A Loosely Coupled Metacomputer: Cooperating Job Submissions across Multiple Supercomputing Sites

GREGOR VON LASZEWSKI

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439. U.S.A.

Contents

A Loosely Coupled Metacomputer: Cooperating Job Submissions across Multiple Supercomputing Sites

GREGOR VON LASZEWSKI

Mathematics and Computer Science Division, Argonne National Laboratory, 9700 S. Cass Ave., Argonne, IL 60439. U.S.A.

SUMMARY

This paper introduces a general metacomputing framework for submitting jobs across a variety of distributed computational resources. A first-come, first-served scheduling algorithm distributes the jobs across the computer resources. Because dependencies between jobs are expressed via a *dataflow graph,* the framework is more than just a uniform interface to the independently running queuing systems and interactive shells on each computer system. Using the dataflow approach extends the concept of sequential batch and interactive processing to running programs across multiple computers and computing sites in cooperation. We present results from a Grand Challenge case study showing that the turnaround time was dramatically reduced by having access to several supercomputers at runtime. The framework is applicable to other complex scientific problems that are coarse grained.

1. INTRODUCTION

Many supercomputing applications can be classified in one of two categories, loosely coupled applications and tightly coupled applications. Loosely coupled applications are characterized by cooperating processes that do not require frequent message exchange between each other. In contrast, tightly coupled applications are characterized by the need for fast message exchange between the cooperating processes. Generally, a metacomputing environment must be designed to support the execution of both application classes. Analogous to the simple characterization for the applications, one can classify metacomputing environments. Thus, a *tightly coupled metacomputing environment* utilizes hardware and software technologies that enable rapid communication between processing elements on different computational resources. In a *loosely coupled metacomputing environment* several supercomputers or workstation clusters are utilized while running jobs in a distributed fashion. In this context a *job* is defined by a set of processes that are executed on one of the machines without communication to another machine during the execution. Although the metacomputing framework introduced in this paper can be used for both application classes, as demonstrated in [16, 15], we concentrate here on issues related to a loosely coupled metacomputing environment. An important requirement for a loosely coupled metacomputing environment is the *efficient* use of the diverse and distributed computing resources.

In this paper, we introduce the different parts necessary to enable such a metacomputing framework. We motivate our work based on analysis of various Grand Challenge applications. We outline the general system architecture and introduce the different components, including a uniform job submission component, a task graph editor, and a monitoring interface. We then present a scripting language for supporting the dataflow computations. The paper ends with a brief discussion of current and future research.

2. MOTIVATION AND PROBLEM ANALYSIS

This work was motivated by an analysis of the development and execution of parallel programs for various Grand Challenge problems and other computationally intense problems, including atmospheric modeling[13], x-ray diffraction[17], linear programming [20], and genome analysis [7]. Several observations from this analysis led to requirements that influence the design of the loosely coupled metacomputing environment introduced in this paper:

- **Reduction of development cost through software reuse.** Programs to solve Grand Challenge problems are highly specialized. Their development takes many person-years. Because of the cost involved in designing and maintaining the parallel code, it is desirable to reuse as many software components as possible that have been proven to be working correctly.
- **Specification via a task graph.** Many large, complex programs are designed and specified in such a way that they can be represented by a task graph. Interfaces between the separate tasks are usually well defined so that independent development teams can implement the assigned task. Often the only communications of these tasks are performed during initialization and termination in order to pass parameters from one task to another.
- **Specialized infrastructure**. These programs are designed with a particular infrastructure in mind. The infrastructure includes specialized hardware, ranging from the processor on which the program is executed, to storage media and networks for secondary data storage and data transfers. Such specialized hardware generally is located at supercomputer centers, because of the cost for providing and maintaining a reliable infrastructure. Job submission to supercomputers in such centers is usually performed in batch mode.
- **High demand of runtime resource in space and time.** The execution of jobs at such centers consumes a lot of memory and time. Because of the limited resources at the centers, applications utilizing fewer resources are frequently assigned a higher priority than applications requesting a large amount of resources. Grand Challenge problems make maximum use of existent hardware infrastructure in order to achieve usable results in the quickest time possible.
- **Parameter studies.** Some applications require an instantiation of the program on a variety of input parameters. Although this requirement can be viewed as additional motivation for a high demand in space and time, equally important are the repetitive characteristics and the potential of generating tens of thousands of independent tasks (as is the case for the molecular x-ray structure analysis [9]).
- **Usability by application users through simple interfaces.** It is necessary to provide a simple interface to the metacomputing environment and the parallel programs in order to hide the often complicated infrastructure that makes the submission of programs a complex task. Most users of the analyzed application preferred to view the computing environment as a "black box."

Before we introduce the detailed architecture of this framework, we emphasize that it is possible to reduce the overall turn-around time while performing one or more of the following strategies:

- **Space reduction by using fewer processors for the same application.** The average wait time of a job in a batch queue is determined not only by the jobs already lined up in the queue, but also by administrative decisions. The number of processors used and the reservation time at queue submission have a big impact on the average waiting time. In many centers, the waiting time in queues for long-running jobs is usually longer. Thus, the turn-around time can be reduced if the parallel program uses fewer processors during the instantiation. Formally, we define the turn-around time for an application A running on p processors as

$$T^A(p) = t_q^A(p) + t_{io}^A(p) + t_c^A(p) \ ,$$

  where t_q^A specifies the average waiting time in the batch queue, t_{io}^A the time to read and write the input and the output data, and t_c^A the time for the actual calculation. If the number of processors is reduced, the job can be reclassified for execution in a shorter queue. As indicated above, many supercomputing centers schedule such jobs with higher priority. This strategy does not require modifying the original application.

- **Time reduction by dividing the application in multiple smaller jobs**. Although the above strategy often leads to the reduction of the time spend for queuing, it does not guarantee that the overall time will be reduced. In many cases the application can be divided into several independent running jobs. Although the number of processors used is not reduced, the execution time of the computations of their parts is smaller. Formally, the minimal turnaround time of an application A that is split into two sequential tasks A_1 and A_2 is defined as

$$\min_{\substack{A_1;\ A_2 = A \\ p_1 \leq p \\ p_2 \leq q}} \quad T^{A_1}(p_1) + T^{A_2}(p_2) \ ,$$

  where p_1 and p_2 are the number of processors used for the different parts, and p is the total number of available nodes. Often it is sufficient to find a combination that is smaller than the original turn-around time.

- **Space-time reduction by using multiple supercomputers.** Using multiple supercomputers at different sites to execute the parallel tasks is likely to reduce the time used to perform the overall calculation. In many cases this is also the only way that the application can be run: namely, by using the combined ability of the distributed resources.

The metacomputing framework introduced in this article supports all of the strategies mentioned above in order to reduce the wall-clock time of a program instantiation. A central feature of this framework is the specification of the task graph as part of a dataflow graph. This specification allows the user to:

- specify the jobs easily in a maximal parallel fashion;
- specify dependencies between the jobs as links in the task graph and as temporal condition in the dataflow graph;
- use the simple representation in a dataflow graph to spawn jobs across multiple supercomputers at potential different sites; and
- utilize a scheduling algorithm that makes it possible to submit jobs uniformly across the sites.

Additionally, it is important that the computational resources in the metacomputing environment be easy to use. The goal of simplicity results in the following requirements for the metacomputing framework:

- enable remote file staging across the different computing sites;
- provide a uniform interface to the queuing systems controlling the submission of jobs to the supercomputers, including cancellation of jobs on remote sites; and
- provide the ability to monitor job status.

Users would also like to be able to develop their programs on their local computers, while the compilation and execution are performed remotely on the appropriate target machine. This feature should be available as a shell command as well as a Graphical User Interface.

3. A MOTIVATING APPLICATION AND THE COMPUTATIONAL TESTBED

Before discussing the architecture design in detail, we briefly examine the requirements for a particular application. We have chosen the Grand Challenge application known as Four-Dimensional Data Assimilation (FDDA) [13] because it takes advantage of all the design features presented in this paper. FDDA is a technique used in climate and weather forecasting to improve the quality of the forecast. The goal is to produce a *best estimate* of the state of the system based on a physical model of the earth system and actual observations collected over time.

An essential problem in FDDA is to interpolate from a set of irregular distributed data values (the climate or weather observations) to a set of data values spawned over the globe at predefined locations. This set of data values (often a grid) is used as an initial condition to a climate model or weather forecast. This calculation is repeated for every six-hour period.

A high level flowchart of the FDDA cycle is shown in Figure 1. During the initialization, the data is prepared and erroneous data points are corrected or rejected. This corrected list of observations is input of an objective analysis algorithm that derives the initial condition for the climate model.

The data preparation, the quality control and the objective analysis can be decomposed based on the geographical location of the input data (the observations and the variables describing the earth system). In Figure 3 we show one possible domain decomposition which divides the globe into four regions. Including the data from the overlap regions allows to formulate the execution graph as shown in Figure 2. In this example four processors

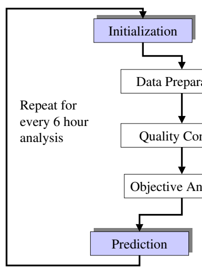

Figure 1. *The Four Dimensional Data Assimilation Cycle.*

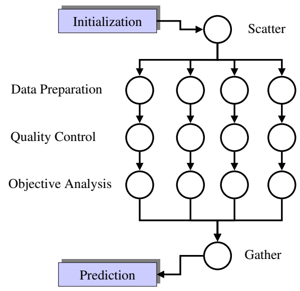

Figure 2. *Decomposition of the Four Dimensional Data Assimilation Cycle.*

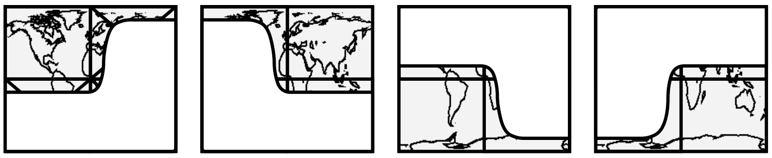

Figure 3. *A possible domain decomposition for the data analysis.*

can be utilized in parallel. More sophisticated decomposition strategies including dynamic decompositions, can be found in [16]. For the mathematical formulation[10, 11], the sequential and parallel algorithm[18, 15], and further information[4] about this problem, we refer the reader to the literature.
During the observation of the scientists conducting their repetitive experiments with different parameters or source modifications, we noticed that the turnaround time is slowed down by the large queuing times for submitting the program to the local scheduling system. At the same time, we observed that the queues at other centers were not as busy. Thus, we included selected cites into a testbed.
For the testbed environment we used computers located at five different geographical locations, as shown in Figure 4. For most of our experiments we utilized a subset: the SP2 at the Northeast Parallel Architectures Center, Cornell Theory Center and Maui High Performance Computing Center. Since the scientists were not familiar with these systems, the requirements for uniform job submission, file stageing, and uniform program execution becomes apparent.

4. ARCHITECTURE

Many possibilities exist to provide a framework that fulfills the design goals and requirements introduced earlier. Because we wish to keep the design as simple as possible, the general architecture is based on a client-server model. The basic components of the architecture are depicted in Figure 5 and are explained in more detail in later sections.

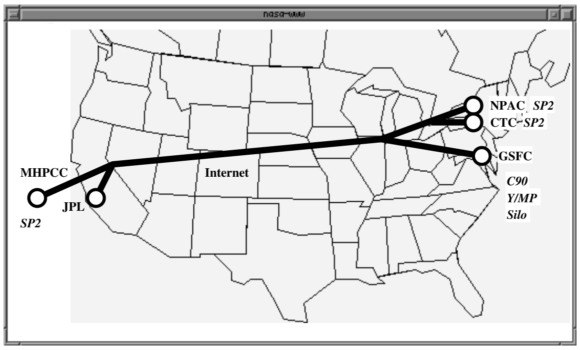

Figure 4. *Geographical distribution of the metacomputing components as used in the NASA Four-Dimensional Data Assimilation Grand Challenge project.*

Four interface components enable users to communicate with the metacomputing environment in the following ways:

- *monitoring* the activities on a remote compute resource (e.g., the batch queues),
- *submitting* jobs uniformly to the different batch queues,
- *composing* a parallel application with the help of a task graph editor, and
- *scripting* a parallel application with a simple text based editor.

For the monitoring, submission and composition requirements graphical user interface component are available. The desire of some users not to use a Graphical User Interface is implicitly addressed in this design, since the components interact with the backend system via a universally used scripting language. Each of the components will be described in more detail later in this article.

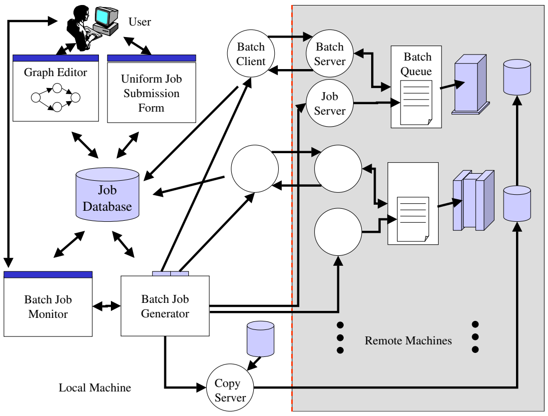

Figure 5. *General architecture of the metacomputing framework.*

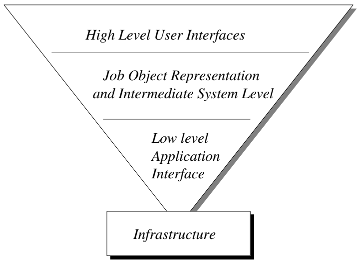

Figure 6. *The layers of the metacomputing framework.*

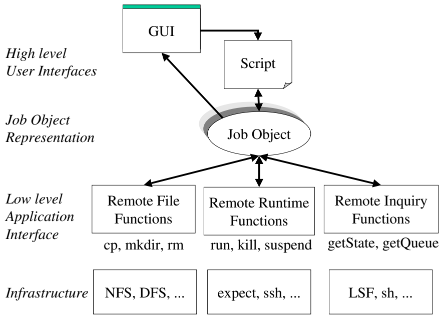

Figure 7. *Components and application interfaces of the metacomputing framework.*

The general metacomputing framework is based on a layered architecture (Figure 6), which contains a small set of application interfaces that can be ported to different infrastructures. These interfaces are distinguished by their functionality (Figure 7):

- **Remote File Functions** allow users to access files on a remote machine. The functions include *cp*, for copying a file from one machine to the other; *mkdir*, for creating a directory on a remote machine; and , for deleting a file on the other machine. The function *exists* is used to test if a file on a remote machine is available. To obtain more information about the size and the access rights, the *property* function is used.
- **Remote Runtime Functions** enable a program to execute on a remote machine. Authentication is performed in clear text, though it is straightforward to provide a key-based infrastructure. Besides starting a program with the *run* command, it is important to be able to *kill* and *suspend* the command on the remote machine.
- **Remote Inquiry Functions** allow users to obtain information about jobs running on remote machines. This includes information not only about the current job, but also about batch queues that might run on the remote machine.

These functions are invoked through methods that are part of an object specification of the jobs submitted through scripts or the graph editor. The job objects are stored in the job database, which provides a persistent storage for the state of the system.
A typical interaction is invoked on the user's home machine by using either a specialized scripting language or the task graph editor. The jobs specified in such a way are recorded in a job database, and the batch job generator is notified. The batch job generator controls the job submission and execution from the client to the appropriate remote computational resource. The status of the batch queuing system, as well as the state recorded in the job database, can be observed with the job monitor. On each of the remote sites a *batch server* and a *job server* supervise the program execution. Specifically, the batch server monitors the batch system at the remote site and reports changes to all jobs running on this machine. The only function of the job server is to submit a job to the appropriate batch

queue. After this step is completed, the job server is terminated at the remote machine in order to conserve valuable compute resources.
The design allows for computational resources that operate in batch and in time-sharing mode. Hence, one can extend the framework to large distributed computing networks. The architecture is implemented by using Java, Perl[2], and tcl/expect[12, 8] using rsh, telnet, ftp, and TCP/IP[14].
Since the metacomputing framework is able to execute a variety of tasks, we first demonstrate the capability to perform uniform job submissions. Later, we will demonstrate the extended batch control mechanism based on the dataflow concept.

5. JOB SUBMISSION

In this section we discuss the mechanisms implemented in our metacomputing system to assist users of different supercomputing sites who are faced with the problem of using different interfaces to the batch queues running on different supercomputers.

5.1. Uniform Job Submission

A uniform job submission form allows users to submit jobs via remote shell execution of the appropriate commands accessing the queuing system or via specialized scripts based on *expect*. Figure 8 shows a screenshot of the general submission form appliaed to the FDDA application. Only a few essential characteristics of the underlying job submission systems are used in order to keep the submission process as simple as possible.
The form requests that the user specify the name of the file in which the job is described, the directory from which the program is executed, the files for standard output and error, the users e-mail for notification upon completion of the job, as well as, parameters determining how long the program should run and how many processors should be used. The status of a queuing system on a remote site can be visualized as displayed in Figure 9.

5.2. Cooperating Job Submission

To demonstrate the way in which a loosely coupled metacomputer program functions, we use the example of the FDDA application. The parallel program to solve the mathematical tasks associated with the FDDA can be divided into several stages [18], each of which is executed in parallel separately (a very common approach for large codes).
Figure 10 shows an example of such a program. The task of the program is to use some input data, process it with a program, forward the output to another program, and show the result of the overall calculation on a terminal. In the execution specification, program A is mapped onto a DEC Alpha workstation farm, while program B can be mapped onto the SP2's from NPAC, Cornell, or Maui.
The uniform job description form, as displayed in Figure 8, is used to specify differences for each machine due to the file system and user accounts, as well as the differences in the batch program operating at the different sites. After the jobs have been properly defined, the program can be executed. A job database is used to supervise the running jobs on the different machines. The contents of the database can be visualized with a monitoring user interface (Figure 11). Each job is assigned a unique identification and can have input and output data dependencies with other jobs (e.g., before it can be started, a job has to wait for

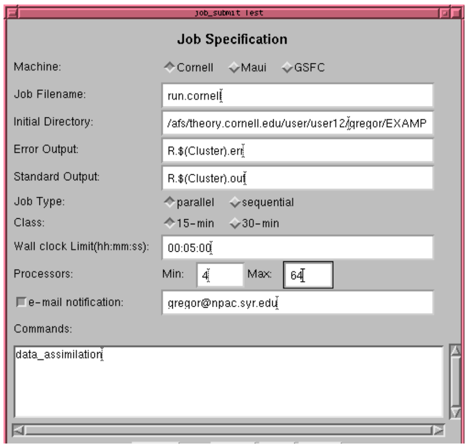

Figure 8. *The uniform job submission form allows the user to submit a job to the batch queues of remote supercomputers as well as interactive shells.*

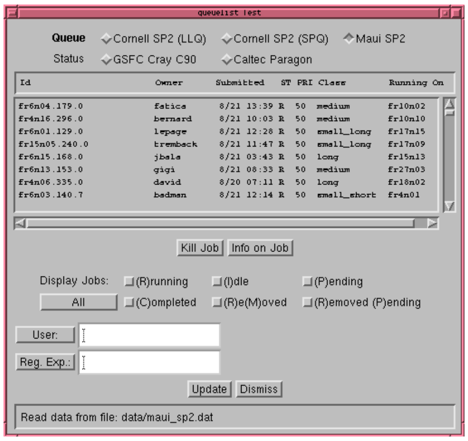

Figure 9. *The job monitor allows the user to observe the status of jobs submitted to remote supercomputers, including the state of the batch queues.*

the completion of the jobs given in the input dependency list). As a job is completed, the state is updated in the job database, and dependencies with other jobs are resolved. Jobs that do not depend on any other jobs (their dependencies are resolved) are then submitted for execution. Once the job is running on the particular machine, the database and the monitoring interface are updated again. The selection of jobs and their execution are repeated until all dependencies are resolved. It is a straightforward implementation of a classical dynamic dataflow concept using global control.

Internally, the dataflow graph is converted to a sequence of shell commands that are executed in a special Java thread. The thread is responsible for scheduling and supervising the parallel jobs that are executed asynchronously. For each job, a separate thread is created. A job can be in one of the following states:

- **Submitted** – the job is submitted and prepared for running on the target machine.
- **Running**– the job is running on the target machine.
- **Completed** – the job is completed, and the result is available for other jobs.
- **Failed** – the execution of the job has failed.
- **Waiting** – the job is waiting for the completion of one of its dependencies (input resolution).
- **Halted** – the job has been halted by the operator.

A separate list is used to collect failed jobs. They are either resubmitted or, after a particular timeout, forwarded to another component of the metacomputing environment. This approach provides a minimal fault-tolerant behavior of the environment. Global control on the user's home machine is used to supervise the jobs in the batch queue.

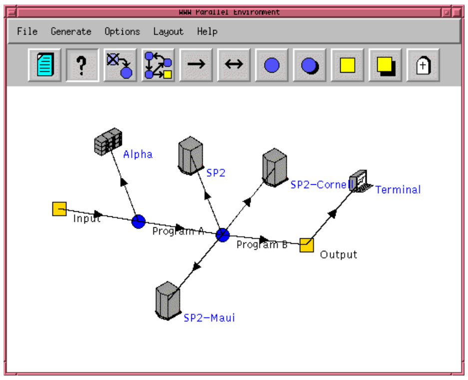

Figure 10. *A loosely coupled metacomputer program.*

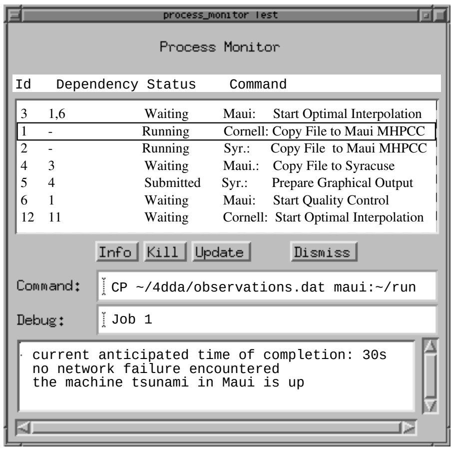

Figure 11. *List of jobs submitted to the metacomputing environment.*

Once a loosely coupled job is generated with the help of the graphical editor, it is transferred to an intermediate language. This language is similar to a shell script. Such similarity is essential to enable programming a tightly coupled metacomputer in textual form as well as in graphical form. Again, simplicity is important because such an environment will be used by scientists who do not necessarily know the newest technology in computer science.

The language is built around a few simple commands allowing remote file access and remote compilation. Each command can be attached with an *id*, which allows the generation of dependencies between jobs. The *SUBMIT* command submits a job to the specified remote computer and returns a unique *id*. Jobs are all submitted asynchronously. The job id can be used to *CANCEL* the job or perform other actions on it. The *WAIT* command waits on the execution of the specified job or job set. It is possible to wait for the completion of all jobs (*WAIT(ALL)*) in ajobset or to wait till one job (*WAIT(ONE)*) of the set is completed. The immediately returning *PROBE* command returns the state of a job. Thus, it is possible to check for the states introduced earlier (SUBMITTED, RUNNING, COMPLETED, FAILED, WAITING, HALTED). In combination with the WAIT command, the PROBE can be applied to sets and returns the first job id for which the probing command returns a value.

Figure 12 depicts a simple script for compiling and running the FDDA program on two remote machines. The script uses a first-come, first-served scheduling policy. One of the strengths of this approach is the simplicity of formulating a program running on multiple platforms through different batch operating systems. Tasks that would have to be executed normally by hand, while inspecting batch queues, can now be executed automatically. Additionally, the method to specify such jobs is for many users intuitive since it is based on the extension of a shell/Perl like language. Thus, the learning curve for utilizing this approach is low.

```
# Create the directory
MKDIR cornell : PROJ
# Copy the sourcecode from Maui
id$1 := CP home : PROJ/Makefile cornell : PROJ/.
id$2 := CP home : PROJ/run cornell : PROJ/.
id$3 := CP home : PROJ/fdda.f90 cornell : PROJ/.
id$4 := CP home : PROJ/fdda.h cornell : PROJ/.
id$5 := CP home : PROJ/Makefile maui : PROJ/.
id$6 := CP home : PROJ/run maui : PROJ/.
id$7 := CP home : PROJ/fdda.f90 maui : PROJ/.
id$8 := CP home : PROJ/fdda.h maui : PROJ/.
WAIT(ALL) id$1 - 8
# Compile the program
id$cornell := APPLY cornell : PROJ/make all
id$maui := APPLY maui : PROJ/make all
WAIT(ALL) id$maui, id$cornell
# Submit the job
id$runcornell := SUBMIT cornell : run
id$runmaui := SUBMIT maui : run
id$executing := WAIT(PROBE(SUBMITTED)) id$runcornell id$runmaui
# Get the result from the machine starting the job first
if (id$executing == id$runmaui) {
    CANCEL id$runcornell
    id$wait := WAIT(ALL) id$runmaui
    $machine = "maui"
} else {
    CANCEL id$runmaui
    id$wait := WAIT(ALL) id$runcornell
    $machine = "cornell"
}
WAIT(ALL) id$wait
CP $machine : PROJ/final.dat .
```

Figure 12. *Example of the scripting language used to define a loosely coupled metacomputing application.*

6. JOB SCHEDULING

The goal of the job-scheduling algorithm is to minimize the overall wall-clock time for executing the program. For efficient job scheduling, a runtime prediction algorithm can be included in the framework.

The mapping of jobs to machines operating in time-sharing and batch mode forces one to use different strategies for predicting the runtime of a job. In time-sharing mode, system values such as current load and load averages are used to predict the completion time of the job. This information is especially useful when a performance prediction function is available for the job.

If performance prediction is not possible or the job is executed on a supercomputer operating in batch mode, the execution time is not as easy to predict. One way to solve this problem is to submit all jobs to a global queue. This global queue decides which machine the job will be executed on. For example, the batch operating software CODINE is able to submit jobs to different machines [3, 1].

In most cases, however, the metacomputing center maintains its own local queues and

policies. In such a case, the jobs can be simply replicated on each one of the machines. The first job completed causes the other jobs to be terminated, which are then removed from the queue. The disadvantage of this strategy is that if jobs are started at the same time, valuable resources in CPU cycles might be wasted. More sophisticated algorithms are possible in case advanced reservation policies are enabled on the batch queues.
Once more performance data of the average wait times in the batch queues and the execution time of the applications on the compute server are available, the machine on which the job is executed can be selected based on its performance characteristics. Specifically, a more sophisticated algorithm can be used that first predicts the expected turn-around time on a variety of machines and then selects only those machines where the time for execution is minimal. For parameter studies on deterministic algorithms of known complexity, this strategy results in a very effective scheduling since the data of previous runs is recorded in the job database and the performance characteristics can be reused in the next job to be scheduled.
The status of the machines can be monitored in multiple ways. It is possible to run a Java server on the supercomputer side and send information about the status upon request. The current setup allows the user to start a remote procedure call and consult the job queue for information about the status of the running jobs.
In the next section we show the design of a more sophisticated dynamic resource selection instead of the rather simple first-come first-served scheduling resource scheduling.

7. DYNAMIC RESOURCE SELECTION

The dynamic execution of a program is driven by three factors: (1) the software modules available for the different hardware platforms, (2) the availability of a computational resource, and (3) the use of the computational resources. Figure 13 illustrates the process responsible for selecting the hardware and software used to execute the program on existing hardware platforms.

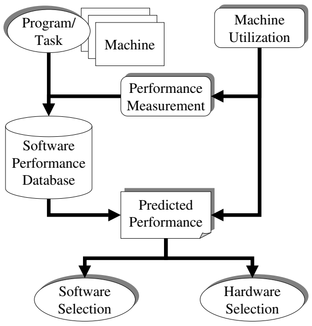

Figure 13. *Dynamic selection process during program execution*

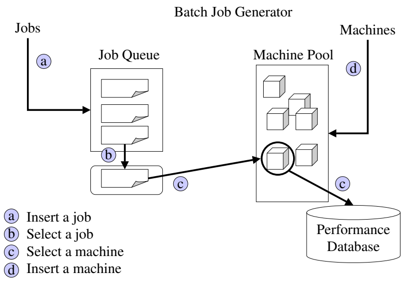

Figure 14. *Steps performed in the dynamic selection process.*

As pointed out previously, in some scientific programs (so-called parameter studies), a problem is solved many times for similar instances of data, with little difference in execu-

tion time. The information about the execution time on different machines is stored in a database, either via direct measurement or via a performance prediction analysis algorithm. This information can then be used to predict the real-time performance of the program. If several choices of software and hardware mappings are available, the one with the shortest execution time is chosen. Hence, the selection not only includes a hardware mapping, but also can include the use of completely different algorithms to solve the (meta)problem.
Figure 14 depicts the steps that are performed to update the information in the performance database. Several important asynchronously running tasks are distinguished with the letters a, b, c, and d.

(a) Jobs are continuously inserted in the job queue.

(b) A job is selected from the queue for execution. Different selection algorithms can be integrated into the environment. The simplest strategy is to select the jobs in a first-in-first-out fashion. More sophisticated algorithms make use of the time and memory used to complete the jobs while sorting them appropriately. The strategy for selecting the jobs can be configured by the user.

(c) Once a job is selected, an appropriate machine has to be found. If no machine seems to be appropriate to run the current job, the job is inserted back into the end of the job queue, and step b will be performed. If a machine for execution was determined, the job is submitted. After completion, the runtime information is stored into the performance database for improving future schedules.

(d) Machines that are exceeding a particular number of currently scheduled jobs are removed temporarily from the machine pool in order to capture the fact that many computing centers have a policy of "maximum number of jobs per user" in place. This number can be configured individually for each machine.

The maintenance of a machine pool in addition to the job queue, from which resources are selected, allows the dynamic resource selection.

8. RESULTS

To test the metacomputing environment, we ran experiments on the SP2 in Cornell, Syracuse, and in Maui. During these experiments, the jobs were submitted asynchronously to the machines. We used the scheduling algorithm, as well as, the predicted waiting times in the queues for both machines. Based on this prediction, the jobs were started on the machine with the least amount of waiting time. Whenever a job was accepted in a machine, the corresponding job on the other machine was deleted. This simple strategy reduced the wall-clock time for the completion of the job by 3–5 hours, and thus enabled us to obtain dramatically reduced turnaround during peak hours.
In addition, it is important to point out that the metacomputing framework, introduced in this article, is easy to use. It allows for easy specification of programs as well as their monitoring in a distributed environment.
Another important result is the ability to port the framework to other metacomputing infrastructures. Recently, a port based on the Globus Metacomputing toolkit was developed and is currently under improvement.

9. CONCLUSION AND FUTURE RESEARCH

The metacomputing system introduced in the preceding sections is very general. It is useful for providing a uniform interface to the job submission systems at different supercomputing sites, as well as integrating multiple jobs in an overall program. Dynamic behavior of the infrastructure can be monitored and used for steering the computation. The system can be ported to different sites, since only a small set of routines interfacing with the environment need to be adapted. A port that replaces the job start-up mechanism via the Globus metacomputing toolkit is under development [5].
The integration of the framework introduced in this paper into a larger framework such as the Globus metacomputing toolkit allows expansion of its usability. We plan additional applications of this framework in other Grand Challenge projects; initial results were demonstrated at Supercomputing '98 [19]. The next step will involve developing more sophisticated scheduling algorithms that make better use of the dynamical changing infrastructure.

ACKNOWLEDGMENTS

This research would not have been possible without the cooperation of several researchers at Northeast Parallel Architectures Center (NPAC) and at Goddard Space Flight Center in Greenbelt, MD. In particular, I thank Geoffrey C. Fox[6] and Miloje Makivic at NPAC for their valuable discussions related to this research. In the Data Assimilation Office, I am grateful to Peter Lyster for his guidance, his many helpful comments, and fruitful discussions. I thank David Lamich, James Stobie, Mike Seabloom, Arlindo DaSilva, and Steve Cohen. I am especially grateful to Richard B. Rood for his support and hospitality during several visits at NASA Goddard Space Flight Center as part of the Universities Space Research Association. Facilities from the Northeast Parallel Architectures Center, Cornell Theory Center, Maui High Performance Supercomputer Center, Goddard Space Flight Center, and the Jet Proportion Laboratory were used to conduct the experiments. This project has been funded by NASA High Performance Computing and Communications Earth Science Project. I thank Gail Pieper for her assistance in preparing the final manuscript. The more recent developments were conducted in the Distributed Systems Laboratory, led by Ian Foster at the Mathematics and Computer Science Division at Argonne National Laboratory.

References

[1] Baker, M., Fox, G. C., and Yau, H. Cluster Computing Review. Tech. Rep. 1995, Center for Research on Parallel Computation, Nov. 1995.

[2] Christiansen, L. W. T., and Schwartz, R. L. *Programming Perl*, 2nd ed. O'Reiley, 1996.

[3] CODINE. http://www.genias.de/genias/english/codine/codine.html.

[4] Daley, R. *Atmospheric Data Analysis*. Cambridge Atmospheric and Space Science Series, Cambridge University Press, 1991.

[5] Foster, I., and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications 11*, 2 (1997), 115–128.

[6] Fox, G. C., Haupt, T., and Furmanski, W. Webflow. http://www.npac.syr.edu/users/haupt/WebFlow/papers, 1998.

[7] Gasterland, T., and Sensen, C. Fully Automated Genome Analysis that Reflects User Needs and Preferences — a Detailed Introduction to the MAGPIE System Architecture. *Biochemie 78*, 4 (1996).

[8] Libes, D. *Exploring Expect*. O'Reiley, 1995.

[9] Miller, R., Gallo, S., Khalak, H., and Weeks, C. SnB: Crystal structure determination via Shake-and-Bake. *Journal of Applied Crystallography 27* (1994), 613–621.

[10] NASA Data Assimilation Office at Goddard Space Flight Center. *Data Assimilation Program Version 1.2*. Greenbelt, MD, 1993.

[11] NASA Data Assimilation Office at Goddard Space Flight Center. *Data Assimilation Program Version 2.0*. Greenbelt, MD, 1993.

[12] Ousterhout, J. *Tcl and the Tk Toolkit*. Adisson Wessley, 1994.

[13] Pfaendtner, J., Bloom, S., Lamich, D., Seablom, M., Sienkiewicz, M., Stobie, J., and da Silva, A. Documentation of the Goddard Earth Observing System (GEOS) Data Assimilation System - Version 1. Tech. Rep. 4, Goddard Space Flight Center, Greenbelt, MD, 1996.

[14] Stevens, R. W. *UNIX network programming*. Prentice Hall, 1990.

[15] von Laszewski, G. An Interactive Parallel Programming Environment applied in atmospheric Science. In *Making its Mark, Proceedings of the 6th Workshop of The use of Parallel Processors in Meteorology* (Reading, UK, Dec. 1996), G.-R. Hoffman and N. Kreitz, Eds., European Centre for Medium Weather Forecast, World Scientific, pp. 311–325.

[16] von Laszewski, G. *A Parallel Data Assimilation System and its Implications on a Metacomputing Environment*. PhD thesis, Syracuse University, Dec. 1996.

[17] von Laszewski, G. The Argonne Grand Challenge X-ray WWW page. http://www.mcs.anl.gov/xray, 1999.

[18] von Laszewski, G., Seablom, M., Makivic, M., Lyster, P., and Ranka, S. Design Issues for the Parallelization of an Optimal Interpolation Algorithm. In *Coming of Age, Proceedings of the 4th Workshop on the Use of Parallel Processing in Atmospheric Science* (Reading, UK, Nov. 1994), G.-R. Hoffman and N. Kreitz, Eds., European Centre for Medium Weather Forecast, World Scientific, pp. 290–302.

[19] von Laszewski, G., Westbrook, M., Foster, I., Westbrook, E., and Barnes, C. Using Computational Grid Capabilities to Enhance the Ability of an X-Ray Source for Structural Biology. *to be submitted* (1999).

[20] Wright, S. OT: Optimization Toolkit for large scale optimization problems. http://www-unix.mcs.anl.gov/ot, 1998.