site scheduling. *Third International Workshop on Grid Computing*, Baltimore, U.S.A., 2002; 219–231.



26. Azzedin F, Maheswaran M, Arnason N. A synchronous co-allocation mechanism for grid computing systems. *Cluster Computing* 2004; **7**:39–49.
27. Foster I, Kesselman C, Lee C, Lindell B, Nahrstedt K, Roy A. A distributed resource management architecture that supports advance reservation and co-allocation. *Proceedings of the 7th International Workshop on Quality of Service*, Montreal, Canada, May–June 1999.
28. Elmroth E, Tordsson J. A grid resource broker supporting advance reservations and benchmark-based resource selection. *Proceedings of 7th International Workshop of Applied Parallel Computing, State of the Art in Scientific Computing*, Lyngby, Denmark, June 2004; 1061–1070.
29. Abramson D, Buyya R, Giddy J. A computational economy for grid computing and its implementation in the Nimrod-G resource broker. *Future Generation Computer Systems* 2002; **18**(8):1061–1074.
30. Venugopal S, Buyya R, Winton L. A grid service broker for scheduling e-Science applications on global data grids: Research articles. *Concurrency and Computation: Practice and Experience* 2006; **18**(6):685–699.
31. Web-site. Grid service broker: A grid scheduler for computational and data grids. <http://www.gridbus.org/broker/> [July 2007].
32. Huedo E, Montero RS, Llorente IM. A framework for adaptive execution in grids. *Software—Practice and Experience* 2004; **34**:631–651.
33. Dail H, Berman F, Casanova H. A decoupled scheduling approach for grid application development environments. *Journal of Parallel and Distributed Computing* 2003; **63**(5):505–524.