The Role of Parallel Computing in Bioinformatics

Research Report

Submitted in partial fulfilment of the requirements for the Degree of Master of Science.

By

Timothy John Akhurst

Supervisor: Professor G. C.Wells

January 2005

Abstract

The need to intelligibly capture, manage and analyse the ever-increasing amount of publicly available genomic data is one of the challenges facing bioinformaticians today. Such analyses are in fact impractical using uniprocessor machines, which has led to an increasing reliance on clusters of commodity-priced computers.

An existing network of cheap, commodity PCs was utilised as a single computational resource for parallel computing. The performance of the cluster was investigated using a whole genome-scanning program written in the Java programming language. The TSpaces framework, based on the Linda parallel programming model, was used to parallelise the application. Maximum speedup was achieved at between 30 and 50 processors, depending on the size of the genome being scanned. Together with this, the associated significant reductions in wall-clock time suggest that both parallel computing and Java have a significant role to play in the field of bioinformatics.

Acknowledgements

I would like to thank the following people for their contributions towards this research project:

- Professor George Wells, my supervisor, for his invaluable guidance and drive during the duration of this research, and for providing an ideal environment for research as well as for kindly proofreading this research report.

- The Computer Science Department for the unlimited access to their facilities and openly welcoming me into their department.

- All the members of the Master's Lab in the Computer Science Department for providing an enjoyable atmosphere in which to work.

- The NBN and Biochemistry & Microbiology Department for their financial assistance.

- My mother, for kindly proofreading and editing this document for which I am extremely grateful.

- My final thanks go to Nina Carstens for all her encouragement, great sacrifices and extreme patience.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

CESDIS	Center of Space Data and Information Sciences
COTS	Commodity Off The Shelf
CPU	Central Processing Unit
DNA	Deoxyribonucleic Acid
FOLDOC	Free OnLine Dictionary Of Computing
GB	Gigabyte
GOLD	Genomes OnLine Database
GUI	Graphical User Interface
HMMs	Hidden Markov Models
HPC	High-performance Computing
IT	Information Technology
JPVM	Java Virtual Parallel Machine
MB	Megabyte
mbps	Megabytes Per Second
MIMD	Multiple Instruction Multiple Data
MISD	Multiple Instruction Single Data
MPI	Message-Passing Interface
NASA	National Aeronautics and Space Administration
NOW	Networks Of Workstations
ORF	Open Reading Frame
PC	Personal Computer
PMS	Parallel Motif Scan
PSSMs	Positive Specific Scoring Matrices
PVM	Parallel Virtual Machine
Regex	Regular Expression
RNA	Ribonucleic Acid
SIMD	Single Instruction Multiple Data
SISD	Single Instruction Single Data
SMS	Serial Motif Scan
WWW	World Wide Web

Glossary of Terms

Codon Block of three nucleotide residues, each of which specifies a different amino acid.

DNA sequencing DNA sequencing is the determination of the precise sequence of nucleotides in a sample of DNA.

Domain A portion of a polypeptide chain that folds on itself to form a compact unit that remains recognisably distinct within the tertiary structure of the whole protein.

Eukaryote Organisms whose cells are compartmentalised by internal cellular membranes to produce a nucleus and organelles.

Gene expression The synthesis of a normal, complete and functional polypeptide or protein from an appropriate gene.

Genome The total genetic information contained in a cell, an organism or a virus.

Homology modelling The use of the structural and functional characteristics of known proteins as a template for the generation of a hypothetical structure for a similar protein of unknown structure.

In silico A process that is completed entirely by use of a computer.

Molecular Dynamics Molecular dynamics (MD) simulation numerically solves Newton's equations of motion on an atomistic or similar model of a molecular system to obtain information about its time-dependent properties.

Motif A protein motif, also called a secondary structure motif, is a sequence of secondary protein structures such that the sequence recurs in a variety of proteins and specifies a characteristic three-dimensional structure.

Open Reading Frame A sequence within a messenger RNA that is bounded by start and stop codons and can be continuously translated. It represents the coding sequence for a polypeptide.

Prokaryote Primitive single-celled organisms that are not compartmentalised by internal cellular membranes.

Promoter A DNA sequence that can bind RNA polymerase, resulting in the initiation of transcription.

Reverse translation The process of converting a protein sequence into a DNA sequence.

Sequence alignment The arrangement of two or more amino acid or base sequences from an organism or organisms in such a way as to align areas of the sequences sharing common properties. The degree of relatedness or homology between the sequences is predicted computationally or statistically based on weights assigned to the elements aligned between the sequences. This in turn can serve as a potential indicator of the genetic relatedness between the organisms.

Chapter One

1.1 Introduction

The completion of the Human Genome Project in 2003 marked the conclusion of a 13-year global enterprise concerned with mapping the entirety of our genetic makeup (Meloan, 2004). Also, over the past decade there has been a dramatic increase in the number of completely sequenced genomes resulting from the race of multibillion-dollar genome-sequencing projects. The results of these achievements have led to a flood of data in genome sequence databases such as EMBL, SWISS-PROT and GenBank, which has caused them to double in size almost every year (See Figure 1) (Janaki and Joshi, 2003).

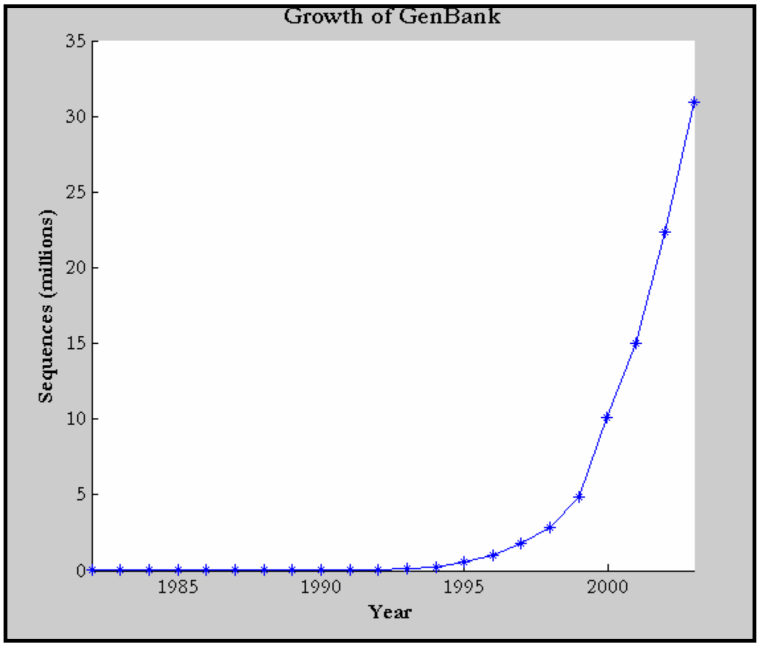


Figure 1: Graphical representation illustrating the growth of the GenBank database. The nature of the growth can be clearly seen as exponential. Taken from www.ncbi.nlm.nih.gov/GenBank/GenBankOverview.html

There are two additional factors which have contributed to and are currently contributing to this ever-increasing volume of data. The first factor can be attributed to some of the larger genomic research facilities generating more than several hundred gigabytes of data per day. The second factor is concerned with the development and implementation of high-throughput techniques for DNA sequencing and analysis of gene expression. The sheer volume of data and the analysis, which spans both multi-national pharmaceutical companies and academic collaborative networks, suggests that the completion of the work could not be achieved without the use of computers (Meloan, 2004 and Bader, 2004).

For example, SWISS-PROT is a protein and knowledge database that is renowned for its high quality annotation, usage of standardised nomenclature and its direct links to specialised databases and minimal redundancies. The current SWISS-PROT release (43.6) contains 153 320 sequence entries comprising 56 402 618 amino acids abstracted from 117 067 references (Boeckmann *et al*, 2003). GenBank is a comprehensive database that contains publicly available DNA sequences for more than 140 000 organisms. GenBank is redundant in nature, and on February 2004 it contained approximately 37 893 844 733 bases in 32 549 400 sequence records (Benson *et al*, 2004).

In addition to SWISS-PROT and GenBank are databases concerned with complete and ongoing genome projects. One such database is GOLD (Genomes OnLine Database) which currently contains 194 published complete genome sequences, 508 ongoing prokaryote genomes and 419 eukaryote genomes (Figure 2) (Bernal *et al*, 2001).

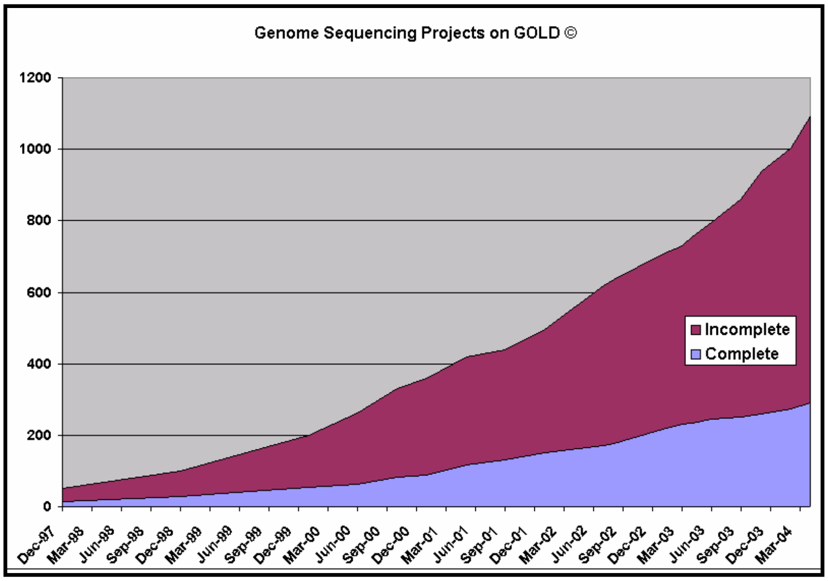


Figure 2: Graphical representation of all genome projects (complete and incomplete) available in the GOLD database. The exponential growth trend, which can also be seen for the growth of GenBank (Figure 1), is also clearly noticeable. Taken from www.genomesonline.org

Subsequently there is an enormous amount of biological sequence data flooding into the sequence databases. This phenomenon drives the development of efficient tools for comparative genome sequence analysis. With the aid of analysis tools, mining the available genome sequence databases plays a major role in comparative and functional genomics. The resulting information from these analyses has various important applications in science such as structural and functional annotation of novel genes and proteins, elucidating the gene order in the genome, gene fusion studies and constructing metabolic pathways, to name a few (Janaki and Joshi, 2003).

These studies are also invaluable for industries such as the pharmaceutical one, with particular reference to *in silico* drug target identification and new drug discovery. An example of this is the publication of the human genome sequence in February 2001. The release of the human genome will potentially result in more genes being identified as novel drug targets. Of the approximately 30 000 genes in the human genome, only a small number may lead to suitable drug targets. It has been estimated that the number of these targets ranges between 3 000 – 10 000 (Janaki and Joshi, 2003). According to Drews (2000), the set of drug targets available to the pharmaceutical industry has been estimated at only 483. When one compares the potential number of new targets to the existing number of drug targets this represents an order of magnitude increase (Reiss, 2001).

This flood of sequence data requires a system of representing, organising, manipulating, distributing, maintaining and finally using the information (particularly in a digital form). The comparatively new discipline of bioinformatics was born in an attempt to tackle the problems of this so-called 'post genomic era'. The functional aspect of bioinformatics is concerned with the representation, storage and distribution of this data. The intelligent design of data formats and databases, coupled with the creation of tools to query these databases and the development of user interfaces that combine the various tools, provide the user with the necessary means with which they can ask complex questions about the data. The second and more scientific aspect of bioinformatics is concerned with the development of the analytical tools required to discover knowledge in the data (Gibas and Jambeck, 2001).

The resultant biological information is used on a number of different levels, for example, in the comparison of sequences to develop a hypothesis about the function of a newly discovered gene, breaking down known three-dimensional protein structures into segments that can aid protein folding predictions, as well as modelling how proteins and metabolites work together to enable the cell to function (Gibas and Jambeck, 2001). The task of mining information from vast data sets is a Herculean one, and has resulted in scientists relying more and more on computational (*in silico*) processes.

The fields of bioinformatics and computational biology have been suggested to enable breakthroughs in basic biological research and improvements in the prevention, treatment and cure of diseases (Stewart, 2004). This project aims to highlight the use of computers with a particular emphasis on the role of parallel computing in the field of bioinformatics. According to David Bader in the November 2004 edition of the Communications of the ACM, the understanding of evolution and the basic structure and function of proteins are two grand challenge problems that can only be solved through the use of high-performance computing.

1.2 Project Proposal

1.2.1 Problem Statement

In the modern era, genomics is arguably one of the most rapidly developing areas in biology with data arising from sequencing projects increasing exponentially over the last five years (Bernal *et al*, 2001). Subsequently, there is flood of sequence data in databases such as EMBL, SWISS-PROT and GenBank. Coupled with this is a need to effectively capture, manage and analyse this data. As a result, bioinformaticians are presented with the challenge of developing specific analysis software packages which are required in order to extract useful information from the vast amount of sequence data (Janaki and Joshi, 2003).

The analysis of large datasets of genome sequences using uniprocessor machines appears to be an impractical approach. However, due to the 'embarrassingly parallel' nature of most biological problems, a far more practical and effective approach incorporates the usage of parallel clusters of workstations (Augen, 2003). Advances in both computer hardware and software algorithms that have revolutionised computational biology further support this approach. The role of high-performance computing has also been credited in being the only solution for two of the grand challenge problems in biology, namely, the understanding of evolution and the basic structure and function of proteins (Bader, 2004).

This project aims to highlight the potential and effectiveness of parallel cluster computing as a viable option to mining large datasets of genome sequences as well as to further support the notion that the Java programming language has a role to play, both in the realm of high-performance computing and in the field of bioinformatics.

1.2.2 Research Hypothesis

Parallel computing utilising networks of workstations is an efficient and effective tool in mining large genome datasets.

1.2.3 Objectives

- Design a genome-scanning program that scans through a whole genome sequence for a set list of regular expressions representing a variety of protein domain or motif signature profiles.

- Develop and implement the genome-scanning program, utilising the Java programming language, to identify the list of regular expressions in complete genomes.

- Execute the program on a single processor machine to serve as the benchmark for later speedup calculations.

- Design the same genome-scanning program in order for it to utilise a varying number of clients (processors) thereby investigating the effects of parallelism.

- Develop the genome-scanning program in conjunction with a suitable Java-based parallel framework to allow implementation in a parallel computing environment.

- Investigate the speedup for a number of different size genomes in order to determine the impacts of parallelism.

1.3 Document Structure

This section provides a brief summary of the content for the remaining chapters in this research report.

Chapter Two contains an overview of the literature relevant to the research undertaken, including the following topics: *In silico* Biology, Parallel Computing, Networks of Workstations, Parallel Computing and Bioinformatics, Java for Scientific Computing and finally TSpaces. Chapter Three describes the design and development of the Serial genome-scanning program (SMS) and provides a description of the requirements of the program, namely, the genomes and the regular expressions (regex's).

Chapter Four contains a description of the design, development and implementation of the genome-scanning program for execution in a parallel computing environment. Chapter Five is concerned with the overall experimental design and the results for each genome scanned. The discussion, conclusion and future work are presented in Chapter Six.

Chapter Two

Review of Literature

This section provides an overview of the role of *in silico* biology, with particular emphasis on its role in bioinformatics. The key concepts of parallel computing are described, including the basic architectures and the type of memory systems available, to name a few. The role of parallel computing in bioinformatics is illustrated with a number of examples of past and current applications. The use of the Java programming language for scientific computing is covered with a number of current uses being highlighted. The TSpaces parallel framework is also introduced as a means of providing the communication required for parallel execution.

2.1 *In Silico* Biology

Dramatic advances in Information Technology (IT) and computer sciences made the launch of *in silico* biology possible. As the field of *in silico* biology matured, researchers have become proficient at both defining biological problems using mathematical constructs and building the necessary computer infrastructure required to solve these problems (Augen, 2003).

In the era of genome projects the goal of biologists is to develop a quantitative understanding of how living things are built from the genome that encodes them. The explosion of data being released into databases such as GenBank (now growing at an exponential rate) and as databases beyond DNA, RNA and protein sequence, are undergoing the same dramatic transformation. The simple managing, accessing and presentation of this data in an intelligible form to the users is now a critical task which has lead to an increasing reliance on human-computer interaction specialists to manage these staggering amounts of data (Gibas and Jambeck, 2001).

Due to the explosive growth being experienced in the biological world in terms of the amount and type of available data, the relationship between bioinformatics and computer science has become unique amongst technical disciplines. In the past, technical improvements in IT were the driving force with respect to growth, as they enabled the employment and testing of new algorithms for *in silico* molecular modelling, pattern discovery, sequence matching and various other complex problems. This trend has now been reversed and it is in fact *in silico* biology that is shaping the IT industry. Bioinformatics has now become a leading indicator for the computer industry (Augen, 2003).

The position of bioinformatics is due in part to a large and growing number of small but technically sophisticated companies with computing needs that often rival those of the largest research organisations. These companies consist of thousands of biotechnology and pharmaceutical organisations who are tackling some of the most computationally intensive tasks imaginable, such as biological simulation, molecular modeling and dynamics, large-scale pattern recognition and X-ray and NMR-based protein structure determination. These companies are driving the emergence of a new model that promises to completely reshape the world of high-performance computing due to their accelerated demand for increased computing power combined with the need to build extensible platforms that minimise the cost-to-performance ratio (Augen, 2003).

The DNA, RNA and proteins of an organism, all of which are linear chains composed of smaller molecules, store information that provide an insight into an organism's heredity and function. Each of these macromolecules are assembled from a fixed alphabet of well-understood chemicals, for example, DNA is composed of four deoxyribonucleotides (adenine – A, thymine – T, cytosine – C and guanine – G), RNA is composed of four ribonucleotides (adenine – A, uracil - U, cytosine – C and guanine – G), and proteins are composed of the 20 amino acids. Due to these macromolecules being linear chains of defined components, they can be represented as sequences of symbols. These sequences can then be compared to find similarities that suggest the molecules are related by form or function, or they can be searched (via pattern searching techniques) to find specific regions such as promoters, open reading frames (ORFs) and motifs (Gibas and Jambeck, 2001).

Presently there are number of analytical tools which have been developed to aid researchers in their quest for knowledge in the 'post-genomic era'. These tools range from gene prediction programs, homology modelling programs which attempt to produce the structure of a protein based only on its sequence, multiple protein or nucleotide sequence alignment programs such as ClustalW and BLAST, gene expression analysis programs and other complex programs.

With particular reference to the variety of sequence analysis tools, a large number of tools available for mining vast amounts of data available in databases rely on FASTA (Pearson and Lipman, 1988) and Smith-Waterman (Smith and Waterman, 1981) algorithms. However, the analysis of large datasets of genome sequences using the aforementioned codes is computationally intensive and tends to be impractical on uniprocessor machines. As a result, there is a need to improve the performance of these tools and a solution to this problem was found in the form of parallel cluster computers (Janaki and Joshi, 2003).

Moreover, with time it has become increasingly clear that most biological problems lend themselves to being solved in a clustered environment after division into a large number of small pieces. Many biological problems are 'embarrassingly parallel' which implies that they can be divided easily into many small pieces in order to be solved (Augen, 2003).

Generally speaking, bioinformatics problems cover two large technical categories, namely, floating-point and integer. Floating-point problems are computationally intensive in nature as they have adopted complex algorithms from physical chemistry and quantum mechanics. Molecular dynamics, protein folding and metabolic systems modelling are examples of floating-point problems. Integer problems are invariably based on algorithms that compare characters in sequences or search for matching phrases and terms. These problems range from gene sequence alignment to pattern discovery, and they are often as computationally intensive as floating-point problems. Both types of problems favour solution using a parallel computation environment as the operations they depend on are atomic in nature (Augen, 2003).

Sequence homology and pattern discovery problems lend themselves perfectly to solution on clustered platforms. In most instances, a large number of sequences need to be matched against a single genome or sequence database. There are two different existing approaches for dividing the problem amongst *n* number of machines. The first approach requires performing a different search on each node with the target sequence stored either locally or remotely in a central database. The second approach requires the division of the target sequence across the cluster and managing overlap at the boundary of each node. Although the latter is fundamentally more complex, it is well suited to situations containing a large target sequence and a small number of search sequences (Augen, 2003).

On a cost-per-calculation basis, clustered solutions are far superior for problems that comprise a large number of isolated calculations, regardless of whether they are floating-point or integer-intensive. In fact, virtually every problem in bioinformatics gains a cost-to-performance advantage when engineered to run in a clustered environment (Augen, 2003). The use of parallel computers for performing sequence database searches appears to be the most realistic when one considers the shift away from conventional supercomputers to cost-effective clusters of workstations and PCs (Janaki and Joshi, 2003).

2.2 Parallel Computing

Many of the concepts for parallel computing date back to the 19th Century. However, no one seems to agree when parallel computing actually began. From a practical point of view, the beginning of parallel computing is considered to be sometime around the mid-1980s. It was during this period that parallel computers were beginning to be programmed as true parallel machines which could compete with the established supercomputers (Womble *et al*, 1999).

The free on-line dictionary of computing (FOLDOC) describes parallel processing as the use of more than one computer to solve a problem (FOLDOC, 2004). According to Professor Hank Dietz, "parallel processing refers to the concept of speeding-up the execution of a program by dividing the program into multiple fragments that can execute simultaneously, each on its own processor" (Dietz, 1999).

The field of high-performance computing (HPC) has traditionally focused on the availability of powerful machines, generally parallel supercomputers such as SGI/Cray T3E or the IBM SP2 (The UK JavaGrande forum, 1998). However, due to the exorbitant costs and long development times associated with these supercomputers, the demand for these machines has remained low since few institutions can afford them, their resources are limited and subsequently their use is restricted to a small number of important projects (The UK JavaGrande forum, 1998 and Sterling, 2001). The current advancements concerning high-speed networks and improved microprocessor performance have resulted in clusters or networks of workstations becoming an important tool in the era of cost-effective HPC (The UK JavaGrande forum, 1998).

The analysis of large biological datasets using a variety of parallel processor computer architectures is a common task in bioinformatics. The proper handling of any redundancies present in these datasets, together with the implementation of the unique features of parallel computing architectures, can significantly improve the efficiency of analysis. Bioinformatics is faced with the problem of handling highly redundant datasets, which in certain instances requires very large computations to be performed in order to gain insights into the meaning of the data (Pekurovsky *et al*, 2004). Fortunately, most of these problems can be readily divided into smaller pieces for solution. By building parallel infrastructures, bioinformaticians have been quick to design algorithms and programs that take advantage of these attributes, most often in the form of Linux clusters composed of commodity-priced machines. These clusters are now a dominant force in bioinformatics (Augen, 2003).

As mentioned earlier, parallel processing is the use of multiple processors to execute different parts of the same program simultaneously, with the main aim of parallel processing being the reduction in wall-clock time (amount of time before achieving a solution). As the number of processors is increased, a characteristic speedup curve demonstrating the effects of the increase up to a threshold number of processors can be seen (Figure 3). Anything above this threshold may be counter-productive and can result in an increase in solution time.

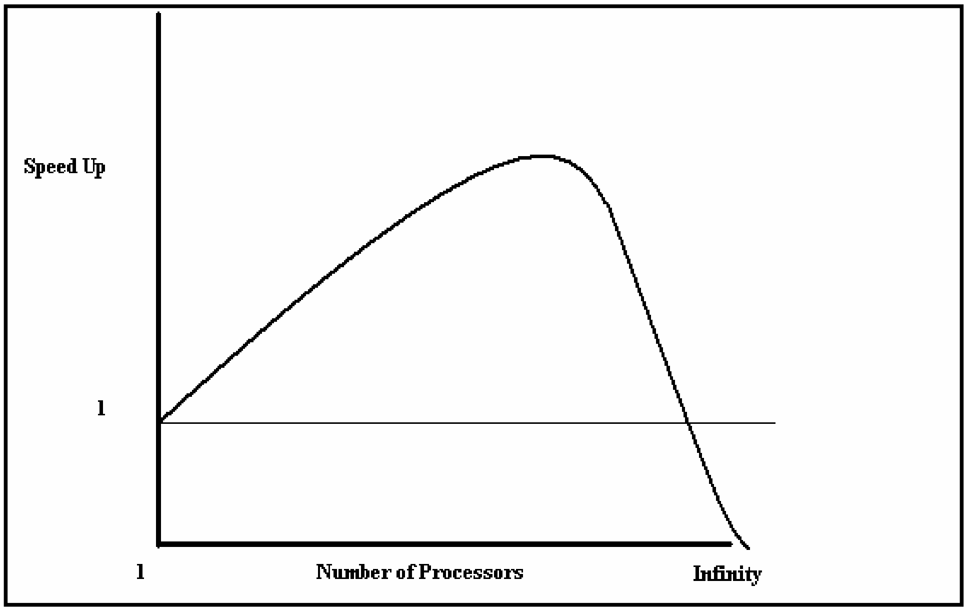


Figure 3: Graph depicting the effects of increasing numbers of processors on the overall computational speedup.

This simple theory of dividing the pieces of a solution amongst many processors represents both the power and the weaknesses associated with parallel computing. On one hand, as you increase the number of helpers (processors) for a given task a beneficial speedup is obtained. However, beyond the threshold limit any further increase in the number of helpers can be viewed as being counter-productive, which the supports the age-old adage that "too many cooks spoil the broth" (Cornell Theory Center, 2000).

2.2.1 Taxonomy of Parallel Computer Architectures

Flynn proposed the following classification of parallel computer architectures in 1966. This classification scheme separates computer architectures according to two independent, binary-valued dimensions. This implies that neither of the two dimensions has any effect on the other and that each dimension has only two states. Flynn proposed that the two dimensions be Instruction and Data, and that the values for both dimensions be Single or Multiple. This led to four possible computer architectures:

- Single Instruction, Single Data (SISD):

SISD is the oldest style of architecture, and is still one of the most important. Most computers ever designed or built, until fairly recently, fit within this category. The SISD architecture refers to the fact that there is only one instruction stream being acted on by the CPU during any one clock tick, and that only one data stream is employed as input during any one clock tick. This class of architecture contains most commonly available computers including most personal computers, all single-instruction-unit-CPU workstations, most mini-computers and mainframes.

- Multiple Instruction, Single Data (MISD):

There are few known working groups of this type of computer system, and as such, there are few examples of computers in this class.

- Single Instruction, Multiple Data (SIMD):

SIMD systems are an important class as they are capable of applying the identical instruction stream to multiple streams of data simultaneously. For *data-parallel* problems, this type of architecture is perfectly suited to bioinformatics as the data can be divided into many small pieces, and the multiple instruction units can all operate on them simultaneously. Thus, this type of architecture lends itself to achieving very high processing rates.

- Multiple Instruction, Multiple Data (MIMD):

This class is the most general of the four, and an MIMD machine is capable of being programmed to operate as if it were any of the four. Multiple instructions streams are simultaneously applied to multiple data streams, and it is believed by many that this particular approach to parallelism will result in the next major advances in computational capabilities (Cornell Theory Center, 2000).

SIMD and MIMD struggled for dominance in the late 1980s. In the struggle between the two approaches, SIMD appears to have fallen by the wayside. The more flexible and more general purpose nature of the MIMD approach has prevailed even though the SIMD approach can be cost effective for certain tasks (Womble *et al*, 1999).

There are two basic ways to divide computational work among parallel tasks, namely, data and functional parallelism. Data parallelism requires that each task performs the same series of calculations, but applies them to varied data. Subsequently, each processor performs exactly the same operations, but works on different parts of a dataset. Functional parallelism requires that each task performs different calculations; that is, each task carries out different functions of the overall problem. This type of parallelism can be applied to the same data or to different data (Cornell Theory Center, 2000).

There are a number of aspects to consider when approaching parallelism. The first is synchronisation which is required to coordinate information exchange among tasks, but which can consume wall-clock time as processor(s) sit idle waiting for tasks on other processors to complete. Thus, synchronisation can be a major factor in decreasing parallel speedup.

The second aspect, parallel overhead, is also important as this involves the amount of time required to coordinate parallel tasks. The three most commonly encountered coordination tasks are:

1. The time to begin a task. This is concerned with the identification of the task, locating a processor to perform the task, loading the task onto the processor, placing the required data onto the processor and finally beginning the task.

2. The time to end a task. In order for a processor to be made available for further tasks, all results need to be combined or transferred and the operating system resources need to be released.

3. Synchronisation. As referred to earlier, synchronisation involves the coordination of information exchange among tasks (Cornell Theory Center, 2000).

Granularity, too, needs to be considered as it is a measure of the ratio of computation performed in a parallel task to the amount of communication. The scale ranges from fine-grained (nominal computation per communication-byte) to coarse-grained (extensive computation per communication-byte). As the granularity becomes finer, the need for synchronisation increases which leads to a greater limitation on speedup. The nature of the parallel system with regards to scalability is also important and this is dependent on some combination of the following components: hardware, parallel algorithm and the actual code (Cornell Theory Center, 2000).

The type of memory to be utilised is also an important consideration. There are two types of memory usage to consider: shared and distributed. With a shared memory system (Figure 4), as the name implies, the same memory is accessible to multiple processors. Synchronisation of tasks is achieved by tasks' reading from and writing to the shared memory. Whilst a concurrent task is accessing the shared memory location, another task must not be able to alter it. One of the advantages of shared memory is that the sharing of data amongst tasks (speed of memory access) is fast. However, it is limited by the fact that the number of access pathways to memory restricts scalability. A further drawback is that the user is responsible for specifying synchronisation (Cornell Theory Center, 2000).

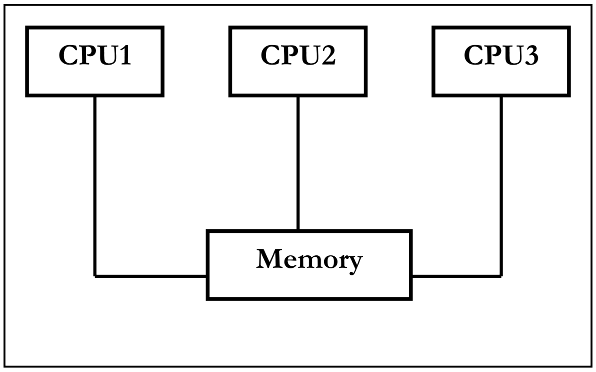


Figure 4: Graphical representation of the basic concept of a shared memory parallel system.

The memory in a distributed memory system (Figure 5) is physically distributed among processors, each local memory being directly accessible only by its processor. Each component of a distributed memory parallel system is invariably a self-contained environment, which is capable of acting independently of all other processors in the system. Synchronisation is required to move data between processors, and this traffic along the communications network is the only link among the processors (Cornell Theory Center, 2000).

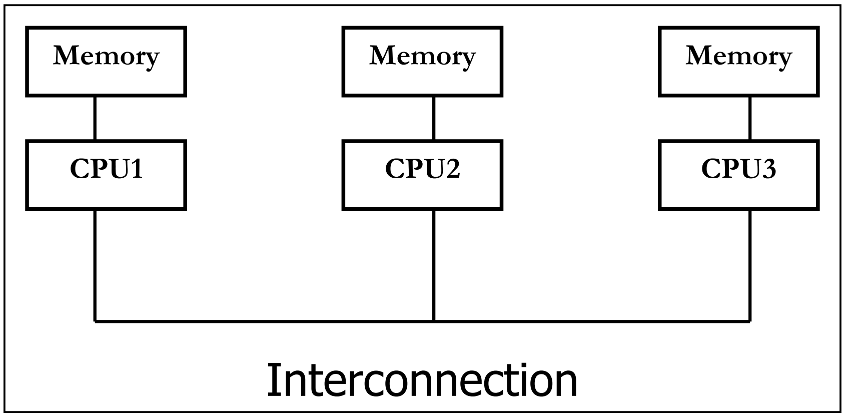


Figure 5: Graphical representation illustrating the basic concept of a distributed memory parallel system.

A major concern in distributed/parallel systems is that of data decomposition, and in particular, how to divide arrays of data among local CPUs to minimise communication. This represents one of the major distinctions between shared and distributed-memory computing. The data structure needs to be decomposed, i.e. divided into small pieces, assigned to a processor and physically sent to that processor, in order for the data to be processed. Whichever processor is responsible for the final result then requires that any results obtained by the other processors must be sent back to it so that it may coordinate the final result (Cornell Theory Center, 2000).

Distributed memory is virtually synonymous with message-passing. Message-passing is an approach that requires that tasks communicate by sending packets to each other. The messages are discrete (they have a definite identity), and can be distinguished from all other messages. Parallel tasks are reliant on these messages to send information and request information among processors. The overhead is proportional to the size and number of packets, i.e. more communication means greater cost, since sending data is generally slower than accessing shared memory. Each message is individually constructed, addressed, sent, delivered and read, all before the information it contains can be acted upon (Cornell Theory Center, 2000).

2.3 Networks of Workstations

The astronomical costs associated with the modern supercomputers, such as the Cray range and the Connection Machine, have meant that few programmers have had an opportunity to use these machines. The quest for ever-increasing computing power at minimal cost has led to several alternatives being examined and tested. Networks of workstations became an attractive alternative to the traditional supercomputers and parallel computing systems for high-performance computing in the early 1990s. There were a number of early projects, two of the most notable being the NASA Beowulf Project (Merkey, 2004) and the Berkeley NOW (networks of workstations) project (Culler *et al*, 1997). The Beowulf project is generally credited with being the first cluster computation project to be built using exclusively COTS (commodity off the shelf) elements.

In 1994, Thomas Sterling and Don Becker, two researchers at the Center of Excellence in Space Data and Information Sciences (CESDIS), assembled a cluster computer consisting of 16 DX4 processors connected by channel bonded Ethernet. The success of the Beowulf machine was instantaneous and the idea of using COTS base systems to satisfy specific computational requirements rapidly spread through NASA as well as into the academic and research communities. Factors driving the ongoing success of the Beowulf project include improved performance in microprocessors and cost/performance gains experienced in the network technology (Beowulf Introduction & Overview, 2004 and Merkey, 2004).

Furthermore, the ongoing development of publicly available software, in particular the Linux operating system and the MPI and PVM message passing libraries, allow for the development of hardware-independent software. A further consideration in the continued interest and research into cost-effective parallel computing systems is the increased reliance on computational science, which directly increases the need for high-performance computing. The cost, effectiveness and Linux support for high-performance networks for PC class machines has provided researchers with the ability to construct balanced systems built exclusively of COTS technology (Merkey, 2004).

2.4 Parallel Computing and Bioinformatics

Due to the ever-increasing number of completely sequenced genomes becoming accessible to the public, currently available genome data is increasing exponentially (Bikandi, 2004 and Gao and Zhang, 2004). Subsequently, bioinformaticians are presented with the challenge of developing specific analysis software packages, which are required in order to extract useful information from the vast amount of sequence data. Of critical concern is the development of computational gene recognition programs for the annotation of vast amounts of uncharacterised DNA sequences (Gao and Zhang, 2004). In addition to this is the need for time-efficient processes. The apparent 'embarrassingly parallel' nature of most biological problems lends itself to the use of parallel computing (Augen, 2003).

One of the most significant applications to date was the assembly algorithm used to construct the human genome from millions of fragments obtained through shotgun sequencing. The execution of this algorithm was computationally intensive, and now represents one of the most complex logical problems ever solved. Molecular dynamics simulations, gene sequence alignment and pattern discovery are further problems that lend themselves to solution in a parallel computing environment (Augen, 2003).

The use of multiple alignments is a key procedure in bioinformatics because a sequence comparison by multiple alignment can provide vast amounts of information about structure-function relationships, such as evolutionary conserved residues or conserved hydrophobicity patterns. ClustalW, T-coffee and Praline are a few commonly used multiple alignment packages. Researchers in the Division of Mathematical Biology at the National Institute for Medical Research have looked at solving the problems associated with compiling large sequence alignments. They implemented parallel processing in the form of a SIMD system into the multiple alignment program, Praline, by using Message Passing Interface (MPI) routines. They found that the parallelised program performed up to ten times faster on 25 processors when compared with the use of a single processor (Kleinjung *et al*, 2002).

Researchers at the Bioinformatics Institute in Singapore further demonstrated the use of parallel computing in conjunction with the ClustalW (protein or nucleotide sequence) multiple alignment tool. They developed software that relies on an MPI library which runs on both distributed workstation clusters and traditional parallel computers. They, too, reportedly found that it is possible to speed up lengthy multiple alignments with the aid of parallel computing (Li, 2003).

Applications harnessing the potentials of parallel computing include the automation of genomic data-mining processes, such as Sight, which is a package that provides a user-friendly interface to generate and connect agents for automatic data mining for individual purposes (Meskauskus, 2004). Other commonly used applications involve small-scale research based molecular dynamics simulations, such as that perfomed by Zubrzycki (2002), in which the molecular dynamics simulation was shared over two processors. Parallel processing was also used in conjunction with pattern searching packages as demonstrated by Krishnan and Tang (2004), who utilised parallel computing to perform exhaustive whole-genome tandem repeats searches. They divided their pattern length evenly between 1, 5, 10, 25 and 50 processors and reported to achieve linear speedup.

2.5 Java for Scientific Computing

Employees of a computer company called Sun Microsystems designed Java during the first half of the 1990s. The language was formally introduced to the public in 1995, and although Java is now commonly associated with the worldwideweb (WWW), it can be used in most programming areas. For this reason, Java is often referred to as a general programming language (Cornelius, 2001 and Russel, 2001).

Java is a high-level programming language which means that it uses instructions that more closely resemble a written language (such as English) than machine language. One of the most important features of Java is that it is platform independent, as you can run Java programs on any operating system without having to rewrite or recompile them for each system. Java is also an object-oriented language as opposed to the more traditional procedure-oriented program that follows a logically ordered set of instructions. Object-oriented languages, such as Java, have the added capability of encapsulating sets of characteristics and functions into classes (Russel, 2001).

The Java language has many advantages over traditional scientific computing languages such as C, C++ and FORTRAN. These advantages are that Java is a small, simple, object-oriented language that is distributed and secure in nature. It is an architecturally neutral language that is also portable, dynamic, multi-threaded and robust. These advantages make Java a likely contender as the language of choice for the future development of scientific libraries and applications (The UK JavaGrande forum, 1998).

Bull *et al*, (2001), state that the nature of many scientific applications makes them well suited to Java execution environments. This was based on the fact that scientific applications typically spend a large amount of execution time in a small number of user-written methods. The use of Java is becoming increasingly popular and in 2003, researchers at the Sanger Institute for genetic study launched BioJava. BioJava, is an active open-source project dedicated to providing genomics researchers with a Java based developer's toolkit. The facility is currently in use at major research and pharmaceutical centers in over 85 countries, and provides bioinformatics developers with over 1 200 classes and interfaces for genomic sequence manipulations (Meloan, 2004).

2.5.1 Current Applications

Java is still in its infancy in terms of its role in scientific computing, and to date most applications have revolved around developing Java-based frameworks for parallel programming on networks of workstations. JavaNOW, which is a parallel computing framework that creates a virtual parallel machine similar to the Message Passing Interface (MPI) model and also provides distributed associative shared memory similar to the Linda model, has been developed (Thiruvathukal *et al*, 2000).

The Java Parallel Virtual Machine (JPVM) library has also been demonstrated as a software system for explicit message-passing based distributed memory MIMD parallel programming in Java (Ferrari, 1999). The JPVM library supports a Java interface in a similar manner to the interface provided by the Parallel Virtual Machine (PVM) library which matches the C and Fortran interfaces (Ferrari, 1999).

A parallel library implemented on Java that supports the execution of massively parallel applications over the Internet, called JET, has also been developed. Java applets that are downloaded through a Web page are responsible for the execution of applications. This type of parallel approach can be implemented to solve long-running problems, thus diminishing the need for supercomputers (Pedroso, 1998).

A number of researchers have focused on utilising the tuplespace model pioneered by David Gelernter and colleagues in the Linda programming system at Yale University (Gelernter, 1988). The tuplespace model presents an attractive means of co-ordinating objects across a distributed computing environment, which results in a different communication paradigm between parallel processors (Hawick *et al.*, 2004 and Dente *et al*, 2004). A very simple set of operations is applied to a shared data collection, shared 'memory' called Tuplespace, and is used for message exchange between processors. The Linda model provides a set of functions to access and modify the data stored in the Tuplespace (Dente *et al*, 2004).

A tuple was devised as the unit of communication. There are two types of unit, namely, active and passive tuples. Active tuples were basically task-description, consisting of elements and functions that had to be evaluated by the Linda server, and needed computation. Passive tuples were values stored in Tuplespace, which represented the results of computation. Once evaluated, an active tuple became a passive tuple. In order to synchronise the parallel process the Linda model provides a set of functions:

- out(tuple)
  - Places a tuple into the Tuplespace.

- rd(pattern-tuple)
  - Retrieves all tuples that match a given template from the Tuplespace.

- in(pattern-tuple)
  - Retrieves and removes all tuples which match a given template from the Tuplespace.

- eval(FUNCTION-TUPLE)
  - Creates an active tuple and evaluates it. The results are then stored as a passive tuple in Tuplespace (Dente *et al*, 2004).

Sun and IBM have both attempted to provide developers with a Java-based distributed-object architecture that includes a development platform, processing environment and addressing mechanism. Both companies have based their approach on the tuplespaces of the now famous Linda prototype, with Sun developing JavaSpaces and IBM, TSpaces. The objects of both JavaSpaces and TSpaces also borrow several other Linda-specific distributed database system solutions for storing collections of data for future computation, and for performing queries that are controlled via a form-driven interface which utilises value-based lookup tables. They are also both further inspired by Linda's distributed computing concept which uses simple application functions to extend basic data typing mechanisms (IEEE, 2004).

2.6 TSpaces

The TSpaces software package was designed as a communication package with the sole purpose of alleviating the problems associated with linking together disparate distributed systems. TSpaces is a global communication middleware component that incorporates database features such as transactions, persistent data and flexible queries. It is also an excellent tool for designing and developing distributed applications, since it provides an asynchronous and anonymous link between multiple clients or services (Lehman *et al*, 2001).

TSpaces was developed at IBM's Almaden research centre to explore the possible use of Java in middleware systems, and was launched in March 1998. It represents a software package that provides a common set of services for a network of heterogenous computers and operating systems. The TSpaces model was based on the Linda model, and thus they share a number of key elements. The simple syntax, which was one of the most popular features of the Linda model, is also employed by the use of simple, intuitive and terse language that can perform a variety of tasks in the TSpaces model (IBM, 2004).

2.6.1 TSpaces Model

The TSpaces model is surprisingly simple; there are clients and there are servers. TSpaces servers can be run everywhere, such as locally to coordinate a few office machines or in department servers for wider range services. Any program that makes calls to the TSpaces server is known as a client program, where the clients read and write data to and from a server using simple method calls (See Table I) (IBM, 2004). The server contains a Tuplespace, which represents the model of interaction for building a globally visible communication buffer in which a Tuplespace represents a globally shared, associatively addressed memory space that is organised as a bag of tuples (Wyckoff, 1998).

The Tuplespace concept embodies three main principles:

- Anonymous communication
- Universal associative addressing
- Persistent Data (IBM, 2004)

Table I: Overview of some of the methods used for reading and writing tuples from or to a TSpaces server.

Method Call Description	**Method Call**
Create the initial tuplespace	Tuplespace ts = new TupleSpace (spaceName, serverName)
Write some data, tagged "ClientsData"	ts.write("ClientsData", dataInstance)
Read the specific data record	resultTuple = ts.read("ClientsData", dataInstance)
Read ALL the records of that type	resultTupleSet = ts.scan(String, dataInstance)

The client interface is very simple. A client is required to create an instance of a tuplespace and then to use the methods of that instance to read and write tuples, which are merely Java vectors of fields. The field class is the most basic component of the Tuplespace data structure hierarchy and it contains a type, value and optional field name. Tuplespace methods are used to send and receive tuples from the shared network depository. As a result tuplespaces are seen as network communication buffers and can be accessed and modified utilising a simple API. There are a number of Tuplespace methods and a few of the interesting ones are:

- write(tuple)
  - Adds a tuple to TSpaces.

- take(templateTuple)
  - Performs an associative search for a tuple that matches the template. If the tuple is found, it is removed from the space and is returned, otherwise null is returned.

- waitToTake(templateTuple)
  - As for take, except that it blocks until a match is found.

- read(templateTuple)
  - As for take, but the tuple is not removed from the tuple space.

- waitToRead(templateTuple)
  - Similar to waitToTake, but the tuple is not removed from the tuple space.

- scan(templateTuple)
  - As for read; however, the entire set of tuples that matches is returned.

- eventRegister(command, template tuple, callback routine)
  - Register for an event corresponding to the command and the template tuple.

- countN(templateTuple)
  - Similar to scan except that it returns a count representing the matching tuples (Dente *et al*, 2004).

The role of parallel computing in bioinformatics is to be investigated by developing a genome-scanning program using the Java programming language and the TSpaces framework. Since most tertiary institutions and research centres are in possession of networks of workstations, the use of an existing network of cheap, commodity PCs provides the necessary environment for parallel execution.

Chapter Three

Serial Program Design, Development and Overall Program Requirements

In order to determine the effects of parallelism and the subsequent role of parallel computing in bioinformatics, a program to execute the genome-scan in serial needs to be designed and developed. This is required as the search time or wall-clock time attained from the execution of this serial program serves as the benchmark for later speedup calculations. In addition to this, a number of genomes are required so that the effects of parallelism can be investigated using a range of different sized genomes. A number of regular expressions need to be created so that they may be scanned for within the selected genomes.

3.1 Serial Motif Scan (SMS)

The first challenge was to design, develop and implement a genome-searching program (later referred to as SMS) that could be executed on a uniprocessor machine. Utilising a suitable parallel framework, this would serve as the benchmark for comparison with the later development and implementation of the same genome-searching program. The first stage in the design of the program was to define the overall problem to be solved which entails the clear definition of the inputs, outputs and the processes of the program. The second stage was the actual design of the problem which requires that the program be broken into a logical sequence of steps and the actions to be executed clearly defined.

3.1.1 SMS Basic Analysis

The overall problem was concerned with searching whole genomes for a pre-defined set of motifs or domains, and to return intelligible information regarding the total number of each pattern found as well as the relative positions (start and end) for each pattern found. As a result, the program would require a list of motif or domain regular expressions (regex's) and a genome file as inputs, and it would also need to output the results obtained from the search together with the total search time.

The process requirements were to read in a designated file containing the genome sequence and to read in the file containing the list of motif or domain regex's. The genome would then be scanned and the results presented to the user.

3.1.2 SMS Design and Development

The program was required to receive as input from the user, the name of the genome file to be scanned. The genome file would be accessed and read into a String to serve as the template to be searched. Once the genome file was successfully loaded, the list of regex's would be accessed and all patterns stored in a vector. A method utilising Java's regex package would be required to receive as a parameter the current regex; the method would then scan the genome using the received regex. All matches, including their start and end position numbers and a corresponding match number, will be returned as a vector.

The getFileName( ) method was written to prompt the user for the name of the genome file to be scanned. Once entered, the genome name is stored as a String and returned to the main method so that the genome sequence may be loaded. In order to obtain the genome sequence the Sequence class was created. This class contains two key methods, one to locate and read in the genome file and the other to return the string representation of the extracted sequence. The initialiseSequence(String) method was created to receive the name of the genome to be scanned as a parameter. This information is then used to read the data from the desired genome file. A boolean value of true is returned once the genome sequence has been successfully loaded, whereas false is returned when the loading is unsuccessful. The sequence is then returned using the getSequence( ) method that simply returns the String representation of the genome file.

The regular expressions representing the protein domain or motif profiles, were then loaded so that the genome may be scanned. A Patterns class was created that would firstly extract each regex from the list and then return all regex's stored in a vector. Like the initialiseSequence( ) method in the Sequence class, the initialiseMotifs( ) method returns either true or false depending on the success in creating the vector of regex's. The vector of regex's is then returned to the main method so that each element may be extracted and searched for within the genome sequence, this being achieved by using the getMotifs( ) method that returns the vector containing all the regular expressions.

A for-loop containing the patterns extracted from the list of motif or domain regular expressions was required in order to access each element (regex) from the vector. Every cycle in the loop would call the findMotif(String, String) method that performed the genome search and pass, as the parameters, the current element of the vector and the name of the String representing the genome to be scanned. The findMotif( ) method utilises the java.util.regex package to scan the genome.

An instance of the Pattern class represents a regular expression that is specified in string form in a syntax similar to that used by Perl. Instances of the Matcher are used to match character sequences against a given pattern. Since the findMotif( ) method is called within the for-loop, each regex is assigned as a String and compiled into a pattern. Invoking the pattern's matcher method where the genome sequence is the input sequence, creates the matcher.

The matcher's find( ) method utilises a matching operation that scans the input sequence looking for the next subsequence that matches the pattern. Each resultant match is then appended onto the end of a results vector. A string is used to store results and a semi-colon (;) is used to separate the results for each regex scanned. Where no matches were found 'null' is concatenated onto the results string, and where matches were found the string of matches is concatenated onto the end of the results string. The results from each regex scanned would then be displayed and written to a file. These results include the total number of matches for each specific regex scanned as well as the sequence matched, and its start and end position numbers.

In order to ascertain the time associated with searching the genome for a number of predefined patterns, the program recorded the time after both the genome file and the list of regular expressions were accessed and loaded. Once the for-loop extracting each pattern from the vector and the search results concerning each pattern were completed, the program would obtain the time and subtract the first time obtained from the last time determined in order to compute the total search time. The total search time was returned in milliseconds, since the System.currentTimeMillis( ) method returns a long data type in milliseconds.

Note: The genome file and the list of motif or domain regular expressions are required to be located in the same directory that the program is stored in.

The initial development of SMS utilised a file containing a short random sequence of DNA and a file containing four artificial regular expressions. Once the initial development was complete, the task of sourcing and downloading a number of motif or domain regular expressions as well as various size genomes began.

3.2 Genomes and Regular Expressions

3.2.1 Genomes

The objective of this project was to determine the effects of parallelism using various sized genomes. It was decided to download the first ten chromosomes from the human genome, and then to create a variety of 'genome' files using this data. The DNA sequences for the human chromosomes were sourced and downloaded from the Ensembl Genome Browser (Birney *et al.,* 2004).

The following files were downloaded and used for analysing the effects of parallelism:

- 60 MB (Human chromosome 20)
- 140 MB (Human chromosome 9)
- 250 MB (Human chromosome 1)
- 1072 MB or 1.072 GB (Human chromosomes 1 – 5)

3.2.2 Regular Expressions

The use of regular expressions was decided on because all versions of Java since 1.4.0 contained the regex package. This was employed as the method to scan the various genomes. The consensus patterns for 100 various protein motif or domain signature profiles were obtained from the PROSITE (Hulo *et al.,* 2004) database. All motif or domain signature profiles obtained were in protein sequence and would thus need to be reverse translated from protein to DNA. The profiles were converted from protein to DNA due to the majority of published genomic data being that of DNA sequences. A selection of the motif or domain signature profiles obtained from PROSITE can be seen below in Table II. Since this project is concerned with searching whole-genomes it represents the most logical approach, as the process of translating a whole genome from DNA to protein could itself require a suitable parallel algorithm due to the nature of the problem.

Table II: A selection of the motif/domain signature profiles obtained from the PROSITE database, prior to their reverse translation into DNA.

Motif/Domain	Protein Signature Profile
Ubiquitin Consensus Pattern	K-x(2)-[LIVM]-x-[DESAK]-x(3)-[LIVM]-[PA]-x(3)-Q-x-[LIVM]-[LIVMC]-[LIVMFY]-x-G-x(4)-[DE]
Zinc Finger RING-type consensus pattern	C-x-H-x-[LIVMFY]-C-x(2)-C-[LIVMYA]
Hsp70 1 consensus pattern	[IV]-D-L-G-T-[ST]-x-[SC]
Hsp90 consensus pattern	Y-x-[NQH]-K-[DE]-[IVA]-F-[LM]-R-[ED]
P53 family signature	M-C-N-S-S-C-[MV]-G-G-M-N-R-R

Note: The x indicates any amino acid single letter code and the number between braces, i.e. x(4), indicates the number of random amino acids in sequence. The [ ] brackets indicate that only one of the amino acids between these brackets can occur at this particular position, i.e. [LIVM].

A suitable tool to perform the reverse translation was sourced and the process of reverse translating and recompiling into suitable regular expressions began. The Sequence Manipulation Suite from the University of PennState's Centre for Computational Genomics was used for the reverse translation. In order to prepare the sequences for reverse translation, all characters other than those representing the single letter amino acids (highlighted in Table II) were removed, which resulted in an unbroken sequence of single letter amino acid characters. This unbroken sequence was then used as the template for reverse translation.

The resulting DNA sequence was then converted into a regular expression based on the consensus patterns obtained from PROSITE. A selection of these can be visualised in Table II. A total of 100 different protein motif/domain profiles were selected from the PROSITE database and subjected to reverse translation prior to their conversion into DNA regular expressions.

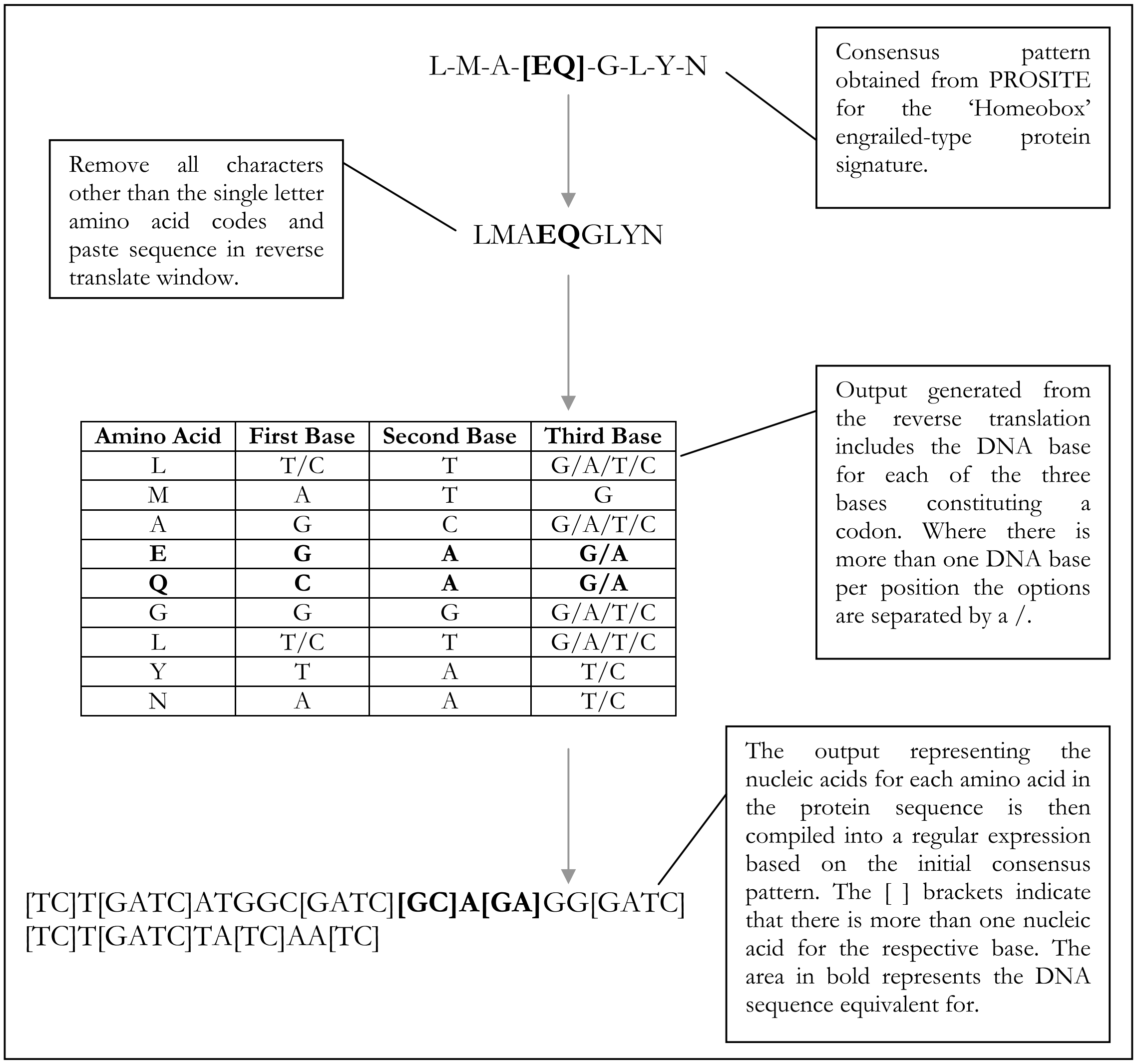
Table III: The DNA regular expressions for the five random protein motif/domain profiles highlighted in Table II.

Motif/Domain	DNA Regular Expression
Ubiquitin Consensus Pattern	AA[AG].*{6}[GATC]T[GATC].*{3}[GAT][GAC][GATC].*{9}[GATC]T[GATC][GC]C[GATC].*{9}CA[GA].*{3}[GATC]T[GATC][GATC][GT][GATC][GATC][AT][GATC].*{3}GG[GATC].*{12}GA[GATC]
Zinc Finger RING-type consensus pattern	TG[TC].*{3}CA[TC].*{3}[GATC][AT][GATC]TG[TC].*{6}TG[TC][GATC][ATC][GATC]
Hsp70 1 consensus pattern	[GA]T[GATC]GA[TC][TC]T[GATC]GG[GATC]AC[GATC][AT][GC][GATC].*{3}[AT][GC][GATC]
Hsp90 consensus pattern	TA[TC].*{3}[AC]A[GATC]AA[GA]GA[GATC][GA][TC][GATC]TT[TC][ATC]T[GATC][AC]G[GATC]GA[GATC]
p53 family signature	ATGTG[TC]AA[TC][AT][GC][GATC][AT][GC][GATC]TG[TC][GA]T[GATC]GG[GATC]GG[GATC]ATGAA[TC][AC]G[GATC][AC]G[GATC]

It should be noted that the initial development and debugging of SMS utilised a short sequence of random DNA (approximately 0.5 megabytes/MB) to serve as the genome and a list containing four non-specific regular expressions, which represented imaginary DNA patterns.

3.2.2.1 Protein consensus pattern to DNA regex

In order to create a list of DNA regular expressions, a number of protein consensus pattern sequences were obtained and subjected to reverse translation into DNA sequences. The task of creating the DNA regex from the protein sequence is compounded by the fact that there are four DNA bases (A, T, G, C) and each group of three bases (codon) can represent as many as 64 possible amino acids (4 x 4 x 4 = 64). Since there are only 20 amino acids, there is a high level of redundancy in the genetic code and some of the amino acids are represented by more than one codon. The flow diagram in Figure 6 highlights the required steps to successfully create the DNA regex's:



L-M-A-**[EQ]**-G-L-Y-N

Consensus pattern obtained from PROSITE for the 'Homeobox' engrailed-type protein signature.

Remove all characters other than the single letter amino acid codes and paste sequence in reverse translate window.

LMA**EQ**GLYN

Amino Acid	First Base	Second Base	Third Base
L	T/C	T	G/A/T/C
M	A	T	G
A	G	C	G/A/T/C
E	**G**	**A**	**G/A**
Q	**C**	**A**	**G/A**
G	G	G	G/A/T/C
L	T/C	T	G/A/T/C
Y	T	A	T/C
N	A	A	T/C

Output generated from the reverse translation includes the DNA base for each of the three bases constituting a codon. Where there is more than one DNA base per position the options are separated by a /.

[TC]T[GATC]ATGGC[GATC]**[GC]A[GA]**GG[GATC]
[TC]T[GATC]TA[TC]AA[TC]

The output representing the nucleic acids for each amino acid in the protein sequence is then compiled into a regular expression based on the initial consensus pattern. The [ ] brackets indicate that there is more than one nucleic acid for the respective base. The area in bold represents the DNA sequence equivalent for.

Figure 6: Annotated flow diagram illustrating the steps involved in the creation of DNA regular expressions.

On the successful development of the serial genome-scanning program, each selected genome is to be scanned using the list of regular expressions which have been created by obtaining a protein consensus pattern for a variety of protein domains or motifs. The protein sequence is subjected to the process of reverse translation to produce its corresponding DNA sequence, which is then formatted into a regular expression based on the initial protein consensus pattern. The wall-clock time is computed and saved in a file for later speedup calculations.

Chapter Four

Design, Development and Implementation of the Parallel Algorithm

The design and development of the parallel algorithm requires that the serial program be divided into a number of pieces such that more than one client may be involved in the program execution. The TSpaces parallel framework is used in the parallel program as a means of achieving the necessary level of communication required between the host and client programs. A Graphical User Interface is also required so that the user may enter the necessary information pertaining to each run, and then display the necessary results.

4.1 Parallel Motif Scan (PMS)

In order to implement the genome-searching program in a parallel environment the program would require a number of new methods, which would thus require that the steps associated with the design and development of the serial motif scanning program be repeated.

4.1.1 PMS Basic Analysis

The overall problem of searching a genome for a number of predefined DNA patterns remained unchanged from that of SMS. The desired inputs and outputs of the program were identical. However, in order for the problem to be developed for implementation in a parallel environment the basic processes would need to be redefined. In this case, both a client and a host program would need to be designed; the host would be responsible for accepting as input from the user, information regarding the genome file to be scanned as well as the number of clients (processors) to be employed for the task. Once inputted, this information would need to be directed to the client programs so that they may extract a subsequence of the genome to scan, and then direct the output generated by the scan back to the host program. The host would need to measure the total search time and assume responsibility for the output of the results in an intelligible form to the user.

4.1.2 PMS Design

The design of PMS required that two independent programs be designed, one to serve as the host program and the other to serve as the client program. The host was designed to receive as input via a Graphical User Interface (GUI) the genome file to be scanned as well as the number of clients to be employed. This information would need to be directed to the client machines so that they may process and execute the given task. The host program would need to remain active in order to receive the resultant data from the client programs. Upon arrival the data would be processed to ensure that there were no duplicate results. All processed data would be written to a file and sent back to the GUI so that the user may visualise the data. An additional task required of the host was to ensure that the total search time was measured and written to a file for storage.

The client program would need to receive the information regarding the genome file to be scanned as well as the number of clients required. Based on the information received from the host, each client would know which genome file was to be scanned, the total number of clients required for the task and which client they were. They would then need to compute the start, end and seek positions in order to read the correct sequence of characters from the genome file so as to divide the work evenly amongst the available clients. In order to compute this, they required both the total number of clients and the actual client number (individual client ID). Once computed, the client would extract and load their desired subsequence and the list of regex's to be scanned. They would cycle through the list of regex's and scan each one separately. When the list was exhausted, all results would be directed to the host program for processing and final presentation.

The TSpaces framework developed by IBM was to be the framework of choice for enabling the desired communications via the GUI and the host program, and between the host and client programs. The communication between the GUI and the host program is required so that the information regarding the genome file to be scanned and the number of clients to be used can be directed to the host program. This is essential as the host program requires this information in order to generate the exact number of tuples as there are clients. Each tuple contains the name of the genome file, the total number of clients and finally a unique client ID. Communication between the host and the GUI is again required in order for the results that have been processed and formatted by the host program to be directed to the GUI so that the user may visualise the output.

TSpaces is also required to ensure that the information, originally entered into the GUI, is relayed from the host to each client. Once each client has successfully scanned their specific section of the genome, which is calculated based on the client ID and the total number of clients, the results are then deposited back into the tuplespace so that the host program may collect them.

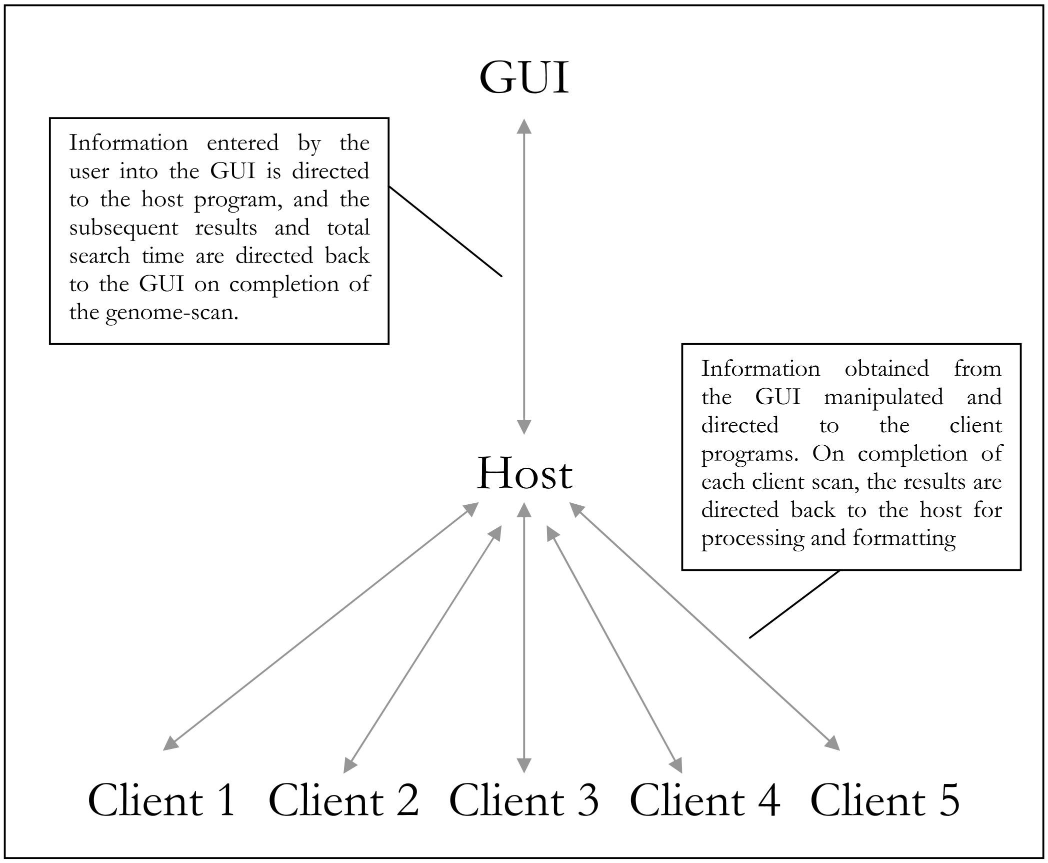

GUI
Information entered by the user into the GUI is directed to the host program, and the subsequent results and total search time are directed back to the GUI on completion of the genome-scan.
Host
Information obtained from the GUI manipulated and directed to the client programs. On completion of each client scan, the results are directed back to the host for processing and formatting
Client 1
Client 2
Client 3
Client 4
Client 5

Figure 7: Schematic overview of communication requirements for PMS.

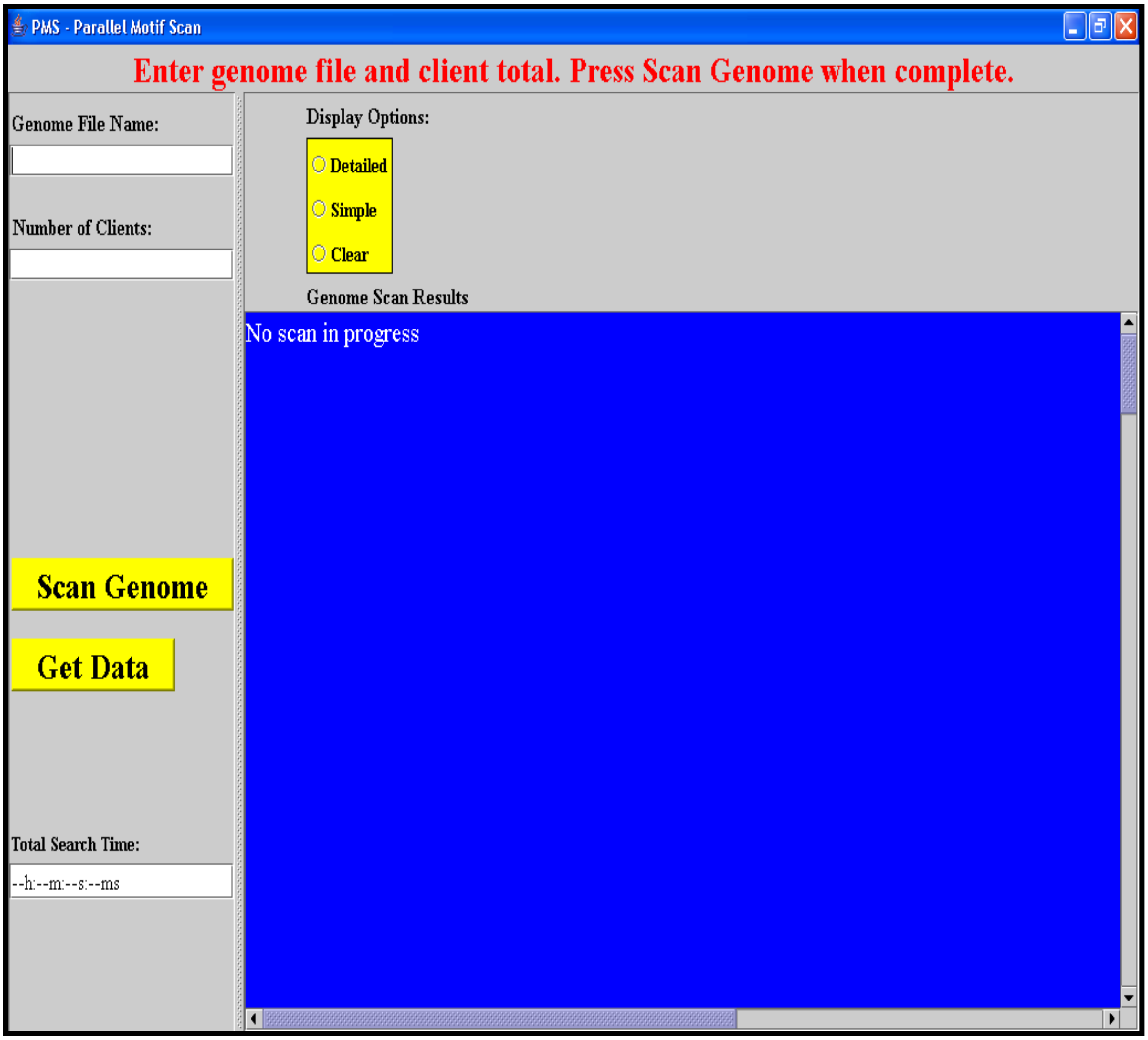
4.2 PMS Host Development

It was decided that the host program would need to consist of the following steps, thus requiring that a class containing the necessary methods be written for each step.

- Create a GUI to accept user input and to display results.
- Receive the information from the GUI.
- Send data including the genome file name, the total number of clients and the actual number to the client program.
- Collect all results returned by the clients.
- Combine and process the results to ensure that no duplicates are recorded.
- Send all results including the total search time to the GUI.

The first action of the host program (PMS_Host) was to create the GUI (Figure 8 illustrates a screenshot of the GUI) that was to serve as a means of obtaining the information required by the client programs, and to display all results on completion of the experiment. Communication between the host and the GUI would be achieved by writing and reading tuples to and from the tuplespace located on a TSserver running locally on the network.

The user would enter the name of the genome file to be scanned, as well as the number of clients to be used, in the text fields provided. Once this information was entered the user would need to click on the Scan Genome button in order to initiate the scan. In order for the user to receive and visualise the results from the genome scan, the Get Data button would need to be pressed. On completion of the task, the total search time was displayed in a text field in the bottom right of the GUI, and the message in the results text field would display 'Genome Scan Complete'. The user was then able to visualise the results by selecting either the detailed or the simple results options.



PMS - Parallel Motif Scan

Enter genome file and client total. Press Scan Genome when complete.

Genome File Name:

Number of Clients:

Display Options:

○ Detailed
○ Simple
○ Clear

Genome Scan Results

```
No scan in progress
```

Total Search Time: --h:--m:--s:--ms

Figure 8: Screen shot of the GUI used as the interface between user and the host program, PMS_Host, which allows the user to specify the number of clients required to scan the genome file entered.

4.2.1 GUI_Info

The GUI_Info class was written to connect to the tuplespace and to take the tuple labelled "ClientInfo". The information concerning the genome file and total number of clients was extracted, and two methods were written to return the relevant information to the host program. The following code was required to connect to the tuplespace and to take the desired tuple:

TupleSpace ts = new TupleSpace("PMS",HOST)

- Create an instance of the tuplespace called PMS, which is found at the server known as HOST (dell.ict.ru.ac.za).

Tuple tempInfo = new Tuple("ClientInfo",new Field(String.class))

- Create a template tuple that will match any tuple with identical values in the respective fields. i.e. field(0) = "ClientInfo", and field(1) = a string object.

SuperTuple info = ts.waitToTake(tempInfo,300*1000)

- Create an instance of a supertuple to store the matching tuple from the tuplespace.

Once the tuple has been collected, the relevant information needs to extracted. This was achieved by retrieving the data that was stored in the second field, field(1), of the tuple since the first field, field(0), contained the unique id for the tuple. The following code was used to extract the string of data and then to split it into a String array:

String temp = (String)info.getField(1).getValue();

- Create an instance of a string object to store the string stored in the second field (field(1)) of the tuple taken from tuplespace.

String[ ] clientInfo = temp.split(",");

- Create an instance of a string array to store each element which results from the split( ) method. The comma is used as the element separator.

GUI_Info
- ClientInfo : String[ ] - HOST : String
+ getHost( ) : void + genomeName( ) : String + clientTotal( ) : int

Figure 9: UML Class Diagram for the GUI_Info class

Two further methods make up the GUI_Info class which result in the genome file name to be scanned and the total number of clients being returned to the main method of the host program. The genomeName( ) method returns a string containing the genome file name, and the clientTotal( ) method returns an integer containing the total number of clients for the task. The genome file name and client total are then passed as parameters to a method within the next class written.

4.2.2 ClientInfo

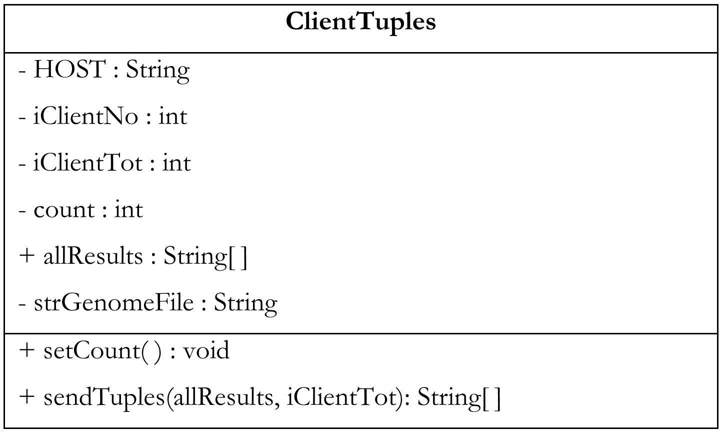
The class known as ClientInfo was written in order to generate the information required by the clients. The void clientFile(String, int) method requires both the genome file and the client total as parameters and creates a one-dimensional String array with a length equal to the number of clients. A for-loop was utilised to create an equal number of String objects as there are clients. Each String object contains the file name to be scanned, an Integer object representing a unique client ID in the range of 0 to the total number of clients and the total number of clients. Each object is stored in a String array, and returned to the host program via the getFiles( ) method that returns the one-dimensional array of client files.

ClientInfo
+ iTotal : int - myClients : String[ ] - strMyGenome : String
+ clientFile(strMyGenome, iTotal) : void + getFiles( ): String[ ]

Figure 10: UML Class Diagram for the ClientInfo Class.

4.2.3 ClientTuples

The ClientTuples class was written to receive the one-dimensional array of String objects containing the client information, extract each element of the array and place the string in a tuple labelled 'ClientTuple'. The one-dimensional array as well as the total number of clients were received as parameters by the sendTuples( ) method. This method not only distributes the client tuples to the tuplespace, but prior to doing so, checks to see if there are any remaining tuples from previous genome scans. Any remaining tuples were removed prior to the program distributing the tuples for the clients. A final check was also done to ensure that the correct number of tuples was distributed; this was achieved by checking that the number of tuples sent corresponded with the total number of clients. In the event of there being a mismatch the program was designed to exit.



ClientTuples

- HOST : String
- iClientNo : int
- iClientTot : int
- count : int
+ allResults : String[ ]
- strGenomeFile : String

+ setCount( ) : void
+ sendTuples(allResults, iClientTot): String[ ]

Figure 11: UML Class Diagram for the ClientTuples Class.

4.2.4 CollectClientResults

Once all client tuples were distributed to the tuplespace for collection by the clients, the host program would then wait for the client results to be returned to the tuplespace. The CollectClientResults class was designed to collect each result tuple on its arrival in the designated tuplespace. A collectResults(int, int) method was written to receive the total number of clients and regex's to be scanned as parameters. This information would be required in order to create a two-dimensional String array to store all client results. The total number of clients would represent the number of rows and the total number of regex's would correspond to the number of columns required to store all the client results.

CollectClientResults
- searchResults : String[ ] - clientResults : String[ ] [ ] - iTotal : int - iRow : int - iCol : int - iClientID : int - iMotifTot : int - HOST : String
+ collectResults(iTotal, iMotifTot) : void + getClientResults( ): String[ ] [ ]

Figure 12: UML Class Diagram for the CollectClientResults class.

The total number of clients would also be required to ensure that the correct number of result tuples were received, i.e. that the number received corresponded with the number of clients employed. Once all results were received, they were combined and stored in the two-dimensional array. The getClientResults( ) method returns the populated two-dimensional array to the host program for further processing and display of results.

The final requirement of the PMS_Host was to ensure that there were no duplicate results, which may have arisen due to the 500-character sequence overlap between the clients, and then to display the results in a user-friendly format.

4.2.5 SortResults

The SortResults class was written to provide the necessary methods required for the final task. The first method within SortResults, namely Results(String[ ] [ ]), requires the two-dimensional array containing all client results as its parameter. The only requirement of this method was to extract all the data from the two-dimensional array and combine it to form a one-dimensional array representing the final results for the genome scan. The second method of the SortResults class, sortResults(String [ ]), receives the one-dimensional array of all client results as it is responsible for the removal of duplicates and formatting the results so that they can be viewed by the user on the GUI. Each element of the array represents the total number of matches found per regex scanned, with the first element, [0], representing all the results for the first regex in the list of motifs or domains, and the last element of the array representing all the results obtained for the last regex in the list of motifs/domains.

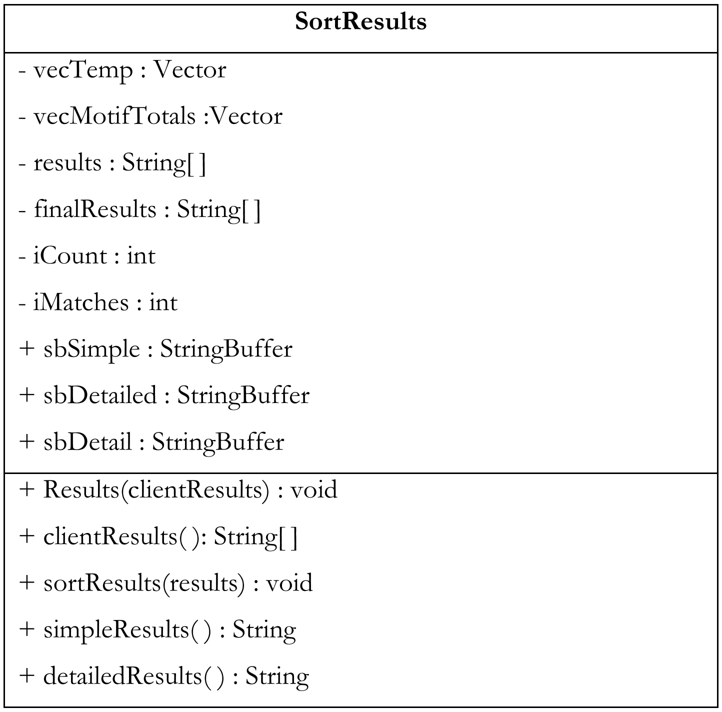

SortResults
- vecTemp : Vector
- vecMotifTotals :Vector
- results : String[ ]
- finalResults : String[ ]
- iCount : int
- iMatches : int
+ sbSimple : StringBuffer
+ sbDetailed : StringBuffer
+ sbDetail : StringBuffer
+ Results(clientResults) : void
+ clientResults( ): String[ ]
+ sortResults(results) : void
+ simpleResults( ) : String
+ detailedResults( ) : String

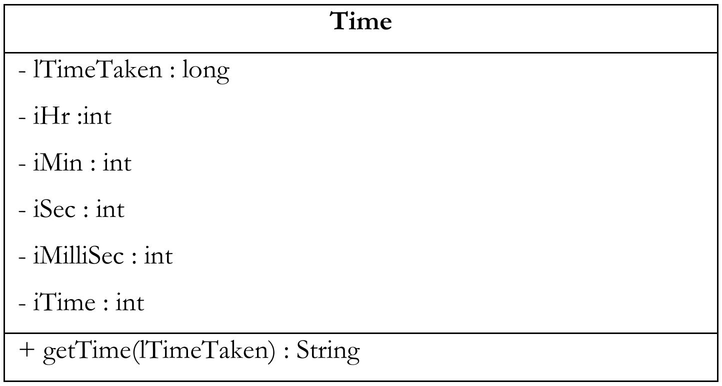
Figure 13: UML Class Diagram for the SortResults class.

The value of each element was either "null" where no matches were found, or it was a string containing details such as the start and end positions of all matches, with a comma (,) separating each match found. This method also reads in the file containing the list of regex's in order to extract the specific name of the protein motif/domain for presentation with their corresponding results. Where matches are found they are separated using the split( ) method and placed in a temporary String array from where they are immediately placed in a hash set. A hash set was chosen to store the final results, as it presented a simple, yet effective, technique to ensure that no duplicate results were stored. The add(Object o) method achieves the removal of duplicates due its mechanism of action as it only adds the specified element to the set if it is not already present, and therefore ensures that no duplicates are stored and recorded in the results.

The SortResults class formats the results so that the user has the option to view both a detailed and a simple set of results each time the program is run. The simple results contain only the results for those regex's that produced matches, and thus displays the number of the regex scanned (in this case a number between 1 and 100) and the total number of matches found for that particular regex. However, the detailed output contains the results for each regex scanned, whether or not any matches were found. Detailed output includes the number of the regex scanned, the name of the regex and whether or not any matches were found. In the case of no matches being found, 'No matches found...' was displayed, and where matches are found, all matches are displayed. The data displayed for each match consists of the total number of matches found, the specific match number and the start and end position numbers in the genome.

4.2.6 Time

A key requirement of PMS_Host was to assume the time-keeping responsibility for the genome scan. This was achieved by obtaining as the start time the system time in milliseconds as soon as all the client tuples were deposited into the tuplespace. The end time was computed by again obtaining the system time, and this occurred after all results had been received, processed and formatted. The start time was subtracted from the end time to calculate the total search time in milliseconds. The Time class was written to accept the search time in milliseconds and return the time in hours, minutes, seconds and milliseconds. The formatted time and the simple and detailed results were then written to three separate files, and sent to the GUI for display purposes.



Time

- lTimeTaken : long
- iHr :int
- iMin : int
- iSec : int
- iMilliSec : int
- iTime : int

+ getTime(lTimeTaken) : String

Figure 14: UML Class Diagram for the Time class.

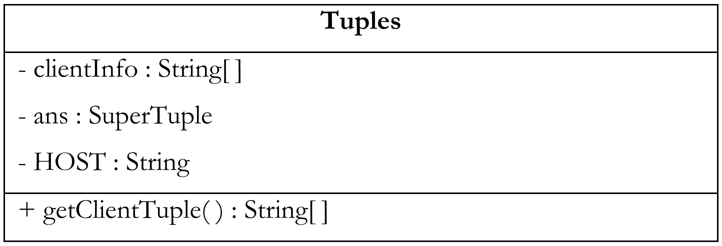
4.3 PMS_Client Development

The client program (PMS_Client) was found to require fewer steps than the host program, since the client's primary objective would be to load both the genome sequence data and the file containing the list of regex's. The genome would then be scanned and the results sent back to the host. The following steps were required in order to achieve this:

- Receive tuple sent from the host program.
- Extract information from received tuple.
- Compute subsequence of genome to scan.
- Load subsequence and regex's.
- Scan genome.
- Send results to host program.

4.3.1 Tuples

The Tuples class is responsible for the collection of the tuples deposited by the host program from the designated tuplespace. Each tuple contains a String object containing the genome file to be scanned, a unique integer ID (actual client number) and the total number of clients. The aforementioned data is represented as a single string with each element being separated by a colon (:). The colon is then used to split the data and place each element into a String array, which results in an array of three elements being formed. This array is then returned to the client program so that the information may be extracted and the scan can commence. In the event that the sequence is to be split by a factor that is larger than the number of clients, a do-while loop was incorporated that waits to collect tuples for a defined period of time; this allows each client to also process more than one job per experiment.



Tuples

- clientInfo : String[ ]
- ans : SuperTuple
- HOST : String

+ getClientTuple( ) : String[ ]

Figure 15: UML Class Diagram for the Tuples Class.

The information contained within the string array provides the client program with information relevant to the current experiment. The first element reveals the genome file that is to be read and searched, the second element contains the unique client ID, and the third element reveals the total number of clients to be employed in the given experiment. This information is crucial to ensure that the correct genome file is loaded as well as ensuring that the job is divided evenly between the available clients.

4.3.2 ClientSequence

A number of options were considered, developed and evaluated in order to divide the genome sequence evenly amongst the clients. The first option was to create a class to divide the genome sequence prior to each experiment. However, it was soon decided that this would not be a viable or efficient process and was soon replaced. The second option entailed each program reading in the whole genome file and then extracting a substring, using the substring(int beginIndex, int endIndex) method which returns a new string that is a substring of this string. This option again proved to be fruitless in that the memory requirements were excessive for large genomes. An increase in the Java virtual memory allocation proved to be an insufficient means of solving this problem even with an increase to the maximum value (i.e. from the standard memory allocation of 64 MB to the maximum of 512 MB).

The third option, presented the 'cleanest' and most efficient means of solving the problem, and revolved around using a RandomAccessFile object to extract the desired sequence of characters from the genome file. The read (byte[ ] b, int offset, int length) method for a RandomAccessFile is utilised in order to divide the genome evenly. The read method simply reads up to length bytes of data from this file into an array of bytes. The offset value may also be set and this is achieved by calling the seek(long position) method which enables the offset value to be adjusted according to client number.

ClientSequence
- lMySeekPos : long - strMyGenome : String - sbSequence : StringBuffer - strFileName : String - strClientFileName : String - iMyClientNo : int - iMyClientTot : int - iMyClientStart : int - iMyClientEnd : int - iSequenceLength : int - iCount : int - iSubSequence : int - iLength : int - iArraySize : int + lMyStart : long + strClientSequence : String + iClientNumber : int + iClientTotal : int + iGenomeSize : int + iMyStart : int + iMyEnd : int
+ getPositions(strMyGenome, iMyClientNo, iMyClientTot) : void + getStart( ) : int + getEnd( ) : int + getSeekPosition( ) : long + getFileSize( ) : int + initialiseSequence(strGenomeIn, iStart, iEnd, iFileLength, lSeekPosition) : Boolean + getFile( ) : String

Figure 16: UML Class Diagram for the ClientSequence Class.

This resulted in the design and development of the ClientSequence class which was responsible for dividing the genome, computing the seek, start and end positions for the client and reading the desired set of characters from the genome file. The getPositions(String, int, int) method receives the genome file name, the unique client number and the client total. The size of the genome file is computed and then divided by the number of clients entered by the user, the result being the size of the file segment that each client should scan.

The unique client ID is then used to compute the relative start and end positions, which are needed to calculate the length or number of bytes, to be read into the byte array storing the genome file characters. The start position is also used to serve as the seek position. Once computed, the desired segment of the file is accessed and stored as a String. The initialiseSequence( ) method receives information including the file name, the start, end and seek positions as parameters and returns true once the file has been successfully accessed and the sequence has been read into a string. False is returned in the event of an error in accessing or reading the file. A simple getFile( ) method is utilised to return a String representation of the genome sequence to the main method for later use.

Once complete, the Patterns class is required in order to read in a file containing the list of regular expressions to be used in the genome scan. All regex's found in the file are read in and appended onto the end of a vector, resulting in a vector of regex's. Similarly with the initialiseSequence method, so too does the initialiseMotifs( ) return true in the event of successful extraction of the regex's and false in the event of an error. The vector containing all regex's is returned to the main method of PMS_Client so that the genome scan may commence. A for-loop is then used to extract each regex from the vector and passed along with the genome sub-sequence to be scanned to the findMotif(String, String) method which uses the java.util.regex package to scan the genome, as described for SMS.

Once all regex's have been scanned the string containing all the search results is deposited into the tuplespace so that the host program (PMS_Host) may combine and process all client results.

In order for the program to be executed in parallel, a host program was designed and developed primarily to create a GUI that enables the user to provide the name of the genome file to be scanned and the number of clients to be used. This information is required by the client programs that are responsible for extracting a particular section of the genome in question, and are using the list of regex's as patterns to be searched for within the genome. The host program requires all results so that any duplicates can be removed which have arisen due to the overlap in sequences assigned to each client. Once all results have been processed and formatted they will be directed to the GUI for visualisation, and then written to file.

Chapter Five

Results

5.1 Experimental Overview

In order to analyse the effects of parallelism and to highlight its possible role in the field of bioinformatics, it was decided to use a range of different sized genomes. The following size genomes were analysed using a varying number of clients:

- 60 MB
- 140 MB
- 250 MB
- 1 072 MB / 1.072 GB

The effects of parallelism were assessed by calculating the level of speedup (S), where speedup is calculated by dividing the time the serial program takes to run (T1) by the time it takes to run the same problem with N processors (T(N)).

$$S = T1 / T(N);$$

Each genome file was initially executed using a single processor (i.e. 1 client). The number of clients was then incremented by a factor of five until the calculated speedup was found to plateau. For each genome scanned and for each number of clients tested, either three or five runs were repeated to ensure the statistical significance of the data collected. The various sized genomes were all initially run on a single processor machine so that a standard protocol for all experiments undertaken could be established.

It soon became apparent that the most efficient serial algorithm which would serve as the benchmark for the calculating the speedup with each genome tested was, in fact, PMS and not SMS. This was directly attributed to the memory demands being placed on the system when the size of the genomes increased above 60 megabytes (MB). The Java Virtual Machine Memory was increased to the maximum allowance of 512 MB. However, an OutOfMemoryError was thrown when SMS attempted to load the larger genome files (> 60 MB). This error is thrown when the Java Virtual Machine cannot allocate an object because it is out of memory, and no more memory can be made available by the garbage collector.

In order for the genomes larger than 60 MB in size to be scanned using a single processor machine, the genome needed to be divided into smaller chunks so that the single processor may process a job in a series of steps. This alone indicates the necessity for parallel computing. The search time associated with the best "serial" algorithm was thus achieved by having one client program running and entering more than one client in the GUI. The single client would then process all tuples deposited into the tuplespace.

All files including the genome sequence files, the list of regular expressions, the Java source code for both the host and the client programs and the TSpace package files (tspace.jar, tspaces_client.jar, tspaces_fixes.jar) were stored in a common directory accessible by all the machines registered on the local network. A subdirectory was created for both the client and host source code. All machines required for a run executed either the client or host program from the /mnt/exports/takhurst/Clients/ or the /mnt/exports/takhurst/PMS_Host/ directory paths, respectively.

5.2 PC Configuration

The network of workstations consisted of a number of commodity PCs, all of which are in possession of an Intel® Pentium® 4 2.4 Ghz processor with 512 MB of RAM. A fast switch Ethernet network connection of 100 mbps is used to create the desired network. Each machine is utilising the Red Hat Linux 3.1 10 version as its operating system and is in possession of the Java 1.4.2_03 version.

5.3 Results

5.3.1 60 MB Genome File: chromo20.fa

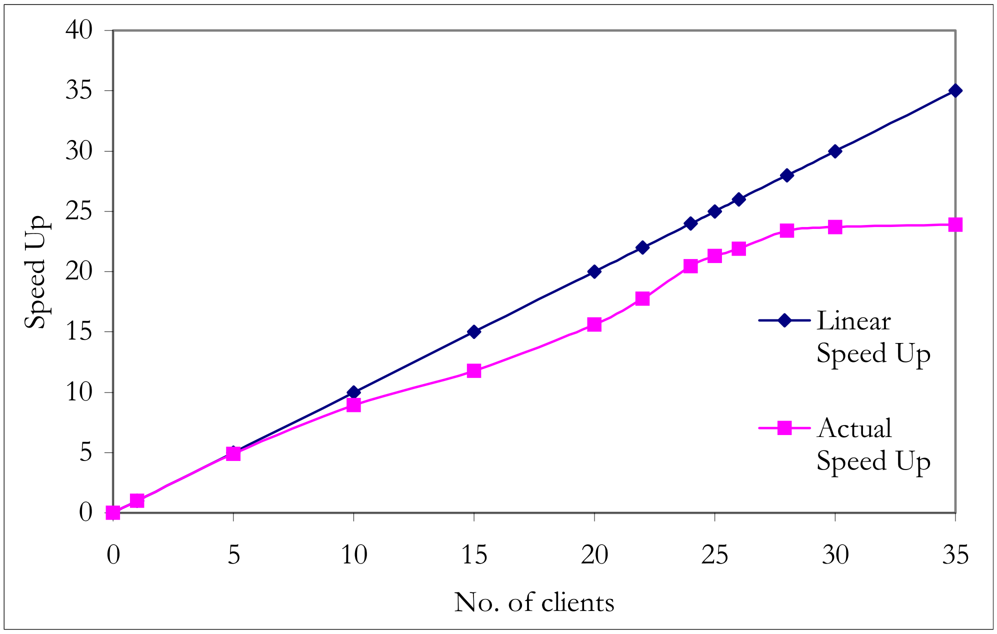


Figure 17: Average speedup achieved over a range of clients (processors) using the 60 MB file, representing human chromosome 20.

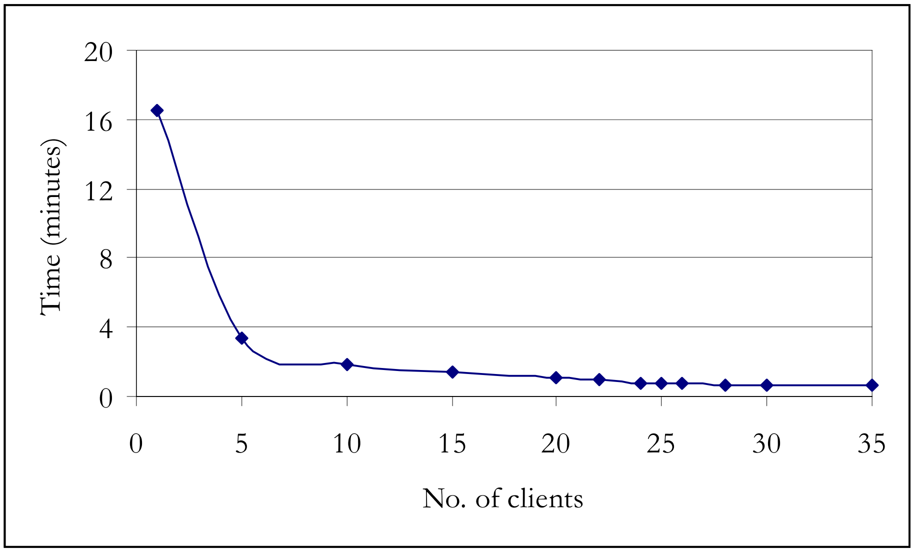


Figure 18: Graphical representation of the reduction in wall-clock time achieved for the 60 MB file.

The first 'genome' scanned was the 20th human chromosome which was 60 MB in size. Near linear speed was achieved up until 10 clients, the speedup increased steadily until 28 clients, whilst after that any further increase resulted in little to no increase in speedup (See Figure 17). The average execution time using a single processor and the whole genome file was found to be 0h:16m:29s:303ms; this was significantly reduced to approximately 0h:0m:42s:320ms when utilising 28 processors.

Figure 18 highlights the reduction in wall-clock time which can be seen to rapidly decrease from 0h:16m:29s:303ms to 0h:3m:21s:728ms when representing the reduction from using a single processor (client) to using 5 clients. There was a further significant reduction in wall-clock time between 5 – 10 clients, after which the wall-clock time gradually decreases to a minimum of 0h:0m:41s:397ms with 35 clients.

5.3.2 140 MB Genome File: chromo9.fa

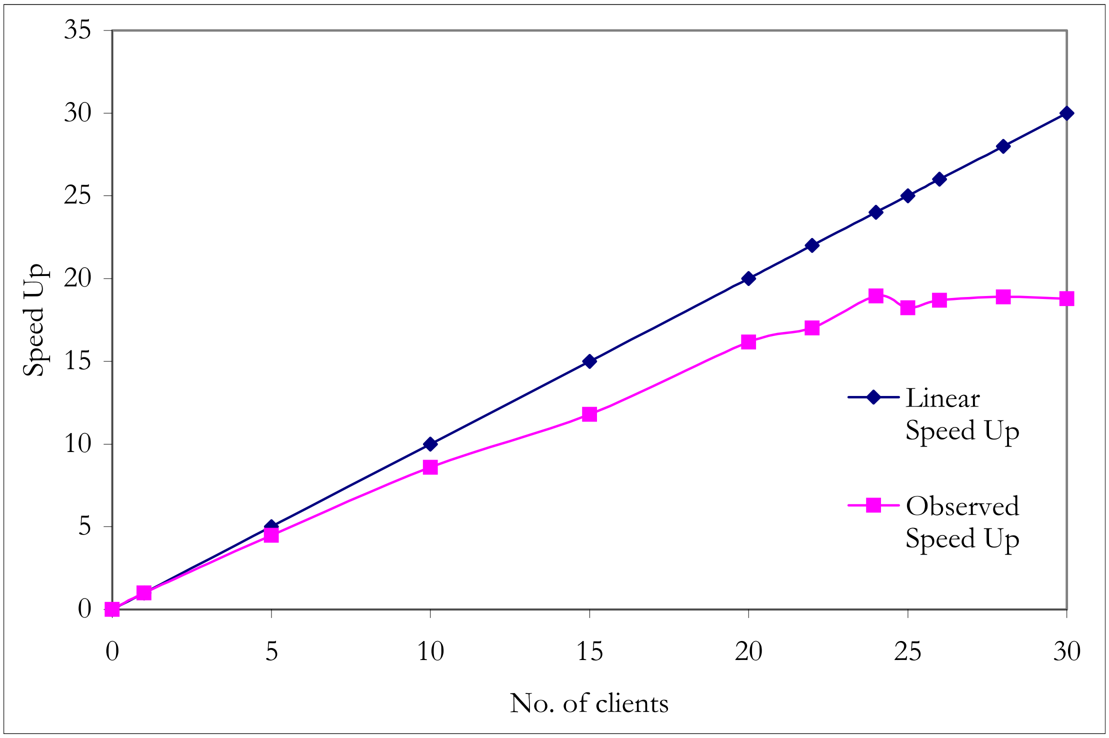


Figure 19: Graphical representation illustrating the speedup achieved using a genome of file size 140 MB (megabytes), which represents the ninth human chromosome.

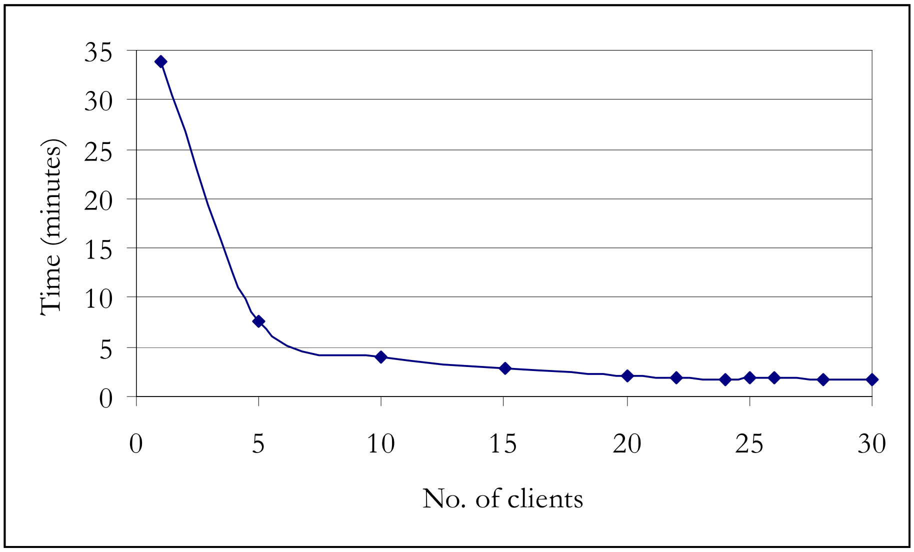


Figure 20: Graphical representation of the reduction in wall-clock time achieved for the 140 MB file.

The second 'genome' scanned represents the ninth human chromosome. Again near linear speedup is exhibited up to 10 clients (processors), after which the speedup can still be seen to increase until an optimal number of approximately 24 clients is reached. Any further increase in client number above 24 can be seen to have little to no effect on the speedup as the curve can be seen to plateau (See Figure 19). This represents a reduction in wall-clock time from approximately 33 minutes on a single processor machine and with the genome split into 3 smaller slices, to approximately 1 minute and 45 seconds using 24 machines.

In a similar profile to that obtained for the first 'genome' scanned, the greatest reduction in wall-clock time can be seen between clients 1 – 10. This represents a decrease from 0h:33m:53s:990ms to 0h:3m:56s:767ms with 10 clients. The wall-clock time then gradually decreases until a minimum, which was found to be with the use of 24 clients and equalled 0h:1m:47s:403ms (See Figure 20).

5.3.3 250 MB Genome file: chromo1.fa

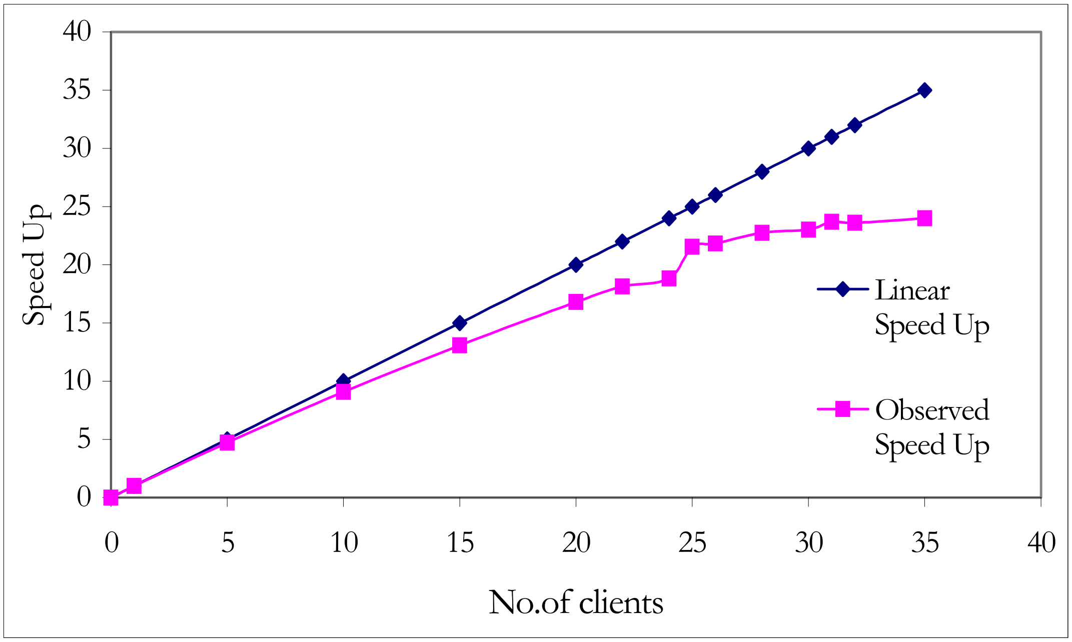


Figure 21: Graphical representation illustrating the speedup attained for the genome of file size 250 MB, which represents the first human chromosome.

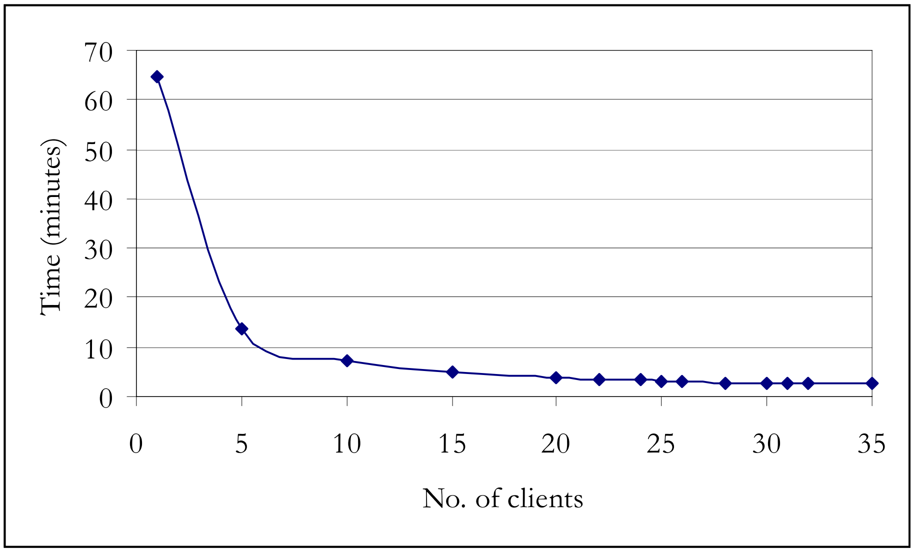


Figure 22: Graphical representation of the reduction in wall-clock time achieved for the 250 MB file.

A similar trend was observed for a larger file tested (See Figure 21), and as for the 60 MB and the 140 MB files, the 250 MB file exhibits near linear speedup. This is seen to occur up until about 15 clients whereby the speedup begins to increase less sharply until it starts to taper off around 25 clients. The slope can be seen to increase very slightly until a maximum speedup is reached around 35 clients. However, the optimal number of clients would appear to be in the region of 25 since any further increase in clients can be seen to have negligible effects on the speedup. This result represents a reduction in wall-clock time from approximately 1 hour and 5 minutes to approximately 3 minutes when one compares the execution time using a single client to that obtained with 25 clients.

The wall-clock time associated with each number of clients tested can be seen in Figure 22, and in this case the most dramatic reductions occur up until 15 clients. A gradual reduction in wall-clock time can then be seen up to about 25 clients after which the reduction in wall-clock time is negligible, if anything at all. The minimum wall-clock time was achieved with 35 clients, and this resulted in the scan taking 0h:2m:42s:42ms as opposed to the 1h:4m:47s:325ms obtained with the use of a single processor and with the genome divided into 5 pieces.

5.3.4 1072 MB Genome File: chromo1-5.fa

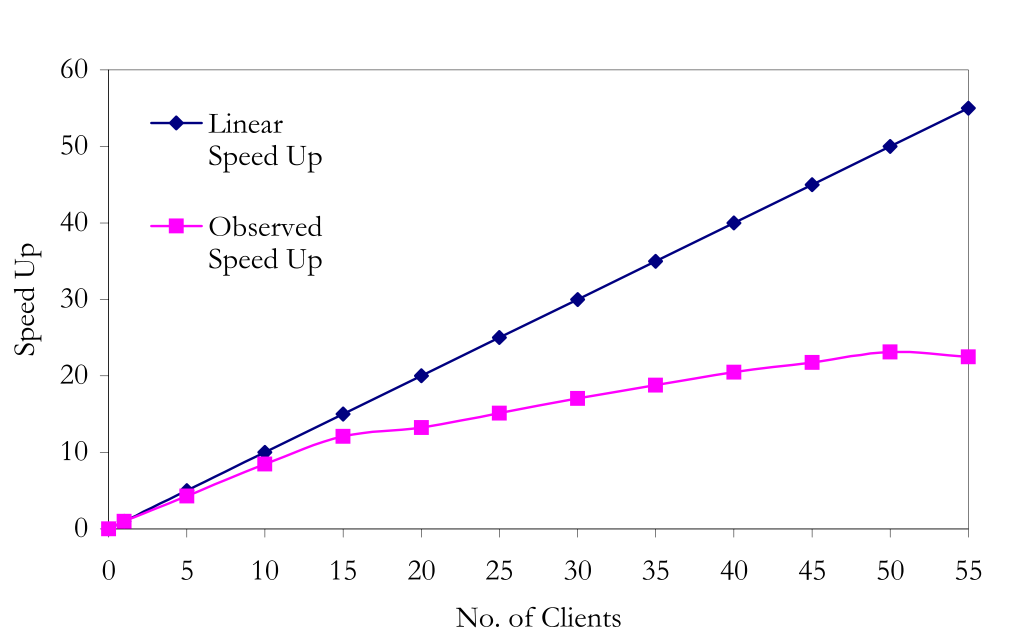


Figure 23: Graphical representation illustrating the speedup attained for the genome of file size 1072 MB, which represents the first five human chromosomes (approximately a quarter of the human genome).

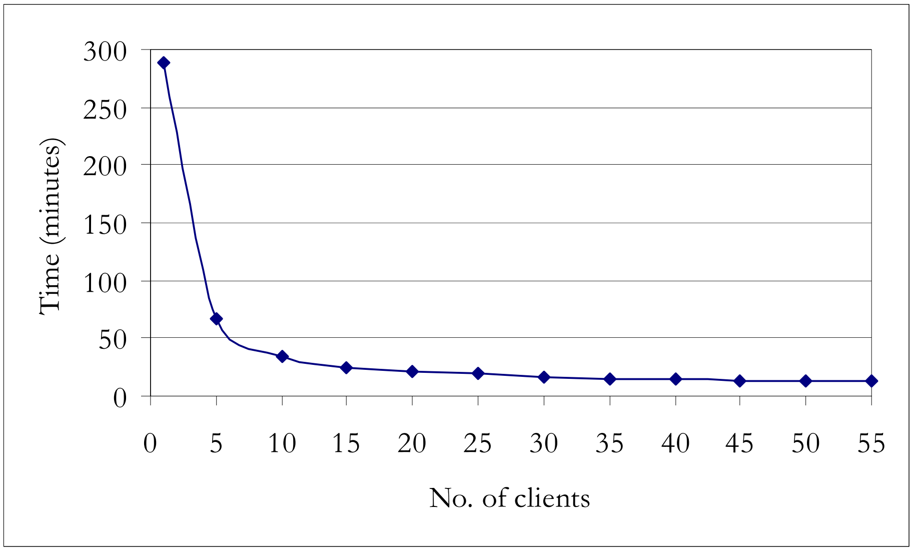


Figure 24: Graphical representation of the reduction in wall-clock time achieved for the 1072 MB file.

The largest 'genome' scanned, and which represents a combination of the first five chromosomes, once again exhibited a similar trend to all the other 'genomes' tested in that it, too, achieves near linear speedup until 15 clients. The speedup can then be seen to gradually increase between 15 – 50 clients where the maximum speedup was attained. The addition of more than 50 clients can be seen to have negligible effects (See Figure 23). An optimal number of clients would be in the region of 20 clients and this represents an overall reduction in wall-clock time from approximately 4 hours and 48 minutes to 21 minutes and 25 seconds. The greatest reduction in wall-clock time can be seen for 50 clients, in which the total search time was roughly 12 minutes and 30 thirty seconds.

The reduction in wall-clock time also exhibits similar trends to those observed for the previous 'genomes' tested in that the greatest reduction can be seen between 1 – 15 clients. This represents a decrease from an average scan time of 4h:48m:6s:606ms with one client (processor) and the genome split into 20 pieces, to 0h:23m:50s:925ms with the help of 15 clients. The wall-clock time can then be seen to gradually decrease until a minimum was obtained with 50 clients, which corresponds to 0h:12m:27s:503ms (See Figure 24). When one compares the speedup attained with 15 clients with that attained using 50 clients, the wall-clock time can be seen to decrease by a further factor of 2.

Chapter Six

Discussion and Conclusion

6.1 Discussion

The principle objective of this research project was to investigate the role of parallel computing in the field of bioinformatics. In order to achieve this, a suitable bioinformatics application needed to be decided upon, designed, developed and implemented. Subsequently, two further research objectives arose; these were firstly, to assess the role of the Java programming language in scientific computing, and secondly, to assess and investigate the use of a parallel framework based on the Linda model.

All three objectives were found to fit neatly into the scope of the research project, since a suitable biologically based problem that would require parallel computing in order to reach a solution, would need to be defined. Once the problem was defined, a suitable programming language would be selected in order to develop and implement the program responsible for solving the problem. The choice of the programming language was a simple one in that Java possesses a large number of important features that make it a suitable candidate. In order to investigate the effects of parallelism on the program, it was decided to use a parallel framework based on the tuplespace model that was first introduced by David Gelernter and his research group at Yale University. They developed a system known as Linda, which represents a parallel programming language that is easy, efficient and portable (Gelernter, 1988). TSpaces was selected, which is a freely available parallel framework written in Java that was developed and actively maintained by IBM.

One of the biggest challenges facing modern bioinformaticians has resulted from the dramatic increases being experienced in a number of genomic databases, such as EMBL, enBank and SWISS-PROT. These databases have been seen to almost double in size every year (Janaki and Joshi, 2003). There are two factors that have directly led to the explosion in publically available genomic sequence data. These factors can be attributed firstly to a number of the larger genomic research facilities generating upwards from several hundred gigabytes of data per day, and secondly, to the development and implementation of high-throughput techniques for both DNA sequencing and analysis of gene expression (Meloan, 2004 and Bader, 2004).

Subsequent to this growth in genomic sequence data, there is a need to intelligibly capture, manage and analyse the data so that important discoveries can be made. Thus it was decided to design, develop and implement a program that would scan whole genome sequences for a number of predefined motif or domain signatures in the form of regular expressions. When one considers that the size of the complete human genome is in the region of 4 gigabytes, it becomes increasingly clear that problems of this magnitude will require computational power in excess of that harnessed with the use of a single processor machine.

The result was the development and implementation of a genome-scanning program, known as PMS (Parallel Motif Scan), which was written in Java and utilised the TSpaces parallel framework in order to achieve the desired communication required for parallel execution. The effects of parallelism were assessed using the DNA sequences from the first, ninth, twentieth and a combination of the first five human chromosomes. The sizes of the various chromosomes were 60 MB (chromosome 20), 140 MB (chromosome 9), 250 MB (chromosome 1), and 1072 MB or 1, 072 GB (chromosomes 1 to 5).

The aforementioned chromosomes were to serve as the range of 'genomes' to be tested. Each 'genome' was initially executed utilising a single client, which represents a single processor, to serve as the benchmark to calculate the speedup associated with parallelism. The use of an existing network of workstations provided the necessary environment to execute the program in parallel. Each 'genome' was executed with varying numbers of clients (processors) and the resultant speedup was calculated; the number of clients was increased until no further increase in speedup was obtained.

An important finding when executing each 'genome' with a single client was that for genomes larger than 60 MB, the use of parallel computing is essential. This is supported by the fact that the 140 MB file needed to be split into three pieces, the 250 MB file into five pieces and the 1072 MB file into twenty pieces in order for these 'genomes' to be scanned using a single processor. The single client would then search each piece until the entire 'genome' had been scanned. The reason that these 'genomes' required processing as a number of smaller pieces was due to limitations in the amount of memory as each client possessed a maximum of 512 MB of RAM. The result of this was that the Java Virtual Memory could only be increased to the maximum of the client which, as previously mentioned, was 512 MB.

When the single client attempted to read in the genome files greater than 60 MB, an OutOfMemoryError was thrown which implies that the Java Virtual Machine cannot allocate an object as it is out of memory, and no more memory can be made available by the garbage collector. Dividing the problem into smaller pieces, which would not compromise the memory limitations of the individual clients, averted this error.

The effect of parallelism for each 'genome' with respect to speedup was immediately realised, and can be seen in the respective figures illustrating the associated speedup for all number of clients tested with the various 'genomes'. All 'genomes' tested exhibited near linear speedup for a total of 10 clients for the 60 MB (Figure 17) and 140MB (Figure 19) and for a total of 15 clients for the 250 MB (Figure 21) and 1072 MB (Figure 23) genome files.

The graphical illustrations representing the speedup for the various 'genomes' tested can be seen to follow almost identical trends for the 60 MB, 140 MB and the 250 MB 'genomes' wherein, after the initial near linear speedup, the speedup in each case increases steadily until approximately 25 clients. Any further increase in clients after this number can be seen to have little to no significant effect on the speedup. A similar trend was found for the largest genome tested. However, the speedup was found to increase gradually up to 50 clients, with any further increase in clients resulting in a slight decrease in speedup. These findings suggest that above a threshold number of clients (processors), any further increase in the number of clients may, in fact, be counter-productive or, as in this case, result in negligible gains in speedup.

An interesting finding was that the speedup attained for each 'genome' tested with 25 clients was found to be greater than that obtained by Kleinjung *et al* (2002), who investigated the use of parallel computing for performing multiple sequence alignments.

Kleinjung *et al* reported to have found that the parallelised program performed up to ten times faster on 25 processors compared to the single processor version. The computed speedups were found to be 21.3 for the 60 MB 'genome', 18.3 for the 140 MB 'genome', 21.5 for the 250 MB 'genome' and 15 for the 1072 MB 'genome' when the program was executed on 25 processors. All these results are in excess of the speedups reported by Kleinjung *et al* (2002) for the same number of processors, albeit for a different application.

The fact that all the 'genomes' tested produced similar profiles when the speedup was plotted against the number of clients utilised, together with the fact that they also all appeared to exhibit a similar number of clients as being the optimal, suggests that the limitation of the parallel environment may be due to the demands being placed on the network. The reason for this is that each client machine is running the genome-scanning program from the same-shared directory. The file for the 'genome' to be scanned is also located in this shared directory and, as such, each client is attempting to access the same file in the same directory at the same time. As a result there could be a bottleneck effect as each client is attempting the same task at the same time.

This was particularly noticeable for the larger genome tested, and was perhaps due to the fact that the size the 'smaller' pieces are assigned for each client are in fact still rather large in terms of megabytes. This would mean that the time required to extract the desired section of the file is greater than is the case when the section is smaller, and thus each client takes slightly longer to extract their piece which in turn delays the other clients still needing to extract their particular section. In order to obtain concrete evidence for this, one could install network-monitoring software which would enable the user to monitor the demands being placed on the network due to each client attempting to extract their section from the centrally located directory.

There are a number of options which one could employ in order to avert this problem, one of which revolves around each client having all the desired genome files stored locally. This, however, would not be a suitable solution as it would require that the user needs to ensure that each client is in possession of all the required genome files. A more logical solution may be in having the genome files stored in a handful of shared directories with an equal number of clients accessing each directory.

Associated with the speedup are the more definitive reductions in wall-clock time, which provide dramatic evidence supporting the role of parallel computing in bioinformatics applications. When the wall-clock time was plotted against the number of clients, distinctively similar profiles were obtained for all ‘genomes’ scanned. All files scanned exhibited the largest reductions in wall-clock time between 10 to 15 clients, and this correlates with the time at which the speedup attained was closest to being linear. After this point any increase in the number of clients can be seen to have slight gains with respect to reductions in wall-clock time.

The reductions in wall-clock time associated with the 60 MB file can be visualised in Figure 18, wherein the greatest reduction is experienced with 30 clients and the initial scan time of 0h:16m:26s:303ms obtained with a single processor is reduced to 0h:0m:41s:397ms, representing an overall reduction of wall-clock time by approximately 95.81%. Figure 20 highlights the reductions in wall-clock time attained with the 140 MB and, as for the 60 MB, the overall reduction in search time was found to be approximately 94.7 % when the lowest search time achieved is compared with the search time obtained for the single processor. This represents a decrease in search time from 0h:33m:53s:990ms with one client to 0h:1m:47s:403ms with 24 clients.

Similar results were also attained for the two larger ‘genomes’ tested, the results of which are highlighted in Figures 22 and 24, for the 250 and 1072 MB genome files, respectively. The average search time for a single client scanning the 250 MB ‘genome’ was 1h:4m:47s:325ms, and this was reduced to an average time of 0h:2m:42s:42ms with 35 clients. This represents a percentage reduction from the initial wall-clock time of approximately 95.8 %. Likewise, the results attained using the 1072 MB ‘genome’ represented a reduction in wall-clock time of approximately 95.7 % computed from an initial search time of 4h:48m:6s:606ms with a single client to 0h:12m:27s:503ms with 50 clients.

These dramatic gains in wall-clock time attained through the division of a large problem into a number of smaller pieces, highlight the role that parallel computing can play in the field of bioinformatics.

The use of parallel computing has traditionally relied on programming languages such as C and Fortran. However, the advent of Java has resulted in an increased interest in the role that Java can play in scientific computing. Java has a number of key elements making it an attractive language for scientific computing, with the most important being its portability. Portability is especially important for high-performance applications; this is in part due to the life span of the hardware architectures being typically shorter than the application software. The platform independence of Java has resulted in it being referred to as the "write once, run anywhere" programming language. Java is also considered to be a better software engineering environment than both C and Fortran. This results from features such as the absence of pointers, automatic garbage collection and strict type checking which allows for rapid prototyping and leads to less buggy code and faster development time (Bull *et al*, 2001).

It has also been proposed by Bull *et al* that the nature of scientific applications lends their solution to Java execution environments, since they typically spend a large amount of execution time in a small number of user-written methods. This makes them ideal candidates for just-in-time compilation and also less susceptible than other applications to poor implementations of the Java API. However, one of the major perceived shortcomings of Java in scientific computing by programmers is its performance. The research undertaken by Bull *et al* found that on Intel Pentium hardware, and especially with a Linux operating system, the performance gap is small enough to be of little or no concern to programmers.

The fact that Java is rapidly becoming the language of choice for many mainstream and commercial applications, as well as it being a very popular teaching language in many institutions, has resulted in the major vendors expending significant resources on developing robust and efficient Java execution environments. This has resulted in one of the most apparent advantages of Java, that is the access to new resources, which includes a wide selection of class libraries and a growing number of trained programmers.

Two such libraries are the java.regex package and TSpaces, which were developed by Sun Microsystems and by researchers at IBM respectively. The java.regex package, which is included in all Java versions post JDK 1.4, provides Java programmers with a simple and very clean interface to utilise the text manipulation features of regular expressions

(Marchal, 2004). TSpaces was developed in order to solve the problems associated with connecting together disparate systems. The TSpaces software package is a messaging middleware component that combines asynchronous messaging with database features. Having been written and implemented in Java, it has the ability to run on virtually any platform from very small devices, such as a palm device, to mainframes. Since TSpaces is a direct descendant of Linda, it utilises the Tuplespace system which operates more as a global communication buffer than a data repository. These systems are tailor-made for distributed programming where a general data delivery mechanism is needed (Lehman, *et al*, 2001 and Wyckoff, 1998).

The TSpaces package provides a communication link that allows application builders the advantage of ignoring some of the harder aspects of multi-client synchronisation, such as tracking names and addresses of all active clients, communication line status and conversation status (Lehman *et al*, 2001). The tuplespace model provides a simple, yet powerful mechanism for interprocess communication and synchronisation. A process with data to share 'generates' a tuple and places it into the space. A process requiring data simply requests a tuple from the space. Although message-passing systems appear to be more efficient, tuplespace programs are typically easier to write and maintain (Wyckoff, 1998).

There are a few key factors that make TSpaces a suitable parallel framework, such as the ease of installation and implementation of the additional classes. The TSpaces server need only be installed on a machine visible on the local network, which all clients connect to in order to place or receive information in the form of tuples. Each client may interact with an arbitrary number of clients by interacting with them through a single space. However, a client is not restricted to a single space or even to a single server. Each client has the freedom to attach to servers and interact with spaces at will, since there is no 'message channel' set up required and there is no penalty for detaching from a server and reattaching later.

The clients access tuples via a standard set of simple method calls that are located in a set of TSpace library files. TSpaces is also easy to install and use both for development and deployment.

6.2 Conclusions and Future Work

The establishment of a computational cluster utilising an existing network of workstations was achieved, and the performance of the cluster in performing key bioinformatics tasks was investigated.

The use of the Java programming language in conjunction with a third party library of classes allowed for the successful design, development and implementation of a genome-scanning program to be executed in a parallel computing environment. The results attained for the various genomes in terms of the speedup associated with parallelism, and as a direct consequence of this speedup the significant reductions in wall-clock time, suggest that parallel computing has an important role to play in bioinformatics. The potential for Java to become a scientific computing language of choice has been demonstrated with a particular emphasis on performing string-matching searches. Networks of cheap, commodity workstations have also been highlighted as possessing sufficient combined computational power to tackle some of biology's major challenges. They have also been shown to be efficient and cost-effective alternatives to the traditional supercomputer, which is a financial luxury few can afford.

In order to complete this research a number of alternatives would need to be investigated. One of the most important future options may be to recompile PMS for execution in a parallel environment using a different parallel framework in order to assess the efficiency and applicability of the TSpaces parallel framework. PMS may also be linked to various databases that house a variety of domain or motif profiles and thus allow the user the freedom to select which patterns they wish to scan for. Another interesting investigation may be to recompile PMS to utilise either Hidden Markov Models (HMMs) or Positive Specific Scoring Matrices (PSSMs) as opposed to the regular expressions used in the current version of PMS. Both options would result in an increase in computational intensity whilst producing more accurate domain or motif recognition.

References

- Augen, J. (2003). ***In silico* biology and clustered supercomputing: shaping the future of the IT industry.** *Biosilico.*, **1**, 47-49.

- Bader, D. A. (2004). **Computational Biology and High-Performance Computing.** *Communications of the ACM.*, **47**, 35-40.

- Benson, D. A., Karsch-Mizrachi, I., Lipman, D. J., Ostell, J. and Wheeler, D. L. (2004). **GenBank: update.** *Nucleic Acid Research.*, **32**, D23-D26.

- Bernal, A., Ear, U. and Kyrpides, N. (2001). **Genomes OnLine Database (GOLD): a monitor of genome projects world-wide.** *Nucleic Acids Research.*, **29**, 126-127.

- Bikandi, J., Millan, R. S., Rementeria, A. and Garaizar, J. (2004). **In silico analysis of complete bacterial genomes: PCR, AFLP-PCR and endonuclease restriction.** *Bioinformatics.*, **20**, 798-799.

- Boeckmann, B., Bairoch, A.., Apweiler, R., Blatter, M. C., Estreicher, A., Gasteiger, E., Martin, M. J., Michoud, K., O'Donovan, C., Phan, I., Pilbout, S. and Schneider, M. (2003). **The SWISS-PROT protein knowledgebase and its supplement TrEMBL in 2003.** *Nucleic Acids Research.*, **31**, 365-370.

- Birney, E., Andrews, D. T., Bevan, P., Caccamo, M., Chen, Y., Clarke, L., Coates, G., Cuff, J., Curwen, V., Cutts, T., Down, T., Eyras, E., Fernandez-Suarez, X. M., Gane, P., Gibbins, B., Gilbert, J., Hammond, M., Hotz, H. R., Iyer, V., Jekosch, K., Kahari, A., Kasprzyk, A., Keefe, D., Keenan, S., Lehvaslaiho, H., McVicker, G., Melsopp, C., Meidl, P., Mongin, E., Pettett, R., Potter, S., Proctor, G., Rae, M., Searle, S., Slater, G., Smedley, D., Smith, J., Spooner, W., Stabenau, A., Stalker, J., Storey, R., Ureta-Vidal, A., Woodwark, C. K., Cameron, G., Durbin, R., Cox, A., Hubbard, T. and Clamp, M. (2004). **An Overview of Ensembl.** *Genome Research.,* **14**, 925-928.

- Bull, J. M., Smith, L. A., Pottage, L. and Freeman, R. (2001). **Benchmarking Java against C and Fortran for Scientific Applications.** *Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande.*

- Cornelius, B. (2001). **Understanding JAVA.** *Pearson Education Ltd*, Essex, England.

- Cornell Theory Center (2000) **Virtual Workshop Module: Parallel Processing Concepts.** Cornell University. http://tc.cornell.edu/services/edu/topics/ParProgCons/more.asp

- Culler, D. E., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C. and Wong, F. (1997) **Parallel Computing on the Berkeley NOW.** *JSPP '97 9th Joint Symposium on Parallel Computing.,* Kobe, Japan.

- Dente, E., Kopecky, J., Moyano, F. J. M., Roman, D. and Toma, I. (11/29/2004). **D21v0.1 Web Service Modeling Execution Environment and Triple Space Computing.** http://www.wsmo.org/2004/d21/v0.1/

- Dietz, H.. (1999). **Parallel Processing using Linux.** http://yara.ecn.purdue.edu/~pplinux/

- Drews, J. (2000) **Drug Discovery: A Historical Perspective.** *Science.,* **287.**, 1960-1964.

- Ferrari, A. J. (1998). **JPVM: Network Parallel Computing in Java.** *Proc. ACM 1998 Workshop on Java for High-Performance Network Computing.* http://www.cs.virginia.edu/jpvm/doc/jpvm-java98.pdf

- **FOLDOC (Free On-Line Dictionary Of Computing)** (02/06/2004). http://wombat.doc.ic.ac.uk/folder/foldoc.cgi?parallel+processing

- Gao, F. and Zhang, C. T. (2004). **Comparison of various algorithms for recognising short coding sequences of human genes.**, *Bioinformatics.*, **20.**, 673-681.

- Gelernter, D. (1988). **Getting the Job Done.** *BYTE.*, **13**(12), 301-308.

- Gibas, C. and Jambeck, P. (2001) **Developing Bioinformatics Computer Skills: Chapter 1 Biology in the Computer Age.** *O' Reilly & Associates Inc.*, California, USA.

- Hawick, K. A., James, H. A. and Pritchard, L. H. (2004). **Tuple-Space Based Middleware for Distributed Computing.** *Technical Report DHPC-128.* http://www.dhpc.adelaide.edu.au/reports/128/abs-128.html

- Hulo, N., Sigrist, C. J. A., Le Saux, V., Langendijk-Genevaux, P. S., Bordoli, L., Gattiker, A., De Castro, E., Bucher, P and Bairoch, A. (2004). **Recent improvements to the PROSITE database.** *Nucleic Acids Research.*, **32, Database issue.**, D134-D137.

- IBM. (09/24/2004). **IBM TSpaces User's Guide.** http://www.almaden.ibm.com/cs/TSpaces/html/UserGuide.html

- Janaki, C. and Joshi, R.R. (2003) **Accelerating comparative genomics using parallel computing.** *In Silico Biology.*, **3**, 429-440.

- Kleinjung, J., Douglas, N. and Heringa, J. (2002). **Parallelized multiple alignment.** *Bioinformatics.*, **18**, 1270-1271.

- Krishnan, A. and Tang, F. (2004). **Exhaustive Whole-Genome Tandem Repeats Search.** *Bioinformatics.*, Advanced Access published May 14, 2004.

- Lehman, T. J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B. and Bowman, P. (2001). **Hitting the distributed computing sweet spot with TSpaces.** *Computer Networks.,* **35**, 457-472.

- Li, K. B. (2003). **ClustalW-MPI: ClustalW analysis using distributed and parallel computing.** *Bioinformatics.*, **19**, 1585-1586.

- Marchal, B. (12/16/2004). **Regular Expressions in Java.** http://www.developer.com/java/other/article.php/1460561

- Meloan, S. (11/23/2004). **BioJava – Java Technology Powers Toolkit for Deciphering Genomic Codes.** http://java.sun.com/developer/technicalArticles/javaopensource/biojava/

- Merkey, P. (06/02/2004). **Beowulf Introduction & Overview.** http://www.beowulf.org/intro.html

- Merkey, P. (12/13/2004). **Beowulf History.** http://www.beowulf.org/overview/history.html

- Meskauskus, A., Lehmann-Horn, F. and Jurkat-Rott, K. (2004). **Sight: automating genomic data-mining without programming skills.** *Bioinformatics.*, Advanced Access published February 26, 2004.

- IEEE (09/08/2004), **Sun's JavaSpaces and IBM's TSpaces**, *IEEE Internet Computing Online*, http://www.computer.org/internet/v2n5/w5tech.htm.

- Pearson, W. R. and Lipman, D. J. (1988). **Improved tools for biological sequence comparison.** *Proc. Natl. Acad. Sci. USA.,* **85,** 2444-2448.

- Pedroso, H., Silva, L. M. and Silva, J. G. (1998). **JET: Massively Parallel Computing with Java** *University of Coimbra, Portugal*, Department of Engenharia Informatica. http://www.mpcs.org/MPCS98/Final_Papers/Paper.38.pdf

- Pekurovsky, D., Shindyalov, I. N. and Bourne, P. E. (2004) **A Case Study of High-Throughput Biological Data Processing on Parallel Platforms.** *Bioinformatics.*, Advanced Access published March 25, 2004.

- Reiss, T. (2001) **Drug discovery of the future: the implications of the human genome project.** *Trends Biotech.* **19**, 496-499.

- Russell, J. P. (2001). **JAVA Programming for the absolute beginner.** *PrimaTech*, California.

- Smith, T. F. and Waterman, M. S. (1981). **Identification of Common Molecular Subsequences.** *J. Mol. Biol.* **147**, 195-197.

- Sterling, T. **How to build a hyper computer.** *Scientific American* **July 2001,** 38-45.

- Stewart, C. A. (2004). **Bioinformatics: Transforming Biomedical Research and Medical Care.** *Communications of the ACM.,* **47**(11), 31-33.

- The UK JavaGrande forum (1998). **Summary.** http://dsg.port.ac.uk/~mab/HPJava/

- Thiruvathukal, G. K., Dickens, P. M. and Bhatti, S. (2000). **Java on networks of workstations (JavaNOW): a parallel computing framework inspired by Linda and the Message Passing Interface(MPI).** *Concurrency: Pract. Exper.*, **12**, 1093-1116.

- Womble, D.E., Dosanjh, S. S., Hendrickson, B., Heroux, M. A., Plimpton, S. J., Tomkins, J. L. and Greenberg, D. S. (1999). **Massively parallel computing: A Sandia perspective.** *Parallel Computing.*, **25**, 1853-1876.

- Wyckoff, P. (1998). **T Spaces.** *IBM Systems Journal.,* **37**(3).

- Zubrzycki, I. Z. (2002). **Homology Modeling and Molecular Dynamics Study of NAD-Dependent Glycerol-3-Phosphate Dehydrogenase from *Trypanosoma brucei rhodesiense*, a Potential Target Enzyme for Anti-Sleeping Sickness Drug Development.** *Biophysical Journal.*, **82**, 2906-2915.

Appendices

Appendix A: Average search times for all genome files scanned.

Table AI: Average search times for all numbers of clients tested, using the 60 MB genome file.

Number of Clients (Processors)	Average search time
0	0h:0m:0s:0ms
1	0h:16m:29s:303ms
5	0h:3m:21s:728ms
10	0h:1m:51s:88ms
15	0h:1m:23s:895ms
20	0h:1m:3s:280ms
22	0h:0m:55s:728ms
24	0h:0m:48s:408ms
25	0h:0m:46s:481ms
26	0h:0m:45s:179ms
28	0h:0m:42s:320ms
30	0h:0m:41s:790ms
35	0h:0m:41s:397ms

Table AII: Average scan times for all clients tested with the 140 MB genome.

Number of Clients (Processors)	Average Genome Scan Time
0	0h:0m:0s:0ms
1	0h:33m:53s:990ms
5	0h:7m:35s:166ms
10	0h:3m:56s:767ms
15	0h:2m:52s:408ms
20	0h:2m:5s:878ms
22	0h:1m:59s:512ms
24	0h:1m:47s:403ms
25	0h:1m:51s:492ms
26	0h:1m:48s:807ms
28	0h:1m:47s:612ms
30	0h:1m:48s:287ms

Table AIII: Average genome scan times using the 250 MB genome file with a range of clients.

Number of Clients (Processors)	Average Genome Scan Time
0	0h:0m:0s:0ms
1	1h:4m:47s:325ms
5	0h:13m:43s:269ms
10	0h:7m:7s:775ms
15	0h:4m:57s:429ms
20	0h:3m:52s:412ms
22	0h:3m:34s:235ms
24	0h:3m:26s:334ms
25	0h:3m:0m:423ms
26	0h:2m:58s:62ms
28	0h:2m:50m:943ms
30	0h:2m:48s:812ms
31	0h:2m:44s:16ms
32	0h:2m:44s:559ms
35	0h:2m:42s:42ms

Table AIV: Average genome scan times for the 1072 MB genome using a range of clients.

Number of Clients (Processors)	Average Genome Scan Time
0	0h:0m:0s:0ms
1	4h:48m:6s:606ms
5	1h:7m:13s:811ms
10	0h:34s:5s:541ms
15	0h:23m:50s:925ms
20	0h:21m:46s:214ms
25	0h:19m:2s:972ms
30	0h:16m:52s:907ms
35	0h:15m:20s:859ms
40	0h:14m:3s:161ms
45	0h:13m:15s:599ms
50	0h:12m:27s:503ms
55	0h:12m:48s:264ms

Appendix B: Average raw time in milliseconds and processed time in hr, min, sec, and millisec for each genome scanned.

Table BI: Average raw (millisec) and formatted (min) times for the 60 MB file.

Number of clients	Raw Time (ms)	Formatted Time (min)
1	989303.3333	16.48838889
5	201728.6	3.362143333
10	111088.6	1.851476667
15	83895.6	1.39826
20	63280.6	1.054676667
22	55728.6	0.92881
24	48408.8	0.806813333
25	46481.2	0.774686667
26	45179.2	0.752986667
28	42320.2	0.705336667
30	41790.0	0.6965
35	41397.8	0.689963333

Table BII: Average raw (millisec) and formatted (min) times for the 140 MB file.

Number of clients	Raw Time (ms)	Formatted Time (min)
1	2033990.333	33.89983889
5	455166	7.5861
10	236767.6667	3.946127778
15	172408.3333	2.873472222
20	125878.6667	2.097977778
22	119512	1.991866667
24	107403.4	1.790056667
25	111492	1.8582
26	108807.4	1.813456667
28	107612.4	1.79354
30	108287.4	1.80479

Table BIII: Average raw (millisec) and formatted (min) times for the 250 MB file.

Number of clients	Raw Time (ms)	Formatted Time (min)
1	3887325.667	64.78876111
5	823269.6667	13.72116111
10	427775.3333	7.129588889
15	297429.3333	4.957155556
20	231412	3.856866667
22	214235.4	3.57059
24	206334.2	3.438903333
25	180423.8	3.007063333
26	178062.4	2.967706667
28	170943	2.84905
30	168812	2.813533333
31	164016.8	2.733613333
32	164559.8	2.742663333
35	162042.4	2.700706667

Table BIV: Average raw (millisec) and formatted (min) times for the 1072 MB file.

Number of clients	Raw Time (ms)	Formatted Time (min)
1	17286606.67	288.1101111
5	4033811.2	67.23018667
10	2045541	34.09235
15	1430925.4	23.84875667
20	1306214.4	21.77024
25	1142972.4	19.04954
30	1012907.4	16.88179
35	920859.2	15.34765333
40	843161.8	14.05269667
45	795599.2	13.25998667
50	747503.8	12.45839667
55	768264.6	12.80441