USGS
science for a changing world

Processing Large Remote Sensing Image Data Sets on Beowulf Clusters

By Daniel R. Steinwand,[1] Brian Maddox,[2] Tim Beckmann,[1] and Gail Schmidt[1]

Open-File Report 03-216

[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

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

Contents

Illustrations

Key Words

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

Abstract

High-performance computing is often concerned with the speed at which floating-point calculations can be performed. The architectures of many parallel computers and/or their network topologies are based on these investigations. Often, benchmarks resulting from these investigations are compiled with little regard to how a large dataset would move about in these systems. This part of the Beowulf study addresses that concern by looking at specific applications software and system-level modifications. Applications include an implementation of a smoothing filter for time-series data, a parallel implementation of the decision tree algorithm used in the Landcover Characterization project, a parallel Kriging algorithm used to fit point data collected in the field on invasive species to a regular grid, and modifications to the Beowulf project's resampling algorithm to handle larger, higher resolution datasets at a national scale. Systems-level investigations include a feasibility study on Flat Neighborhood Networks and modifications of that concept with Parallel File Systems[2].

[2] 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.

High-performance computing is often concerned with the speed at which floating-point calculations can be performed. The architectures of many parallel computers and/or their network topologies are based on these investigations. Often, these benchmarks are compiled with little regard to how a large dataset would move about in these systems. This part of the Beowulf study addresses that concern.

Background

The USGS has many computational processes that involve large numbers of floating-point operations but also rely on moving vast amounts of remote sensing image data from place to place during the computation. Since Beowulf clusters are often constructed around commodity networking equipment, it is apparent that the USGS must study the movement of large datasets among cluster nodes and must tune its cluster implementations and applications software to move that data in the most time-efficient method possible.

Hypothesis: Techniques and methodologies can be developed to allow a Beowulf cluster to efficiently process very large remote sensing datasets.

This report describes the approaches taken at the EDC and at the MCMC regarding the efficient processing and movement of large amounts of image data. The EDC applications software is described first, followed by system-level investigations performed at MCMC.

Applications

The EDC staff developed several image processing applications to support project work at the Center. Included are algorithms to smooth Normalized Difference Vegetation Index (NDVI) time-series data (input data to the Global and Continental Landcover Characterization activities), an algorithm to compute decision trees for image classification (used by the Landcover Trends project), a Kriging algorithm (used by Biological Resources Division (BRD) Midcontinent Ecological Science Center (MESC) in its Invasive Species studies), and additions to the tile-based resampler to support pixel-by-pixel projection change calculations. Each of these algorithm implementations is described below.

NDVI Time-Series Data Smoothing

Time-series data from the Advanced Very High Resolution Radiometer (AVHRR) satellite consisting of NDVI composites of multiple satellite passes are smoothed to reduce noise and other artifacts present in the data. These datasets are often very large in their X and Y dimensions (typically 3,025 lines by 6,331 samples) and often have 200 or more bands of data. A typical dataset spanning 4 years would be approximately 4 gigabytes in size.

Much of the processing of these data is band by band (time slice by time slice), so the data are stored in a band sequential (BSQ) format. The smoothing operation, however, operates in the time dimension only. For the smoothing algorithm, the data would be better stored in a band interleaved by pixel (BIP) fashion, but this data orientation does not lend itself to most of the processing done to the data, both before and after smoothing. The challenge of the parallel algorithm was not only to process the smoothing filter computations faster but to optimize the input/output of the data without rewriting the image into a BIP format.

The smoothing algorithm (Swets and others, 1999) uses a weighted least squares approach designed specifically for this type of data. Because the algorithm is complex and is often modified and refined by project investigators, the Beowulf project investigators decided that it was desirable to use the smoother unmodified, if possible.

Since the smoothing algorithm operates only in the time (Z) dimension of the data, the surrounding pixel values in the X and Y dimensions have no bearing on the calculations. Therefore, the parallelization of this algorithm can be approached from a data-partitioning point of view. This condition, and the fact that the Beowulf project team decided to use the smoothing filter un-modified, led to a parallel implementation that is run entirely from the command shell using remote processes.

An additional consideration given to the implementation was that of program conversion time and effort. For this class of problem, where the data may be partitioned and the computational algorithm would remain as-is, could the program implementation time be minimal?

Two generic utilities were written to accomplish the data partitioning for this type of problem. "Splitbyrows" divides the imagery into N separate images, with each resulting image having the same X and Z dimensions as the original. N is chosen so that it equals the number of CPUs available to the processing job (Note that this is taken here to be the number of available processors, not the number of machines in the cluster. The idea here is that dual processor machines should be doing twice the work.) "Mergerows" is the inverse; it takes the segmented files and merges them back into a contiguous image. It is applied after the smoothing algorithm. All data in this process are (and remain) in a BSQ format.

For a given input image, "splitbyrows" produces N output images:

Input: inImage
Output: inImage.blk1, inImage.blk2, … inImage.blk*N*,
where N is the number of blocks

The next step in the process is to distribute the data among the nodes of the cluster, one dataset per processor. This is done with simple remote process commands (rcp and rsh in this case).

```
> rcp inImage.blk1 processor1:/data/inImage.blk1
> rcp inImage.blk2 processor2:/data/inImage.blk2
:
:
> rcp inImage.blkN processorN:/data/inImage.blkN
```

When the data are distributed, the smoother is run on each processor in a free-for-all fashion. There is no load balancing.

```
> rsh processor1 "bin/smooth /data/parmfile1" &
> rsh processor2 "bin/smooth /data/parmfile2" &
:
:
> rsh processorN "bin/smooth /data/parmfileN" &
```

Note that the smoothing algorithm requires parameter files for each dataset.

The last step in the process is to retrieve the processed datasets using the following:

```
> rcp processor1:/data/outImage.blk1 /localdata/outImage.blk1
> rcp processor2:/data/outImage.blk2 /localdata/outImage.blk2
:
:
> rcp processorN:/data/outImage.blkN /localdata/outImageN
```

and run mergerows on the resulting image strips to create the final output image.

Program runtimes, including the splitting, copying, and merging operations are given in figure 1. It should be noted that for one CPU, splitting, copying, and merging operations are unnecessary and were not included. Times are given for 12 nodes of the EDC Beowulf cluster, with the dual processor master running two processes. A speedup of 8.12 times is noted for the parallel job over the serial version.

Task	Single Process	13 Processes	Speedup Factor
Split the image by rows	n/a	2 min, 20 sec	
Distribute data to nodes	n/a	2 min, 1 sec	
Process the image	9 hrs 33 min 34.4 sec	48 min, 9 sec	11.91 times
Retrieve processed pieces	n/a	12 min, 10 sec	
Merge the image	n/a	6 min, 1 sec	
Total Time:	9 hrs 33 min 34.4 sec	1 hr 10 min 41 sec	8.12 times

Figure 1. Parallel NDVI smoothing performance data

The programming effort for this type of parallelization is minimal. The splitting, copying, and merging routines are generic and can be reused. This type of parallelization should be considered if the above condition of data partitioning is present. The process takes only a couple of hours to set up and run. If the process is to be run on a frequent basis, however, the program implementer may wish to automate the process of script creation to reduce typographical errors.

This type of parallel program is portable to several computer architectures. Networks of workstations and Beowulf clusters look identical to this algorithm. Symmetric mulit-processor (SMP) machines also work well with this class of algorithm, and the network copy of image segments is not necessary.

Decision Trees

The “c50mapi” application is an image classification program that uses a decision tree to determine the landcover classification of a given ground location on the basis of a set of rules for the decision tree. This algorithm is used in the Landcover Trends Project to periodically classify landcover for the Nation. Input to the algorithm includes a set of image files (Landsat and other image layers) and rules for the decision tree. Training (to determine the rules) must be done before running the algorithm; the program writes a class image and a confidence image.

The decision tree algorithm can, as in the NDVI smoothing application above, be approached with the data-parallel scheme, as the data are processed in the Z-

dimension only. The processing of neighboring pixels (in X and Y) is independent of each other, allowing one to split the input data into separate blocks for processing. The algorithm can be run in its original mode or in boost mode. Boost mode requires more processing time, because the algorithm builds a number of sequential trees to improve the accuracy of the classification at the cost of added runtime.

The objective of this task was to write a “wrapper” that would enable the “c50mapi” application to be run on a Beowulf cluster, multiprocessor Sun workstation, and/or a cluster of Sun workstations. Since the software was originally implemented on a Sun workstation, the multiprocessor Sun implementation was constructed first, and the results are presented here. At the time of this writing, the Beowulf implementation had not been completed.

A Perl script is used for job control, because Perl is supported on each of the target platforms. This script splits the input image data files according to the number of processors or cluster nodes, runs “c50mapi” on the subsets of data (one per processor/node), then merges the resulting classifications and confidences subsets into one large classification and confidence file. The processing flow of the program is shown in figure 2.

Results of running the serial version and the parallel version on a four-processor, shared-memory Sun SPARC workstation are shown in figure 3 below. The small dataset has 20 input data files with 5,829 lines x 214 samples each. The full-sized dataset has 20 input data files at 20,022 lines x 21,301 samples each.

The results on the small data set indicate that speedups are possible. The overhead for splitting the input files is 26 seconds, and in the boosting mode where the processing time is twice as long, this overhead time is short compared with other type of processing.

The results on a full-sized dataset are not as promising. Speedups were observed when using two processors, but when using three or four processors, the program became data bound and speedups were not achieved.

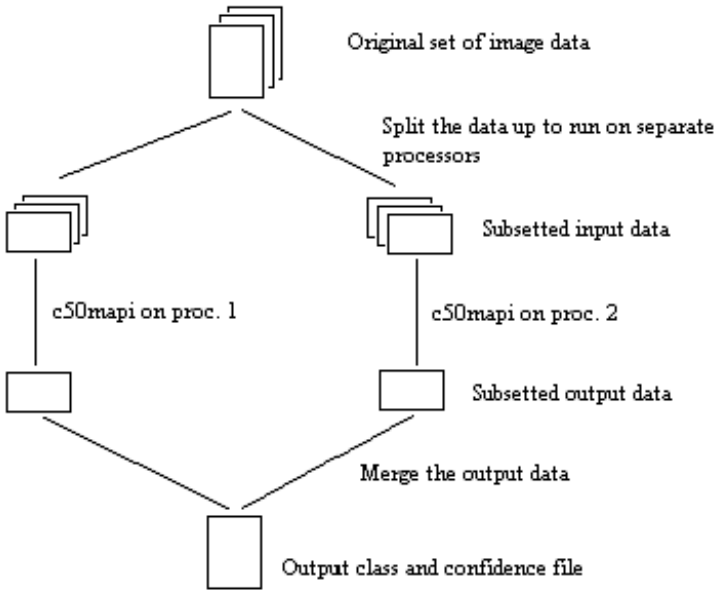

Figure 2. Breaking up the input data and merging the output image data.

The data input and output issues observed during tests with the four-processor SMP Sun system are not expected to be present on a Beowulf system. The overhead time to split the input files into subsets would be similar, and added time will be needed to send each image subset to its specified node, but the I/O contention that was observed in the SMP experiment should not be present. This part of the task is still in the implementation stage.

# Processors	Time	% Speedup
1 processor	214.9 sec	
2 processors	140 sec	53%
3 processors	115 sec	89%
4 processors	99 sec	117%

Figure 3. Small dataset: 20 input image data files, 5,829 lines x 214 samples

Kriging Algorithm

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. This algorithm is used by the BRD/MESC in its study of invasive species. The Beowulf project was asked to parallelize an implementation of a Kriging algorithm to run on a Beowulf cluster in an attempt to reduce processing times. The

Message Passing Interface (MPI) Applications Programmer Interface (API) was chosen for implementing the algorithm on the cluster.

The original algorithm was implemented in FORTRAN. The first step was to convert the main processing sections of the code to C to make the use of MPI easier. The conversion to C was straightforward; the only challenges were caused by slightly different results obtained owing to floating-point round-off errors. Most of the numerical subroutines remain in FORTRAN for this implementation.

A high-level view of the original Kriging algorithm shows that it does the following:

- Reads the input parameters and set of point measurements (x,y,value tuples)
- For each point in the output grid
    - Finds the N points closest to the grid point
    - Estimates the value at the grid point on the basis of the N closest points
    - Writes the results to a file

To port this to use MPI efficiently, we split the algorithm into master and slave node parts. The master node algorithm is as follows:

- Read the input parameters and set of point measurements
- Broadcast the parameters and points measurements to the slave nodes
- Assign each slave node a row of the grid to calculate
- Loop until the work is done
    - Receive results from the slave and save them in memory
    - Send the slave more work
- End the loop
- Shutdown the slaves
- Write the results to a disk file

The slave node algorithm is as follows:

- Receive the input parameters from the master node
- Receive the set of point measurements from the master node
- Loop until told to exit by the master node
    - Receive a row assignment
    - Calculate the grid points for the row assigned
- Send the results to the master node
- End the loop

Assigning a row of the output grid to the slave nodes provided a good balance between minimizing communications with the master and slave nodes and assigning small packets of work to the slave nodes to help load balance the system.

In actuality, the master node algorithm is a little more complex than shown. To support processing any size of output grid, the master node periodically stops assigning work to slave nodes, waits for all of them to send back outstanding

results, writes the output to disk, then starts assigning work again. This is done to limit the memory used by the application. All the results are needed from all the slaves so that a part of the output file can be written in sequence. The output file can be written optionally in ASCII with variable length fields, so the results must be written in order to match the original output format.

The MPI version of the application attained a nearly linear speedup for each node added.

Beowulf Tile-Based Image Resampler Model Additions

This work extends the Tile-Based Image Resampler that was implemented in FY 2001 (Crane and others, 2001). In FY 2002, a pixel-for-pixel cartographic projection-based model was added. The projection model uses the General Cartographic Transformation Package (GCTP), along with the Land Analysis System (LAS) datum transformation layer, to perform a precise image projection change transformation. The addition of this model was one of the improvements suggested in last year's Open File Report.

This model was added when a production user request was made to reproject a digital elevation model (DEM) image of the continental United States to a resolution of 240 meters. None of the existing resampling tools at the EDC (commercial or those written in-house) were able to perform the task because of memory limitations and the sheer size of the input imagery. To accomplish this task, we reprojected the image by applying the existing grid model in the previous version of the Beowulf resampler. The grid itself turned out to be over 1 gigabyte in size to maintain the error tolerances expected. The slave nodes on the EDC cluster have 256 MB of RAM and a 1 gigabyte swap space. This proved to be insufficient to run the task. The master node has 768 MB of real memory and 1 gigabyte of swap space. It was able to perform the resampling task running the resampler without the MPI support enabled. However, it took more than 8 hours to resample the 4.5 gigabyte input image to a 700 MB output image. The long runtime was due to the constant memory swapping that occurred. At that point, it was decided that the projection model would be beneficial in this case.

The projection model performs a coordinate transformation for each pixel in the output image to find the location of that pixel in the input image. It then performs resampling using either nearest neighbor, bilinear interpolation, cubic convolution, or table-based resampling with kernel sizes of up to 16 x 16. Computationally, it is much more expensive to do a coordinate transformation for each pixel than to use a grid for approximating the mapping from output space to input space. However, it does avoid the memory use of the grid and is able to work where the grid model falls apart, such as with projections that are not

contiguous or do not have a one-to-one mapping relationship. The Interrupted Goode Homolosine projection is a good example of this (Steinwand, 1994).

Adding the projection model proved that the design of the resampler is flexible. The projection model code was written as a single source code module and required almost no changes to the rest of the code. No modifications were required to the MPI-related code.

After the projection model was implemented, it was applied to resample the U.S. DEM image to 240 meters using the cluster. It took approximately 1.25 hours to complete the resampling task. This is more than six times faster than doing the same task using the grid resampler on the master node. This is not a direct comparison since the computational cost of doing a coordinate transformation for each pixel is several times higher than the cost of using a grid to approximate the transformations. The projection model was not used on a single node to get a comparison of runtimes. Later, the same image was resampled to120 meters, and it took approximately 1.5 hours. This suggests that the resampler is still bandwidth limited since it performed four times the computations and it only resulted in 0.25 hours of additional time.

When the projection model was working, an option to check for wrapping around the edge of a world map was added. When the wrap around is detected, the projection model treats the transformation the same as it would a break in a projection. The result in the resampler is that areas that are considered to be wrap-around areas are filled with a fill pixel in the final output. This is useful for world projections such as Robinson, Sinusoidal, Mollweide, and Interrupted Goode Homolosine.

An additional coordinate transformation is needed to detect when the wrap around case happens. After the coordinate transformation of a pixel location from the output projection to the input projection is performed, the input projection location is transformed back to the output projection. If the original and newly calculated output coordinates disagree, wrap around has occurred. (Because of limitations of the GCTP and the LAS datum transformation layer, GCTP was used to transform the coordinates in both directions for the test, and then the datum layer was used to do the final transformation. In the future, these limitations need to be eliminated and the proper method used).

Systems-Level Investigations

Research also continued at MCMC in FY 2002 in processing large amounts of data. Research focused on several areas: Flat Network Neighborhood (FNN), distributed file system through the Parallel Virtual File System (PVFS), and MOSIX and its use in large-scale data processing.

Flat Neighborhood Network

The FNN is a method of increasing the bisection bandwidth over a network. Typically, many machines are connected to a network switch that allows them to communicate with one another. However, often there are more machines on a network than there are connections on the switch. When this happens, the switches themselves are connected through uplink ports. This can hamper attempts to pass large amounts of data through a network because the single uplink connection causes a bottleneck when multiple machines on different switches need to communicate. Numerous network collisions can occur in this case because every machine on a switch must go through the uplink port to communicate with a machine on another switch. A collision means that packets sent from multiple machines interfere with each other and are either dropped or retransmitted again after a small, random period of time. FNN technologies are an attempt to solve this problem by using multiple network devices per machine and multiple switches so that each machine must only pass through one switch to communicate with any other machine. FNN does this by setting the routing so that different machines have to take different routes to access another machine. An example of an FNN system is presented in figure 4.

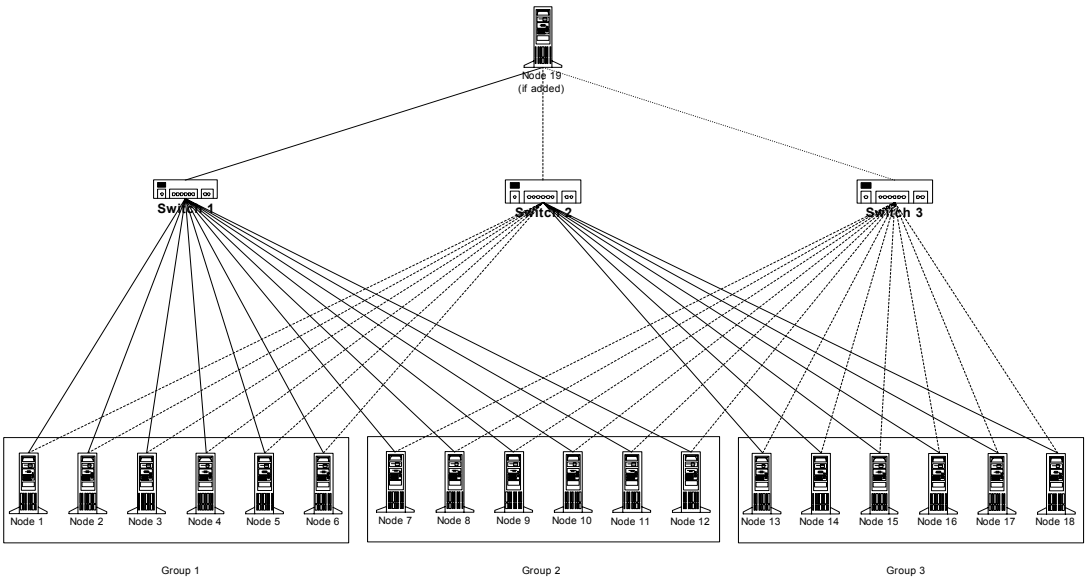

Figure 4. MCMC FNN setup.

In addition to solving the network bottleneck problem, FNN designs provide additional communication channels into a machine. In theory, these additional channels could allow a single machine to process multiple network requests more quickly. For example, consider the case where machine A is on an FNN and has two network devices in the machine. Two other machines, B and C, are

also on the FNN and wish to communicate with A. In a normal network, both B and C would attempt to communicate with A, yet only one at a time would be able to send packets to A. In an FNN setup, B and C would communicate with A through different communication channels. Neither machine would interfere with the other, and both machines could send and receive from A at the same time.

Testing was done to check whether FNN designs were indeed faster in transferring data and also to determine how much faster an FNN would be than a traditional network arrangement using a single interface. For testing purposes, a file copy using the standard Unix *cp* command was sent from multiple machines to a Network File System (NFS) share on a single machine. Two tests were done: one using a single interface and one using the FNN configuration. In the single interface tests, each test consisted of one to five machines copying to the receiving node at the same time. The FNN case consisted of an equal number of nodes per interface simultaneously copying to a single receiver. Additionally, these tests were repeated by copying to a single and to a dual processor machine to determine any differences an additional processor could have in responding to network packets. In all cases, each test was done several times and the results were averaged together for the average maximum, minimum, and

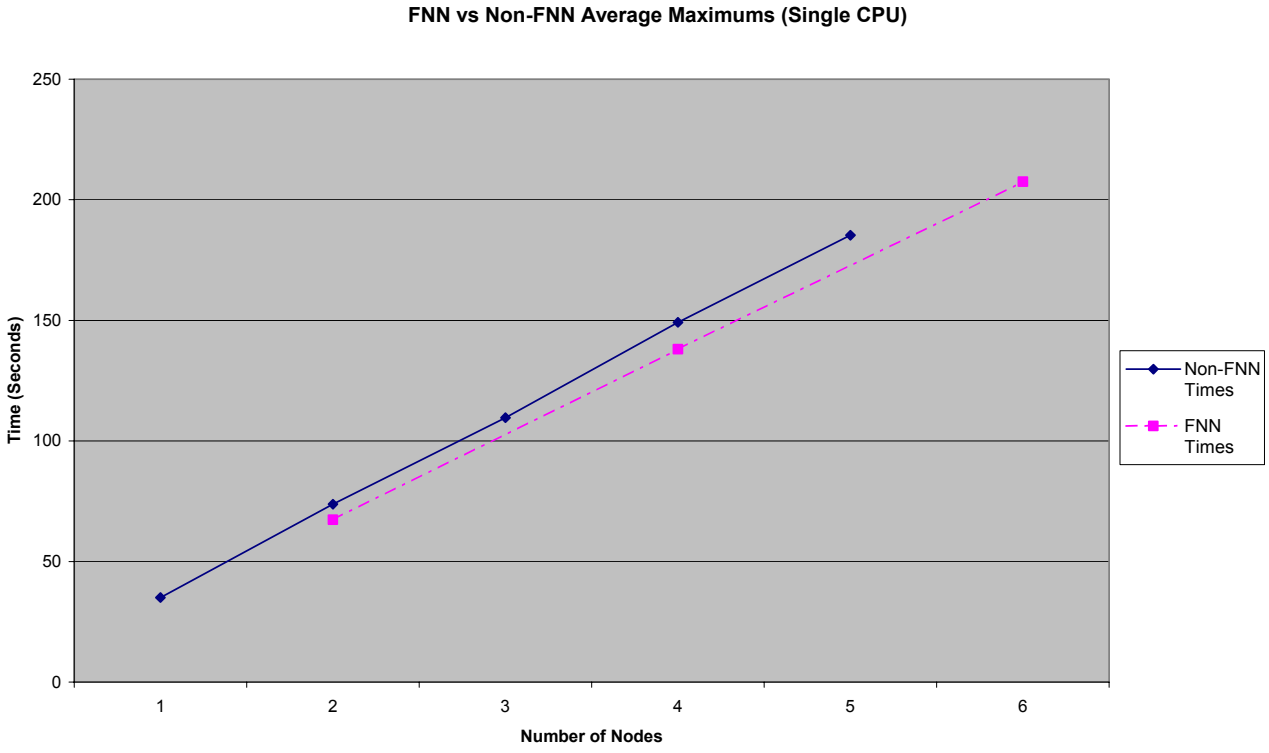

Figure 5. FNN versus Non-FNN for a single processor.

overall average times. Between test runs, a software application was run that flushed the operating system's disk buffer. This was done to prevent caching from interfering with the test results.

The first set of tests involved copying to a single machine, both with and without FNN, and the comparison of the average maximums can be seen in figure 5. The average maximums are compared here, as total runtime is dependent on the slowest copy time. This is much like the total runtime of a distributed system being dependent on the slowest processing node. As figure 5 shows, the FNN times were slightly faster than the non-FNN copy times. The results of this test also demonstrate that for this type of data access pattern (and probably most patterns), FNN will not result in double the performance simply because there are multiple network cards as opposed to a single one. To understand this, we must consider what goes on "inside the box." The operating system can only service so many things within a given time period. The computer hardware also has a limitation to how much data it can transfer internally between things such as processor, memory, and hard drive. Although adding network interfaces can provide multiple communication channels, limitations such as the operating system response time and hardware bandwidth will limit the throughput realized. This all affects the maximum amount of information that the system can process during a given time period.

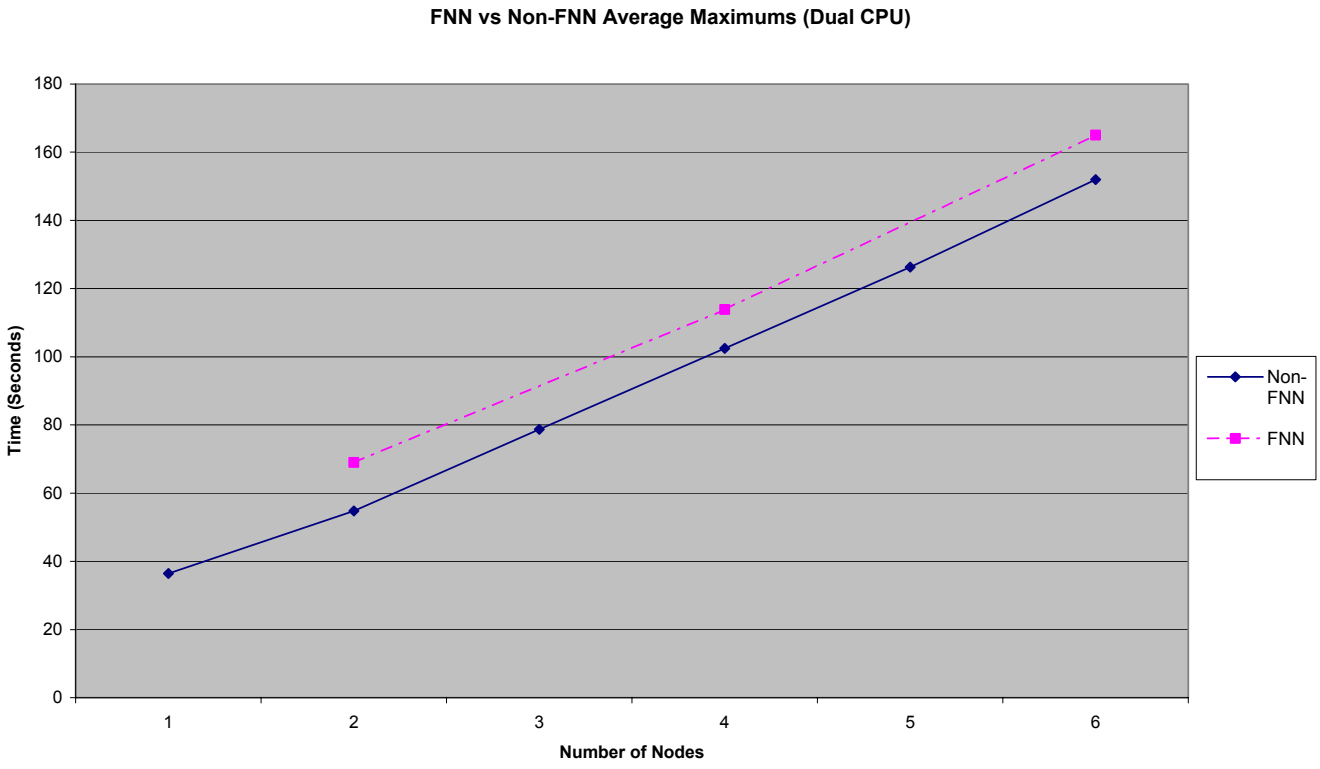

Figure 6. FNN versus Non-FNN for a dual processor.

Testing was then done on a dual processor target that has more memory and more modern and faster computer architecture. Because this architecture had more internal bandwidth and an additional processor, it was assumed that it could handle the single interface and FNN tests more quickly than the older single processor machine used in the previous tests. However, as figure 6 shows, this was only partly correct.

The dual processor machine is indeed faster at receiving data through the single interface. For example, five nodes copying to the dual processor machine finished approximately 58 seconds faster than when copying to a single processor machine. This would seem to verify that the increased internal bandwidth of the dual processor machine, combined with the ability to perform multiple tasks simultaneously, provides the dual system an advantage at receiving data through a standard network configuration. The maximum copy times are again compared here because they represent the completion of the task as a whole.

The dual processor copy times, however, were actually slower with the FNN than with a traditional configuration. As can be seen in figure 6, the average maximum times for file copies through the FNN were consistently slower than those for the single interface. On average, the FNN times are approximately 15 percent slower than the non-FNN times. Some initial observations of the operating system on the dual processor seem to suggest that the machine is still receiving data as quickly as expected. However, it appears that the machine very quickly reaches a point where it cannot write all of the data to permanent storage quickly enough and therefore becomes “overloaded.” This point appears to be reached much more quickly on the dual processor, yet this cannot be accurately compared with the results from the single processor machine, as they are too different architecturally. Further testing will be done by booting a single processor kernel on the dual to attempt to isolate the differences caused by the more modern architecture from the addition of the second processor.

There are other observations of interest when comparing both dual and single processors with FNN and non-FNN configurations. The following chart (fig. 7) provides a graphic illustration of the grouping between the multiple configurations. With both single and dual processors, the FNN and non-FNN data points exhibit similar characteristics per processor type. For example, FNN and non-FNN configurations on a single processor experience a similar increase in copy times based on the number of nodes copying to the single node. The same behavior can be found in the dual processor tests. Another observation is that single processor machines experience a much greater time increase per number of nodes as opposed to the dual processor configurations.

The most interesting observation that can be made, however, concerns numbers of nodes and advantages of FNN technologies. For example, smaller clusters that are mainly single processor machines would appear to benefit from an FNN

configuration. However, this may be true only up to a certain number of machines, because data-bound processing on larger clusters may try to send so much data that it completely overloads single processor-based FNN configurations. Although this particular data access pattern is slower over an FNN to a dual processor, the dual is much better at receiving data through a single channel. As the chart illustrates, the time increase per number of nodes is much lower on a dual processor. Thus, it would seem that they would indeed scale much better as more nodes try to access according to this pattern. However, the sample size here is much too small to predict how long this would prove valid. It may prove that past a certain point, the dual processor FNN pattern would finally become faster than the dual processor single channel configuration.

For a real world system test, a distributed projection application was run over the FNN. This application reprojects digital data between various map projections. It was developed in FY 1999 and has been modified since then for various aspects of distributed processing testing. It was chosen because it was a perfect candidate application that needed huge amounts of data for processing and performed a common function that was representative of some of the processing necessary for *The National Map*. The distribution method involves a master node assigning groups of scanlines in the output dataset to each of the processing nodes. Groups of scanlines have proved to be a more efficient

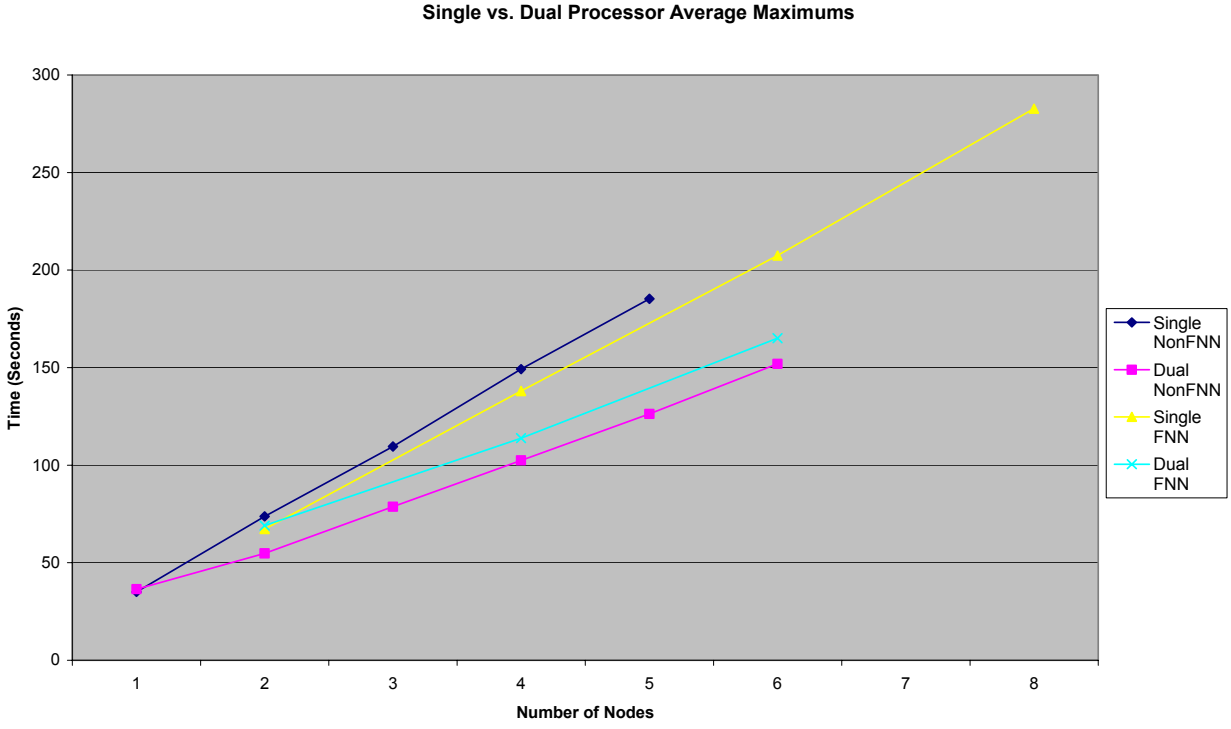

Figure 7. Single Processor versus Dual Processor Averages.

means of distribution than single scanlines, because the processing nodes can read and cache input data more efficiently to generate groups of scanlines than individual ones.

The data access patterns of this application involve the processing nodes reverse projecting the coordinate block to determine what input data will be used to create the output. They then read in scanlines from the input file and cache them in a least recently used (LRU) cache structure. This structure keeps track of which entries have not been accessed recently and replaces the oldest entry with newly read scanlines. The processing nodes then create and send each output scanline back to the master node that writes it to the output file. The master node is normally responsible for file output, because the software libraries that are used to create various file formats are not always written so that different processes can safely use them simultaneously.

The testing consisted of projecting a 2-gigabyte dataset composed of several digital orthophoto quadrangle files that were merged into a larger dataset and converted into GeoTIFF format. This file was then stored on a single file server on the cluster. The processing application ran the master process on the same node as the file server, and each processing node read from here and sent processed scanlines to the master so that it could write them to a local disk. Testing involved taking multiple runs with the same number of processing nodes and averaging the results together. Between each run, the disk caches of each machine were flushed to minimize the chance of any leftover data still being in the operating system cache. The results of these tests are presented in figure 8.

The runtime curves for the FNN and non-FNN configurations with this data

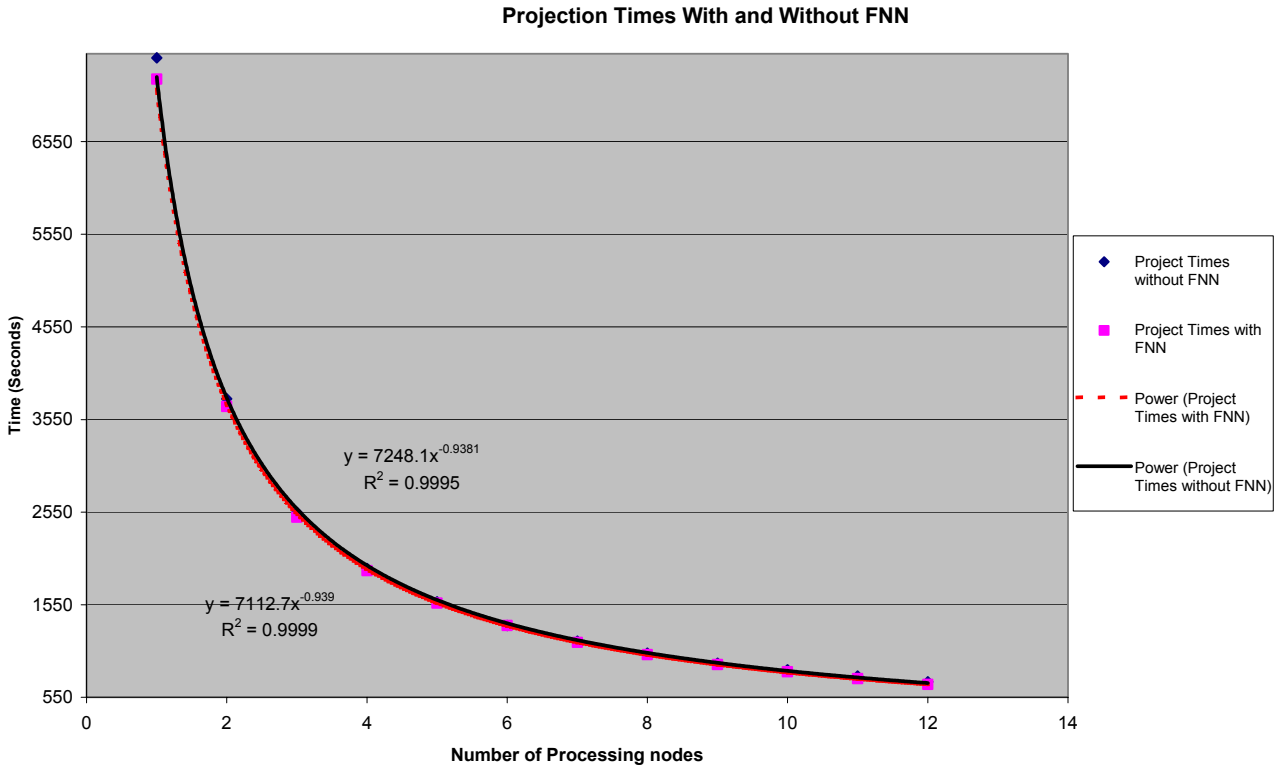

Figure 8. Comparisons of FNN and non-FNN projection runtimes.

access pattern are nearly identical. A regression curve was fit through each series, and the resulting equations and R^2 values are displayed in figure 8 to illustrate how close the runtime curves match each other. On average, the FNN runtimes are only approximately 44 seconds faster than the non-FNN runtimes. However, this average can be misleading. While the FNN time at two nodes is 228 seconds faster than the non-FNN equivalent, the time difference decreases between both configurations as processing nodes are added. At six nodes, both configurations actually have an equivalent runtime. Beyond six nodes, the FNN times again become faster and the time difference between the systems increases. This increase, though, is not a dramatic one.

It is again important to note that in this case, with this data access pattern, the FNN design appears to offer no real benefit over the non-FNN configuration. However, this is a small sample size, and many more nodes are necessary before statistical characterizations can be accurately made. Also, this is not an exhaustive test, and future research into nodes of different architectures should also be done to determine if FNN might exhibit different characteristics in these cases.

Parallel File System Research

The next research concerning large-file processing dealt with using parallel file systems instead of the more traditional single file server model. A parallel file system in this case is one that is scalable and spans multiple machines. This operates much like a redundant array of Inexpensive Disks (RAID), where data storage can be spanned across several hard drives within the same machine. The advantage of parallel file systems is that they provide more disk bandwidth than the traditional method of reading and writing data from a single machine. With a parallel file system, portions of the data are read from and written to various machines across the network. In this scheme, multiple machines read their data from multiple file server nodes instead of trying to connect to a single machine.

This research also studied several different data access patterns over parallel and nonparallel file systems to determine which method provided the greatest amount of network throughput to the test application. Traditional access patterns with data on a single disk do not necessarily work well with a distributed processing architecture. These techniques are generally geared for high bandwidth and low latency situations, such as those found inside a computer where data is transferred across the system bus. The distributed access methods ranged from a traditional read and write from a single node to a distributed read and write across the parallel file system volume.

For testing, the PVFS project was chosen. PVFS provides Linux kernel modifications and user space utilities for interacting with a distributed file system

volume. The kernel modifications allow applications to access the PVFS volume even though they have not been specifically written to use PVFS, such as the standard Unix file utilities. However, PVFS also allows applications to be written that directly interact with the system. The advantage to this method is that non-PVFS specific applications must go through the kernel module that then has to translate their requests to work with PVFS. This translation can add significant overhead, because it requires the kernel to switch from kernel to user mode several times during the process.

PVFS operates by having an overall manager running on a node that controls access to the system. This master node contains file location information across the distributed volume. The individual file server nodes then run input and output (I/O) managers that handle read and write requests to their part of the distributed volume. When a process wishes to access a PVFS volume, it first contacts the manager node. The manager node will then send back the IP address of the machine that holds the data. The requesting process then contacts the I/O manager on the node to handle the read or write request.

In FY 2001, MCMC began experimentation with PVFS across a traditional network. For FY 2002, the work focused on combining PVFS with the FNN technology. To do this, we had to make some minor modifications to the PVFS software. PVFS sends IP addresses in response to requests to locate certain parts of a file. The problem is that, as previously mentioned, FNN machines typically have multiple IP addresses. The address sent back to processes requesting access to PVFS is the IP address that the manager node uses to get to the individual file node. This is not necessarily the same IP address that the process on the requesting node would use to communicate with the file-serving node. The solution to this problem was to modify PVFS to send hostnames back so that the requesting nodes would be able to connect to the proper IP address. Some other fixes were made to a utility in the suite that allows a file to be copied to the distributed volume with a user-defined number of file stripes per node.

Even with these fixes, PVFS exhibited numerous problems on the MCMC cluster. The master node of PVFS would periodically return incorrect sizes for files across the cluster. This caused crashes not only in the application software accessing these files but also in the operating system itself on certain nodes. PVFS was even less stable when running on the dual processor machines of the cluster, where it would generally fail to operate at all. This caused the testing to use only the single processor machines to store the PVFS volume. When PVFS did work, it would also periodically crash during runtime tests. This caused testing to take much longer than originally anticipated.

The initial testing with PVFS was to compare it with a traditional file server configuration. For this testing, the projection application was again used, because it requires large amounts of simultaneous data requests. It was modified to use the library provided with PVFS so that it could directly access the

distributed volume instead of using the slower, kernel-based translation method. Libtiff (libtiff.org), a software library used to read TIFF and GeoTIFF files, also had to be modified to use the PVFS library since the input file was in GeoTIFF format. Otherwise, libtiff would have used the slower kernel translation method to access files on the volume. The PVFS volume was spanned across three file-serving nodes, with each node being a single processor machine. The master PVFS node was also running on a single processor machine. Each of the file-serving nodes was dedicated to file serving and did not take part in the processing. For the NFS case, the file-serving node was separate from the master-processing node. The data access pattern used was the same as previously described, except that the three-node PVFS volume handled the read part. Output scanlines were still sent back to the master node, which would write to local storage. The input file was copied to the volume so that each I/O node contained one-third of the file. Process partitioning was used during these tests. Each test was again run three times and averaged; the results are depicted in figure 9.

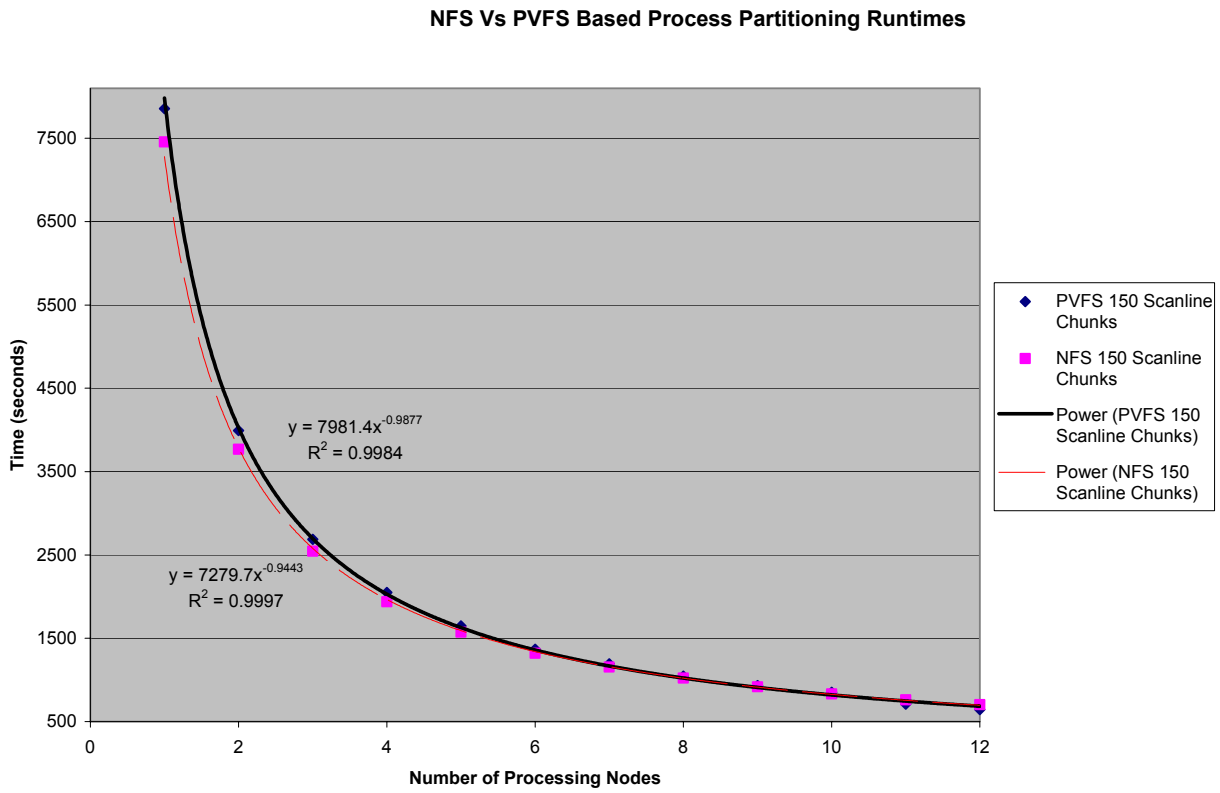

Figure 9. NFS versus PVFS file read.

For this data access pattern with three I/O nodes, PVFS is initially slower than traditional NFS file serving. With small numbers of nodes reading data, PVFS is less efficient owing to the synchronization required and the additional step of contacting the PVFS head node to determine file locations. As the number of processing nodes increases, PVFS begins to overtake NFS file serving in speed. However, the speed increase seen from this configuration is still not significantly faster. Statistically, the power approximation curves can only be extended out a few nodes depending on the number of samples present. Even then, this configuration (three I/O nodes serving data for reads with one-third of the file each) does not appear to provide a significant advantage.

This led to another series of tests that involved not only increasing the number of I/O nodes but also varying the amount of data stripes per node. The first idea originated from some initial tests in turning the entire cluster into one large PVFS volume. In this sense, a node would not only be processing data but would also be serving files to other nodes.

The striping idea originated from a study of the data access patterns used by the test application. The input file is written on a per-scanline basis, with each scanline written one after the other in order. When the processing nodes read data to generate the output scanline, they typically request scanlines all within a certain area. When used with the processing partitioning method, where nodes are assigned a group of scanlines to process, this creates a certain locality in the file accesses for each processing node. But this locality is not always in the same area for each processing node. For example, some nodes may read mainly from the beginning of a file, and others may read from the end of the file.

A possible means to exploit this locality may be to stripe the files across all nodes in the cluster and also to have multiple stripes per node. For example, with a file size of 2 gigabytes and a stripe size of 32 kilobytes, a 16-node PVFS volume would contain four stripes per node. These stripes would be written in a round-robin fashion, and each stripe would then be from a different location in the file going from beginning to end. Keeping stripe sizes smaller also reduces the amount of time any given node is trying to read data from another node. There is then a good chance that various processing nodes would be simultaneously reading data from different nodes around the cluster. This large distributed volume approach could then work to reduce network congestion per node when combined with the FNN technologies.

The first striping test examined the read-only case, where the input file was distributed amongst varying numbers of file-serving and processing nodes with the file stripe size also varied. Output scanlines were still sent to the master node for file output. The read-only case is depicted in figure 10.

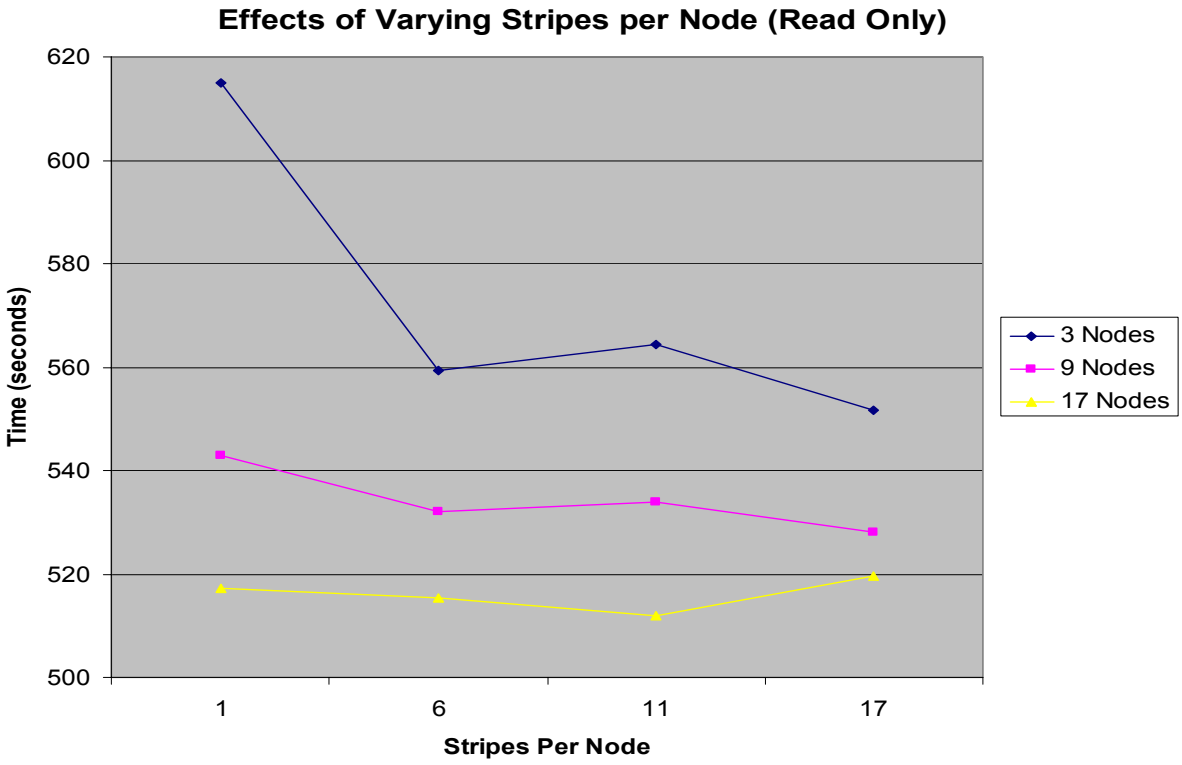

Figure 10. Read-only effects of varying stripes per node.

In the read-only case, varying the stripes per node has a greater impact with smaller numbers of I/O nodes than with each node in the cluster serving as a distributed file server. As can be seen in the case of three nodes, going from a contiguous one-third of the file on the node to six different parts of the file decreased processing times by 56 seconds. With each processing node also serving as an I/O node, the processing time did not change much and actually began to increase as more stripes were added per node. Although a decrease of a minute may not seem like much, it can save huge amounts of time when considered in terms of something like a distributed batch system, where hundreds of large files may need processing. The decreases in runtimes demonstrate not only the potential of PVFS as the number of file-serving nodes increases, but also the potential in using multiple stripes per node to take advantage of locality in the access patterns.

The next case to consider is that of reads and writes being done to a PVFS volume. This was more difficult to implement, because the software could not safely write the output file as a GeoTIFF owing to synchronization issues with the various software libraries. For this, the projection software was modified to output the file in a raw file format. This way, the file could be safely written to the distributed volume without risking data corruption. After these modifications were made, the software was tested by reading and writing by means of the PVFS

volume. Again, differing numbers of file-serving nodes and different stripe sizes were used in these tests. The results of those tests are portrayed in figure 11.

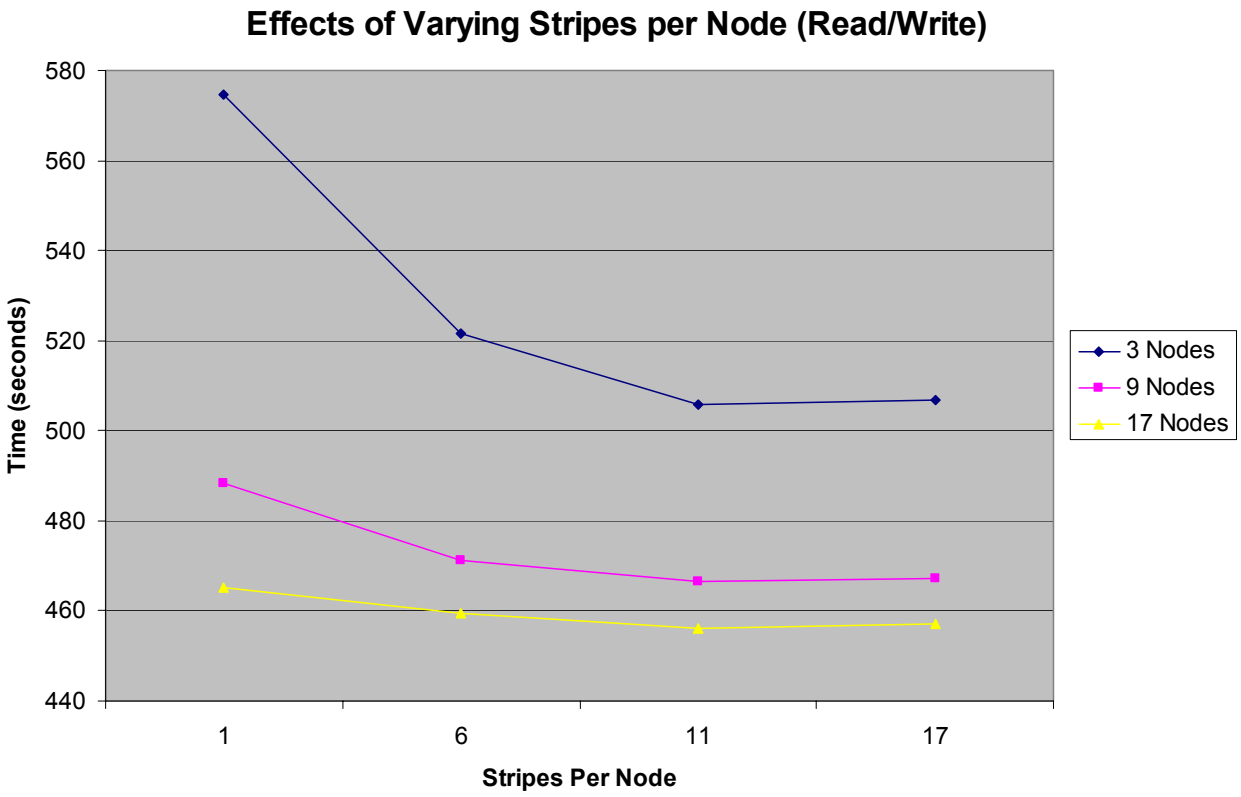

Figure 11. Read/write effects of varying stripes per node by means of PVFS.

The read-and-write case shares some characteristics with the read-only case. For example, three I/O nodes see a dramatic decrease in runtimes when going from a single contiguous file stripe to six noncontiguous file stripes. The curves also can be seen to flatten out as more file stripes are written, to the point where the runtimes actually increase again when going from 11 to 17 stripes per node.

However, the most notable difference is that the read/write case is significantly faster than the read-only case from a distributed volume. The three I/O node and single file stripe case takes 615 seconds in the read-only method, and 574 seconds in the read/write method. Also, the fastest runtime with the read/write method is 56 seconds faster than the read-only case. In fact, 17 processing nodes with 11 stripes per node exhibited the fastest runtimes ever recorded with this particular application.

This speed can be expected, though, because the distributed read-and-write case is a much more efficient access pattern than the read-only case. With read only, each node still sends output scanlines to the master node. This is a significant bottleneck as some of these scanlines can become quite large. The

master node must also store and write these data to a file in addition to assigning work to various processing nodes. With the read/write case, data are sent to the various file nodes across the entire cluster. This reduces the bottleneck problem, as all nodes are not going to a single node. In fact, fully distributing reads as well as writes reduces the problem to only a few nodes accessing the same node at any given time. Because of this, it can be predicted that this type of file volume scheme may well be best suited to distributed data-bound tasks.

References

Crane, M., Steinwand, D., Beckmann, T., Krpan, G., Liu, S., Nichols, E., Haga, J., Maddox, B., Bilderback, C., Feller, M., and Hamer, G., 2001, A parallel-processing approach to computing for the geographic sciences: Applications and systems enhancements: U.S. Geological Survey Open-File Report 01-465.

Steinwand, D.R., 1994, Mapping raster imagery to the Interrupted Goode Homolosine projection: International Journal of Remote Sensing, v. 15, no. 17, p. 3463 – 3471.

Swets, D.L., Reed, B.C., Rowland, J.R., and Marko, S.E., 1999, A weighted least-squares approach to temporal smoothing of NDVI: ASPRS Annual Conference, From Image to Information, Portland, Oregon, May 17-21, 1999, Proceedings, American Society for Photogrammetry and Remote Sensing, Bethesda, Md., CD-ROM, one disc.

Important Web Sites

The Aggregate: FNN, Flat Neighborhood Networks. University of Kentucky. <http://aggregate.org/FNN/>.

The Parallel Virtual File System Project. Clemson University. <http://parlweb.parl.clemson.edu/pvfs/>.

PVFS: Description. Clemson University. <http://parlweb.parl.clemson.edu/pvfs/desc.html>.

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

LibTIFF – The Tagged Image File Format <http://www.libtiff.org.>.