Design and Implementation of a Parallel Pulsar Search Algorithm

by

Tsepo Sadeq Montsi

submitted to the Department of Electrical Engineering,
University of Cape Town,
in partial fulfillment of the requirements for the degree of
Bachelor of Science in Electrical and Computer Engineering

Cape Town, October 2006

Declaration

I declare that dissertation is my own, unaided work. It is being submitted for the degree of Bachelor of Science in Engineering in the University of Cape Town. It has not been submitted before for any degree or examination in any other university.

Signature of Author .....

Cape Town

October 2006

Abstract

This Thesis illustrates the design and implementation of a Parallel Pulsar Search Algorithm. Pulsars are a are rare form of Neutron Star. They are of great interest to astronomers due to their unique properties. Pulsars rotate rapidly and emit beams of radio energy. If the axis of rotation and the direction of the radio beams do not align, then from Earth these beams are perceived as lighthouse like pulses. Due to an interaction with the Interstellar Medium called Dispersion, these beams are generally undetectable from a direct observation. A Pulsar Search Algorithm takes in observation data from a Radio Telescope, and counteracts the effects of Dispersion on this data, it then performs an analysis in the frequency domain to detect possible Pulsars. An algorithm that performs these manipulations is computationally intensive, as such a method of speeding up this computation is desirable. One method of improving speed is running the algorithm simultaneously on multiple computers, i.e. Parallel Computing. After analysing a sequential Pulsar Search Algorithm, a SIMD and MISD parallel Pulsar Search Algorithm were designed. These were implemented in a program and then tested on the KAT cluster to measure their performance. Both proved to be faster than the sequential algorithm, however the MISD implementation proved to be the fastest.

Acknowledgments

I wish to thank my Thesis Supervisor, Professor Mike Inggs for his assistance in the completion of this thesis, but most importantly, for making such a fascinating and challenging topic available at an undergraduate level.

I would also like to thank Thomas Bennett for his enthusiastic assistance in all matters Parallel, and for making the fantastic resource that is is the KAT cluster available to me.

Dr. Duncan Lorimer, the author of the seminal book on Pulsar Astronomy the Handbook of Pulsar Astronomy, assisted greatly in the creation and debugging of the implemented program.

I greatly appreciate the assistance of my fellow final year student, Mr Roger Deane. His research on and in depth knowledge in the fields of Pulsar and Radio Astronomy was instrumental in making the outcome of this project a success.

I would like to thank my Mother, Father and Brother for their support and encouragement.

List of Figures

Nomenclature

Algorithm:	Sequence of steps used to solve a computational problem
FFT:	Fast Fourier Transform
Dispersion:	Smearing of the frequency components of a signal due to interaction with the Interstellar Medium
Dispersion Measure:	The measure of the degree of dispersion of a signal
Interstellar Medium:	Charged dust and plasma, inhabiting interstellar space
MPI:	Message Passing Interface, A specification for libraries implementing common Parallel Computing Functions
Parallel Computer:	A group of computers computing a single algorithm
Pulsar:	Rotating Neutron star, emitting beams of radio energy, which are perceived to be pulsing

Table of Contents

1 Introduction

1.1 Project Background

Parallel Computing involves harnessing the collective computing power of multiple computers in order to increase the rate at which data can be processed[7]. It is a field that has revolutionised many activities and fields of study[7], by removing the shackles imposed by the computational limitations of traditional computers. One field that has benefited greatly from the use of Parallel Computers is Astronomy. The size of the observational data collected by sensitive modern astronomical apparatus is extremely large when compared to other computational applications, coupled with this is the fact that extremely computationally intensive operations are often required before any meaningful information can be gleaned from the data. A perfect example of this is searching for Pulsars.

Pulsars are astronomical objects that are of great interest to astronomers due to their unique characteristics. They are collapsed stars, known as Neutron Stars, that rotate rapidly[1]. Due to the physical processes occurring in and around them, they emit signals in the radio frequencies, which because of their rotation, are perceived as lighthouse like pulses[1]. Due to various interstellar phenomena, these signals are generally undetectable from a direct observation, and require manipulation before the presence of a pulsar can be confirmed. An algorithm that performs these manipulations is computationally intensive, as such a method of speeding up this computation is desirable. One method of improving speed is running the algorithm simultaneously on multiple computers, i.e. Parallel Computing.

1.2 Project Objectives

The author has been tasked with the creation of a Parallel Pulsar Search Algorithm to increase the speed of Pulsar searches. This algorithm should be implemented in a program able to be run on a parallel computer, in the case of this project, a beowulf type cluster. The program should be based on an existing sequential Pulsar Search Algorithm. The program should take a file containing observation data from a radio telescope as its input. The data from the file should be distributed to the computers in

the cluster and processed by them. This processing should compensate for dispersion over a wide range of Dispersion Measures. Once de-dispersed and other manipulations of the data have been completed, the program should be able detect possible Pulsar signals in the processed search data. This process should occur faster than with an equivalent sequential algorithm.

In order to quantify the improvement in computational speed, the program should be profiled to determine the computation time and other parameters that may provide insight into the relative merits of the parallel algorithm over the sequential.

1.3 Scope and Limitations

The Author was tasked with the implementation of a parallel algorithm from an existing sequential algorithm and getting it to run on a cluster. The implemented sequential algorithm, outside of the parallel extensions, while being a collaborative effort, was not wholly implemented by the author, this has been indicated in the chapter describing the implementation and in the source code. Theoretical design of the sequential algorithm was largely the work of Roger Deane, a fellow student. While the focus of the designs depicted in this document were speed, correctness of the algorithm was considered to be equally important. As part of the research required for the implementation, background information on Parallel Computing, Pulsars and Radio Astronomy was referred to and has been included in this document. While some of this information may not be wholly relevant to the actual implementation, it was felt that its inclusion would be advantageous with regard to justifying choices made in the design and implementation of the project.

1.4 Plan of Development

Following this introduction is a review of literature relevant to the completion of the project at hand. After this are chapters describing the theoretical bases and design of the algorithms, the first for the sequential Pulsar Search Algorithm and the second for the parallel extensions to this, following these is a chapter describing the actual implementation. The subsequent chapter describes the testing methodology along with the results obtained. An analysis of the results of testing follows the chapter on testing, and after this analysis, is a conclusion combined with a contemplation of possible future work that could be based on the work carried out in the fulfillment of the requirement. This chapter will continue with a brief summary of each of the chapters mentioned above.

Chapter 2: Literature Review

This chapter gives an overview of the material referenced in this document, as well as other literature thought to be relevant. An in depth analysis of the information gleaned from the literature is performed in the background section preceding each of the design chapters.

Chapter 3: Pulsar Search Algorithm Design

This chapter begins with a description of Pulsars, they are shown to be remnants of stars, that have collapsed in on themselves to form rapidly rotating and highly magnetic Neutron Stars[1]. The scientific implications of this are briefly discussed as a justification for searching for them. The source of their radio signals is briefly described, along with the mechanisms of dispersion. The next section deals with the components of a basic Pulsar Search Algorithm; De-dispersion, FFT and a Frequency Domain Search. This chapter concludes with an analysis of the computational requirements of the algorithm.

Chapter 4: Parallel Algorithm Design

In this chapter the the theory of parallel computing is described as well as the four categories of algorithms, Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data(MIMD). The first being the sequential model and the others parallel[7][8].

After analysis of the sequential algorithm, two methods of parallelisation are chosen and described, a SIMD approach, where the channels of the source data are split and distributed equitably to the nodes for dedispersion, and a MISD approach where each node performs the entire sequential algorithm for a group of dispersion measures. An analysis of the computational requirements of each algorithm is given, along with a prediction of their performance. This analysis indicates that the MISD approach is likely to prove the most effective.

Chapter 5: Implementation

This chapter begins with a description of the "apparatus" used, i.e. software libraries, cluster specifications and input file formats. Following this is a section that describes the programming methodologies used, and a description of the translation of the designs into code. Compromises made on the design and unexpected problems with the implementation are also described in this chapter.

Chapter 6: Testing

This chapter illustrates the testing methodologies used in analysing the implemented parallel algorithms. It begins with a comparison of the output of the sequential algorithm with that of the parallel algorithms, all with identical inputs. A description of the comparison criteria is then given, these being completion time, percentage speedup and communications efficiency. Graphs displaying performance the each of the algorithms with regard to the aforementioned criteria are then shown.

Chapter 7: Analysis of results

An in depth analysis of the results obtained in the testing phase is performed in this chapter. It is shown that all algorithms performed the processing on the data file identically, and can be considered to be functional Pulsar Search Algorithms. The performance results of the MISD implementation are shown to correspond to the results predicted in the design. The results of the SIMD implementation proved anomalous, not achieving their theoretical potential, however, it still proves to be faster than the sequential algorithm, also it is shown that this design does have unique benefits resulting from efficient memory usage. The possible reasons for the results observed are given.

Chapter 8: Conclusion and Future Work

This chapter begins with a discussion of the process followed in completing the project, from design to implementation. It is shown that the designs were somewhat naive. The results and their analysis are assessed to ascertain if the objectives of the project have been achieved and to what degree. After this are descriptions of possible enhancements to the implemented designs. Following this is a section describing improvements to the implementations, and other possible future work that could be based on the work carried out for this project.

2 Literature Review

The field of Parallel Pulsar Search Algorithms is not as documented as one would have expected. Information available is generally targeted at GRID computing (computing nodes located at different geographical locations, generally different academic institutions). The information that is available is extremely general with no description of the algorithm, it is therefore outside of the scope of this thesis.

Information on parallel algorithms in general and parallel algorithms implemented for other applications is readily available, the same is true for general pulsar search algorithms, although to a lesser extent.

The first resource consulted was the Internet, specifically general reference sites, in order to gain familiarity with the subjects handled in this thesis. Wikipedia, the online collaborative encyclopedia proved to be a great asset, as it gave a very good overview of most of the subjects dealt with in this document. The Internet is also a good source of Dissertations and Theses relating to the subjects at hand. The most exciting resource available online was the new Google Books service, where the entire scanned collections of many major libraries are made available. Material which is still under copyright has limited access, and one is only able to view a few pages at a time. Tutorials provided by academic institutions and government laboratories also proved helpful with respect to the implementation of the parallel algorithms.

One of the most effective resources were the Masters dissertations from previous students at the University of Cape Town. They helped immensely especially with the decisions made on the structure of this thesis. They also proved to be a good source of other reference material.

Below are descriptions of the most helpful reference sources, they are also the most cited sources in this thesis.

2.1 Pulsar Search Literature

The online encyclopedia, Wikipedia, was a good starting off point with regards to general information on Pulsars.

The book Handbook of Pulsar Astronomy[2] is probably the only point of reference one needs when creating a Pulsar Search Algorithm. It portions of it were available on Google books, and a summarized excerpt of the most pertinent parts was also made available by Roger Deane. This resource proved to be great help in the understanding of the problems and solutions to searching for Pulsars.

The writeup of a lecture given by Robert Hulse entitled, The Discovery of The Binary Pulsar[9] provides a well written, step by step examination of the process of Searching for Pulsars as well as containing fascinating and often witty anecdotes with regards to the many potential problems the process can throw up. This document had several excellent figures illustrating various concepts relating to Pulsars, two of these figures are used in this thesis (**Figure 1.1** and **Figure 1.2**).

2.2 Parallel Computing Literature

The book High Performance Cluster Computing by Rajkumar Buyya[7] was the first reference consulted with regards to Parallel Computing. It is an extremely thorough source of information and contains sections regarding everything from taxonomy to actual implementation. It includes many practical notes such as the overhead associated with different middleware utilities.

The Design and Analysis of Parallel Algorithms, a book by G. Akl[8], was a good source of information on the various strategies for splitting up a sequential algorithm into parallelizable components.

The Sourcebook of Parallel Computing by J. Dongarra et al[6], was available on Google books and helped greatly in clarifying some of the basic concepts of Parallel Computing. It had a good section on the taxonomy and classification of parallel Computers.

The Masters dissertations of T.G.H Bennet[3] , S. Mukherjee[5] and O. Fadiran[10] proved to be great sources of information with regard to the design and implementation of parallel algorithms.

2.2 Parallel Pulsar Search Literature

Within the scope of this thesis, not much information is available on Parallel Pulsar Search algorithms. Most institutions that have Pulsar Search programs seem to have implemented Parallel Computers to perform searches although specifics of the algorithms are not mentioned. An example of an implemented Parallel Computer used for Pulsar Searching is COBRA at the University of Manchester[24]. Most of the focus on parallel Pulsar Search Algorithms appears to be focused on GRID computing[25]. Information about these projects is also lacking.

The most relevant information that could be found was on the AIPS++ package. AIPS++ stands for Astronomical Information Processing System[27]. It is a software package that implements commonly used radio astronomy data processing functions. It defines data structures and functions that are of use to in the calculations carried out in the analysis of data from a radio telescope observation. AIPS++ has been expanded to include parallel functionality[28].

3 Pulsar Search Algorithm Design

This chapter begins with a description of Pulsars, they are shown to be remnants of stars, that have collapsed in on themselves to form rapidly rotating and highly magnetic Neutron Stars[1]. The scientific implications of this are briefly discussed as a justification for searching for them. The source of their radio signals is briefly described, along with the reasons for dispersion. The next section deals with the components of a basic Pulsar Search Algorithm, De-dispersion, FFT, Frequency Domain Search. This chapter concludes with an analysis of the computational requirements of the algorithm.

3.1 Background to Pulsars

Stars rely on the fusion of hydrogen as their source of energy. The radiant energy generated by this fusion counteracts the gravitational force created by the huge amount of matter present in a star. When a large star (one with a mass greater than two or three times that of the sun) exhausts its hydrogen fuel, the gravitational forces, now without any opposition, force the star to collapse in on itself[1]. The intense and now concentrated gravitational forces create a super-dense sphere about 20 km in diameter, known as a neutron star[1].

Because the angular momentum of the star is conserved and the moment of inertia is significantly decreased, the rotation of the neutron star is much greater than that of its parent star[1]. This rotation and the strong magnetic forces present in the pulsar result in the emission of beams of radio at the pulsar's magnetic poles[1]. The magnetic poles of some pulsars are not in alignment with the rotational poles, as a result these radio beams are not necessarily co-linear with the axis of rotation. If the pulsar is correctly orientated, the radio beams point towards the Earth periodically during the rotation of the Pulsar, resulting in a lighthouse effect, which we can perceive as periodic pulses of radio[1].

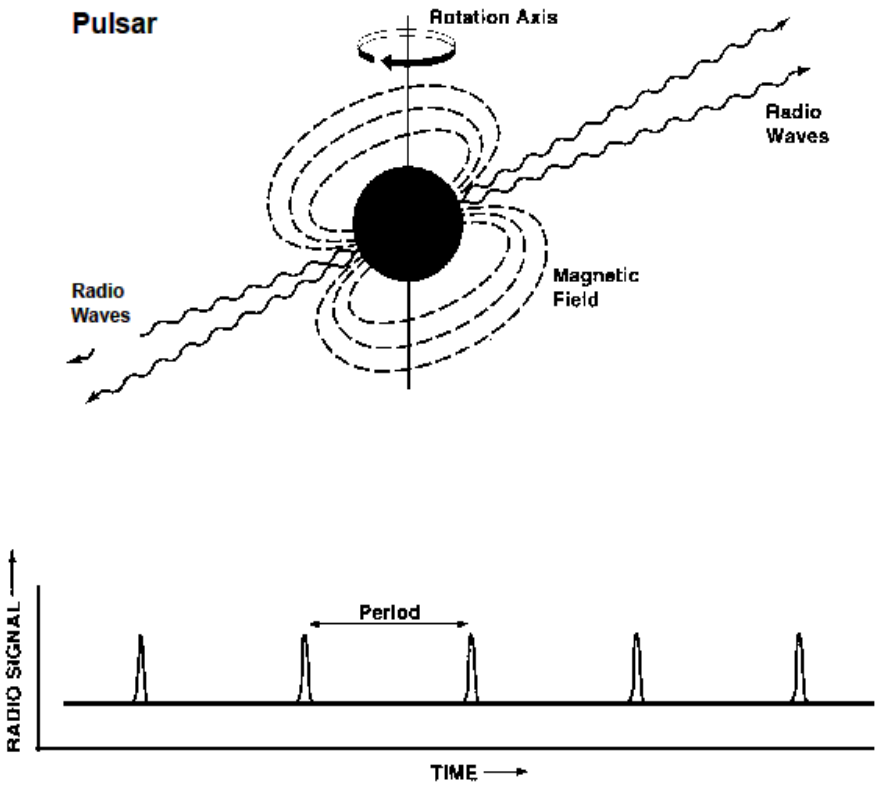

Figure 3.1: Illustration of a Pulsar and the associated periodic signal

(Taken from R Hulse, The Discovery of the Binary Pulsar[9])

Interstellar space is not a true vacuum, and can be considered to be a sea of charged particles. Different frequency components of a pulsar's radio pulse interact with this sea of particles differently, resulting in different velocities of propagation for different frequency components[2] . The observed signal is therefore "smeared" in the frequency and time domains, resulting in a signal that is superficially indistinguishable from the background noise. This smearing is called dispersion[2]. As dispersion is due to the passage of the signal through the interstellar medium, the amount of dispersion that a signal suffers is proportional to the distance it has traveled. The amount of Dispersion experienced by a Pulsar's signal is known as the Dispersion Measure. The Dispersion of a pulsar's signal is unknown, therefore a Pulsar Search Algorithm should iterate through a range of Dispersion Measures[2].

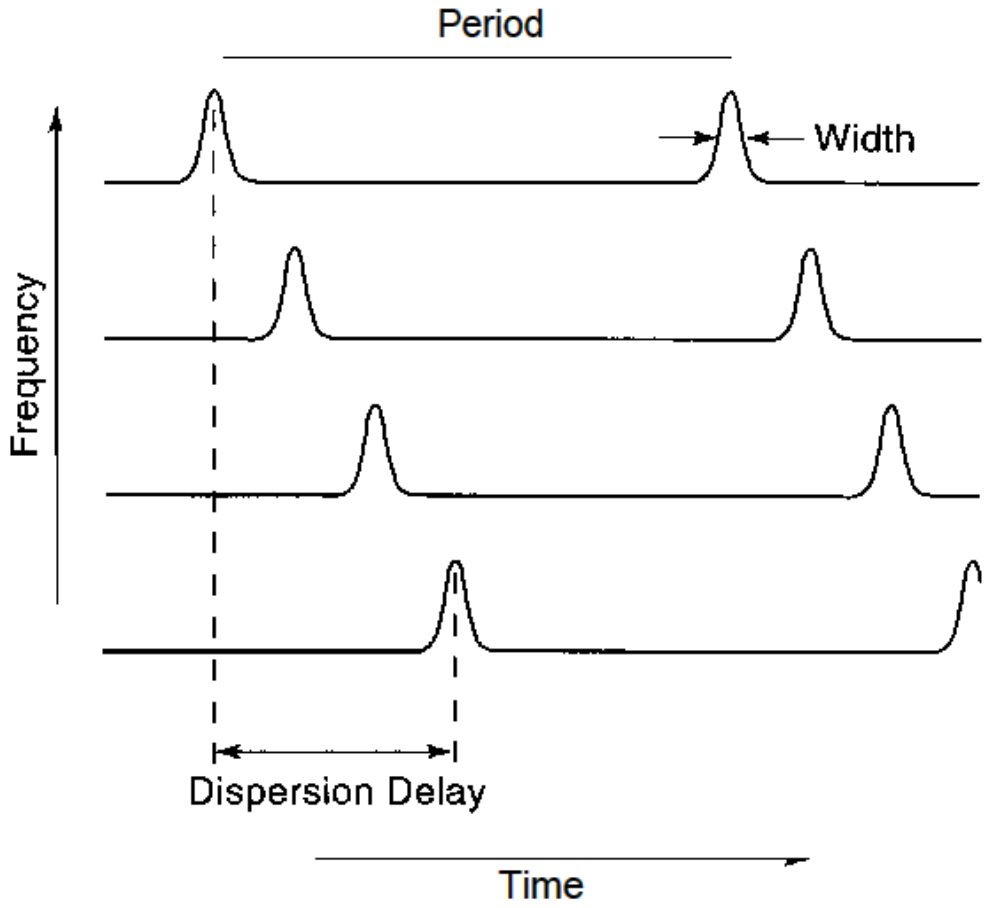

Figure 3.2: The effects of dispersion on a Pulsar's signal

(Taken from R Hulse, The Discovery of the Binary Pulsar[9])

To detect a Pulsar's signal in the data collected by a radio telescope, a dedispersion operation must be applied to the signal data to counteract the effects of dispersion, this process is described in section 3.2. Once this compensation has been made, the signal's frequency spectrum is analyzed using a Fast Fourier Transform (FFT). The FFT is an efficient digital implementation of the Fourier Transform, it decomposes a time domain signal into a set of sinusoidal bases, which represent the different frequency components in the signal. The frequency spectrum is analyzed to see if any periodic signals indicative of a pulsar are present. This is done by analyzing the frequency components of the signal , looking for frequency components which have a larger amplitude than that of the background noise, and also looking at the characteristics of the periodic signal to ascertain that they are not of terrestrial or non-

pulsar origins. This process is described in section 3.3.

Once possible pulsars signals are discovered, the signals are folded, which basically means that, a few periods of the each potential signal are added and averaged, creating an accurate definition of the characteristics of a single period of the possible Pulsar.

The operations described above are computationally expensive, this is exacerbated by the large amount of data that must be sifted through. This can be remedied by increasing the speed of the computer that the algorithm is running on. This has limitations in that after a certain threshold, increasing the speed of a computer is subject to the law of diminishing returns, in that cost of increasing the speed of the computer(in terms of both financial and complexity), begins to exceed the benefit gained from the speed increase. There are also finite limits on the performance capabilities of a computer.

A possible way to compensate for this would be to run the algorithm in parallel on multiple machines.

The following design was primarily based on the research carried out by Roger Deane.

3.2 De-Dispersion

De-dispersion operates on the raw data file by shifting each channel by an amount that compensates for the delay experienced by that channel's frequency. The delay for a given frequency is given by the equation:

$$S = \frac{\left(4.148741601 \times 10^{6}\right)}{T} \times DM \times \left(\frac{1}{F_0^2} - \frac{1}{F_i^2}\right)$$

Equation 3.1: Dispersion delay of channel F_i

Equation 3.1 gives the shift S in terms of number of samples shifted back in time to compensate for the dispersion delay at channel F_i. The value 4.178741601×10^6 known is as the the dispersion constant[2], DM is the Dispersion Measure, T is the sampling interval of the observation, F_0 is the frequency of the first channel and F_i is the frequency of the ith channel below the first, ie. the channel to be shifted. F_0 is always the highest frequency, and as can be seen in the equation never experiences a shift, all other channels are therefore shifted in relation to channel F_0.

The range of Dispersion Measures to search is governed by the location of the observed point with respect to Galactic Longitude[2]. Galactic Longitude being the angular measure of the observed point relative to the Galactic Equator[2]. Density of the Interstellar Medium, within our Galaxy, is greatest near the Galactic Equator i.e. for low Galactic Longitude, and decreases as Galactic Longitude increases. As the Interstellar Medium is the main contributor to dispersion, it follows that dispersion increases with a decrease in Galactic Longitude[2]. Therefore observations near the Galactic Equator will search a range of large Dispersion Measures, whereas observations that are far from the Galactic Equator will search a range of relatively small Dispersion Measures[2].

The value by which the Dispersion Measure is incremented when traversing a range of Dispersion Measures is given in Equation 3.2.

$$DM_{step} = \frac{t_{samp}}{4.148741601\text{x}10^{6}} \times \left(\frac{1}{F_0^2} - \frac{1}{F_i^2} \right)^{-1}$$

Equation 3.2: Dispersion Step Size

As can be seen from equation 3.1, the only knowledge needed to perform a de-dispersion shift is the dispersion constant, T, DM, F_0, F_i. This has profound implications for a parallel implementation of a Pulsar Search Algorithm, as it means that any node can perform a de-dispersion operation on any channel, provided it has of the aforementioned values, which can all be assigned at the start of operation as they are constant for the duration of the algorithm.

Once this operation has been performed the data can now be considered to have its time series aligned[2], at least in the context of the dispersion measure used, therefore each sample within each channel corresponding to an instance of time is added to sample at each time instance instance of all the other channels, resulting in a vector that represents the de-dispersed time series of the observation over the entire bandwidth of the observation. The de-dispersion operation is illustrated in **Figure 3.3**.

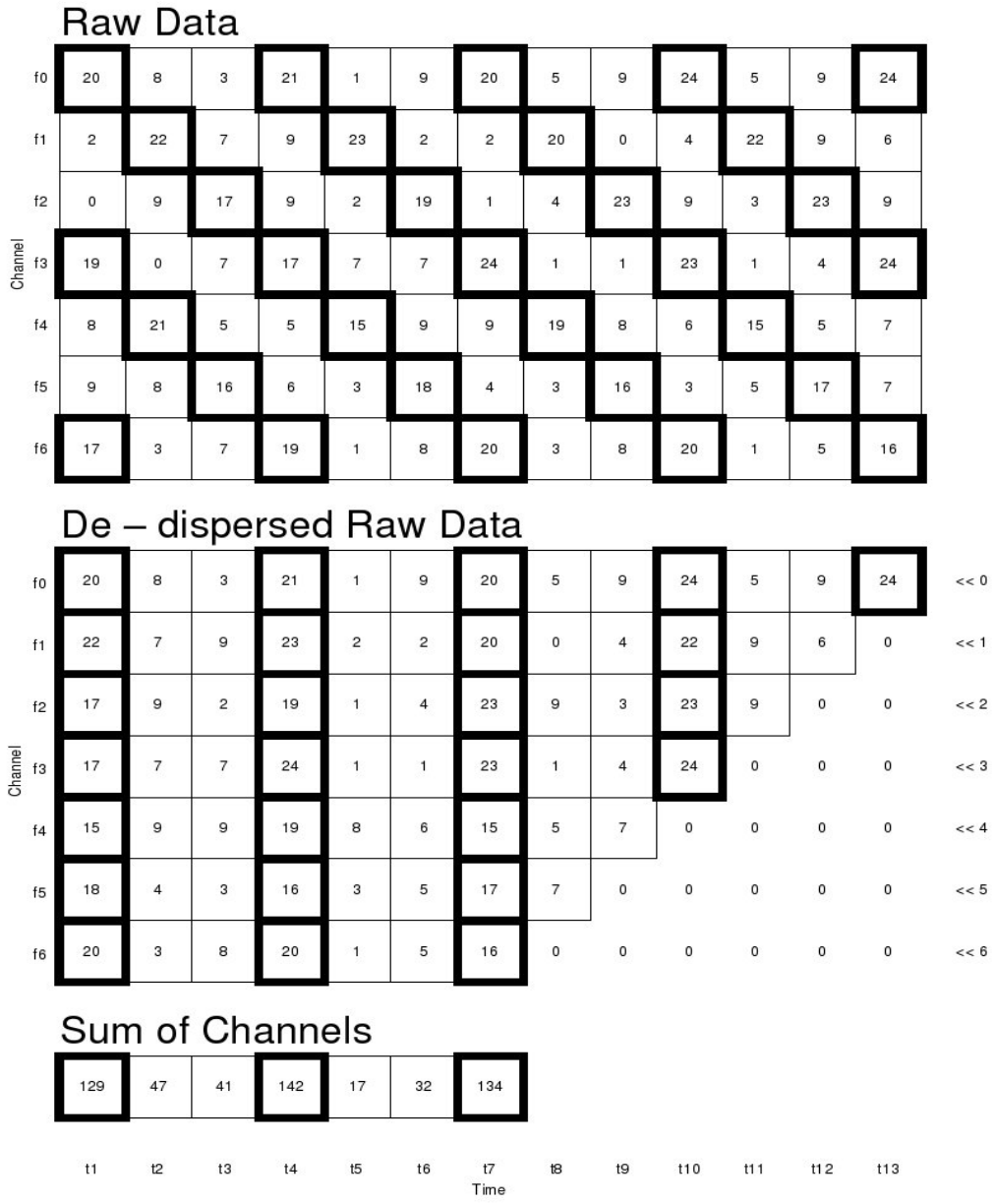

Figure 3.3: Graphical Example of the De-dispersion Operation

Rows represent time series, bold blocks indicate presence of a pulse peak from a pulsar

3.3 Frequency Domain Processing

Pulsar signals are periodic, therefore if their period of rotation is sufficiently regular, which they generally are[1] then if one analyses their signals in the frequency domain, the majority of their signals power will lie at the frequency that most contributes to their period of rotation[11].

To convert a Pulsar's de-dispersed time series into the Frequency Domain, a Fourier Transform must be applied to the data. Because the signal is sampled and represented digitally, a Discrete Fourier Transform (DFT) is the appropriate transform to use[18], in this case a computationally efficient variant called a Fast Fourier Transform (FFT).

The output of an FFT is a complex vector, representing the amplitudes and phase of the various frequency components that combine to form a signal[11]. As the energy in each frequency component is shared over the real and imaginary parts of the complex vector, a more useful method of analysing the data is to convert the output of the FFT into a Power Spectral Density, or Power Spectrum. This involves squaring each real and imaginary term of the FFT output and adding the results for each frequency component. This gives a vector that represents the power present in each frequency component[2][16].

Data collected from a radio telescope contains noise from the many astrophysical radio sources and also from the thermal noise of telescope components[17]. This noise is generally not white noise[2], which has a flat Power Spectrum[17], its shape is characterised by higher amplitudes in the low frequencies, tapering off as the frequency increases, this type of noise is termed Brown or Red noise. If a Pulsar's signal is present in the signal and an amplitude threshold detection scheme is used to identify it, Brown noise in the low frequencies may have a higher amplitude than the frequency component generated by the Pulsar, hiding the Pulsar's signal from the detection algorithm. An operation called whitening is performed on the Power Spectrum to counteract the effects Brown noise[2]. This entails splitting the Power Spectrum into segments, calculating the median value of each segment, subtracting this median from each component of the segment. This decreases the mean for each segment, and the entire power

spectrum, to zero, counteracting the tapering shape of brown noise[11]. In addition to this functions, the Whiten operation also divides the whitened Power Spectrum by its root mean square[11], this, combined with the subtraction of the mean has the effect of normalising the Power Spectrum, this implies that the amplitude of each frequency component in the normalised Power Spectrum represents the signal to noise ratio of that component[2].

A pure sinusoidal signal will give a single frequency component at its frequency of oscillation[11], this frequency is termed the fundamental frequency. However any periodic signal deviating from the purely sinusoidal form in terms of its duty cycle[2], will have a frequency component at the fundamental, and will tend to have additional frequency components at integer multiples of the fundamental[2], these additional frequencies are termed harmonics[11]. Therefore if a Pulsar's signal is not truly sinusoidal, the power in its frequency components will be spread amongst the fundamental component as well as the harmonic components. An operation called Harmonic Summing, iterates through the frequencies of the Power Spectrum, and at each frequency adds the components at integer multiples of that frequency[2]. Once Harmonic Summing has been performed on a Power Spectrum, each frequency component will have the power of its own component in addition to the power present in its harmonics[2].

3.4 Candidate Selection

The ultimate aim of the above operations is to put the observation data into a form from which the presence of a Pulsar can be most easily detected. The data now represents a vector of Frequency Components, with the amplitude of each component being the Signal to Noise ratio of that component. To detect possible Pulsars or candidates, a threshold signal to noise ratio can now be selected. This threshold can be arbitrary, so long as it is set at a level that eliminates spurious peaks in amplitude due to noise and other unwanted sources. For simplicity's sake, the highest amplitude in the Power Spectrum can be selected as the candidate. Dedispersion and the Frequency Domain Processing each occur once for every Dispersion Measure tested. Therefore a list of candidate Pulsar's will be generated by each Dispersion Measure tested.

A Pulsar whose signal has been dispersed by a Dispersion Measure M will have progressively more of its unsmeared signal recovered by de-dispersing with Dispersion Measures progressively closer to M. This is because the closer the Dispersion Measure gets to M, the more the peaks and troughs of the Pulsar's signal align. This further implies that the Signal to Noise ratio of a Pulsar 's signal will also increase, as the Dispersion Measure gets closer to M. The main implication of this is that if a frequency of a single candidate Pulsar is selected, and the signal to noise ratio is plotted against the Dispersion Measure used, then for that frequency selected, the most probable Dispersion Measure by which it has been dispersed by will be the one at which the Signal to Noise ratio is maximised. This method can be used to generate a final list of Pulsar candidates.

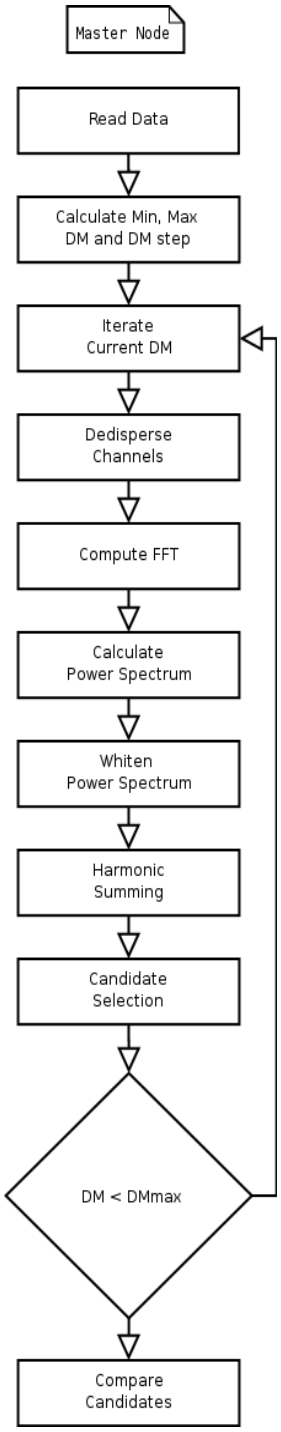

Figure3.5: Sequential Pulsar Search Algorithm

3.5 Computational Analysis of the Sequential Algorithm

To gain insight into the ability of the sequential algorithm to be parallelised, it is advantageous to analyse the computational requirements of each step of the algorithm. The computational time of each step will be measured in the number of operations required to complete the task. This analysis makes the assumption that all operations require the same amount of processing time. Calculations of this kind, termed Big O[20] calculations, generally compute the order of magnitude of the computational requirements rather than the actual than the actual requirements, therefore integer factors are generally excluded[20], however for illustrative purposes they have been retained.
Assuming that the input file contains C channels, with S samples per channel, and our range of Dispersion Measures contains M dispersion Measures:

Read Data: Reading in the file will require at least C*S operations. Reading the data in from a hard drive is typically very slow when compared to the speed at which a CPU operates, therefore C*S is an optimistic estimate. It must be noted that Pulsar search data will typically contain 128 channels with over 1 million samples per channel.

Calculate DM values: This step requires the computation of 3 different values, and so can be considered to take 0 operations when compared with reading the data.

Iterate Current DM: The same applies as above. This step can be considered to take 0 operations.

De-disperse Channels: This step requires the shifting of each sample in each channel, therefore it will take around C*S operations. It also requires the addition of all the samples into a single time series, another C*S operations. Therefore de-dispersion requires 2C*S operations.

Compute FFT: An FFT takes N*log(N)[19], operations where N is the length of the input vector. The output of the de-dispersion operation, the de-dispersed time series, is at most S samples long. Therefore this step should take S*log(S) operations.

Calculate Power Spectrum: The power spectrum requires the squaring of each value of the FFT, therefore it takes S operations as the output length of an FFT is the same as its input, S. The length of the Power Spectrum is S/2.

Whitening: This stage is computes the mean of each element in the Power Spectrum and subtracts each element by the mean. Assuming calculation of the Median requires S/2 calculations and subtraction the same, this step should take S operations. It also computes and the RMS value of the Power Spectrum, which should take S/2 operations, then subtracts the RMS from each value, again S/2 operations, resulting in S operations. The final outcome for whitening is therefore 2S operations.

Harmonic Summing: Harmonic Summing requires traversing the length of the Power Spectrum and adding the components at integer multiples of each frequency component, to itself. If we assume that we add only the first 16 harmonics, Harmonic Summing has around 8S operations.

Candidate Selection: This step requires traversal of the the entire Power Spectrum, making a comparison of each element with a threshold value, therefore this step has S/2 operations.

Compare Current DM with DM max: This is one comparison and will therefore be considered to take 0 operations.

Compare Candidates: This operation compares the candidates generated by Candidate Selection, over the entire range of Dispersion measures. Assuming one candidate generated per Dispersion Measure tested, then this step requires M operations.

The steps from Iteration of the Dispersion Measure through to the Dispersion Measure comparison occur once for each Dispersion Measure in the range, therefore the sequential algorithm takes:

$$C*S + 0 + M*(0 + C*S + S*\log(S) + S + S + 2S + 8S + S/2) + M$$

$$= C*S + M + M*(C*S + S*\log(S) + 12.5S) \text{ operations}$$

4 Parallel Algorithm Design

In this chapter the the theory of parallel computing is described as well as the four categories of algorithms, Single Instruction Single Data (SISD), Single Instruction Multiple Data (SIMD), Multiple Instruction Single Data (MISD) and Multiple Instruction Multiple Data(MIMD). The first being the sequential model and the others parallel.

After analysis of the sequential algorithm, two methods of parallelisation chosen and described, a SIMD approach, where the channels of the source data are split and distributed to the nodes for dedispersion, and a MISD approach where each node performs the entire sequential algorithm for a group of dispersion measures. An analysis of the computational requirements of each algorithm is given, along with a prediction of their performance. This analysis indicates that the MISD approach is likely to prove the most effective.

4.1 Background to Parallel Computing

The traditional model of computing has consisted of a single processor, with dedicated memory (in which instructions and data are stored) and input/output devices, the so called "Von Neumann" design[12]. Computers of this type execute instructions sequentially, i.e. one instruction travels from memory along an instruction path to a processing unit, the processing unit interprets the instruction, it then retrieves any data required by the instruction from memory via the data path, the operation specified by the instruction is performed on the data, with the output being the result of the operation on the data, this output may then be placed into memory (Note that the data path and the instruction path may share a single physical connection), the next instruction is then read and is handled in the same manner[13]. Since their invention, computers of this design have increased exponentially in performance, with the price decreasing in a similar manner[14], allowing more and more applications to benefit from better and better information processing.

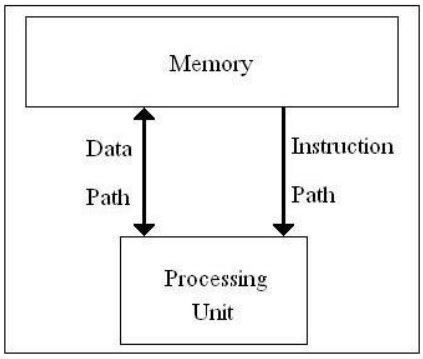

Figure 4.1: Sequential Computer Architecture

This configuration is not without limitations, specifically the processor, its ancillary components, and the connections between them are limited by the laws of physics in their speed and efficiency[21]. These limits result in a law of diminishing returns which affects any effort to increase a computer's performance[14]. This law manifests itself by requiring a progressively larger investment of design time (and money) for each incremental increase in performance, and after a threshold, no further performance increase is possible.

The above mentioned limits are reached with many modern scientific computing applications. These applications are characterised by requiring many complicated calculations on a massive scale (Pulsar Searching being an example), such that even the fastest possible traditional computers are rendered unsuitable to running them.

An attempt to overcome the limitations of the traditional model has been attempted in the form of Parallel Computing. Whereas a traditional computer processes instructions sequentially, a parallel computer runs instructions concurrently. This requires that there are as many data paths, instruction paths and processing units as the desired number of concurrently processed instructions[13].

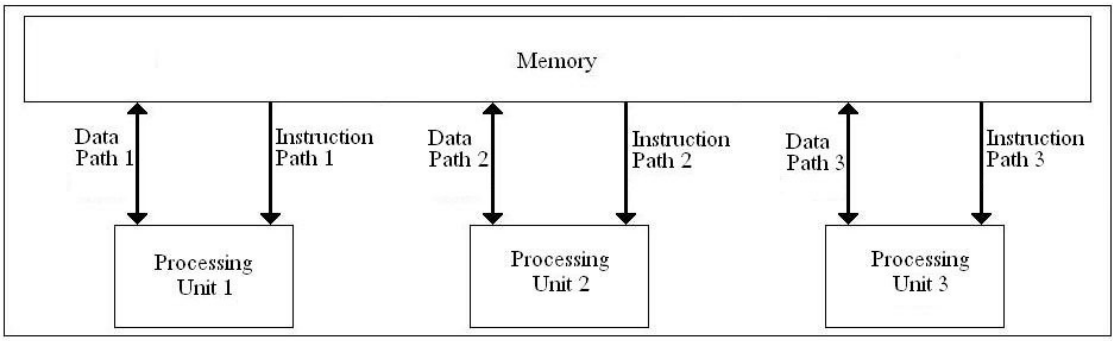

Figure 4.2: Parallel Computer Architecture

(allowing concurrent execution of 3 instructions)

Note that in diagram 4.2 the processing units share memory, this is not always the case[13].

4.2 Parallel Computing Theory

Most algorithms have portions that are inherently sequential, and parts that can be executed in parallel. The sequential parts, unlike the parallel parts cannot be shared among multiple processing units and therefore do not benefit from parallelization. A formula called Amdahl's law, named after the computer architect Gene Amdahl, gives the potential speedup of a parallel computer in relation to the sequential and parallel parts of the algorithm[3].

$$S \leq \frac{1}{f + \frac{1-f}{N}}$$

Equation 4.1: Amdahl's Law

S refers to the speedup factor, the amount of times that the parallel program is faster than the sequential algorithm, f is the fraction of the algorithm that is inherently sequential. N refers to the number of parallel processing units.

If one takes the sequential percentage of an algorithm to be constant, then this law represents an example of the law of diminishing returns, as increasing the number of processing units to increase the speed of computation is limited to a maximum threshold[4], for example, if one takes the inherently sequential percentage of an algorithm to be 20%, then using equation 1.1, the maximum speedup possible in the parallel computer is achieved when $N \to \infty$ [3], which gives a maximum theoretical speedup factor of 5.

This law is simplistic and represents the ideal case because in a real parallel computer, communication (such as assigning data to nodes and scheduling the nodes), housekeeping such as formating of data and diagnostic operations are sources of overhead that are not represented by Amdahl's law.

Parallel Computing places certain requirements on the algorithm to be parallelised, firstly the

algorithm must be able to be split into individually executable components able to be executed on separate processing units. Another requirement is that the prerequisites for execution of each of the selected components are known before execution, so that the parallel algorithm can be efficiently scheduled[7]. If the prerequisites of the components are not known, or are not considered when creating the algorithm, situations such as deadlock can occur. An example of deadlock is if two processes, process A and process B are running in parallel, and process A requires an output from process B before it can continue, but process B similarly requires an output from process A before it can continue, obviously all execution will stop while the processes wait for each other.

4.3 Classification of Parallel Computers

Parallel Computers consist of a number of processing units, connected to each other in a manner that allows the individual units to collaborate in the processing of an algorithm. These processing units are known as nodes. The manner in which the nodes are arranged and connected are used to classify them.

Parallel Computers are implemented in a variety of sizes and flavours. Large particle physics experiments which result in huge volumes of data (typically many gigabytes or terabytes) require computational resources and academic skills that no one institute can provide, therefore they are split amongst the best computers on offer at the various institutions (which themselves may be Parallel Computers), each connected by high speed Internet connections. This paradigm of Parallel Computing is termed Grid Computing.

Some problems can be tackled by more pedestrian computational resources, although a traditional sequential computer will still take too much time to process them. These problems have traditionally been tackled by monolithic so-called supercomputers whose software and hardware are provided by a single vendor. These are extremely expensive when comparing the increase in price with the increase in performance. They generally use arrays of cutting edge CPUs with high speed proprietary interconnection networks that depend heavily on the vendor for implementation and maintenance. The vendor may also not have the skill set required in dealing with the problem at hand and provide one size fits all solutions with redundant and unnecessary features.

The trend in modern Parallel Computing tends to rely on relatively inexpensive networks of standard Personal Computers or Commercial Servers to provide the parallel nodes. These Parallel Computers, especially with regards to those constructed with Personal Computers, generally have open-source or commodity tools and utilities to provide the synchronization and communication requirements. This method of Parallel Computing is referred to generally as clustering, with the most predominant model called the Beowulf Cluster. This is the method that is used in the implementation of this project.

The most popular method of classifying Parallel Computers is called Flynn's Taxonomy. This

classification scheme is based on the concept that computers conceptually have one instruction stream, and one data stream. Parallel Computers can be considered to extend these to create multiple pathways for instructions and/or data[6] thereby increasing the throughput of the system. The categories of classification are:

Single instruction/single data stream (SISD) - A traditional sequential computer[7].

Multiple instruction/single data stream (MISD) - Multiple processors simultaneously executing different instructions on the same data [7].

Single instruction/multiple data streams (SIMD) - Multiple processors simultaneously executing the same instructions on different data[7].

Multiple instruction/multiple data streams (MIMD) - Multiple processors simultaneously executing different instructions on different data[7].

These categories can be further expanded to include factors such as memory architecture[3], as well methods of memory access[8][7] .

Generally to efficiently schedule a Parallel Algorithm, a node may be retained that is not involved in the direct computation of the algorithm[27]. This node assigns data and initiates the various steps of the algorithm on the other nodes. The node that is not involved in the computation is termed the Master node, and those that are involved in calculation are known as Slave nodes[3].

4.4 Communication in Parallel Computers

If an algorithm can be parallelised, one of the more important factors regarding the efficiency of the parallel algorithm is the amount of communication between the processing units[6]. While communications technology has increased in performance greatly over the last few years, it can be assumed that the data rate of the communications paths between the processing units is not as fast as the rate at which the processing units can process data. In many cases this difference is by many orders of magnitude. As such communication in a parallel algorithm should be limited to the minimum as any communication can be considered a bottleneck[6].

Therefore communications tends to put an upper limit on the processing efficiency and hence the speedup of a parallel algorithm[*]. This should not be interpreted to imply that communication is a "necessary evil," as the fundamental difference, and the main source of benefit, between a parallel computer and a group of sequential algorithms running on different computers is the communication between different processing units. Therefore communication between processing units should be implemented in the most efficient manner.

Communications in parallel computers can in general be classified in 2 ways, whether the communication is point to point or collective and whether the communication is blocking or non-blocking[7]. Point to point communication occurs between one node and another. Collective communication involves communication between more than 2 nodes[7]. There are generally two forms, Broadcast, where a single node sends a copy of a single piece of data to multiple nodes, and Gather, where single node collects many pieces of data from multiple nodes[23]. A blocking communication means that the nodes involved in the communication must wait for the communication to complete before they continue other computations, non-blocking means that the nodes may attempt a communication and then continue with other computations before the communication has completed[23].

Communication in Parallel Computers can be implemented in many ways. The first way is to directly program the networking commands into the implemented parallel program. This is a difficult and intricate task and as the network infrastructure may vary between different systems, this method is

generally not portable (able to run on different architectures). Efforts have been made to create Messaging Interfaces (also called middleware) which implement all the communication requirements of Parallel Computers. The two most common are Parallel Virtual Machine (PVM), and Message Passing Interface (MPI). These provide a common interface to a parallel program, and are implemented on many architecture[7]. The specified interface simplifies the creation of parallel programs and allows the implemented programs to be run on any architecture that supports the messaging interface[7].

4.5 Parallelisation of a Pulsar Search Algorithm

As is illustrated in figure 3.3, a pulsar search algorithm can be split into the following components,

1. Read Data
2. Select a range of DMs, calculate DM step size
3. De-disperse Channels
4. Calculate Power Spectrum
5. Whiten
6. Harmonic Summing
7. Candidate Selection
8. Iterate DM and if within the range of DMs, go to step 3
9. Compare Candidates

Read Data: This step requires the reading in of an input file. Logically this takes place on the one node where the input file has been stored, generally a master node, therefore it is not parallelisable.

Select a range of DMs: This step requires calculation of the Dispersion Measure step size given by equation 3.2, after this the minimum and maximum dispersion measures are determined from the orientation of the observation relative to the Galactic Plane. These values are determined from the information in the input data and are common to all nodes, therefore they may be performed by the master node that reads in the input data, and then distributed to all slave nodes, or if the relevant data has been copied to the slave nodes, the slave nodes may compute these values for themselves.

De-disperse Channels: Each iteration of this step requires the time series data of the channel being shifted, as well as the Dispersion Measure step size and maximum and minimum Dispersion Measures. There are no other prerequisites to execution. Therefore assuming that the Dispersion Measure values have been calculated and distributed to all slave nodes, and the channels have been split up and distributed amongst the slave nodes, then this step is parallelisable. Once each slave node has added up the channels assigned to it, it sends the de-dispersed time series back to the master, which further adds the time series it receives, resulting in a single time series. Note that as indicated in section 3.2, the raw

data is generally not modified, but a copy of each channel is shifted and added to a summation vector on the fly, preserving the raw data for future de-dispersions.

Calculate Power Spectrum: This step requires the computation of an FFT on the de-dispersed Time Series. Parallel FFTs do exists, these split each step of the FFT and place them on individual slave nodes, therefore if the summation vectors of all slave nodes have been sent to a master node, then all slave nodes are now idle, and the parallel FFT may now be split up amongst the slave nodes, with the resultant DFT being placed on the master node. The Power Spectrum of the time series is then computed directly from the FFT. If however the choice has been made not to aggregate the time series onto a single node, then each slave node may calculate the FFT, for the de-dispersed time series of the channels allocated to it. The resultant DFT is then sent to the master node, which, due to the linearity property of the Fourier Transform[11], adds up the DFTs to create an aggregate DFT. The Power Spectrum is then calculated from the DFT.

Whiten: This step involves whitening the Power Spectrum to compensate for red noise, and then modifying the Power Spectrum into a form where the amplitude of a spectral component indicates signal to noise ratio for that component. Whitening requires the root mean square of the entire Power Spectrum to be calculated (requiring access to the entire Power Spectrum), therefore the Power Spectrum cannot efficiently be split up over the nodes to parallelise this step, therefore from hereon this step will be considered as part of the calculation of the Power Spectrum.

Harmonic Summing: This step requires that each component of the Power Spectrum have its harmonics added to it, the harmonics may extend over the entire range of the Power Spectrum, therefore this operation cannot be split up amongst many nodes to Parallelise it, therefore from hereon this step will be considered as part of the calculation of the Power Spectrum.

Candidates Selection: This step involves choosing a threshold signal to noise ratio. If a component of the power spectrum is above this threshold, then that component is considered to represent the frequency at which a periodic signal indicative of a pulsar is located. The Power Spectrum vector may be split up amongst the nodes and each slave node may search a segment of the Power Spectrum for

possible Pulsar Candidates.

Iterate DM: This step involves the calculation of the next Dispersion Measure to test. This can occur on the master node which can then distribute this value to the slave nodes, or the slave nodes may calculate this step for themselves if they have the required data. the algorithm then proceeds to the De-dispersion step.

Compare Candidates: This step consists of accumulating the possible Pulsar candidates and comparing them to each other as depicted in section 3.5. This requires that the all the information related to all discovered candidates, such as Dispersion Measure used, the period of the candidate, and the signal to noise ratio are known at the node where the comparison is taking place, as the selection criteria of a final Pulsar candidate is based on measurements relative to the other possible candidates. Therefore this step is not feasibly parallelisable and should occur at the master node.

On further analysis it can be seen that De-dispersion through till Iteration of the Dispersion Measure be parallelised in their entirety. This is due to the fact that all that is required to De-disperse the data for each iteration of the Dispersion Measure is the data itself, the parameters regarding frequency of the channels and the channel bandwidth and equation 3.1. This requires that all nodes have all copies of or access to data from the input file and the associated parameters such as frequency of the channels etc. Thereafter the DM range can be split amongst the nodes, and the nodes proceed with the entire algorithm from De-dispersion onwards.

As we have seen there are two feasible routes for parallelisation of a Pulsar Search Algorithm as defined in section 3:

1. An algorithm where the raw data is split into groups of contiguous channels and distributed equitably amongst the slave nodes. A range of Dispersion Measures is determined. The first Dispersion Measure in the range is selected and each slave node then dedisperses its assigned channels by this Dispersion Measure. The slave nodes send their De-dispersed time series to the master node. The master adds the time series that it received and computes the FFT. The

master node then computes the Power Spectrum. It then whitens the Power Spectrum and performs Harmonic Summing on it. After this the master selects candidates from the Power Spectrum. Note that while the master Performs the frequency domain calculations, the slave nodes are free to de-disperse their data by the next Dispersion Measure. This process is repeated for every Dispersion Measure in the range. The final set of candidates are compared by the master node and a candidates that best represent possible Pulsar signals are the output. As each node computes the same operation on each segment of data, and each node's segment of data is different. This algorithm has a Single Instruction in that the operations performed by the slave nodes are identical, and Multiple Data in that the channels distributed to each node are unique, therefore this is a SIMD algorithm.

2. An algorithm where all the data from the input file is distributed to each node. A range of Dispersion Measures is calculated and each slave node is assigned a subrange of Dispersion Measures. The slave nodes then de-disperse the data by their individual Dispersion Measures. The slave nodes complete the steps of computing an FFT, Power Spectrum and candidate selection. The nodes send their discovered candidates back to the master node once they have searched the data over the entire range of Dispersion Measures. The retrieved candidates are appended by the master, and compared, with the candidates that best represents possible Pulsar signals being the output. This algorithm has Multiple Instructions in that each slave node de-disperses by a different Dispersion Measure, and Single Data in that all slave nodes have identical copies of the data, therefore this is a MISD algorithm.

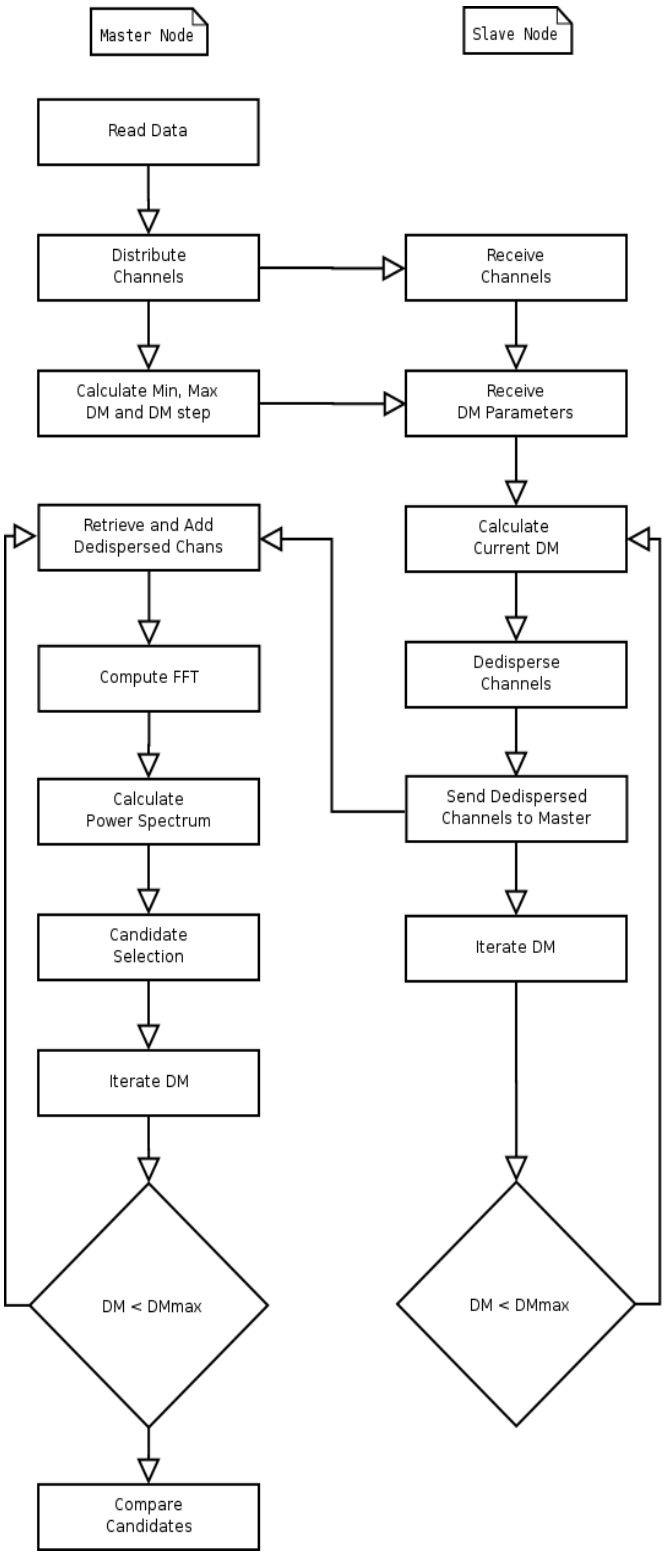

Figure 4.3: SIMD Parallel Pulsar Search Algorithm

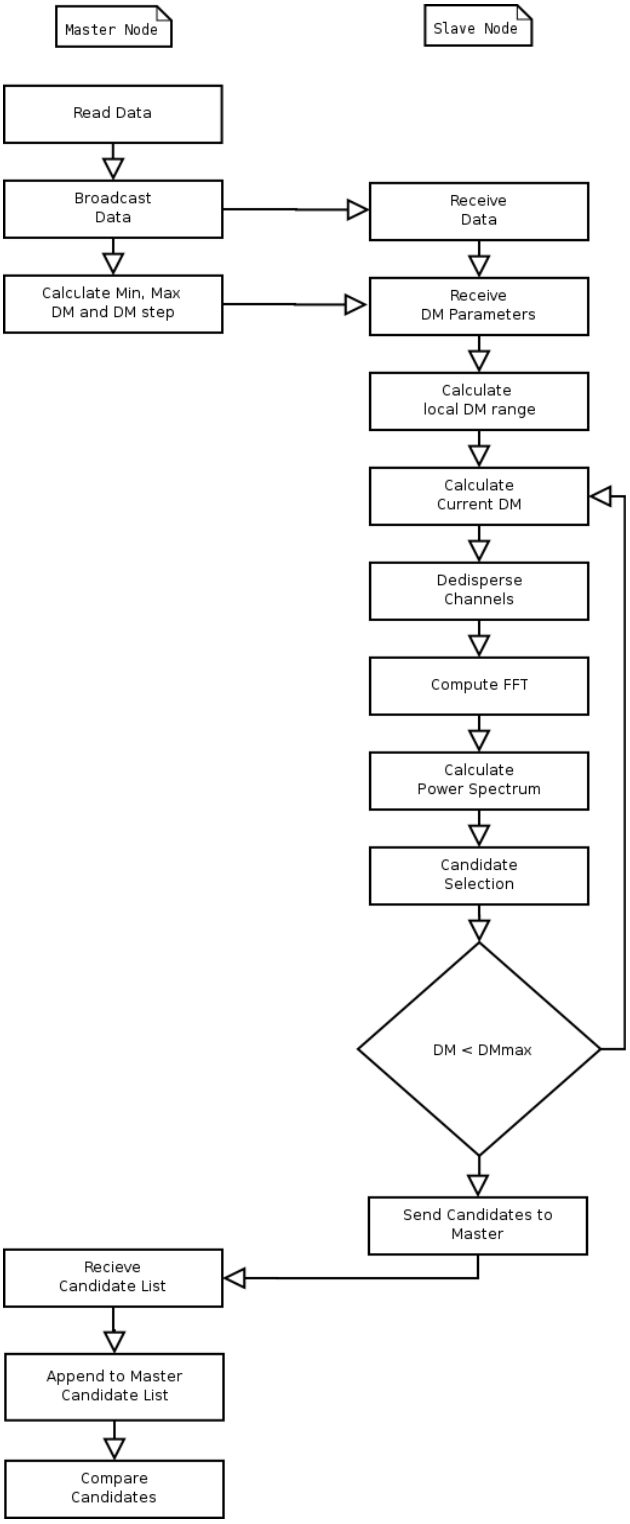

Figure 4.4: MISD Parallel Pulsar Search Algorithm

4.6 Computational Analysis of the Parallel Algorithms

The expected performance of the Parallel Algorithms with respect to the sequential algorithm are calculated below. Amdahl's law will be used as the as measure of relative performance. The expression created in the analysis of the sequential algorithm will be used as the basis for comparison. The same assumptions are made as when calculating the performance of the sequential design. The communication times are assumed to be zero, they will however they will be discussed.

Assuming that the observational data input file contains C = 128 channels, the is S = 270000 number of samples per channel and DM range to search is M = 230, then the sequential algorithm will spend 3.456×10^{7} operations reading in the file, 1.59×10^{10} operations de-dispersing, 11.204×10^{9} operations computing the FFT, 6.21×10^{7} operations calculating the Power Spectrum, 1.242×10^{8} Whitening, 4.968×10^{8} operations Harmonic Summing, 3.105×10^{7} operations performing the Candidate Selection and 2.3×10^{2} operations comparing the candidates. This gives a total of approximately 2.785×10^{10} operations.

The parallel computer to be simulated in this analysis will be assumed to have between 1 and 24 nodes, ignoring the difference between slave and master nodes.

4.6.1 Computational Analysis of the SIMD Algorithm

The SIMD algorithm only parallelises the De-dispersion stage, therefore the amount of operations that are parallelisable is equal to the amount of operations in the De-dispersion stage, 1.59×10^{10} , therefore the fraction of the sequential algorithm that is inherently sequential, with regard to the SIMD algorithm, is:

$$\frac{TotalTime-ParallelTime}{TotalTime}=\frac{2.785\times10^{10}-1.59\times10^{10}}{2.785\times10^{10}}=0.429$$

Using this figure with Amdhal's law, and plotting against the number of nodes, from 1 to 24 (ignoring the distinction between master and slave nodes) gives the following graph:

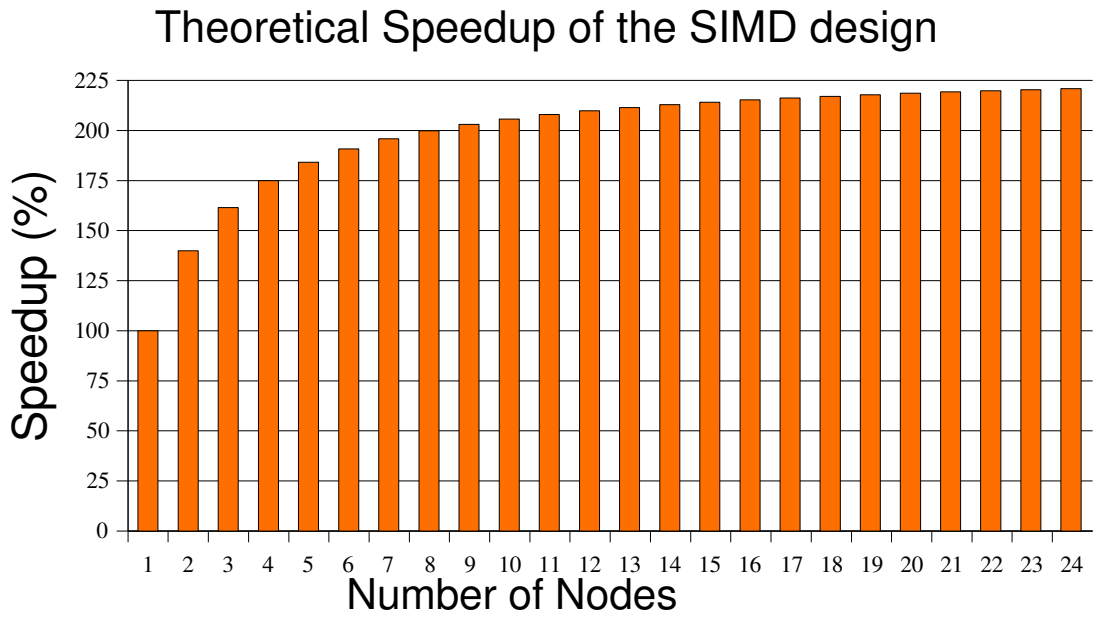

Figure 4.5: The Theoretical Speedup of the SIMD Algorithm

As $N\to\infty$, $S\to1.746$, giving this design a maximum theoretical speedup of nearly175%. This graph illustrates that this algorithm approaches this limit fairly quickly for small numbers of nodes and then temps asymptotically as the amount of nodes increases to Infinity.

4.5.2 Computational Analysis of the MISD Algorithm

The MISD algorithm parallelises everything between the calculation of the initial Dispersion Measure values and the Candidate Comparison. The total time of these components in operations is 2.782×10^{10}, therefore the fraction of the sequential algorithm that is inherently sequential, with regard to the MISD algorithm is:

$$\frac{TotalTime-ParallelTime}{TotalTime}=\frac{2.785\times10^{10}-2.782\times10^{10}}{2.785\times10^{10}}=0.0012$$

Again using this value with Amdhal's Law and plotting against the number of nodes gives the following graph:

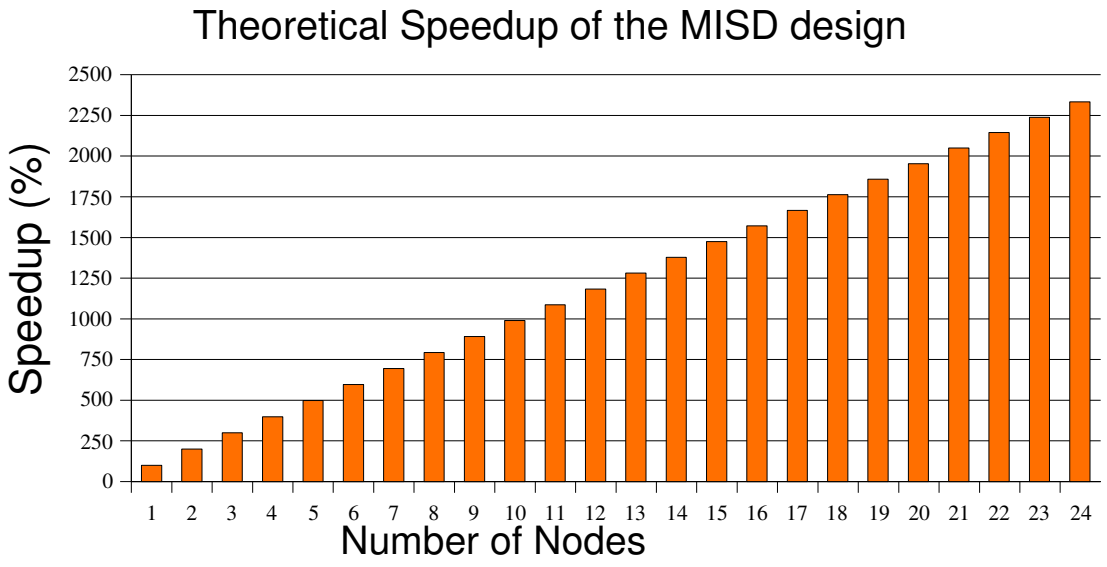

Figure 4.5: The Theoretical Speedup of the SIMD Algorithm

As $N\rightarrow\infty$, $S\rightarrow805.87$, giving this algorithm a maximum theoretical speedup of a phenomenal 80587%. As can be seen from the graph the speedup is nearly linear, but from Amdahl's Law, it can be inferred that the speedup will asymptotically approach 80587% as the number of Nodes tends to Infinity. This high value and near linear speedup indicates that Pulsar Searching is Embarrassingly Parallel.

4.5.3 Analysis of Communications in the Algorithms Designs

The parallel algorithms designed both have two main stages of communication, these stages are, distribution of observation data, and retrieval of results.

Distribution of observation data is handled in different ways in each algorithm. In the SIMD algorithm, the master node reads in the file in its entirety. It then distributes the channels equitably between the nodes. This amounts to a transfer of single copy of the raw data independent of the amount of nodes. In the MISD algorithm, an entire copy of the data is copied to each slave node. Therefore this amounts to a transfer of the observation data size multiplied by the number of nodes.

Retrieval of results is also different in the two parallel algorithms. In the SIMD algorithm each of the slave nodes returns the sum of its de-dispersed channels. this occurs once for each Dispersion Measure tested. This will typically be in the order of a megabytes to a tens of megabytes of information, depending on observation length and sample rate, therefore it might have an effect on the algorithm's performance. As more nodes are added, this effect will increase.

In the MISD algorithm each slave node returns its accumulated list of Pulsar candidates. This occurs at the end of the computation of all the Dispersion Measures. Typically if one candidate is generated for each Dispersion Measure tested, this will be in the order of only a few hundred bytes, and should have little effect on the performance.

5 Implementation

This chapter begins with a description of the "apparatus" used, i.e. software libraries, cluster specifications and input file formats. Following this is a section that describes the programming methodologies used, and a description of the translation of the designs into code. Compromises made on the design and unexpected problems with the implementation are also described in this chapter.

5.1 The KAT Cluster

The Parallel Computer to be used for the implementation is a 24 node cluster resident at the Karoo Array Telescope offices in Pinelands Cape Town. Each node has an Intel Core Duo dual core processor and 2GB of RAM. They are connected by 2 Gigabit Local Area Networks, providing a maximum of around 1000Mbits/s data rate. One network is for administrative use, and the other for parallel computing communications. They are arranged in a star topology, with a switch as the central hub node. There is a command terminal for diagnostics and control of the cluster. All nodes run the Linux operating system. The command terminal is connected to the Internet, enabling remote use of the cluster.

Figure 5.1: KAT cluster Command Node (left)

Figure 5.2: KAT Cluster Switches (Inside the large black box on the right), and nodes (Computer cases on the left)

Figure 5.3: KAT Cluster nodes

5.2 Messaging Interface

As mentioned earlier, Message Passing Interface (MPI) and Parallel Virtual Machine (PVM) are the two most prolific examples of middleware in contemporary parallel computing.

Both installed on the KAT cluster, however MPI was selected for the implementation of this project as it has become somewhat of a standard in Parallel Computing.

The MPI functions used were[23]:

MPI_Isend:	Non Blocking Point to Point send
MPI_Irecv:	Non Blocking Point to Point receive
MPI_Wait:	Waits for completion of a non-blocking communication, therefore turns a non-blocking communication into a blocking communication.
MPI_Barrier:	When called, requires all nodes to reach the same point in the code before execution can continue
MPI_Bcast:	Point to Multi-point Broadcast
MPI_Reduce:	Multipoint to Point Gather, it also performs an operation on the gathered data, eg. retrieve time series vectors from all nodes and add them.
MPI_Time:	Outputs the time in seconds from un unspecified point in the past. The time recorded at the beginning of the program execution, is therefore subtracted from the values recorded during execution.

In MPI, all nodes are identified by a unique ID called a rank. The master has a rank of 0, and slave nodes have ranks, greater than 0. All nodes that participate in an MPI communication must call the corresponding function, for example, if Node A wants to receive data from Node B, then Node A must call MPI_Irecv, and Node B must call MPI_Isend. For a node A to broadcast data to Node B and Node C, the must all call MPI_Bcast. The source and destination of communications is specified in the function call.

MPI programs are compiled using an MPI specific compiler called mpicc.
MPI programs are run using the command mpiexec -n N “program” “args”. Where N is the number of nodes to be used, “program” is the name of the compiled MPI program, and “args” are the arguments

passed to that program. A file called mpd.hosts must be created before an MPI program is run on multiple computers. This file contains the hostnames of the nodes to be used in the parallel computation.

5.3 Simulated Pulsar Data

Simulated Pulsar data was generated with a program called fake. This program is a part of the Sigproc package of Pulsar searching applications. The program takes in the parameters as arguments when executing the program. The parameters that were used in the implementation and testing of the implemented algorithms specified the Period of the Pulsar, peak Signal to Noise Ratio, duty cycle of the Pulsar, dispersion of the signal, observation length, sample time and number of bits per sample, the other available parameters were set to the default values. The output of the program is a Filterbank format file.

5.4 Search Data

The data format used is the called the Filterbank format. It is the format used by Duncan Lorimer's Sigproc package and by many radio observatories around the world. The Filterbank format defines many forms of data representation, however the format used by the software created for this thesis is the channelised time series format. The file format is described in the Sigproc manual[22]. The file starts with a header which contains all the values needed to perform a Pulsar search on it. The header data used in this implementation are sample time (tsamp), bits per sample (nbits), number of channels (nchans), frequency of the first channel (fch1), channel bandwidth (foff) and number of samples per channel (nsamples). All other required values can be inferred from this data. The data is in the form of a vector, with concatenated time instances for each sample, from each channel. This data format can conceptually be considered to be a two dimensional array as indicated in the diagram below:

Time Instance of Sample \ Channel	***0***	***1***	***2***
0	0	1	2
1	3	4	5
2	6	7	8
3	9	10	11
4	12	13	14
5	15	16	17

Figure 5.4: Example of Filterbank Data Format, for an observation with three channels and 6 samples per channel.

(Number inside the blocks indicates the index of the sample inside the file)

Therefore to access a sample in the file from its channel and sample instance, the formula for the index is $Index_{sample} = SamplingInstance \times nchans + ChannelIndex$ [22]

5.5 Algorithm Implementation

The sequential algorithm discussed below was implemented jointly by Roger Deane and the author of this document. The algorithms were implemented in the C programming language as this was the language both authors of the code were most familiar with. Every attempt was made to stick to the conventions of the ANSI C standard.

5.5.1 Sequential Algorithm Implementation

The sequential implementation is built in a modular fashion, with each block corresponding to a component of the designed algorithm, although a extra functions are needed to perform the necessary frontend functions for the ancillary calculations.

Read Data: There are two functions that implement the reading of the data file. The first called readHeader(), reads the data file and extracts the header information, this function is copied from the Sigproc program. The second function is called readData(), this reads the data file, from the end of the header onwards 1 byte at a time and placed the extracted value directly into an array called rawData.

Calculate DM values: The implemented algorithm does not calculate the Dispersion Measure, but the user of the program specifies the a range of Dispersion Measures by passing the minimum and maximum dispersion measure as arguments to the program. The first Dispersion Measure to test is set to the minimum Dispersion Measure specified. The Dispersion Measure step size is calculated by a function called initDM() which implements equation 3.2. All the Following stages up until the Dispersion Measure comparison are in a loop.

Compare Current DM with DM max: This is a single comparison between the current Dispersion Measure and the maximum specified by the user. If the current DM is greater than the maximum, the loop ends.

De-disperse Channels: This is accomplished by a function called dedisperse(). This function creates an empty array called sumChans to store the final de-dispersed time series. The function iterates through each channel, and for each iteration calls a function called Dmshift(). Dmshift() takes the index of the current channel in relation to the highest channel, and returns the amount by which the current channel should be shifted. A function called shiftRow() is then called which takes in the shift value and the current channel index as parameters. shiftRow() then iterates through each sample of the channel and adds this to the sumChans array, although at an index equal to the index of the sample – shift value. the contents of the data in memory is accessed by using a function called getSample, this implements the equation:

$$Index_{sample} = SamplingInstance \times nchans + ChannelIndex$$

and returns the value of data at that index in RawData. Note that rawData must be cast into the appropiate datatype to extract the sample eg. if there are 8, 16 or 32 bits per sample, rawData is cast into an unsigned char, short or int array. For 4 bits per sample, the appropriate nibble of the byte referred to by double the index is returned.

Compute FFT: A library called FFTW is used to implement the FFT. This library requires the creation of a "plan" which is a data structure, holding the parameters and relevant optimisation information for the FFT. The plan and the vector sumChans are then passed as parameters to a function called rfftw_one(), which is an FFT algorithm optimised for real, one dimensional data. This function returns the vector of the FFT of the array sumChans. The first half of the vector contains the real part of each frequency, arranged in ascending order of the frequency of the component, the second half contains the imaginary parts, arranged in descending order of frequency. This all takes place inside a function called getPS().

Calculate Power Spectrum: The real and imaginary parts of the FFT vector are squared and the squared components corresponding to each frequency are added. The results are placed in an array called PowerSpectrum. This also takes place inside getPS(). The code has been copied from the FFTW documentation, but has been modify to work with our code.

Whitening: The function that performs this operation is called whiten() on the array PowerSpectrum. This code has been copied from the Sigproc package. The mean is calculated by a function called getMean(), also copied from the Sigproc package, but originally by Numerical Recipes Software.

Harmonic Summing: Harmonic Summing is accomplished by a function called HarmonicSumming(). This function iterates through the array PowerSpectrum, and for each iteration, adding the components at integer multiples of its index, up to a maximum harmonic. The maximum harmonic is defined in the code as 16.

Candidate Selection: This step takes the component with the highest amplitude present in the post Harmonic Summing and whitened Power Spectrum. It appends this value along with the period of the candidate and the current Dispersion Measure to an array called candidates.

Calculate Current DM: This merely adds the DM stepsize to the current DM, after this has been performed, the loop moves back to the comparison between the current and maximum Dispersion Measures.

Compare Candidates: This function has not been implemented as such. The program simply writes all the candidates collected to a file, which the user can view. The user may plot the data within the file to observe if any frequencies have had their signal to noise ration maximised by a Dispersion Measure.

5.5.2 Parallel Algorithm Implementation

Both Parallel Algorithms required major changes to the code implemented for the sequential algorithm. The most visible being the requirement for MPI initialisation. these were placed in the beginning of the main() function. The changes to the algorithm are described along with the relevant steps of the algorithms.

The nodes (including the master and slave) in the implementation all use the same executable and therefore the flow of execution of their programs are the same. If a function in the executable requires activities to be performed by only the master, the activity is placed inside an if statement to compare the rank of the node calling the function with that of the master, the statement is of the form:

```
if (myRank == 0){
        ...Activity...
}
```

where myRank is the rank of the node calling the data similarly and the master node's rank is 0, if the activity is to be performed only by the slaves the form is:

```
if (myRank != 0){
        ...Activity...
}
```

Both parallel algorithms call a function called distParallelData() before the processing of the data begins, this function distributes the header to the slave nodes using the MPI_Bcast() function.

SIMD Implementation:

All functions described here are called from the processSIMD() function, which is in turn called from the main function when the operation mode is specified as SIMD.

Read Data and Distribute Channels: This is implemented in a function called readDataSIMD(). When the master calls the function, it reads data from the file, one sample at a time (note, not 1 byte), but does not place it directly into memory. Instead the Master computes the location in the rawData array if the data is laid out thus:

Channel \ Time Instance of Sample	*0*	*1*	*2*	*3*	*4*	*5*
0	0:0	3:1	6:2	9:3	12:4	15:5
1	1:6	4:7	7:8	10:9	13:10	16:11
2	2:12	5:13	8:14	11:15	14:16	17:17

Figure 5.5: Layout of rawData in memory for the SIMD implementation, for an observation with three channels and 6 samples per channel.

(The first number inside each block refers to the index of the sample in the file, the second, refers to the index of the sample in rawData)

This is to ensure that when the data is distributed to the nodes, the adjacent channels are contiguous in memory, allowing the channels being passed to a node, to be passed as a single block of data. Once this has been performed, then for each node the master calls MPI_Isend() with the block of data containing the channels assigned to that node. The slave node receives the data by calling MPI_Irecv().

Calculate DM values: The minimum and maximum Dispersion Measure values are passed to the master as arguments when the program is run. These are in turn distributed to the slave

nodes when the distParallelData() function is called. In operation this stage works in an identical manner to the sequential implementation.

De-disperse Channels: In the SIMD implementation, the dedisperse() function is replaced with dedisperseSIMD(). This function is only called from the slave nodes. This function iterates through the channels allocated to the node, and when DMshift() is called, the offset of the channel index of the first channel allocated to the node is given, so that the appropriate shift value for the channel is obtained. shiftRow() is replaced by shiftRowSIMD(). This function is identical to shiftRow(), except that it calls getSampleSIMD() instead of getSample(). getSampeSIMD() retrieves the sample in the reformatted rawData array using the equation:

$$Index_{sample} = ChannelIndex \times nsamples + SamplingInstance$$

Send and Retrieve De-dispersed Channels: The slave and master nodes both call the retrieveSIMDdata() function. This function calls the MPI_Reduce() function. The slave nodes specify sumChans as the input to the MPI_Reduce() function, and the master node specifies an empty array, the same length as sumChans, as the output. The reduce operation is MPI_SUM, this adds the corresponding elements in the input arrays and places them in the output array on the master. The master then assigns sumChans the address of the array, effectively copying the temporary array into its own sumChans vector. Once the slave nodes have performed this, they may continue De-dispersing the next iteration of the Dispersion Measure, while the Master performs the Frequency Domain operations.

Compute FFT: This is only performed by the master, and is identical in operation to in the sequential implementation.

Calculate Power Spectrum: This is only performed by the master, and is identical in operation to in the sequential implementation.

Whitening: This is only performed by the master, and is identical in function to the sequential

implementation.

Harmonic Summing: This is only performed by the master, and is identical in function to the sequential implementation.

Candidate Selection: This is only performed by the master, and is identical in function to the sequential implementation.

Calculate Current DM: This merely adds the DM stepsize to the current DM, after this has been performed, the loop moves back to the comparison between the current and maximum Dispersion Measures. This occurs on both the slave and the master nodes.

Compare Candidates: This is only performed by the master, and is identical in function to the sequential implementation.

MISD Implementation:

All functions described here are called from the processMISD() function, which is in turn called from the main function when the operation mode is specified as MISD.

Read Data and Broadcast Data: This is implemented in readDataMISD(). The master calls the same readData() function as the sequential implementation. The master and the slave nodes all call MPI_Bcast() on the rawData array, with the master specified as the source. This sends a copy of rawData to each slave node.

Calculate DM values: This operates in the same manner as in the sequential, however, the DM values are broadcast to the slave nodes when distParallelData() is called. The slave nodes calculate their portion of the Dispersion Measure range by comparing their rank to the total number of nodes.

Compare Current DM with DM max: This is a only called by the slave nodes, and is identical to the sequential implementation.

De-disperse Channels: This is a only called by the slave nodes, and is identical to the sequential implementation.

Compute FFT: This is a only called by the slave nodes, and is identical to the sequential implementation.

Whitening: This is a only called by the slave nodes, and is identical to the sequential implementation.

Harmonic Summing: This is a only called by the slave nodes, and is identical to the sequential implementation.

Candidate Selection: This is a only called by the slave nodes, and is identical to the sequential implementation.

Calculate Current DM: This is a only called by the slave nodes, and is identical to the sequential implementation.

Send and Retrieve Pulsar Candidates: This operation is performed in retrieveDataSIMD(). Each slave node creates an empty vector that is as long as the amount of Dispersion Measure iterations times the amount of values used to represent each candidate (3, one for DM, one for Signal to Noise and one for the frequency). The candidate information generated by the slave node is then placed in this vector, the initial index depending upon which subrange of the Dispersion Measure range the node searched. These are added into the candidates array on the master using the MPI_Reduce() command.

Compare Candidates: This is a only called by the master node, and is identical to the sequential implementation.

6 Testing

This chapter illustrates the testing methodologies used in analysing the implemented parallel algorithms. It begins with a comparison of the output of the sequential algorithm, with that of the parallel algorithms, all with identical inputs. A description of the comparison criteria is then given, these being completion time, percentage speedup and communications efficiency. Graphs displaying performance the each of the algorithms with regard to the aforementioned criteria are then shown.

6.1 Testing Methodology

Testing was performed on the implemented algorithms by generating simulated Pulsar signals using the "fake" program. Testing involved two stages. First, the designs were verified by feeding them data with known characteristics and observing the output to ensure that it matched the expected values. The next stage of the testing was to ascertain whether the objectives set out at the beginning of the project had been achieved, and to gather data on the algorithms for further analysis.

6.2 Verification Test

Using the fake program provided with Sigproc, simulated observation data was created. This data simulated a pulsar with a peak signal to noise ratio of 5, a period of 1s and a duty cycle of 30%, the level of dispersion was set to 121. The three implemented algorithms were then run using the simulated data as input. The Dispersion Measure range was specified to be from 0 to 200. Signal to Noise ratio and Dispersion Measure for each candidate was extracted from each algorithm's output file. These values were plotted against each other to see whether the outputs of each algorithm matched, and to see if the algorithms had correctly processed the data from the file.

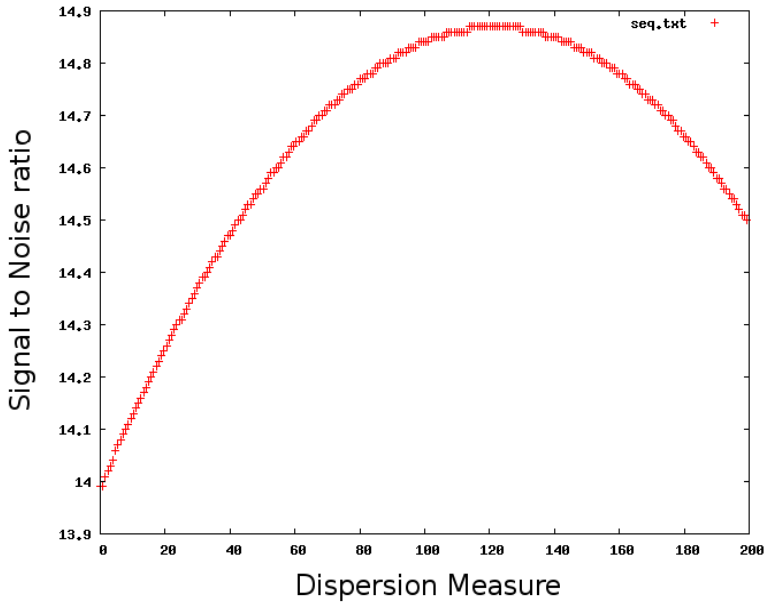

Figure 6.1: Candidates Output by Sequential Implementation

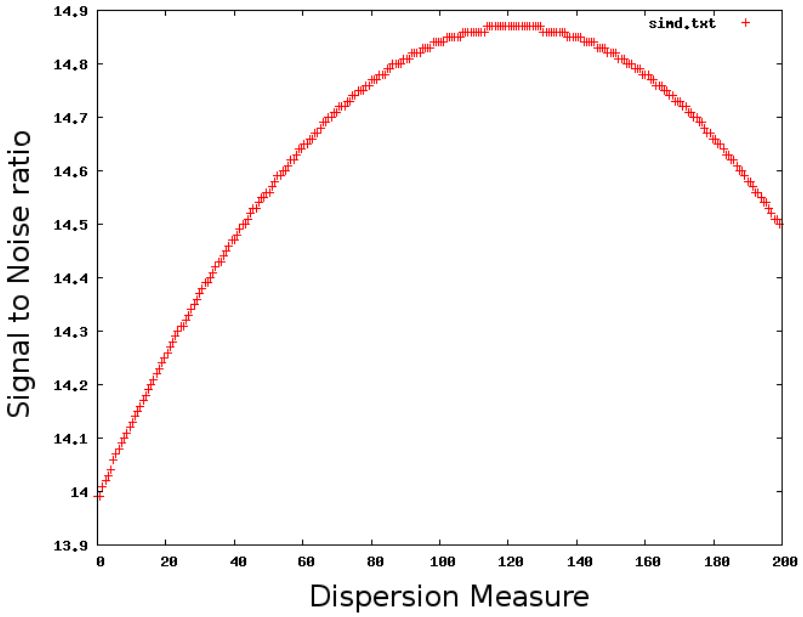

Figure 6.2: Candidates Output by SIMD Implementation

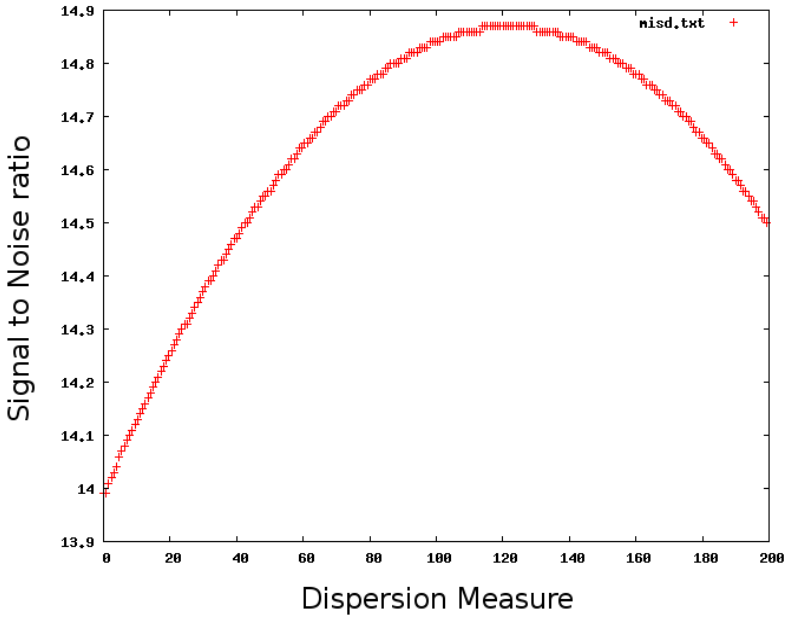

Figure 6.3: Candidates Output by MISD Implementation

6.3 Algorithm Performance Tests

These tests involved inserting MPI_Time() functions in the implemented program to record the time taken to complete various operations. The entire run time of each algorithm was measured, and for the parallel algorithms, the amount of time spent sending the data from the master to the nodes, and also retrieving data. As the processing load on the cluster was not constant (due to use by other users), each test was performed twice and the average value of the two runs was for plotting the results.

The metrics used for the tests were the completion time of the algorithms, the percentage speedup of the parallel algorithms over the sequential algorithm and the efficiency of the parallel algorithms, with efficiency defined here as the fraction of the running time time spent processing (as opposed to communicating). The tests were performed on the cluster using 2 nodes (1 master node and 1 slave node) to 24 nodes (1 master and 23 slaves). The data was 65 megabytes in size, consisted of 128 channels with 266752 16 bit samples per channel. The Dispersion Measure range was set to 0 to 200 and consisted of 232 increments. The sequential implementation had an average run time of 778 seconds. The data collected for the tests is tabulated in Appendix A.

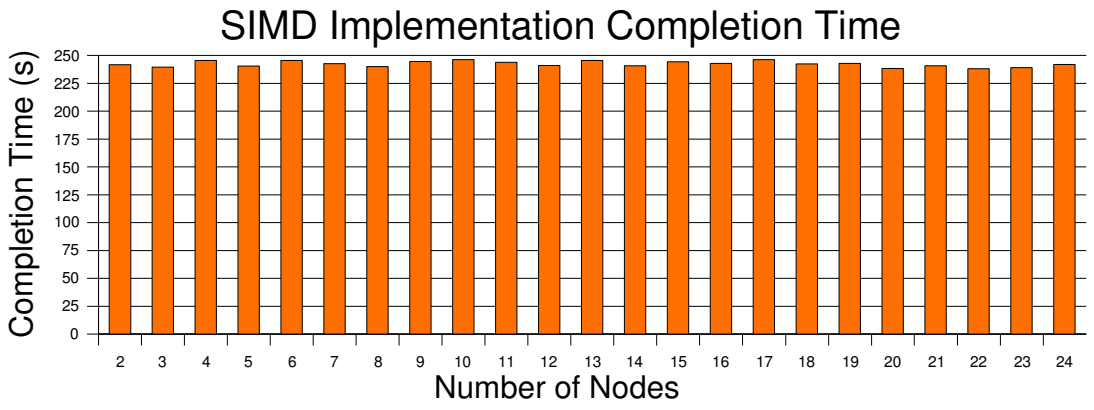

Figure 6.4: SIMD Implementation Completion Time

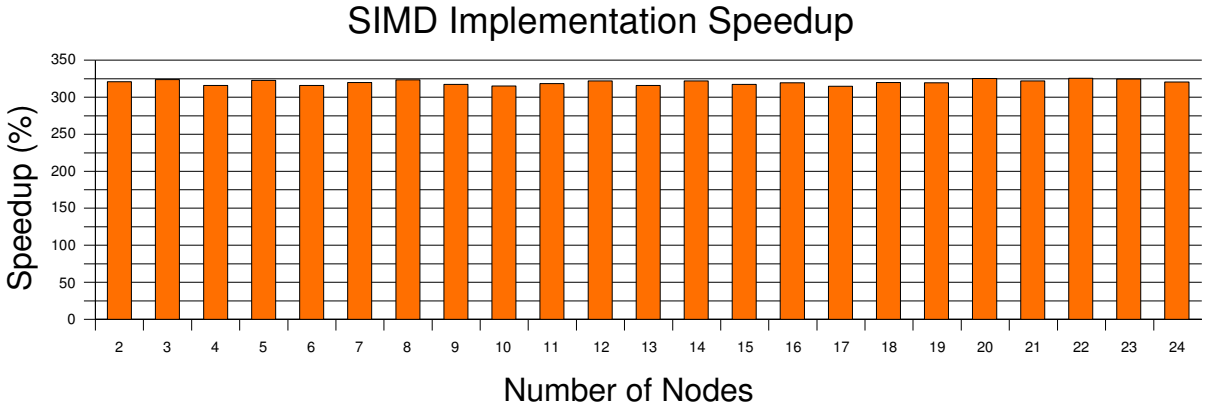

Figure 6.5: SIMD implementation Speedup over the the Sequential Implementation

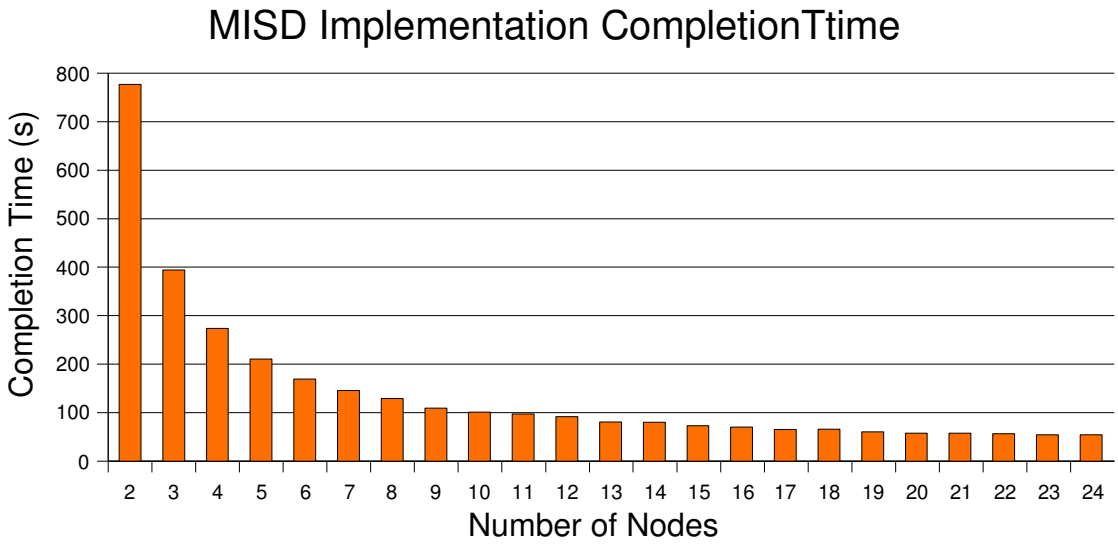

Figure 6.6: MISD Implementation Completion Time

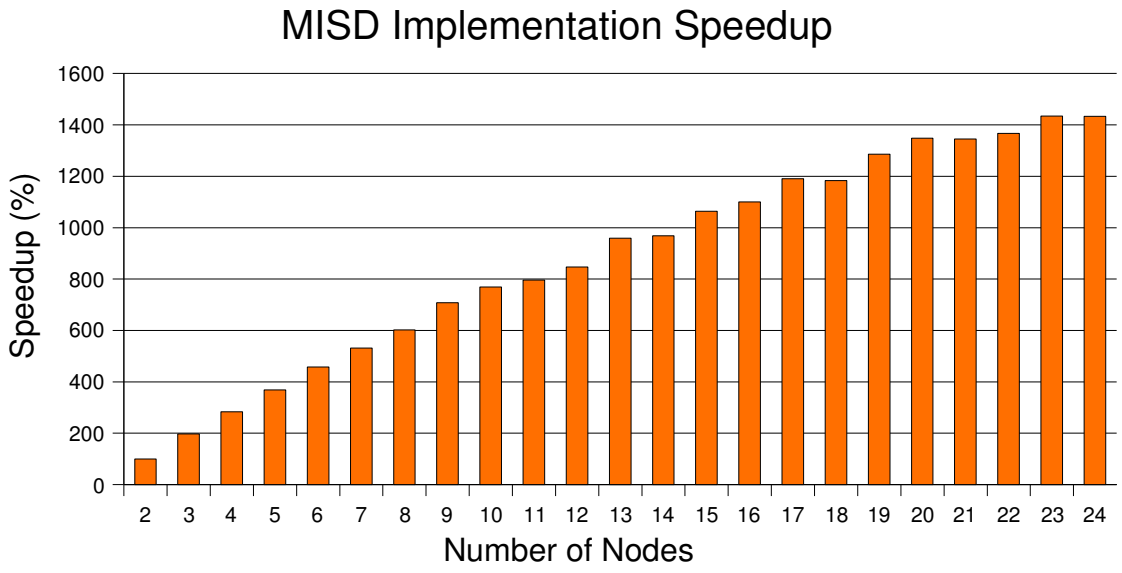

Figure 6.6: MISD Implementation Speedup over the Sequential Implementation

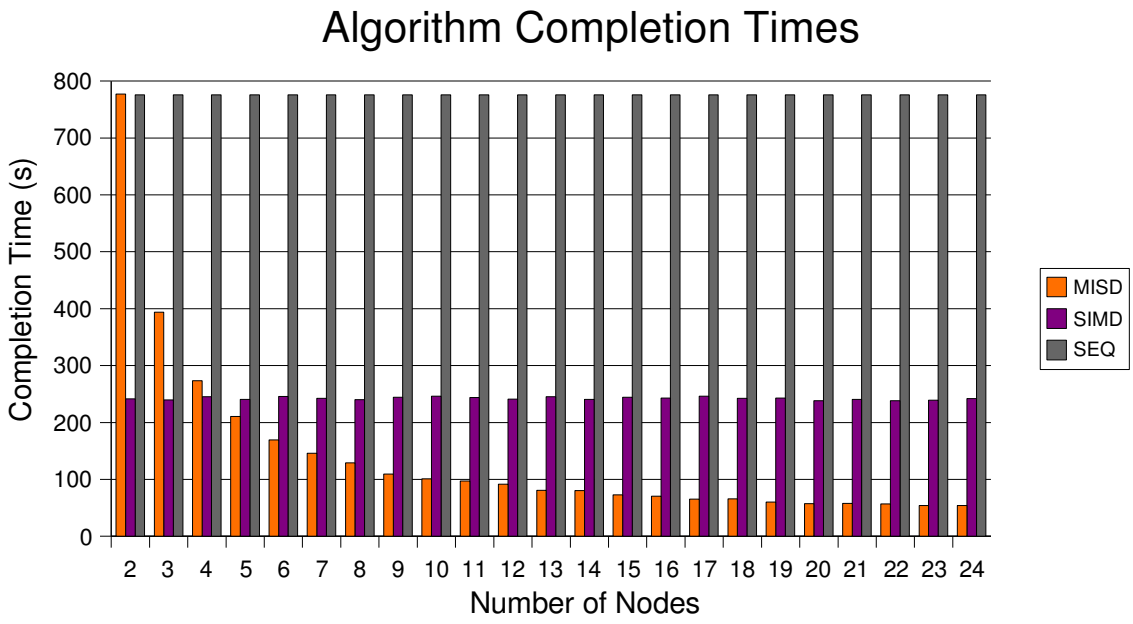

Figure 6.6: Comparison of the Algorithm Completion Times

(Note that sequential algorithm is always executed on one node, it is displayed for comparison)

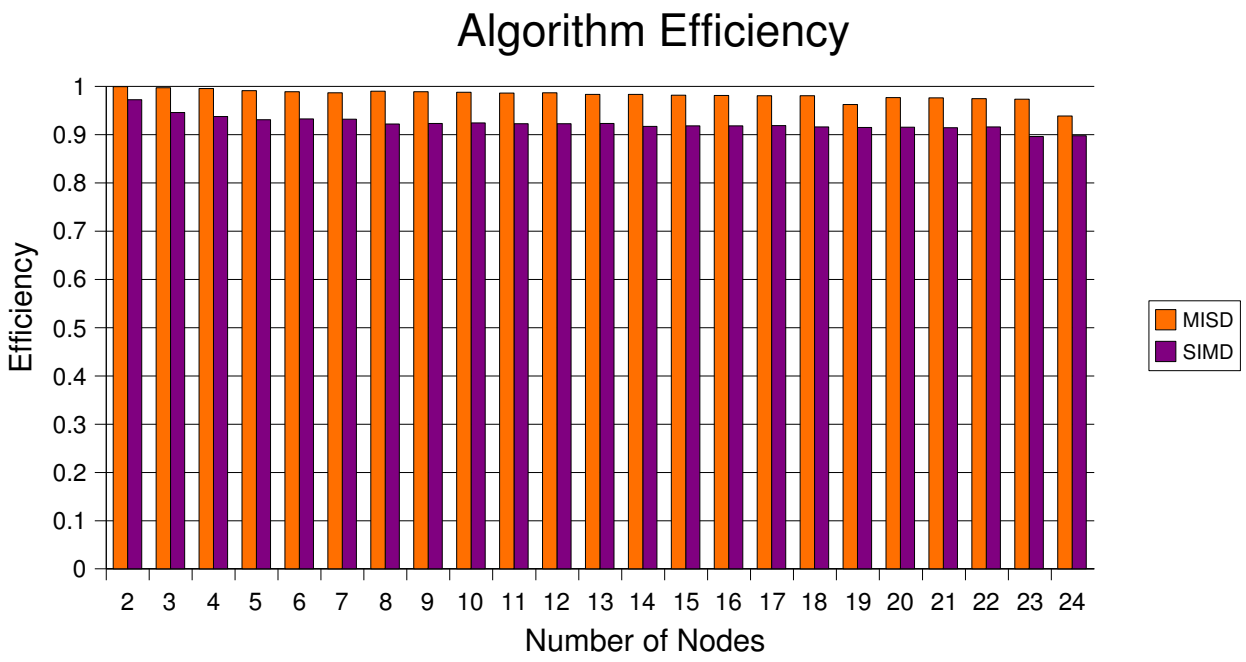

Figure: 6.8: Algorithm Efficiency (Proportion of Time Spent Processing)

7 Results and Analysis

An in depth analysis of the results obtained in the testing phase is described in this chapter. All algorithms performed the processing on the data file identically, and can be considered to function as Pulsar Search Algorithms. The performance results of the MISD implementation are shown to correspond to the results predicted in the design, although there are some performance penalties. The results of the SIMD implementation proved anomalous, not achieving their theoretical potential or exhibiting signs of parallel execution, however, it was faster than the sequential algorithm. The possible reasons for the results observed are given.

7.1 Verification

All algorithms gave identical outputs when presented with identical inputs. This was not only observed in the verification test, but also when gathering data for performance and analysis tests.

The graphs indicate that as the Dispersion Measure being tested, approaches 121, the signal to noise ratio increases and is maximised when the Dispersion Measure reaches 121. This corresponds with the expected behavior of a Pulsar Search Algorithm, and also with the values specified in the data file.

The signal to noise ratio output proved to be anomalous. When converted to decibels, the maximum signal to noise ratio observed in the verification test was seen to be 11.7. This proved to be the case for all tests performed with all three of the algorithms. The signal to noise ratio as defined in fake is termed the “peak” signal to noise ratio, no adequate definition of this term could be found, and as such no relationship can be inferred between the signal to noise ratio in the input file, and the signal to noise ratio in the algorithms' output, other than the fact that the signal to noise ratio dynamics with respect to the Dispersion Measure are correct. This discrepancy should have little to no effect on the detection of possible Pulsars, as it is the relative differences in the components of Power Spectrum is used to select candidates.

7.2 Algorithm Performance

The performance tests indicated that both algorithms have a definite speed advantage over the sequential algorithm. However, when compared with the performance predicted in the design, both algorithms fall short. All the results obtained were displayed fluctuations, probably due to different operating loads present on the cluster for each of the test runs. The results discussed below ignore this fluctuation in the interpretation of the test results, unless the fluctuations are deemed to have greatly affected the outcome of the tests.

The graphs indicate that the SIMD implementation shows no observable increase in speed when more nodes were added, however, it is consistently faster than the sequential algorithm, even with one master and one slave node. This is to some extent, expected, as while de-dispersion stage in the slave nodes is paralellised, there is further parallelisation, in that after data from the slaves has been collected, the master node is able to perform the frequency domain search as the slaves compute the next Dispersion Measure iteration. This holds even for the case of one slave and one master, although the level of speedup, 300%, is much greater than expected. The maximum speedup for one node and one master should be 200%, as the greatest efficiency should be experienced when the sequential algorithm is split evenly between them.

A factor that may contribute to this speed benefit in the SIMD design is that the data for the SIMD implementation is reordered into concatenated time series, therefore consecutive samples are physically next to each other in memory, whereas with the sequential and MISD implementations, consecutive samples are nchans -1 apart. If the processor on which the algorithm is being run implements caching, which the Core Duo definitely does, then for each memory access, the processor will fetch the surrounding blocks of memory to the cache to speed up the next access (the processor assumes that the next access will occur next to or near the first), then there is a greater chance that consecutive samples in the SIMD implementation will be fetched from the cache, which is much faster than a fetch from memory. The time of completion for the de-dispersion operation for the nodes in the SIMD operation is given in table 2 in Appendix A. For the case of 2 nodes, then there is one slave, and this slave is allocated the entire contents of the raw data. Therefore the dedispersion operation is performed on the

same amount of data as with the sequential implementation. Note that the time taken for de-disperision is 0.64s, when the sequential algorithm was run with the same input data, the time taken to de-disperse data was 1.86 seconds, almost 3 times as long, indicating that there are definite advantages to the SIMD algorithms data storage format.

The lack of increase in computation speed when more nodes are added is probably due to the fact that as the frequency domain search time on the master node remains constant due to the operations being performed being constant. If the de-dispersion stage on the slaves is computed faster than the frequency domain stage is on the master (due to parallelisation), then the slave nodes will have to wait for the master to finish the the frequency domain processing, then send their results to the master before they will be able to continue into the next iteration of the Dispersion Measure. Therefore the lower limit on the completion time is dependent on the time it takes to perform the frequency domain search. A further test was performed to see if the de-dispersion stage did indeed proceed faster than the frequency domain processing.

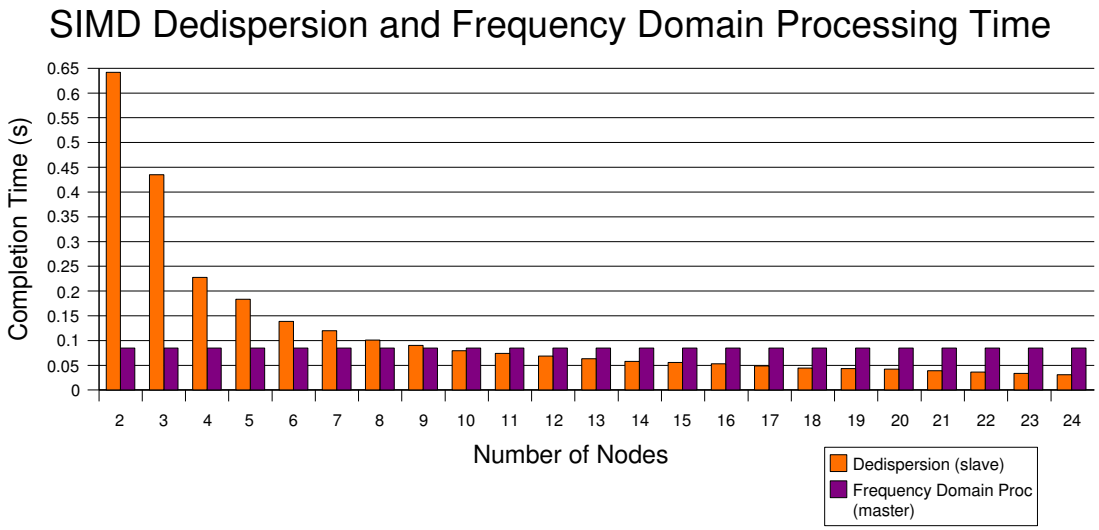

Figure 6.9: Comparison of the De-dispersion and Frequency domain processing in the SIMD implementation

When designing the algorithm it was assumed that de-dispersion would always be many times slower than any of the other operations even for large numbers of slave nodes. Obviously this is not the case. This does not however explain the relatively constant completion time exhibited by the SIMD

implementation. This proved to be an elusive feature to analyse, and none of the tests performed indicated any reason for this.

The communication efficiency of the SIMD implementation is as predicted in the analysis of the design. The data in **Table 1** of Appendix A shows that for more than one node, the time required to distribute the data is relatively constant (ignoring fluctuations in the data), especially when compared to the time required to retrieve the channels, this is as expected as the same amount of data is transferred irrespective of the number of nodes. The small increases in the time required to distribute the data as the number of nodes increases, is probably due to the overhead involved with each additional point to point communication. The time required for retrieving the time series increased as expected with each additional node.

The MISD implementation was a lot more well behaved with regard to its predicted performance. The effects of adding additional nodes to the computation are clearly visible. Up to approximately ten nodes, the graph displays an almost linear increase in performance, increasing from one slave node to two, increases the speedup to 197.3%, very close to the theoretical prediction of 200%. after ten nodes, the incremental increase in performance appears to decrease although, fluctuations in the values makes interpretation difficult. However if one calculates the percentage difference between the theoretical speedup and the achieved speedup, the discrepancy is seen to increase almost linearly, as the number of nodes increases, this is illustrated in figure 6.9 giving a clear indication of the effects of communication and other overheads. Despite this, this algorithm clearly provides the most scope for parallel execution.

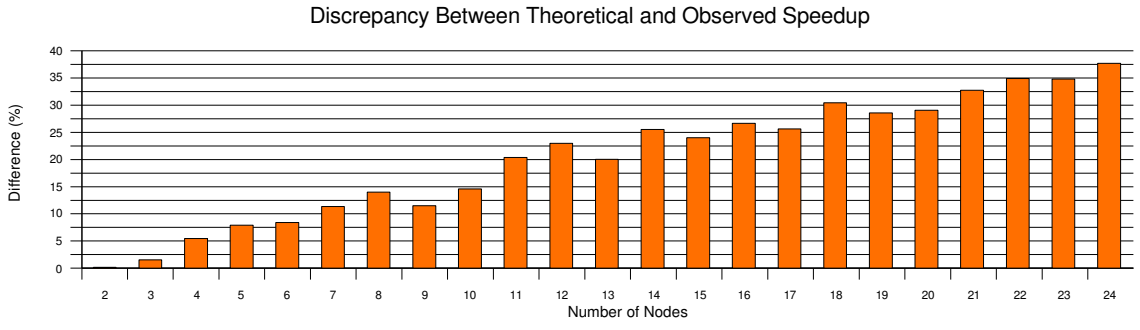

Figure: 6.10: Discrepancy Between the Theoretical and Observed Speedup for the MISD Implementation

8 Conclusions

All the implemented algorithms are able to detect simulated Pulsar's in the simulated observation data, therefore the algorithms can be considered to function as Pulsar Search Algorithms. The method of Candidate Detection implemented is however crude, and does not automate the detection of the optimum Dispersion Measure.

The parallel algorithms are both faster than the sequential algorithm, although by different degrees, All in all, the MISD algorithm proves to be the most effective in terms of a parallel speedup. A benefit of the SIMD implementation, is that as the number of nodes increases, the the more the data is split up, and therefore the less memory required by each node for execution leading to a relatively efficient usage of memory. All in all, the objectives set out at the beginning of the project have most definitely been achieved.

The SIMD algorithm proved to be of a fairly naïve design, as illustrated by the discrepancies between the designed and implemented performance. The performance increase from its use is definite, however further parallelisation beyond the use of 2 nodes is of no benefit. The MISD algorithm would therefore be the most appropriate in a real Parallel Pulsar Searching environment, as it is greatly parallelisable and provides the greatest ultimate performance boost.

Implementation of a more thorough candidate selection, and Dispersion Measure optimisation strategy would be the next step if the sequential algorithm is to be further refined. For the Parallel Algorithms, an in depth analysis of the communications would probably prove to be the most effective method of improving performance. Implementation of further Parallelisation schemes, such as a parallel FFT would be the logical next step in refining their design. An investigation into different algorithm topologies such as a pipeline might result in even better parallel algorithms.

Bibliography

[1] *Wikipedia Article on Pulsars*, en.wikipedia.org/wiki/Pulsars

[2] D.R. Lorimer, M. Kramer,*Handbook of Pulsar Astronomy*, Cambridge University Press

[3] T.G.H. Bennett, *Development of a Parallel SAR Processor on a Beowulf Cluster*, Cape Town: University of Cape Town, 2003

[4] *Wikipedia Article on Amdahl's Law*, http://en.wikipedia.org/wiki/Amdahl's_law

[5] S. Mukherjee, *Parallel Implementation of an Algorithm for High Resolution Range Profiling using Stepped Frequency Radar*, Cape Town: University of Cape Town, 2004

[6] J. Dongarra, I. Foster, G.C. Fox, W. Gropp, K. Kennedy, L. Torczon, & A.White *The Sourcebook of Parallel Computing,* Morgan Kaufman Publishers, San Francisco, USA, 2003

[7] R. Buyya, *High Performance Cluster Computing Volume 1*, Prentice Hall, New Jersey, USA, 1999

[8] S.G. Akl, *The Design and Analysis of Parallel Algorithms,* Prentice Hall, USA, 1989

[9] R.A. Hulse, *The Discovery of the Binary Pulsar, Nobel Lecture*, December 8, 1993 Princeton University, Plasma Physics Laboratory, Princeton, NJ 08543,
USA

[10] O. Fadiran, *Design and Implementation of a Parallel Registration Algorithm for SAR Images*, Cape Town: University of Cape Town, 2001

[11] N. Morrison, *Introduction to Fourier Analysis,* Wiley-Interscience, USA, October 1994

[12] *Wikipedia Article on the Central Procesing Unit*
http://en.wikipedia.org/wiki/Central_processing_unit

[13] *Wikipedia Article on Parallel Computing* http://en.wikipedia.org/wiki/Parallel_computing

[14] *Wikipedia Article on Moore's Law*
http://en.wikipedia.org/wiki/Moore%27s_law

[15] *Wikipedia Article on the Galactic Coordinate System*
http://en.wikipedia.org/wiki/Galactic_coordinate_system

[16] A.J. Wilkinson, *EEE3058S Notes on Power Spectral Density,* Cape Town: University of Cape Town, 2001

[17] A.J. Wilkinson, *EEE3058S Notes on Noise Power,* Cape Town:
University of Cape Town, 2001

[18] F.Nicolls, *EEE4086F Notes on Discrete time Signal Processing*, Cape Town: University of Cape Town, 2001

[19] *Wikipedia Article on the Fast Fourier Transform*
http://en.wikipedia.org/wiki/FFT

[20] *Wikipedia Article on Big O notation*
http://en.wikipedia.org/wiki/Big_O_notation

[21] *Ars Technica Article on the Ultimate Limits of Computers*
http://arstechinca.com/wankerdesk/01q2/limits/limits-1.html

[22] D.R. Lorimer, *Sigproc Manual*
http://sourceforge.org/sigproc/

[23] *MPI Specification*
www.mpi-forum.org/docs/docs.html

[24] *University of Manchester Cobra cluster website*
www.jb.man.ac.uk/~pulsar/cobra/science/

[25] J. Blythe et al, *Transparent Grid Computing: a Knowledge-Based Approach*
www.isi.edu/~gil/papers/iaai03.pdf

[26] *University of Illinois at Urbana-Champaign Radio Astronomy Imaging Group Website*
http://monet.ncsa.uiuc.edu/

[27] A.G. Willis, *Astronomical Data Reduction on Parallel Computers Using AIPS++*
http://monet.ncsa.uiuc.edu/aips++/index.html

[28] Parallel and Distributed Processing with glish and AIPS++
www.drao-ofr.hia-iha.nrc-cnrc.gc.ca/science/ska/other_formats/jodrell_parallel_proc.pdf

Appendix A: Test Data Results

	MISD			SIMD	
Number of Nodes	Completion Time	Send Time	Receive Time	Completion Time	Send Time
2	777.1	0.59	0	241.6	0.59
3	394.02	1.18	0	239.55	0.59
4	273.56	1.24	0	245.49	0.59
5	210.68	1.85	0	240.41	0.6
6	169.48	1.85	0	245.61	0.6
7	145.92	1.9	0	242.62	0.6
8	128.9	1.28	0	239.98	0.61
9	109.6	1.22	0	244.57	0.61
10	100.93	1.21	0	246.28	0.61
11	97.47	1.36	0	243.7	0.62
12	91.62	1.24	0	240.92	0.64
13	80.9	1.34	0	245.53	0.65
14	80.14	1.34	0	240.84	0.64
15	72.95	1.33	0	244.32	0.68
16	70.55	1.33	0	242.84	0.7
17	65.22	1.28	0	246.34	0.69
18	65.61	1.29	0	242.51	0.69
19	60.37	2.26	0	242.82	0.74
20	57.57	1.33	0	238.46	0.78
21	57.7	1.37	0	240.75	0.76
22	56.78	1.44	0	238.19	0.76
23	54.11	1.43	0	239.12	0.76
24	54.15	3.34	0	241.96	0.97

Table 1: Running time, The taken to Distribute Data, and Time taken to Receive Data

(The Sequential Algorithm had a runtime of)

Number of Nodes	Dedispersion (slave)	Frequency Domain Proc (master)
2	0.64	0.08
3	0.43	0.08
4	0.23	0.08
5	0.18	0.08
6	0.14	0.08
7	0.12	0.08
8	0.1	0.08
9	0.09	0.08
10	0.08	0.08
11	0.07	0.08
12	0.07	0.08
13	0.06	0.08
14	0.06	0.08
15	0.06	0.08
16	0.05	0.08
17	0.05	0.08
18	0.04	0.08
19	0.04	0.08
20	0.04	0.08
21	0.04	0.08
22	0.04	0.08
23	0.03	0.08
24	0.03	0.08

Table 2: Comparison of the De-dispersion and Frequency domain processing in the SIMD implementation.

Appendix B: Readme File

Tsepo Montsi
mnttse002

This CD contains the document and source code of Tsepo Montsi's undergraduate Thesis.

The document is inside the Document Folder, the source code is inside the Source folder.

The layout of the files is:

Folder	File	Description
./Document/PDF:	Cover.pdf	Front Cover in PDF format
	Pre.pdf	Table of contents and preamble in PDF format
	Body.pdf	Thesis Document in PDF format
./Document/DOC:	Cover.odt	Thesis Document in Open Document Text format
	Pre.odt	Table of contents and preamble in Open Document Text format
	Body.odt	Thesis Document in Open Document Text format

./Source	makefile	Makefile to compile the source code
	header.h	Sequential algorithm header file
	frontend.c	Source file containing sequential algorithm frontend functions
	processing.c	Source file containing processing functions
	P_header.h	Parallel algorithm header file

P_psa_main.c	Source file containing main function
P_functions.c	Source file containing functions used in the parallel algorithms
moveexe.c	Program to read in hostnames from mod.hosts, and copy the compiled executable to the different nodes

The code within the header.h, processing.c and frontend.c source files was jointly implemented by Tsepo Montsi and Roger Deane, except where otherwise indicated. The code in all the other source files was implemented by Tsepo Montsi.

Compilation Instructions:
This Program requires an MPI library and the FFTW library to be installed!

1. Open a terminal and enter the directory where you have placed the source folder

2. Open the makefile and follow the instructions at the top

2. Type make

3. To run the program an MPD ring must be established, consult your MPI distribution's documentation as to the procedure

4. to run the program type:

```
mpiexec -n N ./ppsa -mode "mode of execution"
-f "name of data file"
```

-dmmin [minimum DM]

-dmmax [maximum DM]

-o "candidate file name"

Where, N is the number of Nodes to use, mode of execution is (all without quotes) "misd" to run the MISD algorithm, "simd" to run the SIMD algorithm, and "seq" to run the sequential algorithm

"Name of data file" refers to the name of the Data file used

minimum DM refers to the minimum DM in the range

maximum DM refers to the maximum DM in the range

"candidate file name" refers to the name of the candidates output file

5. A file called moveexe.c has been included. Compile this by typing:

gcc moveexe.c -o moveexe (substitute whatever C compiler you use if you don't use gcc)

usage: moveexe "source binary" "destination directory"

For example if your executable is in /home/tsepo/bin and you want the executable copied to
the same directory on the nodes type:

./moveexe /home/tsepo/bin/ppsa /home/tsepo/bin/

Run this program in the same directory as the mpd.hosts file