Beowulf Distributed Processing and the United States Geological Survey

By Brian G. Maddox[1]

Open-File Report 02-15

[1] Mid-Continent Mapping Center, Rolla, MO 65401

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

Contents

Figures

Introduction

In recent years, the United States Geological Survey's (USGS) National Mapping Discipline (NMD) has expanded its scientific and research activities. Work is being conducted in areas such as emergency response research, scientific visualization, urban prediction, and other simulation activities. Custom-produced digital data have become essential for these types of activities. High-resolution, remotely sensed datasets are also seeing increased use.

Unfortunately, the NMD is also finding that it lacks the resources required to perform some of these activities. Many of these projects require large amounts of computer processing resources. Complex urban-prediction simulations, for example, involve large amounts of processor-intensive calculations on large amounts of input data.

This project was undertaken to learn and understand the concepts of distributed processing. Experience was needed in developing these types of applications. The idea was that this type of technology could significantly aid the needs of the NMD scientific and research programs. Porting a numerically intensive application currently being used by an NMD science program to run in a distributed fashion would demonstrate the usefulness of this technology.

There are several benefits that this type of technology can bring to the USGS's research programs. Projects can be performed that were previously impossible due to a lack of computing resources. Other projects can be performed on a larger scale than previously possible. For example, distributed processing can enable urban dynamics research to perform simulations on larger areas without making huge sacrifices in resolution. The processing can also be done in a more reasonable amount of time than with traditional single-threaded methods (a scaled version of Chester County, Pennsylvania, took about fifty days to finish its first calibration phase with a single-threaded program).

This paper has several goals regarding distributed processing technology. It will describe the benefits of the technology. Real data about a distributed application will be presented as an example of the benefits that this technology can bring to USGS scientific programs. Finally, some of the issues with distributed processing that relate to USGS work will be discussed.

Description and History of Distributed Parallel Processing

A few definitions are in order to understand what parallel processing is. The distinction must first be made between concurrent processing and parallel processing. Concurrent processing is defined as referring to "environments in which the tasks we define can occur in any order. One task can occur before or after another, and some or all tasks can be performed at the same time" (Nichols 1998, p. 16). Parallel processing is then defined to "specifically refer to the simultaneous execution of concurrent tasks on different processors" (Nichols 1998, p. 16). What this means is that in parallel processing, tasks must be able to run correctly at the same time on multiple processors.

In parallel processing, tasks are truly running on multiple processors simultaneously. This is different from a common misconception that operating systems such as Microsoft Windows or the various UNIX variants allow multiple applications to run at the same time on a single processor. What happens in such cases is that the operating system divides the processor's time up into various units. It then schedules things so that each running application gets one of these units one after the other. These units are switched so quickly that the operating system gives the illusion that the programs are running simultaneously. In parallel processing, multiple processors are being used at the same time, giving multiple tasks the ability to be truly running simultaneously on a computer.

This speeds up the "wall clock" execution time of a program, since it can run many parts of itself at the same time. It also allows a program to do things that it would not ordinarily be able to do. For example, an accounting program could divide its yearly total work by running a half-year on two processors at once instead of doing the whole year on a single processor. The task would take less total time, as it would be doing multiple calculations simultaneously.

Distributed parallel processing can then be defined as the simultaneous execution of concurrent tasks across multiple processors on multiple computers. This is typically done by a series of machines connected by means of a local area network. The main difference is that instead of running on multiple processors in the same machine, the multiple processors are in different machines. It is still a form of parallel processing, though, just on a larger level. It is also more difficult, as communication between different nodes must be synchronized and some nodes may process data at different rates than others.

The idea of parallel processing has long been studied in computer science. It can be traced back as far as 1958, when John Cocke and Dr. Daniel Slotnick discussed the idea of parallelism in numerical calculations in an IBM research memo (Wilson, 1994). Dr. Slotnick was granted a patent for parallel processing in the mid-1960s after construction of the ILLIAC IV, the world's first supercomputer (Computers in Your Future, 2000). Today, the theories of parallel processing are taught in many computer science programs throughout the country.

A Beowulf cluster is a specialized distributed processing cluster that has several distinguishing features. Thomas Sterling and Donald Becker originated the idea at the Center of Excellence in Space Data and Information Services in 1994. As defined in the *Mailing List FAQ* (Sitaker and others, 1999), it is "a kind of high-performance massively parallel computer built primarily out of commodity hardware components, running a free-software operating system like Linux or FreeBSD, interconnected by a private high-speed network. It consists of a cluster of PCs or workstations dedicated to running high-performance computing tasks. The nodes in the cluster don't sit on people's desks; they are dedicated to running cluster jobs. It is usually connected to the outside world through only a single node." Many clusters are constructed out of upgraded, surplus computers. A Beowulf cluster generally has a low construction cost because few, if any, specialized components are used.

Because of the low cost of Beowulf clusters, and because they are now recognized by the high-performance computer community, many organizations are using them to satisfy

their high-end computing requirements. As an example, the NOAA Forecast Systems Laboratory has constructed a 256-node Beowulf cluster for weather predictions. Over the next three years it will be upgraded to use over one thousand processors (NOAA, 2000). The Los Alamos National Laboratory has their *Avalon* cluster that consists of one hundred forty Alpha workstations (*Avalon, 2000*). Many other organizations are using clusters for activities that range from computing galactic-collision simulations to serving information on the World Wide Web.

Comparison With Other Techniques

To better understand distributed parallel processing, we must compare it with other techniques, such as parallel processing and SETI@home type processing. Although these techniques are similar, they do have several important differences that must be examined.

With parallel processing, multiple tasks run simultaneously on multiple processors. The processors are generally part of the same computer. All communication between tasks happens inside the same "box." While tasks may run at the same time on multiple processors, they are limited by having to use common systems, such as memory and disk storage. The problems begin here because these systems generally can only be used for one task at a time due to their connection to the processor through a system bus. Hard drives and memory may appear to be doing multiple tasks at once, but in reality they can do only one thing at a time. These devices are limited mainly by physical limitations, such as a hard drive only being able to write to one location on the disk at a time. Access to these resources is scheduled and given out by the operating system. While one task is using a resource, such as a hard drive, the other tasks must wait in line until it becomes available. Fortunately, this happens quickly enough that most users do not notice the wait.

Contrast this with distributed processing, where the processors are housed in different computers. Each task running on a node has its own memory, hard drives, and other hardware. The processing task is only competing with the operating system for processor time, not with the operating system and other processing tasks as in a parallel environment. The drawback to this setup is that communication must happen on a network connecting the systems. Communication over the network is generally several orders of magnitude slower than it would be between processors running inside the same computer. The reason is that a network operates at a much slower speed than does the hardware connected to the computer's system bus. Data sent over a network must also have information added to it so that it can be sent to the right machine. The advantage here is that the operating system usually does not use as many resources as a numerically intensive application. Dual-processor machines can also be used so that the application runs on one processor and the operating system runs its tasks on the other.

Another form of distributed parallel processing is an Internet-distributed scheme made popular by projects such as *SETI@home*. In this scheme, users can download a client to run on their home computers. These computers will connect to a central server, download work units, and process them in the machine's spare time. When done, they will send the results back to the server and get another work unit (*SETI,* 2000).

This type of arrangement does have some problems, though. Processing is done on more of a voluntary basis. There is no guarantee that the work unit will ever be completed. The system must also be designed so that communication is minimized. There can be no node-to-node communication, and minimal communication with the server. Time-critical tasks also make poor candidates. As there is no dedicated, high- speed network connecting the nodes, and as processing is voluntary, there can be tremendous time lags before processing is finished. SETI-type tasks are well suited to this method because there are tremendous amounts of data, processing does not depend on other nodes, and such tasks are not quite as time critical (do not have the requirement of being done within X hours).

Approach

To examine the benefits of Beowulf technology, we had to choose an application that was well understood and that could significantly benefit from running in a distributed manner. The choice of applications also had to be representative of some of the current high-end processing that is going on within the NMD.

The choice was made to take the USGS Mid-Continent Mapping Center (MCMC) modified version of the Clarke Urban Growth Model (UGM) and port it to run on a Beowulf cluster. This software takes in multiple input layers of urban and land use data, runs three calibration phases on those layers, and then attempts to predict future urban growth. It outputs a series of graphic images that visually display the predicted urbanization of the study area. Computer scientists at MCMC were familiar with this software because several performance optimizations had been made to it for the Urban Dynamics project. It was also predicted that this software could be ported to a distributed parallel environment within the fiscal year.

Once a software system was selected, the cluster itself had to be put together. For this, the decision was made to collect sixteen surplus machines and upgrade them. Many of these systems had very old computer hardware that was in desperate need of replacement. An upgrade strategy was selected that would produce a low-cost, fairly homogeneous cluster. To keep the upgrade price to a minimum, we kept the cases, power supplies, video cards, network cards, and floppy drives from each machine. The rest of the computer hardware was replaced with new equipment.

Each node ended up with the following hardware components: FIC VA-503+ motherboard, AMD K6-2 500 processor, 128 MB of PC100 SDRAM, Tekram U2W SCSI controller, and an IBM nine-gigabyte hard drive. The 3Com 3C905 10/100 Ethernet cards, video cards, and floppy drives were kept from the original machines. Several machines also kept their original IDE CD-ROM drives. The FIC motherboards were chosen as they could function in both AT and ATX form-factor cases. The old hardware was then removed from each node and replaced over a period of several days. A sixteen port 100-Megabit Ethernet switch was purchased to allow the nodes to have a private, high-speed network.

Figure 1 MCMC Beowulf Cluster

As for software, the Red Hat Linux 6.2 distribution was chosen. This software is freely available from the Red Hat ftp site. This distribution was chosen as it includes the Parallel Virtual Machine (PVM) library from Oak Ridge National Laboratories (2000) as well as other software libraries necessary for this type of development. The Red Hat distribution also includes kickstart, an automated method of cloning an installation on different machines over a network (Red Hat, 2000). The distribution was slightly modified by the principal investigator to include all of the Red Hat 6.2 updates and the Linux 2.2.17 kernel release.

The installation process began with setting up an initial node with the Linux install. This involved partitioning the hard drive and installing various software applications and libraries. Once this node was verified to be operating correctly, a kickstart configuration was created to clone the setup on various machines. The other nodes were then connected to the Ethernet switch and automatically configured via the kickstart method. Once the systems were up and running, the kernels in each of these machines were then patched to use the vendor drivers for the SCSI and network cards. These drivers were necessary to correct some problems experienced with the default drivers included in the Linux 2.2.17 kernel.

The initial port of the UGM software was to a multithreaded application. This was in part due to delays with the cluster construction and hardware procurement. The primary reason, though, was that once it was broken up to run in multiple threads, it could then be easily extended to a distributed system when the cluster construction was complete. It also allowed initial experimentation to be done on how best to break apart the computational logic of the UGM. Debugging is also easier on multiple threaded applications as opposed to distributed applications.

Several designs were examined, with the final threaded design following a worker thread and dispatcher system. In this type of design, a master thread assigns work to the worker threads and then waits for them to finish processing. Once they are done, the master thread will dispatch more work if any remains. This is a fairly common practice in the design of multithreaded software, and it worked efficiently in this situation. This technique also required the least amount of rewriting of the existing software and allowed the original logic to remain intact.

As the UGM is mainly controlled by a series of nested programming loops, the decision was made to divide up the processing at the loop level. The master thread keeps track of the loop passes and dispatches the values to the worker threads. Once the work is dispatched, the master thread will then put itself to "sleep" for a period of time (it tells the operating system that it wants to temporarily stop running). The worker threads then process their data and output their results to a data file. When a worker thread is done, it places its thread identifier onto a queue data structure and then waits for a message from the master. The master thread then periodically checks the queue to see which threads have finished. If more work remains, the master removes the thread identifier from the queue and sends the worker thread the next series of loop values. At the end of processing, the master thread signals the worker threads to exit. The data files are then merged into one large file for examination.

After the multithreaded code was finished, it needed to be tested to make sure that splitting the processing into separate parts did not introduce any errors in the results. The baseline for this testing was the original single-threaded code that MCMC had been using for its urban dynamics studies. The software uses the Monte Carlo method (Woller, 1996), which uses a large amount of random numbers. This method is used in tasks such as deciding which areas should be examined for urbanization. The use of random numbers means that no two test runs will ever produce the exact same results. The prediction results should, however, be similar to a certain degree. The new code was tested on multiple platforms and compared with the baseline. The predictions were then examined and agreed with each other. Agreement in this case means that the differences in the output images were minimal.

With the multithreaded software finished, work turned to designing the distributed version of the UGM. The method chosen was similar to the multithreaded version, as many of the techniques remain the same. In this design, the master node would direct the processing of the worker nodes and wait for them to finish processing their data. Once finished, they would communicate with the master node through messages instead of a queue data structure. Communication between the nodes would be done by means of the PVM package.

In the implementation of the design, the master node starts and then spawns worker processes on the other nodes. When it creates them, it also passes them certain information, such as where the data is located on the network. The worker nodes then prepare for processing by reading in some initial data from the work unit. Once they have completed their startup processes, the worker nodes send a message to the master node that they are ready for processing. The master node, like the master thread, keeps track of the nested loops for processing and dispatches work to the nodes. When a node is done, it writes its data to a temporary file and sends a message to the master that it has finished processing that work unit. If any work remains, the master dispatches it to the worker.

As with the multithreaded version, the prediction part of the UGM was kept as a single thread. Rewriting this part of the software to run in a distributed version would have entailed considerable work and testing to verify that the logic had remained intact. When

compared with the calibration phases, the prediction component takes up a small part of the total runtime of the application. The decision was made to concentrate on reducing the wall clock time of the calibration phase and to leave the prediction phase intact to stay within the time constraints of the project.

Testing had to again be done to check that distributing the work of the UGM did not produce incorrect results. The sample data was again run through the baseline single-threaded application and compared with the results from the distributed version. As before, an exact match with the urbanization predictions was impossible because of the way that the model performs its calculations. The results were examined visually to check that the growth patterns were similar between the different programs. The images were also tested by subtracting them from each other so that only the differences remained. The differences were examined and found to be small enough to conclude that the integrity of the original UGM logic was preserved.

Testing

Once the software was debugged, we needed to run time trials to determine the effects of distributing this software. All of the time tests used the data50 sample dataset that was included with the UGM software. This dataset is a 50x50-pixel, low-resolution region of a typical urbanized area. It comprises of a series of Graphics Interchange Format files of various themes, such as urbanized areas, land slopes, and waterways. The urbanization files cover various years and are used by the calibration process to fit its parameters to the specific area at hand. The other files are used in the urbanization calculations, such as limiting urbanization to areas where the slope of the land is less than a certain angle.

Each of the tests was performed in the same fashion. They were all run five times. The runtimes of each of the tests were then averaged to produce the final time for the test. The tests were all recorded to an accuracy of one second. The timing was done using the standard Unix *time* utility. This utility calculates the total runtime of the program, as well as statistics such as percentage of processor time used.

Multiple tests were performed with the software. The single-threaded version was first tested on machines of various computer architectures around the Mapping Center. This was done to compare how different types of computer processors could manipulate the data, as some processors are better at certain types of operations than others. The tests included Sun UltraSPARC, SGI MipsPRO, and AMD K6/2 processors running the Linux, Irix, and Solaris operating systems.

The distributed version of the software was tested in a similar manner on the cluster. The time trials were run on different numbers of nodes. This was done to show the effects of distributing the UGM to quantify the time differences. It was hoped that a function that establishes the relationship between the number of nodes and the runtime could be found.

Results

As previously mentioned, testing was first done with the single-threaded version of the model on different computer architectures. Figure 2 shows the runtime of the UGM software on some of these systems.

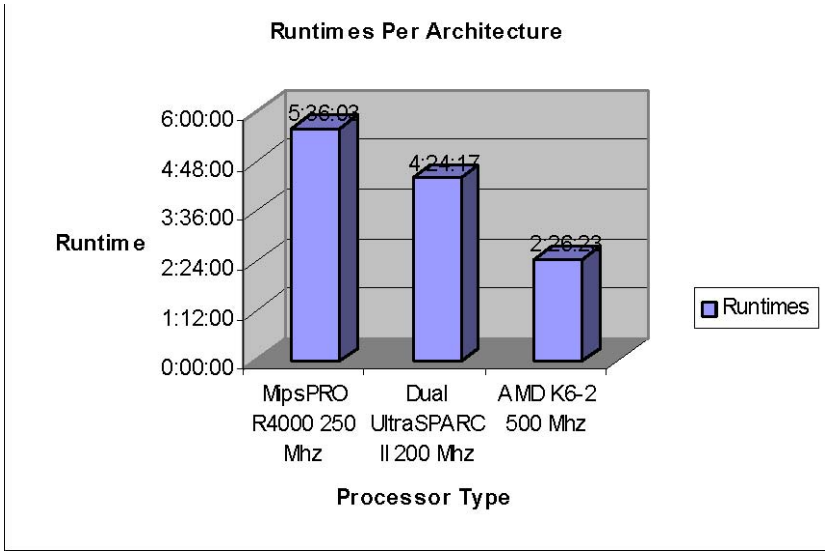

Figure 2. Runtimes on various processor architectures

Figure 2 illustrates several points about types of processors. Even at similar clock speeds, certain architectures can process UGM data faster than others. One of the primary reasons for this is that the model calculations make extensive use of floating point numbers in both the calibration and prediction phases. As an example, there are millions of random numbers that must be calculated for application of the Monte Carlo method, and most random number generation algorithms are dependent on floating-point arithmetic. Some processor architectures are much faster at doing these types of computations than others. Desktop processors, such as those from Intel and AMD, are usually slower at performing floating-point calculations than workstation processors, such as Alphas and UltraSPARCs (Engstorm, 2000).

This information is important when determining what type of hardware should be used in the construction of a cluster. Processing that is numerically intensive and uses floating-point operations extensively will benefit more from workstation-class processors rather than those found on the average user's desktop. It should be noted, though, that while the high-end workstation processors perform calculations faster than the desktop processors, they are also more expensive.

When a job is run on a cluster, the runtime is drastically reduced up to a certain point by adding more processing nodes. Figure 3 illustrates the reduction in runtime when adding multiple processing nodes in the cluster. The times used were the average results of running the software on various numbers of processing nodes. The dashed line indicates

a best-fit power curve through data points. The equation of the curve and the statistical R^2 value are displayed in the upper right-hand corner of the graph.

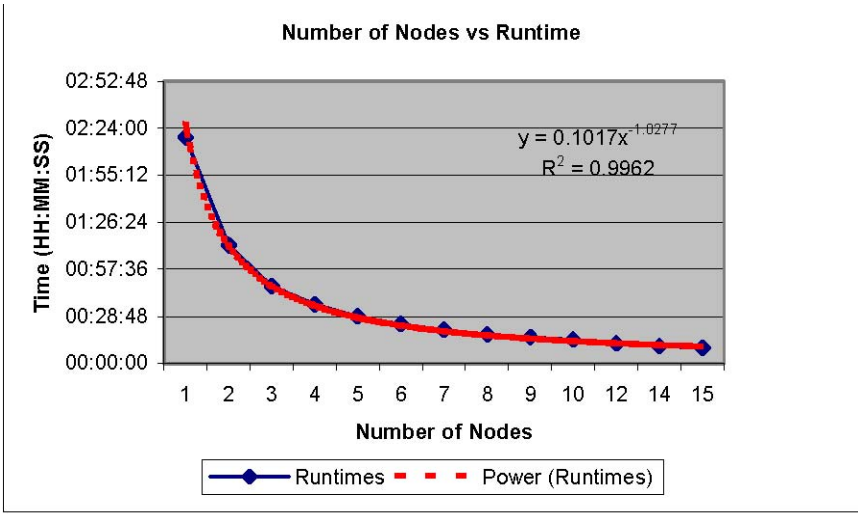

Figure 3. Number of nodes vs. runtime.

The processing time based on number of nodes is closely modeled by the function $y = 0.10173 \bullet x^{-1.0277}$. The statistical R^2 value indicates how closely the curve fits the data, with a value of 1.0 being an exact regression match. In this case, the R^2 value of 0.996 indicates a very close fit between the estimated runtimes and those actually recorded.

As can be seen in Figure 3, the effects of adding processing nodes have several characteristics that should be examined. Initially, adding more processing nodes causes a significant decrease in the processing time of the UGM. With a single processing node, it takes approximately two hours and eighteen minutes to process the calibration. But simply increasing the number of processing nodes to four drops the runtime to approximately thirty-six minutes. This small number of additional nodes causes a 73.9-percent decrease in the runtime for the calibration. The numbers may seem insignificant at this scale, but consider a process that may take thirty-one days to run. A 73.9-percent decrease in runtime would result in a new runtime of only eight days.

The runtime effect of adding more processing nodes begins to decrease after about seven processing nodes. After this point, the above curve can be seen to flatten out, and it adheres to the Law of Diminishing Returns. This means that adding additional processing nodes past this point does not have as significant an impact as it did initially. Figure 4 is based on the above power curve function taken out to thirty nodes.

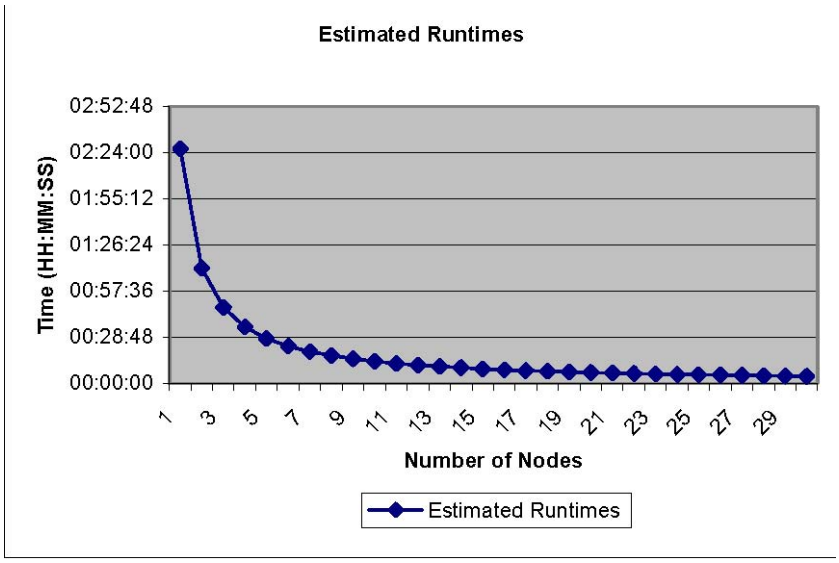

Figure 4. Estimated runtimes.

As can be seen, the cost/benefit ratio of adding additional nodes becomes much smaller past a certain point. Going from fifteen to thirty nodes, for example, only decreases the runtime from nine minutes to four and a half minutes. The cost of adding an additional fifteen nodes can be rather large, though, depending on what equipment is used to construct the cluster. This demonstrates that care must be taken when constructing such a system, as large resource investments do not always reap equally large benefits in processing ability.

Another point that must be made is that this function curve will not completely hold true. The function shows that with a huge number of nodes, the processing could get to within a few seconds of zero runtime. This is not totally accurate as there are effects of the networking hardware and software that will keep this bounded. There is also some overhead involved with the PVM communication between the nodes on the network that affects the processing time of a large number of nodes.

The figures also illustrate an important point that must be realized about this type of processing. There is not always a linear relationship between the number of processing nodes and the time it takes to process. As with multithreading, adding another processing node does not mean that the processing time will always be halved. The above function curves will not hold true for all types of applications, either. Many nodes may need to be added before significant benefits can be realized. This type of curve is common in parallel or distributed processing, though.

Discussion

In addition to discussions about the data, several lessons were learned during the design and development of the software that should be discussed. One of the primary reasons for this project was to gain experience and insights in developing distributed-processing applications. Other goals included examining and understanding the theories related to

this type of processing, and learning how to apply them to scientific research within the USGS.

The first lesson was that it is much easier to design and develop some projects in a distributed environment than it is in a multithreaded environment. One of the primary problems with developing multithreaded software is the issue of synchronization, especially with shared memory. With multithreading, the software must be designed so that the threads will not corrupt the shared memory areas. Corruption is usually caused by changing something in memory at the same time another thread is trying to use it. This problem is solved through various mechanisms, the most common being a mutex lock. A mutex lock can be thought of as a flag that one thread uses to alert the others that it is currently using that resource. Sometimes this is enforced by the operating system; other times it is voluntary between the threads.

In a distributed environment, each node has its own memory resources and only shares data when it sends it to another node through some type of message over the network. This removes a large part of the synchronization complexity from which multithreaded applications suffer, and it can lead to cleaner and simpler software designs. Applications may also realize a slight speed increase, as they do not have to continually check for and wait on synchronization mechanisms.

The real difficulties with designing and implementing this type of software come from the large number of ways that a problem can be designed to run over a network. While the problem may have multiple solutions, only a few of them can be done efficiently. A design may be distributed along the dispatcher/worker lines and end up flooding the network to such a degree that processing actually takes longer than it would with a single-threaded design. Computer science has done much work in designing distributed algorithms, but many of these algorithms are very complicated and require careful implementation to function properly. The UGM was a case that scaled to a distributed design very well, but not all applications can be developed using the same type of design that a multithreaded application uses.

Although a distributed system may not suffer from the same types of synchronization problems that a single-threaded system does, distributed systems do introduce their own problems with synchronizing processing. An example of this can be found with the dispatcher/worker design. In this case, the dispatcher cannot assume that work units will be returned in the same order in which they were dispatched. A cluster may be made up of machines of different speeds that will be processing data at different rates. The work units will be returned in almost any order, meaning that the design must be able to deal with the situation properly. Even homogeneous clusters suffer from this problem. Two machines may have processors of the same type and speed, but will not process at the same rates due to subtle differences in the hardware and state the operating system is in at the time.

Machines of different speeds also introduce issues with load balancing. Load balancing is a technique used to try to keep each machine from being overloaded or under utilized. This area of computer science is still being extensively researched; no one has yet found the perfect scheme. Many efforts use a simple round-robin approach, in which data is

sent to the next available computer. The problem with this technique is that it can cause computers to be overloaded and can seriously affect processing synchronization. Other, more complex techniques are also available, but they have their own pitfalls, ranging from extreme complexity of implementation to large amounts of processor time for calculations.

The types of processing that USGS scientific programs perform also introduce their own unique problems. In many cases, the processing not only is numerically intensive, but also has large amounts of data that must be processed. As another fiscal year 2000 project at MCMC showed, large amounts of data could easily flood a cluster's network and slow the processing down. This was the case in projecting a standard USGS Digital Orthophoto Quadrangle from Universal Transverse Mercator to the State Plane Coordinate System. Systems that handle large amounts of data must be carefully planned and designed to handle these types of data processing issues.

There are other issues to be considered when designing clusters to perform data bound processing. A theory proposed by Dedkov and Eadline (1995) states that for data-intensive processing, multiprocessor systems outfitted with the latest high end processors may actually be slower than systems with slower processors. While this may seem counterintuitive, the explanation lies with the inner workings of a computer. Devices on a computer are all connected through some type of system bus. Basically, high-speed processors can flood the system bus and slow the transfer of data. This is because the processors must constantly communicate with each other to keep synchronized. Faster processors must communicate more often with each other as they can process more information during one time unit than slower processors can. Although this theory was proposed more for multiprocessor systems, it is also applicable to cluster environments, as multiprocessor nodes will generally perform better than single-processor ones for numerically intensive processing tasks.

Future Work

Some issues should be examined in the future to improve the performance of this type of technology in USGS scientific tasks. The first issue to study would be how to process large amounts of data efficiently over a cluster. With decision support and emergency response research on the rise, the amount of data that needs to be processed will continue to increase. Efficient techniques in processing this much information will help to ensure that these activities can be completed in a reasonable amount of time.

Another related issue is to determine how theories such as the one presented by Eadline and Dedkov affect data processing on a cluster. The theory is applicable to multiprocessor nodes in the cluster, but the effects on the cluster as a whole should be studied. This work could help determine what types of equipment are best suited for certain tasks. It may well turn out that spending large amounts of money for the fastest equipment may be a bad investment if the cluster is to process primarily data-intensive tasks.

Conclusion

Distributed-processing systems are a viable technology that the USGS can turn to in order to fulfill its high-end processing needs. Beowulf cluster technology can cheaply construct supercomputer-class processing systems from aging equipment that can be found throughout the organization. This processing capability can then be applied to tasks ranging from map projections to emergency response situations. This can enhance current capabilities or it can be used to allow scientists to perform tasks that were previously impossible.

The technology does have certain issues that must be studied before a group attempts to construct a cluster for its work. The system must be tailored depending on whether the work is processor or data intensive. Issues such as problems with processing large amounts of data in a distributed fashion must be resolved. A group must first decide what it hopes to achieve in order to keep from spending large amounts of money to build a system that does not fully meet its processing needs.

This is the type of technology that the NMD must pursue in order to fulfill its future missions. Technology must be studied and developed to produce the wide range of custom digital data that is being requested. Emergency response systems must be equipped with the technology to perform time-critical tasks. Studies such as those performed by the Urban Dynamics group are in need of large processing capabilities to advance research in predicting how humans will affect the land.

References

Avalon. Los Alamos National Laboratories. <http://cnls.lanl.gov/avalon/>.

Beowulf Mailing List FAQ, version 2. Sitaker, Kragen, et al. <http://www.dnaco.net/~kragen/beowulf-faq.txt>.

"Computers in Your Future. Lesson 1B-The Historical Perspective on Computers". Desert Chapel. <http://www.desertchapel.org/ciyf/lesson1b/>.

Dedkov, A. F. and Eadline, D. J., 1995, *Performance Considerations for I/O Dominant Applications on Parallel Computers:* Paralogic Corporation, <ftp://www.plogic.com/pub/papers/exs-pap6.ps>.

"Kickstart Installations". Red Hat, Inc. <http://www.redhat.com/support/manuals/RHL-6.2-Manual/ref-guide/ch-kickstart2.html>.

Dietz, Hank, 1998, Linux Parallel Processing HOWTO: Purdue University. <http://yara.ecn.purdue.edu/~pplinux/PPHOWTO/pphowto.html>.

Nichols, Bradford, Buttlar, Dick, and Farrell, Jacqueline Proulx, 1998, *Pthreads Programming*: O'Reilly.

Govett, Mark, 2000, *NOAA/FSL/AD Advanced Computing Branch,* <http://armstrong.fsl.noaa.gov/ac/index.html>.

Engstrom, D. D. "Processor Performance Comparisons". Mac Speedzone, <http://www.macspeedzone.com/4.0/WinvsMacSPECint.html>.

PVM: Parallel Virtual Machine. 5 Sept. 2000. Oak Ridge National Laboratories. <http://www.epm.ornl.gov/pvm/pvm_home.html>.

SETI@home: Search for Extraterrestrial Intelligence at Home. Search for Extraterrestrial Intelligence. <http://setiathome.ssl.berkeley.edu/>.

Woller, Joy, 1996, "The Basics of Monte Carlo Simulations": University of Nebraska – Lincoln. <http://wwitch.unl.edu/zeng/joy/mclab/mcintro.html>.

Wilson, G. V., 1994, *The History of the Development of Parallel Computing:* Virginia Polytechnic Institute and State University. <http://ei.cs.vt.edu/~history/Parallel.html>.