Harnessing the Capacity of Computational Grids for High Energy Physics

J. Basney[1], *M. Livny*[1], *P. Mazzanti*[2]

[1] University of Wisconsin-Madison, USA
[2] INFN and University of Bologna, Italy

Abstract

By harnessing available computing resources on the network, computational grids can deliver large amounts of computing capacity to the high energy physics (HEP) community. Supporting HEP applications, which typically make heavy memory and I/O demands, requires careful co-allocation of network, storage, and computing resources. The grid manager must ensure that applications have the necessary resources to run efficiently while respecting the usage policies imposed by the owners of those resources. The Condor research group is engaged in an effort to develop effective co-allocation mechanisms for computational grids. We present two of these mechanisms, checkpoint domains and file system domains, which attack the wide-area network bottleneck by providing local access to checkpoint and data storage.

Keywords: grid computing, network scheduling, high throughput computing

1 Introduction

Computational grids [1] have the potential to deliver large amounts of computing capacity to the high energy physics community by federating the computing resources on the network for large experiments. 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.

Careful management of network resources is central to the success of the grid. In this paper, we present two network management mechanisms we have developed to support the communication needs of grid applications characterized by the following attributes:

- **The application's work can be divided into many independent tasks.** This attribute signifies that the application can potentially harness many distributed CPUs to accomplish its goal. Master-worker or data parallel applications (for example, Monte Carlo studies) are particularly good candidates for high throughput grid computing [2, 3].
- **Each task is long-running with large intermediate state.** The fact that the task is long-running indicates that a checkpointing mechanism is required for successful execution on non-dedicated resources. The task must be checkpointed when it is preempted so the work it has accomplished on the borrowed workstation is not lost. Additionally, a long-running task must be checkpointed periodically to limit the amount of work lost due to hardware or software failures. The fact that the task has large intermediate state indicates that the checkpoint will be large, requiring significant network capacity to transfer.

- **Each task is I/O intensive.** The task processes a large dataset and/or produces a large amount of output. This indicates that the task must have fast access to the data storage medium to execute efficiently.

These applications can spend a significant proportion of their run time waiting on the network, transferring checkpoints and accessing remote file servers. To attack the network bottleneck, we group the computational resources in the grid into logical *execution domains* according to network connectivity. Applications store checkpoints and access data within the boundaries of the execution domains, utilizing high-speed local area networks and avoiding slower wide-area links. In the following sections, we describe our approach for defining execution domains in the Condor High Throughput Computing environment. We begin with an overview of ClassAd Matchmaking, the resource management framework we use to implement execution domains.

2 ClassAd Matchmaking

The ClassAd Matchmaking framework [4] enables flexible, distributed resource management in a grid environment. The most important feature of the framework for the purpose of defining execution domains is the ability to dynamically inject information into the scheduling system to achieve custom scheduling goals. The framework defines a resource description language which 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 requested or offered resource, a `Requirements` expression which specifies the compatibility between requests and offers, and a `Rank` expression which indicates preferences. New information is injected into the scheduling system by simply adding additional attributes or by modifying the `Requirements` or `Rank` expressions.

The following example shows a resource offer on the left describing a Sparc Solaris workstation with 256 MB of memory which is available to run grid applications when the Unix load average on the machine is less than 0.3. Applications from the AI research group are preferred, according to the `Rank` expression. The example also shows a compatible resource request on the right for a Sparc Solaris workstation with more than 80 MB of memory, where workstations in the UWCS cluster are preferred.

```
[
  OpSys = "Solaris2.6";
  Arch = "Sun4u";
  Memory = 256;
  LoadAvg = 0.25;
  Cluster = "UWCS";
  Requirements = My.LoadAvg < 0.3;
  Rank = Target.Group == "AI";
]
```

```
[
  Group = "AI";
  Requirements = Target.Memory > 80 &&
    Target.OpSys == "Solaris2.6" &&
    Target.Arch == "Sun4u";
  Rank = Target.Cluster == "UWCS";
]
```

`Cluster` and `Group` are examples of attributes added to implement custom scheduling policies. Since the application owners in this example prefer to run their applications on workstations in their department, each resource offer includes a `Cluster` attribute which specifies each workstation's location. Similarly, since the workstation owners in this example prefer to run the applications of their research group, each resource request includes a `Group` attribute which specifies each application's research group. In the next section, we show how we inject attributes into resource requests and offers to implement execution domains for checkpoint transfers.

3 Checkpoint Domains

To provide facilities on the local-area network for storing checkpoints, we organize the workstations in the grid into *checkpoint domains*. The workstations in a checkpoint domain share a checkpoint server. Checkpoint servers provide dedicated disk space for storing large checkpoint files, since disk space may not be available for storing checkpoints directly on the borrowed workstation. Checkpoint servers also allow tasks to access their checkpoints at any time without participation from previously borrowed workstations. However, when the intermediate state of a task is large, transferring a checkpoint over the network can be an expensive operation. Checkpoint domains provide a checkpoint server on each local area network and direct tasks to write their checkpoints to the local checkpoint server. When the workstation is allocated to the task, the task reads the `CkptServer` attribute from the resource offer to know which server to use. For example, the resource offer may contain:

```
CkptServer = "ckpt.cs.wisc.edu";
```

The `CkptServer` attribute serves both as a pointer to the local checkpoint server and as an indication of the checkpoint domain of the workstation.

Once a task has performed a checkpoint, the task should continue executing on workstations in the same checkpoint domain to avoid transferring the checkpoint over the wide-area network. The task must read its checkpoint to continue its execution on a new workstation. The faster the task can transfer the checkpoint to the new workstation, the sooner the task's state can be fully restored and the task can continue execution. We establish a scheduling affinity between a task and its checkpoint domain by modifying the task's request for a new workstation. We augment the resource request to include the location of the task's checkpoint (`LastCkptServer`) and a requirement that the task continue executing in that checkpoint domain. For example:

```
LastCkptServer = "ckpt.cs.wisc.edu";
Requirements = My.LastCkptServer == Target.CkptServer;
```

Under this policy, a task may begin execution in any checkpoint domain, but once it has completed its first checkpoint, it executes only on workstations 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. We can manually migrate the task by modifying its `Requirements` to include additional checkpoint domains. For example:

```
Requirements = Target.CkptServer == "ckpt.cs.wisc.edu" ||
               Target.CkptServer == "ckpt.bo.infn.it";
```

If a workstation in available in the ckpt.bo.infn.it checkpoint domain, the task will transfer the checkpoint from ckpt.cs.wisc.edu directly to that workstation to resume the task. If the task is preempted again, it will send its checkpoint to ckpt.bo.infn.it and update its resource request to look for a new workstation in the new checkpoint domain.

It is also possible to implement migration between checkpoint domains automatically by specifying more complex `Requirements` and `Rank` expressions. For example, the task may be allowed to migrate to a new checkpoint domain only if it has been waiting for an available workstation for over 24 hours. Or, the task may be allowed to migrate between checkpoint domains only at night, when demand for capacity on the wide-area network is lower. We may also envision a scheduler which performs automatic migrations by transferring checkpoints directly between checkpoint servers and updating the `LastCkptServer` attribute of the task's resource request to migrate the task to the new checkpoint domain.

4 File System Domains

In the previous section, we described how checkpoint domains improve the performance of checkpoint transfers. In this section, we introduce a similar mechanism for improving I/O performance. To provide fast access to each task's data, we organize the workstations in the grid into *file system domains*. Workstations in a file system domain share one or more file servers on the local-area network. We inject an attribute into each resource offer which lists the file system domain for the workstation. For example:

```
FileSystemDomain = "cs.wisc.edu";
```

To run I/O intensive tasks in the grid, a user first distributes the input data for each task to the available file servers. Then, by modifying the task's `Requirements` expression, the user specifies that each task should only run on workstations in the file system domain which contains the task's input data. For example:

```
Requirements = Target.FileSystemDomain == "cs.wisc.edu";
```

The task writes its output to the local file servers, and the user transfers the output data from remote domains when the task completes.

For higher performance file access, data may also be staged directly on the remote workstations. For example, each task may analyze a section of a very large dataset, where each section of the dataset is itself large and the analysis requires significant CPU time. To provide fast access to the dataset, a user may distribute the sections of the dataset to the local disks of workstations in the grid and augment each workstation's resource offer to include an attribute for each available local dataset. For example:

```
HasDataSet172 = True; HasDataSet192 = True;
```

Each task requests a machine with local access to the datasets it needs. For example:

```
Requirements = Target.HasDataSet172 && Target.HasDataSet192;
```

Each dataset may be staged on multiple workstations to provide greater availability.

5 Conclusion

Computational grids can deliver large amounts of computing capacity to the high energy physics community. Supporting data intensive applications on wide-area computational grids requires careful management of network resources. We have presented two network management mechanisms developed for the Condor High Throughput Computing environment which attack the wide-area network bottleneck by providing local access to checkpoint and data storage. Additional information about the Condor research project and Condor software is available at `http://www.cs.wisc.edu/condor/`.

References

1 M. Livny and R. Raman. High-Throughput Resource Management. In I. Foster and C. Kesselman, Editors, *The Grid: Blueprint for a New Computing Infrastructure*, Chapter 13. Morgan Kaufmann Publishers, Inc., 1998.

2 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.

3 J. Goux, J. Linderoth, and M. Yoder. Metacomputing and the Master-Worker Paradigm, September 1999. `http://www.cs.wisc.edu/condor/mw/`.

4 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.