And Away We Go: Understanding The Complexity of Launching Complex HPC Applications*

Il-Chul Yoon
Dept. of Computer Science
University of Maryland
College Park, MD 20742
iyoon@cs.umd.edu

Alan Sussman
Dept. of Computer Science
University of Maryland
College Park, MD 20742
als@cs.umd.edu

Adam Porter
Dept. of Computer Science
University of Maryland
College Park, MD 20742
aporter@cs.umd.edu

1. INTRODUCTION

To meet the computational and economic demands of high-performance computing (HPC) applications, developers often structure their applications as coupled software components running in parallel on a single computational resource (e.g., a workstation cluster) or across multiple distributed computing resources (i.e. the *Grid*).

Our experience suggests that launching these applications is more difficult than many people realize. Although it hasn't been deeply studied, we believe this situation greatly hinders productivity and that solutions to this problem could save considerable time and effort for HPC developers and end users.

To investigate this problem we conducted a small case study of a two-program coupled simulation of Sun to Earth *space weather*. During the study, we enacted a single researcher workflow operating in two modes: exploring the simulation code to better understand and improve it, and using the production code to study a particular scientific phenomenon.

During the study, we repeatedly executed the following steps for every launch. First, we discovered the computing resources on which to run the programs. Then we allocated the resources and transfered the input data for each program to their specified locations. Next, we launched the programs, producing simulation output data. Finally, we gathered the data and analyzed them.

This high-level process was the same for both workflow usage modes, although the difficulty of the individual steps changed. For example, in development mode, the study subject launched the application to plan modifications, to run tests and to assess performance under multiple use scenarios. In use mode, he launched the application with many different parameter settings and with different launching commands tailored for different hardware, OS, and middleware combinations.

Based in part on this study, we concluded that the productivity in HPC application development and utilization can be significantly improved by providing better support for launching. In the next sections we describe the case study that motivated our research, explain the issues that arise when launching complex HPC applications, and discuss how existing systems handle these issues. After that we present our prototype launch support system.

2. CASE STUDY AND MOTIVATION

Space weather simulations are used to help understand how the Sun influences the Earth's environment. Such simulations typically require coupling together software components that simulate different regions of space, including the solar corona, the Earth's magnetosphere, and the region between the Earth and the Sun.

In this case study, we investigated launching a 2-component space weather simulation, with one component employing a magnetohydrodynamics model of the region between the Sun and the Earth and the other component modeling the Earth's ionosphere [13]. These two components periodically exchange data at the shared boundary of their domains, to maintain a consistent set of physical values across that boundary. We believe that this application is typical of many (but certainly not all) HPC simulation applications.

To implement and execute this application a developer repeatedly follows the development cycle shown in Figure 1. In contrast to many other kinds of applications, we quickly observed that launching HPC applications is quite complex (i.e., not just clicking on an icon or making a command line invocation). In fact, the process is time-consuming, error-prone and cumbersome since multiple resources must be discovered, allocated and managed. In the next section we describe several issues arising from HPC application launching.

3. LAUNCHING HPC APPLICATIONS

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #CCF-0205265, by NASA under Grant #NAG5-12652, and by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. NBCH3039002.

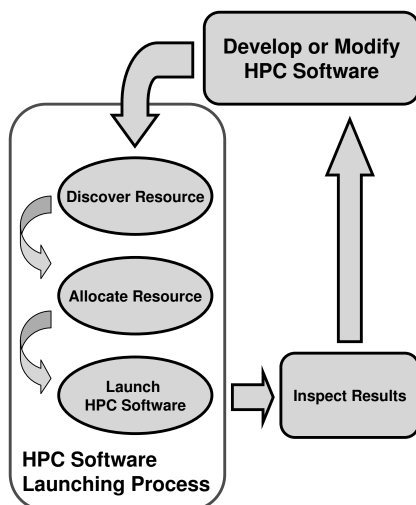

Figure 1: HPC Software Development Cycle

In this section, we address specific issues arising from each core activity in the launching process shown in Figure 1, as well as some issues that cut across multiple activities. We also describe how each issue is handled by two widely-used toolkits: the Globus Toolkit (GTK) [4] and Condor [12].

3.1 Resource Discovery Issues

In this section, we describe issues related to job specification, resource information management and resource discovery.

To do resource discovery software developers currently provide job specifications that contain low level information about the resources and software needed to launch the HPC application. This might include information such as the name of the executable and the file system location for a component, and the number of processes desired for a parallel component.

In practice, developers often are only concerned with having certain quantities or types of resources, not with having specific named resources. In these cases it would be desirable to provide an abstract job specification, specifying only information such as the program(s) to be run and the number of parallel tasks (if a program is to be run in parallel), along with the input/output arguments for each program. Of course, if the developer knows and wants to specify detailed information, such as a particular resource to run on, he should be able to write the details directly into the job specification.

A *discovery service* refines the job specification by searching an information repository for the software and resources needed to complete the job, if the specification is not already fully bound to a set of resources. To enable this functionality, services for managing information about the software and available resources must be provided to the developers and resource owners before resource discovery, and an optimized discovery mechanism should be provided to find the desired resources. Such a mechanism is crucial for efficient HPC application execution since the application components may communicate with each other and potentially exchange large amounts of data.

Condor users provide the job specification and resource specification separately using a *ClassAd* - essentially an extensible property list. The *ClassAds* are advertised to a *matchmaker* process and the matchmaking is performed based on the properties. Both **Requirements** and **Rank** are specified in the *ClassAds*. **Requirements** are the minimum constraints that must be satisfied by other *ClassAds* for matchmaking, and **Rank** is a metric to evaluate the fitness of the satisfying *ClassAds*. The matchmaker assigns a job to the resource with highest rank. GTK users describe each job with the necessary software and resource information, such as program path/name, cluster name and the number of hosts. This information is specified in the Resource Specification Language (RSL), and the RSL is refined with the help of an information service before launching the program(s) that make up the job. In Condor and GTK, the relationships between the jobs in a specification are not considered in the resource discovery process, even though such relationships are important for executing complex HPC applications consisting of multiple programs, as described in Section 2.

3.2 Resource Allocation Issues

After resources are discovered, jobs will be allocated to them. In this section, we describe several issues related to resource co-allocation and advance reservation.

Resource co-allocation is required when coupled components must run in parallel on multiple resources. For example, during our case study we ran the space weather simulation components on two different computing clusters. This required co-allocation as the two programs must frequently exchange run-time data.

Resource co-allocation is very cumbersome for software developers, and makes it more difficult for them to focus on the software development process. In addition, co-allocation can be time-consuming in complex computational environments, and also error-prone since developers must learn the properties of the diverse resources for successful co-allocation.

A co-allocation manager must be aware of the diverse conditions under which the resources are managed. For example, resources might be actively managed by independent single-resource schedulers such as PBS or LSF for workstation clusters, or passively managed as a shared set of machines. Each structure requires different allocation mechanisms. In addition, the co-allocation manager must release pre-allocated resources in the case that allocation failure occurs. The Version 2 release of GTK supported this feature through its Dynamically-Updated Request On-line Co-allocator service (DUROC) [3], which treats resource co-allocation like an atomic transaction. Condor also adopted DUROC for executing parallel software modules in a Grid environment. However, Version 3 of GTK does not provide resource co-allocation.

Advance reservation is another important issue in resource allocation. Advance reservation provides quality-of-service (QoS) guarantees for running the application. If programs must be launched on multiple coordinated resources, ad-

vance reservation is crucial for resource allocation. For example, if a program needs to transfer data to another program periodically with a minimum data transfer rate, the network bandwidth required for transfers between the resources should be reserved before launching the programs. CPUs, disk and network bandwidth can also be reserved. By providing advance reservations, multiple components can employ all the reserved resources simultaneously and reliably. Advance reservation also allows high-level resource management environments to best use resources flexibly and effectively. GTK Version 2 supported advance reservation for nodes and a single-domain network using General-purpose Architecture for Reservation and Allocation (GARA), which is an extension of the Globus Resource Allocation and Management (GRAM) package. However, GTK Version 3 and Condor do not support advance reservations.

3.3 Issues After Launching HPC Applications

Failure and output handling issues also affect launch-time behavior. Failures can happen at any point, from resource discovery to application launching. Those occurring prior to launch, such as *insufficient available resources*, can be caught by the launching environment and the user notified. However, it is very hard to detect program execution failures because the launching environment has little control over the programs running on remote resources. GTK and Condor handle such errors by redirecting the *stderr* stream from the program. Execution failures can also happen when allocated resources fail or launched programs crash accidentally. To detect diverse runtime faults, GTK Version 2 used Heartbeat Monitors (HBMs) running on each resource [11]. Each HBM monitors the status of a resource and the launched programs running on that resource using the standard Unix *ps* command, and reports the status to a data collector process that takes appropriate actions on failures. The collector can restart the program or reallocate the resource if necessary. Condor takes a similar approach to GTK Version 2.

In addition, program output(s) must be made accessible to the user. The outputs such as *stdout* or other application-specific files should be redirected to the user or stored persistently at a reliable location. GTK Version 2 realized this strategy with the Global Access to Secondary Storage service (GASS) [2] and GridFTP [1], whereby the files explicitly described in the RSL can be stored at the specified locations during or after program execution. In GTK Versions 3 and 4, the Reliable File Transfer service (RFT) utilizes GridFTP for large data file transfers, and two instances of the File Stream Grid service (FSS) are dynamically created to redirect *stdout* and *stderr*, respectively. The streams are transferred via standard SOAP messages from the remote resources to the locations specified in the RSL. In Condor, all the generated files are moved by default from the remote resources to the user's originating machine.

3.4 Cross-Cutting Issues

There are also several issues that cross-cut launching activities. For example, security must be addressed since all Grid resources and services should be protected from unauthorized access. Secure resource access can be achieved by applying strong user authentication and authorization. The types of access allowed can be controlled through the use of user credentials local to each resources. GTK and Condor provide authentication mechanisms based on X.509 certificates or Kerberos. For authorization, they adopt user-based authorization based on files that describe access controls on each resource.

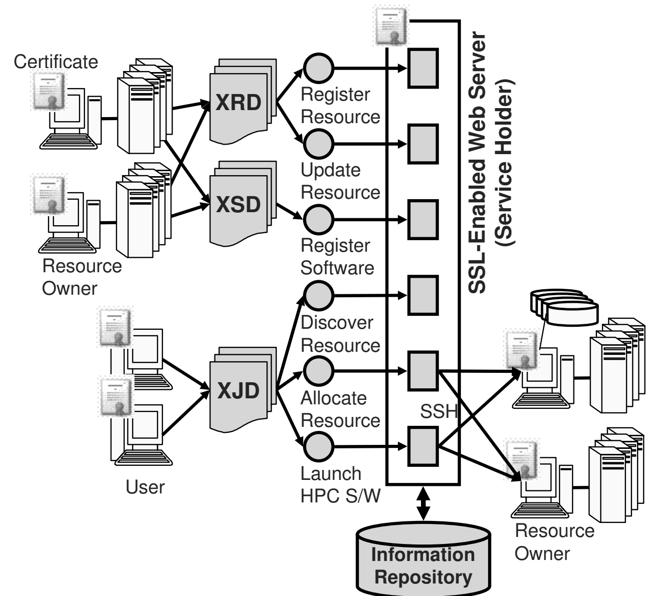

Figure 2: The Overall Architecture

Another issue is how to design the launching environment. The emergence of service-oriented architectures [5] holds great promise, since they make Grid services flexible, modular, reusable and easy-to-integrate, by separating the interface from the implementation of a Grid service. Version 3 of GTK supports a service-oriented architecture, by being compliant with the Open Grid Service Architecture (OGSA) [6]. However, Condor is not service-oriented, although efforts are underway to offer Condor services via standard Grid services interfaces.

4. OUR PROTOTYPE ENVIRONMENT

To address the above issues we are designing and implementing a prototype HPC application launching environment. Figure 2 depicts the environment's design.

To provide secure Grid services to consumers we use a strong authentication mechanism based on *two-way SSL handshaking*, like GTK and Condor. We also assume that the authenticated service consumers are authorized to invoke the desired services. For SSL handshaking, each service provider and consumer must have a certificate issued by a certificate authority, and the certificates are exchanged to mutually authenticate. After successful authentication, a service can be securely invoked over the secure session using SOAP, and commands, such as a resource allocation request, can be passed to the resource owner using SSL. This has the advantage that a service can be invoked even if it is behind a firewall, and enables a reasonable level of security. GTK Version 3 goes further and provides services compatible with recently available Web Service security standards [7].

Our environment is service-oriented. Unlike GTK Version 3, our implementation is solely based on Web Services. In the future, other features such as service lifetime management specified in the OGSA can be plugged into our design and implementation.

4.1 Resource Discovery

As discussed Section 3.1, we must manage resources and refine a given job specification with a resource discovery service. Before resource discovery begins, resources must be registered with an information repository and updated periodically. Independent resource managers deployed by resource owners register and monitor their resources. For example, if the resource type is a workstation cluster managed by a local scheduler (e.g., LoadLeveler or PBS), the resource manager deployed at the cluster head node would register information about the cluster to the repository. A resource manager provides an XML Resource Description (XRD) that describes the resource, such as a resource identifier, number of processors, type of processors (e.g., 2.6 GHz Pentium4), operating system (e.g., Linux) and resource owner. This data is registered using the *RegisterResource* service and resource availability is periodically updated using the *UpdateResource* service. Our current implementation supports two distinct resource types, **cluster**, managed by a local (PBS or LoadLeveler) scheduler, and a time-shared **node**, because HPC applications are usually launched on such resources. Unlike GTK and Condor, in our approach, we regard the software deployed by the resource owners as another type of resource, which is also used for XML Job Description (XJD) refinement. Thus, that information must be registered by invoking the *RegisterSoftware* service with an XML Software Description (XSD) that contains information such as the software owner, directory path for the software and the name of the executable program.

In addition to the registered resources, a job specification is required for the discovery process. A user provides an XJD consisting of two sections for describing the required components and how those components are connected. A software identifier and the number of tasks requested to execute the (parallel) software component are among the attributes described in the software section of the XJD. If the user wants to execute a software component on a specific resource, that can also be specified in the XJD. In this case, the discovery service only checks for the availability of the resource. The connection section describes the information between the jobs, which consists of an identifier and the job-to-job properties such as data transmission rate between two components. Appropriate resources satisfying the XJD are discovered by the *DiscoverResource* service, and then a message object is returned to the client with either an error message or the discovered resource information. For example, a discovery failure message will be sent if there is no set of resources fully satisfying the XJD.

4.2 Resource Allocation

To allocate resources manually, a developer must understand the details of the underlying resources, and requires direct access to the resource for allocation. This is made even more difficult because different resources have different allocation mechanisms that depend on the type of the resource. Since resource allocation is handled transparently by our allocation service, a developer can focus on software development.

For each job description (XJD), resource allocation starts by obtaining a unique job identifier from the allocation service. The identifier is stored in a repository and managed until all jobs in the description finish. The identifier is also used to create temporary work directories at the server running the allocation service and at the nodes managing the remote resources, because both the allocation service and the HPC software components may generate files during job execution. For example, machine files and lock files are often created by a local resource scheduler in a user-owned directory. The allocation service then uses the resource requirements in the refined XJD to make allocation requests based on the type of each resource. If a resource is directly SSH accessible, the service just checks its availability. However, if the resource is managed by a local scheduler (i.e. it is managed as a space-shared cluster), a script is queued to the scheduler that starts the job using the resources assigned by the local scheduler. The main purpose of the script is to prevent the scheduler from releasing the resources too early. Since the script waits until the lock file created by the allocation service is removed by the launching service after the software running on the resources terminates, the job can use the resources exclusively until the entire job (potentially at multiple resources) completes execution.

The sequence described above works for allocating resources for a single program execution. However, it is not adequate to execute complex HPC applications that dynamically exchange large amounts of data, because such applications often use specific runtime libraries and other services for efficient data exchange. For example, the space weather application introduced in Section 2 uses InterComm [9] for exchanging and redistributing data between two software components running on different number of processes on different resources. In this case, the resources must be joined into a single communication group (in InterComm, using PVM [8]) after allocation on each resource, due to the constraints enforced by InterComm. To enable this functionality, the script queued to the local schedulers at each resource must be modified to join the allocated resources into a communication group after allocation.

After sending allocation requests to the resource owners, the allocation service periodically monitors the allocated resources until all resources are acquired or a failure message arrives. If all the allocations succeed, the launching service can then take over. However, resource allocation can fail occasionally because the resources are inherently shared among multiple users and simultaneous accesses to a resource can happen. For example, a resource can be allocated by other users bypassing our allocation service. Therefore, the allocation service stops after a given number of retries and releases the allocated resources. Finally, the user is notified of the failure by sending a message with an error code.

4.3 Launching HPC Applications

Complex HPC applications often consists of multiple components, each of which may be launched in different ways. For example, one component may run as a message passing program with MPI [10] while another uses PVM. We have

implemented different launching modules to support diverse launching methods for HPC software components.

For each type of HPC software component, we require a machine file defining the resources used by that component. Our launching service generates that file by partitioning the machine file allocated by the allocation service. For example, if 12 nodes (processors) are allocated by the allocation service, a machine file with 12 nodes is created in the (temporary) work directory of the server holding the allocation service, and that set of nodes will be partitioned for the components to be launched on each resource. However, generating the machine file is not all that is needed to launch the components. For example, if a component transfers data to another component dynamically using InterComm, information such as the name and the numbers of tasks (processes) for the component on the other end of the transfer must be known by each component for successful execution.

The HPC components are executed at the resources specified in the machine files and may generate output data. In our implementation, *stderr* and *stdout* are redirected to user-specified files and forwarded to the user in a message object containing the execution result. Other output files specified in the XJD are also delivered to the client after a component terminates. To do this, we must determine when a component terminates. Currently this is done by inspecting the processes of the launched component running at the resources specified in the machine file, using the standard Unix **ps** command. If no processes related to the component are found, the service assumes that the component has terminated. Such periodic checking uses cycles at each resource, and is not a perfectly reliable method for checking component termination. Future research will investigate more reliable termination detection methods.

After all the components terminate, the launching service removes the lock files created at the resources. If the components utilize InterComm for data exchange, the launching service must also terminate PVM by sending a **kill** command to each node before removing the lock files, since removing lock files allows the local schedulers to release the allocated resources. Finally, the launching service clears the temporary directories created when the resource was allocated.

5. CONCLUSIONS

Launching HPC applications on multiple resources occurs frequently throughout the HPC software development cycle, and includes many repetitive, time-consuming and possibly error-prone tasks. Since this additional work can seriously decrease the productivity in development and utilization, we argue that the use of an effective and automated HPC software launching environment can improve productivity.

We have explored several issues that should be handled by a launching environment, and have described a prototype implementation of such an environment that uses a secure service-oriented design. The services provided by our environment encapsulate the low-level details needed for resource management and effective HPC application launching on multiple computational resources. In future work, we plan to improve this environment by devising diverse resource discovery algorithms and by employing additional HPC software runtime services developed both within our project and in the wider HPC community.

6. REFERENCES

[1] W. Allcock. GridFTP: Protocol Extensions to FTP for the Grid. Technical report, Global Grid Forum, Apr. 2003.

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

[3] K. Czajkowski, I. Foster, and C. Kesselman. Resource Co-Allocation in Computational Grids. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing (HPDC-8)*, 1999.

[4] I. Foster and C. Kesselman. Globus: A Metacomputing Infrastructure Toolkit. *Intl J. Supercomputer Applications*, 1997.

[5] I. Foster and C. Kesselman. *The GRID2, Blueprint for a New Computing Infrastructure*. Elsevier, 2004.

[6] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid services for distributed system integration. *IEEE Computer*, 36(6):37–46, June 2002.

[7] J. Gawor, S. Meder, F. Siebenlist, and V. Welch. GT3 Grid Security Infrastructure Overview. Technical report, The Globus Alliance, June 2003.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiand, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine: A User's Guide and Tutorial for Networked Parallel Computing*. MIT Press, 1994.

[9] J.-Y. Lee and A. Sussman. High Performance Communication Between Parallel Programs. In *Proceedings of Joint Workshop on High-Performance Grid Computing and High-Level Parallel Programming Models (HIPS-HPGC)*, 2005.

[10] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI–The Complete Reference, Second Edition*. Scientific and Engineering Computation Series. MIT Press, 1998.

[11] P. Stelling, I. Foster, C. Kesselman, C.Lee, and G. von Laszewski. A Fault Detection Service for Wide Area Distributed Computations. In *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, pages 268–278, Chicago, IL, 28-31 July 1998.

[12] D. Thain, T. Tannenbaum, and M. Livny. Distributed Computing in Practice: The Condor Experience. *Concurrency and Computation: Practice and Experience*, 17(2–4):323–356, 2005.

[13] W. Wang, M. Wiltberger, A. Burns, S. Solomon, T. Killeen, N. Maruyama, and J. Lyon. Initial results from the coupled magnetosphere-ionosphere-thermosphere model: thermosphere-ionosphere responses. *Journal of Atmospheric and Terrestrial Physics*, 2004.