Gathering at the Well: Creating Communities for Grid I/O *

Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny
Computer Sciences Department
University of Wisconsin - Madison
1210 West Dayton Street, Madison WI 53706

ABSTRACT

Grid applications have demanding I/O needs. Schedulers must bring jobs and data in close proximity in order to satisfy throughput, scalability, and policy requirements. Most systems accomplish this by making either jobs or data mobile. We propose a system that allows jobs and data to meet by binding execution and storage sites together into I/O communities *which then participate in the wide-area system. The relationships between participants in a community may be expressed by the ClassAd framework. Extensions to the framework allow community members to express indirect relations. We demonstrate our implementation of I/O communities by improving the performance of a key high-energy physics simulation on an international distributed system.*

1. INTRODUCTION

Grid applications have demanding I/O needs [3]. Applications in fields such as high-energy physics need high-throughput access to a wide selection of data files chosen from repositories measured in petabytes. Due to the large number of users, the size of the data, and the distances involved, online access to data repositories is neither scalable nor efficient for large numbers of jobs.

I/O systems that solve this problems have generally fallen into two camps: those that move the data to the job, and those that move the job to the data. Neither of these approaches is universally applicable, and both suffer from a scalability problem. Network and storage capacities limits both the number of replicas that may be made as well as the number of jobs that may use each replica.

We propose a balance, shown in Figure 1. In the local area, execution sites band together into *I/O communities* that share data within locally-determined physical limits. Each I/O community then hosts a storage appliance to serve that data both locally and within the existing wide-area replication system.

A scheduler may then make a number of informed choices. Jobs requesting particular data may be moved to communities where it is already staged. Or, data may be staged to the community in which a job has already been placed. Of course, the balance point between the two is not fixed. The ratio of supportable jobs to replicas depends on properties of the application, the data, the storage devices, and the networks.

In order to structure such communities, the participants must be able to express relationships between themselves. Some of these relations are direct – a job may require a machine with a particular CPU. Others are indirect – a job may require a machine associated with a storage device containing a particular dataset. We will demonstrate how the ClassAd framework, [25] with some additions for indirection, can be used to express these relationships.

Communities are traditionally constructed using distributed filesystems which usually require special privileges to deploy and configure. We will present building blocks that permit the construction of I/O communities from unprivileged, user-level software. These building blocks communicate their state to the

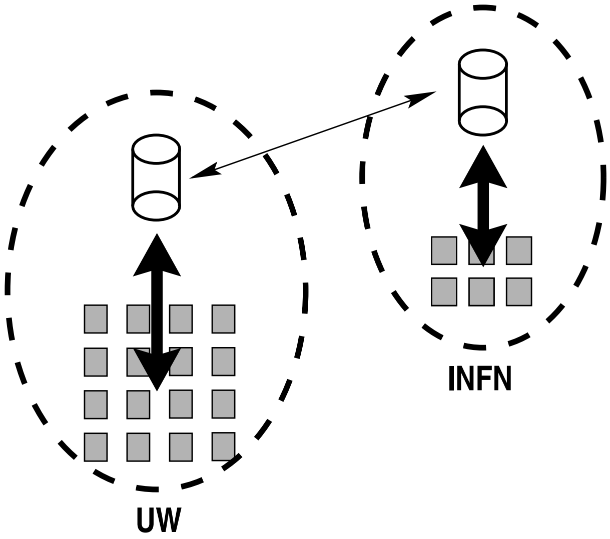

Figure 1: I/O Communities

*This research was supported in part by the NSF under contracts ITR-0086044 and EIA-9870684 and by NASA ARC under contract NCC 2-5323.

SC2001 November 2001, Denver

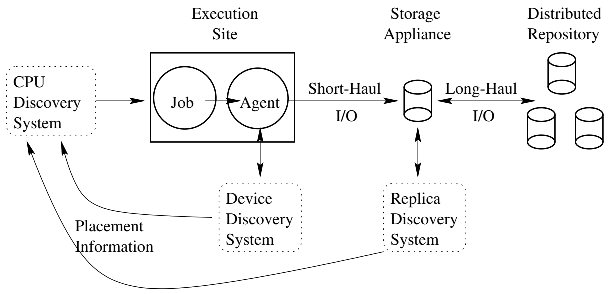

Figure 2: Model

Condor [20] distributed batch system, which then places jobs appropriately.

We will not address the matter of selecting policies for I/O communities. Policies may govern the size of communities, the contents of storage devices, and the decisions to relocate jobs from one place to another. Appropriate policies are wholly dependent upon the particulars of individual applications, physical network capacity, number of available CPUs, and the like. We hope that our work on mechanisms will enable a further study of appropriate policies.

That said, we will demonstrate the feasibility of this model by applying it to a key high-energy physics simulation run on an international grid. By constructing communities with sensible policies – given our knowledge of the application – we demonstrate a marked improvement in simulation capacity.

2. I/O COMMUNITIES

2.1 Building Communities

An I/O community consists of several CPUs that gather around a storage device. Programs executing at such CPUs are encouraged (or perhaps required) to use the community I/O device for storing and retrieving data. By sharing the device, similar applications may reduce their consumption of wide-area resources.

I/O communities may reflect both physical and administrative boundaries. The number of CPUs that may be effectively served by one storage device is limited by the connecting network and the load offered by running programs. The users admitted to a community may depend on membership in social structures.

The most familiar form of an I/O community is a distributed file system shared among members of a workgroup. This sort of community is semi-permanent – it may require special privileges and coordinated software changes between all the participants.

In contrast, services offered on the grid are intended to be flexible. As users, applications, and loads change, communities must be set up, reconfigured, and torn down.

To permit agile deployment of such communities, they should be constructed from building blocks that can be applied by normal users without special privileges. To accomplish this, we employ storage appliances and interposition agents, as shown in Figure 2.

A *storage appliance* serves as the meeting place for an I/O community. A storage appliance is frequently conceived as a specialized hardware device. However, a general-purpose computer equipped with the right software may serve equally well as a storage appliance.

The appliance is most useful if it speaks a number of protocols. This allows members of the community to select the most appropriate protocol for the situation. For example, an application selecting elements of a database should use a fine-grained block-access protocol. Conversely, an application processing large amounts of sequential data should choose a streaming protocol.

However, standard applications rarely speak such protocols. Although certain brave users may be willing to rewrite applications to work with new systems, there exists a large body of programs that cannot or will not be rewritten.

This problem is solved by *interposition agents*. An agent is a small piece of software that inserts itself between an application and the native operating system. The agent is responsible for converting a program's standard I/O operations into suitable actions on the I/O community. With an agent in place, unmodified applications can run in a grid environment.

2.2 Discovering Communities

In a computational grid, community resources come, go, and change without warning. In such an environment, programs must have rich methods for finding communities that meet their needs. Once placed, they must be able to determine their membership in a community. During execution, they may need to find resources outside of the community in order to bring them inside.

Each of these three actions is a different form of *discovery*. We refer to them as CPU discovery, device discovery, and replica discovery. Figure 2 shows where each form of discovery fits into an I/O community. Before execution, CPU discovery must be performed to find a CPU with the proper architecture, operating system, and so on. During execution, device discovery must be performed to find one's membership in a community. Also during execution, replica discovery may be performed to locate items outside of a job's immediate community.

Replica discovery has been an important area of research in recent grid efforts. [34, 28] Device discovery is closely related, but subtly different. It is frequently employed by self-configuring systems such as Jini [36] that locate storage or human-interface devices for mobile software and hardware. We would like to briefly comment upon the difference.

Replica discovery answers this question: *If my data is not in local storage, where can I get it?* Replica management systems track the various copies of datasets are they are spread to storage devices around a grid. When a dataset must be retrieved, a replica management system finds a suitable remote copy for the requestor.

Device discovery answers this question: *Where is my local storage device?* Once executing, jobs need to discover what device is willing to offer bandwidth and storage space for inputs, outputs, and temporary files. If a replica discovery system is used to locate remote data, the device discovery system must locate a place to put the incoming data so that the caller and other members of the community may find it.

Device discovery systems need not be complex to be useful. Below, we will define I/O communities simply by giving every execution site a *NearestStorage* propery that points to a storage appliance. This approach is simple and effective.

However, more complex systems may be imagined. An execution site may be associated with several storage appliances, each with its own policy restrictions. For example, a device may only allow access to members of an administrative group. In this case, the device discovery system must query the available devices and return the nearest device that accepts the job.

Indirection is a critical feature of any discovery system. In addition to querying direct properties of devices, a user might request a chain of relations. For example, a user may request to use any CPU associated with a storage device containing dataset x. It is not enough for the user to first look up all storage devices containing x, and then request the set of CPUs associated with any in the set – the situation may change without the user's knowledge. The user must be able to submit a request expressing the chain of indirection.

A language is needed to express all of these different relations. Scheduling and policy management systems need a concrete way to represent all of the properties, requirements, and preferences involved in an I/O community. The ClassAd language is uniquely suited to this task.

3. EXPRESSING COMMUNITIES WITH CLASSADS

ClassAds are currently used within the Condor system to describe properties, requirements, and preferences of participants in a distributed computing system. ClassAds are named after the classified advertisements found in newspapers, where multiple parties publish requests for service and offers to serve in a well-known place.

A single ClassAd is a list of (attribute,value) pairs. The values may be simple atoms such as strings or integers, or they may be complex expressions referring to potential matches.

Figures 4 and 5 shows how an example job and machine might be represented in this language. Each describes certain simple properties – the machine mentions its CPU and operating system, while the job mentions the name of the executable and owner. Both have requirements on a potential match. The machine will only accept jobs owned by a particular user, while the job will only accept machines running the correct operating system.

Unlike the newspaper, Condor provides a central matchmaking service that pairs offers with requests. When a suitable match is found, the two parties are informed and then become individually responsible for contacting each other and accomplishing work. This process is known as *bi-lateral matchmaking* and is described extensively by Raman et. al. [25]

To build I/O communities, we must add a third participant to the match – the storage appliance. As shown in Figure 3, an incoming job requests a CPU, but places indirect requirements on the associated storage. The ClassAd representing the CPU decides what storage is to be referenced.

Figure 4 shows how the job specifies an indirect reference. In the *Requirements* field, it states that it will only accept a job such that *NearestStorage.HasCMSData* is true. *NearestStorage* is evaluated in the context of each potential CPU.

The CPU cannot simply point to the nearest storage device by way of an address or a unique name. ClassAds are schema-free, so a single ad does not have a distinct name. Instead the

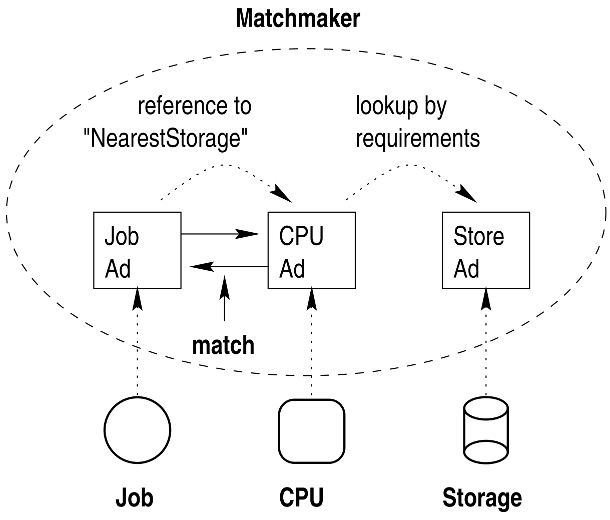

Figure 3: **Matchmaking with References**

Type	=	”job”
TargetType	=	”machine”
Cmd	=	”sim.exe”
Owner	=	”thain”
Requirements	=	(OpSys==”linux”)
		&& NearestStorage.HasCMSData

Figure 4: **Example Job ClassAd**

Type	=	”machine”
TargetType	=	”job”
Name	=	”raven”
OpSys	=	”linux”
Requirements	=	(Owner==”thain”)
NearestStorage	=	(Name==”turkey”)
		&& (Type==”storage”)

Figure 5: **Example Machine ClassAd**

Type	=	”storage”
Name	=	”turkey”
HasCMSData	=	True
CMSDataPath	=	”/cmsdata”

Figure 6: **Example Storage ClassAd**

NearestStorage property gives a set of constraints that identify a unique storage ClassAd, shown in Figure 6.

The rest of the reference expression may then be evaluated in the context of the referred-to ClassAd. So, each reference contained in the job ad evaluates as follows:

```
NearestStorage.HasCMSData = True
NearestStorage.CMSPath = "/cmsdata"
NearestStorage.Name = "turkey"
```

As the contents of the storage appliance change, it simply sends updated state to the matchmaker. If a dataset is added to a device, jobs that require it will match to the community. If a dataset is removed, jobs will no longer match.

Of course, information from the matchmaker is necessarily stale. The state of either a CPU or a storage appliance may change after a match has been made. Both sides have the responsibility of verifying that their requirements are still satisfied. This is done during a claiming protocol following a successful match.

3.1 Example Policies

By adding a level of indirection between the job and the storage, the user is freed from specifying *where* jobs must run. The user must simply state *what is needed* in order to execute their jobs. As the state of storage devices changes, jobs will run according to the user's policy.

Such policies are expressed at submit time in the ClassAd language. Each job has a boolean *Requirements* expression that determines which machines are suitable execution sites. If it evaluates to *True*, then the execution site is accepted, otherwise it is rejected. An integer expression *Rank* gives a value to all potential matches. Given several machines for which *Requirements* evaluates to *True*, the machine with the highest *Rank* will be chosen. With these expressions, we may control whether jobs move to data or wait for it to arrive.

For example, if the user is willing to let the job move to any site that already has a particular dataset, then she may express this:

```
Requirements =
 (NearestStorage.HasCMSData)
```

On the other hand, if the user knows that moving the job is an expensive operation, then she may require it stay in a particular community:

```
Requirements =
 (NearestStorage.Name
  == "turkey.cs.wisc.edu")
```

If she simply *prefers* to run in the local community, but does not require it, she *Rank*s the local community at ten, and others at zero.

```
Requirements =
 (NearestStorage.HasCMSData)
Rank =
 (NearestStorage.Name
  == "turkey.cs.wisc.edu")
 ? 10 : 0
```

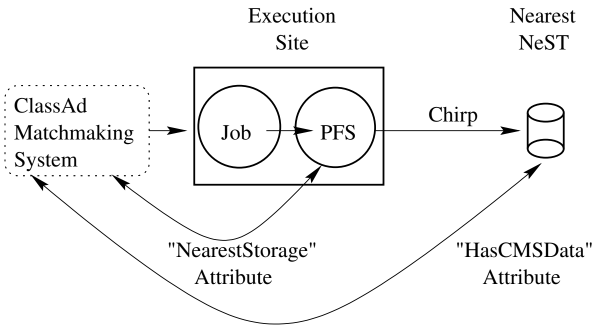

Figure 7: Implementation

Of course, some machines are better than no machines. If the user is willing to execute anywhere, and use remote I/O when a local copy is not available, then she may eliminate the *Requirements* statement above.

More complicated information may be included in either expression to set the policy under which migration is permitted. For example, the job may be required to execute in a particular I/O community except during the night, when network traffic may be lower:

```
Requirements =
 (NearestStorage.Name==
  "turkey.cs.wisc.edu")
 || (ClockHour<7) || (ClockHour>18)
```

4. IMPLEMENTATION

We have built a prototype of these concepts within the Condor distributed batch system. Condor itself provides the CPU scheduling system and the ClassAd framework. An interposition agent, the Pluggable File System, is used to attach jobs to the local storage appliance, implemented with software called NeST. Each of these devices are sufficiently general purpose that they can be put to use individually or together within other systems.

NeST [7] is software for creating general-purpose storage appliances on commodity computers without special privileges. Externally, it supports a variety of network protocols, allowing applications to choose the most appropriate way to interact with storage. We have made use of two in particular, GridFTP [2] and Chirp. The former provides strong authentication and high-throughput transfers using a variety of techniques such as multiple TCP streams. The latter is the native NeST protocol, and provides simple RPC-like partial-file access on a single TCP connection. We have used GridFTP as the *lingua franca* for communicating with other grid services over long-haul connections. We have used Chirp for short-haul partial-file access, as it does not require the overhead of a new TCP connection for every data operation.

PFS [31] is an interposition agent constructed with Bypass [32, 33]. PFS adapts legacy applications to new storage systems by 'mounting' them in the application's view of the file system. No special privileges or kernel-level changes are required. A number of standard network protocols, including GridFTP and Chirp, are supported. For example, with PFS loaded, unmodified UNIX programs may be used to interact with a NeST running on `turkey.cs.wisc.edu`:

```
% vi /chirp/turkey.cs.wisc.edu/my_file
```

The system needs a way of getting the CPU's selection of an I/O device into the parameters of the application. Condor allows a ClassAd property of a job or execution site to be inserted into a program's environment variables or arguments at run time by macro-expanding expressions beginning with two dollar signs. For example:

```
Arguments =
 "/chirp/$$(NearestStorage.Name)/input.data"
```

Condor currently understands an executable to consist of a single file. To submit a PFS-enabled application to Condor, we must resort to a little trick of submitting a self-extracting archive containing the application, PFS, and a script to invoke the two properly. We may take this level of indirection one step further by omitting the application from the archive, and modifying the script to fetch the executable from the I/O community using PFS. We will use this technique below to retrieve a common executable from the local appliance.

Finally, we have noted above that users of ClassAds must be prepared to handle a stale match. Suppose that stale information causes a job to match to a community that no longer has the needed dataset. PFS will discover a "file not found" error as it performs I/O to the nearest NeST. Simply passing the error to the application is incorrect – this would likely cause it to exit normally with an error message, forcing the user to manually understand the error and resubmit the job. The correct action taken by PFS is to cause the application to exit abnormally with the "kill process" signal. Condor interprets this signal to mean "execution aborted," and will re-queue the job for another execution attempt.

5. PERFORMANCE

To demonstrate our implementation, we have chosen to examine the simulation component of the CMS experiment to be performed at CERN. The large I/O needs of this experiment have been well documented [3]. Users in Italy and the United States make heavy use of this application in Condor pools at the Istituto Nazionale di Fisica Nucleare (INFN) and the University of Wisconsin (UW.)

We began by assuming the role of a scientist at INFN that wishes to execute a large number of instances of the simulation. Although the INFN pool is equipped with a fair number of CPUs, competition between users of the pool limits us to the use of about thirty at once. How can the additional CPUs at UW be leveraged? We explored the deployment of I/O communities in order to solve this problem.

5.1 Application

Viewed from the perspective of the system, the CMS simulation works as follows. It reads an input file of several KB, and following its instructions, reads a variety of files from a 'database' directory. The database is provided with the application and consists of a mixture of input files, data files, libraries, and source files.

Although a user might conceivably determine the exact set of database files needed by a particular run of this simulation, our experience is that few care to, citing the cost of analysis as more expensive than dealing with the data. The files needed are not trivially predictable from the input. For the sake of this application, we assume that an arbitrary simulation run needs access to the entire directory.

We trimmed the libraries and source from the database, yielding a directory of 303 MB, containing 54 directories, 33 symbolic links, and 432 files. [1]

We chose a sample run of the simulation that uses a 2.5 KB input file, reads a total of 1.5 MB of input from 20 files in the database directory, and generates .97 MB of output in three files. The executable is 17 MB, but compresses to 5.4 MB for network transfer. On a 600 MIPS machine using only local storage, the sample runs for 160 seconds.

The simulation executable was not directly submitted to the system. Instead, a 1.2 MB self-extracting archive containing PFS and a script were submitted. At the execution site, the script downloaded the simulation executable from the appropriate storage appliance and invoked it with the appropriate arguments.

The sample run is not entirely representative of the real CMS needs – it has a *higher* I/O to CPU ratio than a real run. Typically, a simulation runs several hours, not several minutes. We have chosen this shorter run for two reasons. Primarily, we want to push the envelope of the I/O system, and open the use of Condor applications with ever greater I/O demands. Secondarily, we did not want to consume excessive amounts of resources that would otherwise be allocated toward real simulations currently in progress.

5.2 Environment

Two Condor pools, one at INFN and one at UW, were employed in running simulations. Each pool was configured as a distinct I/O community.

The INFN Condor pool consisted of 236 CPUs, of which about 30 were available to us at any time. The processing power of the various CPUs ranged from 100-1200 MIPS, and the available memory ranged from 60-500 MB. The CPUs were physically spread around the country at the various departments of the institution. A workstation providing 750 MIPS and 378 MB of memory was established in Bologna as the storage appliance for the INFN community. A variety of networks ranging from 10Mb/s to 100Mb/s connected the execution sites to the storage appliance.

The UW Condor pool consisted of 911 CPUs, of which 100 were reserved for our use. Each of the reserved CPUs provided 600 MIPS and 512 MB of memory. An identical machine was established as the storage appliance for the UW community. The reserved machines were connected with the appliance via a dedicated 100 Mb/s ethernet switch.

The two communities were connected via the public Internet. The bandwidth available on the path between varied from 0.2 MB/s to 1.0 MB/s with a latency of 150 ms.

5.3 Measurements

We began by assuming that the necessary executables and data files are stored on a workstation at INFN. On this workstation, we installed an instance of Condor for submitting jobs, and an instance of NeST to serve input data and provide output space.

[1] When moving the collection from site to site, reproducing the symbolic links is important, otherwise, the archive size swells to 543 MB.

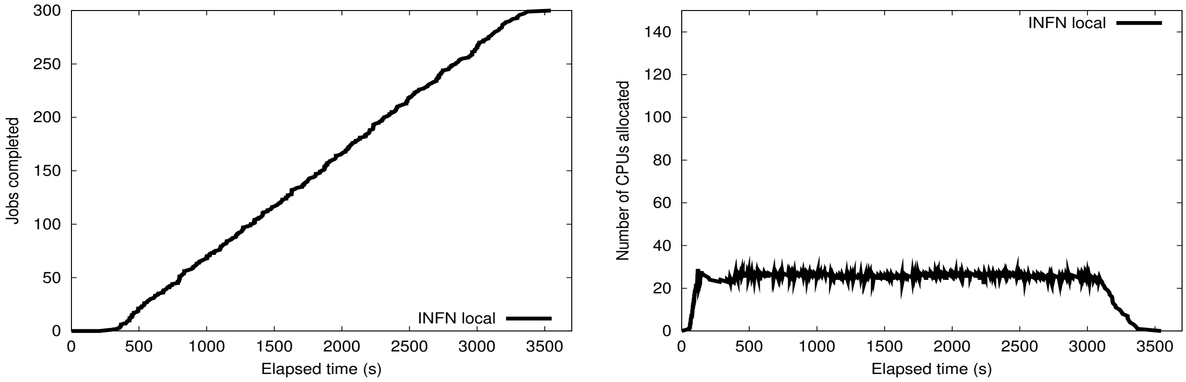

Figure 8: **INFN Community:** ***This figure details the execution of 300 batch jobs running in the INFN community only. The left-hand graph shows the number of jobs completed as time progresses. The right-hand graph shows the number of CPUs in use as a function of time.***

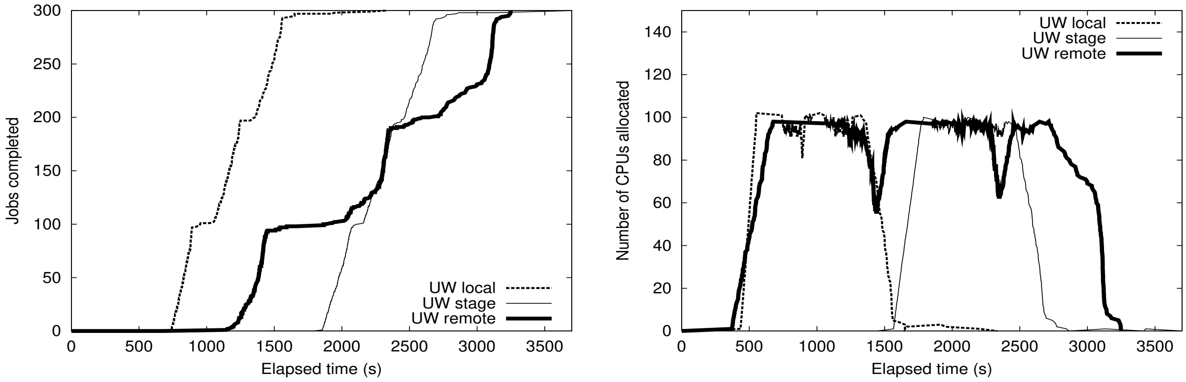

Figure 9: **UW Community:** ***This figure details the execution of 300 batch jobs running in the UW community only. The left-hand graph shows the number of jobs completed as time progresses. The right-hand graph shows the number of CPUs in use as a function of time. Each line represents a different I/O discipline in use at UW. Note the anomalies in the 'UW remote' discipline at elapsed times 1500 and 2400 seconds. This is an artifact of the number of available machines. Because the bandwidth to the storage appliance is shared fairly, jobs finish at approximately the same time and a necessary delay is incurred until the next jobs start to complete. These anomalies are present in the other disciplines as well but are less pronounced due to the decreased time necessary to fetch the executable and input files.***

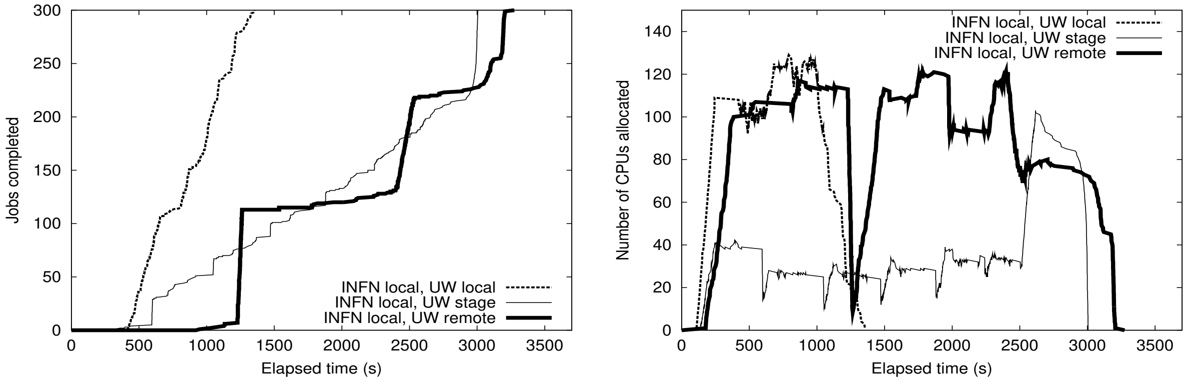

Figure 10: **Both Communities:** ***This figure details the execution of 300 batch jobs running simultaneously in both communities. The left-hand graph shows the number of jobs completed as time progresses. The right-hand graph shows the number of CPUs in use as a function of time. Each line represents a different I/O discipline in use at UW. The anomalous behavior of the 'INFN local, UW remote' lines are due to the same phenomenon as described in Figure 9.***

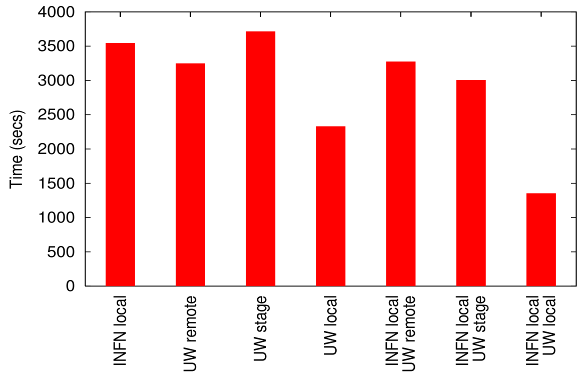

Figure 11: **Overall Completion Time:** ***This graph shows the overall execution time for 300 simulations in each configuration. Lower values indicate a better turnaround time from the user's perspective.***

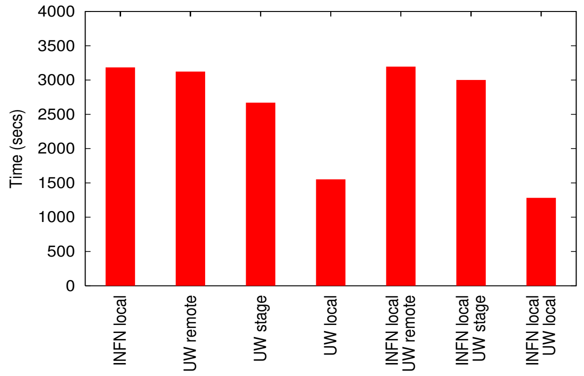

Figure 12: **Ninety-Five Percent Completion Time:** ***This graph shows the 95 percent completion time for 300 simulations in each configuration. Lower values indicate a better response time from the user's perspective.***

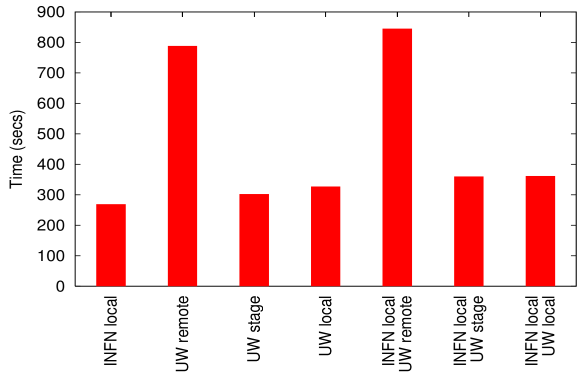

Figure 13: **Average CPU Consumption:** ***This graph shows the average CPU time consumed by jobs in each configuration. Lower values indicate a more efficient use of resources from the system's perspective.***

To benchmark the capacity of the INFN pool, we submitted 300 simulation jobs. They completed as shown in Figure 8. Throughout the run, the number of available machines fluctuates. This is an inevitable property of a distributed system involving hundreds of users with changing minds and machines. Nevertheless, we make steady progress, roughly one job every 10 seconds.

Next, we explored the feasibility of using the many available CPUs at UW. With 100 CPUs reserved for our use, we submitted 300 jobs to the UW pool in three different configurations. The results are shown in Figure 9.

In the case labelled 'UW remote', the jobs performed their I/O against the appliance at INFN. Despite the much larger number of CPUs, the run was no faster, as the jobs were constrained by the very small available bandwidth.

To address this situation, we deployed a NeST at the UW pool on the same switch as the reserved CPUs. The CPUs were then updated to advertise themselves as members of the same community, and 300 jobs were submitted with the constraint that they run at UW, and only where the CMS data is available. Of course, no machines immediately satisfied this requirement. To satisfy them, we manually staged the necessary data to the appliance and instructed it to advertise its contents. This procedure took roughly 1300 seconds, after which jobs were able to run and completed as shown in the 'UW stage' case.

Accounting for the time necessary to transfer, the 'UW stage' case was only marginally faster than 'UW remote' case. However, future executions would be able to take advantage of the already-transferred data. Such a run is shown in the 'UW local' case.

A fourth configuration was also measured but is not shown in Figure 9. In this configuration, we combined the 'UW stage' and the 'UW remote' models, and performed the stage of the database while concurrently allowing jobs to execute via remote I/O. Our idea was that when the staging operation would complete, then the jobs could access their data locally. However, the bandwidth to the remote server was then shared between the stage operation and the jobs and all of the jobs finished before the stage completed. Additionally, due to this bandwidth contention, the jobs finished even more slowly than in the 'UW remote' configuration.

We should note that not all of the I/O was done in the local community. Although the executable and database files were fetched from the local storage appliance, the process-specific input files were syncronoulsy fetched from the storage appliance at the submission site and the output files were also delivered there.

Finally, we made use of the two communities in concert. These results are shown in Figure 10.

In the 'INFN local, UW remote' case, jobs were run in both communities while performing I/O against the appliance at the submitting host. Although the number of CPUs in use was high, bandwidth constraints limited performance.

In the 'INFN local, UW stage' case, jobs were run in the INFN community while data was staged to UW as described above. The stage completed near the end of the run, whereopon a large number of UW CPUs finished off the remaining jobs, yielding a brief increase in performance.

Finally, with the data available in either community, a third run of 300 jobs would be able to match in either community, as shown in the 'INFN local, UW local' case.

5.4 Evaluation

We may evaluate the various configurations from two points of view. Users are generally concerned with the overall completion time of any workload, while system operators are generally concerned with the efficient use of resources consumed.

The user's perspective is summarized in Figure 11. This graph shows the completion time of the 300th job in each configuration. In general, applying more CPUs to a run yields faster results. However, the larger numbers of CPUs available at UW only provide marginal improvement when used remotely. Localized I/O yields faster results.

Figures 8, 9, and 10 show that each configuration completes a large fraction of jobs quickly. The overall completion of some are delayed by a small number of jobs at the very end of execution. We may compare the configurations while disregarding the contributions of the long tail by examining the completion time of ninety-five percent of the jobs, shown in Figure 12.

We examined jobs in these long tails and discovered three distinct sources of delay. In a few cases, jobs were starved for I/O in the input phase and did not enter the computation phase until I/O competition decreased. In others, jobs late in the run were evicted from execution sites by owners returning to their workstations. A few jobs simply had longer execution times due to competition with local users for CPU, memory, and network capacity.

The problem of starvation suggests the need for an examination of fairness in the storage appliance. However, the latter two problems are more difficult to address. Although they could be eliminated in a tightly-controlled environment, they are an ever-present feature in large-scale grid computing. Any large computation performed using resources that are partially shared is likely to receive interference in performance from other users. The long tail might be prevented by executing multiple copies of jobs when the number outstanding is less than the number of CPUs available.

The operator's perspective is summarized in Figure 13. This graph shows the average CPU consumption per job in each configuration. Each figure was arrived at by dividing the allocated CPU time by the number of jobs in each run. The most efficient configurations involve localized I/O. Although remote I/O provides some improvement in completion time, it holds CPUs idle while waiting for I/O, yielding a poor efficiency.

Of course, the performance of each configuration changes with the parameters of the runs. For example, the 'stage' cases only provide an improvement when the time necessary to transfer the datasets is less than the execution time of the jobs performing remote I/O.

A few details of the execution were surprising.

In Figures 9 and 10, the 'remote' cases incur several dramatic delays when the frequency of job completion drops drastically. These are reflected by corresponding drops in CPU allocation. These cases occurred when large numbers of jobs, previously contending for I/O, completed at once. An examination of the submitter's logfile shows that Condor was not able to start new jobs as quickly as old ones completed. This is due to the overhead of re-transferring the self-extracting archive for every newly started job. Although it could be alleviated by a cache at the execution site, it does not appear to be a major obstacle to throughput for this application.

6. RELATED WORK

Many simple distributed I/O systems make use of a centralized server to connect jobs with data. The canonical example is of course the Network File System (NFS) [29]. An analogue in grid computing is the Condor [20] remote system call [19] facility, in which each running job performs remote procedure calls [9] back to the orginating computer. Both of these central-server models have limited scalability, because the number of clients is limited by the aggregate bandwidth provided by the central server. The performance of individual clients may also be limited by the bandwidth or the latency of the network. The reliability of the whole system decreases as the number of networks and participants increases.

Several systems address these difficulties by copying data to the site of job execution. In so doing, the Andrew File System (AFS) [15] is able to scale to a larger client/server ratio than that of NFS. An analogue in grid computing is the Globus GASS system [8], in which whole files are fetched from distributed repositories at first reference and stored locally until they are no longer referenced. Hierarchical data grids [3] expand this idea into trees of servers that replicate data from a production site.

Whether jobs or data are moved is orthogonal to the question of how the data is located. We should note that although we have described a system which matches data and jobs that data is just a type of resource. Many applications will require not just the discovery of data but also of more arbitrary types of resources as well.

A replica management system [34] can keep track of all of the data and their locations. The Storage Resource Broker (SRB) [5] pulls many of these pieces together to provide a coherent view of multiple replication sites. Our arrangement of I/O communities is also very similar to that of a shared web proxy cache [37]. However, web clients are fixed to a particular location, and do not have the option of choosing the best proxy behind which to run.

There is a large body of research about, and available software for general resource discovery. Some of these projects include JINI [36], replica catalogs [34], LDAP [39], SNMP [10], and even some of the more recent peer-to-peer file sharing protocols such as Napster [21] and Gnutella [13]. The advantage of the ClassAd framework [20, 25, 26, 24] within Condor is its unique ability to integrate resource discovery with scheduling.

ClassAds have been used for resource discovery in several contexts.

Vazhkudai et. al. [35] describe how ClassAds may be used to match jobs with storage devices. In this model, a replica manager is first consulted to discover the list of available replicas, and then matchmaking is performed to find which the best storage device. The job is then submitted for execution while bound to the discovered device. It is assumed changes in the distribution of replicas will not change after the lookup.

Basney et. al. describe the use of ClassAds in *execution domains* [6]. In this model, execution sites bind themselves to checkpoint servers. Jobs write checkpoints to the nearest available server, and then express a policy controlling how far they are willing to migrate from the last checkpoint image.

Our contribution to ClassAds is to introduce indirection. In our model, jobs express constraints on storage devices, but allow each execution site to declare its binding to storage. The storage ad is referred to, but does not become a member of the match – it is not promised exclusively to the requesting job.

In contrast, *gang-matching* of ClassAds, also described by Raman, et al., [26] allows multiple entities to be exclusively promised to each other. An example of this is an arrangement in which an organization has a limited number of licenses to run some proprietary software. In such a case, gangmatching could match licenses, machines and jobs and thereby ensure that licensing agreements are not violated.

A variety of research ventures are exploring storage devices under various names, such as NASD [12], Active Disks [27], Flash [22], IBP [23] and buffer servers [4]. Some commercial vendors such as NetApp [17] and EMC [11] also offer storage servers as a hardware package [14]. We are making use of NeST [7], because it is an easily deployable software only appliance that speaks protocols suitable for grid computing and can run without special privilege.

A wide variety of mechanisms for building interposition agents have been proposed, including system call interception [1, 18], static relinking [19], binary rewriting [40, 16] and emulation through an existing interface [38]. We are making use of Bypass [32, 33] due to its low overhead and ability to be used without special privileges.

7. CONCLUSION

Communities are natural structures for localizing application I/O on the grid. By binding CPUs and storage together into organizations that reflect the physical reality, we may increase the performance of applications and the utilization of systems.

Users need the ability to express relations between participants in a community. In particular, indirect relations allow the user to express requirements on the storage associated with a CPU. The ClassAd framework, with some extensions for indirection, is well-suited for describing and managing such communities.

By employing several general-purpose building blocks – Condor, NeST, and PFS – we have demonstrated the easy deployment of I/O communities without special privileges. By deploying a reasonable configuration, we have improved the throughput of a high-energy physics simulation.

We see several avenues for future work.

Currently, the configuration of communities is left to a human. However, the appropriate ratio of CPUs to storage appliance depends on offered loads as well as physical constraints. We envision that higher-level software may reconfigure communities by deploying or removing storage appliances as load changes.

Given a static set of communities, the user may also find it difficult to choose an appropriate policy. Should jobs move to data, or vice versa? Our mechanisms admit both possibilities, but do not select or trigger such moves.

Our current staging mechanism allows only complete transfer of the necessary data files. In the future we would like to investigate different caching policies that might allow a finer granularity of data transfer. For instance, files in the dataset could be demand fetched and cached at the local storage appliance. This is similar to the configuration we tested in which jobs execute remotely during the complete stage of the dataset. In that case, we found the performance to be very low due to the bandwidth contention between the executing jobs and the stage operation. However, if the level of file sharing is sufficiently high then demand caching may well outperform staging the data.

Finally, we have concentrated on the problems of delivering input data. Other efforts in the Condor research group, such as Kangaroo [30], address the problems of reliably moving output data to a distant destination. This data movement is done asyncronously and allows remotely executing jobs to vacate their execute machines more quickly. A combination of Kangaroo with I/O communities would be able to address the I/O needs of grid applications from beginning to end.

8. ACKNOWLEDGEMENTS

We would like to thank Vladimir Litvin for assistance and advice with the CMS software, and Paolo Mazzanti for offering a dedicated NeST machine and generally making the INFN Condor pool available to us.

9. REFERENCES

[1] Albert Alexandrov, Maximilian Ibel, Klaus Schauser, and Chris Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[2] Bill Allcock, Joe Bester, John Bresnahan, Ann Chervenak, Ian Foster, Carl Kesselman, Sam Meder, Veronika Nefedova, Darcy Quesnel, and Steven Tuecke. Secure, efficient data transport and replica management for high-performance data-intensive computing. Submitted for publication.

[3] William Allcock, Ann Chervenak, Ian Foster, Carl Kesselman, Charles Salisbury, and Steve Tuecke. The data grid: Towards an architecture for the distributed management and analysis of large scientific datasets. To appear in the *Journal of Network and Computer Applications*, 2001.

[4] D. Anderson, K. Yocum, and J. Chase. A case for buffer servers. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, April 1999.

[5] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[6] Jim Basney, Miron Livny, and Paolo Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. Submitted to Computer Physics Communications, 2001.

[7] John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. NeST project. http://www.nestproject.com, 2001.

[8] Joseph Bester, Ian Foster, Carl Kesselman, Jean Tedesco, and Steven Tuecke. GASS: A data movement and access service for wide area computing systems. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 78–88, 1999.

[9] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Februrary 1984.

[10] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol. Request For Comments 1157, Internet Engineering Task Force, 1990.

[11] EMC Corporation. http://www.emc.com.

[12] Garth A. Gibson, David P. Nagle, Khalil Amiri, Fay W. Chang, Eugene Feinberg, Howard Gobioff Chen Lee, Berend Ozceri, Erik Riedel, and David Rochberg. A case for network-attached secure disks. Technical Report CMU-CS-96-142, Carnegie-Mellon University, 1996.

[13] Gnutella. http://gnutella.wego.com.

[14] D. Hitz. A storage networking appliance. Technical Report TR3001, Network Appliance, Inc., 2000.

[15] J.H. Howard, M.L. Kazar, S.G. Menees, D.A. Nichols, M. Satyanarayanan, R.N. Sidebotham, and M.J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systmes*, 6(1):51–81, February 1988.

[16] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, February 1999.

[17] Network Appliance Inc. http://www.netapp.com.

[18] Michael B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM symposium on operating systems principles*, pages 80–93, 1993.

[19] M. J. Litzkow. Remote UNIX: Turning Idle Workstations into Cycle Servers. In *Proceedings of the 1987 Usenix Summer Conf.*, pages 381–384, 1987.

[20] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, 1988.

[21] Napster. http://www.napster.com.

[22] V. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable web server. In *Proceedings of the Usenix Technical Conference*, 1999.

[23] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[24] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, October 2000.

[25] Rajesh Raman, Miron Livny, and Marvin Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, July 1998.

[26] Rajesh Raman, Miron Livny, and Marvin Solomon. Resource management through multilateral matchmaking. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 290–291, Pittsburgh, PA, August 2000.

[27] E. Riedel and G. Gibson. Active disks - remote execution for network-attached storage. Technical Report CMU-CS-97-198, Carnegie-Mellon University, 1997.

[28] Asad Samar and Heinz Stockinger. Grid Data Management Pilot. To appear in *IASTED International Conference on Appliaed Informatics (AI2001)*, February 2001.

[29] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conferece*, pages 119–130, 1985.

[30] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, San Francisco, California, August 2001.

[31] Douglas Thain and Miron Livny. The Pluggable File System. http://www.cs.wisc.edu/condor/pfs.

[32] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, Pittsburg, PA, August 2000.

[33] Douglas Thain and Miron Livny. Multiple bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[34] Sudharsan Vazhkudai, Steve Tuecke, and Ian Foster. Replica selection in the Globus data grid. In *Proceedings of the International Workshop on Data Models and Databases on Clusters and the Grid*, 2001.

[35] Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the globus data grid, May 2001.

[36] Jim Waldo. The Jini architecture for network-centric computing. *Communications of the ACM*, 42(7):76–82, 1999.

[37] Jia Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, 29(5):36–46, October 1999.

[38] B. White, A. Grimshaw, and A Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the 9th IEEE Symposium on High Performance Distributed Systems*, August 2000.

[39] W. Yeong, T. Howes, and S. Kille. Lightweight directory access protocol. Request For Comments 1777, Internet Engineering Task Force, March 1995.

[40] Victor C. Zandy, Barton P. Miller, and Miron Livny. Process hijacking. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, 1999.