Utilizing Widely Distributed Computational Resources Efficiently with Execution Domains

Jim Basney Miron Livny

University of Wisconsin-Madison, USA

Paolo Mazzanti

INFN and University of Bologna, Italy

Abstract

Wide-area computational grids have the potential to provide large amounts of computing capacity to the scientific community. Realizing this potential requires intelligent data management, enabling applications to harness remote computing resources with minimal remote data access overhead. We define *execution domains*, a framework which defines an affinity between CPU and data resources in the grid, so applications are scheduled to run on CPUs which have the needed access to datasets and storage devices. The framework also includes *domain managers*, agents which dynamically adjust the execution domain configuration to support the efficient execution of grid applications. In this paper, we present the execution domain framework and show how we apply it in the Condor resource management system.

Key words:
PACS: 07.05.Bx, 89.80.+h
Cluster Computing; Computational Grids; Network Scheduling; Remote I/O; Checkpointing; High Throughput Computing; Condor

1 Introduction

Computational grids (1; 2) have the potential to deliver large amounts of computing capacity to the scientific community by federating the computing resources on the network for large experiments (3). Pooling resources into a computational grid enables resource sharing: a scientist can harness computing resources owned by others when they are not using them. Many large computing experiments operate in cycles, where researchers plan an experiment and collect input data before beginning the computation. When the computation completes, the researchers analyze the results of the computation and plan their next experiment. Sharing computing resources naturally increases the computing capacity available to all participants, since one research project may use many computing resources while other projects are in the planning or analysis stages.

Intelligent data management is central to the success of a computational grid (4). The Particle Physics Data Grid [1] and the CERN Data Grid [2] are two recent efforts to address the challenges of supporting data-intensive Physics applications on the grid. Data-intensive grid applications access large input datasets, produce large volumes of output, and generate large intermediate data files (for checkpointing or out-of-core computation). If these applications naively perform I/O to remote storage devices, they may spend a significant proportion of their run time waiting on the network (5). We have been engaged in a collaborative effort to deploy the Condor resource management system at INFN sites across a wide-area network (6). As part of this effort, we have developed a scheduling framework for CPU and data resources called *execution domains*. The framework defines an affinity between CPU and data resources, so applications are scheduled to run on CPUs which have the needed access to datasets and storage devices. The framework also includes *domain managers*, agents which dynamically adjust the execution domain configuration to support the efficient execution of grid applications. In this paper, we describe the execution domain framework and how we apply it in the Condor resource management system.

2 Execution Domains

Execution domains ensure that applications which require a given level of access to a data resource run on CPUs which can provide the needed access. If an application's input data is stored on a network file server, execution domains ensure that the application runs only on CPUs with reliable, high-speed access to that network file server. For applications which produce large volumes of output, execution domains ensure that the application runs only on CPUs with access to sufficient storage capacity for the application's output. If the application produces large intermediate state (for checkpointing or out-of-core computation), execution domains ensure that, once the application begins its execution using a storage device, it migrates only among CPUs with high-performance access to that storage device. Since data resources may not be accessible from all CPUs in a wide-area grid for system administration and

[1] `http://www.phys.ufl.edu/~avery/mre/`

[2] `http://nicewww.cern.ch/~ingoa/DataGrid.html`

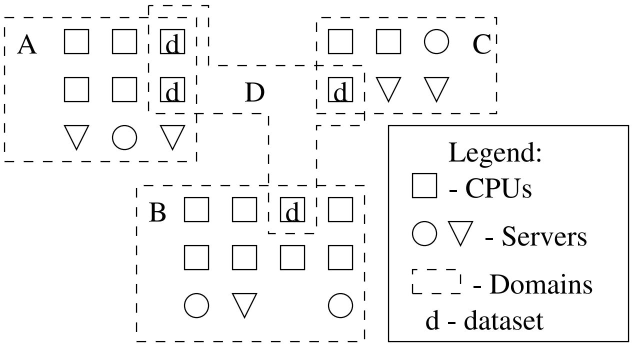

Fig. 1. Example Execution Domain Configuration

security reasons, execution domains are useful even for applications which can tolerate low data rates, to ensure that the application only runs on CPUs with some type of access to the application's data. Also, an execution domain may be empty if a dataset is not yet available. Applications which require the dataset will not run until it is produced. So, execution domains also serve as a mechanism for controlling data dependencies between grid applications.

An execution domain is a set of CPUs with a defined level of access to a data resource. Levels of access to a data resource may be defined according to performance, network distance, reliability, security, or other criteria. Execution domains may be defined at different access levels for a given data resource. However, a CPU is a member of at most one execution domain for each data resource. A data resource may be a *server* or *dataset*. A server is any data storage device, including network file servers and database servers. Servers occupy a fixed position on the network and can not be easily relocated. A dataset is a file or set of files stored on a server. Datasets may be dynamically replicated and migrated between servers on the network. Figure 1 illustrates an example execution domain configuration. Domains A, B, and C are each defined by their proximity to a set of data servers. As illustrated, a logical execution domain may include multiple physical servers. Domain D, which overlaps with the other domains, is defined by copies of a dataset staged on the local disks of four CPUs.

Domain managers are responsible for defining execution domain membership and maintaining the affinity between applications and the execution domains of their data. We envision the implementation of two types of domain managers. The first type of domain manager is a *domain migration agent*. The migration agent schedules the initial placement of application data files on file servers and ensures that applications run within the execution domains of the file server(s) which store their files. The agent also monitors the demand for and availability of CPUs in the server domains. If there is an insufficient number of CPUs in the execution domain of a file server, the domain migration agent can transfer the job's data files from the current file server to a server in a domain where more CPUs are available. The agent can then modify the job's domain requirements so it will run in the new execution domain. The second type of domain manager is a *data staging agent*. The data staging agent schedules the initial distribution of datasets in the grid and configures the execution domain membership for those datasets. This agent also monitors the demand for datasets in the grid by watching the job queues. If there are an insufficient number of CPUs in the execution domain of a dataset to meet current demand, the data staging agent expands the execution domain. The agent stages a copy of the dataset on additional storage devices and injects the dataset attributes into the resource offers of the CPUs in the domains of those storage devices. Since domain managers are external to the grid scheduler, they can be used to implement scheduling policies or algorithms which were not anticipated by the scheduler's developers. For example, the domain managers can use network load information to explicitly schedule data transfers over wide-area links, for staging and migration.

3 Execution Domains in Condor

The Condor ClassAd Matchmaking framework (7) enables easy implementation of execution domains with no changes to the Condor resource management system. ClassAd Matchmaking gives us the ability to dynamically inject information into the system to achieve custom scheduling goals. The ClassAd resource description language encodes requests and offers for computing resources in the grid. This language uses a semi-structured data model, so there is no fixed schema for the representation of resource requests and offers. Each resource request or offer contains a set of attribute definitions which describe the request or the offered resource. They each also contain a *Requirements* expression which specifies the compatibility between requests and offers and a *Rank* expression which indicates preferences.

Domain managers can use the Condor APIs to modify the execution domain definitions and application requirements. To modify the domain definitions, the domain managers can insert, delete, or modify attributes in resource offers. When the domain staging agent stores a dataset on the local disk of a workstation, it inserts a new attribute for that dataset in the workstation's resource offer. When the domain migration agent moves an application's data files to a new file server, it modifies the *Requirements* and *Rank* expressions in the application's resource request so the application will execution in the domain of the new file server.

For example, the following resource offer describes a Sparc Solaris workstation with 256 MB of memory and a MIPS rating of 200:

OpSys = "Solaris2.6";

Arch = "Sun4u";
Memory = 256;
Mips = 200;

To include a workstation's CPU in an execution domain, the domain manager inserts an attribute into the resource offer. For example, if this workstation has local access to the cs.wisc.edu AFS network filesystem, the manager inserts the following attribute:

AFSDomain = "cs.wisc.edu";

Resource offers with different values defined for the *AFSDomain* attribute describe CPUs in different execution domains. The domain manager also inserts a Boolean attribute for each dataset to indicate that the CPU is a member of the execution domain of that dataset. For example:

HasDataSetXYZ97S3 = True;

The dataset may be staged differently for different CPUs. It may, for example, be located on the local disk of some CPUs and available to other CPUs via a network file server. If the applications which require this dataset have different I/O characteristics, it is useful to define more restrictive execution domains. The domain manager can define an execution domain which includes only those CPUs with the dataset staged on the local disk as follows:

HasDataSetXYZ97S3Locally = True;

So, applications which need local-disk access speeds to this dataset should run only in this more restrictive execution domain. The Condor remote I/O library can be used to hide the different file access methods from the application. The library instruments the application's I/O system calls and redirects them to the appropriate file access method. Support for many I/O access protocols in the Condor remote I/O is in progress, including FTP, HTTP, and GASS (8).

An application's resource request indicates its requirements and preferences for an execution site. For example, the resource request below is compatible with the resource offer above. The request asks for a Sparc Solaris workstation with more than 80 MB of memory, with a preference for the CPU with the highest MIPS rating (i.e., CPUs are ranked in descending order by their Mips value):

Requirements = (Target.OpSys == "Solaris2.6") &&
    (Target.Arch == "Sun4u") && (Target.Memory > 80);
Rank = Mips;

To indicate that the application stores its data files in the cs.wisc.edu AFS filesystem and so should run in the execution domain of that filesystem for best performance, the domain manager modifies the *Requirements* in the application's resource request:

Requirements = (Target.OpSys == "Solaris2.6") &&
(Target.Arch == "Sun4u") && (Target.Memory > 80) &&
(Target.AFSDomain == "cs.wisc.edu");

AFS is an example of a global filesystem, where any file on an AFS server can be accessed from any AFS client, assuming the user has the necessary credentials. So, if this application can tolerate higher-latency access to its AFS files, it could feasibly run on any CPU which serves as an AFS client. In this case, the application's resource request requires only that *AFSDomain* is defined, indicating that the CPU is an AFS client. The resource request indicates a preference for the cs.wisc.edu AFS domain, however, since a CPU in that domain will have the best access to the application's data files. This example uses the ClassAd =!= operator to test if an attribute is defined. It also uses the Rank expression, where a value of True is ranked higher than a value of False.

Requirements = ... && (Target.AFSDomain =!= Undefined);
Rank = (Target.AFSDomain == "cs.wisc.edu");

To indicate that the application should run on a CPU in the execution domain of a dataset, the domain manager modifies the *Requirements* attribute of the resource request as follows:

Requirements = ... && Target.HasDataSetXYZ97S3;

When the application begins its run, it uses the location attribute in the resource offer to find the dataset on the workstation. Applications which require local disk access speeds to this dataset will require a CPU in the more restrictive execution domain:

Requirements = ... && Target.HasDataSetXYZ97S3Locally;

Other applications may not strictly require local disk access speeds, but will perform better at higher speeds. In this case, the domain manager specifies the application's preferences in the *Rank* expression of its resource request (where True is ranked higher than False):

Requirements = ... && (Target.HasDataSetXYZ97S3 ||
Target.HasDataSetXYZ97S3Locally);
Rank = Target.HasDataSetXYZ97S3Locally;

4 Managing Checkpoints

We have also applied the execution domain mechanism to the management of checkpoints in Condor. In our experience, checkpoint transfers are often the main cause of network overhead for Condor applications. For example, in one Condor pool of approximately 700 CPUs, we often see more than 100 GB of daily checkpoint traffic. The checkpoint of an application's state includes its entire memory state, so memory-intensive applications can generate large checkpoints. When long-running applications obtain short CPU allocations, they must store a checkpoint at the end of each allocation to save the work they have accomplished. Dedicated checkpoint servers, deployed throughout the grid, provide storage space for these large application checkpoints.

To localize the transfer of checkpoints in the network, we define execution domains according to proximity to checkpoint servers. These *checkpoint domains* are defined by inserting a *CkptDomain* attribute into each CPU's resource offer. Applications write their checkpoints to a checkpoint server in the current checkpoint domain. Applications are restricted to migrate only to CPUs in the current checkpoint domain, to avoid transferring the checkpoint to a CPU beyond the domain. To implement this policy, the *Requirements* of the application's resource request must specify the checkpoint domain once the application has written its first checkpoint. For example:

CkptDomain = "ckpt.cs.wisc.edu";
Requirements = ... && (My.CkptDomain == Target.CkptDomain);

A task may begin execution in any checkpoint domain, but once it has completed its first checkpoint, it executes only on CPUs in the chosen checkpoint domain.

Since a task may wait a long time for an available workstation in its checkpoint domain, we support migration between checkpoint domains. As with other execution domains, a domain migration agent could transfer the checkpoint to a new checkpoint server and modify the *CkptDomain* attribute in the job's resource request. However, it is also possible for the domain migration agent to migrate the job without transferring the checkpoint between checkpoint servers by simply modifying the job's resource request. The job will transfer its checkpoint from the old checkpoint server directly to the CPU in the new checkpoint domain when it begins execution. For example, the domain manager can modify the resource request as follows so the job will run in either the ckpt.cs.wisc.edu domain or the ckpt.bo.infn.it domain:

CkptDomain = "ckpt.cs.wisc.edu";
Requirements = ... && ((My.CkptDomain == Target.CkptDomain) ||
(Target.CkptDomain == "ckpt.bo.infn.it"));

Rank = My.CkptDomain == Target.CkptDomain;

The *Rank* expression specifies that the application should remain in the current checkpoint domain if a CPU is available there. Otherwise, if a CPU in available in the ckpt.bo.infn.it checkpoint domain, the application will transfer the checkpoint from the ckpt.cs.wisc.edu checkpoint server directly to that CPU to resume its execution. If the application is preempted again, it will send its checkpoint to the local checkpoint server in the ckpt.bo.infn.it domain and update its resource request to look for a new CPU in the new checkpoint domain:

CkptDomain = "ckpt.bo.infn.it";
Requirements = ... && (My.CkptDomain == Target.CkptDomain);

It is possible for the migration agent to transfer a checkpoint only to find that CPUs are no longer available in the new checkpoint domain. By delaying the migration until the CPU is allocated to the job, the domain migration agent avoids performing potentially unnecessary checkpoint migrations. This savings represents a trade-off, because the job will need to transfer the checkpoint over a longer network distance at the start of the CPU allocation, increasing network wait time.

It is also possible to implement migration between checkpoint domains automatically (without intervention of a domain migration agent) by specifying more complex *Requirements* expressions. In the following example, the application is allowed to migrate to a new checkpoint domain if it has been waiting for an available CPU for over 24 hours.

LastCkptDomain = "ckpt.bo.infn.it";
Requirements = ... && ((My.LastCkptDomain == Target.CkptDomain) ||
((CurrentTime - My.StartIdleTime) > 24*60*60));

Or, the application may be allowed to migrate between checkpoint domains only at night, when demand for capacity on the wide-area network is lower, as in the example below:

LastCkptDomain = "ckpt.bo.infn.it";
Requirements = ... && ((My.LastCkptDomain == Target.CkptDomain) ||
(ClockHour < 7) || (ClockHour > 18));

We can also implement an even more permissive policy, which allows the checkpoint to migrate to a new domain at any time if there is insufficient CPU capacity in the current domain. We use the *Rank* expression to specify that we would prefer that the job remain in the current checkpoint domain, but it may migrate to any of the other domains specified in the *Requirements* expression when necessary.

```
Requirements = (Target.CkptDomain == "ckpt.bo.infn.it") ||
               (Target.CkptDomain == "ckpt.cs.wisc.edu") ||
               (Target.CkptDomain == "ckpt.ncsa.uiuc.edu");
Rank = (My.CkptDomain == Target.CkptDomain);
```

Checkpoint domains differ from other execution domains due to the flexibility provided by checkpoint servers. Unlike file servers, which are often not under the administrative control of grid administrators, checkpoint servers may be installed on any machines with available disk space in the grid. Since all grid nodes have access to all checkpoint servers through Condor APIs, checkpoint servers can be used as more general-purpose data staging areas to improve accessibility to application data. This improved flexibility provides greater scheduling opportunities to the execution domain managers.

5 Conclusion

We have presented execution domains, a mechanism for ensuring that grid applications have local access to needed network services, and we have shown how execution domains are realized in the Condor environment. Computational grids have the potential to deliver large amounts of computing capacity to the scientific community, but supporting data intensive applications on wide-area computational grids requires intelligent data management. Work for deploying execution domains in the INFN Condor Pool is ongoing. Additional information about the Condor research project and Condor software is available at `http://www.cs.wisc.edu/condor/`.

References

[1] I. Foster and C. Kesselman, Editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, Inc., July 1998.

[2] W. Johnston, D. Gannon, and B. Nitzberg. NASA's Information Power Grid: Distributed High-Performance Computing and Large-Scale Data Management for Science and Engineering. *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, Padova, Italy, February 2000.

[3] J. Basney, R. Raman, and M. Livny. High Throughput Monte Carlo. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, March 1999.

[4] A. Chervenak, I. Foster, C. Kesselman, C. Salisbury, and S. 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*.

[5] J. Basney and M. Livny. Improving Goodput by Co-scheduling CPU and Network Capacity. *International Journal of High Performance Computing Applications*, 13(3), Fall 1999.

[6] D. Bortolotti, T. Ferrari, A. Ghiselli, P. Mazzanti, F. Prelz, F. Semeria, M. Sgaravatto, and C. Vistoli. Condor on WAN. *Proceedings of the International Conference on Computing in High Energy and Nuclear Physics*, Padova, Italy, February 2000.

[7] R. Raman, M. Livny, and M. Solomon. Matchmaking: An extensible framework for distributed resource management. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 2:129–138, 1999.

[8] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A Data Movement and Access Service for Wide Area Computing Systems. *Sixth Workshop on I/O in Parallel and Distributed Systems*, May 5, 1999.