Testing a “Live” Updating High Count Point Application on a Distributed System

Master’s Project

Department of Computer Science

Montclair State University

Adam Schwartz

Advisor: Dr. Stefan A Robila

May 2017

Table of Contents

List of Figures

List of Tables

Abstract

Processing large amounts of data that can bring systems to their knees is an expanding problem for the direction of computing. This project uses large data sets of animal research location data as a way to test the speed of processing for massive data influxes of real world (or real world simulation) data. Real world information is the leading term in where data stores are skyrocketing. Such simulations are in a class of algorithms that are readily parallelizable. It is this type of parallel and distributed programming that is being tested for performance on a modern system.

Acknowledgements

Prof. Stefan Robila for his support on this project.

Prof. David Bostain for making it unambiguously clear what something looks like when you've done it right and for so much else.

1. Introduction

The objective of this project was to test a distributed system via vector calculations. The data chosen was GPS tracking data for animals from MoveBank.org, a project of the Max Planck Institute[1]. The data was formatted as comma separated values and contained enough specificity to determine a unique animal and time instant data point for the location.

In order to keep up with Moore's Law hardware design has migrated towards an increasing number of processing units (called cores) on the same chip. Programs that divide their labors among several of these units are called parallel programs. Because of the size of the datasets (hundreds of thousands or records each) and the fact the processing on them can be readily parallelizable testing this software for it's performance when parallelized makes sense as an attempt to keep pace with modern hardware. Testing on the copou cluster is especially well suited for this because it has enough nodes (linked computers with numerous cores) to test this on a large scale. In time, personal computers at more accessible prices will approach a similar number of cores meaning a likely analogical problem from a programming standpoint.

Despite the data used in this experiment being prerecorded, the test is referred to as "live" (in quotes) because it simulates the type of processing that goes on were there live data to constantly update the system with. The conditions that it simulates are: (1) the parallel read of data at a rate which is presumed to be the capacity of the system not necessarily the rate data would be added to memory had this been a live system and; (2) the processing of each such input into something meaningful before moving on to the next data point. The effective speed of the program at reading and processing each data point is presumed to be approximately the capacity of the hardware to deal with an altered version of this program, which was updated with live data. So despite not having live and constantly updating data this project implicitly tests the ability of the hardware to take in that sort of information and process it at a given rate.

While the computer cluster used has up to 64 compute nodes, the primarily focus was on running 1,2,4,8, or 16 processes during the tests, measuring the complete time for the program to execute, but tentative tests were also done on numbers of processes as high as 512 in one case which is about the upper limit of cores across the whole cluster. The reason for not going higher was that it was found that beyond 16 processes the system started to have undefined behavior, and beyond 64 processes this became even more erratic to the point that even the countermeasure that had been implemented in the software to permit accurate testing on parallel runs beyond 16 was not enough to makes the program's output seem trustworthy.

---

[1] *Wikelski, M., and Kays, R. 2017, Movebank: archive, analysis and sharing of animal movement data. Hosted by the Max Planck Institute for Ornithology. www.movebank.org, accessed on May 5th, 2017.*

2. Problem Background

The purpose of this project is to design, develop and implement multidimensional vector calculation software in a distributed system, with a focus on two-vector manipulation. This will involve an investigation into how to allocate the data amongst the system's cores and to measure the timing and performance of these decisions. The project will result in a computer application designed for these calculations that can be adjusted to show that it runs reliably for a certain period time for a up to a certain size dataset, that is divided among a certain number of cores on a test system, and with a certain number of iterations (updates) within a certain time. Special focus in the metrics will be given to testing the data against a fixed time frame to see what specifics are needed for the program to run a certain amount of data during that period.

a. Data Description

All the data for this project was obtained from MoveBank.org. MoveBank is a free online repository for researchers to host their animal tracking data by the Max Planck Institute for Ornithology.

All data is formatted as a comma separated values (CSV) file. Each instance of the data is separated by a new line character and each line is broken into attributes and demarcated by commas indicating their separate values. The first line of every file contains attribute names instead of values.

The original collection of the data was made by micro-GPS trackers to study migration patterns and other behaviors.

In this project the data was used to produce a real world simulation. Although the point of the project is for vector calculation testing on the copou hardware if the data processed were used it could be extended to visualize a "region" that denotes the ability of all the animals being tracked in a given frame to change location in a given time or short term range.

The first data set was information on pigeons[2]. Specifically it was called "Data from: Homing pigeons only navigate in air with intact environmental odours--a test of the olfactory activation hypothesis with GPS data loggers." The use of this data consisted of taking the coordinates from the data point and discarding the rest. However, the data also included a time stamp and numerous other meta data like the species being observed (Columbia livia), the name of the study and other information extraneous to the experiment. Here is a sample line from the pigeons file (wraps). `2010-07-02 05:17:01.000,10.5764505,43.2185161,,22799644,true,"#2246374","1306","gps","Columba livia","Homing Pigeons`

[2] Gagliardo, A., Ioalè, P., Filannico, C., Wikelski, M., 2012, Data from: Homing pigeons only navigate in air with intact environmental odours--a test of the olfactory activation hypothesis with GPS data loggers: Movebank Data Repository. doi:10.5441/001/1.q8b02dc5/1 Downloaded from Movebank.org Retreived on March 2017.

```
Olfaction",628038.6347671859,4786287.466355227,"32N","Central
European Time",2010-07-02 07:17:01.000
```

This is typical for all three files. Only the second and third comma separated values are needed. These values are the coordinates.

The picture below (of the pigeon data set) should illustrate the nature of the data for all three datasets. This data set contains 952,368 data points.

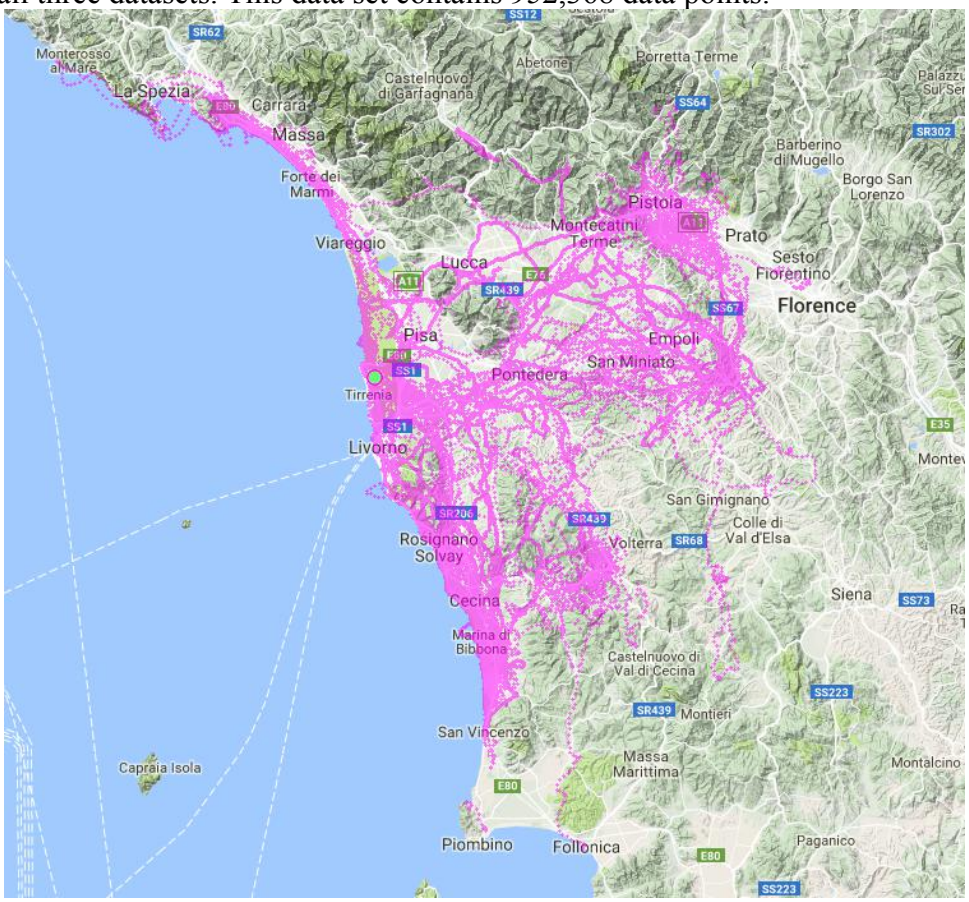

Figure 2-1: The pigeon data plotted on the coast of western Italy. (Source: MoveBank.org).

The second data set was information on vultures[3]. The name of this data set was "Factors influencing foraging search efficiency: Why do scarce lappet-faced vultures outperform ubiquitous white-backed vultures?" and only coordinates were pulled from it and in the same manner. This data set contains 346,650 data points.

The third and final data set was information on turkey vultures[4]. This was a distinct species from the other vultures that was understood to travel at a slower speed then the animals in the prior data set referred to specifically as vultures. This data set contains

---

[3] Spiegel O, Getz WM, Nathan R (2014) Data from: Factors influencing foraging search efficiency: Why do scarce lappet-faced vultures outperform ubiquitous white-backed vultures? Movebank Data Repository. doi:10.5441/001/1.pr1vj29n Retrieved March 2017.

[4] Barber, David, and Bildstein, Keith. 2017. Turkey Vulture Acopian Center USA GPS. Downloaded from Movebank.org Retrieved April 30, 2017.

721,051 data points.

In total there are 2,020,069 data points across all three files.

b. Related Work

A search for related work found several papers that cover similar problems to the challenges faced in this project. They do however do little to add new information and instead give guidance on what parts of the project have issues to consider and technical challenges, which, are not new issues and have possible extant or even well known solutions. What follows is knowledge gained from related works and organized by topic.

The Essentials of Parallel Computing

This simply means working on multiple processors[5]. Technically, the processors don't have to be working on the same tasks for it to be a parallel computing application, but in this project that the processors do very little in that manner with the exception of a few tasks reserved for the first process. This section came from a journal article on designing a Parallel Computing course. That parallel system must rely on multiprocessor hardware with an operating system that supports multiprocessing and the cluster must be networked. The design runs control from a master node. This is common with parallel systems on modern operating systems, unless more than one node is used to direct the use of the others (think main thread and *nix operating system design). The last step is benchmarking. It is fair to note though that in practice other related activities also go on, but essentially that is all that was intended for this project so the overlap is obvious.

Basics of Design Principles

A major problem with implementing an HPC solution arises when there are fault tolerance issues[6]. Addressing these issues takes a tradeoff for time. Many processes rely on heavy memory duplication at some level of the implementation, like the processors (viewed as nodes), so if there is an incident that causes undefined behavior the data can be quickly recovered. Others are saving a recent safely processed state of the system and setting up a publisher-subscriber design for sending information so redundancy can be built in. They all take additional time. This may be done at the middleware level on all modern systems. Algorithm and data structure design at the application level can directly relate to what types of multiprocessing errors can go wrong at different levels of the system design.

Metrics of Performance

One way of measuring the performance of the system is by dividing it by both the

---

[5] Karl Frinkle, Mike Morris, "Developing a Hands-On Course Around Building and Testing High Performance Computing Clusters," Procedia Computer Science, vol. #51 (2015), 1907-1916.

[6] Basem Almadani, Abdallah Rashed "Enforce a Reliable Environment in Parallel Computing Applications," Procedia Computer Science, vol. 63 (2015), 24-31.

processing time (which goes without saying) and the time spent moving data around[7]. Because HPC systems are used to handle very large data sets the time itself that the system spends transferring data is itself a relevant factor. And according to "A Performance Characterization of Streaming Computing on Supercomputers,"[8] the future focus of HPC is likely shifting from compute intensive computing to data intensive computing. This article introduces two metrics that might be applicable to this project:
1. A metric of the size of a stream element divided by the time it takes the transfer function acting upon it to return; AND
2. A metric of the size of the stream element divided by the time it takes for processing function acting upon it to return.

Adaptive Approaches

There are tractability problems with certain types of complex optimizations[9]. Since fully reliable prediction can be excessively resource intensive or impossible strategies are developed to predict likely outcomes weighted as expected values. The evolution of complex optimization methods, at least when it is automated, has a tendency to be based on statistical systems that depend on detecting trends in available aggregate data but do not require a precise answer of what is optimal or a proof of the answer's quality. Instead, it hopes that statistical observations will bear out the likelihood of a future event.

Finally, in addition to those four works information about the specifics of MPI were gathered from the book "Using MPI-2: Advanced Features of the Message Passing Interface"[10].

---

[7] Stafano Markidis, Ivy Bo Peng, Roman Iakymchuk, Erwin Laure, Gokcen Kestor and Roberto Gioiosa, "A Performance Characterization of Streaming Computing on Supercomputers," Procedia Computer Science, vol. #80 (2016), 98-107.
[8] Ibid.
[9] Jinn-Tsong Tsai, Jia-Cen Fang, Jyh-Horng Chou, "Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm," Computers & Operations Research, vol. #40 (2013), 3045-3055
[10] Gropp, William. Lusk, Ewing. Skjellum, Anthony. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Boston. 1999.

3. Proposed Solution

This section covers data sources, algorithms & data structures, performance metrics, optimization strategy, and record keeping.

3a. Overall Design Plan

The overall design plan is simple. First choose data sources. Then write a program to parse the data, process it, and write the results to an output file. Finally, take performance measurements and use these measurements to determine under what conditions to run the program to get optimal performance and under what load you can run it without reliability issues. Since algorithms to process data once it has been read into the program can simply be added onto without any dependencies from other aspects of the design plan and because it is a complicated segment it appears after this Overall Design Plan as a further zoom in on details.

3a.1 **Data Sources**

It seems almost totally impractical to find data sources that have a large enough number of independent points over time, but there is plenty of data with fewer numbers of individual points but a sufficient number of data points in total. Research yielded several large data sets amounting to hundreds of megabytes from MoveBank.org (detailed earlier). This is sufficient to work with. Because the system is supposed to be going as fast as possible there is not need to worry about the time stamps on the data points, but the data should still approximate the general goal while being of sufficient size to challenge the cluster and better than a random data set because such data can validate calculation speed but is not structured like geographic data so determining how much processing capacity (or even data collection ability) won't be reliable. As such it will be difficult to predict how much processing power will be needed were this a live system with unknown data expected in the future.

3a.2 **Reading from Input Files and Recording to Output Files**

Interference from reading and writing can be a serious problem. There are issues with both what is allowed safely, the speed of those reads and write safety. In this application reads are handled by multiple accesses to the same file simultaneously. This is considered safe[11]. This is a design decision to balance system space with system speed. If all processes read from a distinct copy of the same file then access speed would be faster. Because writing is done to a separate file on each process there are no access issues or delays.

Another issue is a kind of overwrite protection. When the application is run it creates output files. This feature ensures that the application writes to a different file every time. It works by accessing a configuration file that persistently stores the test number. When a new test is run the program increments the number in the file and uses

---

[11] Anonymous Author. 2013. "Read file simultaneously with two different programs"http://www.unix.com/unix-for-dummies-questions-and-answers/216405-read-same-file-simultaneously-two-different-programs.html Retrieved May 7, 2017.

the new number to make a new directory, which has the same name as that number. Then it writes output files to that specific file thus preventing subsequent runs from overwriting the current ones.

Finally when reading the file the system must manage to ensure that all of the processes read on their sections to avoid duplicative work. This is done by reading all of the lines but only processing the ones where the line number modulus the number of processes is equal to the process ID number. In this way the process only processes every nth line where the n is the process ID number.

3.3 **Performance Metrics**

In addition to the formulas and data structures designed for the calculations there are the metrics of how these formulas perform on a given system during a given time period. This can be used to determine essentially the frames per second. That means the number of times per second the system can check for an update containing up to a given number of points and without causing a delay in the number of times this can happen per second due to processing, writing output or anything else.

3.4 **Optimization Strategy**

When all of this is finally put together there should be enough information to estimate what would be effective improvements to the configuration of the program in it's environment and it's constraints and inputs. That is of course just the general idea, there are many things that could be relevant in practice, much more than what this report will cover.

This concludes the description of the Overall Design Plan.

3b. Algorithms

The algorithms will describe the formulas in a way that addresses technical aspects of the design requirements.

3b.1 **The Distance Formula**

The distance formula calculates the distance between the given data point's coordinates and the fixed point (which is at coordinates 0,0).

The program runs a function f(x,y) that takes the coordinates of a point at a given time and calculates the distance between that point and a fixed point (XCONST, YCONST). Deriving the formula from Pythagorean the theorem for the special case of a right triangle and multiplying by the number of miles in a degree 69.11 gets:

$$f(x,y) = \text{sqrt}[\text{absval}(XCONST-x)^2+\text{absval}(YCONST-y)^2\ ](69.11) \quad (1)$$

Although the absval function has semantic value eliminating it yields the equivalent:

$$f(x,y) = \text{sqrt}[(XCONST-x)^2+ (YCONST-y)^2](69.11) \quad (2)$$

3b.2 **The Direction Formula**

This formula is to determine the direction in degrees that an animal traveling towards the fixed point must travel to reach it by a straight line. It is derived from the tangent formula for the distance between the x coordinates divided by the distance between the y coordinates. Because the fixed point is at 0,0 this part of the function is

simply taken as x/y where x and y are the coordinates of the given animal. Because tangent will return the direction in radians it is then converted to degrees with the factor 180/PI.

$$f(x,y) = (180 * \tan(x/y))/PI \quad (3)$$

3b.3 **The Time to Arrival Formula**

This formula is determines the best case scenario time in hours for the animal to arrive at the fixed point based on it's known top speed.

$$f(dist) = dist/\text{Speed of this Animal} \quad (4)$$

3b.4 **The Density Formula**

The density formula is actually two formulas in one. The first counts how many points are within a specified rectangle by incrementing a counter each time a data point is read from it's file. The second formula takes the number of said points and divides them by the area of the rectangular region in miles. The variables for the box are referred to as X1 (left bound), X2 (right bound), Y1 (top bound), Y2 (lower bound).

1st: f(x,y) if (X1 <= x <= X2) AND (Y1<= y <Y2) then pointsInBoxCount = pointsInBoxCount + 1 (5)

Else return;

2nd: density() = pointsInBoxCount/(X2-X1)(Y2-Y1)(69.11)^2 (6)

3c. Code Structure

Overall Design Plan -

While the 3a described the overall design plan and 3b described the basic algorithms being used this section will describe the general structure and flow of logic in the code. Although several versions of the program were tested this basic structure remains the same.

State 0: Set-Up (Process 1)

1. Take in settings for this session.
2. Create Subordinate Threads

State 1: Read The Next Data Point from the Input Files. (Branch Processes)

1. Feed Data into Memory

State 2: Data Received and Parsed (Branch Processes)

1. Data is tokenized and the coordinates saved.
2. Convert Tokenized Coordinates to Doubles

State 3: First Calculation — (Branch Processes)

1. Determine the Distance between the coordinates point and the fixed point.

State 4: Second Calculation — (Branch Processes) – program versions 2,3 & 4 only

1. Determine the Direct the Animal must travel in to reach the fixed point

State 5: Third Calculation — (Branch Processes) – program versions 2,3 & 4 only

1. Determine the Time To Arrival to the Fixed Point given the speed of the specific animal

State 6: Fourth Calculation — (Branch Processes) – program version 4 only

1. Determine if the data point is within the area of interest rectangle specified on program start.
2. If so increment a count for this process.

State 7: Write Results to Output File (Branch Processes)

1. Do it.

State 8: Return to State 2

1. Continue this pattern until all the points in all the data sets in the given version of the program have been processed.

State 9: Print to Stdout

1. Calculate Density from the Count generated in state 6 and the area known from the specified rectangle at system startup.
2. Print Density and Any Time and Function Testing Information to Stdout.

Figure 3-1: The basic code structure.

4. Programming Environment

4a. System Information

For the programming environment I used Message Passing Interface (MPI) extended C language on the Montclair State University cluster machine copou.csam.montclair.edu. It runs on top of Red Hat Linux 5.

The machine is a 65 nodes, 1 master and 64 slave each running with two 64 bit quad core AMD Opteron 2378 chips at 2.4Ghz and 16GB of RAM local to each node. In all this totals 512 cores on 128 CPUs forming 64 slave nodes in addition to the 8 cores, 2 CPUs and two CPUs on the master node.

On the Message Passing Interface

The Message Passing Interface or MPI [12]is a standard for running multiple instances of a program in coordination with one another and with the capability to pass messages from one instance to another in order to share information or synchronize the programs.

If run with a sufficiently low number of instances it is assumed that each instance, which is a process, will use the dedicated resources of a single processing core and so while each processor is sequentially executing segments of code from a single process work will be done simultaneously on multiple instances of the program or as it is also called in parallel execution.

To run MPI programs first MPI programs must be compiled through a special compiler that extends the programming language it is built to work with and allows the use of special MPI function calls. Then in order to run multiple instances of the program the compiled application must be called by a purpose built application.

When writing an MPI program in C the compiler requires that:

```
#include <mpi.h>
```

be included as a header in order to call MPI functions.

The particular implementation of MPI installed in this system was MPICC 1.2 a C language implementation that is run as an alternative to gcc (or Intel's icc) and must be used in order to compile the program. Besides the capability to compile MPI programs mpicc is a clone of gcc. In order to compile an this project the command used was of the form:

```
mpicc –o testG mp_6r.c –lm
```

where `–lm` is the option to link in the math libraries. This option was simply needed due to the precise configuration of the copou server and may be absent on any other machine that the project code may be compiled on.

MPI programs are run with the following command:

```
mpirun –np [number of processes] [application path]
[Arguments .. 1.. 2… etc…]
```

[12] Ibid. 10.

4b. Programming Experience

Here is some pseudo code to explain the general flow of the processing:

Figure 4-1 Pseudocode

```
simplepath(){
                int processID = MPI_rank();
                int num_procs = MPI_size();
                openread(file processor);
                openwrite(file);
                while(
                        ( (char[] str = getline()) != null)
                 ){
                        if(linenumber==1) skip; //because the first line is the header
                        time s,t;
                        partitionline(); //break the line up
                        extract x and y coords (type double) from partitioned data //extract
nums from text, convert to doubles
                        Run the Processing Algorithms Here
                        write2fileForThisProcess(value,s,t); //log the analysis of this line
                        i++; //increments the number
                }
                close(read);
                close(write);
        }
```

Numerous difficulties arose during the development of this program. Some of the difficulties did not exist with small files.

One of those that happened was that while updating the software and running new tests, some of the older results the older results were being overwritten so a way needed to sort each log into a unique name and path combination to prevent overwriting or ambiguity on the ownership of the files was devised. This design choice itself took some time to decide.

There was some difficulty with version control and maintenance. This led to the loss of some early versions of the code as well as the loss of logs mentioned earlier. Creating this updated code introduced bugs into the program that did not impair prior versions. All of this was ironed out by version 1.

At this stage more debugging, adding to the code and improving the organization to this documentation were sorely needed.

The versions represent stages of completion of the project as well as different testing conditions.

Early testing results concluded that the server ran far faster than was necessary to get reasonable performance for this type of application. Given the number of instances involved had their been more specimens among these data points instead of more snapshots of the same community the system would have been able to update at a frame rate at least several times a second (this was later shown).

Early versions of the code also had a four process limit. This four process limit was a bottleneck in the earliest design because there were some concerns about what machines would be accessible to test the code on immediately. There were a couple of unusual counting errors when 8 processes were specified. That run was after the program had been altered to accommodate all available processors that copou had.

These updated tests expanded the panel to include more processes beyond four. The file was still limited in the sense that it wasn't currently integrating all the datasets so there was no integration lag time added to the end to end process time. This, however, is diminished by the fact that the file that had been tested contained more instances than the total amount of instances in the other two source files combined, so it should not change the order of magnitude of the performance.

Later still, there were two types of additions planned. The first was totally relevant to the project: to add more complex data operations for the hardware to run as tests. The second would merely be to improve code organization and add enhanced logging features that would make it easier to test these systems in the future.

At this stage it was observed that far more time goes into designing the precise test mechanisms than running the code. In this case tests can be run on the datasets in quite minimal time even with only one process active. This was a lesson in what programming a parallel system entailed.

Up until this point in the project there were no frame rate results. The end to end execution time of the program was being determined. This was on the metrics side of the equation and was not a programmatic solution to calculating the metrics.

In the following iteration, experiments indicated that increasing the number of processors seemed to do little to improve the end-to-end operation time of the program. The bottleneck was still very substantial.

At this stage writing to the log files seemed to be the biggest problem because the program had been crashing without populating the log files. It seemed like the only work around to this was to make all of the processes idle for a period of time to give the writing operations a chance to catch up with the other calculations that don't require IO. It would later be discovered that this was not the reason for the functional problems and it actually had more to do with the way data was being parsed. This parsing was creating memory overflow errors that looked like the types of errors that might be expected from writing operations.

Upon the next iteration, the focus shifted to adding all of the additional algorithms, meaning all of the algorithms except the density algorithm, which was added later to make the program a little more useful. It was useful in the sense that now the program could be used to focus in on a particular area for a data set or reveal that for a certain rectangular region it was an empty set. This could help develop new tests by revealing what size and density of areas the hardware really needs to process well in. Once that is known it can be put in terms of a metric of performance.

After another five tests it became clear that there was a problem inherent in the OS or its dependencies that was making it impractical to write from multiple processes. Later though this was revealed to likely be a memory leak in the design of the parser, or

at least that the parser design was a contributing problem. This parser problem came down to safety inside C and its libraries not the operating system.

Just before version three, a couple of speculated features were cut from the plan because they seemed to take too long to develop and were of little value.

First, density functions would be the only density processing routines. Prior to this the density functions included density change which was calculated at the end of various present time segments, but creating that mechanism began to seem impractical given the need to get a draft done early enough to ensure that something would be done well before the last minute.

Second, time in the project was changed to refer to the amount of time it took to process everything and frames. Since it's complicated and not really an issue in a live system where the system would be waiting for new inputs the times included in the data points in the input files were ignored. Therefore, no synchronization of the data points across files in order to group the data points more tightly would be done. Synchronizing in any case was considered a possible way to skew the results of the performance tests because it would prevent the program from being run as quickly as possible.

The development of program version three came next. While testing this version the first time it was discovered that there was a typo in the code that kept the file reader for vultures from closing. This presumably caused behavior, which worked on a single processor test and hung thereafter. This however did not fix the problem but a work around was designed by migrating the testing from using the linux time function to the C clock function.

Below was the interesting result. Somehow processing completed when only one processor was assigned. Stranger the code started to hang about the time that the process would have completed on any other operation. The best guess to date was that something happened with the operating system.

For example proc1.txt has 81,194 lines * 16 which is about as many lines as get covered when execution finishes with one program. Therefore, the function actually seems to be finishing execution of the program and then hanging instead of cleaning up and exiting.

For Table 4 – 1 below note that NP stands for Number of Processes so column NP=1 are the run times for when the program was running on a single process.

Table 4-1 Test of Pigeons and Vultures files with Distance, Direction and ETA computed and printed to output files.

Test #	*NP = 1*	*NP = 2*	*NP = 4*	*NP = 8*	*NP = 16*
1	0m44.800s	HANG	HANG	HANG	HANG
2	0m38.297s	--	--	--	--
3	0m41.409s	--	--	--	--
4	0m42.290s	--	--	--	--
5	0m32.678s	--	--	--	--
AVG					

Because the processing seemed to complete before the program hanged this was seen just as a matter of needing to quit the program when done. A timing function was inserted in the C code itself and new numbers were available from that. A further note is that it appears the program is actually finishing everything except printing and that it was just being kept open for it to return.

It was found that trying to go beyond that would make results even more erratic. A test of the same program running 128 processes will yield `3.410000s` and yet it will take much longer than that for the program to print all of the output to the screen.

Somehow 256 processes goes up and allegedly yields 10.36s. Specify 512 process which is equal to the total number of cores in the slave processes (64*2*4) and it yielded 23.73s. Plus, it becomes difficult to track when the system is done because all of the messages are shuffled. It seemed impractical to go any farther with tests beyond 64 because it seemed that beyond this number of processes the program could not be trusted to display an accurate end to end time for operation.

5. Experimental Results

This program was tested hundreds of times across several different versions of the code. Below is the write up of the tests run on each version of the program, and some interpretation of those results. The test for each version of the program is referred to by the name of the binary that was run. For instance version 1 is referred to as Test C or TestC without the space (the name of the binary) because the two prior version relied on code with defects in the design or an incomplete implementation.

Version 1 runs just one data set known as pigeons.csv (and described earlier in the report) and one algorithm, the distance algorithm.

Version 2 runs the same single data set but runs the direction algorithm and the time to arrival algorithm after it finishes the distance algorithm.

Version 3 runs almost as Version 2 does but it supports an additional data set called vultures and has been designed determine the execution time of the program itself.

Version 4 adds a third data set called turkey vultures and includes the density per area calculating algorithms (there are two working in concert).

Table for Test C

An interesting factor is that that going over one process introduces a bottleneck and 2, 32, and 64 processes cause huge performance problems. In fact, for 32 and 64 the process gets to the end of computation at decent speed and then seems to hang indefinitely.

Table 5 – 1 shows a program version where, with the minimum number of calculations (just the distance computed) per data set, NP=16 is slower than sequential execution. Thus the bottlenecks are similar in size to the time saved by running multiple processes.

To clarify, NP refers to the number of processes running on the program.

Table 5-1 Test of Pigeons file Only with Distance computed and printed to output files

Test #	*NP = 1*	*NP = 2*	*NP = 4*	*NP = 8*	*NP = 16*
1	0m25.441s	3m14.518s	1m11.758s	0m46.093s	0m31.069s
2	0m26.683s	3m14.749s	1m10.424s	0m45.262s	0m31.023s
3	0m24.361s	3m15.237s	1m10.743s	0m44.880s	0m31.611s
4	0m28.902s	3m15.988s	1m10.083s	0m45.950s	0m31.377s
5	0m20.621s	3m14.832s	1m9.961s	0m45.694s	0m31.494s
AVG	25.2016s	195.0648s	70.5938s	45.5758s	31.3148s

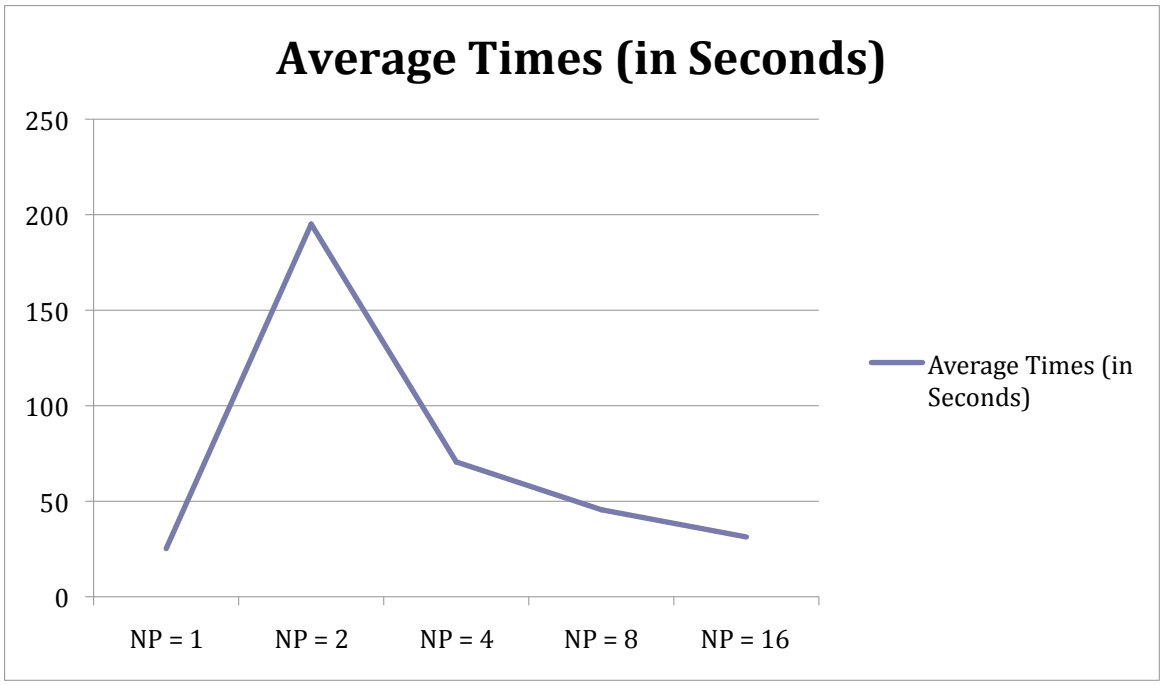

Figure 5 – 1 Average Times for the run of Version 1, which is compiled as TestC.

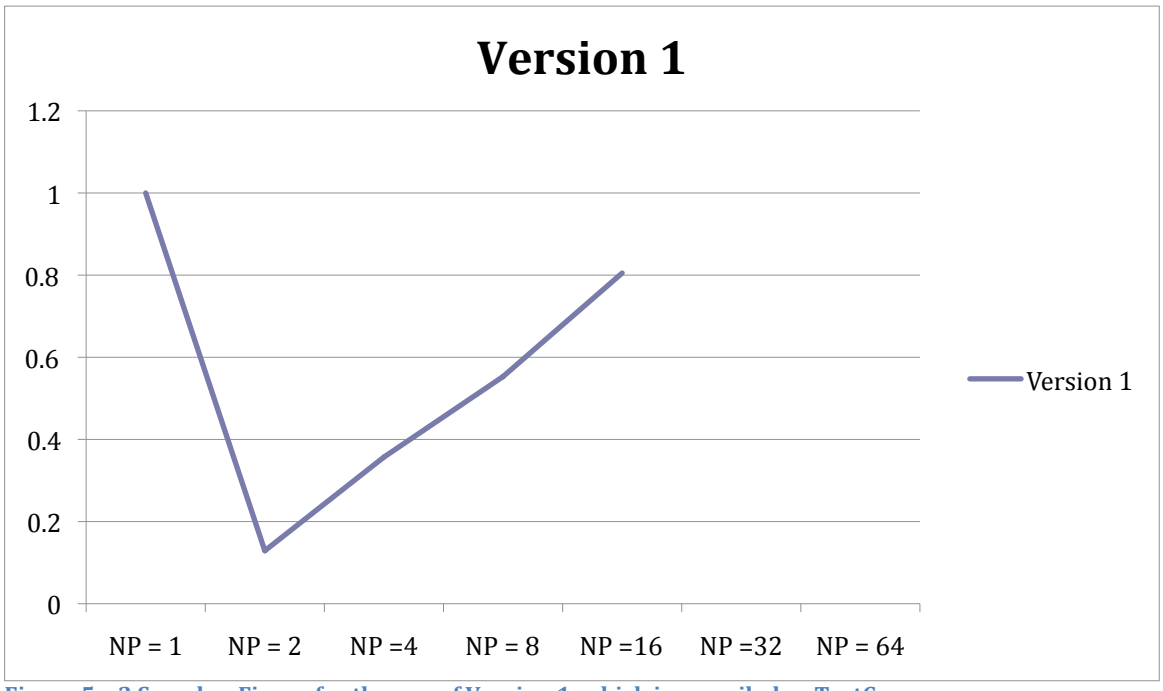

Figure 5 – 2 Speedup Figure for the run of Version 1, which is compiled as TestC.

The Speedup formula calculates the speedup of the parallel version of the program relative to the sequential version. Speedup is expressed as the ratio of the given run time for the parallel program with a given number of processes running (referred to as NP = #) divided by the run time for the sequential version of the program. Speedup = TimeRunninginParallel/TimeRunningSequntially.

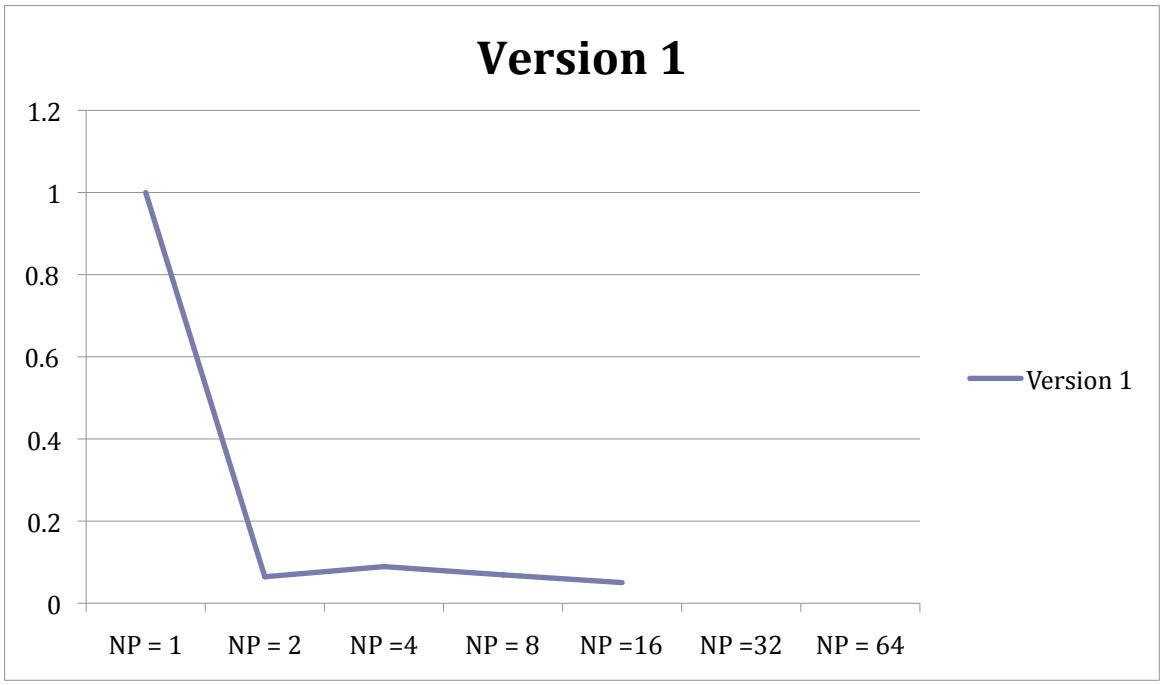

Figure 5 – 3 Efficiency Figure for the run of Version 1, which is compiled as TestC.

Efficiency measures how much work each processor (when Number of Processes>1) is doing relative to a single processor version. It is the ratio calculated by the speedup divided by the number of processes (NP). Efficiency = Speedup/NP.

Table for Test D

Test D included NP = 1,2,4,8 and 16 with the pigeons data set and some changes to the test.

The distance algorithm was modified to show miles instead of degrees. 1 degree was estimated to be 69.11 miles.

A direction function was added. This function provides the direction of travel to the fixed point in degrees.

A time to arrival function was added. This function provides the travel time at the speed of the animal in question, in hours, from the animal's location to the fixed point.

Table 5 – 2 (below) shows data for program version 2, which increases the number of calculations each time the system processes a record. Thus the gap should between NP=1 and NP=2 should be smaller in the second program than the first. It is clear that it is although it is not yet fast enough to deal with the bottleneck.

Table 5-2 Test of Pigeons file Only with Distance, Direction and ETA computed and printed to output files

Test #	*NP = 1*	*NP = 2*	*NP = 4*	*NP = 8*	*NP = 16*
1	0m39.598s	2m57.536s	1m12.980s	0m46.015s	0m29.564s
2	0m21.394s	2m56.975s	1m11.890s	0m47.078s	0m30.585s
3	0m32.040s	2m58.384s	1m11.944s	0m46.215s	0m31.122s
4	0m29.147s	2m57.764s	1m11.394s	0m46.818s	0m30.969s
5	0m22.046s	2m56.152s	1m12.913s	0m46.115s	0m31.844s
AVG	28.845s	177.3622s	72.2242s	46.4482s	30.8168s

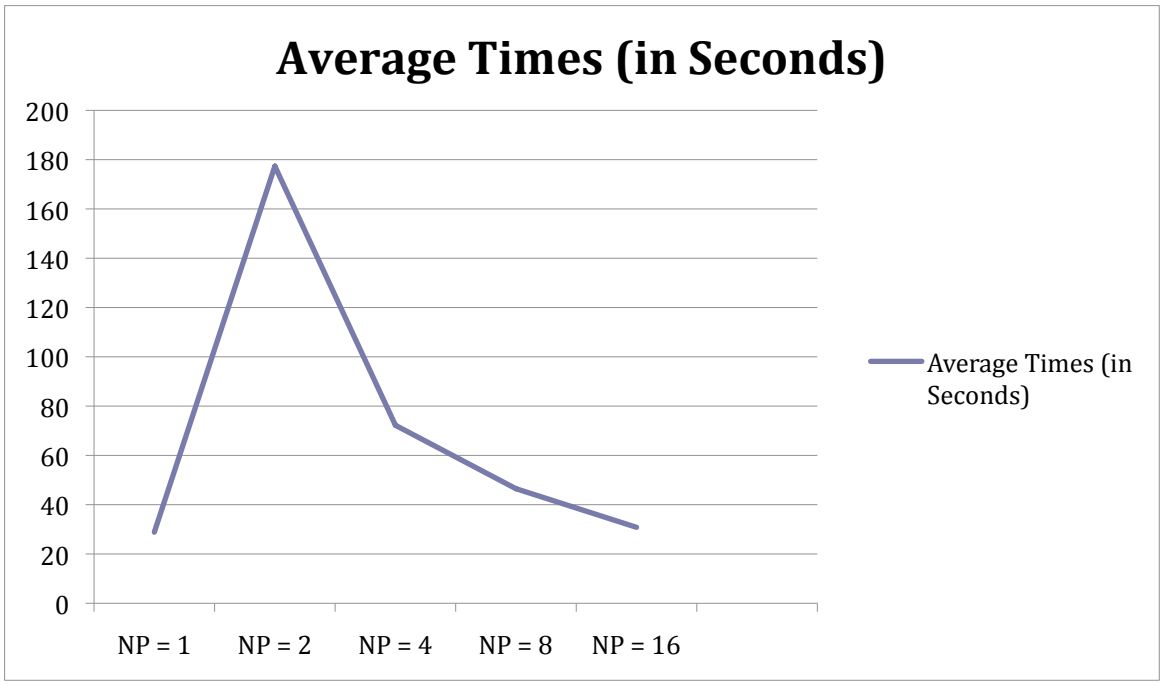

Figure 5 – 4 Average Times Figure for the run of Version 2, which is compiled as TestD.

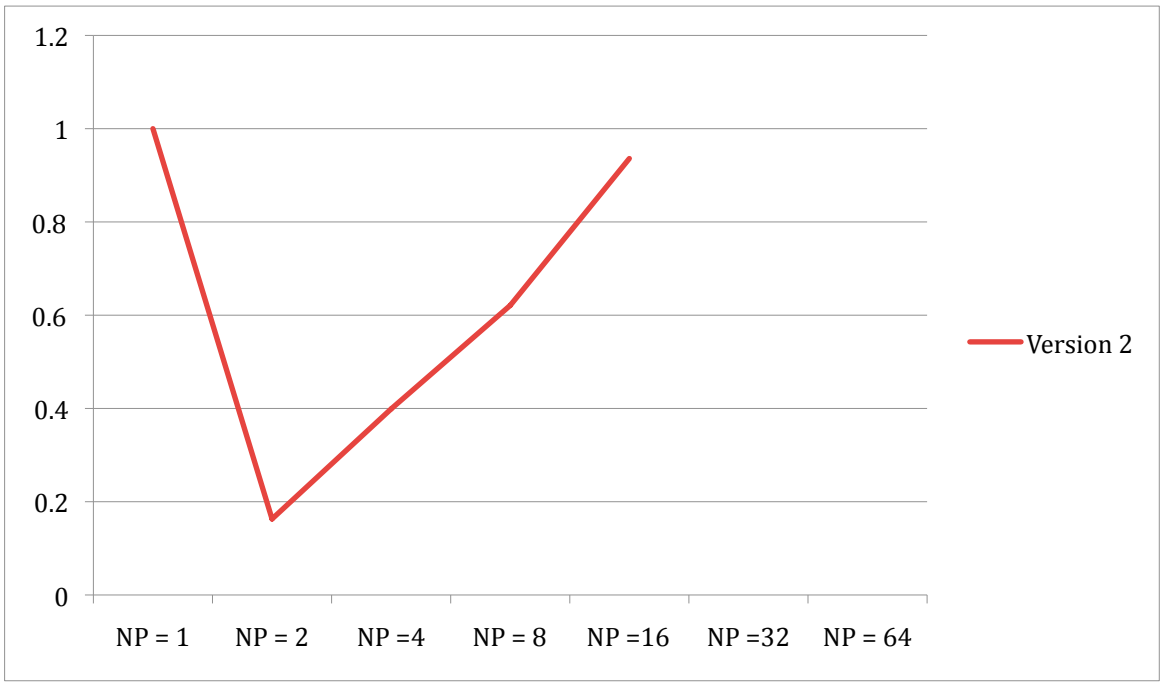

Figure 5 – 5 Speedup Figure for the run of Version 2, which is compiled as TestD.

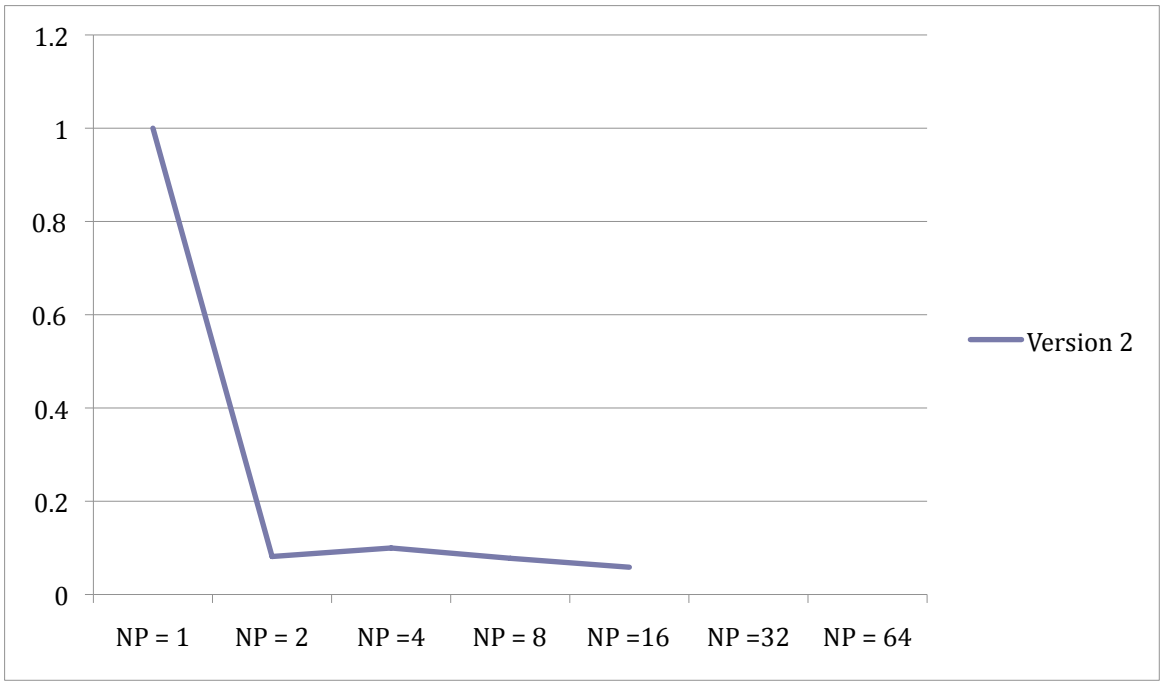

Figure 5 – 6 Efficiency Figure for the run of Version 2, which is compiled as TestD.

Test E:

This time vultures are also read. The ground speed is given in the file but the objective is to calculate the speed relative to the maximum speed to answer the best-case scenario arrival time.

Looking at NP=1 and NP=16 in Tables 5 – 3 and 5 – 3E for program version three, the addition of a new data set seems to slow things down; however, the bottleneck is tangibly overtaken by running the program at NP=64.

Table 5-3 – Corrected Program: Test of Pigeons and Vultures files with Distance, Direction and ETA computed and printed to output files

Test #	*NP = 1*	*NP = 2*	*NP = 4*	*NP = 8*	*NP = 16*
1	30.110000s	235.340000s	94.800000s	61.210000s	38.710000s
2	36.640000s	235.050000s	93.640000s	59.260000s	40.590000s
3	30.130000s	232.220000s	92.680000s	59.480000s	39.170000s
4	35.160000s	233.080000s	94.640000s	59.120000s	39.160000s
5	28.460000s	233.610000s	95.180000s	58.940000s	38.660000s
AVG	32.1s	233.86s	94.188s	59.602s	39.258s

Table 5 – 3 is extended in Table 5 – 3E on the next page.

Table 5-3E - Extended Results for Prog3 (TestE) for NP = 32 and NP = 64

Test #	*NP = 32*	*NP = 64*
1	36.340000s	26.560000s
2	32.920000s	25.780000s
3	33.180000s	22.640000s
4	33.490000s	25.690000s
5	32.480000s	27.590000s
AVG	33.682s	25.452s

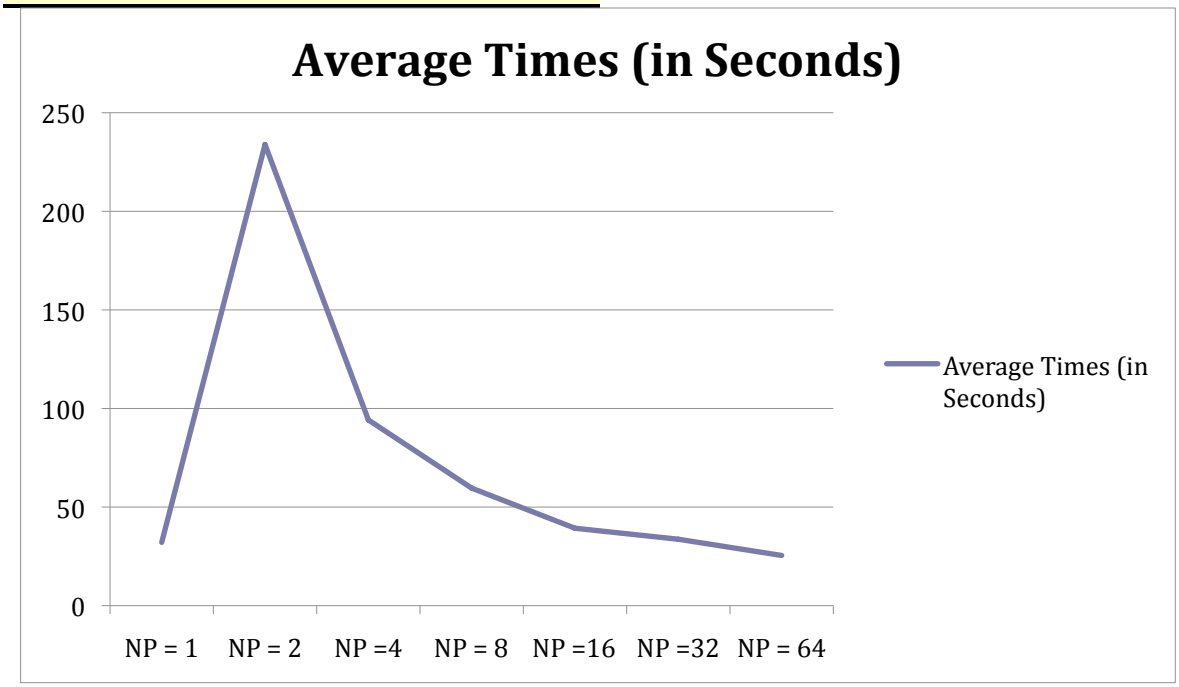

Figure 5 – 7 Average Times Figure for the run of Version 3, which is compiled as TestE.

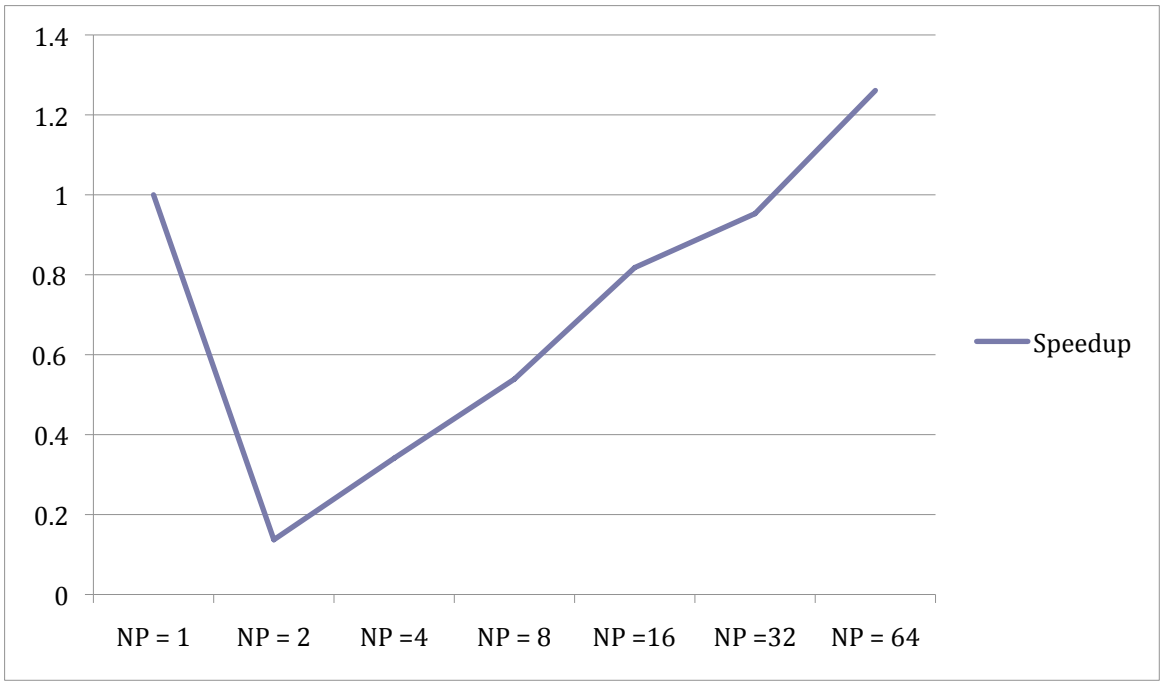

Figure 5 – 8 Speedup Figure for the run of Version 3, which is compiled as TestE.

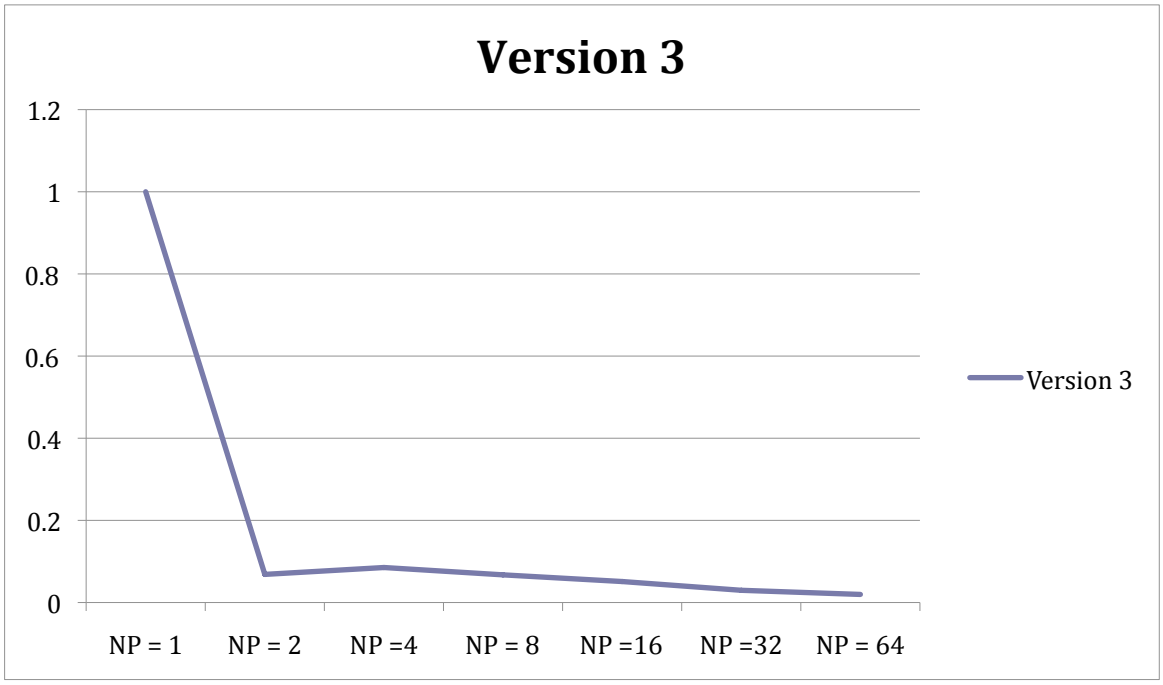

Figure 5 – 9 Efficiency Figure for the run of Version 3, which is compiled as TestE.

TestG:

This was the last test. It added the density functions and the third data set known as turkey vultures. Because this version had a density function it needed to take in a region (in this case a rectangle) that could be used as a means of specifying the area. In order to run this version with the density test it was necessary to enter four arguments which where the latitude and longitude of the top and bottom and sides of the rectangle.

To makes matters easy those parameters were -180 180 -180 180 to encompass the globe.

Table 5 – 4 below describes data from version 4 of the program and is a similar phenomenon to the data observed. It takes slightly longer on average. Comparing NP=1 and NP=16 there is a clear reduction in the overall processing advantage compared to the last set of tables. That continues with a divergence from the slight lead that NP=64 had over NP=1 in program 3.

Table 5-4 – Final Program – Contain just over 2 million records over all three data sets.

Test #	*NP = 1*	*NP = 2*	*NP = 4*	*NP = 8*	*NP = 16*
1	58.170000s	368.330000s	149.330000s	94.950000s	62.060000s
2	43.130000s	370.310000s	151.180000s	93.660000s	62.670000s
3	40.220000s	369.680000s	149.970000s	93.940000s	61.500000s
4	52.400000s	367.870000s	147.770000s	93.020000s	61.940000s
5	48.490000s	371.120000s	151.530000s	94.140000s	61.720000s
AVG	48.482000s	369.462000s	149.956000s	93.942000s	61.978000s

Test #	NP = 32	NP = 64
1	57.460000s	48.460000s
2	55.420000s	50.640000s
3	58.880000s	52.790000s
4	56.730000s	52.690000s
5	59.830000s	48.270000s
AVG	57.664000s	50.570000s

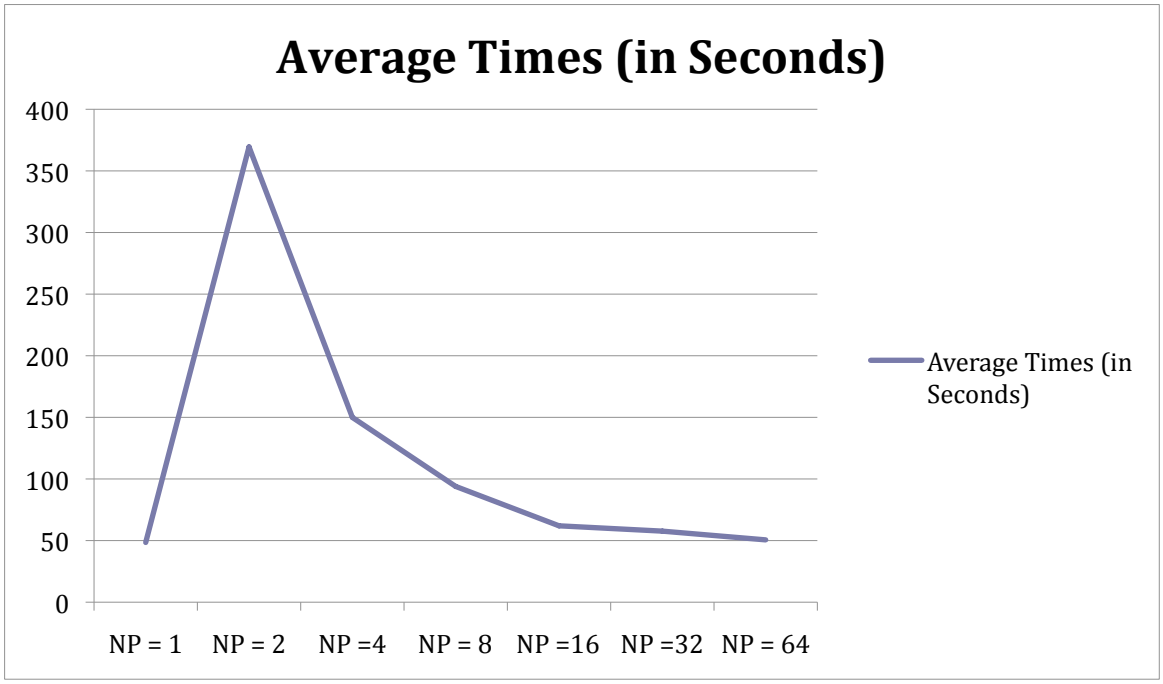

Figure 5 – 10 Average Times Figure for the run of Version 4, which is compiled as TestG.

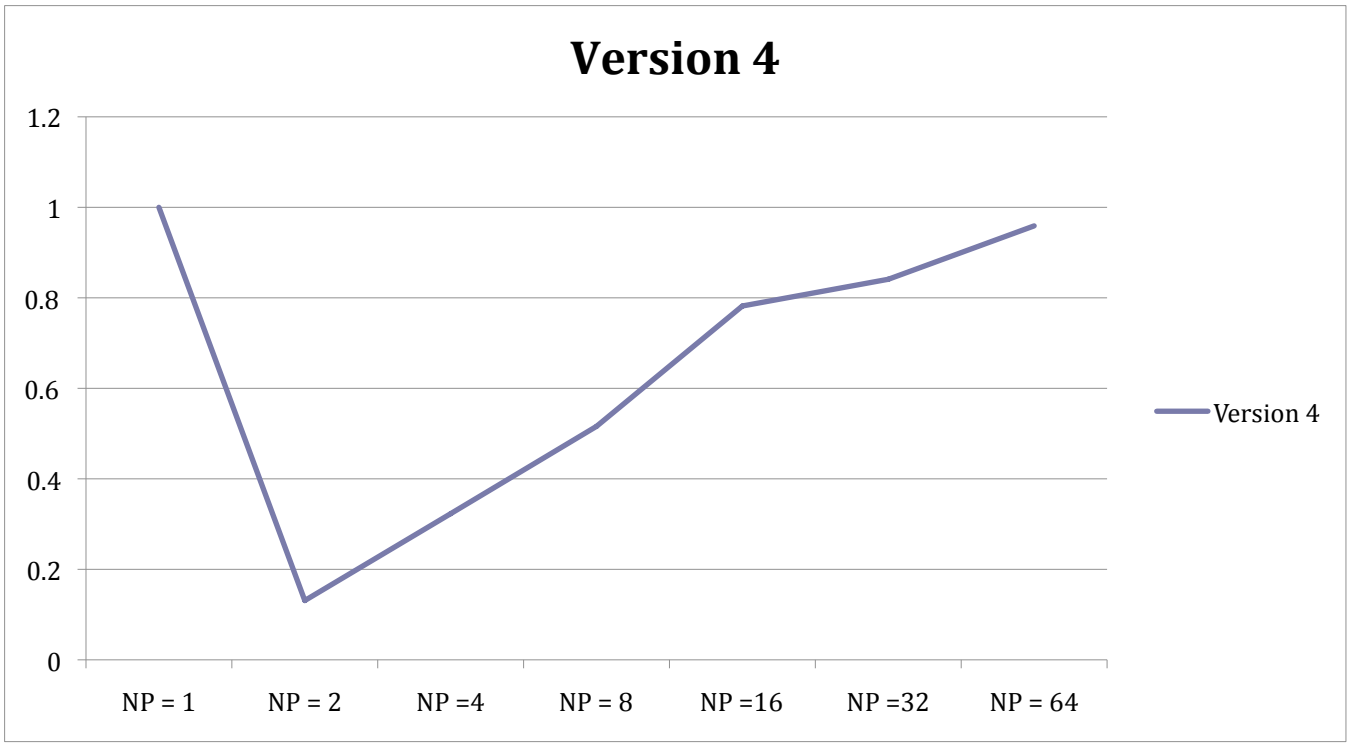

Figure 5 – 11 Speedup Figure for the run of Version 4, which is compiled as TestG.

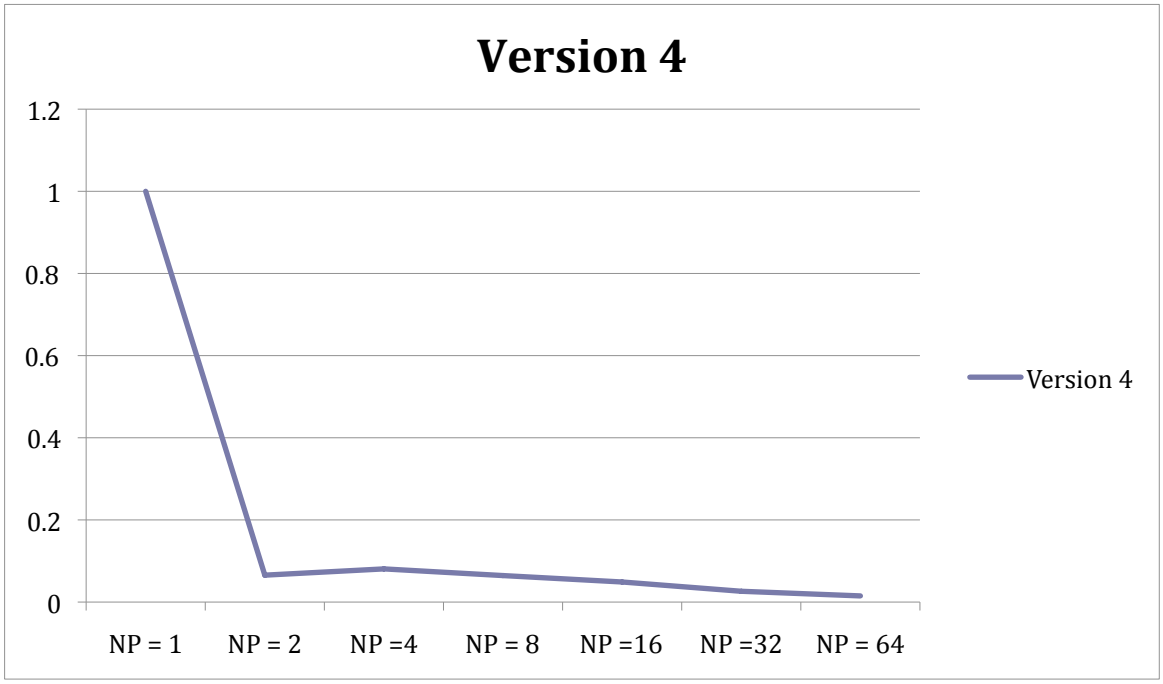

Figure 5 – 12 Efficiency Figure for the run of Version 4, which is compiled as TestG.

5a. Metrics

After getting the experimental results, all of the necessary information to answer the original question is available, but providing an answer requires bringing all of that information together and computing it as a metric. The tables on the next page have the core metrics, which are how many data points can be computed per second and how

many points per frame per second? For the second part only the answers for a few different frame rates are provided since otherwise there would be tables for any number of frames equal to the number of points plus one for all frame rates greater than the number of points since all such numbers would be impossible. Then the calculations would have to consider how high a frame rate the hardware could reliably sustain which is another problem. Below only frames of 4,6, 12 and 30 are provided in order to be practical.

Table 5-5 Basic Metrics

# Data Pts	*NP=1 Pts/Sec*	*NP=2 Pts/Sec*	*NP=4 Pts/Sec*	*NP=8 Pts/Sec*	*NP=16 Pts/Sec*	*NP=32 Pts/Sec*	*NP=64 Pts/Sec*
952,368	33016	5369	13186	20503	30904		
1,299,018 (adds 346,650)	40467	5554	13791	21794	33089	38567	51037
2,020,069 (adds 721051)	41666	5467	13471	21503	32593	35031	39945

Table 5-6 Frames/ Second at Process Numbers and Approx 2 Million Records

#Frames/Sec	*NP=1*	*NP=2*	*NP=4*	*NP=8*	*NP=16*	*NP=32*	*NP=64*
4	10416	1366	3367	5375	8148	8757	9986
6	6944	911	2245	3583	5432	5838	6657
12	3472	455	1122	1791	2716	2919	3328
30	1388	182	449	716	1086	1167	1331

It is clear that in both tables the amount of points/second (which is equal to 1 frame/second) or frames/second dips at NP=2 and then gets closer to the sequential execution numbers as NP approaches 64. The actual bottleneck interestingly seems to fluctuate despite and does not consistently seem to diminish or increase as the number for NP gets higher.

6. Challenges and Future Work

6.1 **Challenges**

This section covers in detail the challenges that occurred after my initial project and solution proposal and how each one was resolved.

6.1.1 **Crashes.**

Apportionment

There was an issue getting the program to measure when to stop reading the input files. This was potentially causing all sorts of trouble that amounted to a crash and a failure to write to the log files. This was also unnecessarily complicated code. Eventually, at the suggestion of Professor Robila this code was substituted for an approach that had each processes only use the nth line in each file where n was the line number modulus the process id.

Memory and Parsing

A big part of the difficulty came from initially unnoticed problems with the parser. This may have been the original cause for the crash. The original custom parser was not working well at its job, that is, to separate the read in lines from the input file into tokens so the needed information could be isolated and used by the right processing function. The parser was then switched to one that used strtok, a standard function.

Later the strtok() method was switched to the supposedly safer strtok_r() method when the crashing issues continued. This seemed to make no difference in crashes. Then subsequent calls to strtok_r() were taken out of its null terminated loop. This seemed to fix the problem. Here are two potential reasons why. First, because the files read from kept information in a predictable order just calling strtok_r() until it came to the needed information may have worked. Second, when char* was copied into a char[] it may have dropped the termination string and the resulting string which was fed in may have caused the function to never return null or access unauthorized memory.

6.1.2 **Record Keeping**

Well before Version 1 when the program initially got working parallel computation wasn't crashing, but each test had its records overwritten by a subsequent one. It was necessary to develop a system for organizing the tests so they posed little risk of overwriting each other during the expected lifetime of the program before maintenance had to occur. Maintenance here means a number of tests so large that, for example, they could not be represented by an integer type (as set on that system since C doesn't specify a fixed integer length) value or had to be loaded from a network connection because of a lack of available space, or to do something about the increased latency when writing to records physically farther from the site of computation.

6.1.3 **Keeping to the Schedule**

The original project schedule turned out to be fairly accurate in one respect and wrong in two others. It is fairly accurate in the sense that multiple responsibilities could be juggled during a day where this project was only one and still have down time. It did not account for the level of technical problems that would be discovered and how to predict the length of time needed to deal with a problem of an unknown nature when it

came up.

6.1.4 **Other Unknowns**

In addition to the technical problems with crashing it took some time to estimate just what should and shouldn't be included and in what order. This led to some time being used to move on to clarity about the exact details of the presentation and report, et cetera.

6.2 **Future Work**

The program could be used to test other streaming applications. This assumes that the equations used in those cases were also of linear order and there was no big variability in the number of linear passes of the data it needed to make or guard conditions that took a variable amount of time whether they were met or not. In that case adapting those tests to this software sounds provisionally feasible though bottlenecks in the hardware could reveal themselves as obstacles. The other obstacle could be that there are different types of operations that also need to be executed in addition to the original program to make a realistic simulation given the needs in practice of other stakeholders.

Graphics and Video have higher sample rates than are necessary for animal tracking and those experiments would be interesting. That model may require less computation and more waiting because the emphasis and probably the limiting factor would be on distribution of resources more than on the parallel nature of the program's execution.

If these were performance tests were to continue, investigating the function of streams external to computation such as the use of printf statements, that seemed to lag and be executed in a non-sequential way even when the operation was supposedly specific to a single core could be very useful in future optimization efforts.

Lastly, to take an idea from the related work section a useful direction would be to break down system timing to take into account overhead created by multiprocessing as much as possible such as message passing.

7. Conclusions

During the research simple linear equations were run on one, two or three datasets with one, three or four calculations being run on each point (detailed in Experimental Results earlier). Regardless of the version of the application being tested though the program first read in data, processed it, printed the processed result to an output file and then repeated the process for the next data point in the open input file. The variance in versions of the application means that the data tested shows not only variance in the length of the run but the load on a proportionate basis to each point calculated once correcting for the substantial overhead from I/O and multiprocessing.

First, and perhaps most surprising was that the distribution of data among many nodes, ostensibly the important part of the assignment, was not the most crucial part of the learning curve. Instead, the main issue was figuring out how to deal with external memory issues. Namely, whenever something was written to a file the program relied on the operating system to handle this. Incidentally, this caused the operating system to be out of sync with the unknown amount of data coming into the program and the program crashed. Figuring out how to deal with these operations was the primary obstacle to the successful execution of the program and consequently after several different attempts to fix this problem and anomalous test data (indicating where the program failed on each process) it became clear that this was effectively the limiting factor to running multiple writes at once on this system and OS as configured. That led to the decision that during write operations the system would have to wait for other processes to catch up. Therefore due to these issues with write speed bottlenecks became a part of what it meant to run this type of parallel program, at least with this OS and hardware as configured.

Lastly, reading and writing access times (as well as waiting for communications) seemed to outpace the multiprocessor advantages most of the time for up to 64 processes. Beyond that number there may have been an improvement but the software was no longer deemed reliable enough in its time measurements to include in the experimental results. An overlay of all four speedup figures summarizes this information in order to put it in perspective. See Figure 7-1 on the next page.

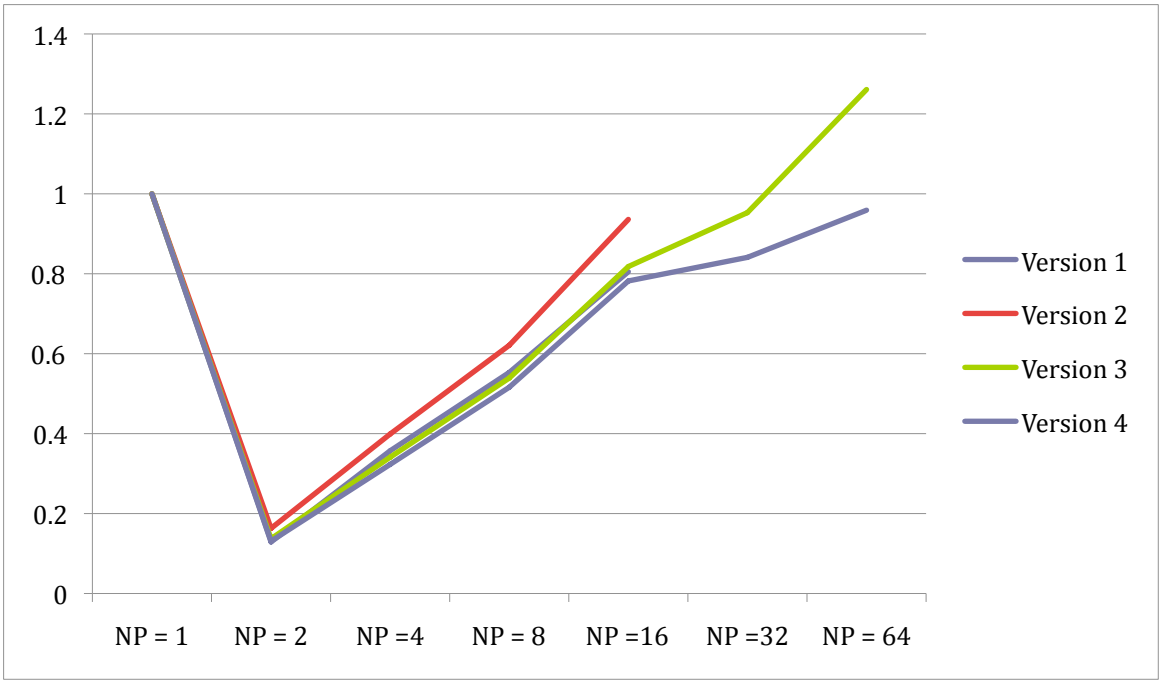

Figure 7-1 This figure summarizes all four prior speedup figures.

References

[1] *Wikelski, M., and Kays, R. 2017, Movebank: archive, analysis and sharing of animal movement data. Hosted by the Max Planck Institute for Ornithology.* www.movebank.org, *accessed on May 5th, 2017.*

[2] Gagliardo, A., Ioalè, P., Filannico, C., Wikelski, M., 2012, Data from: Homing pigeons only navigate in air with intact environmental odours--a test of the olfactory activation hypothesis with GPS data loggers: Movebank Data Repository. doi:10.5441/001/1.q8b02dc5/1 Downloaded from Movebank.org Retreived on March 2017.

[3] Spiegel O, Getz WM, Nathan R (2014) Data from: Factors influencing foraging search efficiency: Why do scarce lappet-faced vultures outperform ubiquitous white-backed vultures? Movebank Data Repository. doi:10.5441/001/1.pr1vj29n Retrieved March 2017.

[4] Barber, David, and Bildstein, Keith. 2017. Turkey Vulture Acopian Center USA GPS. Downloaded from Movebank.org Retrieved April 30, 2017.

[5] Karl Frinkle, Mike Morris, "Developing a Hands-On Course Around Building and Testing High Performance Computing Clusters," Procedia Computer Science, vol. #51 (2015), 1907-1916.

[6] Basem Almadani, Abdallah Rashed “Enforce a Reliable Environment in Parallel Computing Applications,” Procedia Computer Science, vol. 63 (2015), 24-31.

[7] Stafano Markidis, Ivy Bo Peng, Roman Iakymchuk, Erwin Laure, Gokcen Kestor and Roberto Gioiosa, “A Performance Characterization of Streaming Computing on Supercomputers,” Procedia Computer Science, vol. #80 (2016), 98-107.

[8] Ibid.

[9] Jinn-Tsong Tsai, Jia-Cen Fang, Jyh-Horng Chou, “Optimized task scheduling and resource allocation on cloud computing environment using improved differential evolution algorithm,” Computers & Operations Research, vol. #40 (2013), 3045-3055

[10] Gropp, William. Lusk, Ewing. Skjellum, Anthony. Using MPI-2: Advanced Features of the Message Passing Interface. MIT Press, Boston. 1999.

[11] Anonymous Author. 2013. “Read file simultaneously with two different programs”http://www.unix.com/unix-for-dummies-questions-and-answers/216405-read-same-file-simultaneously-two-different-programs.html Retrieved May 7, 2017.

[12] Ibid. 10.

Appendix - Code

Source Code for Version 4 of the Program
Listing 1-1 mp_6r.c

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <string.h>
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <time.h>
#include <sys/sysinfo.h>

#define PI 3.14159265

char filename[50];
char outputFilename[50]="./logs/";
int ierr, num_procs, pid, partitionlength, out2, isfirst1,
isfirst2, writtenproc0;
double XCONST = 0;
double YCONST = 0;
char * SINGLEDIST = "Calculate Distance";
int SINGLEDISTBOOL,DIRBOOL,TTAP,TTAT,TTAV,TTA,DENSITY; // 0  =
UNSET; 1 = SET
int BOOLTV, BOOLVU, BOOLPG;// 0 is default and means run. Turn
off is also available.
char singdist[50] = ""; // blank by default
time_t current_time;
char* c_time_string;
char brokentemp[20][512];//number tokens for pigeons = 16;
vultures = 19; bats = 14;
char *pigeonX;
char *pigeonY;
char *vultX;
char *vultY;
char *tvX;
char *tvY;
double PIGEONSPEED = 50; // 50 mph
double VULTURESPEED = 35; //35 mph
double TVULTURESPEED = 17;
double LATLONDEGINMILES = 69.11;
double BOXX1;
double BOXY1;
double BOXX2;
double BOXY2;
double densityfinal;

struct sysinfo si;
```

```
double distcalc(double x, double y){//returns the distance in
miles
        double result;
        result = sqrt(
                pow(fabs(XCONST-x),2) + pow(fabs(YCONST-y),2)
        );
        result = result * LATLONDEGINMILES;
return result;
}

double ttapigeon(double dist){ //returns total travel time in
hours
        return dist/PIGEONSPEED;
}
double ttavulture(double dist){//returns total travel time in
hours
        return dist/VULTURESPEED;
}

double ttaTV(double dist){//returns total travel time in hours
        return dist/TVULTURESPEED;
}

double direction(double x, double y){//direction in degrees
        return ((180*tan(x/y))/PI);
}

double updatedensity(double x, double y){
    //if the coordinates are in the box or it's edges increment
the point count
    if(
        (
    (BOXX1<=x) && (x<=BOXX2)
                ) && (
    (BOXY1<=y) && (y<=BOXY2)
        )
    ){
        densityfinal++;
    }
    return 0;
}

double printdensity(){
    double density = (BOXX2 - BOXX1)*(BOXY2 -
BOXY1)*(69.11)*(69.11);
    printf("\n entered print density");
    double denr = densityfinal/density; //number per square mile
        printf("\n Density for Proccess %d:  %lf Items Per Square
```

```
Mile \n %lf/%lf (Items/Square
Miles)",pid,denr,densityfinal,density);
    return 0;
}

int printtolog(double dist, double dir, double eta){ //for each
process: print line to output file
        int num6;
        FILE *mainw;
        int rth = 1;
        mainw = fopen(outputFilename, "a");
        if(pid == 0){
                fprintf(mainw, "\n Distance: %lf miles Direction:
%lf ETA: %lf hours",dist,dir,eta); //paste resultant into log
                fclose(mainw);
                while(rth<num_procs){
                        MPI_Send(&num6, 1, MPI_INT, rth, 0,
MPI_COMM_WORLD);
                        rth++;
                }
        }
        if(pid > 0){
                fprintf(mainw, "\n Distance: %lf miles Direction:
%lf ETA: %lf hours",dist,dir,eta);
                fclose(mainw);
                MPI_Recv(&num6, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        }
        return 0;
}

int recordtestrundetails(){
        //Variables: Connection Variables, Index Variables, Time
Variables, Tests Being Run Variables
        FILE *rd; //an index with the test number and the data,
number of processors and tests run recorded.
        FILE *rdcount; //a count of what test number it is
        FILE *rdcountread;
        char *line = NULL;
        size_t len = 0;
        int rind = 0;
        //read test number from index
        rdcountread = fopen("./logs/current.txt","r");
        if(getline(&line, &len, rdcountread) != -1){
                rind = atoi(line);
        }
fclose(rdcountread);

        //update test number
        rdcount = fopen("./logs/current.txt","w");
```

```
        rind = rind + 1;
        fprintf(rdcount, "%d", rind);
        fclose(rdcount);

        //determine test details - so far that's just: pid,
c_time_string,sing
        if(SINGLEDISTBOOL){
                strcpy(singdist,SINGLEDIST);
        }
        current_time = time(NULL);
        c_time_string = ctime(&current_time);

        //append test details
        rd = fopen("./logs/testindex.txt","a");
        fprintf(rd,"#Process %d, Computer Time %s,
%s",pid,c_time_string,singdist);
        fclose(rd);
        return rind;
}

int simplepath(int testnum){
        //Variables:
        int out = 0;
        double distresulttemp, timeresulttemp, directiontemp;
//temporarily holds the resultant of the distance calculation
        FILE *ifp, *ofp;
        size_t len = 0;
        char *mode = "r";
        char *line = NULL;
        char temp[256];
        double densityFinal;
        double densityCh; //

        //create a string version of the filepath with the pid
and directory variables
        char teststr[5];
        char pidstr[11];
        sprintf(pidstr, "proc%d.txt", pid);
        sprintf(teststr,"%d/", testnum);
        strcat(outputFilename,(const char *)teststr);
        strcat(outputFilename,(const char *)pidstr);
        //create test folder
        int num2;
        if(pid == 0){
                char logfiledir[20] = "./logs/";
                strcat(logfiledir,teststr);
                strcat(logfiledir,"/");
                int status = mkdir(logfiledir,S_IRWXU);
                int kth = 1;
                num2 = -1;
```

```
                while(kth<num_procs){
                        MPI_Send(&num2, 1, MPI_INT, kth, 0,
MPI_COMM_WORLD);
                        kth++;
                }
        }
        if(pid > 0) MPI_Recv(&num2, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        //open streams
        ifp = fopen("./data/pigeons.csv", mode);
        ofp = fopen(outputFilename, "w");
        fprintf(ofp, "\n File Header \n FileName: %s \n Test
Number: \n",outputFilename);

        //stream error handlers
        if (ifp == NULL) {
                fprintf(stderr, "Can't open input file!\n");
                exit(1);
        }
        if (ofp == NULL) {
                fprintf(stderr, "Can't open output file %s!\n",
                        outputFilename);
                exit(1);
        }

        fclose(ofp);

        //Most Processing is called from this loop.
        while (
              (getline(&line, &len, ifp) != -1)
        ) {
                if( (out != 0) && ( (out % num_procs) == pid) ){
                        //extract numbers in a loop
                        tokenizePigeons(line);

                        //convert numbers to values in double
                        double xout = atof(pigeonX); //longitude
of pigeons file, x coords
                        double yout = atof(pigeonY); //latitude
of pigeons file, y coords
                        //output debug
                        if((out % 40000) == 0){
                                printf("\n pigeonX %s  |
%lf",pigeonX,xout);
                        }
                        //compute distance
                        distresulttemp = 0;
                        distresulttemp = distcalc(xout,yout);
                        timeresulttemp =
```

```
ttapigeon(distresulttemp);
                        directiontemp = direction(xout,yout);
                        updatedensity(xout,yout);
                        printtolog(distresulttemp,directiontemp,
timeresulttemp);
                }
                out++;

        }
        fclose(ifp);
        FILE *fpvr = fopen("./data/vultures.csv", mode);
                //Most Processing is called from this loop.
        while (
              (getline(&line, &len, fpvr) != -1)
        ) {
                if( (out != 0) && ( (out % num_procs) == pid) ){
                        //extract numbers in a loop
                        tokenizeVultures(line);

                        //convert numbers to values in double
                        double xout = atof(vultX); //longitude of
pigeons file, x coords
                        double yout = atof(vultY); //latitude of
pigeons file, y coords
                        //output debug
                        if((out % 40000) == 0){
                                printf("\n vultX %s  |
%lf",vultX,xout);
                        }
                        //compute distance
                        distresulttemp = 0;
                        distresulttemp = distcalc(xout,yout);
                        timeresulttemp =
ttavulture(distresulttemp);
                        directiontemp = direction(xout,yout);
                        updatedensity(xout,yout);
             printtolog(distresulttemp,directiontemp,
timeresulttemp);
                }
                out++;

        }
        fclose(fpvr);
        FILE *fptvr = fopen("./data/tv1.csv",mode);
    while (
          (getline(&line, &len, fptvr) != -1)
    ) {
                if( (out != 0) && ( (out % num_procs) == pid) ){
                        //extract numbers in a loop
                        tokenizeTV(line);
```

```
                        //convert numbers to values in double
                        double xout = atof(tvX); //longitude of
pigeons file, x coords
                        double yout = atof(tvY); //latitude of
pigeons file, y coords

            //output debug
                        if((out % 40000) == 0){
                                printf("\n tvX %s  |
%lf",tvX,xout);
                        }

            //compute distance
                        distresulttemp = 0;
                        distresulttemp = distcalc(xout,yout);
                        timeresulttemp = ttaTV(distresulttemp);
                        directiontemp = direction(xout,yout);
                        updatedensity(xout,yout);
                        printtolog(distresulttemp,directiontemp,
timeresulttemp);
                }
                out++;

        }
    if(pid == 0) printf("\n %d records & 3 headers read",out);
        printdensity();
    return out;
} // end simple path

int tokenizePigeons(char *line){
        char *savestr = malloc(512 * sizeof(char));
        char *str = malloc(512 * sizeof(char));
        char str2[512];
        strcpy(str, line);
        pigeonX = strtok_r(str, ",",&savestr);
        pigeonX = strtok_r(NULL,",",&savestr);
        pigeonY = strtok_r(NULL,",",&savestr);
        return 0;
}

int tokenizeVultures(char* line){
        char *savestr = malloc(512 * sizeof(char));
        char *str = malloc(512 * sizeof(char));
        char str2[512];
        strcpy(str, line);
        vultX = strtok_r(str, ",",&savestr);
        vultX = strtok_r(NULL,",",&savestr);
        vultX = strtok_r(NULL,",",&savestr);
        vultX = strtok_r(NULL,",",&savestr); //4th
```

```
        vultY = strtok_r(NULL,",",&savestr);
        return 0;
}

int tokenizeTV(char* line){ //Turkey Vultures
        char *savestr = malloc(512 * sizeof(char));
        char *str = malloc(512 * sizeof(char));
        char str2[512];
        strcpy(str, line);
        tvX = strtok_r(str, ",",&savestr);
        tvX = strtok_r(NULL,",",&savestr);
        tvX = strtok_r(NULL,",",&savestr);
        tvX = strtok_r(NULL,",",&savestr); //4th
        tvY = strtok_r(NULL,",",&savestr);
        return 0;
}

int readIndex(){
        int ix;
        FILE *ri;
        char *line = NULL;
        size_t len = 0;
        ri = fopen("./logs/current.txt","r");
        if(getline(&line, &len, ri) != -1){
                ix = atoi(line);
        }
        fclose(ri);
        return ix;
}

int main(int argc, char **argv){
        int index = 0;
        clock_t end, begin;
        double time_spent;
        ierr = MPI_Init(&argc, &argv);
        ierr = MPI_Comm_rank(MPI_COMM_WORLD, &pid);
        ierr = MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
        if(pid == 0) begin = clock();
        sysinfo(&si);
        int number;
        if(argc == 5) BOXX1 = atof(argv[1]);
        if(argc == 5) BOXX2 = atof(argv[2]);
        if(argc == 5) BOXY1 = atof(argv[3]);
        if(argc == 5) BOXY2 = atof(argv[4]);
        pigeonX = malloc(512*sizeof(char));
        pigeonY = malloc(512*sizeof(char));
        vultX = malloc(512*sizeof(char));
        vultY = malloc(512*sizeof(char));
        tvX = malloc(512*sizeof(char));
        tvY = malloc(512*sizeof(char));
```

```
        if (pid == 0) {
                recordtestrundetails();
                number = 0;
                int jth = 1;
                while(jth<num_procs){
                        MPI_Send(&number, 1, MPI_INT, jth, 0,
MPI_COMM_WORLD);
                        jth++;
                }
        }
        if(pid > 0) MPI_Recv(&number, 1, MPI_INT, 0, 0,
MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        index = readIndex();
        printf("Hello world! I'm process %i out of %i
processes\n", pid, num_procs);
        simplepath(index);
        if(pid == 0) end = clock();
        if(pid == 0) time_spent = (double)(end - begin) /
CLOCKS_PER_SEC;
        if(pid == 0) printf("\n Finished Execution after %lf
seconds. Okay to quit program.",time_spent);
        ierr = MPI_Finalize();
        return 0;
}
```