Extending Beowulf Clusters

By Daniel R. Steinwand,[1] Brian Maddox,[2] Tim Beckmann,[1] and George Hamer[3]

Open-File Report 03-208

[1] USGS, EROS Data Center, SAIC, Sioux Falls, SD 57198-0001. Work performed under U.S. Geological Survey contract 03CRCN0001.
[2] Mid-Continent Mapping Center, Rolla, MO 65401
[3] South Dakota State University, Brookings, SD 57007

U.S. Department of the Interior
U.S. Geological Survey

Contents

Illustration

Abstract

Beowulf clusters can provide a cost-effective way to compute numerical models and process large amounts of remote sensing image data. Usually a Beowulf cluster is designed to accomplish a specific set of processing goals, and processing is very efficient when the problem remains inside the constraints of the original design. There are cases, however, when one might wish to compute a problem that is beyond the capacity of the local Beowulf system. In these cases, spreading the problem to multiple clusters or to other machines on the network may provide a cost-effective solution[4].

Key Words

Parallel Processing, Beowulf Clusters, High-Performance Computing, Remote Sensing, Image Processing

[4] Any use of trade, product, or firm names is for descriptive purposes only and does not imply endorsement by the U.S. Government.

Introduction

The project described in this paper is a continuation of work that commenced in fiscal year (FY) 2000 with the identification of individuals at U.S. Geological Survey (USGS) Mapping Centers interested in building an information science research infrastructure within the National Mapping Division (NMD) (now called the Geography Discipline). At that time, employees at USGS sites (the EROS Data Center (EDC) in Sioux Falls, South Dakota, the EDC/Alaska Field Office (AFO) in Anchorage, Alaska, the Mid-Continent Mapping Center (MCMC) in Rolla, Missouri, and the Rocky Mountain Mapping Center (RMMC) in Denver, Colorado) prepared and submitted a research proposal to begin investigations into high-performance computing. Approval of follow-on proposals for continued funding in FY 2001 and again in FY 2002 has enabled the Centers to enhance performance and communication on their existing clusters and to test various applications on these systems.

This part of the project focused on looking beyond what a single Beowulf cluster in the USGS system could compute. Three specific topics were addressed, and each is described in detail in this report. First, researchers at MCMC looked at and modified the Multicomputer Operating System for UnIX (MOSIX) as a way to dynamically allocate cluster nodes. Second, at EDC, and at South Dakota State University (SDSU), researchers looked at Condor as a way to link two or more clusters, as well as individual desktop computers, on a network. Finally, researchers at EDC did an experiment with “Internet supercomputing” as an alternative to the cluster approach.

MOSIX Research

One of the biggest problems with distributed processing is that applications must be specially written to run in a distributed environment. For older software, this generally requires redesign and reimplementation, which can make it cost prohibitive to move to a distributed processing scheme. Writing software explicitly for distributed processing also reduces the portability of that software, as it is then tied to some form of specialized cluster environment.

A solution to this problem may well be the use of MOSIX. MOSIX was developed by a team led by Professor Amnon Barak of The Hebrew University in Jerusalem (Barak and others, 1999). MOSIX differs from a traditional distributed processing cluster in that it makes every machine in the cluster appear as part of one large parallel computer. It does this by migrating processes between nodes in a cluster, so that a process running on a heavily loaded node can be migrated to one with resources available. The interesting thing about this process migration is that it is done transparently to the process that is migrated. MOSIX leaves a small “stub” program on the original node that is used for communication purposes, while the process itself can be moved around the cluster. This stub process enables the main process to communicate with the originating computer as if it were still running on that computer. The advantage of this technique is that programs do not have to be explicitly written to run on a MOSIX cluster. Older software need only be compiled to run under Linux to take advantage of the process migration.

MOSIX also offers options for parallel I/O operations over a cluster. The Direct File System Access (DFSA) mechanism is “a re-routing mechanism that reduces the extra overhead of executing I/O oriented system calls of a migrated process” (Amar, 2002). It does this by redirecting requests so that they run on the node the process is currently running on and are not sent to the stub on the originating node. The MOSIX DFSA mechanism can also migrate a process to the node where most of its I/O operations take place. This can help, for example, when a process may be reading a large amount of data from a traditional NFS file-serving node. DFSA can move the process to that node so that reading takes place locally instead of over the network.

The ability to migrate processes transparently is the main characteristic of MOSIX. As previously mentioned, MOSIX leaves a small stub process on the originating node when that process is migrated. MOSIX will then redirect communications to and from the stub. The interesting aspect of this is that the process need not know it has been migrated. It is so transparent that applications that interact with a user can be migrated to another node, and the user will be unaware that the migration has happened. This stub is critical because it allows processes to run remotely even if they have not been specifically written to use MOSIX. This is beneficial since older programs can be run under MOSIX and take advantage of the dynamic load balancing over a cluster.

The process migration transparency also enables MOSIX to function alongside traditional Beowulf processing environments. For example, implementations of the Message Passing Interface (MPI) standard can be run alongside MOSIX. MOSIX can provide better load balancing than typical MPI implementations provide, and DFSA can in theory help to balance I/O requests over the cluster.

MOSIX is not without problems, however. MOSIX is implemented as a series of patches to the Linux kernel. These patches are quite extensive, and therefore make it harder to apply any other patches alongside MOSIX. The monitoring utilities that are available through a download also do not fully implement certain functionalities, such as easily determining where a specific process is currently running on the cluster. For this, a small application was developed at MCMC to check each node to see if a specific process had been migrated there. The reference utilities available through a download only provide a simplistic graph of system load per machine in a MOSIX cluster. Better management utilities may be available only in commercial versions of the software.

The OpenMOSIX (Bar, 2002) version was initially used for testing. OpenMOSIX is derived from Barak's original MOSIX version. OpenMOSIX is also fully placed under the General Public License (GPL), along with the necessary user utilities. The choice of OpenMOSIX was made owing to the personal convictions of the MCMC project lead about the use of truly open software. However, the choice of OpenMOSIX proved problematic as it presented numerous difficulties. The first problem noticed was that the user utilities provided with OpenMOSIX were more primitive than those with mainline MOSIX. The OpenMOSIX kernel itself also experienced numerous crashes that could not be explained.

The biggest difficulty with OpenMOSIX, however, was that it had serious problems when the number of processes started on a given node passed a certain threshold. For example, an image reprojection application was initially modified so that it would simply start all MPI tasks on the master node and let OpenMOSIX migrate them to the various nodes around the cluster. This worked up to a certain point, but past this point the originating node would either lock up for a period of time or crash. Numerous attempts were made to diagnose this problem. The only determination made was that there appeared to be a process threshold, but it was not fully consistent. The results of the threshold, system crash or temporary lock up, also were not consistent.

In the end, these difficulties forced researchers to abandon OpenMOSIX in favor of the mainline MOSIX version. There were several immediate benefits to this. MOSIX had an automatic installer script that took care of many operations that had to be done by hand with OpenMOSIX. MOSIX also was more sophisticated, both in stability and in user utilities.

MOSIX was also found to have the same "number of processes" problem that OpenMOSIX had. However, MOSIX never actually crashed when this threshold was broken, and that threshold was far more predictable in MOSIX. It was found that this problem in MOSIX is related to the "power" of the machine (processor speed, amount of memory, and so on). With mainline MOSIX, the application would still finish, but the head node would become unresponsive until the number of processes dropped past a certain level. To work around this problem, we modified the projection software so that it would spawn MPI tasks across several nodes and let MOSIX migrate those tasks to the rest of the cluster. This kept the number of processes started on any given node below the limit.

Another problem was observed with MPI tasks when run in MOSIX. After a processing node is finished, its piece of the processing application exits when told there is no work left. If this exit took place while the processing application was still migrated, the application would crash. The solution to this problem was to modify the projection software to directly tell MOSIX to migrate it back to the originating node before exiting.

When these changes had been made, tests were done to observe how well MOSIX could load balance a data-bound application. With the projection software modified and the initial problems with MOSIX solved, the software was set to start several MPI tasks on a small number of nodes. To see how well MOSIX would allow an MPI task to run on a non-MPI enabled machine, we only started MPI on the head node and the nodes that began the processing tasks. MOSIX was running on each machine in the cluster during these tests. The tests were set up to run in overloaded and underloaded states. In the overloaded state, there were more MPI processing tasks started than there were nodes on the cluster. The underloaded case involved starting fewer processing tasks than nodes on the cluster.

The overloaded and underloaded states were chosen to test various aspects of MOSIX for data-bound processing. In the overloaded state, some machines will run more than one processing task at a given time. This test was designed to see not only if MOSIX would intelligently select the dual processor machines to run multiple tasks but also how well overloading would work for processing. The underloaded state was chosen to see if MOSIX would keep processes on their migrated nodes or if it would periodically move processes around to various nodes. This test was to determine if MOSIX would act like a multiprocessor machine, where the operating system will sometimes move a process from processor to processor. For input, a file was stored on a single node and traditional NFS file serving was used.

Figure 1 compares processing times under MOSIX with comparable non-MOSIX runtimes. As can be seen, MOSIX does suffer a performance penalty when performing data-bound processing. The first and third columns compare MOSIX and DFSA. DFSA is actually slower than non-DFSA for overloaded processing. It was observed that MOSIX kept trying to migrate processing tasks to the file server node, which slowed down processing because task migration consumes some time. Overloaded processing with and without DFSA is slower than overloaded run without MOSIX using traditional MPI task spawning. The last two columns show that the underloaded case was slower with MOSIX than without. DFSA was also turned on in the underloaded case, causing the same problems here as with the overloaded case.

This demonstrates that although MOSIX may be able to load balance traditional distributed processing tasks, it is not well suited for data-bound processing where large amounts of data are passed through the network. In theory, DFSA would help as it moves the processing jobs to the node that stores the data. However, large numbers of processing tasks suffer bottleneck problems as MOSIX tries to move all of them to the file-serving node each time these tasks try to read data. Even without DFSA, MOSIX is slower owing to penalties incurred from the communication with the MPI system through the stub process on the originating nodes. However, DFSA may perform better when multiple file-serving nodes are used.

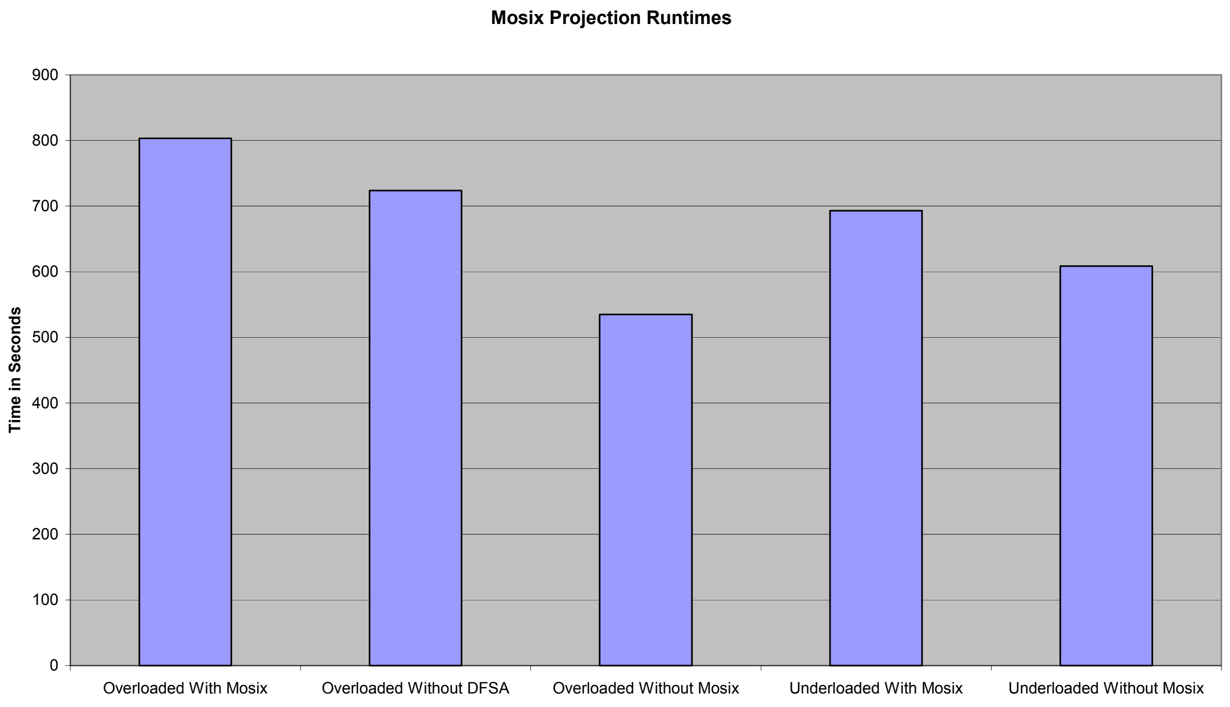

Figure 1.   MOSIX runtimes.

MOSIX also demonstrated that its load balancing algorithms attempt to move processes randomly between nodes. It was observed that MOSIX would move processes around between similarly loaded nodes instead of just leaving a single node overloaded and not move it around to other equally overloaded nodes. However, this is not necessarily a problem specific to MOSIX because distributed load balancing is an incredibly difficult computer science problem that still has not been successfully solved. This is similar to what happens in dual or multi-processor computers where an operating system may periodically switch a task between processors instead of exercising processor affinity (leaving the process running on the same node for the duration of the process).

This experimentation with MOSIX did lead to some ideas about how it could be used to implement massively parallel processing clusters within an organization. MOSIX can be easily enabled or disabled. This means that a node can enter and leave a MOSIX system at any time. MOSIX will also migrate processes off a node that is either rebooting or voluntarily leaving the MOSIX system. The point to note here is that a machine can easily enter or leave a MOSIX processing system without negatively affecting the rest of the system.

These capabilities of MOSIX can lead to a model where an organization can install MOSIX on large numbers of desktop machines and control when the machines are processing for the desktop user or when they are part of a MOSIX cluster. A machine, for example, can join a MOSIX cluster after the user has gone home and leave the cluster when the user arrives at work. For batch processing tasks, such as traditional data processing activities, this means that an organization can utilize machines when they are normally idle, especially during nonwork hours when most desktops are unused. The data-bound processing problems previously noted may not affect this type of system as much because these traditional data processing activities usually consist of a large number of tasks instead of a single distributed task. Data could also be served from of multiple file servers and DFSA turned off for this type of system.

Additionally, users of the Windows operating system would not be excluded from contributing to these types of organizational processing. Products such as VMware can allow a virtual machine to run on a host system. This emulation, for example, enables a Linux machine to run a Windows operating system virtually. It can be set to full-screen mode and shield most users from ever knowing that they are running a virtual form of the Windows operating system. In this case, MOSIX could run on the desktop and contribute to processing during idle times, or it could be set to enter and leave the processing cluster automatically.

Because MOSIX will allow things such as MPI tasks to run on non-MPI machines, processing tasks would not necessarily have to be concerned about executing on machines that contain all of the necessary software libraries and other support applications. This can help administration tasks, since machines may not necessarily be configured just for a given application. Instead, they would only need standard system libraries installed.

The organizational cluster concept could allow any group to contribute massive amounts of computer power to processing tasks. When computers do not have to be dedicated, it may be easier to take advantage of distributed processing, since the nodes that process are preexisting desktop nodes. Implications to consider would include the reinstallation of operating systems on the machines and the configuration of VMware (or something similar to it) to fulfill any Windows requirements.

Condor Research

Extending the Beowulf Cluster to the Desktop

Existing Beowulf clusters are normally constrained by the number of compute nodes physically connected to the cluster's network switch. To extend the size of the cluster requires adding new compute nodes to the switch. At some point, the capacity of the switch will become the limiting factor in cluster size. For example, a 48-port switch can house no more than 48 compute nodes. When this point is reached, an additional switch will be needed to increase the size of the cluster. This increases the cost of the cluster more than just the cost of an additional compute node.

Many work sites have computers that are underutilized a high percentage of the time. After normal working hours, this represents a tremendous computing resource that goes largely untapped. Exploiting this idle resource by making these underutilized machines part-time cluster nodes makes sense as a way to increase the computing power in a cluster. Idle computers—still in their native office environments on their office networks—can be polled by the cluster's master node and incorporated into the cluster if the candidate machine's load conditions warrant.

There are many methods that can be used to extend clusters. One method is to use Parallel Virtual Machine (PVM) to add hosts beyond the cluster. A more complex method is to use the Globus Toolkit to create a compute grid. The Condor system from the University of Wisconsin is a middle-ground solution that uses the office network and offers scheduling and authentication services. Using PVM places most of the work on the programmer to allocate and deallocate compute nodes, whereas the Globus Toolkit allows the grid designer to hide this complexity.

An investigation of these concepts is being conducted at the EDC in conjunction with SDSU. This investigation will continue into FY 2003, but preliminary results are discussed here.

The Beowulf cluster at SDSU was extended with machines from student computer labs using PVM. Normally, the cluster is composed of 18 dedicated PIII 500-MHz machines, each with 128 megabytes of memory and an 8-Gigabyte hard drive. Each of these nodes runs the Linux Mandrake 8.0 operating system running the 2.4.3-20mdk kernel. The Computer Science Department labs contain 51 Dell Optiplex GX-240 computers, each with P4 1.8-GHz processors with 256 megabytes of memory and 40-Gigabyte hard drives. The lab machines boot either Windows XP Professional or RedHat Linux version 7.3 running the 2.4.18-10 kernel.

PVM was installed and used to extend the cluster with computers from the computer lab, and a small test program was written to demonstrate this functionality. The next step was to create a parallel version of EDC's *All Possible Regressions* algorithm that exhibits O(2^n) growth characteristics. The current implementation will run for approximately 4 months with 32 variables on a single CPU. The goal is to reduce this to a few days (or hours) by spreading the job over multiple machines. At SDSU, 100 machines have been identified that could be used to test this theory. Jobs could be scheduled to run during the night or over a weekend and will be able to use all compute nodes.

After the parallel version is complete, the compute pool will be configured to use the Condor software from the University of Wisconsin. This will allow the dynamic scheduling of jobs and machines to the compute pool. The current parallel environment—with PVM alone—requires the programmer to manually schedule jobs and resources. When Condor and PVM are combined, the programmer will no longer have to embed the scheduling code in the applications software. Condor also features job rollback so that a job can be stopped in progress on one node, moved to another, and then restarted on the new node. This will allow the user to run jobs on unoccupied desktop machines during a normal working day. Condor will identify idle machines and schedule jobs to run until the owner returns. The Condor project is also using Windows NT computers in a Unix or Linux Condor configuration.

Although it is beyond the scope of the current investigation, the Globus Toolkit could be used to create a grid of computers that exceed the boundaries of an organization. This could conceivably be used to tie together Beowulf clusters in the Geography Discipline into one computing system. The Condor system has recently released a version called "Condor-G" that can tie into the Globus Toolkit.

Internet Supercomputing Research

An Investigation With Java and the Frontier API

The following discussion documents the process followed to port the Biological Resources Division's (BRD) Mid-continent Ecological Science Center (MESC) Kriging algorithm to Java and Parabon's Frontier application programmer's interface (API) for providing massive computational power and describes the results obtained. This task was undertaken in support of a USGS venture capital proposal by EDC Beowulf investigators—the same investigators who completed the MPI version of the Kriging algorithm. Parabon was a partner in that proposal.

Parabon's Frontier API provides access to large amounts of CPU power by providing access to idle CPUs on Internet connected computers. One of the first efforts to use this type of computing model was the SETI project that allows home computer users to install a screen saver application that performs calculations while the screen saver runs. Parabon has generalized this model and made it available through their Frontier API as a commercial product. For more information on Parabon and Frontier, visit Parabon's Web site at http://www.parabon.com.

Kriging Algorithm Background

Kriging is a process that can be used to estimate the values of a surface at the nodes of a regular grid from an irregularly spaced set of data points. The EDC team was asked to parallelize an implementation of the BRD/MESC Kriging algorithm to run on a Beowulf cluster in an attempt to reduce processing times. The MPI API was chosen for implementing the parallelization on the cluster. For that effort, the original higher-level code was ported from FORTRAN to C; some of the numerical subroutines were left in FORTRAN. The resulting port was successful, and the application attained a nearly linear speedup on the 12-node cluster available at EDC (Steinwand and others, 2003).

Porting to Java

When the MPI Beowulf implementation was complete, the same application was ported to the Frontier API. Frontier is implemented in Java, so the first step in this effort was to port the entire application to Java. The part of the code that had previously been converted to C was easy to port. The part that remained in FORTRAN was more difficult, because of the structure of the original implementation. After the conversion to Java, testing revealed a bug in the original FORTRAN version in which a sort routine was sometimes giving incorrect results. The two versions produced very similar results; differences that exist appear to be due to floating-point roundoff and the now fixed sorting bug.

When the Java port was completed, a small amount of performance testing was done to determine how the single-processor speed of the Java implementation compared with the single-processor speed of the C/FORTRAN compiled version. The Java version performed approximately 35 percent slower than the C/FORTRAN compiled version. (This, however, contained an unnecessary square root operation that was removed from the Java version but remained in the compiled version. Without that change, the Java version was approximately 50 percent slower.)

Porting to the Frontier API

The Frontier API differs from the MPI API. A typical MPI application has a master node that assigns work to slave nodes. The application can choose between assigning each slave node a large chunk of work at once or it can dynamically assign work on the basis of how quickly each slave node completes its work. The slave nodes can communicate with each other or the master node at any time with relatively small latencies.

Frontier limits the communication that can take place. The application must split all the work for a job into separate tasks at the start of the job, and all the individual tasks are submitted to the server. The server schedules and runs the tasks on the nodes available. The tasks are not allowed to communicate with each other during execution. They also are not allowed to communicate with the submitting application, except for returning intermediate or final results. These limits are understandable owing to the computing resources used to perform the computing tasks. Some flexibility is given up in exchange for cheap computing cycles. However, the limits do eliminate the Frontier API from being used on some classes of problems. Another limit that needs to be considered is that much of the communication to computing nodes probably takes place over relatively slow Internet connections, so the amount of data that need to be transmitted should be relatively low compared with the computing power required.

Parabon has prepared an excellent white paper on the capabilities of the Frontier API, as well as example code and tutorials in the developer section of its Web site.

The Kriging application is not a perfect fit for the Frontier API. For each location in the output grid that is calculated, a 4-byte floating-point result needs to be returned. The Java implementation running on a single 733-MHz processor is capable of producing results at a rate of nearly 10 KB per second. If compute nodes are connected with a relatively slow connection or a faster computer, the Internet connection bandwidth can easily become a limiting factor in how quickly results are received.

Frontier requires a large amount of code to be written to create jobs and tasks, submit them to the server, and receive the results. A rough estimate is that it requires twice as much code to interface to the Frontier API as it does to use MPI. If this code were to be written from scratch, it would be a daunting task. Luckily, the Frontier software development kit (SDK) includes several example applications and most of the code can be reused with minimal changes.

The “RemoteApp” demo was used from the Frontier SDK as a basis for the port. It was a relatively simple task to modify the code to support the different needs of the Kriging application. The “RemoteApp.java” file was renamed to “KrigApp.java” and modified in the following ways:

- Command line options were added to allow specifying input and output file names.
- The input data file was packaged up and sent to the server to be used by the tasks.
- The parameters passed to the defined tasks were changed to control which rows in the output grid a task is assigned.
- Code was added to assign each task a different set of rows in the output grid.
- The code that listens for results was modified to receive an array of results and write them to a binary file as they are received. Note: Because of the platform-independent way Java stores floating-point numbers, the binary file byte order may not match a binary file written from a C program on the same machine.
- Code was added to convert the binary file to ASCII after all the results have been received. Note: The binary file is needed to allow for randomly

writing results for a given row to a known location in the output file since the order in which results are returned is essentially random. The results are then converted to ASCII to match the default output format of the original code.

In addition to the “KrigApp.java” file, a new file named “KrigTask.java” was created. This code is modeled after the “RemoteTask.java” example code. An instance of the KrigTask class is created on the compute nodes and assigned a set of output grid rows to create. The task calculates those rows and returns them by posting the results. Much of this code is simply ported from the C/FORTRAN implementation. The only special items that were needed to support the Frontier API were the following:

- Supporting a stop exception so the task can be stopped easily.
- Adding methods to allow input parameters to be passed.
- Returning the output grid rows. This was complicated since Frontier does not directly support sending an array of floating point numbers. However, the development Frequently Asked Questions (FAQ) section suggested converting the array to a byte array and returning that as a “BinaryParameterValue”. This worked well.

One design decision was to avoid the use of checkpoint logging for this application. The large amount of checkpoint data required for this application would decrease throughput by a large amount. See the Frontier API documentation for a description of logging checkpoints.

Running the Application

After the code was ported, it was run locally for testing. The Frontier SDK allows jobs to be run either remotely (that is, on the Parabon system) or locally by emulating much of the Parabon system on the local machine. For initial testing, the local mode was used to work out bugs and make sure the correct results were being returned. The “remote.sh” from the RemoteApp demo was modified slightly to set up the environment and run in either local or remote mode.

To run the application locally, the command line is as follows:

```
local.sh input_data_file_name output_data_file_name
```

To run it remotely, the “local.sh” is just replaced with “remote.sh”. These scripts are essentially identical; the script name used determines whether the local or remote mode should be used. Note that the script is a Unix shell script. To run the application on the Windows operating system, one must modify the “remote.bat” file from the Frontier SDK.

Testing with the local mode allowed the developers to work out a few minor bugs and to check the results before attempting the remote method. This is an excellent method to make sure the application works as designed. After a job is submitted to run remotely, it is much more difficult to debug.

After the local application was running correctly, remote tests were conducted. The first step was to sign up for a 30-day free trial on Parabon's system. This simply involved filling in some information on a Web form and submitting it. A few hours later, an e-mail was sent confirming that the account had been set up and was ready for use. The free trial account provides access to 10 compute nodes with a low priority and is meant to allow testing applications in the remote mode.

Running the application in remote mode is very similar to the local mode. The only differences are as follows:

- The “remote.sh” script must be run twice. The first time is to submit the job to the system, and the second time is to listen for the results. The second time could be avoided, but the example code was structured this way, and it was deemed unnecessary to change it. Parabon's documentation mentions that this allows a job to be submitted and the results retrieved later.
- Each time the “remote.sh” script is run, passwords must be entered to authenticate the account information.

The first small remote job ran to completion and produced the same results as the local run.

Timing Results

The application was run twice with different-sized datasets as shown in the following table:

Rows	Tasks	Local runtime	Remote runtime
12	10	24 seconds	511 seconds
221	12	302 seconds	1,561 seconds

These results did not bode well for expecting a speedup from running a job in parallel. However, these jobs were most likely too short to give a fair assessment of the system.

Next, the full dataset was submitted. The full dataset requests 3,401 rows in the output grid. Also, for this test, Parabon allowed us to run the test on 500 nodes (instead of the original 10). The additional runs are shown in the following table:

Rows	Tasks	Local runtime	Remote runtime
3,401	171	3,744 seconds	2,346 seconds
3,401	500	(not run again)	1,724 seconds

These last two runs show that the system can produce results more quickly.

However, these tests revealed additional implications. Frontier includes a "jobcontroller" application that allows jobs to be monitored using a graphical user interface. While watching the status of the job using the jobcontroller application, the tester noticed that the jobcontroller application reported all 500 tasks in the job had been completed a full seven minutes before the results had finished downloading. It appears the initial suspicion that the Kriging application was not a perfect fit for the Internet computing model was correct, because of the large amount of data returned. There may be ways to reduce the time required to obtain results, but they were not explored. It may be possible to restructure the application to pull results down from more than one task at a time. However, it is unknown if the Frontier API can support this concept.

Also, it should be noted that the computing nodes are working on a task when the normal user is not actively using the machine. So a simple time comparison can be skewed by a task or two out of hundreds being delayed by the available idle time on the compute nodes.

Summary

The Frontier API shows promise for a select class of problems that are computationally intensive. Developers familiar with Java and distributed programming techniques will have no problems adjusting to the programmer's interface. It took less than 2 days to read the Frontier white paper and modify the Java code to make use of the Frontier interfaces, using the demo code in Parabon's software development kit as a starting point. It took significantly longer to port the original C/FORTRAN MPI implementation to Java.

The results indicate that the Kriging application does not appear to be a good fit for Frontier. It returns a relatively large amount of data as a result of the calculations, and the time to transmit those results quickly becomes a bottleneck. The phrase "does not appear to be a good fit" is used since no further investigation was done to see if the Frontier API provided a mechanism that would allow a quicker return of the results. It might be possible that simply compressing the data with the Java API for data compression might result in a significant savings since there is quite a bit of repetition in the data returned.

A decision to use Frontier and Parabon's system would need to be made on a case-by-case basis. The limitations of the system quickly rule out any application that returns a large amount of data relative to the computation time. However, if a job exists that takes weeks or months of computations, does not transmit much data, and does not need to share results part way through the calculations, Frontier is a good fit. Also, if the application code is not in Java or is not easily portable to Java, it could rule out the use of Frontier.

References

Amar, Lior, Barak, and Shiloh, 2002, The MOSIX Parallel I/O System for Scalable Performance: Institute of Computer Science, The Hebrew University of Jerusalem.

Bar, Moshe, 2002, OpenMosix Internals: presented at the Linux Kongress, Germany.

Barak A., La'adan O. and Shiloh A., 1999, Scalable Cluster Computing with MOSIX for Linux: Proc. Linux Expo '99, p. 95-100, Raleigh, N.C.

Steinwand, D.R., Maddox, B., Beckmann, T., and Schmidt, G., 2003, Processing Large Remote Sensing Data Sets on Beowulf Clusters: U.S. Geological Survey Open-File Report 03-216

Important Websites

MOSIX. Dr. Amnon Barak. Hebrew University. <http://www.MOSIX.org>.

MPI – The Message Passing Interface Standard. Argonne National Laboratory. <http://www-unix.mcs.anl.gov/mpi/>.

OpenMOSIX, an Open Source Linux Cluster Project. Moshe Bar. <http://openMOSIX.sourceforge.net>.

Licenses – GNU Project – Free Software Foundation. Free Software Foundation. <http://www.gnu.org/licenses/licenses.html#TOCGPL>.

Processor Affinity and Binding. AIX Versions 3.2 and 4 Performance Tuning Guide. <http://usgibm.nersc.gov/doc_link/en_US/a_doc_lib/aixbman/prftungd/procaffin.htm>.

VMware – Virtual Machine Software. VMware, Inc. <http://www.vmware.com>.

Condor – http://www.cs.wisc.edu/condor

GLOBUS – http://www.globus.org

Parabon – http://www.parabon.com

SETI -- http://setiathome.ssl.berkeley.edu/