Spring 2015

Studying the effect of parallelization on the performance of Andromeda Search Engine: A search engine for peptides

Jigna Shah
Purdue University

Graduate School Form 30
(Revised 08/14)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Jigna Shah

Entitled
STUDYING THE EFFECTS OF PARALLELIZATION ON THE PERFORMANCE OF THE ANDROMEDA SEARCH ENGINE : A SEARCH ENGINE FOR PEPTIDES

For the degree of Master of Science

Is approved by the final examining committee:

John Springer

Dawn Laux

Michael Kane

To the best of my knowledge and as understood by the student in the Thesis/Dissertation Agreement, Publication Delay, and Certification/Disclaimer (Graduate School Form 32), this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

John Springer

Approved by Major Professor(s): _____

_____

Approved by: Jeffrey Whitten 04/27/2015

Head of the Department Graduate Program Date

STUDYING THE EFFECT OF PARALLELIZATION ON THE PERFORMANCE OF ANDROMEDA SEARCH ENGINE: A SEARCH ENGINE FOR PEPTIDES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Jigna Shah

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2015

Purdue University

West Lafayette, Indiana

ACKNOWLEDGEMENTS

I extend my gratitude to my major Professor, John Springer for motivating me and guiding me from the beginning to the end. I also want to thank Prof. Dawn Laux and Prof. Michael Kane for their invaluable inputs and support.

I am also grateful to Lake Paul and Ernesto Nakayasu for their patience and for working with me and helping me to understand the Andromeda Search Engine.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

GLOSSARY

Biological Marker (Biomarker) – It is an indicator of a disease and severity of it that can be me.

Computational performance – The time taken by a software and its constituent functions to complete their execution and return results is called computational performance. (Lichtenberg et al., 2010)

iTRAQ Reagents – "The iTRAQ Reagents are the first set of multiplexed, amine-specific, stable-isotope reagents that can label all peptides in up to eight different biological samples enabling simultaneous identification and quantitation." (iTRAQ Reagents, n.d)

Mass Spectrometry- Mass spectrometry is a technique used in analytical chemistry that measures the mass-to-charge ratio and gas phase ions abundance that in turn help in characterizing the sample on the basis of the chemicals present. (Sparkman, O. David , 2000).

Proteomics- A study of a cell proteome. (Srinivas, Verma, Zhao, & Srivastava, 2002)

SNP genotyping- "It is the measurement of genetic variations of single nucleotide polymorphisms (SNPs) between members of a species. It is a form of genotyping, which is the measurement of more general genetic variation." (Harbron S; Rapley R 2004)

Speed-up - The speedup of any computer algorithm is obtained by dividing the time taken to execute the algorithm in parallel by the time taken to execute the algorithm serially. (Lichtenberg et al., 2010)

TPL- "Task Parallel Library is a set of APIs that is present in System.Threading and System.Threading.Tasks namespaces of .NET framework."(Task Parallel Library, 2009)

LIST OF ABBREVIATIONS

- API- Application Program Interface
- BLAST- Basic Local Alignment Search Tool
- CLR- Common Runtime Library
- DNA- Deoxyribonucleic Acid
- FDR- False Detection Ratio
- GSEA- Gene Set Enrichment Analysis
- HDFS- Hadoop Distributed File System
- HPC- High Performance Computing
- iBAQ- Intensity Based Absolute Quantification
- I/O- Input/Output
- LCMS- Liquid Chromatography Mass Spectrometry
- NCBI- National Center for Biotechnology Information
- PEP- Posterior Error Probability
- SSD- Solid State Drive
- SNP- Single Nucleotide Polymorphism
- TPL- Task Parallel Library

ABSTRACT

Shah, Jigna. M.S., Purdue University, May 2015. Studying the Effect of Parallelization on the Performance of the Andromeda Search Engine: A Search Engine for Peptides. Major Professor: John Springer.

Human body is made of proteins. The analysis of structure and functions of these proteins reveal important information about human body. An important technique used for protein evaluation is Mass Spectrometry. The protein data generated using mass spectrometer is analyzed for the detection of patterns in proteins. A wide variety of operations are performed on the data obtained from a mass spectrometer namely visualization, spectral deconvolution, peak alignment, normalization, pattern recognition and significance testing. There are a number of software that analyze the huge volume of data generated from a mass spectrometer. An example of such a software is MaxQuant that analyzes high resolution mass spectrometric data. A search engine called Andromeda is integrated into MaxQuant that is used for peptide identification.

One major drawback of the Andromeda Search Engine is its execution time. Identification of peptides involves a number of complex operations and intensive data processing. Therefore this research work focuses on implementing parallelization as a way to improve the performance of the Andromeda Search Engine. This is done by partitioning the data and distributing it across various cores and

nodes. Also multiple tasks are executed concurrently on multiple nodes and cores.

A number of bioinformatics applications have been parallelized with significant improvement in execution time over the serial version. For this research work Task Parallel Library (TPL) and Common Library Runtime (CLR) constructs are used for parallelizing the application. The aim of this research work is to implement these techniques to parallelize the Andromeda Search Engine and gain improvement in the execution time by leveraging multi core architecture.

CHAPTER 1. INTRODUCTION

1.1 Introduction

This chapter contains an introduction to the research by stating the research question and then elaborating on the problem statement. This chapter also contains the scope of this research and its significance. In the end the chapter concludes by giving the assumptions, limitations and delimitations.

1.2 Research Question

How does parallelization effect the performance of Andromeda Search Engine: A probabilistic search engine for peptides?

1.3 Statement of Problem

The Bindley Bioscience Center at Purdue University Discovery Park has historically employed its Omics Discovery Pipeline for quantifying and identifying proteins. As per the National Cancer Institute, Office of Cancer Clinical Proteomics Research , “Proteomics is comprehensive study of a specific proteome, done on large scale that provides information on protein abundances, their variations and modifications, along with their interacting partners and networks, in order to understand cellular processes.”(What is Cancer Proteomics, n.d.)

Cancer is a major health issue, and cancer research had shown that early cancer detection can lead to better treatment and higher chances of recovery. Proteomics based techniques help in identifying the difference between the biomarkers of patients and healthy people. These results are used to design individual therapy that result in effective treatments. During a mass spectrometry based proteomics analysis for cancer detection, a large number of subjects are present and large amount of data is obtained. This data then undergoes a series of complex computations to get the final output. MaxQuant is one of the tools that is used for quantification and identification of proteins. For identification purposes, MaxQuant utilizes a search engine called Andromeda. This helps in analyzing large volumes of data in a simple and easy to understand workflow on a commodity computer. However the search process is still slow.

Therefore, this thesis determined whether the performance of the Andromeda Search Engine can be improved by leveraging a multithreading and multiprocessing architecture

1.4 Scope

In this research standalone Andromeda Search Engine was deployed on multiple cores using virtualization. The thesis work used Task Parallel Library (TPL) for the analysis of input files on a multi core architecture.

The code for Andromeda search engine is written in C# language. Hence implementing TPL, which is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces in the .NET framework 4.5, ("Task Parallel Library", 2009) was effective because it has been written specifically for the C# language.

TPL constructs were used to execute the functions and tasks concurrently to reduce the time of execution.

TPL uses the concept of a task that is a higher-level abstraction of a system's thread. The functions were remodeled as tasks and then multiple functions were executed concurrently. Finally the results from all the cores were collected and displayed.

This thesis work involved incremental parallelization. This means that the entire code was not re-written using a parallel programming approach. Instead certain parts of the code that involved a lot of loops or complex computations were parallelized first using TPL. Also dependencies between tasks were studied and then independent tasks were executed concurrently or else dependent tasks were pipelined together to get performance gain.

For parallelizing the Andromeda source code a combination of fine-grained parallelism and coarse-grained parallelism was used on a multi core architecture.

1.5 Significance

Past studies have shown that certain diseases like cancer can alter the structure of proteins in human beings. Thus the proteins in a healthy person and a cancer patient will be different. To identify these differences protein analysis is done. A quicker analysis would lead to early detection of cancer and thus increase the chance of recovery of the patients. Two major steps are involved in this protein analysis: Quantification and Identification. Both these steps are computationally intensive and process a huge amount of data. This data is then compared against a given protein database to identify the peptides and consequently the entire protein. If the time taken to process this data was

reduced by executing operations concurrently the performance can be improved. This was beneficial for this application as it involved complex processing of large amount of data.

1.6 Assumptions

The given assumptions were made for this research:

- The network between various cores and various nodes was assumed to be constant at all times.
- The underlying hardware of the cores and nodes has no effect on the parallelization strategies used for this research work.
- The effect of parallelization remained the same as we increased the file size.

1.7 Limitations

This research study had the following limitations:

- The research work was based on incremental parallelization. This means that the author did not rewrite the entire code with the aim of parallelization. Instead the author took snippets of the code and parallelized it.
- Only the most recent version of Andromeda Search Engine was used to incorporate the parallelization constructs into the code. Older versions were not touched upon.

1.8 Delimitations

This research study had the following delimitations:

- Andromeda is a part of the MaxQuant software suite that can also be used as a standalone software. This research work studied the effect of parallelization on the standalone version of the Andromeda software. It did not study the effect of parallelization on MaxQuant with Andromeda integrated into it.
- This experiment did not test the performance of the parallelized version for complex configurations and parameters of the search engine. Instead for performance evaluation it performed a basic search with default parameters.

1.9 Chapter Summary

This chapter contains an introduction to the research that was done for this thesis work. This chapter also explained the author's motivation to do this research and its significance. Also this chapter outlined the research question and the assumptions limitations and delimitations that applied to this research.

CHAPTER 2. LITERATURE REVIEW

This chapter contains a brief summary of recent advancements in the fields of protein analysis, code parallelization and Task Parallel Library pertaining to the scope of this research work. This literature review helps the author in understanding the existing methods pertaining to the research area and in formulating a robust methodology for the experiment.

2.1 Introduction

The advent of distributed programming has had a great effect on the development and implementation of various computational software. Distributed computing architecture makes use of parallelization to execute code on multiple cores and multiple nodes. This is done by concurrently executing multiple independent functions and by dividing the data that serves as the input to the various functions among various nodes. This improves the execution time of the software that is parallelized. A number of existing bioinformatics applications have already been parallelized using multicore and multi node architecture. Bioinformatics applications deal with huge amount of data and involve complex processing on those datasets. As a result techniques are required which lead to faster processing and are scalable. Bioinformatics applications involve intensive

data processing on huge datasets. This chapter contains information about parallelization of existing bioinformatics applications using distributed programming approach by leveraging multicore and multi node architecture Different paradigms used for parallel computing like OpenMP (Open Message Passing), MPI (Message Passing Interface) and MapReduce are also studied. With respect to mass spectrometry based proteomics Lewis (2012) found the following:

> For shotgun mass spectrometry based proteomics the most computationally expensive step is in matching the spectra against an increasingly large database of sequences and their post-translational modifications with known masses. Each mass spectrometer can generate data at an astonishingly high rate, and the scope of what is searched for is continually increasing. Therefore solutions for improving our ability to perform these searches are needed.

Thus there is a vast scope for parallelization in the Andromeda Search engine and significant time gains can be achieved.

2.2 Overview of the Andromeda Search Engine

Andromeda is an open source search engine that uses probabilistic scoring for searching and identifying peptides.

Cox et al. (2011) in their paper on Andromeda discuss the algorithm that Andromeda uses and how it can be integrated with MaxQuant as well as used as a standalone application. When Andromeda is used in an integrated environment with

MaxQuant it presents a simplified and pipelined workflow for the analysis of large datasets that can be easily used on a desktop computer (p. 69).

Andromeda uses a probability based search algorithm for searching for peptides that uses binomial distribution. It takes as input a peak list file that is obtained from the mass spectrometer and a parameter file that sets the parameters for the search. The output is a scored list of peptides. Andromeda search engine generates a scored list of peptides based on the matches with the fasta database. The peptide with the highest score is the best match. For performing the search, Andromeda uses indexing of this generated peptide lists instead of maintaining the entire peptide list in memory. These indices contain the location of the records with respect to the beginning of the file. The list can be very large and hence the indices can exceed the available memory; therefore instead of the index pointing to each record, the index entries point to a block of elements. These block of elements are contained in a file whose block sizes are chosen in such a way that the indices have a fixed size and fit in the memory. This approach is typically called a sparse index. A two-layered index structure is used to store protein list. The first layer is present in the primary memory and its entries point to the secondary index that is stored on the disk memory. This secondary index contains index entries that point to block of data stored on the disk. These blocks of data contain all the information about proteins. The protein list is stored alphabetically and the index and disk entries are sorted on the basis of increasing peptide mass. This results in quick retrieval of candidate peptides. After this the scoring algorithm is used to match the fragments to peptides. This algorithm is computationally very intensive and scores the peptides based on their

probability of matching to the fragments. The peptide with the highest score is the best match.

As seen from the discussion above, Andromeda Search Engine processes a large amount of data by using a computationally intensive algorithm, and hence it takes a long time to generate results. To improve the efficiency, the code needs to effectively use a parallel and distributed computing framework that will distribute the data on multiple nodes on which computations can be performed in parallel, and this in turn will improve the overall efficiency of the algorithm.

2.3 Improving the Performance of Bioinformatics Algorithms

Trelles (2001) gave the factors to be taken into account in order to improve the execution time of bioinformatics algorithms. Time complexity of the algorithm is the first factor that should be considered when trying to improve the performance of any algorithm. If the time complexity has already been considered then the next step is to consider code parallelization by using a distributed computing approach that leverages a multicore or multimode architecture. Parallel computing model has two important aspects: communication and granularity. Granularity in parallel computing model is of the can be achieved in the following different ways:

- Parallelizing the instructions at a hardware level.
- Parallelizing using compiler directives. This is called software level parallelism and is achieved by dividing the data among the various available cores and nodes and then executing individual instructions on them.

- Analyzing the code to determine the code snippets that can be parallelized such that multiple instructions run in parallel.

The final type that involves parallelization of multiple instructions can be done at two different levels: coarse grained parallelism and fine-grained parallelism. In fine-grained parallelism the instructions within a function are parallelized to run concurrently whereas in coarse-grained parallelism multiple independent functions are executed concurrently.

Communication is another key aspect of parallel computing model. Depending on how processes access the memory communication requirements may vary. Processes can communicate with each other using semaphores, messages, pipes, signals or shared memory. (Stallings, 1992). Out of these mechanisms semaphores, pipes, signals and shared memory are used to perform inter process communication when there is a shared memory architecture. For communication between processes running on a distributed memory system messages are passed over the network connecting the distributed system.

The way network is structured can also effect the communication between nodes in a parallel computing. Jiuxing et al. (2003) analyzed the effects of varied network structures on the performance of parallel computing systems.

Analysis of different bioinformatics applications reveals some prominent types of algorithms. Majority of these algorithms are based on sequence- database searching. In Andromeda search engine the search for peptides identification is made against a fasta database. This is a huge database and a lot of complex operations are involved which makes this process time consuming. In this case parallelizing the application can result in

substantial performance gain. This is done by finding the code snippets that are computationally intensive and executing them on multiple cores concurrently. This leads to efficient load division among various cores and nodes and hence an improvement in the execution time can be achieved.

2.4 Application of Parallelization Paradigms in Bioinformatics Algorithms

Message Passing Interface (MPI) has found number of applications in the field of bioinformatics. One such application of MPI is Basic Local Alignment Search Tool (BLAST) that contains algorithms for searching against a sequence database. In this application a protein or a DNA sequence is searched against a database that contains previously known sequences. These databases against which the search is made are huge and the size keeps increasing exponentially. Hence BLAST algorithms are slow and it is important to improve their performance. Chi et al. (1997) made the initial efforts to parallelize BLAST by using multithreading. In this new algorithm they split the database between multiple threads for execution. This splitting was however confined to a single node and all the threads worked on the same node. However due to increase in the size of the database storing the complete database on one node was not feasible.

Thorsen et al. (2007) tried to parallelize BLAST by implementing it using distributed memory. In this algorithm the sequence being looked for or the database against which the search is made is partitioned across the multiple participating nodes where each node is independent, has a separate memory and is connected to the other nodes by some interconnection network. They called this implementation mpiBLAST. This implementation used Message Passing Interface for communication between the

nodes. This implementation uses a master –slave architecture wherein all the I/O operations are executed either by the master or the slaves. If the master alone performs the I/O operations then it should have sufficient memory that is expensive to get. Also this will hamper the performance of master and other tasks that the master does like distribution of data across nodes and load balancing will be effected. Thorsen et al. (2007) found out that if all the workers together perform the I/O operations the load will be equally distributed among the slaves and this will result in an improved performance. The workers write the results at a certain offset that is communicated to them by the master. MPI-IO (Corbett et al., 1995) is used to implement parallel I/O. This is an example of application in which tasks are run on multiple cores and multiple nodes. MPI is used for communication in this setup for communication between virtual nodes that are actually the cores of one machine.

2.5 Parallelizing C# Code

"The Task Parallel Library (TPL) is a set of public types and APIs in the System.Threading and System.Threading.Tasks namespaces" (Task Parallel Library, 2009). TPL has simplified the way developers parallelize applications and add concurrency to them thereby making developers efficient. TPL constructs are scalable and are designed in such a way that they can dynamically add concurrency to the number of available cores. TPL also handles a number of low level details like division of work between the various cores, scheduling threads on the ThreadPool, task state management,

cancellation and exception handling. By performing these tasks TPL ensures that programmer can focus on the performance, robustness and correctness of the code.

Although TPL is the preferred way to write code which uses multithreading and follows a distributed and parallel computing model on a .NET framework, it should be realized that parallelizing code is not always beneficial. Parallelizing any code involves a lot of overhead and the tradeoff between the performance gain and overhead should be analyzed before parallelizing any code. For instance it is not beneficial to parallelize a loop that does not perform complex operations, runs a small number of iterations or processes a small amount of data. Also with parallelization the program execution becomes more complex.

The Task Parallel library employs both data parallelism and task parallelism. In data parallelism the input data is partitioned among various cores or nodes so that same operations can be performed on that data. This is done by creating multiple threads that operate simultaneously on the partitioned data.

The basic concept of the Task Parallel library is 'task' which is a higher-level abstraction of a thread. ("Task Parallel Library", 2009). For implementation of Task Parallel library constructs a task is considered as an asynchronous operation. When one or more tasks are executed concurrently it is called task parallelism. Using tasks has the following benefits:

- Tasks are queued for execution on the ThreadPool. This ThreadPool is highly advanced in the sense that it does load balancing to generate maximum

throughput by adjusting the number of tasks to suit the number of threads. Since tasks are lightweight any number of tasks can be created to achieve fine-grained parallelism. All these things lead to better efficiency and scalability of the system.

- Tasks have an advanced set of APIs associated with them which perform a number of operations like scheduling, continuations, cancellations, waiting, detailed status and exception handling. Thus better control is available with a task.

For the reasons mentioned above, Task Parallel Library is preferred for writing parallel code which employs multithreading in the .NET framework.

Another important factor to consider while parallelizing any algorithm is Amdahl's law (Rogers 1985, p. 226). This law is used in parallel computing for the prediction of maximum theoretical speedup that can be attained when using multiple processors. According to this law the maximum speedup that can be attained by a parallel program is bound by the sequential part of the program. Hence the time taken by sequential part of the program places an upper limit on the speedup that can be attained.

2.6 Hadoop in Bioinformatics Applications

Lee et al. (2009) stated that an open source implementation of MapReduce is Apache Hadoop. It can be installed on a commodity Linux server and is used when analyzing large-scale distributed data. The commodity servers can be used as it is without any change in the configuration. Shvachko et al. (2010) stated that Hadoop resides on top of HDFS (Hadoop Distributed File System) that is used to access data. It uses a Java based API or Python scripts to run and execute codes. In HDFS data is partitioned and replicated among the various compute nodes. This replication ensures fault tolerance in

case any of the nodes go down. In that case the data could be pulled from any other node on which it was replicated. Hadoop utilizes data localization for quicker data access and computation thus improving data bandwidth and performance. The tasks using Hadoop are independent of each other except for the mappers whose output goes into reducers under the control of Hadoop. In case a node fails the computations being executed on it can be restarted and executed on any other node. Thus Hadoop provides a very simple framework that is very reliable, scalable, robust, fault tolerant, where dataflow is implicit and requires no coding. The following paragraphs present a few examples of a few bioinformatics applications implemented using Hadoop.

Taylor et al. (2009) came up with the Cloudburst software that was used for SNP genotyping. In this next generation short read data was mapped to a reference genome using Hadoop. This was the first paper that outlined Hadoop application in bio-informatics. Their study focused on how Hadoop provides a reliable, fast and effective way to process huge datasets.

Schatz et al. (2009) developed algorithms that analyzed next generation sequence data using Hadoop. In their paper they elaborate on the following tools:

- There are a number of tools like Crossbow that use Hadoop for genome sequencing and SNP genotyping. This is very similar to proteomics identification that is performed by the Andromeda search engine.
- Myrna is another algorithm that is used for calculating differential gene expression from large RNA-sequence data sets. This algorithm also has some common aspects with the Andromeda search algorithm.

The study done at Indiana University by Qiu et al. (2009) analyzed a number of cloud-based solutions like Apache Hadoop, Microsoft Azure and Dyrad. These technologies were used for implementing datasets that were doubly data parallel (all pairs). The input data for Andromeda search engine uses the same kind of data. The studies found that these cloud technologies will become preferred option for bioinformatics applications because of the flexibility provided.

Gene set enrichment analysis (GSEA) is a method for testing association between a gene expression profile and a subset of genes. The method can also be reversed to test for interesting expression profiles given a subset of genes. Gagerro et al. (2008) have implemented BLAST and GSIA using Hadoop and have reported their work as very positive with MapReduce being a very versatile framework. They found Hadoop particularly impressive because of its scalability, reliability and fault tolerance.

Matsunaga et al. (2008) compared a Hadoop based version of NCBI BLAST2 algorithm called CloudBlast with mpiBLAST which is a leading parallel version of BLAST and it was found that Hadoop based implementation was advantageous in terms of failure management, job scheduling and data partitioning. This again can be attributed to the replication of data and the presence of independent jobs characteristic of the Hadoop architecture and HDFS.

Studies done by Leo et al. (2009) show that Hadoop provides a robust and scalable environment for all types of applications that is compute intensive, data intensive or a combination of both. The MapReduce paradigm used by Hadoop provides some flexibility in the sense that the researcher could simply use only the Map part, only the Reduce part or both and if needed multiple maps and reduce can also be chained together.

When examined for skewed and randomly distributed datasets, Hadoop proved to be very scalable. According to Lewis et al. (2012), "A sequence search engine called Hydra has been specifically designed to run on distributed computing framework i.e. MapReduce. The search engine uses the K-score algorithm and produces comparable output as the original implementation."

2.7 Conclusion

A summary of how various parallel computing models have been used to implement parallelization in various software and algorithms in recent times is presented in this chapter. The techniques studied above provided a basis for formulating a methodology and determining a way to approach the research that was studying the effects of parallelization on the performance of the Andromeda Search Engine.

CHAPTER 3. METHODOLOGY

This chapter outlines the research framework, data sets and methodology used for running experiments in this thesis.

3.1 Apparatus/Details

The server was configured as follows for this experiment

Processors

- AMD Opteron™ Processor 6172 @2.10 GHz
- Processor Speed: 2.10 GHz
- Processor Socket: 4
- Processor Cores per Socket: 1
- Logical Processors: 4
- Hyperthreading Enabled Processors

System

- System Manufacturer: VMware, Inc
- System Model: VMware Virtual Platform
- BIOS Version: Phoenix Technologies LTD 6.00
- Release Date: 4/14/2014

Memory

- Installed Physical Memory (RAM) 4.00 GB
- Total Physical Memory 4.00 GB
- Total Virtual Memory 8 GB

Server Middleware

- VMware® ESXi™ 5.5

Server Software

- OS Microsoft Windows Server 2008 HPC Edition
- Version 6.1.7601 Service Pack 1 Build 7601

Compute Node Configuration

Processors

- AMD Opteron™ Processor 6172 @2.10 GHz
- Processor Speed: 2.10 GHz
- Processor Socket: 4
- Processor Cores per Socket: 1
- Logical Processors: 4
- Hyperthreading Enabled Processors

System

- System Manufacturer: VMware, Inc
- System Model: VMware Virtual Platform
- BIOS Version: Phoenix Technologies LTD 6.00
- Release Date: 4/14/2014

Memory

- Installed Physical Memory (RAM): 4.00 GB
- Total Physical Memory: 4.00 GB
- Total Virtual Memory: 8 GB

Compute Node Software

- OS Microsoft Windows Server 2008 HPC Edition
- Version 6.1.7601 Service Pack 1 Build 7601

For baselining Andromeda experiments were also executed on a standalone system that belonged to the D.A.T.A lab of the Computer and Information Technology department at Purdue University.

Standalone System Configuration

Processors

- Intel® Core™ i7-45000 CPU @ 1.80GHz, 1801 MHz
- Processor Speed: 1.80 GHz
- Processor Socket: 1
- Processor Cores per Socket: 2
- Logical Processors: 2
- Hyperthreading Enabled Processors

System

- System Manufacturer: Dell Inc.
- System Model: VMware Virtual Platform

- BIOS Version: Dell Inc. A01
- Release Date: 7/24/2013

Memory

- Installed Physical Memory (RAM): 8.00 GB
- Total Physical Memory: 7.71 GB
- Total Virtual Memory: 8.96 GB

System Software

- OS Microsoft Windows 8.1
- Version 6.3.9600 Build 9600
- CPU 4vCPU

3.2 Conditions

This program ran on a cluster whose specifications are provided in the details section. The performance of the program was affected by the following factors.

- Size of the peptide database. This covers
    - The size and the number of proteins considered
    - The enzyme used for cleavage
    - Number of variable modifications and fixed modifications.
- The size of the data sets used.
- Available computer resources.

Andromeda Configuration allowed one to add in new protein databases, as they are updated, new or unusual modifications, and different enzymes or combinations of

enzymes. This enabled the search engine Andromeda to interrogate the MS data the way the author required it to be done.

Andromeda requires three important specifications

- Protease that the proteins were cleaved.
- A sequence database to search against.
- Modifications or labels present.

While using Andromeda search engine the above mentioned parameters need to be specified like labels and modifications settings. These settings describe the chemistry done to the proteins. Any chemistry done which may have an effect on mass must be included in these settings. Modifications that might be possible that is Variable Modifications were not considered. Modifications that must occur are Fixed Modifications and the database was searched only with this modification. The enzyme chosen for digesting the protein was trypsin. Labels were not specified since no labeling strategy is used. Multiplicity is selected as 1. This did a label free search. The First Search 20ppm and Main Search 6ppm were left as they were- this was for the purpose of Andromeda identifying the maximum number of peptides for mass and retention time calibration, and then to refine the results at the 'Re-Quantification' step. This was particularly effective since we were using a first search database. Missed cleavages accounted for the enzyme not being 100% effective- this is common and the default setting was the accepted tolerance for this. The 'Type' setting was machine dependent. The Author plans on using an Exactive, therefore All Ion Fragmentation was selected. A double digest with 2 enzymes was not used so Separate Enzyme for First Search was not

checked. The Andromeda config files were not modified for labels and modifications and the standard versions were only used.

The machine specific settings are found in MS-MS sequences field. The defaults found in the top panel were fine for the majority of searches. The fixed modifications author used was Carbamidomethyl. The .fasta files for the database to be searched was the one that is supplied with MaxQuant. Human and mouse first search .fasta files are provided with MaxQuant. These contain commonly seen proteins which are expected in samples, and were used in the first search as calibration points for the more exact Main Search and re-calibration steps later on.

The values in the top panel described the stringency of the searches performed, such as False Discovery Rate (FDR), number of peptides required for identification, and Posterior Error Probability (PEP) score cut off. The author used default values for False Discovery Rate (FDR), number of peptides required for an identification and Posterior Error Probability (PEP) score cut off for setting the stringency of the searches performed since they were a good standard set up. 'Filter Labelled amino acids' box was deselected since the Author is doing label –free. Second peptides looks for mixed spectra and is very useful for further peptide identification, this setting was left as it is. The author was not using any variable modifications hence those boxes are left unchecked. The settings in the Protein Quantification panel are appropriate for most analyses and were left as the defaults. The Misc. panel defaults are also appropriate for most analyses and were left as it is. The iBAQ Quantification (Intensity Based Absolute Quantification) was used for label free quantitation (calculating intensities from peak intensities, including isotopic peaks, with some additional calculations) and can match retention times between samples.

Re-Quantify was also used, as this allows for a second peak finding to occur after protein identification has been done.

For Andromeda configuration modifications the composition of modifications (C, H, N, O etc.) was entered from the drop down list, specifying the number of molecules with the count arrow on the right. Next the author specified which amino acid was affected by the modification- in the specificity tab. This amino acid was trypsin. There were no neutral losses or diagnostic peaks associated with the modifications. Correction factors that were given in the commercial information for iTRAQ reagents were inserted in the Correction factors panel. The default proteases found in Andromeda were used since they cover most of the experiments.

3.3 Procedure

The author wanted to test the program against larger searches so as to validate the performance and speed with respect to the ability to process large loads and scalability. Sample data sets used for the experiments ranged from 100 to 1000 MB. The serial version of the Andromeda search engine source code ran on a single core on the standalone machine whose configuration is provided in the Apparatus/Details section. The time taken for the execution of the serial version on single core was used for benchmarking.

The author had used a combination of fine-grained parallelism and coarse-grained parallelism. Coarse-grained parallelism involved parallelizing multiple functions and chaining them together. Fine-grained parallelism focused on parallelizing individual

functions. It concentrated on the loops and tried to parallelize the loops by executing the iterations simultaneously.

The author had used task parallelism for parallelizing the code of the Andromeda search engine. The code of the Andromeda search was examined to find parts that contain nested loops or were otherwise computationally intensive. These snippets were appropriate for parallelization. The author used the following strategy to determine which code snippets to parallelize. First of all hotspots were found in the code. These hotspots were the parts of the code that contain loops (that is, For loops and While loops) and take significant time to run. To determine hotspots execution timing of functions were measured. The ones that took similar time were executed together; this lead to efficient utilization of cores. TPL constructs were used to divide the indices of the for loop into chunks and then to running these chunks concurrently on the multiple cores. Each core executed the iterations that were assigned to it. Similarly the iterations were also divided among various nodes. One node was assigned the head node and it performed the task of dividing the input between the various nodes. Each node then processed the input assigned to it. After the completion of the execution each node sent its result back to the head node. The head node combined all the results and displayed. The various nodes communicated with each other and with the head node using TPL constructs which used MPI.

After analyzing the code for hotspots the author found that there were mostly for loops and hardly any while loops. Therefore the author decided to use the parallel.for construct. The author found that the computationally intensive iterations of certain for loops were independent of each other. Thus in that case the author used parallel.for with a

custom partition and converted the for loop range into chunks and then processed them. This led to optimization according to the number of available cores.

The author also used imperative task parallelism to optimally parallelize the code. Under this, tasks were created pertaining to individual functions that had no dependencies and then these tasks were executed in parallel. This involved much less overhead and gave better performance. Also using tasks parallelism had many other advantages; for instance, individual tasks could be chained to each other such that the result of one can be used as the input to other. The author was careful to mention critical sections when using task parallelism and chaining.

An extension of chaining is pipelining. While parallelizing the code the author had extensively used pipelining. Pipeline used the concept of producer consumer wherein the producer produced results that are in turn used by the consumer. The advantage of using pipelining was that as soon as the producer gave a result the consumer started working on that result. It need not wait for the producer to complete its execution thus saving time. While pipelining the pipeline was divided into multiple stages. For optimal execution the number of stages in pipeline should be equal to the number of cores. All these stages were executed in parallel.

When using pipelining the author needed to synchronize concurrent tasks. This was because a group of tasks run a series of phases in parallel but each new phase has to start after all the other tasks finish the previous phase. This cooperative work was synchronized with an instance of barrier class. Each phase requires synchronization between tasks, a Barrier object prevented individual tasks from continuing until all tasks reach barrier. Each task in the group called participant signals its arrival at the Barrier in

each given phase and implicitly waited for all other participants to signal their arrival before continuing. The same barrier instance was used for multiple phases.

The author broke up the problems in such a way that synchronization became explicit not implicit. The functions were broken in such a way that they can work independently so as to avoid explicit synchronization and make code more efficient and scalable. This is because explicit synchronization, atomic operations and locks always add an overhead, require processor time and reduce scalability. If they can be avoided better speedup can be achieved.

Andromeda search engine matches the peak list file against the fasta database. The database used for this experiment is approximately 2.5GB is size. As this database was not too big in size the author stored this database on every node instead of distributing it across various nodes or storing it only on the head node.

Exception handling was done using timeouts and cancellations when working with barriers and other synchronization mechanism because an error in the code or an unpredictable situation can generate a task or a thread that will be waiting forever.

The experiments were executed on the High Performance Computing cluster at the CIT department of the Purdue University. These experiments were benchmarked against the experiments ran on the standalone machine. The experiments involving the parallel version of the code ran on the Windows Server HPC cluster at Purdue University. They were called ParAndromdeda-I. The experiments that ran on the standalone system were called ParAndromeda-II. The time of execution for both the variants of the experiment ParAndromeda-I and ParAndromeda-II were recorded. The document henceforth uses the following conventions:

- pan1-I denotes the time of execution time for the serialized code of the Andromeda search engine executed on the standalone system using ParAndromeda-II,
- pan1-II denotes the time of execution of the parallelized version of the Andromeda code executed on the cluster using ParAndromeda-I,

The author measured pan1-I values by changing the number of cores as 4,8,12 and 16. The effect of varying input sizes was also analyzed by using files sized 100MB, 200MB, 400MB, 600MB, 800MB and 1000MB. In order to verify the correctness of the experiments the outputs from the serial and parallel versions were verified and matched for all the combinations of number of cores and file sizes.

The experiment was done 15 times for every combination of input file size and the number of cores. These results were recorded in a tabular format which maps the execution time to the number of nodes for a given file size.

3.4 Method

The section contains information about the data, its source and the manipulations done.

3.4.1 Population

The population consisted of a collection of MS-MS proteomics spectra present in .wiff.scan format.

3.4.2 Sample

The sample datasets consisted of MS-MS datasets selected from ftp://bpcore@ftp.bbc.purdue.edu which is maintained by the Bindley Bioscience Center at the Purdue University. The input was divided into different sizes of 100MB, 200MB, 400MB, 600MB, 800MB and 1000MB.

3.4.3 Data Collection

The MS-MS datasets of varying sizes described above were provided to both the parallel version and serial version of Andromeda Search Engine. For the parallel version the same input were provided to the parallelized code running on different number of cores. The execution time for all the combinations of input size and number of cores was recorded in a .CSV file.

3.4.4 Variables of the Experiment

The independent variables of the experiments were

- Task Parallel Library
- Total number of cores used
- Size of the input

The dependent variable of this experiment was the time of execution of the parallelized variant of the code of the Andromeda Search Engine. It was measured in seconds and this unit was used throughout the research work.

3.4.5 Hypothesis

H_o: Parallelizing Andromeda source code does not improve the execution time of Andromeda Search Engine.

H_a: Parallelizing Andromeda source code does improves the execution time of Andromeda Search Engine.

3.4.6 Data Analysis

Finally, a paired t test was performed in order to test the above stated hypothesis. As per McDonald (2009) a paired t test can be used if we need to analyze the before and after effects of certain treatment on the given group. In this experiment the author measured how parallelizing the Andromeda search engine effects the time of execution for different file sizes. The p-values were calculated by comparing the execution times for the parallel version and serial versions.

In order to perform a t-test certain conditions should be met. These are:

- There should be one dependent variable, which was the execution time in this experiment.
- There should be one categorical independent variable which was the number of cores.
- The dependent variable should be normally distributed. This was tested using chi-squared test. The results of the chi-squared test indicated that the dependent variable was indeed normally distributed.

After fulfilling all the above conditions a paired t-test was performed.

The results of the experiments were summarized in the form of various tables and graphs. The graphs showed the relationship between the execution time and the number of cores and the input file size. The graphs also showed the speedup achieved by the parallelized code when compared to the serial code for all possible combinations of number of cores and input sizes.

3.5 Threats/Weaknesses

The searching algorithm that Andromeda used generated a lot of false positives. Using Task Parallel Library to implement this algorithm increased the processing speed but had no effect on the false positives generated. If the false positives generated were to be reduced the algorithm needs to be changed which is beyond the scope of this thesis work. The network speed was also a bottleneck in this approach. Since this research work was implemented on a distributed computing framework, data transfer between the nodes was an integral part. Hence if the network slows down or fails it will affect the overall performance of the system.

The Andromeda code that was available with the author had certain gaps in it. Thus the time gain achieved by parallelizing can change when the complete code is parallelized and executed.

The search performed by author for this experiment was very basic and did not deal with complex configurations. Therefore when settings are changed and some complex scenarios are taken into account with variable modifications, labels and multiple enzymes there might be a difference in the results and some fine-tuning in the code might be required.

CHAPTER 4. DATA ANALYSIS

This chapter presents the evaluation of the performance and correctness of the parallelized version of the Andromeda Search Engine according to the findings of different matrices. It also contains summary of execution of the serial version on one core and the parallel version on 4,8,12 and 16 cores.

4.1 Correctness

Two different versions of the Andromeda search engine source code were used as per the methodology

- Serial
- Parallel

The execution of the serial version on a single core was used for the creation of the baseline. The correctness of parallel version of the Andromeda search engine was determined by comparing its output with the serial version. Hence, an output folder was created for each core and file size combination. The outputs converted into .txt file are stored in these folders. For correctness the outputs from the serial version and the parallel versions should match. The outputs were found identical upon comparing the output folders for all the versions.

4.2 Performance

The experiments for the parallelized version of Andromeda, were executed according to the methodology discussed in Chapter 3. The author changed the number of cores used for executing the parallel version. This was achieved using Windows HPC server wherein only the required number of cores were active and connected to the server. These number of cores were incremented or decremented as per the requirement. The standard deviation was calculated by running the experiments multiple times, 15 times to be exact as mentioned in chapter 3.

4.2.1 ParAndromeda-I Experiments

Parallelized version of the Andromeda search engine code was run on Windows HPC-cluster on 4,8,12, and 16 nodes. According to the methodology discussed in chapter 3, the time taken to complete the execution of the parallelized code was noted and is called pan-I. Table 4.1 contains the values for pan-I. Serial code for the Andromeda search engine is executed on a single core for the creation of baseline. Since the code was not completely available the time in the table essentially represents the time taken to complete the execution of the available code in both parallel and serial versions. When a sample of size 600 MB is executed on a single core it takes 854 seconds. This serves as our baseline. However when parallelized version of the code is executed on 16 nodes it takes 224.981 seconds for a file of 600 MB.

Table 4.1 Parallel Andromeda experiments – Time of execution for the samples on varying core sizes

Sample Size/ Cores	Baseline	4	8	12	16
100 MB	84	44.001	34.123	23.453	23.412
200 MB	169	86.225	66.864	48.356	46.364
400 MB	435	227.242	172.983	133.432	115.223
600 MB	854	438.6434	332.4893	263.298	224.981
800 MB	1752	901.345	689.340	535.893	474.256
1000 MB	3296	1679.422	1323.364	977.751	893.513

Figure 4.1 represents the execution time for pan-I experiments on different number cores for varying sample sizes. In the graph the pan-I values are plotted against the various sample sizes. Different number of cores are represented using different colored bars. The right hand side of the graph shows the color mapping used. Appendix B contains the detailed summary and values for average and standard deviation for all the core and sample size combination.

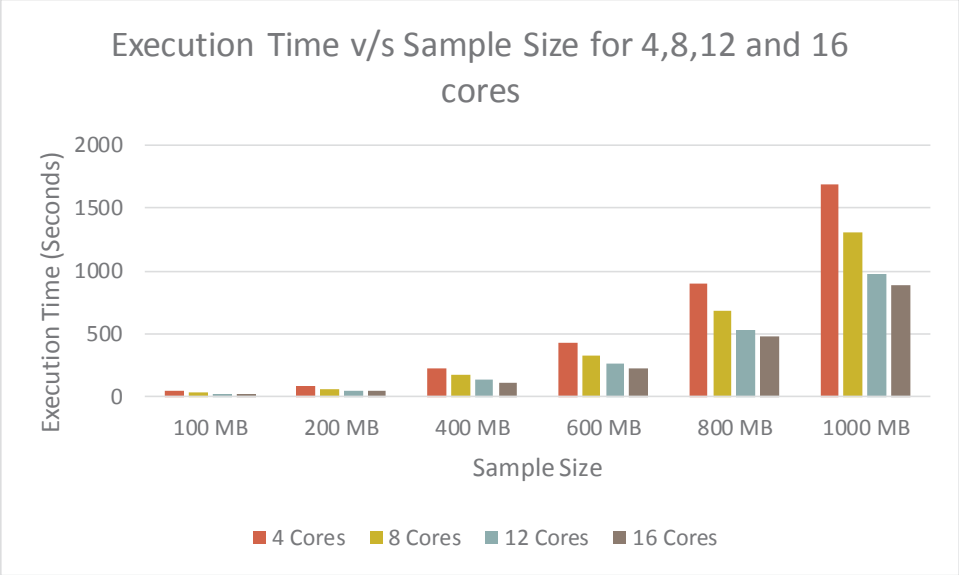

Figure 4.1 Time of execution for the samples on different number of cores in Parallel version of Andromeda

The table 4.2 shows the speedup achieved for different file sizes on different number of cores. As is evident from the table, file sizes do not have much effect on the speed up. However speed up increases significantly on increasing the number of cores. Speedup values helped the author to evaluate the gain in performance as compared to the serial code. The graph 4.2 which maps speedup vs. the number of cores helped the author understand how scalable the parallelized version of the Andromeda source code is. An average speed up of 3.5 was obtained when the parallelized version of the Andromeda code was run on 12 cores.

Table 4.2 Parallel Andromeda experiments – Speedup for the samples on varying core sizes

Sample Size/Cores	Baseline	4	8	12	16
100 MB	1	1.909	2.464	3.227	3.706
200 MB	1	1.925	2.572	3.370	3.863
400 MB	1	1.921	2.544	3.271	3.767
600 MB	1	1.954	2.568	3.255	3.765
800 MB	1	1.947	2.546	3.266	3.671
1000 MB	1	1.952	2.528	3.365	3.690

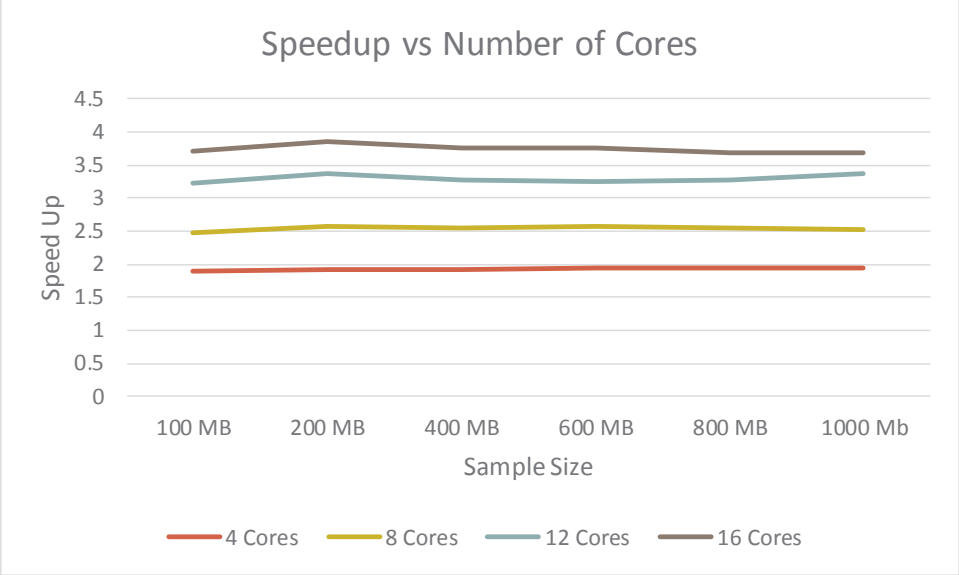

Figure 4.2 Speed Up for the samples on different number of cores in Parallel version of Andromeda

4.3 Statistical Analysis

The data obtained by performing the above experiments was then analyzed to determine its statistical significance. This was done using T-test as mentioned in section 3.4.6. The result of that statistical analysis is presented in the tables below.

Table 4.3 Statistical Significance for Various File Sizes for 4 cores

Implementation Name	Sample Size	p-value	t-value
Parallel Andromeda	100	<0.0001	11.765
Parallel Andromeda	200	0.0030	8.1045
Parallel Andromeda	400	0.0002	8.3364
Parallel Andromeda	600	0.0001	8.5882
Parallel Andromeda	800	0.0002	8.3247
Parallel Andromeda	1000	0.0020	8.5938

Table 4.4 Statistical Significance for Various File Sizes for 8 cores

Implementation Name	Sample Size	p-value	t-value
Parallel Andromeda	100	<0.0001	11.985
Parallel Andromeda	200	0.0002	8.3125
Parallel Andromeda	400	0.0030	8.1032
Parallel Andromeda	600	0.0040	8.0796
Parallel Andromeda	800	0.0003	8.2248
Parallel Andromeda	1000	0.0001	8.5874

Table 4.5 Statistical Significance for Various File Sizes for 12 cores

Implementation Name	Sample Size	p-value	t-value
Parallel Andromeda	100	<0.0001	11.604
Parallel Andromeda	200	0.0040	8.0865
Parallel Andromeda	400	0.0002	8.3263
Parallel Andromeda	600	0.0001	8.5677
Parallel Andromeda	800	0.0003	8.2133
Parallel Andromeda	1000	0.0030	8.0943

Table 4.6 Statistical Significance for Various File Sizes for 16 cores

Implementation Name	Sample Size	p-value	t-value
Parallel Andromeda	100	<0.0001	11.897
Parallel Andromeda	200	0.0002	8.3142
Parallel Andromeda	400	0.0001	8.5612
Parallel Andromeda	600	0.0002	8.3203
Parallel Andromeda	800	0.0004	8.0871
Parallel Andromeda	1000	0.0030	8.1024

As can be seen from the above tables 4.3, 4.4, 4.5 and 4.6 the p-values for all the combination of file sizes and core sizes are less than alpha (0.01). Hence it was proved that the result is statistically significant and hence the null hypothesis that Parallelization has no effect on the performance of the Andromeda Search engine was rejected.

4.4 Summary

The graphs and tables used in this chapter provide an adequate summary of the result of the research work. It further contains the trends followed by the data with respect to the metrics used to measure the performance. It analyzes the correctness and performance of the parallelized version of the Andromeda search engine by studying the time taken and speed up obtained for all the core and sample size combinations.

CHAPTER 5. CONCLUSIONS, DISCUSSIONS AND FUTURE DIRECTIONS

This chapter contains the findings of the research work. It also provides in brief detailed discussion regarding the future scope and extension of this research work.

5.1 Conclusions

In this thesis the author had implemented a parallelized version for the Andromeda search engine that leveraged multi-core architecture in order to improve the performance of the Andromeda Search Engine. The study primarily concentrated on incremental parallelization of the source code by determining the computationally intensive and parallelizable parts of the source code and then using parallel constructs based on TPL to parallelize the same.

The results obtained by parallelizing the source code of Andromeda showed that execution time was improved with increasing number of cores. An analysis of the results show that there was a speedup of about 2 times was obtained for 4 cores, 2.5 times for 8 cores, 3.5 times for 12 cores and little under 4 times for 16 cores.

An analysis of the Andromeda search engine determined that significant parallelization could be achieved in the loops. Processes and threads were synchronized across loops. Other than this tasks were chained together to be executed in a pipeline. Dependencies between tasks were removed and independent tasks were executed concurrently. To synchronize all these operations an instance of the barrier class was used.

To resolve dependencies between tasks Public variables and Private variables were used. The variables which were used in a single task were declared using Private construct in C#. The variables shared between multiple tasks were declared using Public construct of C#. The communication between tasks and that of shared variables was taken care of using TPL constructs.

5.2 Discussion

As mentioned earlier barrier class was used to synchronize operations between the Using a barrier class had certain drawbacks with the most important one being that it negatively affected the performance of the code. This is because all the functions for which the barrier was set had to wait till the data from the previous stage was received before proceeding on to the next stage. This meant that if a certain stage completed its execution it still had to wait for the remaining stages to finish their execution before going forward. In this way the performance of the parallelized code was limited by time taken by the slowest part of the algorithm.

It was observed that there is a significant improvement in execution time from 1 core to 4 cores and from 4 cores to 8 and 12 cores. However the improvement from 12 cores to 16 cores was not that significant. Although each part of the code that had been

parallelized can be scheduled either statically or dynamically the author had not used dynamic scheduling instead static scheduling was used. Also the division of chunks for the for loop indices and the number of tasks pipelined together was such that the code had been optimized for 12 cores. Another reason for this was that when implementing pipelining the maximum number of stages in which the author was able to divide the pipeline into was 11. All the stages were executed concurrently on cores. Since the number of stages is close to the number of cores that is 12 therefore the code was optimized for 12 cores and hence not much improvement was observed as we increased the number of cores from 12 to 16. This was as per Amdahl's law which was discussed in section 2.5. As per the law there is a limit to the amount of speedup that can be attained. The speedup is limited by the serial part of the code which was also true for this case since until the serial code completed its execution and returned the result the parallel part of the algorithm could not start its execution.

Another application of Amdahl's law that can be seen in this thesis is the speedup achieved per core. The speedup achieved for 4 cores was about 2 times which gave 0.5 times per core, speedup achieved for 8 cores was about 2.5 times which meant about 0.25 per core, for 12 cores the speedup was about 3.5 times which equaled a little over 0.25 per core and finally for 16 core the speedup was under 4 times and hence a speed up of less than 0.25 times per core was achieved. This was also evident from the speedup obtained for various number of cores that is 2 times for 4 cores but the speed up obtained for 12 cores is 3.5 and for 16 cores is 4 which is not proportional to the speedup obtained for 4 cores. So although the speedup increases as we increase the number of cores the speedup achieved per core decreases which is in accordance with Amdahl's law which

says there is only so much parallelization that can be done in an algorithm. By extrapolating these results we can safely say that as we keep on increasing the number of nodes after a certain point there will be no speedup and the graph between number of cores and speedup will flatline and become constant.

For this experiment a virtualized environment is used. As a result the interconnection network between the nodes did not play a big role since all of the nodes resided on the same virtual network. This lead to a better execution time and higher performance gain. As mentioned in section 3.3 the fasta database against which the sequence was matched was stored completely on every node. This saved the time required by the nodes to access the data present on other nodes which is done by using TPL constructs for communication which are based on MPI. Both of the above mentioned factors reduced the effect of the interconnection network between the nodes on the performance of the Andromeda Search Engine.

The search done by the Andromeda search engine for the purpose of this research work was kept very simple. Complex configurations of the parameters in the Andromeda search engine were not explored. The author decided to use the basic and default parameters because most of results after the first run can be obtained with these parameters and there is seldom any need for running the search using the complex combinations.

In order to study the practicality of implementing this software the author did a cost benefit analysis. As per this analysis the author calculated the total cost of this implementation as a sum of the money spent on hardware and the money spent on skills for developing and deploying this algorithm. The benefit was the amount of time saved

which the author converted into monetary value as well. On performing the calculations the author found that it would take close to 75 weeks to break even and any return on investments would come after that. In order to reduce the amount of time taken to break even the speedup obtained needs to be at least doubled. That is the maximum speedup obtained in this experiment was about 3.8 times. The value of speedup needs to go up to about at least 8 times to attain a practical breakeven point such that implementation of this algorithm becomes feasible and profitable.

The author hopes that this attempt to improve the performance of the Andromeda search engine will have benefits in the field of cancer detection and recovery. Using the parallelized algorithm and further improving on it to get faster results will help in identifying the cancer patients and with this identification made in good time the chances of recovery of the patients will improve.

5.3 Future Directions

The scope of this research focuses only on the performance gain of the Andromeda Search Engine. However given the improvement in execution achieved by parallelizing the Andromeda code parallelization in other stages of the MaxQuant software can also be considered. This research work does not consider the effect of the interconnection network. A study evaluating the effect of the network on the performance gain can also be included in the future scope.

The author had used default configuration in Andromeda for searching and scoring peptides. This configuration can however be changed as per specifications to

individualize search. In future various configurations can be tried and performance can be measured.

5.4 Summary

Most important finding of the research work were presented in this final chapter. It also included a discussion section and some recommendations for future improvements and work on this research.

LIST OF REFERENCES

LIST OF REFERENCES

Cox, J., Neuhauser, N., Michalski, A., Scheltema, R. A., Olsen, J. V., & Mann, M. (2011). Andromeda: a peptide search engine integrated into the MaxQuant environment. *Journal of Proteome Research*, *10*(4), 1794-1805.

Dai, L., Gao, X., Guo, Y., Xiao, J., & Zhang, Z. (2012). Bioinformatics clouds for big data manipulation. *Biology Direct*, *7*(1), 43.

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM, 51*(1), (pp. 107-113).

Dean, J., & Ghemawat, S. (2010). MapReduce: A Flexible Data Processing Tool. *Communications of the ACM, 53(1)*, (pp. 72-77). DOI:10.1145/1629175.1629198.

Ekanayake, J., Gunarathne, T., & Qiu, J. (2011). Cloud technologies for bioinformatics applications. *Parallel and Distributed Systems, IEEE Transactions on*, *22*(6), 998-1011.

Gaggero, M., Leo, S., Manca, S., Santoni, F., Schiaratura, O., Zanetti, G., & Ricerche, S. (2008). Parallelizing bioinformatics applications with MapReduce. *Cloud Computing and Its Applications*, 22-23.

Huai-hsin Chi, E., Shoop, E., Carlis, J., Retzel, E., & Riedl, J. (1997). Efficiency of Shared-Memory Multiprocessors for a Genetic Sequence Similarity Search Algorithm.

Liu, J., Chandrasekaran, B., Wu, J., Jiang, W., Kini, S., Yu, W., & Panda, D. K. (2003, November). Performance comparison of MPI implementations over InfiniBand, Myrinet and Quadrics. *Supercomputing, 2003 ACM/IEEE Conference* (pp. 58-58). IEEE.

Leo, S., Santoni, F., & Zanetti, G. (2009, September). Biodoop: bioinformatics on hadoop. *Parallel Processing Workshops, 2009. ICPPW'09. International Conference on* (pp. 415-422). IEEE.

Lewis, S., Csordas, A., Killcoyne, S., Hermjakob, H., Hoopmann, M. R., Moritz, R. L., & Boyle, J. (2012). Hydra: a scalable proteomic search engine which utilizes the Hadoop distributed computing framework. *BMC bioinformatics*,*13*(1), 324.

Lichtenberg, J., Kurz, K., Liang, X., Al-Ouran, R., Neiman, L., Nau, L. J., ... & Welch, L. R. (2010). WordSeeker: concurrent bioinformatics software for discovering genome-wide patterns and word-based genomic signatures. *BMC bioinformatics*, *11*(Suppl 12), S6.

Matsunaga, A., Tsugawa, M., & Fortes, J. (2008, December). Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. *eScience, 2008. eScience'08. IEEE Fourth International Conference on* (pp. 222-229). IEEE.

Qiu, X., Ekanayake, J., Beason, S., Gunarathne, T., Fox, G., Barga, R., & Gannon, D. (2009, November). Cloud technologies for bioinformatics applications. *Proceedings of the 2nd Workshop on Many-Task Computing on Grids and Supercomputers* (p. 6). ACM.

Rabenseifner, R., Hager, G., & Jost, G. (2009, February). Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on* (pp. 427-436). IEEE.

Schatz, M. C. (2009). CloudBurst: highly sensitive read mapping with MapReduce. *Bioinformatics, 25*(11), 1363-1369.

Taylor, R. C. (2010). An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics. *BMC Bioinformatics*, *11*(Suppl 12), S1.

Thorsen, O., Smith, B., Sosa, C. P., Jiang, K., Lin, H., Peters, A., & Feng, W. C. (2007, May). Parallel genomic sequence-search on a massively parallel system. *Proceedings of the 4th International Conference on Computing Frontiers* (pp. 59-68). ACM.

Trelles, O. (2001). On the parallelisation of bioinformatics applications. *Briefings in Bioinformatics*, *2*(2), 181-194.

What is Cancer Proteomics. *In Office of Cancer Clinical Proteomics Research.* Retrieved from http://proteomics.cancer.gov/whatisproteomics.

Zou, Q., Li, X. B., Jiang, W. R., Lin, Z. Y., Li, G. L., & Chen, K. (2013). Survey of MapReduce frame operation in bioinformatics. *Briefings in Bioinformatics*, bbs088.

APPENDICES

Appendix A Steps Followed to Parallelize Code

The following steps were followed for parallelizing the code

- Loops along with the variables and data structures were identified
- Loop parallelization was done using TPL and CLR constructs
- Nested loops were analyzed for parallelization
- For loops indices were divided into chunks according to number of available cores
- Functions that are independent were taken into account
- Dependencies between functions were reduced
- Independent functions were executed concurrently
- Functions were chained such that pipelining is possible
- Barrier Class was used to synchronize the execution of the task
- Exception handling was implemented using time outs.

Appendix B Data Analysis Report

Table B.1The summary values for 15 iterations performed for parallel implementation of Andromeda

Sample	Cores	Average	Standard Deviation
100 MB	4	42.4562	0.000131
100 MB	8	32.4519	0.000500
100 MB	12	26.1986	0.000352
100 MB	16	23.2349	0.095741
200 MB	4	90.458	0.013541
200 MB	8	64.7895	0.231145
200 MB	12	49.3987	0.248879
200 MB	16	43.3279	0.918752
400 MB	4	230.7942	1.078439
400 MB	8	169.5875	1.097282
400 MB	12	145.7872	1.785722
400 MB	16	120.5875	2.987756
600 MB	4	441.2845	3.557511
600 MB	8	352.1765	3.854751
600 MB	12	270.9782	3.975851
600 MB	16	231.8765	3.875755

Table B.1 Continued

Samples	Cores	Average	Standard Deviation
800 MB	4	921.8689	3.907576
800 MB	8	687.5428	4.872752
800 MB	12	551.2788	5.428517
800 MB	16	487.5855	7.245891
1000 MB	4	1723.8712	9.024874
1000 MB	8	1204.7513	10.17863
1000 MB	12	998.2751	11.0751
1000 MB	16	887.1721	9.24713