A Dynamic Co-Allocation Service in Multicluster Systems

J.M.P. Sinaga, 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

Web address: `www.pds.ewi.tudelft.nl/~epema`

E-mail: `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 in multiple clusters to improve their performance. In previous work, we have studied processor co-allocation through simulations. Here, we extend this work with the design and implementation of a dynamic processor co-allocation service in multicluster systems. While an implementation of basic co-allocation mechanisms has existed for some years in the form of the DUROC component of the Globus Toolkit, DUROC does not provide resource-brokering functionality or fault tolerance in the face of job submission or completion failures. Our design adds these two elements in the form of a software layer on top of DUROC. We have performed experiments that show that our co-allocation service works reliably.

1 Introduction

Computer systems consisting of multiple clusters, and more generally grids, offer the promise of transparent access to large collections of resources for very demanding applications. In fact, the needs of a single application may exceed the capacity available in any subsystem making up such a system, and so *co-allocation*, i.e., the simultaneous access to resources of possibly multiple types (processors, data, network bandwidth) in multiple locations, managed by different resource managers [8], may be required. Then, the jobs executing such applications consist of multiple components, with each component using resources in a different subsystem. When co-allocation is not used in multiclusters and grids, such systems only act as large load-balancing devices with higher-level schedulers trying to find good single locations for jobs to run. The real challenge of using such systems is in trying to achieve good mechanisms and policies for co-allocation.

Among the simplest types of applications that need co-allocation are parallel applications that require the simultaneous allocation of processors managed by different schedulers. The feasibility of running such applications in multicluster systems with their relatively slow wide-area connections has been demonstrated for instance in [4]. One of the main problems of processor co-allocation is to achieve the simultaneous availability of the processors managed by different local schedulers. The basic mechanisms for processor co-allocation have existed for a number of years in the form of the Dynamically Updated Request Online Co-allocator (DUROC) [8] component of the Globus Toolkit [2]. However, even though DUROC serves its basic purpose of submitting multicomponent jobs, it cannot be regarded as a co-allocating scheduler for general use in multiclusters or in grids. First, it lacks resource allocation policies by requiring jobs to specify exactly the sites where their components should run. Secondly, DUROC does not provide good fault tolerance in that it may wait for enough available resources for an unspecified amount of time, and in that it requires the user's intervention when the submission or completion of a job fails.

In previous work we have studied processor co-allocation by means of simulations [4, 6, 7]. In this paper, we extend this work by the design and implementation of a prototype called the Dynamic Co-Allocation Service (DCS) on our wide-area Distributed ASCI[1] Supercomputer (DAS, see Section 4.1) that implements mechanisms and policies for processor co-allocation in multicluster systems. Our design is built on top of DUROC and consists of a Scheduler that implements policies such as FCFS and a form of backfilling, a Resource Monitor that reports on the avail-

---

[1] ASCI is the acronym of the Advanced School for Computing and Imaging in the Netherlands.

able resources, a Resource Broker that maps jobs onto the most suitable clusters, and a Co-allocator that submits jobs to DUROC. In particular, our DCS solves the two problems with DUROC mentioned above. The results of the experiments with our prototype show that it works reliably.

2 The Problem of Processor Co-allocation

In this section we formulate the problem of co-allocation in multiclusters, we discuss the DUROC component of Globus, and we describe the structure of multicomponent jobs and the placement and scheduling policies used in our design.

2.1 Processor Co-allocation

In itself, processor co-allocation is a simple notion: Assign processors in different systems in a multicluster or a grid to single jobs simultaneously. The potential advantage to a job is that it may employ more processors than available in a single cluster and so may experience a shorter runtime, and the potential advantage to the system is that the system load may be increased. Of course, due to the relatively slow wide-area communications, not all applicatons will benefit from using processors in clusters connected by a wide-area network, but some definitely will [4].

An important issue in processor co-allocation is that the processors in the different clusters have to be available at the same time. What would be very helpful to guarantee the simultaneous allocation of processors is a reservation mechanism of the local resource managers. However, hardly any of the popular local resource managers such as PBS [3] supports such a mechanism. A processor co-allocation mechanism built on top of resource managers without reservation capabilities cannot do anything else than repeatedly try to assess whether sufficient numbers of processors are available, and then claim these.

2.2 Motivation

Currently, the only available standard tool for processor co-allocation in grids is the DUROC component of the Globus Toolkit. In general, Globus can accept job descriptions written in the Globus Resource Specification Language (RSL), in which such things as the name of the executable and the input and output files can be specified. When a job consists of a single component, one of the things that also has to be specified is the name of the resource manager of the system where the job has to run. After having parsed the RSL specification of a job, Globus sends the job to the Globus Resource Allocation Manager (GRAM) running on the specified resource, which in turn hands over the job to the local resource manager. DUROC [8] is the

component of the Globus Toolkit that contains the basic mechanisms for co-allocation. It accepts what are called multirequests, which consist of multiple simple requests, each written in RSL and each specifying a component of a multicomponent job. DUROC supports two approaches for co-allocating resources to jobs. In the atomic trannsaction approach, all resources specified by a job have to be available otherwise the job's submission fails. In the interactive transaction approach, some resources may be specified as nonessential or optional, in order to tolerate resource failures.

DUROC cannot be regarded as a full-blown co-allocating scheduler in multiclusters or grids, as it lacks certain functionality. First, it does not do any resource brokering by picking suitable resources for a job, and the RSL specification of a job request must be complete in that the subsystems (clusters) to be used by a job must be exactly specified in advance. In other words, DUROC implements what we call a *static* co-allocation mechanism, and it can only deal with what we call ordered jobs (see Section 2.3). Secondly, a job submission may fail because the resources required by the job are not immediately available, or because a job may not complete successfully. When the first happens, DUROC cannot do anything except sending an error message to the user telling that the submission has failed or just waiting until sufficient resources do become available. In the latter case, there is no time-out mechanism for removing jobs that are waiting too long. When the second happens, the user simply has to resubmit the job.

This situation has motivated us to design our Dynamic Co-Allocation Service (DCS) for multicluster systems on top of DUROC. The DCS detects the states of the clusters and *dynamically* allocates resources according to those states. It gives users a more flexible way of specifying multicomponent jobs in that they do not have to tell in advance the locations where the the jobs' components have to run. In addition, the DCS will repeatedly submit a job that experiences submission failures (for a maximum number of times), and it will repeatedly resubmit a job that experiences completion failures (again for a maximum number of times).

2.3 The Structure of the System and of Job Requests

Our model of multicluster systems is very simple: We assume a system of say C clusters consisting of possibly different numbers of identical processors.

Jobs submitted to our DCS that consist of multiple components and require co-allocation, have to specify the number and the sizes of their components, i.e., of the numbers of processors needed in the separate clusters. We assume jobs to be rigid, which means that they do not change size

over their lifetime. So a job is represented by a tuple of C values (some of which may be zero), indicating its component sizes. We will consider two cases for the structure of job requests:

1. In an *ordered request* the positions of the request components in the tuple specify the clusters from which the processors must be allocated.

2. For an *unordered request*, by the components of the tuple the job only specifies the numbers of processors it needs in the separate clusters, allowing the scheduler to choose the clusters for the components. Here, we do allow different components of unordered jobs to go to the same cluster.

Ordered requests are used in practice when a user has enough information about the complete system to take full advantage of the characteristics of the different clusters. Unordered requests model applications like FFT, which needs few data, and in which tasks in the same job component share data and need intensive communication, while tasks from different components exchange little or no information.

The RSL descriptions of unordered jobs submitted to the system are incomplete in that the locations where the components should run have not been filled out; the RSL descriptions of the ordered jobs submitted are complete.

2.4   The Placement and Scheduling Policies

For ordered requests it is clear when a job fits on the system or not, given the current numbers of idle processors. To determine whether an unordered request fits, we use the Worst-Fit (WF) placement policy avoiding as much as possible reusing the same clusters. When placing a job, we first order the job components according to decreasing size, and then assign the job components in that order. When assigning a job, we keep two lists of clusters, both ordered according to decreasing numbers of idle processors. The first is the list N of clusters that do not yet have a job component assigned (initially all clusters), and the second is the (initially empty) list Y of clusters that already have at least one job component assigned to them. For every job component, if it fits on the cluster at the head of list N, it is assigned to that cluster and that cluster is removed from N and inserted into the appropriate place into list Y. If it does not fit on the cluster at the head of list N, the job component is assigned to the cluster at the head of list Y if it fits there, and then that list is reordered if necessary. If the component also does not fit on the cluster at the head of list Y, then the whole job cannot be placed. Our motivation for using WF is that it balances the load, leaving roughly equal numbers of idle processors in all clusters. However, WF can easily

be replaced by any other placement policy that better suits a multicluster's objectives.

As we will see in Section 3, the DCS maintains a single global queue. As the scheduling policy we use First Come First Served (FCFS) or Fit Processors First Served (FPFS). In FPFS, when the job at the head of the queue does not fit, the queue is scanned from head to tail for any jobs that may fit. FPFS may cause starvation, which may for instance be repaired by putting a maximum to the number of times a job can be overtaken by other jobs, but we have not implemented this. FPFS is a variation of backfilling [16], for which it is usually assumed that (estimates of) the service times are available before jobs start, and in which the job at the head of the queue may not be delayed by jobs overtaking it. However, we assume that we do not have runtime estimates, and so we cannot implement this type of backfilling.

3   The Design of the Dynamic Co-allocation Service

In this section we will present the design of our Dynamic Co-allocation Service (DCS). We will first give an overview of the architecture of the DCS and the flow of a multicomponent job through it. Then, we will focus on each component of the DCS in more detail.

3.1   An Overview of the DCS

The basic idea underlying our design is to employ DUROC and to add several higher-level components to it to implement fault-tolerant dynamic co-allocation. The software components of the DCS are the Scheduler, the Resource Broker, the Resource Monitor, and the Co-allocator. In addition, we maintain as data structures a wait queue and a run list. Figure 1 depicts all of these components.

When a user submits a job to the system, the Scheduler appends it to the tail of the *wait queue*, which contains all jobs submitted but not yet allocated. As long as the wait queue is not empty, the Scheduler tries to schedule jobs from it by contacting the Resource Broker.

When the Resource Broker receives a job request from the Scheduler, it attempts to fit the job on the system taking into account the job type and the available resources. If there are sufficient resources, the Resource Broker decides on an allocation and sends the job request back to the Scheduler; otherwise it sends a failure message back to the Scheduler. In order to fit a job to the resources, the Resource Broker needs to know about the current resource status, which it gets from the Resource Monitor.

When the Scheduler gets a failure message from the Resource Broker, it will simply keep the job in its current location in the wait queue, and it will later attempt to resched-
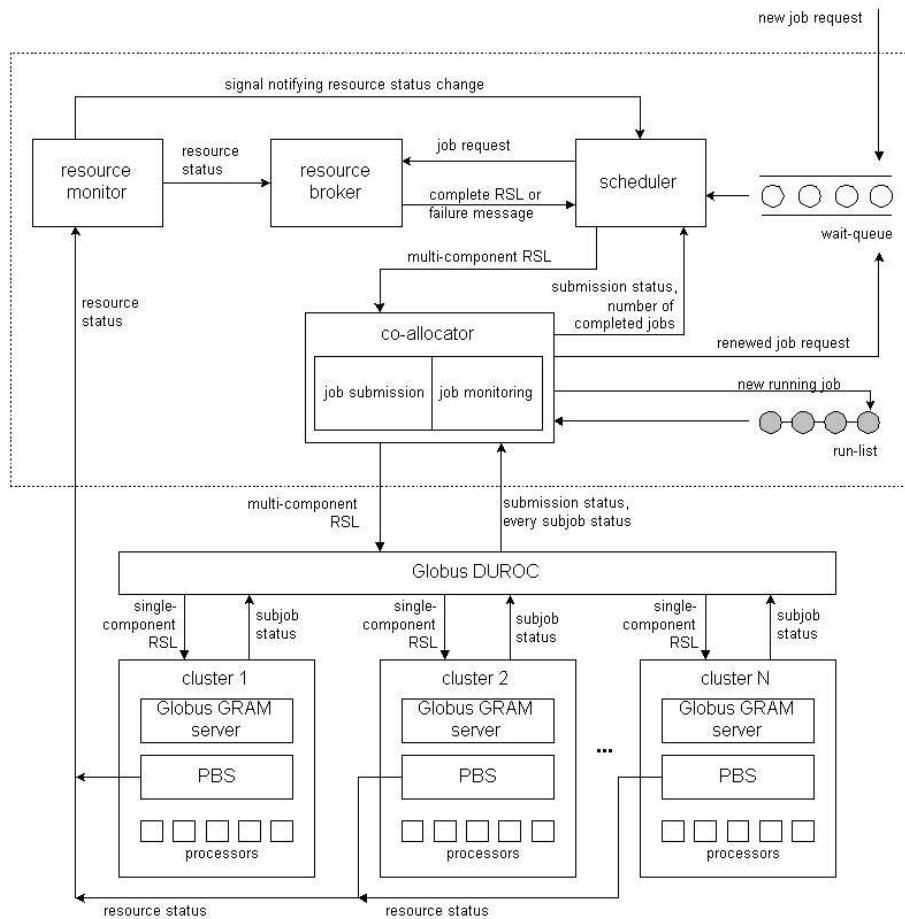
Figure 1. The architecture of the Dynamic Co-allocation Service.

ule it. When the Scheduler gets a completed RSL file from the Resource Broker, it will send the job request to the Co-allocator, which in turn will forward it to DUROC. DUROC will use its co-allocation mechanism to submit all subjobs to their destination clusters.

The success or failure of a job submission to DUROC is noted by the Co-allocator in a so-called submission status, which it returns to the Scheduler. If the job submission is successful, the Scheduler removes the corresponding element from the wait queue, and the Co-allocator puts the job request into the *run list*, which contains a record of all currently running jobs so that the Co-allocator can monitor their progress.

However, even if the Resource Broker has found a suitable allocation for a job request, it is possible that the job submission fails. For instance, there may have been a change in the resource availability while the Resource Broker is working to fit the job on the system so that the allocated resources are not available anymore, the executable file cannot be found, etc. If this happens, the Co-allocator

will cancel the job submission and tell the Scheduler about the failure. The job request is then moved to the tail of the wait queue.

The Co-allocator also keeps track of the completion status of jobs, which indicates whether or not a runnning job has completed its execution successfully. If a running job fails to complete its execution, the Co-allocator will put the job request back at the tail of the wait queue so that it can later be rescheduled. When a running job finishes successfully, the Co-allocator removes the job from the run list, and sends a message to the Scheduler that contains the number of jobs that have successfully finished so far.

Now, we will see in more detail how each main component is designed.

3.2 The Scheduler and the Wait Queue

The Scheduler is the central component in our design. It manages the wait queue as the place for the requests of jobs that have not yet been allocated, it gets allocations of jobs

from the Resource Broker, and it calls the Co-allocator to actually submit jobs to DUROC.

An element of the wait queue includes the following fields:

- The type of the job request (ordered or unordered).

- The text of the original job request in RSL that is still incomplete in case of unordered jobs.

- The number of times the job has suffered a submission failure.

- The number of times the job has suffered a completion failure.

As long as the wait queue is not empty, the Scheduler tries to schedule jobs, based on the scheduling policy, which is FCFS or FPFS. The Scheduler does this whenever it gets a signal from the Resource Monitor that a change in the resource availability has occurred. With FCFS, the Scheduler then invokes the Resource Broker for the job at the head of the wait queue. Only when that job fits does the Scheduler invoke the Resource Broker for the next job, etc. With FPFS, the Resource Broker is invoked once for every job request in the queue. Nevertheless, with FPFS, the Resource Monitor will be invoked only once during the activity of the Resource Broker in a single scheduling action.

If the Resource Broker finds that an ordered job fits, or is able to find a suitable allocation of an unordered job and can fill out its RSL specification, the Scheduler sends the job to the Co-allocator, and waits until the Co-allocator notifies it whether or not the job has been successfully submitted.

If there is a submission failure, the Scheduler increments the relevant field of the job request in the wait queue. If the number of submission failures is then still less than a configurable maximum number, the job request will be moved from its current position to the tail of the queue. If the number of submission failures has reached the maximum number, the Scheduler will remove the job from the queue.

3.3 The Resource Monitor

The Resource Monitor is responsible for collecting information about the resource status of all the clusters and for providing this to the Resource Broker. In our case, the only such information is the processor availability in the clusters. We considered two options for the Resource Monitor to retrieve the processor availability, namely using the Globus Toolkit's MDS component, and directly contacting the local resource managers, all of which in our case are PBS [3]. The MDS command grid-info-search in principle provides the information we need, but unfortunately, the MDS information is often not up-to-date since the GRAM reporter is not activated all the time to collect

the resource status and report it to the MDS. Therefore, we rejected the MDS as the basis for the Resource Monitor.

PBS provides the qstat command to retrieve status information from a cluster, which gives us the number of jobs running in the cluster, the number of compute nodes that are used by each job, and the identifier of each processor executing job processes, etc. After straightening out some little problems in the output of qstat, we found that we get accurate and timely information on processor availability, and so this is the option we used. The Resource Monitor stores the information in an output file, which is read by the Resource Broker.

3.4 The Resource Broker

When the Resource Broker gets a job request from the Scheduler, it depends on the request type how it operates. For ordered requests it simply checks whether the requested numbers of processors the job wants to use are available in the specified clusters. After it has done so, the Resource Broker sends a message back to the Scheduler whether the job fits or not.

For unordered jobs the Resource Broker employs the WF algorithm avoiding reusing clusters as much as possible (see Section 2.4). When the job can indeed be assigned to the system according to this algorithm, the Resource Broker completes the RSL specification of the job with the identifications of the clusters to which it has assigned the job's components, and sends it back to the Scheduler. Otherwise it sends a failure message back.

Note that compared to the situation with only plain DUROC, when a job does not fit, in our design DUROC is not called unnecessarily.

3.5 The Co-allocator and the Run List

The Co-allocator is responsible for the submission to DUROC of job requests it gets from the Scheduler. It is also responsible for monitoring the progress of all subjobs in every job while they are being executed. Therefore, it needs a so-called run list which stores the elements representing the running jobs. Each element of this list includes:

- The ID given by DUROC to the job during its execution.

- The number of subjobs in the job.

- A set of states describing the status of every subjob in the job.

The Co-allocator continuously waits for the Scheduler to send it a job request. When it receives such a request, the Co-allocator calls DUROC's job request function to submit the job through DUROC to the system. This function is

synchronous (blocking) so the Co-allocator must wait until the function returns. When it does, the Co-allocator gets the information of whether each subjob has been able to get to its destination cluster. If any subjob fails to do so, the Co-allocator will call the DUROC job cancel function to remove all subjobs associated to the job from their clusters, and it will send a submission failure message to the Scheduler. We have DCS use the Global Access to Secondary Storage (GASS) component of the Globus toolkit to automatically move the executable of the job to all clusters where a job component is going to run.

If all subjobs do get to their destination clusters, the Co-allocator must guarantee the job submission success by calling the DUROC barrier release function. This function will hold until all subjobs have entered their own barriers. It may happen that there is a subjob that fails to enter the barrier; after a time-out, the function then returns a failure message to the Co-allocator. However, if the function returns correctly, all the subjobs have been released from their barriers, and the Co-allocator will send a success message to the Scheduler. If the job submission succeeds, the Co-allocator creates a run list element for the job.

The monitoring component of the Co-allocator is active as long as the run list is not empty. For each element in this list, the Co-allocator will call DUROC's `globus-duroc-control-subjob-states` function which has the `Subjob_States` array as its output parameter. The Co-allocator can get the status of every subjob of the corresponding job from this array. If all subjobs have completed their execution successfully, the Co-allocator will remove the element of the completed job from the run list, and record the time when the job completes. When a job experiences a completion failure, the job's number of completion failures is incremented, and when this number does not exceed a maximum, the job is appended to the tail of the wait queue and its number of submission failures is reset to zero. All the progress of the job will be recorded in a log file.

As a note on the implementation, the whole of the DCS consists of four threads, one for the Scheduler and Resource Broker together, one for Resource Monitor, one for the submission function of the Co-Allocator, and one for the monitoring function of the Co-Allocator.

4 Experiments with the DCS

In this section we present some experiments with our Dynamic Co-allocation Service on the DAS. The purpose of these experiments is to show that indeed this service works correctly and reliably, we do not pretend to do a complete performance analysis of it here. Before we present the results of our experiments, we describe the DAS and the application we submit to it in our experiments.

4.1 The Distributed ASCI Supercomputer

The DAS [1] 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 [3]. Before the DCS was implemented, jobs spanning multiple clusters could only be submitted with plain DUROC [2].

4.2 The Application

The application that we repeatedly submit to the DAS to test the DCS implements a parallel iterative algorithm to find a discrete approximation to the solution of the two-dimensional Poisson equation (a second-order differential equation governing steady-state heat flow in a two-dimensional domain) on the unit square. For the discretization, a uniform grid of points in the unit square with a constant step in both directions is considered. The application uses a red-black Gauss-Seidel scheme (see for instance [15], pp. 429–433), for which the grid is split up into "black" and "red" points, with every red point having only black neighbours and vice versa. In every iteration, each grid point has its value updated as a function of its previous value and the values of its four neighbours, and all points of one colour are visited first followed by the ones of the other colour.

The domain of the problem is split up into a two-dimensional pattern of rectangles of equal size among the participating processes; in our experiments, only one process is assigned to each processor. Every process communicates with each of its neighbours in order to exchange the values of the grid points on the borders and to compute a global stopping criterion. When we execute the Poisson application on multiple clusters, the process grid is split up into adjacent vertical strips of equal width, with each cluster running an equal consecutive number of processes (we assume processes to be numbered in column-major order).

In [4] we have reported extensive measurements on the multicluster performance of this application, showing that for this type of applications, co-allocating them across wide-area systems is a viable option.

4.3 The Experimental Setup

In all of our experiments, we submit a batch of 40 jobs to the system, all of which run the application explained in Section 4.2. That is, rather than have the jobs arrive over
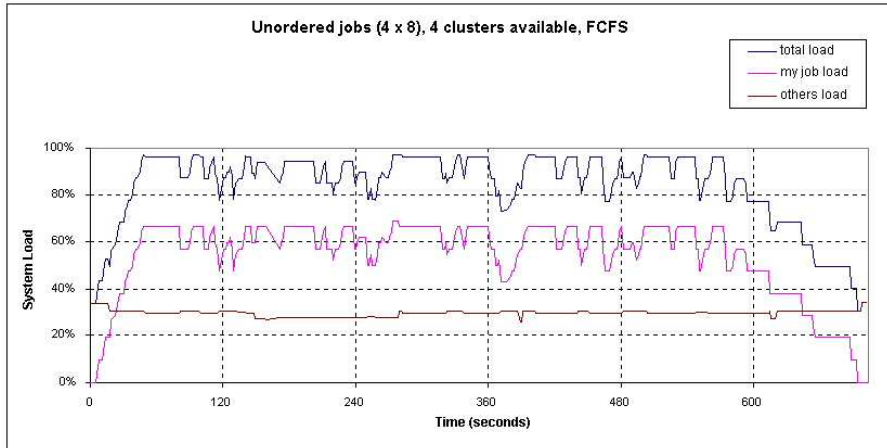
	response time	runtime	overhead
	(seconds)		
avg	377.2	79.4	32.9
stdev	177.2	29.8	17.8

Figure 2. The system loads and the job response time for the experiment with 4 clusters (144+3x64), unordered jobs of size 4x8, and FCFS.

some period of time, they arrive simultaneously. We note that this strains our DCS more than when the jobs would not arrive together, as now many jobs will initially not fit and the wait queue will be long. In all but one experiment we submit only ordered or unordered jobs; in the remaining experiment we submit an even mix of these types. All jobs always have 4 components of equal size, which is either 4 or 8 (indicated by 4x4 and 4x8). In none of our experiments was any job removed from the system because it reached the maximum number of submission or completion failures, which were both set to 3.

Only one of our experiments uses 4 clusters of the DAS, namely the largest cluster with 144 processors and three clusters with 64 processors each (indicated by 144+3x64). In all the other experiments, we could only employ two clusters, one of 144 and one of 64 processors (indicated by 144+64). In our experiments we use both the FCFS and the FPFS policies.

4.4 Experimental Results

We will present the results of five experiments. For each experiment we show a graph that plots the system loads due to our own jobs and due to the jobs of other users, and the sum of these over the time period from the jobs' submission until the last one of our jobs completes. These system loads are normalized with respect to the total capacity of the clusters that are actually used. In all of our five exper-

iments the system loads due to our jobs is (much) higher than the load due to the jobs of other users. In addition, for all experiments we report the average and the standard deviation of the job response time (total time in the system), of the run time, and of the time due to the overhead caused by DUROC. As it turns out, the standard deviation of the response time is rather large, which is caused by all jobs being submitted at the same time. The overhead due to DUROC has two components, one at job initialization and one at the job completion—the former is by far the largest.

In the first experiment, 4 clusters are employed, all jobs are unordered, and the scheduling policy is FCFS; the results are in Figure 2. We find that the DCS is able to drive the system load to very high levels. This is not very surprising as the jobs are unordered and the job component sizes are relatively small. The sudden drops in system load due to our jobs and the subsequent increase occur at job departures. This phenomenon is caused by the overhead of DUROC and of the DCS. When a job departs from the system, it takes at least a few seconds before the Resource Monitor notices a change in resource status, and then the Scheduler and Resource Broker have to do their work before the Co-Allocator can submit another job to DUROC. Note that here FPFS would have exhibited the same performance as FCFS because all jobs are of equal size.

In Figure 3 we compare two situations with 2 clusters, unordered jobs, and FCFS, where the only difference is the job size. The graphs show the same high total system load

and spiky behavior as in Figure 2. Note that the total duration of the experiment with the large job size is much longer, which is due to the larger job size but also to the longer average job runtime. Similarly, a comparison of the graph in Figure 2 and the top graph in Figure 3 shows that with identical workloads, the experiment in the 2-cluster case takes much longer (although not quite twice as long because the background load is lower).

In Figure 4 again we compare two situations with 2 clusters, one with ordered jobs and FCFS, and one with an even mix of ordered and unordered jobs and FPFS. Here, the ordered jobs consist of more components than there are clusters, and we specify two components of those jobs to go to either cluster (so in fact, we would achieve the same situation with ordered jobs of size 2x16). With only ordered jobs and FCFS (again we would have the same behavior with FPFS) we find that the total system load achieved is quite low. The reason is that the cluster with 64 processors is quite heavily used while the cluster with 144 processors is not so, but still for every job we need equal numbers of processors in either cluster. In the case of a mix of jobs and FPFS, the total system load is again quite high (and the duration of the experiment is much lower). This is caused by the presence of unordered jobs (which can use the capacity in the large cluster) and the use of FPFS which can schedule unordered jobs even when an ordered job is stuck at the head of the queue.

We conclude from our experiments first that our prototype works reliably. Furthermore, we can conclude from our sketchy experiments that ordered jobs may be an obstacle to achieving high utilizations, and that when there are both unordered and ordered jobs in the system, FPFS is definitely to be preferred over FCFS as the scheduling policy.

5 Related Work

Not very much work has been done on the design, implementation, and performance analysis of co-allocation in multicluster systems and in grids. In terms of designs and implementations, a system that is able to perform allocation of resources in different administrative domains to a single job is Condor with its DAG-manager [14]. Condor's DAGMan takes as input job descriptions in the form of Directed Acyclic Graphs (DAGs), and schedules a task in such a graph when it is enabled (i.e., when all its precedence constraints have been resolved). However, no simultaneous resource possession implemented by a co-allocation mechanism is implemented. In [17], the Condor class-ad matchmaking mechanism for matching single jobs with single machines is extended to "gangmatching" for co-allocation. The running example in [17] 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.

In [9], 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 [10] planning tool and the Chimera [13] data definition tool. The workflows are represented by DAGs, which are assigned to resources using the Condor DAGMan and Condor-G [14].

In our previous work [4, 6, 7] we have studied the performance of processor co-allocation in multiclusters through simulations for a wide range of such parameters as the number and sizes of the job components, the number of clusters, the service-time distributions, and the number of queues in the system. There, we considered both synthetics workloads, and workloads derived from the logs of the DAS and from application runtimes on the DAS. In [11, 12], co-allocation (called multi-site computing there) is studied with simulations, with as performance metric the average weighted response time. One of the most important findings is that when the slowdown of jobs due to the wide-area communication is less than or equal to 1.25, it pays to use co-allocation. In [5], we consider the maximal utilization, i.e., the utilization at which the system becomes saturated, as a metric for assessing the performance of processor co-allocation.

6 Conclusions and Future Work

In this paper we have presented the design of a Dynamic Co-Allocation Service (DCS) for processor co-allocation in multicluster systems, which has been implemented on our DAS multicluster system. We have also shown the results of experiments that indeed show that this DCS works reliably, and that it is able to achieve a quite high total system load, although the jobs submitted in our experiments were not very large. As far as the authors know this is the first implementation of processor co-allocation with proper resource-brokering functionality and fault tolerance.

We are only at the beginning of our design and implementation efforts of co-allocation in grids. In particular, we are planning to extend the current design of the DCS to more types of resources, to more heterogeneous systems both with repect to the hardware and the local resource managers, and to more complicated job types (e.g., work flows). We note that we have been experimenting with a design of mechanisms for the co-allocation of both processors and information resources which does away with DUROC altogether, but which does use components of the Globus toolkit. Finally, we would like to do a better performance analysis. One of the complicating factors here is the lack of reproducibility of experiments in systems that have a background load submitted by other users that we cannot con-
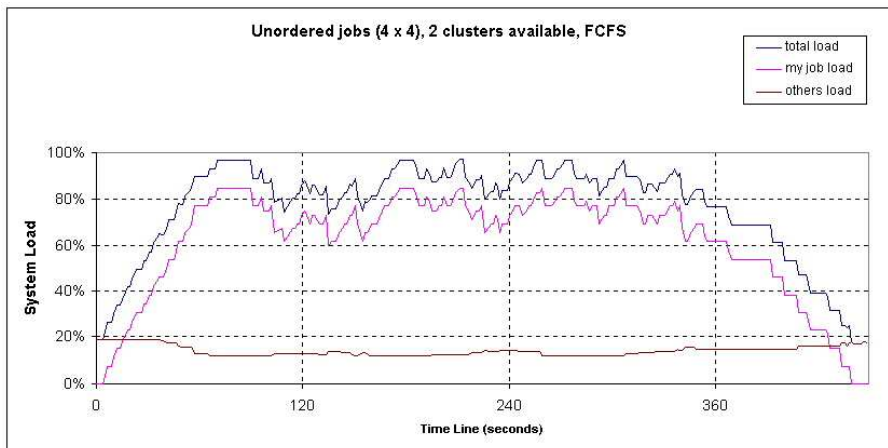
trol.

References

[1] The Distributed ASCI Supercomputer (DAS). www.cs.vu.nl/das2.

[2] The Globus Toolkit. www.globus.org.

[3] The Portable Batch System. www.openpbs.org.

[4] 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, *Proc. of the 9th Workshop on Job Scheduling Strategies for Parallel Processing*, volume 2862 of *LNCS*, pages 105–128. Springer-Verlag, 2003.

[5] A. Bucur and D. Epema. The Maximal Utilization of Processor Co-Allocation in Multicluster Systems. In *Proc. of the Int'l Parallel and Distributed Processing Symp. (IPDPS)*, pages 60–69. IEEE Computer Society Press, 2003.

[6] A. Bucur and D. Epema. The Performance of Processor Co-Allocation in Multicluster Systems. In *Proc. of the 3rd IEEE/ACM Int'l Symp. on Cluster Computing and the GRID (CCGrid2003)*, pages 302–309. IEEE Computer Society Press, 2003.

[7] A. Bucur and D. Epema. Trace-Based Simulations of Processor Co-Allocation Policies in Multiclusters. In *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, pages 70–79. IEEE Computer Society Press, 2003.

[8] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proc. of the 8th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-8)*, pages 219–228, 1999.

[9] E. Deelman et al. Mapping Abstract Complex Workflows onto Grid Environments. *J. of Grid Computing*, 1:25–39, 2003.

[10] E. Deelman et al. Pegasus: Mapping Scientific Workflows onto the Grid. In *Proc. of the 2nd European Across Grids Conference*, 2004.

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

[12] 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.

[13] 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.

[14] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Proc. of the 10th IEEE Symp. on High Performance Distributed Computing (HPDC-10)*, pages 7–9. IEEE Computer Society Press, 2001.

[15] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing*. Benjamin/Cummings, 1994.

[16] D. Lifka. The ANL/IBM SP Scheduling Systems. In D. Feitelson and L. Rudolph, editors, *Proc. of the 1st Workshop on Job Scheduling Strategies for Parallel Processing*, volume 949 of *LNCS*, pages 295–303. Springer-Verlag, 1995.

[17] R. Raman, M. Livny, and M. Solomon. Policy Driven Heterogeneous Resource Co-Allocation with Gangmatching. In *Proc. of the 12th IEEE Int'l Symp. on High Performance Distributed Computing (HPDC-12)*, pages 80–89. IEEE Computer Society Press, 2003.

Unordered jobs (4 x 8), 2 clusters available, FCFS
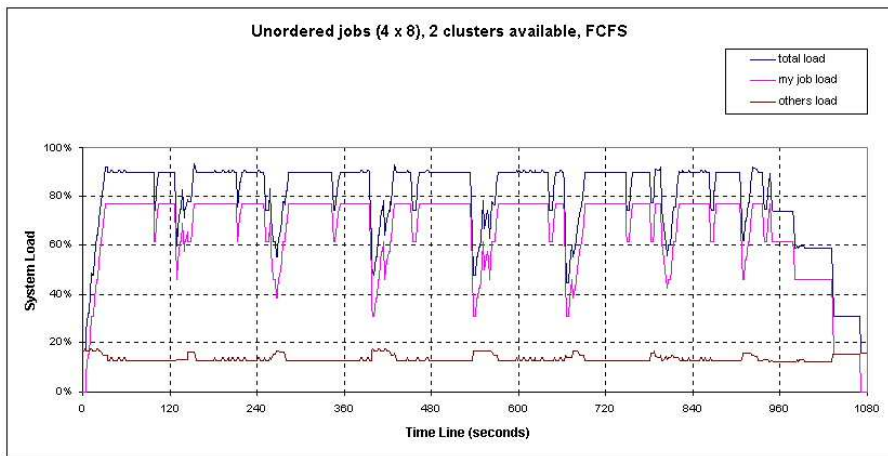
	response time	runtime	overhead
	(seconds)		
avg	581.3	102.1	25.4
stdev	295.2	12.5	3.5

Unordered jobs (4 x 4), 2 clusters available, FCFS

	response time	runtime	overhead
	(seconds)		
avg	267.2	76.8	26.9
stdev	108.2	17.0	3.2

Figure 3. The system loads and the job response time for the experiments with 2 clusters (144+64), unordered jobs of size 4x8 (top) and 4x4 (bottom), and FCFS.

	response time	runtime	overhead
	(seconds)		
avg	1135.2	87.8	21.9
stdev	611.1	10.0	3.4

	response time	runtime	overhead
	(seconds)		
avg	634.7	92.4	25.3
stdev	335.3	15.6	6.5

Figure 4. The system loads and the job response time for the experiments with 2 clusters (144+64), ordered jobs (top) and an even mix of ordered and unordered jobs (bottom) of size 4x8, and FCFS (top) and FPFS (bottom).