CORNERSTONE
MINNESOTA STATE UNIVERSITY MANKATO

Journal of Undergraduate Research at Minnesota State University, Mankato

Volume 8 Article 12

2008

Improved Storm Data Processing through Parallel Computing Approaches

Shauna Smith
Minnesota State University, Mankato

Recommended Citation

Smith, Shauna (2008) "Improved Storm Data Processing through Parallel Computing Approaches," *Journal of Undergraduate Research at Minnesota State University, Mankato*: Vol. 8, Article 12.
DOI: https://doi.org/10.56816/2378-6949.1082
Available at: https://cornerstone.lib.mnsu.edu/jur/vol8/iss1/12

Improved Storm Data Processing using Concurrent Computing Approaches

Shauna Smith (Department of Computer Science)
Rebecca Bates, Faculty Mentor (Department of Computer Science)
Deborah Nykanen, Faculty Mentor (Department of Mechanical and Civil Engineering)

Abstract

A previous research study conducted at Michigan Technological University by Dr. Deborah Nykanen and her colleague Dr. Daniel Harris analyzed storm data in order to develop algorithms that will allow coarse resolution rainfall forecasted by weather models to be optimally used in high resolution hydrology models with the goal of improving stream flow predictions and early detection algorithms that can be used to warn communities about potential flash floods. This research was performed by analyzing a series of independent radar images derived from Weather Surveillance Radar-1988 Doppler (WSR-88D) data obtained from Dr. James A. Smith at Princeton University using a series of computer programs written by the original researcher and her colleagues. The program was run using a sequential algorithm that can take up to 17 hours to execute. Because of the structure of the problem, there was an opportunity for applying concurrent computing techniques to the program code. In order to speed up the program execution time, several different concurrent computing approaches have been applied to the code. Speedup analysis has been conducted for each different concurrent approach improving the code execution time by up to a factor of 93. The analysis results show how different concurrent approaches affect the speedup of code. The faster code will aid in analyzing future storm data, allowing more data to be analyzed in a shorter amount of time, and will eventually be used in improving lead time on high resolution stream flow predictions and flash flood warnings. The speedup provided by the concurrent computing approaches has been verified on previously analyzed data.

Introduction

Research conducted at Michigan Technological University by Dr. Deborah Nykanen and her colleague Dr. Daniel Harris analyzed storm data in order to develop algorithms that will allow coarse resolution rainfall forecasted by weather models to be optimally used in high resolution hydrology models. The goal of that research was to improve stream flow predictions and early detection algorithms that can be used to warn communities about potential flash floods. The research was performed by analyzing a series of independent radar images derived from Weather Surveillance Radar-1988 Doppler (WSR-88D) data obtained from Dr. James A. Smith at Princeton University using a series of computer programs written by the original researcher and her colleagues (Harris, 1998; Nykanen and Harris, 2003). The problem with these programs was that they used sequential algorithms that took up to 17 hours to execute. The structure of the problem, where many independent calculations are made, created an opportunity for applying concurrent programming techniques to the sequential code. Because of this opportunity, the goal of this research project is to make the code faster by applying concurrent programming techniques. In the future, the new code can be used to analyze future storms more quickly and efficiently.

In this paper, background of concurrent computing will be presented followed by a description of the rainfall analysis algorithm and storm data used in this project. The methodology used to apply concurrent computing approaches and the results of each method is also presented. It concludes with a summary and future work for the project.

Concurrent Computing Background

"Two events are said to be concurrent if they occur within the same time interval. Two or more tasks executing over the same time interval are said to execute concurrently" (Hughes & Hughes, 2004, p. 2). The concept of concurrency can be applied to computing when there is the possibility that multiple things can or need to be happening at the same time. Concurrent computing is also often known as parallel computing.

Parallel vs. Distributed Programming

There are two different ways of implementing concurrency within a computer program. One way is to implement parallel programming. Parallel programming is the assignment of work to two or more processors within a single physical or virtual computer as processes or threads (Hughes & Hughes, 2004, p. 3). Another way of implementing concurrency is to use distributed programming. Distributed programming is the assignment of work to two or more processors that might not exist on the same computer as processes (Hughes & Hughes, 2004, p. 3). These approaches can also be implemented as a hybrid of parallel and distributed programming. The hybrid can occur if there are multiple processors on a single physical computer and that computer lies within an accessible network of many other distributed computers. To determine whether to implement a parallel, distributed, or hybrid programming approach, the problem that is going to be solved using some type of concurrent computing approach needs to be analyzed and fit into one of these different designs.

Process vs. Thread

There are two basic ways of splitting up work that a computer has the capability to schedule and manage. These two ways are either to create processes or to create threads. "A process is a unit of work that is created and managed by the operating system" (Hughes & Hughes, 2004, p. 37). A thread, on the other hand, is a unit of work that is created within a process and that shares resources with other threads.

The main difference between a process and a thread is that a process has its own address space and a thread does not (Hughes & Hughes, 2004, p. 102). This difference causes both advantages and disadvantages for implementing threads rather than processes. The advantages and disadvantages are outlined in Table 1.

Table 1: Implementing Threads vs. Processes (Hughes & Hughes, 2004, p. 108)

Advantages of Implementing Threads	Disadvantages of Implementing Threads
Fewer system resources needed during context switching	Requires synchronization for concurrent read/write access to memory
Increased throughput of an application	Can easily pollute address space of its process
No special mechanism required for communication between tasks	Only exist within a single process and therefore not reusable
Simplification of program structure	

Good Computing Performance

When designing parallel algorithms, it is important to consider the following question: what is good computing performance? The intuitive notion is that the program runs fast, uses a small number of processors, and uses a small amount of memory. However, a more expert notion of good computing performance means that the program maintains a balanced load, has low overhead, and implements a scalable algorithm.

The concepts of a balanced load, low overhead and scalable algorithms are very important to understand when dealing with the design and creation of concurrent computer algorithms. If work has been distributed evenly across multiple processors and all processors are staying busy for the same amount of time, then a parallel algorithm is considered to have a balanced load. The communication between separate processing entities in a concurrent algorithm can take extra time that a sequential algorithm would not require. The extra communication time is typically known as overhead. And lastly, a scalable algorithm is one that can take advantage of more processors if more were to become available. The goal of implementing any type of concurrent computing is to keep all available processors busy doing useful work for as long as possible (Haglin et al., 2008). Thus, a good parallel algorithm will result in a balanced load and minimal overhead and will be scalable if the number of available processors increases.

Where Concurrent Computing is used

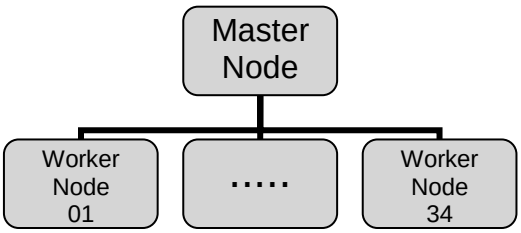
Parallel computing can easily be used on programs that need more work done over a given time frame, need a simplified solution, need to be faster, can make use of specially designated processors, or address large problems with time-consuming sequential solutions (Hughes & Hughes, 2004, p. 3-6). One example of an area where concurrent computing is being used is in data visualization. One specific data visualization program that uses concurrent algorithms is a program that uses structural information obtained through X-ray crystallography and electron microscopy to determine spherical-virus structures (Martin & Marinescu, 1998). There is a need for concurrency in this type of application because of the large amounts of data that need to be processed in order to create a visualization.

Weather analysis and weather prediction are very important procedures that benefit from the use of concurrent computing. "The use of numerical weather prediction models to provide input of spatial rainfall patterns for distributed hydrologic models has gained increasing popularity over the past decade to study the hydrologic response of alpine catchments" (Nykanen, 2008). These models are very dependent on the quality and resolution of the input precipitation data. The data can often be quite large, especially if the storm lasts for a long time or covers a large area. Because of the large amount of data and the computation intensive algorithms used for weather analysis, concurrent computing is an important tool for weather analysis and prediction.

Rainfall Analysis Algorithm

The rainfall analysis programs used in this work had previously been implemented in a sequential form on computers at the Minnesota Supercomputing Institute (*Minnesota Supercomputing Institute*, 2004). The rainfall analysis algorithm could benefit from a hybrid approach of parallel and distributed programming because of its structure, where a time-sequence of data frames is analyzed independently. The rows and columns of the data matrices derived from the data frames can also be processed independently. This allows the data to be split in two different ways, at the frame level and at the row or column level of the matrices. The

structure of the High Performance Computer (HPC) at Minnesota State University, Mankato allows for this type of hybrid implementation of both parallel and distributed programming (*High-Performance Computing at MSU*, 2007). Figure 1 features a diagram and description of the HPC. One of the key features of the HPC is that there is a master node and 34 worker nodes that are assigned processes by the master.



- 1 master node & 34 worker nodes
- All jobs are sent from master to worker nodes
- 2 dual-core AMD Opteron chips = 4 logical processors/node
- 8GB of RAM per node for a total of 280 GB
- 7.4 TB of disk
- Linux OS
- Scratch disk space available on each node

Figure 1: High Performance Computer (HPC)

A flowchart of the sequential rainfall analysis algorithm is shown in Figure 2. The details of the algorithm can be found in Nykanen and Harris (2003). The flowchart below gives a short description of the key features of the algorithm and shows where the algorithm loops in order to process each of the independent data frames. These frames could be parallelized by sending individual frames to different processors on the HPC. The gray box, structure function analysis, includes array calculations that can be parallelized. Specific parallel implementations will be described later in this work.

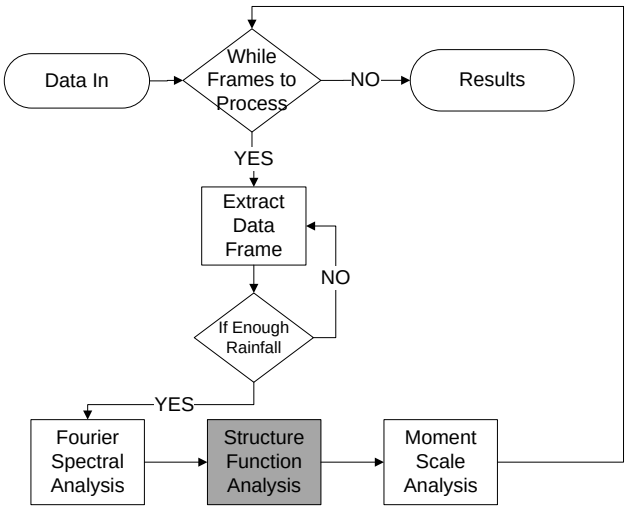


Figure 2: Rainfall Analysis Sequential Algorithm

Fort Collins, Colorado Storm

The rainfall data that was used as test data in this project was taken from a storm that occurred near Fort Collins, Colorado on the eastern slope of the front range of the Rocky Mountains on July 28 - 29 of 1997 (Nykanen, 2008). It produced record flash flooding causing five deaths and $200 million in damage. The data was in the form of a series of independent radar images derived from Weather Surveillance Radar-1988 Doppler (WSR-88D) data and was obtained

from Dr. James A. Smith at Princeton University (Nykanen, 2008). The Fort Collins, Colorado storm data consists of a sequence of 187 rainfall fields derived from WSR-88D radar at 1-km horizontal resolution and 6 minute temporal resolution. The extent of the rainfall field in each frame is 450 x 450 square kilometers. Figure 3 represents one of the 187 data frames in the sequence of data covering a total of 18 hours and 42 minutes.

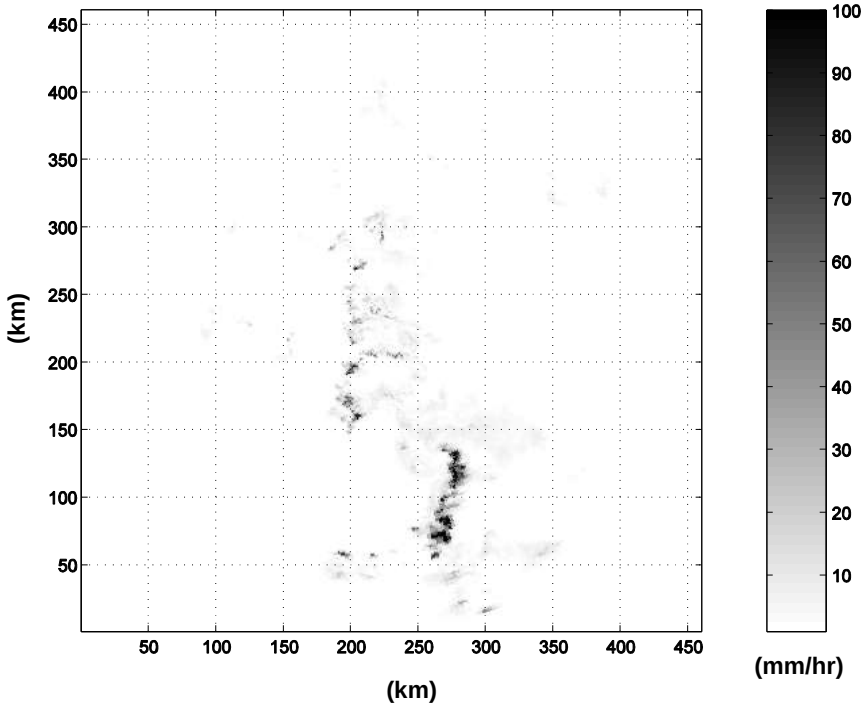


Figure 3: Fort Collins, Colorado High Resolution Rainfall Image

Methodology

In order to speed up the program execution time, several different concurrent computing approaches were applied to the rainfall analysis sequential algorithm. These approaches were applied and tested separately in order to determine which approach was best.

One improvement made in all concurrent programming approaches was the use of "scratch space" on each node. Scratch space is a term to represent local hard disk on a node. This disk space is available for programs to read from or write to. The sequential algorithm only used disk space connected to the master node. When accessing disk space connected to the master node, there is some overhead added because of this communication. During the sequential algorithm, there is a negligible amount of this overhead when comparing it to the larger amount of overhead that would be created if the concurrent approaches listed below all tried to access disk on the master node at the same time. This large amount of overhead is created because of interfering traffic and limited bandwidth. An advantage to using scratch space is that it is quicker to access local disk space than disk space residing on the master node, especially for large distributed implementations.

The four concurrent programming approaches implemented are outlined below.

1) Separate distributed processes

The first concurrent programming approach applied was distributed processing. The Fort Collins, Colorado storm had 187 independent radar images that needed to be analyzed. These independent radar images are considered to be single independent pieces of data known as frames. Each separate distributed process was assigned a single data frame to analyze. All processes executed the same calculations but on different data. A single process required the exclusive use of a processor. After all frames were completed, another quick process was run in order to concatenate all output files. Figure 4 is a graphical representation of the distributed processes implementation. The flowchart only represents the sequence of events for a single data frame. For the Fort Collins, Colorado storm, there would be 187 processes running this sequence of events, each on a separate processor running in parallel.

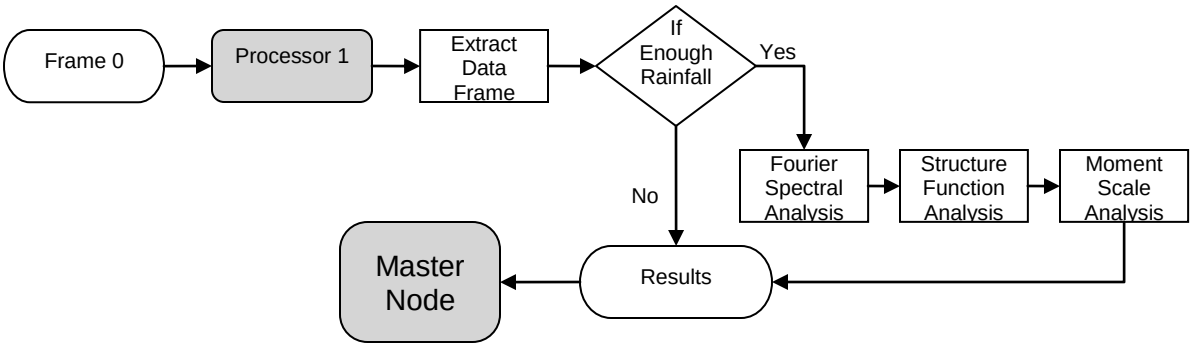


Figure 4: Rainfall Analysis Algorithm using Distributed Programming

2) Threading

Within the sequential rainfall analysis algorithm, the two dimensional autocorrelation function was threaded. The two dimensional autocorrelation function is contained in the structure function analysis portion of the rainfall analysis algorithm (gray box in Figure 2). This portion of the code was a bottleneck, taking up a significant amount of processing time per frame. For that reason, it was chosen to be the threaded portion of the code. In this implementation, a single process was created that was in charge of processing all frames of data. That process requested the exclusive use of an entire node (4 processors). Within that single process, four threads were created during the two dimensional autocorrelation function and each thread was then allowed the exclusive use of a processor. An individual thread was in charge of working on one fourth of the data matrix that was being accessed within the two dimensional autocorrelation function. Figure 5 is a graphical representation of how the matrix data can be split and processed in parallel using threading. The gray boxes represent the function that was threaded.

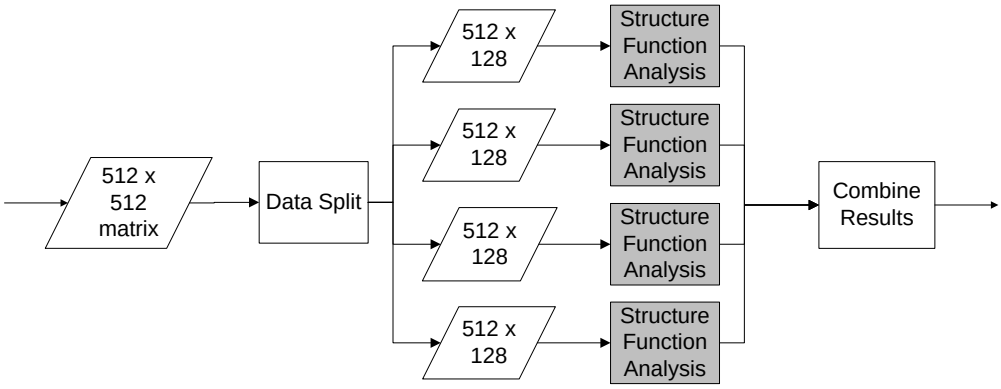


Figure 5: Rainfall Analysis Algorithm using Parallel Programming

3) MPI communication between processes

Using a similar approach to the threaded method, an approach using MPI communication was implemented. "MPI (Message Passing Interface) is a standard of communication used in implementing programs that require parallelism" (Hughes & Hughes, 2004, p. 330). Just like the threaded implementation, a single process was created that was in charge of processing all frames of data. That process requested the exclusive use of an entire node (4 processors). The difference between the threaded and MPI implementations was that the two dimensional autocorrelation function was broken into four processes instead of threads. Each of these four processes was in charge of working on one fourth of the data matrix and was allowed the exclusive use of a processor on the node, just like the threaded implementation. Figure 5 also graphically represents how the matrix data was split and processed in parallel using MPI communication. The difference between implementing processes rather than threads is that each process has its own address space while threads are all sharing one single address space. This difference causes the advantages and disadvantages of implementing threads rather than processes that were mentioned in Table 1.

4) Hybrid of distributed processing and threading

The final approach was designed as a combination of (1) using separate distributed processing and (2) threading. The Fort Collins, Colorado storm was again broken up into 187 different processes. Each separate distributed process was assigned a single data frame to analyze. A single process required the exclusive use of a node (4 processors) in order to implement threading. Again, the two dimensional autocorrelation function was threaded within each of the separate distributed processes. Within each single distributed process, four threads were created and each thread was then allowed the exclusive use of a processor. An individual thread was in charge of working on one fourth of the data matrix during the two dimensional autocorrelation function. After all separate distributed processes were complete, another quick process was run in order to concatenate all output files. Figure 6 is a graphical representation of the hybrid implementation. The flowchart through the hybrid implementation is for a single data frame. For the Fort Collins, Colorado storm, there would be 187 processes running this sequence of events, each on a separate node running in parallel.

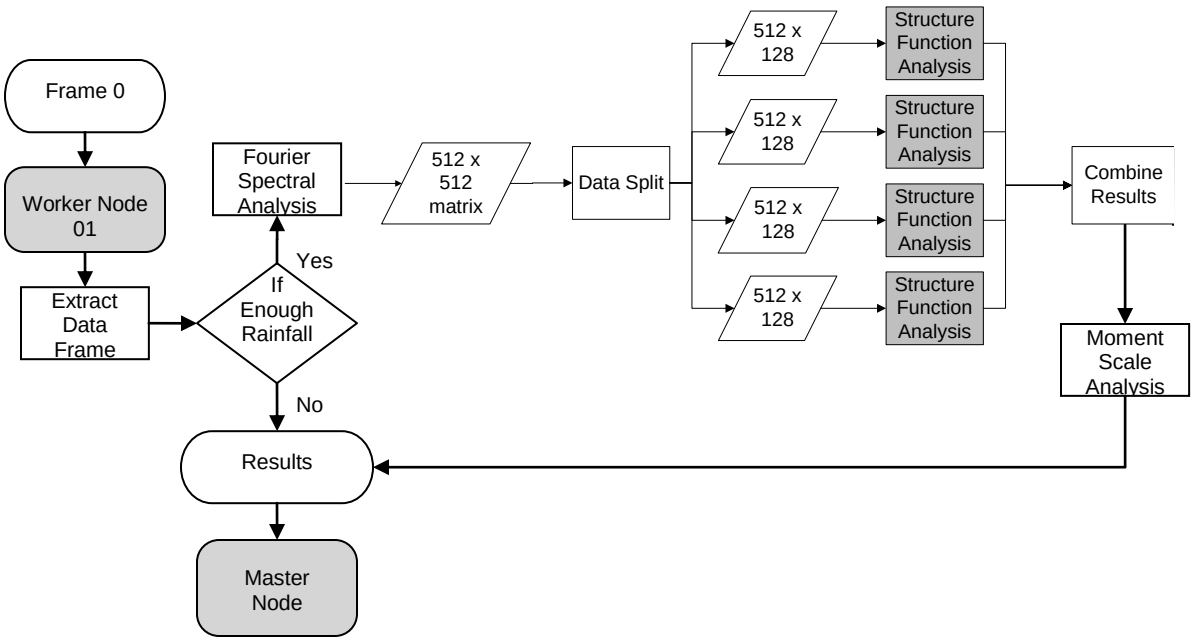


Figure 6: Rainfall Analysis Algorithm using Distributed and Parallel Programming

Performance Assessment

After completing and testing the implementations of the different concurrent programming approaches on the HPC, performance assessment based on process time was conducted on each approach. The evaluated metrics were the factor of speedup, the amount of overhead (expressed in hh:mm:ss), and the percentage of efficiency. The following equations were used to calculate each metric:

- Speedup $S = T_S / T(p)$
- Overhead $T_0 = p * T(p) - T_S$
- Efficiency $E = S / p$

T_S : baseline time

T(p) : time using p processors

p : number of processors used

The ideal factor of speedup for an algorithm is equal to the number of processors used in the parallel implementation. An ideal parallel algorithm would also have no overhead and would be 100 percent efficient. The number of processors used in each implementation was needed to calculate the overhead. These numbers are given in Table 2.

Table 2: Number of Processors Used in Concurrent Implementations

	Number Of Processors
Separate Distributed Processes	94
Threading	4
MPI Communication	4
Distributed Processing and Threading	128

Timing Results

The timing results for the sequential algorithm and all concurrent approaches implemented on the HPC are listed in Table 3. The times were recorded as total processing time until completion and average processing time per data frame.

Table 3: HPC Timing Results

	Total Process Time	Average Time Per Frame
Serial Implementation (baseline)	T_S = 17:16:38	00:05:33
Separate Distributed Processes	T(p) = 00:11:07	00:05:12
Threading	T(p) = 07:26:52	00:02:23
MPI Communication	T(p) = 05:35:16	00:01:48
Distributed Processing and Threading	T(p) = 00:16:24	00:02:14

(All results given in format hh:mm:ss)

Performance Assessment Results

The performance assessment results for each concurrent programming implementation are listed in Table 4. The performance assessment metrics include factor of speedup, amount of overhead (expressed in hh:mm:ss), and percentage of efficiency.

Table 4: Performance Assessment Results

	Speedup S	Overhead T_0	Efficiency E
Separate Distributed Processes	93.25	00:08:20	99.20%
Threading	2.32	12:30:50	58.00%
MPI Communication	3.09	05:04:26	77.30%
Distributed Processing and Threading	63.21	17:42:34	49.38%

Analysis

The concurrent algorithm implementations were analyzed by comparing the different performance assessment results and weighing the benefits and disadvantages of each implementation based upon these results. The notion of scalability was also analyzed based upon the HPC timing results and the performance assessment results.

Speedup Analysis

Both implementations using separate distributed processes across multiple processors provided the two highest factors of speedup. The ability to take advantage of multiple available processors decreased the total execution time until completion. A shorter execution time results in a higher factor of speedup.

A rule of thumb when analyzing speedup is that the ideal speedup should be equal to the number of processors used in the parallel implementation. In the separate distributed processes implementation, the number of processors used was 94 and the factor of speedup was 93.25. These numbers are very close, implying that the implementation resulted in nearly ideal speedup. In the threaded and MPI communication implementations, 4 processors were used. Neither of these two approaches came as close to obtaining ideal speedup. In the distributed processing and

threading approach, 128 processors were used but the factor of speedup was only 63.21. The factor of speedup in this approach was only one half of the amount of processors used.

The threaded, MPI communication, and hybrid implementations might not have succeeded in obtaining ideal speedup, but they were able to cut down on the average processing time per data frame. The use of threading or MPI communication provided shorter execution time during the bottleneck portion of the code, which resulted in shorter average processing time per data frame. This decrease in average time per data frame is very important because of how it can be implemented in hybrid approaches of distributed and parallel processing, especially for implementations using smaller storm data or on a system with fewer processors.

Overhead Analysis

The equation used to calculate overhead in this project requires the term "overhead" to be defined slightly differently than normal. Traditionally, overhead is defined as the extra time it takes to communicate between separate processing entities in order to utilize parallel programming techniques. The equation used to calculate overhead ($T_0 = p * T(p) - T_S$) works well to calculate the extra communication time for algorithms that have a well balanced work load. The problem with the parallel rainfall analysis algorithms used in this project was that they did not have the quality of having a well balanced work load. This problem caused the overhead calculation to also include processor idle time, not just extra communication time. Because of the unbalanced processor load, some of the processors would stand idle while others were busy working.

In terms of the overhead calculated, the separate distributed processes implementation provided the least amount of overhead while the distributed processing and threading implementation had the highest amount of overhead. When comparing these results, it is important to keep in mind whether or not a high amount of overhead is worth a large factor of speedup and faster execution time. A final decision between implementations would require an analysis of the tradeoffs between speedup, overhead, and efficiency for a given context.

Efficiency Analysis

The use of separate distributed processes had a very impressive efficiency of 99.20%, which was the most efficient approach implemented. However, the hybrid of distributed processing and threading provided the worst efficiency of 49.38%. The threading, MPI communication, and hybrid approaches all suffered in efficiency assessment because of having an unbalanced work load. In all of these implementations, some processors would sit idle while only one of the processors was doing any work.

Scalability Analysis

All performance assessment results depend on the number of available nodes and number of data frames that exist within a particular storm. These dependencies have a big impact on the scalability of each of the concurrent programming approaches, especially the approaches involving distributed programming.

For example, the Fort Collins, Colorado storm could have completed in approximately 5 minutes and 12 seconds (average time per data frame) using the separate distributed processes algorithm on a system with at least 187 available processors. This way, each separate distributed process would get its own processor instead of each processor being in charge of two processes like the implementation on the HPC. If the HPC had only 13 more nodes, the Fort Collins, Colorado

storm could have completed in that approximated time. On the other hand, if the Fort Collins, Colorado storm had 136 or less data frames, the HPC would be able to provide the approximated execution time of 5 minutes and 12 seconds because there would be enough available processors to assign one single distributed process to a single processor.

Conclusion

After analyzing the performance assessment results, the best concurrent programming approach implemented on the HPC using the Fort Collins, Colorado storm data was separate distributed processing. This approach had the fastest execution time, the highest factor of speedup, was the most efficient, and had the least amount of overhead.

However, because all assessment results depend on the number of available nodes and number of data frames, the combination of distributed processing and threading could outperform the separate distributed processing implementation on a larger node system or a smaller storm. Threading was able to cut the average time per data frame in half and distributed processing provides the ability to utilize multiple processors across many nodes. Both of these techniques help obtain the result of a faster execution time and higher factor of speedup.

For example, if the Fort Collins, Colorado storm only had 34 frames of data, it would be able to complete execution in approximately 2 minutes and 14 seconds on the HPC if all 34 worker nodes were available. Similarly, the Fort Collins, Colorado storm would also complete execution in only approximately 2 minutes and 14 seconds on a large system that consisted of 187 nodes.

The dependency on number of available nodes and number of data frames has a big impact on the scalability of each concurrent algorithm. Therefore, these dependencies can help or hinder the use of different implementations, especially when using a distributed programming approach.

Since the goal of this research project was to create a fast algorithm that would contribute to the ultimate goal of warning communities about flash floods, speedup is the most important performance assessment metric. With this goal in mind, the separate distributed processes implementation was the best solution on the HPC but there is still the possibility that the distributed processing and threading approach could be even faster in warning communities on a larger node system.

Future Work

Since the rainfall analysis algorithm will eventually be used in improving stream flow predictions and early detection algorithms that can be used to warn communities about potential flash floods, it is paramount that concurrent computing implementations do not alter the accuracy and correctness of the rainfall analysis. In order to ensure that these parallel changes are not affecting the integrity of the output, the code needs to be tested multiple times using the Fort Collins, Colorado storm data by running it on various machines using different concurrent computing approaches and then comparing the output to the serial implementation results. Additional tests using storm data from other case studies are also needed.

The conclusions of this project lead directly into future work related to the scalability of the parallel implementations. There are two ways to analyze the scalability in terms of distributed processing:

1) Implement rainfall analysis programs on a larger node system

2) Test using different sized storms (different number of data frames)

In either case, the scalability of the algorithms with respect to distributed programming can be analyzed. More research should also be conducted on the scalability of the threaded and MPI communication approaches by increasing the number of available processors to more than 4. This would allow more than 4 processes or threads to be created within the two dimensional autocorrelation function, possibly further decreasing the average processing time per data frame.

References

Frye, J. (2003). Parallel Optimization of a NeoCortical Simulation Program. (Thesis, University of Nevada Reno, 2003).

Haglin, D. J., Mayes, K. R., Manning, A. M., Feo, J., Gurd, J. R., Elliot, M., & Keane, J. A. (2008). Factors affecting the performance of parallel mining of minimal unique itemsets on diverse architectures. *Concurrency and Computation: Practice and Experience*, submitted for publication.

Harris, D. (1998). Multiscaling properties of rainfall: Methods and interpretation. (Ph.D. dissertation, University of Auckland, pp. 185).

High-Performance Computing at MSU. (2007). Retrieved May 18, 2008, from http://www.mnsu.edu/hpc/.

Hughes, C., Hughes, T. (2004). *Parallel and Distributed Programming using C++*. Boston: Addison Wesley.

Martin, I. M., Marinescu, D. C. (1998). Concurrent Computation and Data Visualization for Spherical-Virus Structure Determination. *IEEE Computational Science & Engineering*, Vol. 5, No. 04, pp. 40-52, Oct-Dec 1998.

Minnesota Supercomputing Institute. (2004). Retrieved May 18, 2008, from http://www.msi.umn.edu/hardware/regatta/index.html.

Nykanen, D. K. (2008). Linkages between Orographic Forcing and the Scaling Properties of Convective Rainfall in Mountainous Regions. *Journal of Hydrometeorology*, DOI: 10.1175/2007JHM839.1, 327-347, June 2008.

Nykanen, D. K., Harris, D. (2003). Orographic influences on the multiscale statistical properties of precipitation. *Journal of Geophysical Research*, Vol. 108, No. D8, 2003.

Nykanen, D. K., Harris, D. (2003). Rainfall Analysis Algorithms [computer software]. Michigan: Michigan Technological University, 2003.

Personal Biography

Shauna Smith is a senior majoring in Computer Science with a minor in Mathematics at Minnesota State University, Mankato. During her enrollment at MSU, Mankato, she has been involved with the IMPACT Concert Company, the Minnesota State University Ski and Snowboard club, intramural softball teams, and the College of Science, Engineering, and Technology Student Advisory Board. She has also been employed with the Computer Science and Information Systems and Technology departments as both a lab instructor and tutor for various undergraduate level classes since she was a sophomore. During the fall 2008 semester, she was employed by Walter's Publishing Company in Waseca, MN to develop graphical user interface tests for the EZBook yearbook software. She has also had two previous internships at IBM in Rochester, MN. The first internship included IBM i and web-based programming for manufacturing software support. During the second internship, she was part of a highly selective

and innovative Extreme Blue Speed Team that developed new server technologies and tools to improve the administration of VoIP products. After graduation in December, she hopes to get an innovative and challenging job in the area of software development.

Faculty Advisors' Biographies

Dr. Rebecca Bates

Dr. Rebecca Bates is a professor in the Department of Computer Science. Her PhD in Electrical Engineering is from the University of Washington where she worked to develop computer modeling of pronunciation to improve automatic speech recognition. She has a degree in theological studies from Harvard Divinity School, an M.S. in Electrical Engineering from Boston University and a B.S. in Biomedical Engineering from Boston University. Current research projects include working on automatic speech recognition in noisy environments, analyzing recorded meetings for style and for areas of importance, and analyzing prosody in adolescents with Williams Syndrome.

Dr. Deborah Nykanen

Dr. Deborah Nykanen is a professor in the Department of Mechanical & Civil Engineering. She received her BSCE, MS and PhD from the University of Minnesota. Dr. Nykanen taught for several years at Michigan Technological University before joining MSU. Her research and teaching interests include hydrology, hydrometeorology, and remote sensing applications. Her research efforts focus on scale issues in hydrometeorology and understanding the role that rainfall and soil moisture play across all scales in modulating land-atmosphere interactions. The overall goal of her research is to improve predictions of the water and energy cycles across all scales. Her research involves numerical weather and climate prediction, coupled land-atmosphere modeling, quantifying scale-invariant properties of rainfall and soil moisture, and development of up scaling and downscaling methods for hydrologic variables.