Resource Co-Allocation in Computational Grids*

Karl Czajkowski† Ian Foster‡ Carl Kesselman†

Abstract

Applications designed to execute on "computational grids" frequently require the simultaneous co-allocation *of multiple resources in order to meet performance requirements. For example, several computers and network elements may be required in order to achieve real-time reconstruction of experimental data, while a large numerical simulation may require simultaneous access to multiple supercomputers. Motivated by these concerns, we have developed a general resource management architecture for Grid environments, in which resource co-allocation is an integral component. In this paper, we examine the co-allocation problem in detail and present mechanisms that allow an application to guide resource selection during the co-allocation process; these mechanisms address issues relating to the allocation, monitoring, control, and configuration of distributed computations. We describe the implementation of co-allocators based on these mechanisms and present the results of microbenchmark studies and large-scale application experiments that provide insights into the costs and practical utility of our techniques.*

1. Introduction

Advances in networking infrastructure have led to the development of a new type of "computational grid" infrastructure that provides predictable, consistent and uniform access to geographically distributed resources such as computers, data repositories, scientific instruments, and advanced display devices [12]. Such Grid environments are being used to construct sophisticated, performance-sensitive applications in such areas as supercomputer-enhanced instruments, desktop supercomputing, tele-immersive environments, and distributed supercomputing [4, 3, 22, 19, 23, 7].

A common characteristic of many of these applications is a need to allocate multiple resources simultaneously. For example, one recent record-setting simulation employed 13 parallel supercomputers containing a total of over 1300 processors [3]. In another recent experiment, a scientific instrument, five computers, and multiple display devices were used for collaborative real-time reconstruction of X-ray source data [27]. While such simultaneous allocation can in principle be achieved manually, in general we require a resource management infrastructure that supports not just the management of individual resources but the coordinated management of multiple resources.

In a previous paper [6], we presented a general resource management architecture for Grid environments and introduced the concept of *resource co-allocation* as a necessary function in this architecture. The co-allocation problem has not previously received attention in the high-performance distributed computing community. While co-allocation issues arise in other situations (for example, co-allocation of processors and memory [20], co-allocation of CPU and networks [25, 24], setup of reservations along network paths [28, 10, 8, 16, 2]), the dynamic nature of the Grid environment poses special challenges including the potential for failure and the heterogenous nature of the underlying resource set.

In this paper, we propose a layered co-allocation architecture that addresses the challenges of Grid environments by providing a flexible set of basic co-allocation mechanisms that can be used to construct a wide range of application-specific co-allocation strategies. These mechanisms allow for the dynamic construction, configuration, and control of collections of separately administered and controlled resources, while providing an application with a single abstraction for monitoring and controlling those resources. An important aspect of our approach is that it has been designed to facilitate collective resource allocation in the face of the diverse failure modes that can occur in the Grid environment.

The main contributions of this paper are that we:

- define the co-allocation problem for computational grids and identify the general requirements that a co-allocation service should meet,

*See `http://www.globus.org` for more information.

†Information Sciences Institute, University of Southern California, Marina del Rey, CA 90292

‡Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439; Department of Computer Science, University of Chicago, Chicago, IL 60637

- define a basic set of mechanisms by which a variety of application-specific co-allocation strategies can be constructed, and
- use both microbenchmark experiments and large-scale application studies to obtain data on the effectiveness of these mechanisms in realistic settings.

The rest of the paper is as follows. In Section 2, we introduce the co-allocation problem and illustrate why it is important to Grid applications. In Section 3, we describe a general co-allocation architecture and define basic mechanisms that can be used to construct application-specific co-allocation solutions. In Section 4, we describe an implementation of our proposed co-allocation mechanisms, and examine their performance in a networked environment. We then discuss the use of these mechanisms in a range of Grid applications. Finally, we summarize our results and discuss future work in Section 5.

2. The Co-Allocation Problem

For the purposes of this discussion, we model the successful execution of a computer program as comprising an *allocation* phase, in which required computational resources are acquired and various computational objects created; a *configuration* phase, in which the application is initialized; and a subsequent *monitoring/control* phase during which the program executes. In Grid environments, we define the co-allocation problem as the provision of allocation, configuration, and monitoring/control functions for the *resource ensemble* required by a single application. While in this paper we focus on computational resources, in general, we define resources to include all devices that an application might require, including networks, memory, storage, rendering hardware, and display devices.

Grid applications typically operate on ensembles of resources that span administrative domains of control, with resources in the ensemble being independently operated. Furthermore, access to resources is in general unreliable, due to either competing demands for the resource or outright failure. These two features of Grids complicate the co-allocation process, rendering ineffective existing approaches based on centralized control and a strategy of aborting on the failure of any resource request.

The following application scenario illustrates some of the difficulties that may be encountered:

> A large distributed simulation requires 400 processors in order to achieve a specified level of fidelity and speed. Five computers are identified that can collectively provide the required 400 processors at the desired time. The allocation process is started, but one of the computers turns out to be unavailable due to a system crash. This failure is handled by dropping that computer from the ensemble and adding another, located dynamically. The application starts on these five systems and 400 processors.
>
> The next step in the co-allocation procedure is initialization of the simulation system. Here, another problem arises. Four of the five systems complete initialization within a few minutes and rendezvous at an initial barrier, but after five minutes the fifth system has not joined them. Later investigation will reveal that this delay occurred because that system was overloaded with other work, but at this moment all that is known is that the startup deadline is compromised. The solution adopted in this case is to drop the "faulty" system from the ensemble, and proceed with just four systems, at a decreased level of simulation fidelity, but with the same completion time.

This example illustrates three important points concerning co-allocation:

- The failure of individual components is a common occurrence.
- There are numerous possible failure modes that evidence themselves in different ways, ranging from an error report to lack of progress.
- The definition of "failure" may be application-dependent. In the example, unexpectedly slow execution was viewed as failure; in other contexts, an application might reject a resource as unacceptable at runtime because of network interface performance, numerical accuracy of system libraries, or cost.

Analysis of Grid application scenarios such as that just described leads us to conclude that no single co-allocation strategy can be effective for all purposes. For example, an atomic transaction capability that ensures that an application starts either on all required systems, or on none, is inadequate for the scenario above, where first of all a failed resource is replaced by an alternative, and later a nonresponsive resource is discarded altogether. We argue instead for mechanisms that can be used to implement a variety of application-specific co-allocation strategies. We present such a set of mechanisms in the next section.

2.1 Related Work

The techniques required to implement co-allocation within a uniprocessor or parallel computer are well understood. Resource managers for parallel computers, such as LoadLeveler [18] coallocate homogeneous collections of

resources (processors) to applications, while more general resource managers such as NQE [1] and PBS [17] allow the co-allocation of heterogeneous resource sets: for example, processors and memory. Co-allocation of networked computational resources is supported by a number of resource management systems, including LSF [29], and Codine [15]; however, these systems assume that they are in total control of all resources, which may not be the case in Grid environments. Furthermore, these systems provide only limited application level control. For example, LSF terminates an entire application if any resource in the co-allocation set is determined to have failed.

Legion, a distributed object-oriented system with goals similar to the Grid systems described here, supports the notion of resource co-allocation via an entity called an *Enactor* [5]. An *Enactor* provides a co-allocation mechanism similar to the the atomic transaction strategy discussed in Section 3, and consequently will suffer from the same limitations as this approach. Legion also relies on the ability to reserve resources in advance of allocating them, with limited recourse if the underlying resource management system does not support reservation. Finally, we note that access to Legion co-allocation methods can only be obtained from applications that can access the Legion object-oriented programming model.

Within the networking community, various protocols have been proposed and are being investigated for co-allocating network resources along a route [28, 10, 8, 16, 2]. Also relevant to the co-allocation problem is multimedia system research concerned with identifying the appropriate mix of resources required to provide desired end-to-end QoS. Multimedia applications have motivated the development of techniques for allocating both memory and CPU for channel handlers [20] and of CPU, bandwidth, and other resources for video streams [25, 24]. However, these techniques are specific to particular mixes of resources and do not extend easily to other resource types.

2.2 Scheduling and Advance Reservation

We make some brief comments on the role that scheduling and advance reservations can play in co-allocation.

A co-allocation request typically requires that each of a set of resources be able to deliver a specified level of service at a specified time. Yet even if a resource is physically capable of meeting a service requirement, competing demands from other computations may lead to a particular request being handled incorrectly:

- If the resource offers simply "best effort" service, such as is often the case with network bandwidth, disk access and CPU access on timesharing systems, a request for a resource may be granted, but with an inadequate service level.
- If access to the resource is managed by a resource management system, then the request for the resource may be denied or may block awaiting resource availability.

Hence, depending on the capabilities of the management system associated with the resource in question, *scheduling conflicts* may result in a particular request suspending, failing, or proceeding with degraded functionality.

We argue that in a Grid environment, it is not reasonable to assume a centralized "global scheduler" that resolves such scheduling conflicts. However, a number of other techniques can be used to improve co-allocation effectiveness. One approach is to enhance the local resource management system. For example, by incorporating *advance reservation* capabilities into a local resource manager, a co-allocator can obtain guarantees that a resource will deliver a required level of service when required. This approach is discussed further in [13].

Alternatively, the resource management system can publish information about the current queue contents and scheduling policy, or publish forecasts (based, for example, on queue time prediction algorithms [9, 26]) of expected future resource availability. This information can be used to improve the success of co-allocation by constructing co-allocation requests that are *likely* to succeed. For example, the co-allocator may use information published by local managers to select from among alternative candidate resources, or it may attempt to allocate more resources than it really needs. Simulation studies have shown that this approach can be effective if there is a minimum period of time over which load information remains valid [14].

3. A Co-Allocation Architecture

We now describe our approach to the co-allocation problem. The basic idea is to define a set of mechanisms that can be used to implement a variety of different co-allocation strategies. These mechanisms permit considerable application-level flexibility in how failure is handled, for example:

- Advance reservations and/or forecasts or other information can be used to reduce the probability of resource unavailability, but are not required;
- A user can specify whether desired resources are essential, disposable, or should trigger callbacks if not available;
- A user can define "failure" and how to respond to it;
- A user can control the order in which resources are allocated, so as to reduce the cost of failure; and
- A user can similarly control the configuration of computations once resources are acquired.

In the following, we first talk briefly about some architectural assumptions and then describe the mechanisms that we have developed to support the allocation, configuration, and monitoring/control phases of a computation.

3.1 Architectural Overview

We assume a distributed resource management architecture that comprises three distinct components:

- A *resource management* component provides mechanisms for managing individual resources. In the following discussion, we assume the use of the Globus Resource Management Architecture [6], which defines a modular infrastructure with distinct information, resource management, security, and other components.
- A *co-allocation mechanism* component, layered on top of the single-resource management component, provides the mechanisms required to implement co-allocation strategies.
- *Co-allocation agents* use co-allocation mechanisms to implement application-specific strategies for the collective allocation, configuration, and monitoring/control of ensembles of resources.

In principle, application programmers could implement co-allocation strategies via direct calls to the underlying resource management infrastructure. However, we believe that there are significant advantages to defining an intermediate co-allocation mechanism component that bridges the gap between, on the one hand, applications (or resource brokers acting on their behalf) that require collections of resources; and, on the other hand, local resource managers that manage a single resource. This set of common co-allocation mechanisms simplifies the design and implementation of co-allocation agents; promotes interoperability between various co-allocation agents; and enables the development of sophisticated co-allocation schemes, for example by nested or hierarchical co-allocators.

We conclude this subsection with a note concerning parallel computers. An abstract view of co-allocation is that it involves the creation of N processes on N distinct resources, each under the control of a local resource manager; if some subset K of those N processes and resources happen to be part of a logical unit called a "parallel computer" (or "network of workstations" etc.), that fact does not change the co-allocation problem. Yet in practice there can be significant performance advantages to treating such "subjobs" as collective units, creating them via a single call (rather than K calls) to the appropriate resource manager, and also monitoring and controlling their execution via similar collective calls. Hence, our techniques and mechanisms allow for this possibility.

3.2 Allocation Mechanisms

We now proceed to describe the co-allocation mechanisms that we have developed, looking first at allocation and then (in subsequent subsections) at configuration and monitoring/control.

We define the allocation phase of a computation to comprise those actions performed between the issuance of a resource co-allocation request and the point when the program on the co-allocated resources can be viewed as being successfully started. In effect, these actions are concerned with mapping a description of the ensemble of required resources into a representation of the acquired resources that can later be used for monitoring and control. The critical questions are how the resource ensemble is specified and the methods used to go from this specification to a initialized application.

An important concern during this process is dealing effectively with the various types of failure noted above. Experience with a range of applications leads us to believe that failure (and hence "successful start") is an application-dependent concept. Hence, when spawning a process on a remote computer, it is not sufficient that the local operating system (or scheduler) on that computer tell us that the process has "started" successfully; we need to hear from the application itself, which may wish to perform various checks (e.g., relating to the numerical accuracy of local libraries, amount of free disk space, etc.) before announcing a successful start.

These considerations lead us to manage the allocation process via a distributed two-phase commit protocol, as follows:

1. A co-allocation agent issues requests to the local resource managers for the resources on which processes are to be created;
2. Either failure is signalled, or processes are started; once started, a process performs any local status checks and then reports either successful or unsuccessful startup and then enters a barrier, awaiting a communication from the co-allocation agent.
3. The co-allocation agent decides whether to proceed with allocation or not, on the basis of status messages received; if the decision is to "commit" then the waiting processes are instructed to proceed.

Notice that this approach requires minor modifications to the participating application program, which must call the "barrier" function prior to proceeding with computation.

We have developed mechanisms to support two variants of this basic strategy: one simple set supports what we term the *atomic transaction* strategy, and a more complex set supports a more flexible *interactive transaction* strategy. We describe both in the following.

Atomic Transaction Mechanisms and Strategy. The most straightforward co-allocation strategy is what we term an *atomic transaction* approach. All required resources are specified at the time the request is made. The request succeeds if all resources required by the application are allocated. Otherwise, the request fails and none of the resources are acquired. (The possibility of indefinite delay can be avoided by using timeouts on individual requests.) The mechanisms provided to support this strategy comprise an `allocation` function on the client side, which returns success or failure, and a `barrier` function for use within the application.

This strategy is effective when the desired resource ensemble is known when the co-allocation request is issued and when the likelihood that a resource fails between the construction of the request and program startup is low. Note that the contents of a co-allocation request can be constructed incrementally, but may not be changed once the request has been initiated. Thus, the only way of dealing with a request failure is to formulate and resubmit a revised co-allocation request, based on more current information.

Initially, we did not believe these constraints were too restrictive. However, experience with several large applications proved that the static approach did not work well in practice. Especially problematic was that application startup and initialization on large computers can take tens of minutes and in many cases failure is due to issues other then failure of the underlying resource: for example, slow system performance or application failure. Consequently, failures in a resource often could not be detected until after the application had been started.

Interactive Transaction Strategy and Mechanisms. The *interactive transaction* approach was developed to overcome problems observed in the atomic transaction strategy. Basically, we generalize the two-phase commit structure used in the atomic transaction model to enable greater application-level control.

In the interactive approach, the contents of a co-allocation request can be modified—via editing operations `add`, `delete`, and `substitute`— until the commit operation. To further facilitate the reconfiguration of a resource set under a variety of failure conditions, we classify each element of the resource set into one of three categories:

- `required`: Failure or timeout of a required resource causes the entire computation to be terminated, regardless of whether a commit has been issued or not. This behavior is as in an atomic transaction co-allocator; it is intended for resources without which a computation cannot proceed.

- `interactive`: Failure or timeout of an interactive resource results in a callback to the application, which can then delete the resource from its resource set or substitute other resources. This behavior is intended for resources that are nonessential, or for which it may be feasible to find replacements.

- `optional`: Optional resources do not participate in the commitment procedure: failure or timeout is ignored. This behavior is intended for resources that do not have to be coordinated within the overall application structure.

These basic mechanisms can be used to construct a wide variety of application-specific behaviors. For example, `interactive` resources allow an application (or co-allocation agent acting on its behalf) to replace slow or failed elements of a request if an alternative resource can be found. Alternatively, one may be able to decrease allocation time by requesting several alternative resources simultaneously and committing to the first that becomes available. Finally, the order of resource acquisition can be controlled via interactive modification of the resource specification: for example acquiring all required resources first and then adding interactive resources to the set.

3.3 Configuration Mechanisms

Successful completion of a co-allocation request results in the creation of a set of application processes on the resources in the specified resource set. The further configuration or initialization of these processes frequently requires that these processes discover and communicate with one another. For example, if the newly created processes are to participate in a Message Passing Interface (MPI) computation, then each process must determine the total number of processes, determine its own name (in this case, an integer "rank" within the set of processes), and establish a (virtual or physical) all-to-all communication structure that allows it to communicate with any other process. In other situations, other naming and communication structures may be appropriate: for example, on machines with networks or disks attached to a specific node, it may be advantageous for this node to always be assigned a specific rank. In more dynamic applications designed to tolerate failure, a linear ordering of resources and an all-to-all communication structure may not be appropriate.

In order to accommodate a wide range of configuration possibilities, we identify a basic set of operations from which alternative approaches to configurations implemented. These mechanisms take into account the presence of subjobs, as noted above, and include the following functions:

- determine the number of subjobs in a resource set;

- determine the size (i.e., number of processors) of a specific subjob;
- communicate between at least one node in a subjob and every other node in the subjob,
- for at least one node in a subjob, be able to communicate with at least one node in every other subjob.

3.4 Monitoring and Control Mechanisms

The allocation and configuration phases of the co-allocation process result in the creation of a set of processes executing on a set of resources. During the program's execution, it is desirable that we be able to monitor and control the ensemble as a collective unit, rather than being required to treat its constituent components independently. The monitoring and control operations that we defined have this property.

Monitoring operations allow a client program to receive notification when the resource set changes state. In addition to the obvious global state transitions of failure and termination, the complex failure modes encountered in Grid applications lead to a need to support and respond to individual process state transitions as well. For example, a large distributed simulation such as [21] might be willing to continue execution even if a component fails. Hence, the monitoring interface should allow for state transitions to be signalled to the monitoring program, which can then act upon this transition in a manner that is appropriate for the application.

Similarly, control operations allow for the manipulation of the resource set as a whole. One required control operation is to "kill" the application; other operations may be required in the future.

4. Experiences

We have constructed implementations of both the atomic and interactive transaction co-allocation strategies in the context of the Globus toolkit and have gathered extensive experience with their use in a range of applications. Here, we first review briefly on the structure of the co-allocator implementations and then report on our experiences using the co-allocators in controlled experiments and in large-scale applications. While the former experiments are not central to the principal point of this paper, namely the importance of interactive transactions for effective co-allocation, the results provide insights into the costs associated with co-allocation.

4.1 Co-Allocator Implementation

We have constructed two co-allocator implementations. Experience with the first, an atomic transaction co-allocator called the Globus Resource Allocation Broker (GRAB) motivated the design and implementation of the second, an interactive transaction co-allocator called the Dynamically Updated Resource Online Co-allocator (DUROC). DUROC forms a central part of the Globus toolkit and has seen extensive use both directly in applications and within tools such as MPICH-G [11], a Grid-enabled implementation of the Message Passing Interface based on the MPICH system.

Both GRAB and DUROC are implemented as a set of libraries designed to be linked with application codes. Co-allocation requests are expressed in terms of an extensible resource allocation language, or RSL [6].

An example of a DUROC co-allocation RSL expression for an application that involves a single master process and several four-processor subjobs ("workers") is shown in Figure 1. The master resource is labeled as `required`, while the worker resources are labeled as `interactive`. During co-allocation, the identity of any successful workers is communicated to the application. If enough worker processors cannot be allocated, the application can abort the computation; once enough resources have been collected, it can terminate subjobs that have not yet responded to the request prior to committing the configuration and proceeding. Alternatively, if the number of workers was not important, all worker processes might be labeled `optional`. In this case, workers join the computation as and when they become active.

```
+(&(resourceManagerContact=RM1)
   (count=1)(executable=master)
   (subjobStartType=required))
 (&(resourceManagerContact=RM2)
   (count=4)(executable=worker)
   (subjobStartType=interactive))
 ...
 (&(resourceManagerContact=RMn)
   (count=4)(executable=worker)
   (subjobStartType=interactive))
```

Figure 1. Example of a RSL co-allocation request for a master/worker computation submitted to DUROC.

The DUROC libraries consist of a *control library* whose functions are used within a co-allocation agent to initiate and control an allocation request, and an *application side library*, which provides the barrier operation used within the co-allocated application. A co-allocation agent uses the control library to:

- create an RSL expression to describe the request;

- monitor the status for the request via DUROC callbacks, adding or deleting subjobs according to its needs; and

- commit to a specific configuration and then wait for all of the subjobs in the final configuration to check into the barrier or for the allocation to fail.

A process that is to run on a co-allocated node starts as normal. The first thing it does is perform any non-side effect producing initialization necessary to determine if the component execution can proceed. It then calls the co-allocation barrier, signalling whether or not it has completed startup successfully. Depending on how co-allocation proceeds, the process may or may not return from the barrier (hence the need to delay any unreversable initialization). If the barrier exits successfully, the application knows that the co-allocation was successful and it can proceed with normal initialization and execution.

DUROC also provides an application with a set of library routines that implement the configuration mechanisms discussed in Section 3.3.

4.2 DUROC Experimental Results

We used a series of microbenchmark studies to evaluate the cost of DUROC co-allocation mechanisms and to compare these costs with those of basic resource management functions provided by the Globus Resource Architecture (GRAM).

Experiments were conducted on an 64-node Silicon Graphics Origin 2000 shared-memory parallel computer running the Irix 6.5 operating system. Allocation requests were submitted from a remote machine. Both the submitting machine and the machine on which resources were allocated were located on a lightly loaded network with a latency between the two computers of about 2 msec. To eliminate any source of queuing delay, GRAM was configured to respond to allocation requests by immediately "forking" the requested number of processes.

Baseline measurements from which to evaluate DUROC performance were obtained by conducted a series of experiments with GRAM alone. A series of GRAM requests were submitted, varying the number of processes created. For each request, we measured the time that elapsed from invocation of the allocation command to successful startup of the processes on the target machine. The results, presented in Figure 2, show that the cost of a GRAM submission is largely insensitive to the number of processes created.

A breakdown of where time is spent in a single process GRAM request is shown in Figure 3. The largest single contributor to the time required to process a GRAM request is in "initgroups," a Unix system call. This call is expensive because it must consult remote group databases (via the Network Information Service). The second major contributor to request execution time is in a call to the Grid Security Infrastructure (GSI) library that performs a mutual authentication of the requestor and target machine. These operations are computationally intensive and also require network communication. All other costs are an order of magnitude smaller.

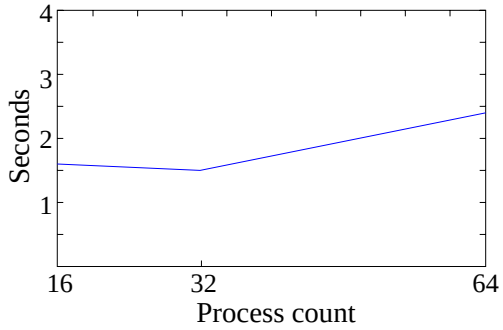

Figure 2. GRAM submission latency for several parallel job sizes.

operation	latency (s)
initgroups()	0.7
authentication	0.5
misc.	0.01
fork()	0.001

Figure 3. Breakdown of times spent in processing a single-process GRAM request

To determine the cost of co-allocation over that of basic allocation, we timed a series of DUROC co-allocation requests, varying both the total number of processes created and the number of subjobs. We measured time for a DUROC request by starting a timer in the submitting program immediately before calling the co-allocation function and then stopping this timer on receipt of a message sent from an application process immediately upon exiting the co-allocation barrier. Results of this experiment with the number of processes fixed at 64 is shown in Figure 4. This graph shows how DUROC submission time changes as the number of subjobs is varied from 1 to 25, while keeping the total number of processes created constant.

Our results show that co-allocation time is essentially independent of the number of processes but varies linearly with the number of subjobs. The linear relationship between number of subjobs and total time is to be expected as each subjob results in a distinct GRAM request, each

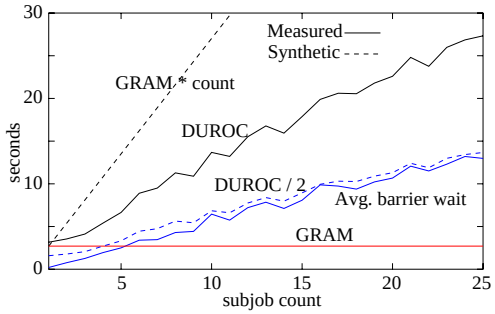

Figure 4. DUROC submission times to a host 2 milliseconds away from the client workstation on the network.

with its inherent authentication and protocol overhead. This can be seen by examining a timeline of a DUROC submission, shown in Figure 5, which illustrates that the individual GRAM requests from which a DUROC request is constructed must be submitted sequentially.

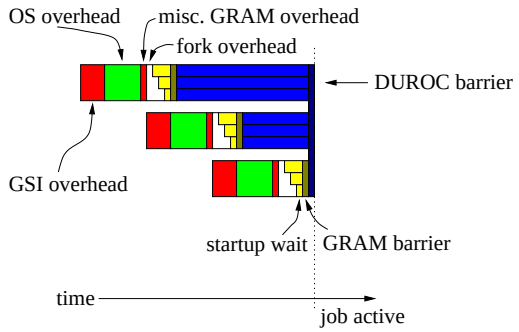

Figure 5. Timeline of a DUROC submission.

Figure 5, however, also indicates that there is some opportunity for overlap in processing a DUROC request once that basic resource allocation request has been processed. The slope of lines in Figure 4 indicates that this is indeed the case. A single subjob takes 2 seconds, and 25 subjobs take 28 seconds; this is 44% less time for multiple subjobs than one would expect with zero concurrency.

A more detailed examination of Figure 4 suggests a model for co-allocation cost. Given our experience that DUROC costs are essentially constant regardless of the number of processes in a subjob, we can assume a simplified model where GRAM imposes some per-transaction latency and then instantaneously starts all processes. Job processes are therefore started in batches, one subjob at a time; and all processes must wait until the final batch is active. Under this assumption, average process wait time (in process-seconds) is:

$$\frac{\sum_{i=0}^{M-1} k\frac{N}{M}i}{N} = \frac{k\sum_{i=0}^{M-1} i}{M} \approx \frac{kM}{2}$$

where M is the number of subjobs and k is the latency introduced by the ith subjob into the $i-1$th subjob's barrier (a time which is less than an independent GRAM request due to pipelining of subjob requests). Note that the total job latency is kM under the assumption of uniform GRAM costs. As the plot shows, our observations verify that the average barrier wait is approximately one half the total job latency, in agreement with the simplified analytical model.

In addition, we have inspected per-process barrier wait times for several runs to verify that the raw data occur in per-subjob blocks. The barrier times do exist in blocks, and the shortest wait time is always zero (with 10 ms resolution). Anecdotal data from large distributed runs also indicate that barrier synchronization costs are negligible in the wide-area compared to local startup delays introduced both by GRAM and by local scheduler queues (remember that the above experiments were with fork-based job starts, impossible on most production parallel machines).

4.3 Application Experiences

We have been using first GRAB and more recently DUROC in a wide range of application experiments over the past 2 years.

Experiences with GRAB. Early experiments demonstrated the benefits of our co-allocation architecture. For example, initial experiments in large-scale distributed interactive simulation described in [21] involved the co-allocation of up to 7 supercomputers at one time. These experiments were conducted initially without and then with the help of GRAB, proving the benefits of co-allocation; the cost of allocation, monitoring, and control operations was reduced from literally tens of minutes when performed manually to a few keystrokes when using GRAB.

These early experiments also convinced us that atomic transactions were inadequate for Grid environments. On several occasions, we had actually acquired an acceptable number of resources, but then had to abort and restart the simulation due to failure or slowness of a single resource. As startup and initialization of large simulations on large parallel computers can take 15 minutes or more, the cost inherent in such unnecessary restarts is tremendous. These problems were overcome with the development of DUROC.

Experiences with DUROC. DUROC has been used in a wide range of settings and has proven to provide an effective co-allocation service. In one example, DUROC was

used to start the largest distributed interactive simulation ever performed, starting a computation on 1386 processors distributed across 13 different parallel supercomputers [3] and 9 widely distributed sites. In this particular example, there were difficulties starting some components of the simulation (including machine, network and application failure) and DUROC was successfully used to configure around these failures.

DUROC has also been used to construct of other Globus components. For example, the Grid-enabled MPICH-G implementation of MPI [11] uses DUROC to start the elements of an MPI job. In this case, all DUROC calls are hidden in the MPI library, and an application does not have to make any modifications to benefit from DUROC co-allocation. The use of DUROC to initiate MPI is particularly advantageous for large-scale "hero" computations, as we can reconfigure the MPI job at startup to overcome resource failure.

5. Conclusions and Future Work

Experience shows that the co-allocation of multiple resources is a challenging problem in Grid environments, due to application requirements for multiple resources and the inherent unreliability of the resources in question. We have described two different strategies for dealing with the problem. The first atomic transaction approach uses a simple two-phase commit strategy to implement atomic co-allocation semantics, in which a specified set of resources is allocated in its entirety or not at all. The innovative interactive transaction approach allows for application-level guidance of resource selection and failure handling prior to commitment, hence providing greater flexibility and resilience to failure. We have developed implementations of both strategies in the context of the Globus toolkit and have demonstrated the utility of both co-allocators in practical settings. The interactive transaction strategy in particular has proven its worth in a wide range of applications.

While the co-allocation strategies presented here provide valuable mechanisms, they do not address the problem of ensuring that a given co-allocation request will succeed. For this, we believe that some form of advance reservation will ultimately be required. We are currently investigating how the current resource management architecture can be extended to include reservation, and how the co-allocation approaches presented in this paper can be applied to co-reservation as well as co-allocation [13].

Acknowledgments

We are grateful to our colleagues within the Globus project for many helpful discussions on these topics: in particular, we acknowledge Steve Fitzgerald, Brian Toonen, and Steven Tuecke.

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, U.S. Department of Energy, under Contract W-31-109-Eng-38; by the Defense Advanced Research Projects Agency under contract N66001-96-C-8523; by the National Science Foundation; and by the NASA Information Power Grid program.

References

[1] Cray Research, 1997. Document Number IN-2153 2/97.

[2] S. Berson and R. Lindell. An architecture for advance reservations in the internet. Technical report. Work in Progress.

[3] Sharon Brunett, Karl Czajkowski, Steven Fitzgerald, Ian Foster, Andrew Johnson, Carl Kesselman, Jason Leigh, and Steven Tuecke. Application experiences with the Globus toolkit. In *Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 81–89. 1998.

[4] C. Catlett and L. Smarr. Metacomputing. *Communications of the ACM*, 35(6):44–52, 1992.

[5] Steve J. Chapin, Dimitrios Katramatos, John Karpovich, and Andrew Grimshaw.

[6] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. A resource management architecture for metacomputing systems. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82. Springer-Verlag LNCS 1459, 1998.

[7] Tom DeFanti and Rick Stevens. Teleimmersion. In *[12]*, pages 131–156.

[8] M. Degermark, T. Kohler, S. Pink, and O. Schelen. Advance reservations for predictive service in the internet. *ACM/Springer Verlag Journal on Multimedia Systems*, 5(3), 1997.

[9] A. Downey. Predicting queue times on space-sharing parallel computers. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[10] D. Ferrari, A. Gupta, and G. Ventre. Distributed advance reservation of real-time connections. *ACM/Springer Verlag Journal on Multimedia Systems*, 5(3), 1997.

[11] I. Foster and N. Karonis. A grid-enabled MPI: Message passing in heterogeneous distributed computing systems. In *Proceedings of SC'98*. ACM Press, 1998.

[12] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a Future Computing Infrastructure*. Morgan Kaufmann Publishers, 1999.

[13] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy. A distributed resource management architecture that supports advance reservations and co-allocation. In *Proceedings of the International Workshop on Quality of Service*, pages 27–36, 1999.

[14] Juern Gehring and Thomas Preiss. Scheduling a metacomputer with un-cooperative subschedulers. In *Proc. IPPS Workshop on Job Scheduling Strategies for Parallel Processing*. Springer-Verlag, 1999.

[15] GENIAS Software GmbH. CODINE: Computing in distributed networked environments, 1995. http://www.genias.de/genias/english/codine.html.

[16] A. Hafid, G. Bochmann, and R. Dssouli. A quality of service negotiation approach with future reservations (nafur): A detailed study. *Computer Networks and ISDN Systems*, 30(8), 1998.

[17] R. Henderson and D. Tweten. Portable Batch System: External reference specification. Technical report, NASA Ames Research Center, 1996.

[18] International Business Machines Corporation, Kingston, NY. *IBM Load Leveler: User's Guide*, September 1993.

[19] William Johnston. Realtime widely distributed instrumentation systems. In *[12]*, pages 75–103.

[20] A. Mehra, A. Indiresan, and K. Shin. Structuring communication software for quality-of-service guarantees. In *Proc. of 17th Real-Time Systems Symposium*, December 1996.

[21] P. Messina, S. Brunett, D. Davis, T. Gottschalk, D. Curkendall, L. Ekroot, and H. Siegel. Distributed interactive simulation for synthetic forces. In *Proceedings of the 11th International Parallel Processing Symposium*, 1997.

[22] Paul Messina. Distributed supercomputing applications. In *[12]*, pages 55–73.

[23] Reagan Moore, Chaitanya Baru, Richard Marciano, Arcot Rajasekar, and Michael Wan. Data-intensive computing. In *[12]*, pages 105–129.

[24] K. Nahrstedt, H. Chu, and S. Narayan. QoS-aware resource management for distributed multimedia applications. *Journal on High-Speed Networking, IOS Press*, December 1998.

[25] K. Nahrstedt and J. M. Smith. Design, implementation and experiences of the OMEGA end-point architecture. *IEEE JSAC, Special Issue on Distributed Multimedia Systems and Technology*, 14(7):1263–1279, September 1996.

[26] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *The 4th Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[27] Gregor von Laszewski, Ian Foster, Joseph A. Insley, John Bresnahan, Carl Kesselman Mei Su, Marcus Thiebaux, Mark L. Rivers, Ian McNulty, Brian Tieman, and Steve Wang. Real-time analysis, visualization, and steering of microtomography experiments at photon sources. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, 1999.

[28] L.C. Wolf and R. Steinmetz. Concepts for reservation in advance. *Kluwer Journal on Multimedia Tools and Applications*, 4(3), May 1997.

[29] S. Zhou. LSF: Load sharing in large-scale heterogeneous distributed systems. In *Proc. Workshop on Cluster Computing*, 1992.