St. Cloud State University
theRepository at St. Cloud State

Culminating Projects in Computer Science and Information Technology

Department of Computer Science and Information Technology

12-2018

Implementation of Parallel Search Algorithm in Computational Biology

Rahim Abdul Mohomed
rahimscsu@gmail.com

Implementation of Parallel Search Algorithm in Computational Biology

by

Rahim A. Mohomed

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

December, 2018

Starred Paper Committee:
Jie Hu Meichsner, Chairperson
Mehdi Mekni
Denis Guster

Abstract

Bioinformatics and Computational Biology are rapidly growing multidisciplinary fields, which includes wide variety of domains from DNA sequencing to sequence alignments. Recent advances in both these disciplines have allowed biologists all around the world to quickly gather a huge amount of DNA sequence data for analysis. DNA sequence alignments are becoming ever more popular due their impact in early disease diagnosis, in drug engineering, as well as in criminal investigations. With the vast growth and popularity of biological data, searching for a DNA sequence of interest in huge databases is not an easy task to produce results within a realistic time, hence there is a need to enhance the efficiency.

The reason why such information is so popular is because biologists can identify genetic information by finding sequences of similar genes or proteins with known behavior or structure without requiring long and expensive laboratory experiments. One of the most widely used tools for performing searches is Basic Local Alignment Search Tool (BLAST), a program for performing pairwise sequence alignments. As the BLAST program becomes ever more popular with biologists around the world, it faces numerous challenges. One of the main challenges is the issue of performance. The BLAST program has been looked at by researchers on how to improve the speed of search by reducing overhead costs. One of the ways to reduce the overhead cost is to incorporate parallelism to improve the performance of the BLAST algorithm.

For this paper, I explored existing variations of parallel implementations of the BLAST algorithm and compared its performance improvements with that of serial implementation of BLAST. The speed-up efficiency noted by the parallel program is far greater compared to the serial program. The paper sheds light on the impact of parallelization of the BLAST algorithm and the advantages it has on the overall field of computational biology.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1: Introduction

Overview of the Field of Biology

The field of biology has been revolutionized with technological advancements since the early 1980s. In the field of bioinformatics and biostatistics, current research and computation are limited by the available number of computer hardware. This problem can be solved by using high-performance computing resources. There are several reasons for using high-performance computing machines: access to huge amounts of data, increased computational requirements due to the use of sophisticated and complex methodologies, and latest developments in computer hardware resources.

Modern biology is facing unprecedented challenges mainly in terms of data management, search and sorting of huge amounts of data. The amount of data stored and retrieved by biologists all over the world has grown exponentially over the years. For example, a human DNA is comprised of three billion base pairs with a personal genome representing approximately 100 GB of data. It is forecasted that data will be an ever-increasing problem in the field of computational biology with the recent developments that rely heavily on computational power and storage needs.

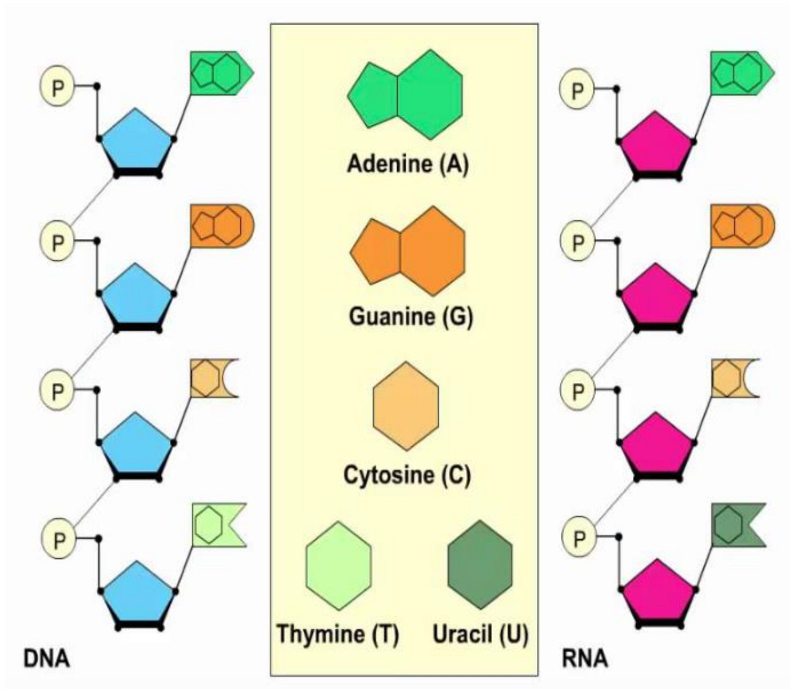
Over the years, there has also been the need for managing huge amounts of data, and the advent of such large datasets has significant storage and computational implications. The rise of cloud computing has been advantageous in the field of biology as researchers need not have to spend huge amounts of money on buying infrastructure to store the data. Instead, they can hire infrastructure on a “pay as you go” basis thereby avoiding large capital infrastructure and maintenance costs.

However, problems arise with different solutions of storing data. Big data presents problems in that it deviates from traditional structural data which are organized and accessed by rows and columns. Instead we are seeing data stored as semi-structured data, such as XML or unstructured data including flat files which are not compatible with traditional database methods. One of the major problems that arise with huge amounts of data is searching through these vast amounts of information to find an article/information of interest.

A major tool that is facing a data management problem is the Basic Local Alignment Search Tool (BLAST). This tool helps biologists all over the world to search DNA sequences of interests from a large database pool. The search times could exceed more than an hour in some instances. Over the years many researchers have proposed new implementations for the BLAST algorithm to help to speed up the search time. One of the areas of interests is to use parallel computing techniques to speed up the search time.

Cells--The Basic Precursor to Living Organisms

Cells is the precursor for every living organism. These cells are essentially made up of proteins and nucleic acids. Nucleic acids are either Deoxyribonucleic Acids (DNA) or Ribonucleic Acids (RNA). DNA is a molecule that has the necessary instructions required for the cell to perform its biological functions. DNA contains genetic information and is responsible to propagate the characteristics of one organism from one generation to another. The DNA molecule is made up of four nucleic acids: adenine (A), guanine (G), cytosine (C), and thymine (T). The DNA molecule has the shape of a double helix and the nucleotides are connected to each other forming strands with two terminals: 5’ and a 3.’ The full DNA sequence of an organism is known as a genome. Figure 1 shows a pictorial representation of a DNA and an RNA molecule.

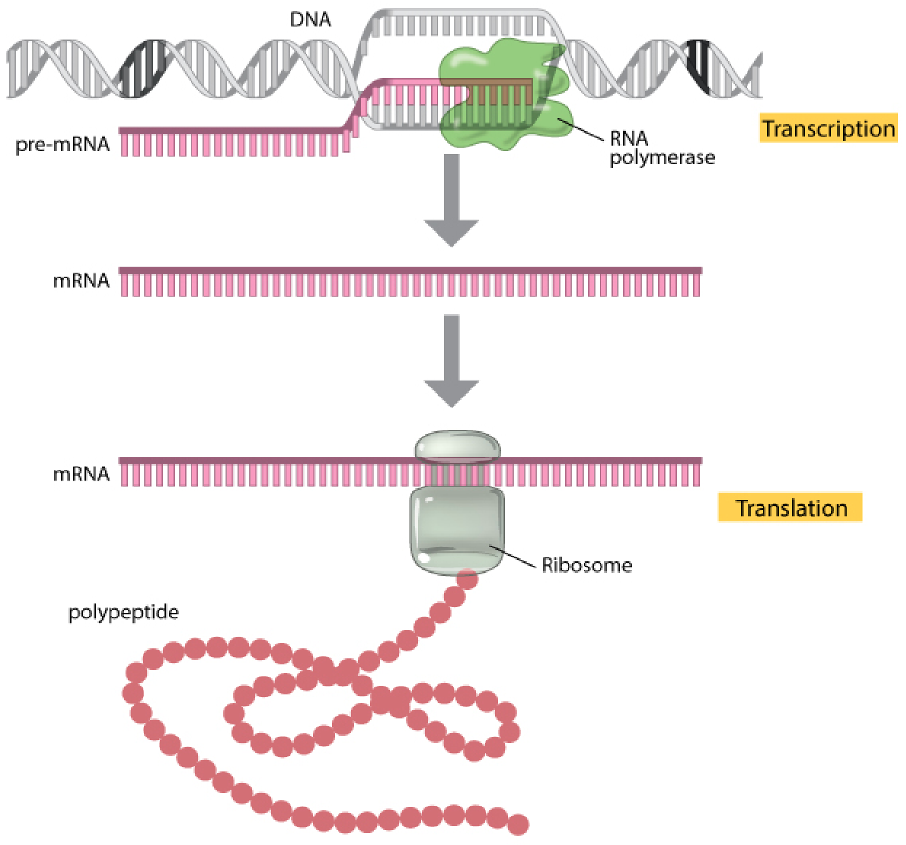


(Isa, 2013)

Figure 1

The Structure of a DNA and RNA Molecule

Genes are made of DNAs and each gene provides instructions for cells in living organisms to make other molecules known as proteins. Proteins arise from DNA from two separate processes: the transcription and the translation process. During the transcription phase, information from the DNA is transferred to a molecule called messenger ribonucleic acid (mRNA). After this step, the mRNA is translated into a protein molecule during the translation process (Isa, 2013). Figure 2 shows the pictorial representation of the transcription and translation phases. During this translation phase, the information from the mRNA is read into 20 main amino acids as shown in Table 1**.**



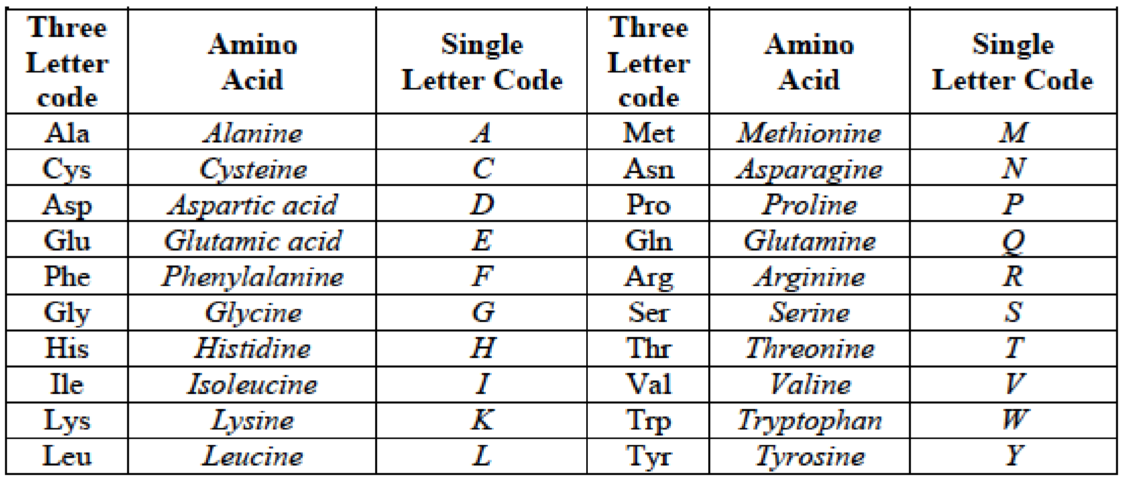
(Isa, 2013)

Figure 2

The Transcription and Translation Process

Table 1

List of 20 Amino Acids



Three Letter code	Amino Acid	Single Letter Code	Three Letter code	Amino Acid	Single Letter Code
Ala	*Alanine*	*A*	Met	*Methionine*	*M*
Cys	*Cysteine*	*C*	Asn	*Asparagine*	*N*
Asp	*Aspartic acid*	*D*	Pro	*Proline*	*P*
Glu	*Glutamic acid*	*E*	Gln	*Glutamine*	*Q*
Phe	*Phenylalanine*	*F*	Arg	*Arginine*	*R*
Gly	*Glycine*	*G*	Ser	*Serine*	*S*
His	*Histidine*	*H*	Thr	*Threonine*	*T*
Ile	*Isoleucine*	*I*	Val	*Valine*	*V*
Lys	*Lysine*	*K*	Trp	*Tryptophan*	*W*
Leu	*Leucine*	*L*	Tyr	*Tyrosine*	*Y*

(Isa, 2013)

Combinations of any of these 20 main amino acids produce different protein sequences, each of which has its own biological role to perform. In the field of bioinformatics, tools known as sequence alignments are used to identify and compare various DNA sequences for reasons such as mutations or general changes (Isa, 2013). For example, in a pairwise sequence alignment, a newly discovered biological sequence also known as a query sequence is compared against sequences in a certain database, while for multiple sequence alignment, a query sequence is compared against many sequences at once. The reason behind this is for biologists to discover regions of similarity between the sequences under study, which may provide useful information on their characteristics. The goal of sequence comparison is to determine the similarities between two genetic sequences.

DNA sequencing is also useful to achieve other goals such as facilitating drug engineering, the determination of protein's function, and construction of evolutionary (DNA) trees. For example, when a human protein is damaged, DNA sequenced is performed to find the most similar sequence in the database. Based on the most similar sequence in the database, researchers can model the entire human protein sequence needed to make the specialized drug that binds to the particular DNA sequence (Isa, 2013).

Goal and Objectives

As the demand for quick data searches and retrieval is gaining popularity in the field of biology, this paper aimed to find solutions to improve the speed efficiency and provide solutions and improvements to the existing BLAST algorithm.

The goal of this paper was to create a parallel implementation of the BLAST algorithm and compare the speed up efficiency with that of serial implementation. My work focused mainly on researching existing variations of BLAST and implementing the parallel algorithm with ideas driven from these variations. This parallel implementation is compared against the serial implementation of BLAST using various sizes of databases. Finally, the paper provides examples of other parallel implementations of the BLAST algorithm and provides solutions and ideas for future work.

The objective of this paper was to create a serial and parallel implementation of the BLAST algorithm. Using the serial program as "building blocks" for the implementation of the parallel BLAST program, the parallel program was developed using the BLAST algorithm in the serial program and will be parallelized. In other words, the serial implementation of the BLAST program was created first and then the parallel BLAST program was developed after that using the exact same BLAST algorithm with the only difference being the parallelization of the program. This was done for the sole purpose of initially focusing on the BLAST algorithm and later focusing on parallelizing the serial implementation using pthreads.

The paper is organized to introduce the reader to the basics of genetics and bioinformatics in Chapter 1. Chapter 2 of this paper describes the BLAST and its uses. Chapter 3 discusses parallel computing its influence in the field of computation biology. Chapter 4 discusses the BLAST algorithm in detail and provides examples. Chapter 5 discusses the study carried out along with comparisons, and Chapter 6 draws examples for future developments and improvements to the program.

Chapter 2: BLAST (Basic Local Alignment Search Tool)

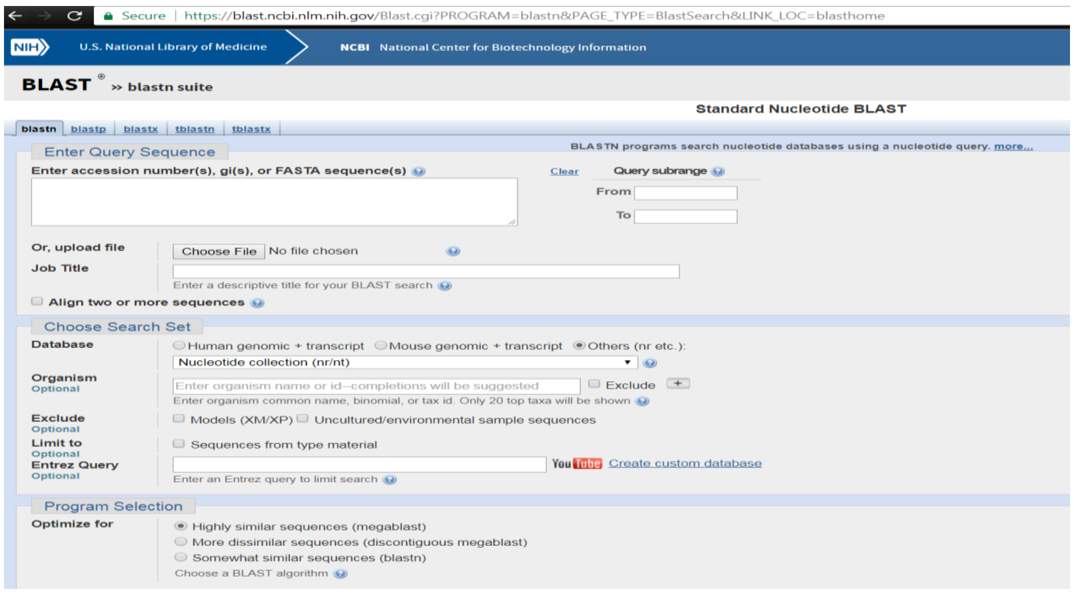
Since the discovery of genetic code, research in the field of biology has undergone a sea of change in the way it is performed. The growth of molecular biology and research in the field of genetics during the 20th century moved biological research from the test tube to more complicated genetic analysis. During this time, biologists started accumulating DNA and protein sequence data at an exponential rate. Currently there are approximately 97 billion bases sequences and over 93 million records stored over various databases all around the world (Zomaya, 2006).

As DNA sequencing grew at the end of the 20th century, scientists turned to computers to help analyze the abundant and massive amount of genetic data. Today, one of the most common tools used to examine DNA and protein sequences is the Basic Local Alignment Search Tool which is also known as BLAST. BLAST is a computer algorithm that is available for use online at the National Center of Biotechnology Information (NCBI) website (Zomaya, 2006). BLAST is used to align and compare a query DNA sequence with a database of sequences, which makes it a critical tool in ongoing genomic research. In recent years, development of BLAST has enabled scientists to study the genetic blueprint of life across many species, and it has also helped connect biology and computer science in combined field known as bioinformatics.

BLAST: A Widely Used Search Tool

One of the current search tools for biologists that is in place is BLAST: Basic Local Alignment Search Tool. This search tool allows the user to find regions of local similarity between protein sequences of DNA strands. This search algorithm compares protein sequences that a user can upload to sequence databases and calculates the statistical significance of matches. This is a useful search tool for researchers to find commonalities between sequences as well as help in identification of members of gene family. Figure 3 shows the Web Application for BLAST that is used by biologists all around the world to search DNA sequences from huge databases.

The search strategy used in BLAST is based on scoring matrices to compare short subsequences (words) in the query sequence against the entire target DNA or protein sequence database to find statistically significant matches, then extending these matches to find the most similar sequences or sub-sequences. Figure 4 shows a pictorial representation of two DNA strands and how the BLAST search algorithm uses local alignment to efficiently search DNA sequences in the database.


Secure | https://blast.ncbi.nlm.nih.gov/Blast.cgi?PROGRAM=blastn&PAGE_TYPE=BlastSearch&LINK_LOC=blasthome

NIH U.S. National Library of Medicine NCBI National Center for Biotechnology Information

BLAST® » blastn suite

Standard Nucleotide BLAST

blastn | blastp | blastx | tblastn | tblastx

BLASTN programs search nucleotide databases using a nucleotide query. more...

Enter Query Sequence

Enter accession number(s), gi(s), or FASTA sequence(s)

Clear

Query subrange

From

To

Or, upload file: Choose File No file chosen

Job Title: Enter a descriptive title for your BLAST search

☐ Align two or more sequences

Choose Search Set

Database: Human genomic + transcript Mouse genomic + transcript [✓ Others (nr etc.):]

Nucleotide collection (nr/nt)

Organism Optional: Enter organism name or id--completions will be suggested ☐ Exclude +

Enter organism common name, binomial, or tax id. Only 20 top taxa will be shown

Exclude Optional: ☐ Models (XM/XP) ☐ Uncultured/environmental sample sequences

Limit to Optional: ☐ Sequences from type material

Entrez Query Optional: YouTube Create custom database

Enter an Entrez query to limit search

Program Selection

Optimize for:

[✓ Highly similar sequences (megablast)]

More dissimilar sequences (discontiguous megablast)

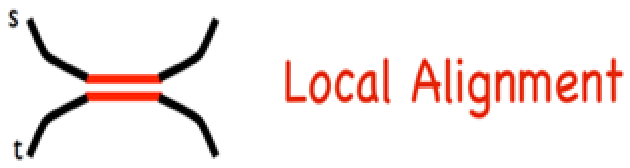
Somewhat similar sequences (blastn)

Choose a BLAST algorithm

(National Center for Biotechnology Information, 2018)

Figure 3

Web-based BLAST Application



(National Center for Biotechnology Information, 2018)

Figure 4

Web-based BLAST Application

Even though BLAST uses a very efficient algorithm, the growing size of sequence databases and number of searches needed are quickly expanding beyond the computational capabilities of individual computers. The ever-increasing need for higher performance has created a demand for a more powerful version of BLAST for use on multiprocessors and PC clusters and has led to the development of enhanced versions of BLAST, which attempts to exploit parallelism to improve performance.

Access and Use of the BLAST Application

There are many ways through which a user can submit BLAST searches, but the most commonly used way to submit searches is via the NCBI website (National Center for Biotechnology Information, 2018). In the submission page, the user simply inputs a raw sequence and clicks the BLAST button. If the user would like to modify the default search, they are given various options. The user can change the database where the DNA sequence will provide searches for or limit their search taxonomically using autocomplete menus. Once the BLAST button is clicked, the BLAST algorithm parses the input and creates an Abstract Syntax Notation (ASN.1) representation of the search, inserts the search into an MSSQL database, and sends a Request ID (RID) to the user. During this time, the BLAST algorithm is working behind the scenes to provide the user with a detailed search output, which includes similar DNA sequences and scores. The user's browser periodically polls the server, checking for complete results. Once the results are complete, the page displays the report. Results are usually saved for 36 hours on the server. The user may use the RID to retrieve the results in the future.

Users may also be able to access NCBI blast through the BLAST+ remote service, which is a network service that uses ASN.1 to communicate between the client and the server. In this case, the client sends the query, parameters, and database to the server in the form of an ASN.1 request. An RID is assigned and sent back to the client. The client polls for the status of the result on a regular basis. Once the search is done, the ASN.1 results are returned to the client.

Another way that users can search is through NCBI "URL API" interface and the HTTP protocol to create BLAST jobs and retrieve BLAST results.

There are many use cases for BLAST, but the most common uses are as follows:

1. Data Collection: For biologists that have to gather and analyze large amounts of DNA sequence data, the biologists can utilize BLAST to query the sequences. For example, a biologist may collect a large number of DNA sequences as expressed sequence tags (ESTs) in a single tissue sample, then compare all EST's against a known DNA sequence database to estimate their biological properties.
2. Self-Comparisons: Biologists may also wish to compare data with itself to discover similarities and differences. After collecting a sample amount of DNA sequences, they may compare each DNA sequence collected against each other to discover highly expressed sequences of fragments that may be combined into longer sequences.

3. Database Comparisons: Biologists may also wish to perform comparisons using existing DNA sequence databases, either comparing databases against one another or against itself. For example, a biologist may compare all human gene sequences against a monkey gene sequences for comparative genomics or compare the genomes of different species (Zomaya, 2006).

The BLAST Family

Many variations of the BLAST have been developed and this paper explored a couple of these variations. In the beginning, the BLAST algorithm was split into two adaptations: the NCBI BLAST and Washington University BLAST (WU Blast). Both of these BLAST algorithms have program variations within themselves. A couple years later, more variations of BLAST were created. Some of them are discussed here:

1. BLASTN can be used to compare two nucleotide sequences;
2. BLASTP can be used to compare two protein sequences;
3. BLASTX can take a nucleotide sequence, translate it, and query it versus a protein database in one step;
4. TBLASTN compares a protein query sequence to all possible databases, to search for a new protein, undescribed genomes;
5. WImpiBLAST which is a web interface for mpiBLAST to help biologists perform large scale annotation using high performance computing. WImpiBLAST provides an easier to use web interface to biologists to perform searches using parallel computing.

Genomic Database

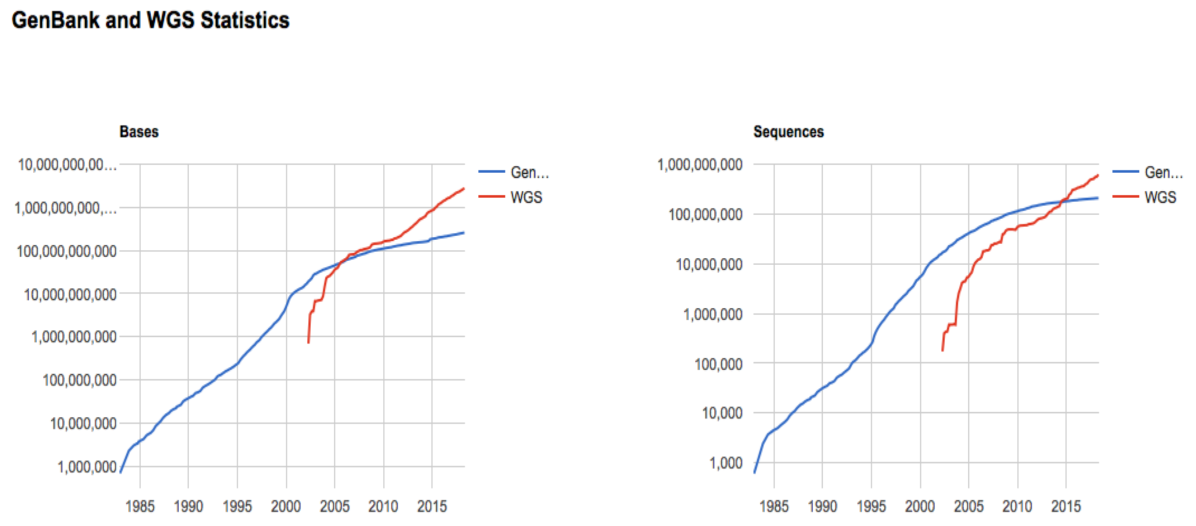
There are millions of sequences stored in databases all around the world. After the completion of the Human Genome Project in 2003, biological information such as DNA sequences is stored securely in databases. Examples of such databases include: GenBank which was created and managed in the US, DNA Databank of Japan, and European Bioinformatics Institute. The most widely used are GenBank from the NCBO (National Center for Biotechnology Information), SwissPort from the Swiss Institute of Bioinformatics, and PIR from the Protein Information Resource.

GenBank is a database that contains publicly available nucleotide (DNA) sequences for more than 200,000 organisms, which are obtained primarily through submissions from individual labs and batch submissions from large scale sequencing projects. Daily data exchanges occur with the European and the DNA Data Bank of Japan to ensure worldwide coverage.

BLAST provides sequences similarity searches of GenBank and other sequences in databases. GenBank data is available at no cost over the internet, via FTP and also via a wide range of web-based applications such as BLAST which operate mainly on GenBank data.

From its inception, GenBank has doubled in size about every 18 months. It now contains over 65 billion nucleotide bases from more than 61 million individual sequences, with about 15 million sequences added in the past year (Zomaya, 2006). Each GenBank entry includes a concise description of the sequence, the scientific name, the taxonomy of the source organism, bibliographic references, and a table of features listing areas of biological significance, such as regions of interest in the DNA, their mutations and modifications.

Figure 5 lists the number of bases and the number of sequence records in each release of GenBank, beginning with Release 3 in 1982. From 1982 to the present, the number of bases in GenBank has doubled approximately every 18 months. This has resulted in a massive explosion in sequence DNA information that is widely available for biologists around the world. The vast amount of DNA information and its popularity among biologists definitely proves that there is a huge need for a repository to store and retrieve DNA sequences.



(Zomaya, 2006)

Figure 5

GenBank Statistics

Sequence similarity searches are the most fundamental and most common type of analysis performed on GenBank Data. NCBI offers the BLAST family of programs to detect similarities between a query sequence and database sequences. BLAST searches are either performed on the NCBI's website or via a set of standalone programs distributed by FTP.

BLAST was designed to search nucleotides and proteins of interest from a vast database. BLAST takes your query (DNA sequence or protein sequence) and searches its protein or DNA databases for levels of identity that ranges from perfect matches to matches of very low similarity. After it searches through the database, it reports back to you what it finds, in order of decreasing significance (in decreasing similarity). BLAST results may take anywhere from a couple of minutes to several hours during peak times. Peak times is defined as times where there is a heavy load on the servers as many users around the world try to access the same resources, which in this case would be the BLAST web application. There are many versions (forms) of BLAST but for simplicity the BLAST algorithm discussed here will be for nucleotide (DNA) which is called BLASTN, the N stands for nucleotide (DNA).

BLAST Heuristic

BLAST increases the speed of the alignment by decreasing the search space or number of comparisons it makes. Specifically, instead of comparing every DNA sequence against each other, BLAST uses short “word” segments to create alignment “seeds.” Requiring only words that are three letters in length to match in order to seed an alignment, means that fewer sequence regions needs to be compared. Larger word sizes usually mean there are even fewer regions to evaluate. Once an alignment is seeded, BLAST extends the alignment and a score is given to the DNA sequence. Since its creation, BLAST has become an essential tool for biologists. Its sensitivity allows scientists to compare nucleotide and protein sequences to both single sequences and large databases.

Chapter 3: Parallel Computing

Parallel Computing is a computing method of concurrent use of multiple processors (CPUs) to do computational work. In comparison, serial programming is a method in which a single processor executes program instructions in a step by step manner. Parallel computation can be performed on shared-memory systems with multiple CPU's, distributed memory clusters made up of smaller shared memory systems or single-CPU systems. Several applications in computational biology have large run time and memory requirements which can be addressed by Parallel Computing. The run-time of applications can be reduced by the use of multiple processors to solve the problem and scaling of memory with processors enables finding solutions of larger problems (Aluru, 2003).

Parallel Computing

A parallel computer uses a set of processors that are able to work together to solve a computation problem. This is made possible through splitting the problem load into parts and by reconnecting the partial computations to create an accurate outcome. The way by which the load distribution and reconnections are managed is heavily influenced by the system that will support the execution of the parallel application program.

Parallel computations are broadly classified into two main models based on Flynn's specifications: Single-Instruction Multiple Data (SIMD) machines and Multiple Instruction Multiple Data (MIMD) machines (Aluru, 2003).

SIMD machine consists of many simple processors each with a small local memory. The complexity and often the inflexibility of SIMD machines, strongly dependent on the synchronization requirements, have restricted their use mostly to special purpose applications.

More commonly used, MIMD machines are more amenable to bioinformatics. In MIMD machines, each computational process works at its own pace in an asynchronous fashion and is completely independent of the other computation processes. Memory architecture has a strong influence on the global architecture of MIMD machines, which becomes a key issue for parallel execution, and frequently determines the optimal programming model (Julich, 1995).

Parallel software enables massive computational tasks to be divided into several separate processes that execute concurrently for the solution of a common task through the usage of different processors. There are two key features that could be used to compare models: granularity, which is the relative size of the units of computation that execute in parallel; and communication, the way the separate units of computation exchange data and synchronize their activity.

An example of granularity is by formulating a block of instructions. At this level, a programmer identifies sections of the program that can safely be executed in parallel and inserts the directives that begin to separate the tasks. When the parallel program starts, the run time support creates a pool of threads that are unblocked by the runtime library as soon as the parallel section is reached. At the end of the parallel section, all extra processes are suspended, and the original process continues to execute (Aluro, 2003).

Ideally, if we have n-processors, the run time should be also n times faster with respect to the wall clock time. In real implementations, however, the performance of a parallel program is decreased by synchronization between processes, communication and load imbalance. Coordination between processors represents sources of overhead, in the sense that they require some time added to the pure computational workload.

Most of the effort of a programmer goes into increasing the efficiency of the program. By lowering the overhead costs such as decreasing the amount of communication between processors, idle time and work distribution efficiency will help increase the efficiency of the program. The simplest way when possible is to reduce the number of task divisions; in other words, to create a coarsely grained application.

Once granularity has been decided, communications needs to be enforced for correct behavior and to create an accurate outcome. When shared memory is available, inter-process communication is performed through shared variables. When several processors are working over the same space, the locks and the critical sections (block of code that only one process can execute at a time) are required for safe access to shared variables.

When distributed memory is used, sending messages over the network must be performed all inter-process communication. With this message passing paradigm, the programmer has to know where the data is, what is to communicate, and when is to communicate with whom. Library subroutines are available to facilitate the message passing constructions: PVM and MPI (Aluru, 2003).

Researchers have attempted to use parallelism techniques to improve BLAST. Various techniques have been implemented over the years and a couple stand for their advantages. One of the most commonly used techniques is called the Vector Parallelism Technique. This is one of the earlier techniques to improve BLAST performance by using low-level vector parallelism to speed up the calculation of scores for sequence alignments. BLAST is able to find hits between statistically significant words and sequences in the database; it must find a local pairwise alignment and calculate a score for the alignment, before returning the best score found.

This local pairwise alignment can be parallelized in a fine-grain manner using vector parallelism (Zomaya, 2006).

Another common parallelism technique is known as Multithreading. This approach improves BLAST performance by using thread-level parallelism to compare queries to different parts of a sequence database. As a BLAST search is performed against sequence databases consisting of multiple sequences, searches can easily be performed in parallel using multiple threads. Some examples of Multithreading include NCBI BLAST and WU BLAST which are programs that can be run in multithreaded mode on shared-memory multiprocessor (SMP) machines. These parallel variations of BLAST and along with a couple more variations will be looked into further later in this paper (Zomaya, 2006).

Chapter 4: BLAST and its Parallelism

The BLAST application is used by biologists all over the world to search DNA and protein sequences. This has caused BLAST to grow in popularity over recent years which has come with a couple downsides. One of the most notable downside is to do with its efficiency in providing quick results to biologists. Various parallelism techniques have been researched over the years to improve the speedup of the search. The BLAST algorithm along with the parallelism techniques will be surveyed in this chapter.

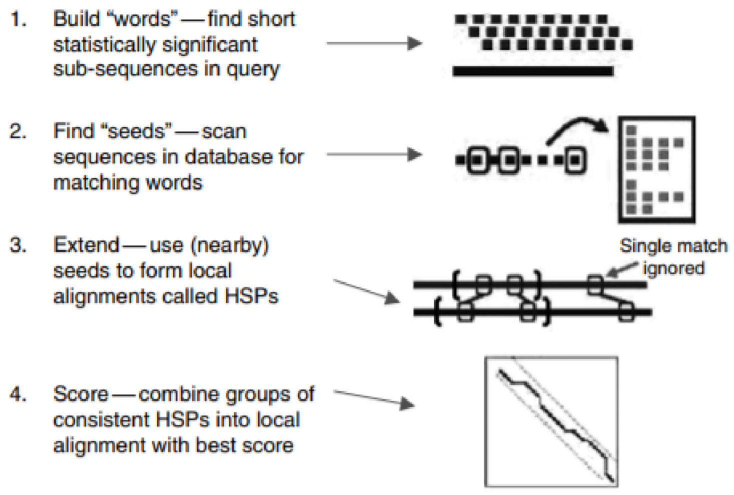
BLAST Algorithm Steps

The BLAST program is used to search DNA and protein sequences against a database of DNA and protein sequences. The BLAST algorithm has four stages: build words, find seeds, extend, and score. Figure 6 gives an overview of these stages.

To explain the algorithm, a DNA query sequence of "ACTGA" and a database sequence of "GACTGC" can be used as an example.

1. Build Words: The first step is to break the user input query into fragments ("words") and then the program compiles a word list. For DNA sequence, for example, the word list includes all the words with an input length of *W* in the query sequence. Therefore, for "ACTGA" if the input length is 3, then the word list is: ACT, CTG, TGA.
2. Find Seeds: BLAST then scans through the database to find all occurrences of the words in the list. The words are used as "seeds" for the next step. For example, the seeds generated in this step are "ACT" and "CTG."

3. Extend: In this step the matching words (Seeds) are extended into ungapped local alignments between the query sequence and the sequence from the database. Extensions are made in both left and right directions and only stop when the "score" drops below a threshold value. The resulting pairwise alignments are called high scoring pairs (HSPs). For example, the seed extension step would result in the query sequence of "ACTG."
4. Score: In this final step: the top scoring HSP's are combined. HSPs are consistent only if they can be combined without any overlapping and while maintaining the same order in both the query and sequential database sequences. Statistical tools are used to assess the significance of the results and to select the most likely alignment (Zomaya, 2006).


1. Build "words"—find short statistically significant sub-sequences in query
2. Find "seeds"—scan sequences in database for matching words
3. Extend—use (nearby) seeds to form local alignments called HSPs
Single match ignored
4. Score—combine groups of consistent HSPs into local alignment with best score

(Zomaya, 2006)

Figure 6

BLAST Algorithm

Purpose of BLAST

Biologists may collect large number of DNA sequences from a breast tissue sample of cancer patients, and then compare this information against known sequence database (BLAST) to estimate their biological function/properties. Each sequence collected (from one patient) becomes a single BLAST query. Hence, biologists frequently perform a large number (batch) of sequence comparisons at once. The data gathered from BLAST may also be used to discover similarities and differences between DNA sequences.

Factors Affecting BLAST Performance

Large numbers of BLAST searches tend to cause intensive workloads which can have an effect on the performance of the search results. Some of the main factors affecting BLAST performance include:

1. Size of the Database: The BLAST program consists of many databases for a number of organisms, and this database size can vary greatly. DNA Sequence Databases consisting of nucleotides (A, C ,T, G) are typically the largest because DNA techniques are most developed and researched. On the other hand, protein databases are smaller because protein sequences are shorter and fewer proteins have been identified and researched.
2. Size of the Searches: Query searches sizes may range from hundreds to potentially millions of sequences simultaneously.
3. Size of the Searched Sequence Length: The length of the search sequence may also vary, depending on the bioinformatics application. Sequence lengths may range from

50-100 for DNA through 300-500 for proteins, to more than 10,000 for genes (Zomaya, 2006).

BLAST Case Study

The use of BLAST to find out if a tumor is benign or malignant (cancerous or non-cancerous):

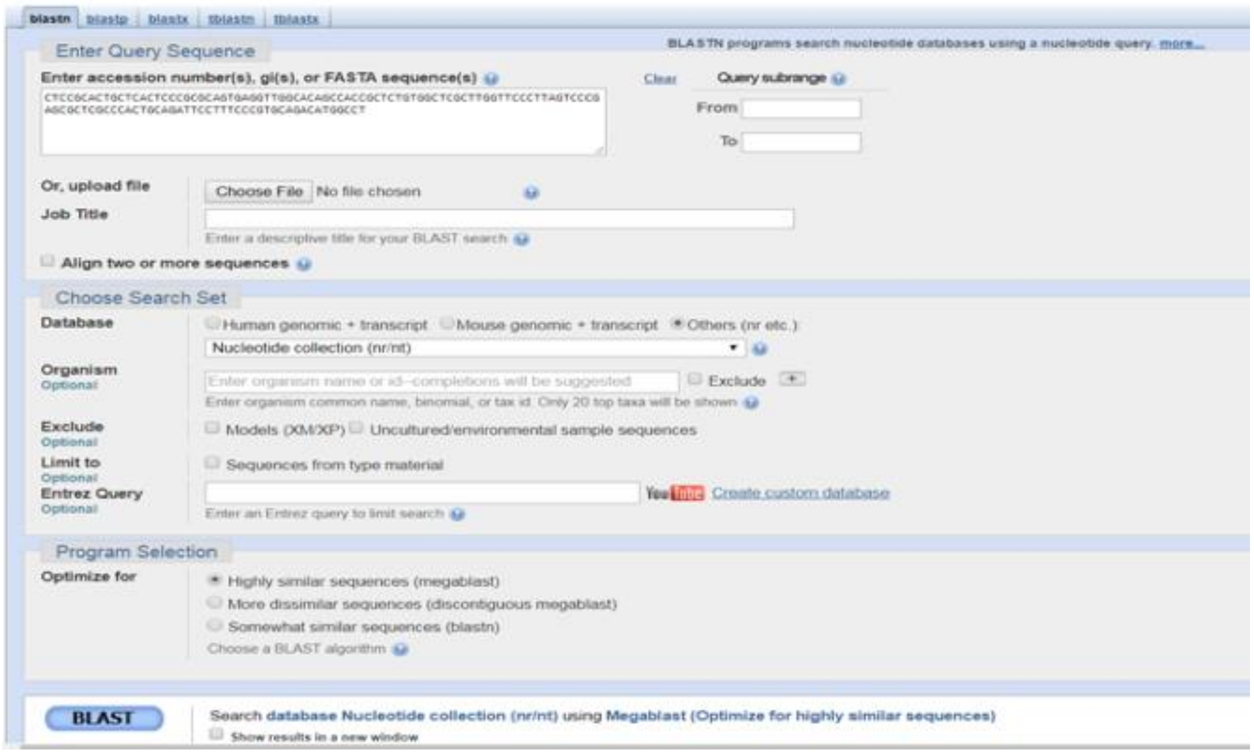
Patient Jane comes in for a regular physical to her doctor and complains about pain in her abdomen area. The doctor performs the physical and does two things during the physical: he takes Jane's blood for further analysis and sets an appointment for her colonoscopy. The stats for Patient Jane are as follows:

- Age = 35, Healthy
- Symptoms: pain in abdomen, blood in stool and tiredness
- Colonoscopy performed**:** Suspicious Polyps (Tumor) found
- Biopsy: Colon Cancer--Confirmed

The doctor then performs a DNA sequencing on the tumor (cancer) cell. The DNA Sequence of the tumor is as follows:

CTCCGCACTGCTCACTCCCGCGCAGTGAGGTTGGCACAGCCACCGCTCTGTGGCT
CGCTTGGTTCCCTTAGTCCCGAGCGCTCGCCCACTGCAGATTCCTTTCCCGTGCA
GACATGGCCT

The doctor then enters the DNA sequence into the BLAST web application as shown in the Figure 7. The search is then performed, which takes about an average of couple minutes (during peak times the time could significantly increase), and then the doctor is presented with the search results as seen in Figures 8 and 9. As seen in the Figure 9, there is a perfect match with a database DNA sequence. A perfect match here is considered to be a 100% score under the identity column.



blastn | blastp | blastx | tblastn | tblastx

Enter Query Sequence

BLASTN programs search nucleotide databases using a nucleotide query. more...

Enter accession number(s), gi(s), or FASTA sequence(s)

```
CTCCGCACTGCTCACTCCCGCGCAGTGAGGTTGGCACAGCCACCGCTCTGTGGCTCGCTTGGTTCCCTTAGTCCCG
AGCGCTCGCCCACTGCAGATTCCTTTCCCGTGCAGACATGGCCT
```

Clear

Query subrange: From / To

Or, upload file: Choose File No file chosen

Job Title: Enter a descriptive title for your BLAST search

Align two or more sequences

Choose Search Set

Database: Human genomic + transcript Mouse genomic + transcript [✓ Others (nr etc.)]

Nucleotide collection (nr/nt)

Organism (Optional): Enter organism name or id--completions will be suggested Exclude

Enter organism common name, binomial, or tax id. Only 20 top taxa will be shown

Exclude (Optional): Models (XM/XP) Uncultured/environmental sample sequences

Limit to (Optional): Sequences from type material

Entrez Query (Optional): Create custom database

Enter an Entrez query to limit search

Program Selection

Optimize for: [✓ Highly similar sequences (megablast)] More dissimilar sequences (discontiguous megablast) Somewhat similar sequences (blastn)

Choose a BLAST algorithm

BLAST

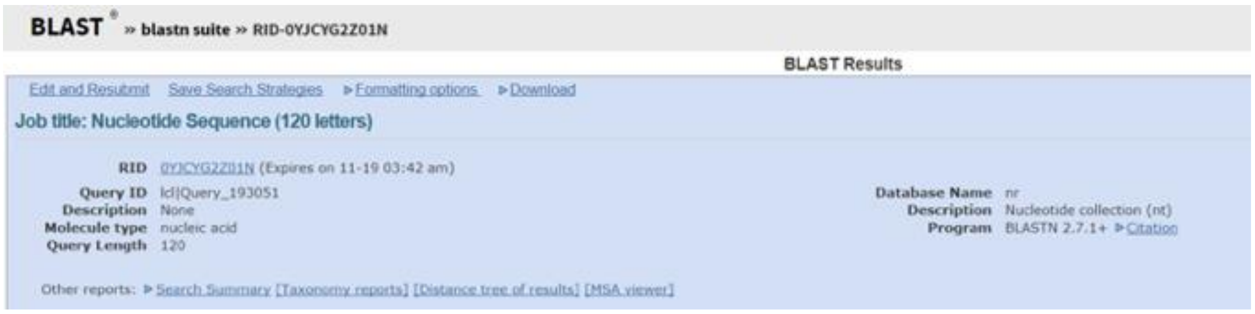
Search database Nucleotide collection (nr/nt) using Megablast (Optimize for highly similar sequences)

Show results in a new window

(National Center for Biotechnology Information, 2018)

Figure 7

BLAST Web Interface



BLAST® » blastn suite » RID-0YJCYG2Z01N

BLAST Results

Edit and Resubmit Save Search Strategies ▸Formatting options ▸Download

Job title: Nucleotide Sequence (120 letters)

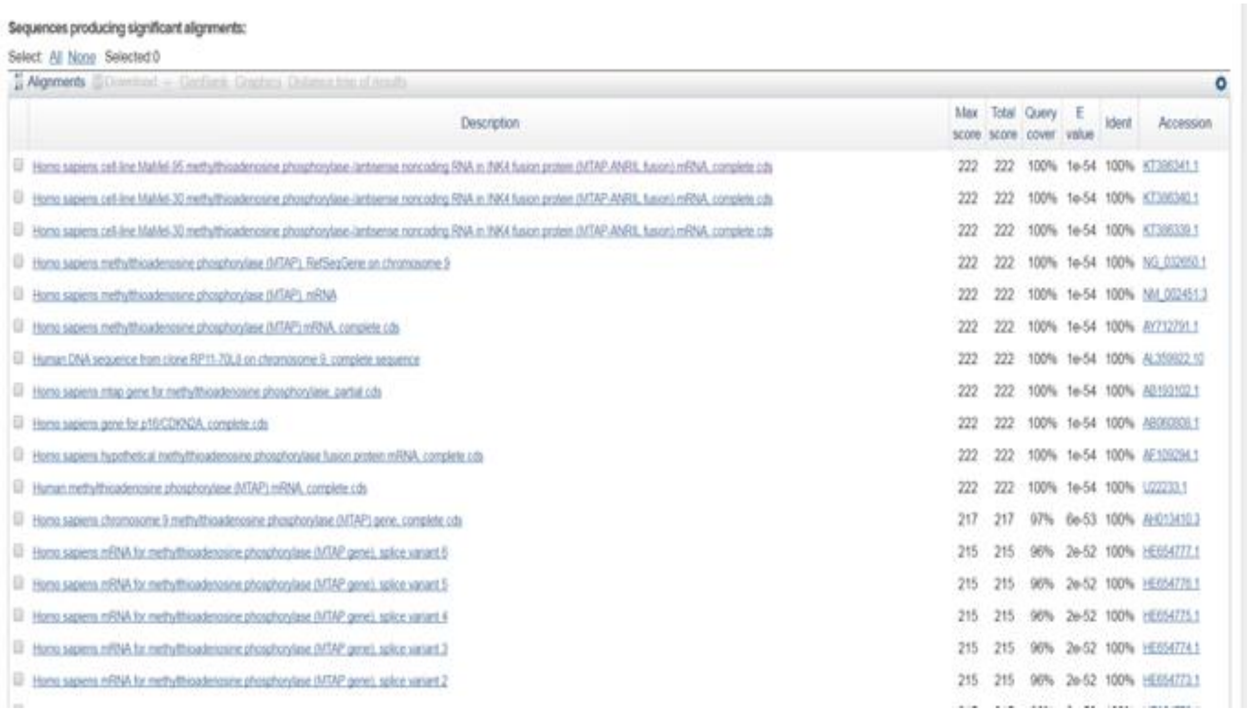
RID	0YJCYG2Z01N (Expires on 11-19 03:42 am)		
Query ID	lcl|Query_193051	Database Name	nr
Description	None	Description	Nucleotide collection (nt)
Molecule type	nucleic acid	Program	BLASTN 2.7.1+ ▸Citation
Query Length	120		

Other reports: ▸Search Summary [Taxonomy reports] [Distance tree of results] [MSA viewer]

(National Center for Biotechnology Information, 2018)

Figure 8

BLAST Results Overview



Sequences producing significant alignments:

Select: All None Selected:0

Alignments Download ⌄ GenBank Graphics Distance tree of results

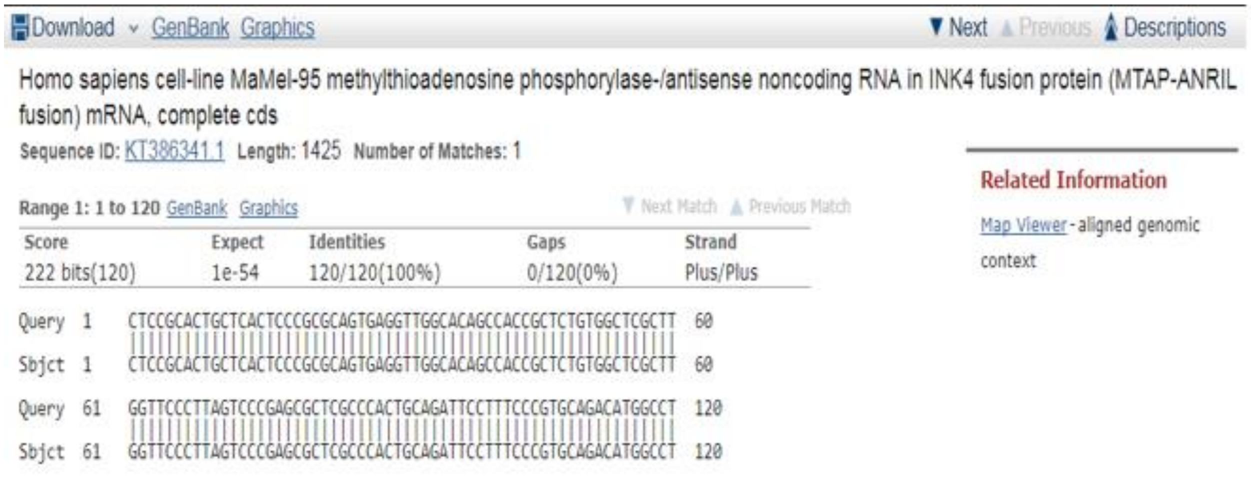
Description	Max score	Total score	Query cover	E value	Ident	Accession
Homo sapiens cell-line MaMel-95 methylthioadenosine phosphorylase-/antisense noncoding RNA in INK4 fusion protein (MTAP-ANRIL fusion) mRNA, complete cds	222	222	100%	1e-54	100%	KT386341.1
Homo sapiens cell-line MaMel-30 methylthioadenosine phosphorylase-/antisense noncoding RNA in INK4 fusion protein (MTAP-ANRIL fusion) mRNA, complete cds	222	222	100%	1e-54	100%	KT386340.1
Homo sapiens cell-line MaMel-30 methylthioadenosine phosphorylase-/antisense noncoding RNA in INK4 fusion protein (MTAP-ANRIL fusion) mRNA, complete cds	222	222	100%	1e-54	100%	KT386339.1
Homo sapiens methylthioadenosine phosphorylase (MTAP), RefSeqGene on chromosome 9	222	222	100%	1e-54	100%	NG_032050.1
Homo sapiens methylthioadenosine phosphorylase (MTAP), mRNA	222	222	100%	1e-54	100%	NM_002451.3
Homo sapiens methylthioadenosine phosphorylase (MTAP) mRNA, complete cds	222	222	100%	1e-54	100%	AY712791.1
Human DNA sequence from clone RP11-70L8 on chromosome 9, complete sequence	222	222	100%	1e-54	100%	AL359922.10
Homo sapiens mtap gene for methylthioadenosine phosphorylase, partial cds	222	222	100%	1e-54	100%	AB193102.1
Homo sapiens gene for p16/CDKN2A, complete cds	222	222	100%	1e-54	100%	AB060808.1
Homo sapiens hypothetical methylthioadenosine phosphorylase fusion protein mRNA, complete cds	222	222	100%	1e-54	100%	AF109294.1
Human methylthioadenosine phosphorylase (MTAP) mRNA, complete cds	222	222	100%	1e-54	100%	U22233.1
Homo sapiens chromosome 9 methylthioadenosine phosphorylase (MTAP) gene, complete cds	217	217	97%	6e-53	100%	AH013410.3
Homo sapiens mRNA for methylthioadenosine phosphorylase (MTAP gene), splice variant 6	215	215	96%	2e-52	100%	HE654777.1
Homo sapiens mRNA for methylthioadenosine phosphorylase (MTAP gene), splice variant 5	215	215	96%	2e-52	100%	HE654776.1
Homo sapiens mRNA for methylthioadenosine phosphorylase (MTAP gene), splice variant 4	215	215	96%	2e-52	100%	HE654775.1
Homo sapiens mRNA for methylthioadenosine phosphorylase (MTAP gene), splice variant 3	215	215	96%	2e-52	100%	HE654774.1
Homo sapiens mRNA for methylthioadenosine phosphorylase (MTAP gene), splice variant 2	215	215	96%	2e-52	100%	HE654773.1

(National Center for Biotechnology Information, 2018)

Figure 9

BLAST Results

The BLAST application shows the user query and the DNA sequence with bars shown in Figure 10, indicating whether the particular letter matches or not. A score and a percentage match are also provided to the user. After the doctor looks through the results, he then clicks on the search result for further information. From this link, the doctor is provided further details about the DNA sequence with information such as what the DNA sequence is, how it was discovered, the organism it belongs to, and the research article link for further details on the gene. Figure 11 provides an example of how a BLAST provides information of a particular DNA sequence. From this the doctor concludes that the patient may have colon cancer and will need immediate treatment.



Homo sapiens cell-line MaMel-95 methylthioadenosine phosphorylase-/antisense noncoding RNA in INK4 fusion protein (MTAP-ANRIL fusion) mRNA, complete cds

Sequence ID: KT386341.1 Length: 1425 Number of Matches: 1

Related Information

Map Viewer - aligned genomic context

Range 1: 1 to 120 GenBank Graphics

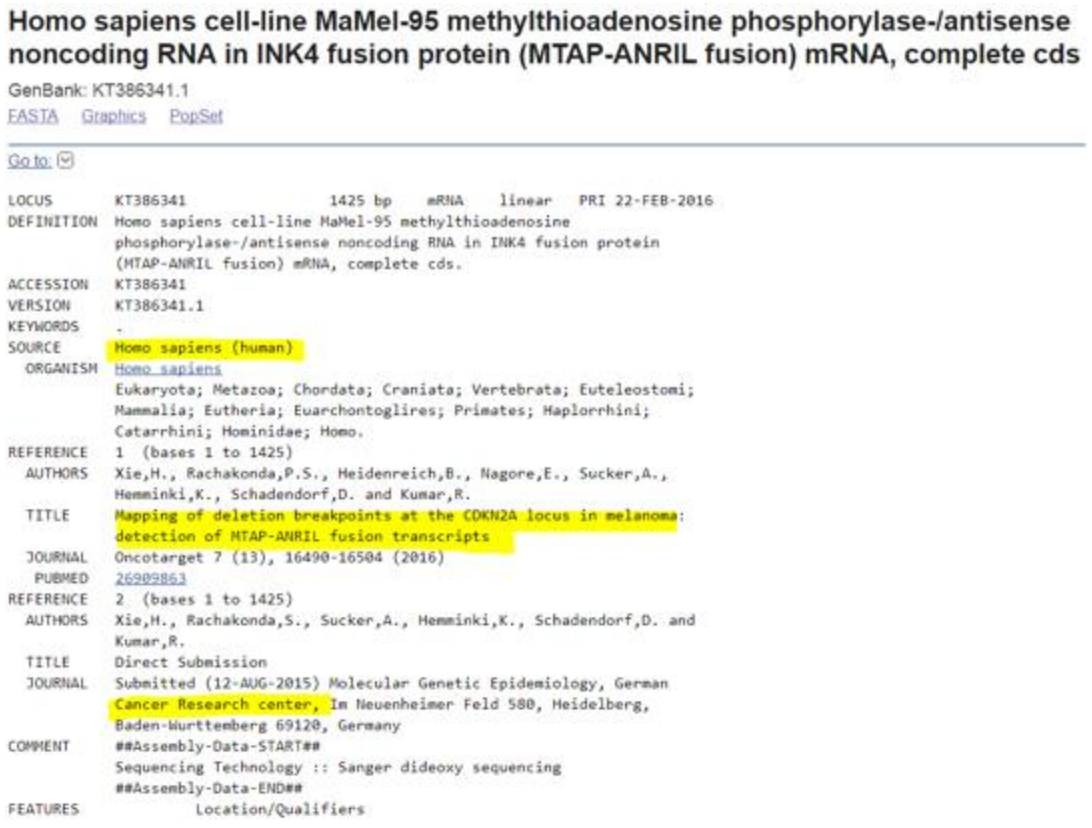
Score	Expect	Identities	Gaps	Strand
222 bits(120)	1e-54	120/120(100%)	0/120(0%)	Plus/Plus

```
Query  1    CTCCGCACTGCTCACTCCCGCGCAGTGAGGTTGGCACAGCCACCGCTCTGTGGCTCGCTT  60
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  1    CTCCGCACTGCTCACTCCCGCGCAGTGAGGTTGGCACAGCCACCGCTCTGTGGCTCGCTT  60

Query  61   GGTTCCCTTAGTCCCGAGCGCTCGCCCACTGCAGATTCCTTTCCCGTGCAGACATGGCCT  120
            ||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
Sbjct  61   GGTTCCCTTAGTCCCGAGCGCTCGCCCACTGCAGATTCCTTTCCCGTGCAGACATGGCCT  120
```

(National Center for Biotechnology Information, 2018)

Figure 10

BLAST Result with User and Database Query



Homo sapiens cell-line MaMel-95 methylthioadenosine phosphorylase-/antisense noncoding RNA in INK4 fusion protein (MTAP-ANRIL fusion) mRNA, complete cds

GenBank: KT386341.1

FASTA Graphics PopSet

Go to:

```
LOCUS       KT386341                1425 bp    mRNA    linear   PRI 22-FEB-2016
DEFINITION  Homo sapiens cell-line MaMel-95 methylthioadenosine
            phosphorylase-/antisense noncoding RNA in INK4 fusion protein
            (MTAP-ANRIL fusion) mRNA, complete cds.
ACCESSION   KT386341
VERSION     KT386341.1
KEYWORDS    .
SOURCE      Homo sapiens (human)
  ORGANISM  Homo sapiens
            Eukaryota; Metazoa; Chordata; Craniata; Vertebrata; Euteleostomi;
            Mammalia; Eutheria; Euarchontoglires; Primates; Haplorrhini;
            Catarrhini; Hominidae; Homo.
REFERENCE   1  (bases 1 to 1425)
  AUTHORS   Xie,H., Rachakonda,P.S., Heidenreich,B., Nagore,E., Sucker,A.,
            Hemminki,K., Schadendorf,D. and Kumar,R.
  TITLE     Mapping of deletion breakpoints at the CDKN2A locus in melanoma:
            detection of MTAP-ANRIL fusion transcripts
  JOURNAL   Oncotarget 7 (13), 16490-16504 (2016)
   PUBMED   26909863
REFERENCE   2  (bases 1 to 1425)
  AUTHORS   Xie,H., Rachakonda,S., Sucker,A., Hemminki,K., Schadendorf,D. and
            Kumar,R.
  TITLE     Direct Submission
  JOURNAL   Submitted (12-AUG-2015) Molecular Genetic Epidemiology, German
            Cancer Research center, Im Neuenheimer Feld 580, Heidelberg,
            Baden-Wurttemberg 69120, Germany
COMMENT     ##Assembly-Data-START##
            Sequencing Technology :: Sanger dideoxy sequencing
            ##Assembly-Data-END##
FEATURES             Location/Qualifiers
```

(National Center for Biotechnology Information, 2018)

Figure 11

Detailed Information on a Specific DNA Sequence

Use of Parallelism to Improve BLAST Performance and Related Work

The use of BLAST by so many biologists has required researchers to find ways to improve the speed and efficiency of BLAST by increasing the overall performance. Researchers have attempted to use parallelism techniques to improve BLAST.

An obvious way to accelerate the BLAST algorithm is by running them in multiple processors or multiple nodes. Many techniques were proposed to parallelize the algorithm and good results were obtained.

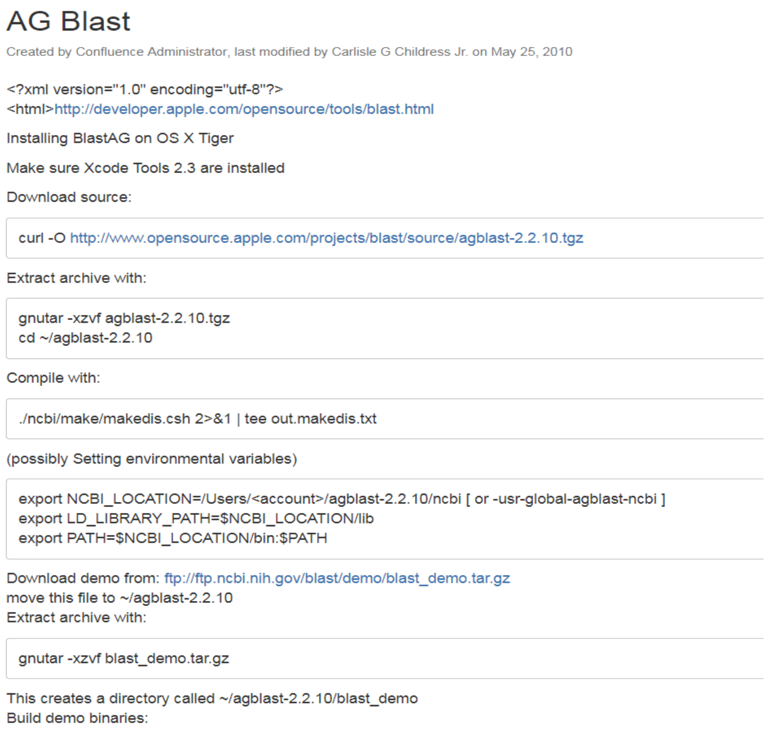
Biological sequence comparison is a very challenging problem. This is mainly due to the fact that we must find approximate pattern matching between a query sequence and a huge database.

Current research techniques use one of the following approaches to improve BLAST performance: vector instructions, multithreading, replicated databases, distributed databases and optimized batch queries (Zomaya, 2006).

1. Vector Instructions:

   One of the earlier techniques to improve BLAST performance was to use low-level vector parallelism to speed up the calculation of scores for sequence alignments. BLAST finds hits between statistically significant words and sequences in the database. It must find a local pairwise alignment and calculate a score for the alignment, before returning the best score found. This local pairwise alignment can be parallelized in a fine-grain manner using vector parallelism (Zomaya, 2006).

A version of BLAST called AGBLAST developed by Apple and Genentech use the Vector parallelism technique to improve BLAST performance. Research on AGBLAST has shown performance improvement up to an average of five times faster for BLAST queries with long sequences, which required more time calculating the local alignments. The web application for AG Blast is shown in Figure 12.


AG Blast

Created by Confluence Administrator, last modified by Carlisle G Childress Jr. on May 25, 2010

<?xml version="1.0" encoding="utf-8"?>
<html>http://developer.apple.com/opensource/tools/blast.html

Installing BlastAG on OS X Tiger

Make sure Xcode Tools 2.3 are installed

Download source:

```
curl -O http://www.opensource.apple.com/projects/blast/source/agblast-2.2.10.tgz
```

Extract archive with:

```
gnutar -xzvf agblast-2.2.10.tgz
cd ~/agblast-2.2.10
```

Compile with:

```
./ncbi/make/makedis.csh 2>&1 | tee out.makedis.txt
```

(possibly Setting environmental variables)

```
export NCBI_LOCATION=/Users/<account>/agblast-2.2.10/ncbi [ or -usr-global-agblast-ncbi ]
export LD_LIBRARY_PATH=$NCBI_LOCATION/lib
export PATH=$NCBI_LOCATION/bin:$PATH
```

Download demo from: ftp://ftp.ncbi.nih.gov/blast/demo/blast_demo.tar.gz
move this file to ~/agblast-2.2.10
Extract archive with:

```
gnutar -xzvf blast_demo.tar.gz
```

This creates a directory called ~/agblast-2.2.10/blast_demo
Build demo binaries:

(AGBLAST-UNIX, 2018)

Figure 12

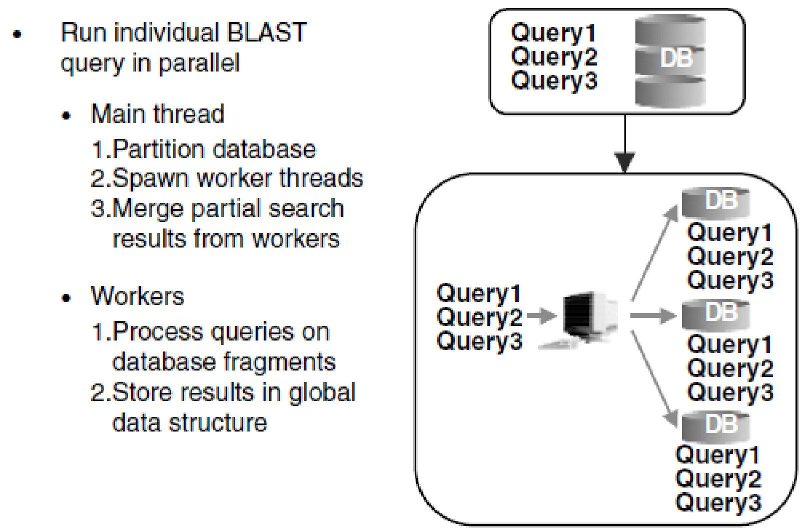
AGBLAST Web Snip Showing User Steps Required to Use AGBLAST

2. Multithreading

Another approach to improve BLAST performance is to use thread-level parallelism to compare queries to different parts of a sequence database. As a BLAST search is performed against sequence databases consisting of multiple sequences, searches can easily be performed in parallel using multiple threads.

A threaded BLAST search slices the given database into equal sized chunks according to the number of available processors. Each database chunk is then distributed to a predefined processor using memory mapping. Then, each processor is responsible for a thread scanning, a different database fragment. The sorted results are stored in a single global structure, which is shared by all processors and then combined to produce the final BLAST result which is displayed to the user. Figure 13 shows a snapshot overview of this process.

Some examples of Multithreading include NCBI BLAST and WU BLAST, which are programs that can be run in multithreaded mode on shared-memory multiprocessor (SMP) machines. The only downside of this approach is that, database portioning, thread creation, management, memory contention, and collecting search results in large overheads, which prevents BLAST from achieving peak performance. However, experimental studies have shown that multithreaded BLAST appears to achieve good performance in practice.


Run individual BLAST query in parallel
Main thread
1.Partition database
2.Spawn worker threads
3.Merge partial search results from workers
Workers
1.Process queries on database fragments
2.Store results in global data structure
Query1
Query2
Query3
DB

Zomaya, 2006)

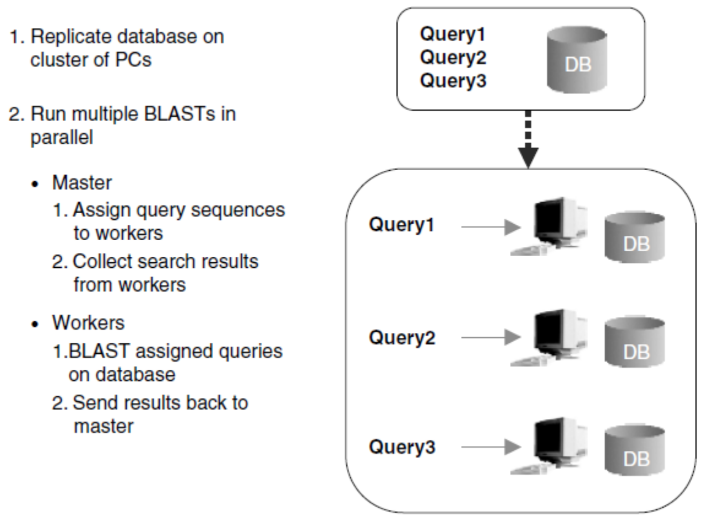
Figure 13

Multithreaded BLAST Algorithm

Replicated Databases

This approach required replicating sequence databases on a distributed memory system. This is one of the most commonly used methods to improve BLAST performance. The implementation adopts a master/slave paradigm to maintain load balance. When database size is small, a full copy of the sequence database can be stored in memory of each node. Batched queries can be split up evenly and assigned to each node.

The slave nodes then perform local BLAST searches and send the results to the master node. This method has shown to improve performance when there are a large number of batched queries. Figure 14 gives an overview of the algorithm. Some of the commonly used BLAST versions that use the replicated database approach include BeoBLAST and Hi-per BLAST.


1. Replicate database on cluster of PCs
2. Run multiple BLASTs in parallel
• Master
1. Assign query sequences to workers
2. Collect search results from workers
• Workers
1.BLAST assigned queries on database
2. Send results back to master
Query1
Query2
Query3
DB
Query1
DB
Query2
DB
Query3
DB

(Zomaya, 2006)

Figure 14

Replicated BLAST Algorithm

Distributed Databases

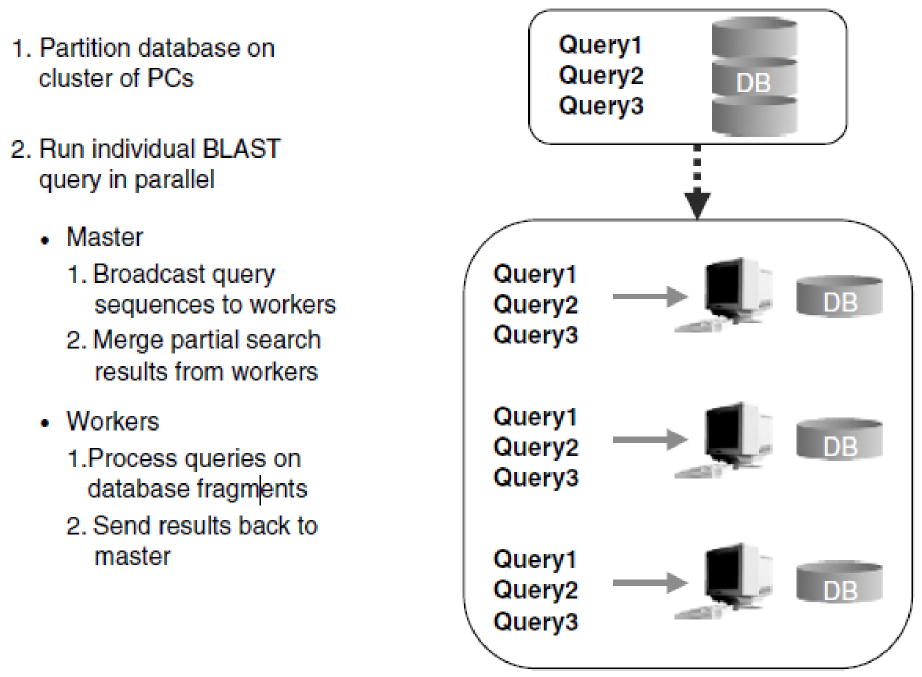
One of the downsides of a replicated BLAST algorithm is that when database size increases, they may no longer fit in the memory of individual computers. This may reduce performance for batched BLAST queries as each query will need to reload the sequence database into memory to scan for words. Distributed database approach solves this problem by exploiting the large amount of aggregate memory available in parallel computers through distributed sequence databases.

In this approach, the database can be split up, with each processor maintaining a portion of the sequence database small enough to fit in memory. Multiple BLAST queries can be processed without retrieving the database from slower disk storage requiring disk I/O. The downside is that implementing this approach is more complex as sequence database needs to be partitioned and also distributed equally between processors. Also, increased overheads are incurred as only partial results are calculated for each query from a single processor. The partial results are then combined between processors for each query. The overall performance is improved due to reduced disk access.

The algorithm for mpiBLAST is as follows. Figure 15 gives an overview of the algorithm.

1. The master node sends a message to each slave node to ask for a list of database fragments in its local directory.
2. The master node then assigns jobs to the worker node according to the database fragments at each node.

3. If a worker node is idle and has a local copy of a database fragment, the master node will send a message to that node, directing it to search the fragment.
4. The worker node then performs a local BLAST search for the database fragment assigned by the master.
5. Once the worker node has finished searching the database fragment, it will send its results and also an idle message to the master node.
6. As partial search results arrive from worker nodes, the master merges them into the master result list. Once all fragments have been searched, the master node will notify all slave nodes to terminate, and then output the merged results (Zomaya, 2006).


1. Partition database on cluster of PCs
2. Run individual BLAST query in parallel
• Master
1. Broadcast query sequences to workers
2. Merge partial search results from workers
• Workers
1.Process queries on database fragments
2. Send results back to master
Query1
Query2
Query3
DB

(Zomaya, 2006)

Figure 15

mpiBLAST Algorithm

Batch Query Optimization

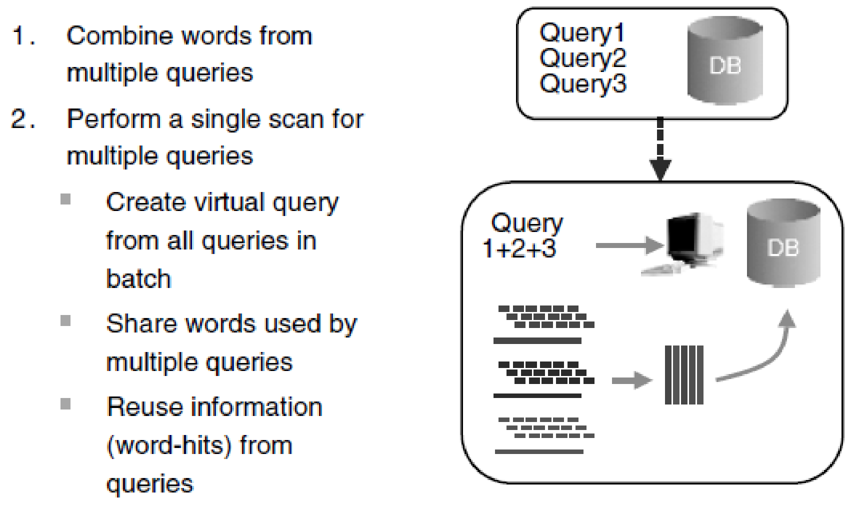
The last approach to improve BLAST performance is by reducing redundancies between multiple queries found in batch BLAST queries. By combining multiple BLAST queries, results and data already used can be reused and the number of scans done on the database can be reduced. This is important particularly for large databases that take a longer time to scan for information.

Batch Query Optimization does not necessarily make use of multiple processors but combining this approach and executing in parallel can greatly improve performance. Versions of BLAST that perform batch query optimization include BLAST++ and HT-BLAST.

The structure of BLAST++ algorithm is similar to BLAST. The major difference is how BLAST++ compiles its word list. The BLAST++ algorithm is as follows. Figure 16 contains the following steps:

1. BLAST++ creates a virtual query consisting of all queries.
2. When building a word list, BLAST++ maintains a list of (query ID, list of offsets) pairs for each word to record all the occurrences of the word in the entire set of batched BLAST queries.
3. The remaining steps of BLAST++ are identical to that of BLAST.

   In this approach a common word is searched only once for all the batched queries, compared to once for each query, hence reducing the computation time (Zomaya, 2006).


1. Combine words from multiple queries
2. Perform a single scan for multiple queries
Create virtual query from all queries in batch
Share words used by multiple queries
Reuse information (word-hits) from queries
Query1
Query2
Query3
DB
Query 1+2+3
DB

(Zomaya, 2006)

Figure 16

Batch Query Optimization

mpiBLAST

mpiBlast is a type of BLAST algorithm which achieves super linear speedup by segmenting a BLAST database. It is designed to work on a computer cluster using MPI library and adopts a master-slave relationship. In mpiBLAST the master node assigns the query sequence and database fragments to each worker node. The worker node performs the BLAST search on queries and sends the result to master node. When one worker node completes a task, the master node assigns a new fragment (of database) to it. This procedure is repeated until all queries have been searched. The master node merges all the results and sorts them according to the score. mpiBLAST performance is evaluated by measuring the speedup and efficiency in comparison to sequential NCBI BLAST version. The mpiBLAST algorithm consists of three steps as shown in Figure 17.

1. Segmenting and distributing the database
2. Running mpiBLAST queries on each node.
3. Merging the results from each node into a single output file (Zomaya, 2006).

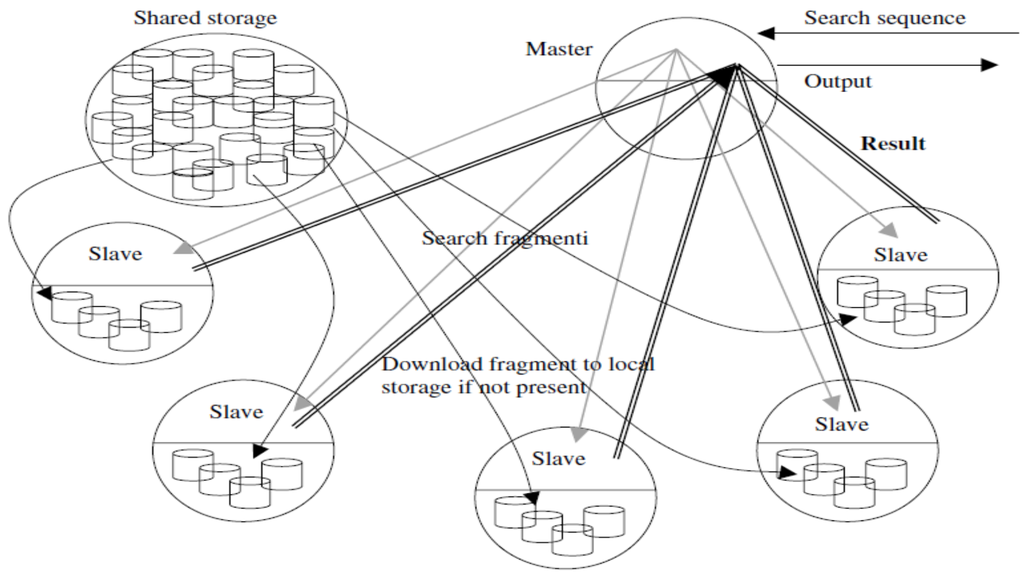


Figure 17

Master Slave Model of mpiBLAST

The Downside to mpiBLAST

One of the major problems with mpiBLAST is to do with the master write problem. In this scenario, the master process is responsible for sorting the intermediate results according to the score. This scenario has two drawbacks: the results processing is serialized by the master which can increase the time and decrease performance; second, the master memory may max out with all the intermediate results coming in. To remedy this issue, a parallel write is used. The worker threads after searching their fragment of database, converting their intermediate results into the final output, and sending the final output metadata to the master. As the size of each result alignment output is known to the master, the master then computes the offset ranges for each output and sends that information back to the workers. With the output offsets, the workers write the local output records in parallel. By locally buffering the output and parallel processing the results, the mpiBLAST removes the performance bottleneck.

Chapter 5: Experiment and Related Work

This chapter focuses on the algorithm that was implemented for the purpose of this paper and also the methodology that was used. Comparisons made between serial and parallel implementation of BLAST with results will being tabulated.

Overview of the BLAST Algorithm Implemented

The algorithm of the BLAST program that was implemented for this paper has the main objective to show performance results is explained in detail below. For the purposes of simplicity and to show performance improvements, the program was kept simple and is a hybrid of various implementation of BLAST.

Below is a detailed pseudocode of the BLAST Algorithm implemented for this paper.

1. User enters the DNA Sequence that he/she wants to search in the database (the database for this project would be a .txt file).

   Example Output of the Program:

   Please enter the DNA Sequence that you would like to match:

   User Entry: ATGCCCGTCATTCC

2. The first step for search is to break the user entry into three letters:

   The program breaks the user input into three-letter words: ATG, TGC, GCC, CCC, CCG, CGT, GTC, TCA, CAT, ATT, TTC, TCC.

3. The program then searches the database (the user specified .txt file) for sequences that match the three-letter words which are known as “Hooks.” Figure 18 provides a pictorial representation of how the hooks are searched by the program in the respective database file that the user decides to search.

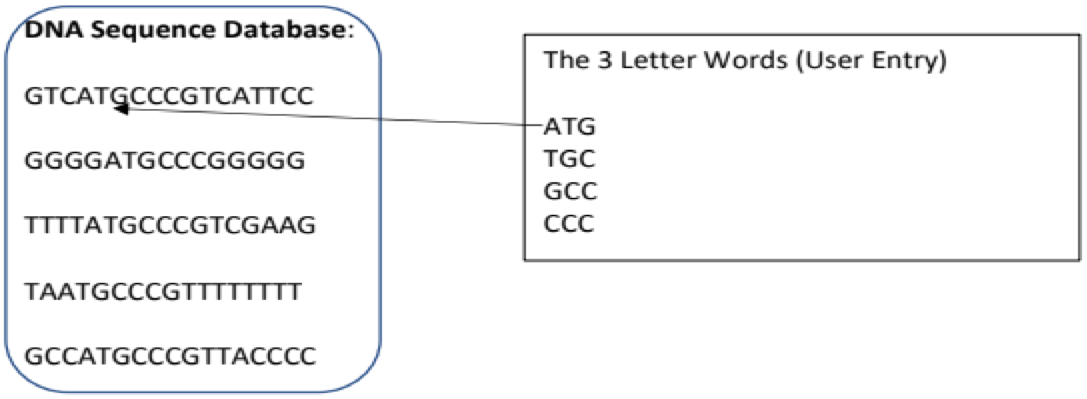


Figure 18

Finding the Hook in the Database

4. After finding the “Hook” by matching the three-letter words to the database, the algorithm then moves left and right along the DNA sequence of interest.
   For example, in the above instance, the first sequence was of interest since we were able to find the three-letter word: ATG in the sequence. Therefore, the algorithm moves left and right through the sequence to look for similarities between other letters in the sequence.

   GTC**ATG**CCCGTCATTCC

5. When the algorithm moves left and right the program keeps a score using a program counter. With every match, the algorithm adds +1 to the overall score, and -1 for any mismatch.

In the above example:

User Entry: **-- ATG**CCCGTCATTCC

Database (Sequence of Interest): TC**ATG**CCCGTCATTCC

The overall score is calculated based on the algorithm keeping a counter for the number of hits and misses. The counter begins with a score of 0, and then begins counting the score from the hook, which in this case is "ATG." As the hook consists of a three-letter word, the score then increments to +3. The algorithm then moves all the way to the left and right of the database sequence and increments/decrements the score based on if the word is a hit or a miss. After the algorithm reaches the end of the sequence or the beginning of the sequence it stops and provides the user with a total score.

The algorithm searches the database sequence only if it "sees" a hook in the sequence. If the database sequence consists of multiple hooks, then it prioritizes the first hook seen compared to other hooks. In other words, the first hook that the algorithm encounters in the database sequence is the hook that it considers.

Another important observation to point out here is that the algorithm for BLAST searches is more complex in nature, with a lot more statistical analysis and comparisons made before a score is provided to the user. For the purpose of this paper, the algorithm here was kept as simple as possible to solely provide comparisons of serial and parallel implementation of BLAST which is the main objective of the paper. This is done so as to not to heavily focus on the complexity of the BLAST algorithm which may cause the reader to stray away from the objective and the intent of the paper. Rather, the objective of the paper has to do with parallelism and its advantages on a search algorithm such as BLAST. Table 2 provides a detailed overview of how the scoring algorithm works.

Table 2

Sequence Scoring

User Entry	**Database**	**Score**	**Total Score (Initial Counter = 0)**
-	T	-1	-1
-	C	-1	-2
A	**A**	**+3 (HOOK)**	**+1**
T	**T**		
G	**G**		
C	C	+1	+2
C	C	+1	+3
C	C	+1	+4
G	G	+1	+5
T	T	+1	+6
C	C	+1	+7
A	A	+1	+8
T	T	+1	+9
T	T	+1	+10
C	C	+1	+11
C	C	+1	+12
		Overall Score:	**12**
		Three letter Hook:	**ATG**

6. After the algorithm matches the user entry with the sequence of interest it provides an output to the user that looks something like this.

   User Entry: - - **ATG**CCCGTCATTCC

   Database (Sequence of Interest): TC**ATG**CCCGTCATTCC

   The program also provides the user with the score and a percentage value of how much matched.

Methodology

The parallel implementation of BLAST algorithm is a done in the language C. The program can be executed simultaneously in more than one processor. In this section, we will discuss the details on how both the design and implementation of parallel BLAST algorithm have been carried out.

<u>Step 1: Splitting the Database</u>. In this algorithm, the database is split up by the master node, with each processor maintaining a portion of the sequence database small enough to fit in the memory. Multiple BLAST queries can be processed without retrieving the database from slower disk storage requiring disk I/O. This is known as distributed databases. Distributing data among the slave processors involves sending data from the root processor (master, a processor with rank 0) to all participating processors including itself.

<u>Step 2: Managing Communication between Interaction Tasks</u>. Most of the communication occurs during database fragmentation and distributing the fragments among processors would take place in the beginning, and at the end while gathering outputs from them. The communication is static and synchronous.

Step 3: Input Query Search in Database. During this step, the user query is broken into words by the master processor and the words are distributed to the worker nodes. These three letter words (Example: “ATG,” “GGC,” GTA”) are then searched by the slave processors in their respective database fragments. Each processor performs the expansion phase on the respective database DNA sequence and calculates the score.

If a node is idle during this phase and has a local copy of the database fragment, the master node will send a message to that node, directing it to search the database fragment for the user entry.

Once the slave node has finished searching the database fragment, the slave processor sends the Highest Scoring Pair’s (HSP’s) to the master processor. After this, the worker node will send an idle message to the master node.

Step 4: Merging of the Results. As the partial search results arrives from the slave processors, the master merges them into a master result. Once all database fragments have been searched, the master node will notify all the slave nodes to terminate and then the master outputs the merged results.

Use of pThreads

pThreads are defined as a set of C language programming types and procedure calls. Implemented with a pthread.h/include file and a thread library. pThreads are used for many reasons; the main motivation is to realize the parallel program performance gains. When compared to the cost of creating and managing a process, a thread can be created with much less OS overhead. Managing threads requires fewer system resources than managing processes.

Also, inter-thread communication is more efficient and, in many cases, easier than to manage inter-process communication (Akl, 2014).

A thread is a sequence of such instructions within a program that can be executed independently. It is an independent set of values for the processor registers (for a single core). Since threads includes the Instruction Pointer (aka Program Counter), it controls what executes in what order. Threads also includes the Stack Pointer, which had better point to a unique area of memory for each thread or else they will interfere with each other (Akl, 2014).

Threads are the software unit affected by control flow (function call, loop, goto) and because instructions operate on the Instruction Pointer, that belongs to a particular thread. Threads are often scheduled according to some prioritization scheme (although it is possible to design a system with one thread per processor core, every thread always runs, and no scheduling is needed).

The value of the Instruction Pointer and the instruction stored at that location is sufficient to determine a new value for the Instruction Pointer. For most instructions, this simply advances the IP by the size of the instruction, but control flow instructions change the IP in other predictable ways. The sequence of values the IP takes on forms a path of execution weaving through the program code, giving rise to the name "thread."

Independent flows of control are possible because a thread maintains its own:

- Program Counter
- Stack Pointer
- Registers
- Scheduling Properties

- Signals
- Thread specific data (Akl, 2014)

Threads within the same process share the same resources so an issue to look out for: the changes made by one thread in a shared system will be seen by all other threads. This brings up the issue of thread safety.

Thread-safeness refers to an application's ability to execute multiple threads simultaneously without clobbering shared data or creating race conditions. A situation where a race condition would take place: an application creates several threads, each of which makes a call to the same library routine. The library routine accesses/modifies a global structure in local memory. The issue that might happen is when as each thread calls this routine it is possible that the threads may try to modify this global structure/memory location at the same time. Therefore, the routine should employ some sort of synchronization constructs to prevent data corruption. This issue where threads trying to modify a global structure at the same time is seen in a shared memory space model. A forked process is considered a child process and forked processes share no like code, data, or stack with the parent process; whereas, a threaded process can share code but has its own stack. The purpose of fork() is to create a new process, which in turn becomes the child process of the system call. The process will then execute the next instruction following the fork() system call. In this platform, two identical copies of the address space, code and stack are created; one for parent and the other for child. On the other hand, the purpose of pthread is to create a new thread. The thread within the same process can communicate using shared memory whereas processes have their own memory space. Forks() are harder to synchronize than pthreads.

To take advantage of the pThreads, a program must be able to be organized into discrete, independent tasks which can execute concurrently. For example, if routine 1 and routine 2 can be interchanged, interleaved, and/or overlapped in real time, they are candidates for threading. In parallel programming, the master/slave method is used. A single thread, the master thread assigns work to other threads, the workers. Typically, the master handles all input and parcels out the work to other tasks. In some instances, the master also gathers the results from the worker threads and performs the necessary calculations.

There are two forms of master/slave models:

1. Static Load Balancing: This type of load balancing is applied before the execution of any process. It is referred to as the mapping problem or scheduling problem. A potential static load-balancing technique is by assigning tasks in sequential order to processes coming back to the first when all processes have been given a task. Another way is to select processes at random to assign tasks.
2. Dynamic Load Balancing: In this form, when the job gets done, the worker thread seeks the next task. Division of tasks is dependent upon the execution of parts of the program as they are being executed. A variant of Dynamic Load Balancing is Centralized Dynamic Load Balancing. In this form, the master process holds a collection of tasks to be performed by the slave process. Tasks are sent to slave processes and when a task is completed a slave process requests another task when done.

POSIX Threads

POSIX threads is a standardized C language threads programming interface designed to develop portable threaded applications for UNIX systems. Implementations that adhere to this standard are referred to as POSIX threads or pThreads. pThreads are defined as a set of C language programming types and procedure calls. pThread library is considered for the implementation for the following reasons:

1. Primary motivation is to realize potential program performance gains.
2. Compared to the cost of creating and managing a process, a thread can be created with much less operating system overhead. Managing threads requires fewer system resources than managing processes.
3. All threads within a process share the same address space. Inter-thread communication is more efficient and, in many cases, easier to user than inter-process communication.
4. Tasks that are more important can be prioritized over less important tasks.

A computer program becomes a process when it is loaded from some store into the computer's memory and begins execution. A process can be executed by a single processor of a set of processors. A thread on the other hand, is a sequence of such instructions within a program that can be executed independently of the other code. In order to define a thread formally, we must first understand the boundaries of where a thread operates.

Example Program with the Use of pThread:
Hello World

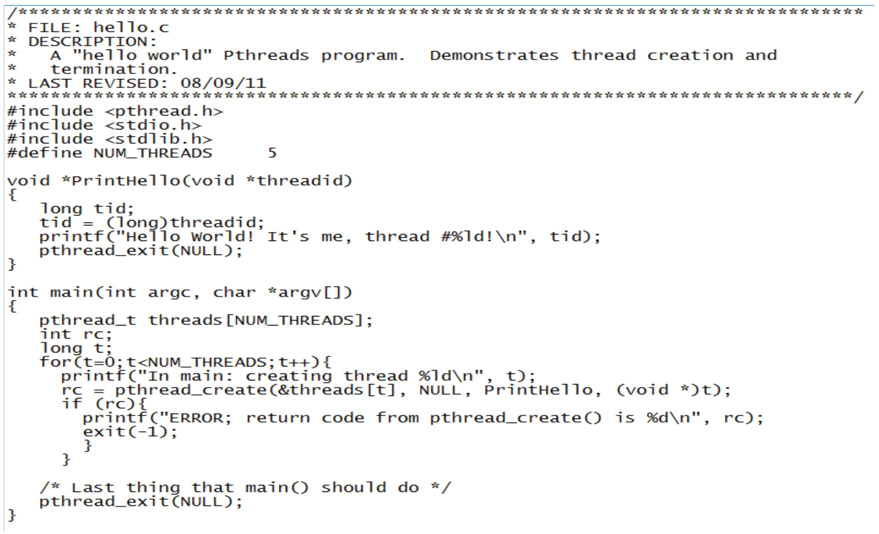
The simple pThreads program in C language in which the threads print "*Hello World!*" message is shown in Algorithm 1. The function *PrintHello* is the routine that will be executed by the child threads. This routine prints the string that has been passed to it.

Following the routine is the main function, the starting point for the program. We start by declaring the variables for child threads. *pthread_t* is the type of the variable to be declared.

After declaring the variables, we need to initialize the threads. We use *pthread_create* routine provided by the Pthreads library to accomplish the same. *pthread_create* will initialize the thread, the thread attributes, the address of the routine the thread has to start executing and the parameters for that routine. As soon as a thread is created, it will start executing the routine that has been assigned to it by *pthread_create*. The pthread_create() routine allows the programmer to pass one argument to the thread start routine.

From the code it can be clearly seen that both the worker threads are executing the same routine. Each thread has its own copy of the stack variables for the routine. For instance, Thread1 will execute the routine with *"Hello World! It's me Thread 1."*

pthread_exit is used to explicitly exit a thread. If main () finishes before the threads it creates, and exits with pthread_exit(), the other threads will continue to execute. Otherwise, they will be automatically terminated when main() finishes. pthread_exit () routine does not close any files. Files that are opened inside of the thread will remain open after the thread is terminated.



```
/******************************************************************************
* FILE: hello.c
* DESCRIPTION:
*   A "hello world" Pthreads program.  Demonstrates thread creation and
*   termination.
* LAST REVISED: 08/09/11
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   long tid;
   tid = (long)threadid;
   printf("Hello World! It's me, thread #%ld!\n", tid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc;
   long t;
   for(t=0;t<NUM_THREADS;t++){
     printf("In main: creating thread %ld\n", t);
     rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
     if (rc){
       printf("ERROR; return code from pthread_create() is %d\n", rc);
       exit(-1);
       }
     }

   /* Last thing that main() should do */
   pthread_exit(NULL);
}
```

Algorithm 1

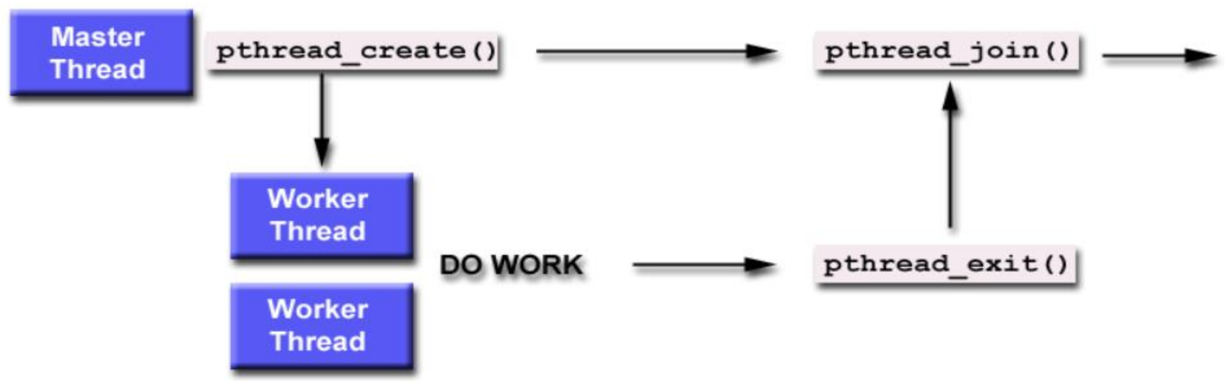
Hello World Using pThreads

Output of the code is as follows:

```
In main: creating thread 0

In main: creating thread 1

In main: creating thread 2

In main: creating thread 3

In main: creating thread 4

Hello World! It’s me, thread #0!

Hello World! It’s me, thread #1!

Hello World! It’s me, thread #2!

Hello World! It’s me, thread #3!

Hello World! It’s me, thread #4!
```

Thread Joining

Joining is a way to accomplish synchronization among threads as shown in Figure 19. The pthread_join () subroutine clocks the calling thread until the specified thread terminates. The programmer is able to obtain the target thread's termination return status if it was specified in the target thread's call to pthread_exit (). If successful, the pthread_join () function returns zero; otherwise an error number shall be returned to indicate the error.



(Costa & Lifschitz, 2003)

Figure 19

pThread Join()

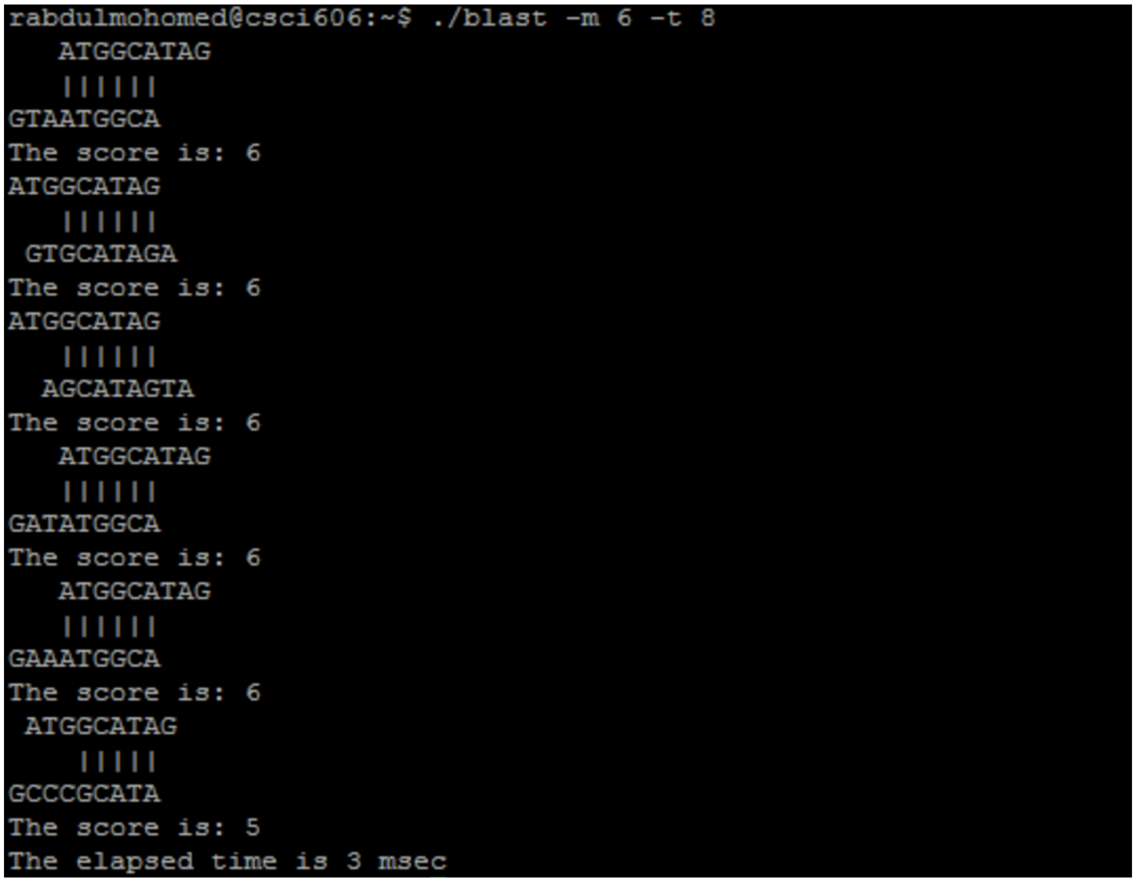
The Experiment and Its Results

BLAST is a relatively fast program that efficiently calculates the sequence alignment of two biological sequences. BLAST search first breaks the query into words of length w (default word length = 3) and compare them to each database sequence. The matching words (or seeds) are then extended in both directions until the score of alignment drops below a threshold to form the High Scoring Segment Pair (HSP). Before conducting a rigorous amount of detailed experiments, a simple and quick experiment was performed to understand how the parallel implementation differs from the serial implementation of BLAST. Both the serial and parallel program were run using a database file that contained 999 DNA sequences. The number of threads used was increased exponentially to a maximum of 16. Figure 20 shows an example run of the program and Table 3 shows an example of the study with increasing amount of threads and keeping the number of database sequences constant.

Table 3

Performance of BLAST with Increasing Amount of Threads

No. of DB Sequences	**No. of Threads**	**Parallel Implementation (Time in ms)**
999	1	51
999	2	27
999	4	18
999	8	16
999	16	16



```
rabdulmohomed@csci606:~$ ./blast -m 6 -t 8
   ATGGCATAG
   ||||||
GTAATGGCA
The score is: 6
ATGGCATAG
   ||||||
 GTGCATAGA
The score is: 6
ATGGCATAG
   ||||||
  AGCATAGTA
The score is: 6
   ATGGCATAG
   ||||||
GATATGGCA
The score is: 6
   ATGGCATAG
   ||||||
GAAATGGCA
The score is: 6
 ATGGCATAG
    |||||
GCCCGCATA
The score is: 5
The elapsed time is 3 msec
```

Figure 20

Example Run of the Program

Efficiency here is defined as the time taken by the program to search the user entry within the database file. From the experiment it was noted that the maximum efficiency was achieved with eight threads and eight database fragments. After that, the efficiency of the program goes down. This could be due to the overhead costs with communication when the number of worker threads increase in parallel write. Sixteen threads were utilized in the last experiment and the efficiency of the program decreased. This could be due to several reasons but one of the main reason would be that the master thread could be waiting on all the worker threads to search through their database fragments and provide the master thread with the searches and the respective score. The worker threads search through their respective database fragments, provide the top results and the scores to the master thread. The master thread then sorts through the matches provided by the worker threads and displays the results to the user.

The following are the results of the experiment shown in this paper using a system with the following configuration:

- Hostname: csci606
- Number of cores: 8
- Processor: Intel Xeon CPU E5-2680 v2 @2.8 Ghz
- Memory: 16 GB

Serial BLAST vs. Parallel BLAST

In the next experiment, Serial BLAST algorithm was compared with Parallel BLAST algorithm. Both programs ran on the exact same hardware using the same database file. The results of the experiment are shown below. The experiment was performed on various sizes of database files and with increasing number of threads. The objective of this experiment is to show the advantage of using pThreads over a serial implementation.

Table 4

Serial vs. Parallel Implementation Performance: DB = 1000 DNA Sequences

Number of DB Sequences = 1000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	51	
2		66
4		61
6		76
8		72

Table 5

Serial vs. Parallel Implementation Performance: DB = 10,000 DNA Sequences

Number of DB Sequences = 10,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	97	
2		98
4		111
6		118
8		120

Table 6

Serial vs. Parallel Implementation Performance: DB = 100,000 DNA Sequences

Number of DB Sequences = 100,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	197	
2		129
4		79
6		65
8		66

Table 7

Serial vs. Parallel Implementation Performance: DB = 500,000 DNA Sequences

Number of DB Sequences = 500,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	403	
2		156
4		132
6		128
8		96

Table 8

Serial vs. Parallel Implementation Performance: DB = 1,000,000 DNA Sequences

Number of DB Sequences = 1,000,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	785	
2		543
4		365
6		298
8		208

Table 9

Serial vs. Parallel Implementation Performance: DB = 5,000,000 DNA Sequences

Number of DB Sequences = 5,000,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	3815	
2		2976
4		2762
6		1428
8		992

Table 10

Serial vs. Parallel Implementation Performance: DB = 10,000,000 DNA Sequences

Number of DB Sequences = 10,000,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	7748	
2		4672
4		3892
6		3196
8		1999

Table 11

Serial vs. Parallel Implementation Performance: DB = 50,000,000 DNA Sequences

Number of DB Sequences = 50,000,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	43,450	
2		28,282
4		17,462
6		10,345
8		8563

Table 12

Serial vs. Parallel Implementation Performance: DB = 100,000,000 DNA Sequences

Number of DB Sequences = 100,000,000 DNA Sequences		
No. of Threads	**Serial Implementation (Time in milliseconds)**	**Parallel Implementation (Time in milliseconds)**
1	82,600	
2		56,272
4		31,938
6		24,726
8		18,282

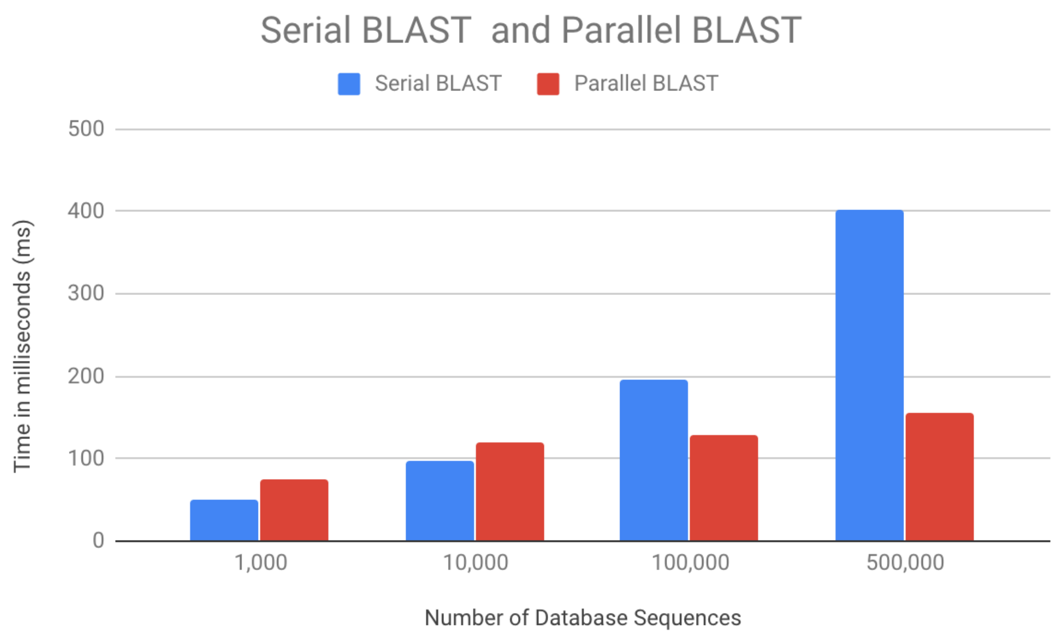


Figure 21

Serial vs. Parallel BLAST 1000-500,000 DB Sequences

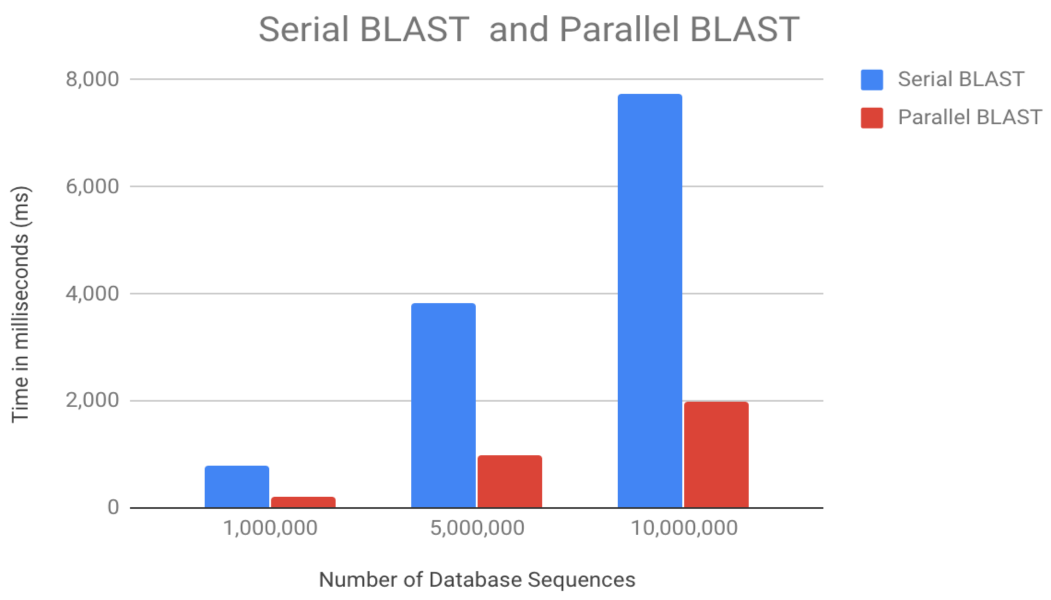


Figure 22

Serial vs. Parallel BLAST 1,000,000-10,000,000 DB Sequences

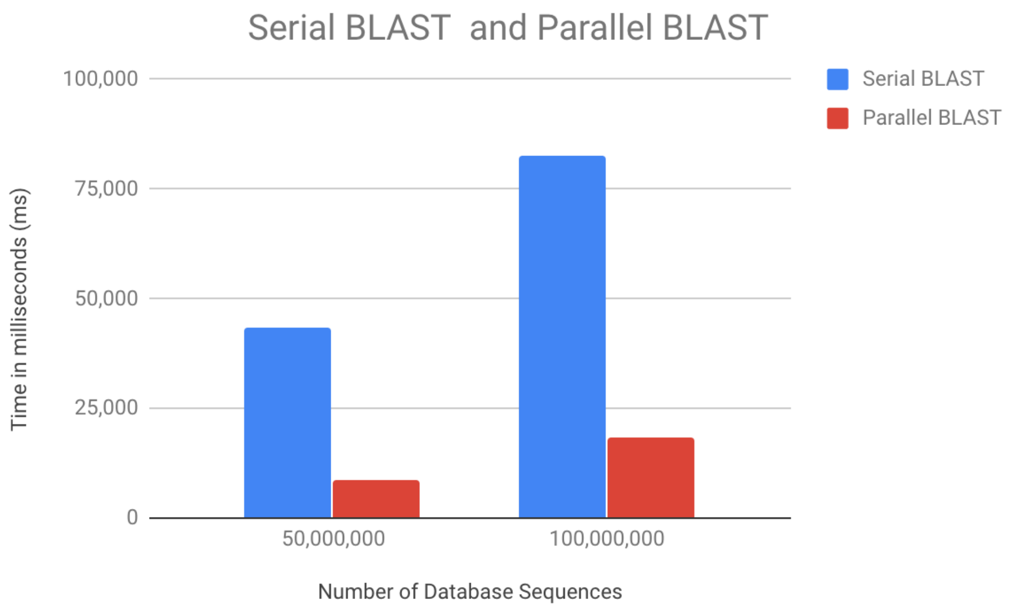


Figure 23

Serial vs. Parallel BLAST 50,000,000-100,000,000 DB Sequences

Per the results in Figures 21, 22, and 23, the parallel program's performance was significantly better compared to the serial implementation of BLAST. As the database file size grew, the serial implementation of BLAST showed a significant time difference compared to the parallel implementation.

Initially, as the database file size was less than 10,000, the parallel implementation was slightly less efficient compared to the serial implementation. This could be due to the overhead costs which worsens the performance when there is less sequences to search. Also, as the database files grew in size, the performance of the parallel implementation improved. As the number of threads increased, the parallel implementation performance was affected. Again, the parallel performance suffered when the number of threads increased. This experiment helps us come to a conclusion that there should exist a right balance in the number of threads with the database file. If the database file is too small or the number of threads is too large for the amount of database sequences to search through, then the parallel performance suffers.

In parallel computation, in which multiple processors are all focused on the solution of a single problem. Many of these applications may be speed up by preprocessing their input data, carrying out a large number of independent and concurrent computation on the preprocessed data which is done in parallel, and post-processing the results of the independent computations to construct the final outputs. A significant approach to speed up the search would be to split the database to create small subtasks which gets assigned to the worker threads.

Speedup here is defined as the ratio of the time for the execution of the sequential code on a single core to the execution time of the parallel code. Therefore, speedup = Ts/Tp where Ts is the execution time of the sequential code, and Tp is the execution time for the parallel code. We saw an average speed-up of 3.67 when using the parallel program compared to the serial program.

Related Work

Researchers have extended the above experiment further and have provided further analysis on performance with regards to size of the database, the size of user batch entries, and the length of the DNA sequence.

Performance vs. Database Size

The first step is to analyze if there is an impact of sequence database size on performance. The actual BLAST database can store four nucleotides as a single byte, so a sequence database with 4 billion sequences would require about 1 GB of space. Figure 24 represents the performance of various versions of BLAST including mpiBLAST for a wide range of database sizes for nine processors on a PC cluster. The Y axis represents execution time and the X axis represents sequence database size in billions of sequences.

The execution times tend to increase for larger databases as more sequences must be examined for each user query. The results show that before the database size reaches the memory limit, both BLAST and BLAST++ perform better than mpiBLAST. This is due to lower parallelization overhead enjoyed by replicated BLAST and replicated BLAST++. However, once database sizes increase over 4 billion DNA sequences, the databases no longer fit in the memory of a single node, this is due to the choice of the researchers to use PC nodes that have 1 GB memory. The performance of BLAST and BLAST++ degrades sharply as portions of databases are evicted from the file cache and need to be slowly reloaded from the disk (Naruse & Nishinomiya, 2002).

In comparison, the performance of mpiBLAST does not experience major performance issues with an increase of the database size to over 4 billion sequences or when the database size exceeds the limit of a certain PC node. Instead performance degrades linearly with an increase of database sizes. Steep performance drops are avoided in mpiBLAST since each node only performs searches on a portion of partitioned database (Naruse & Nishinomiya, 2002). Sharp performance drops will only be noticed in mpiBLAST when each database fragment becomes too large for the memory of each individual node. This experiment results demonstrates that mpiBLAST works best for large sequence databases that do not fit into the memory of a single node.

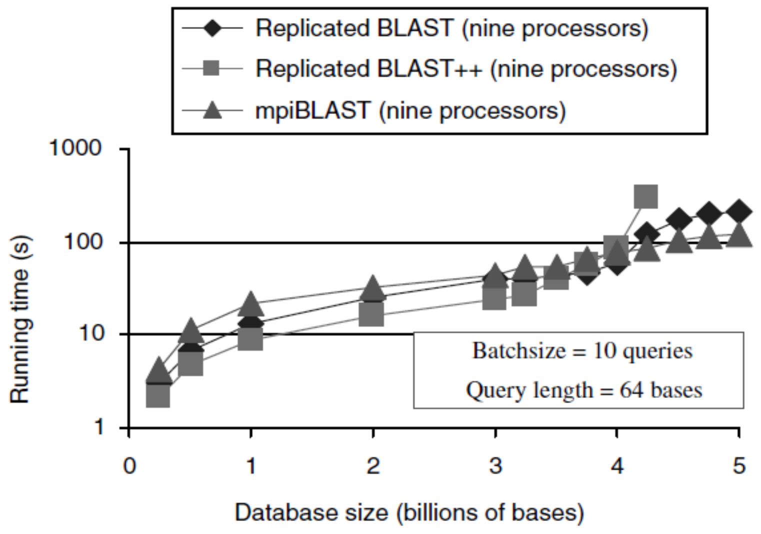


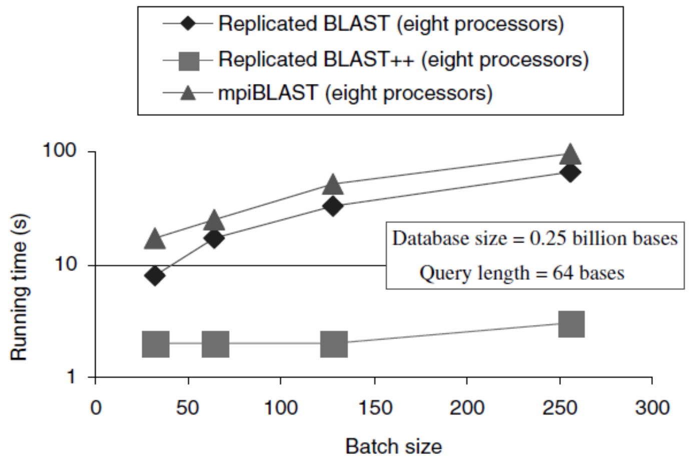
Figure 24

Performance vs. DB Size

Performance vs. Batch Size

The next step that the researchers looked at is to see the impact on the BLAST algorithm performance with respect to the number of queries entered by the user. Figure 25 represents the performance of the various versions of BLAST as the batch size varies between 32 and 256. The results were obtained on a PC cluster using eight processors. In the figure, the Y axis represents the execution time and the X axis represents query batch size. The database size of 250 million DNA sequences and the query length is 64 letters long (Costa & Lifschitz, 2003).

Results as expected show that the execution time increases for all the variations of BLAST as the batch size increases because more user queries need to be searched through the database. BLAST++ is able to reduce execution times for batched BLAST queries even for small batches. It also reduced redundancy and the execution time only increases slightly as the batch grows. BLAST++ is able to accomplish this efficiency by creating a virtual query sequence that consists of all the user query sequences (the batch sequence). Unfortunately, a side effect of maintaining all the additional information for the combined query sequence, increases the memory usage for BLAST++ compared to BLAST.



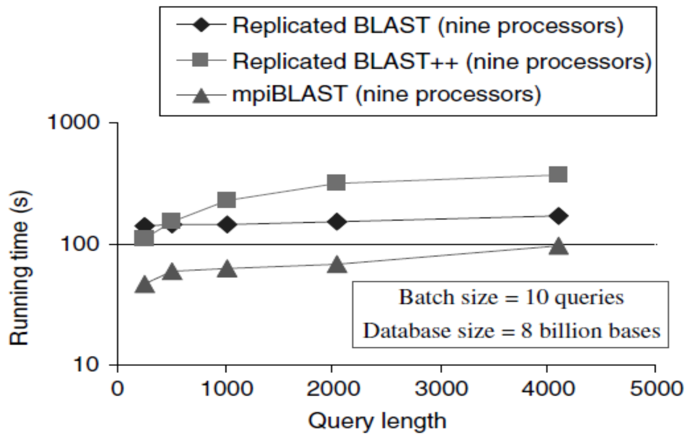
(Zomaya, 2006)

Figure 25

Performance vs. Batch Size

Performance vs. Query Length

The researchers vary the user query length from 256 words to 4096 words. mpiBLAST achieves the best performance because the large sequence database cannot fit in the memory of a single node. Serial implementation of BLAST required 2131.8 seconds on one of the machines. Compared to the parallel implementation of BLAST with 11 threads, only 130 seconds was required, representing an improvement in the speedup of the algorithm. Figure 26 displays the time in seconds required with increasing query length sequences for various parallel implementations of BLAST.



(Zomaya, 2006)

Figure 26

Performance vs. Query Length

Chapter 6: Conclusion and Future Work

The experiments carried out shows that there are many factors affecting the performance of BLAST searches. These factors include, size of DNA sequence database, the length of the user query, and the number of user queries. Among these factors, the size of the sequence database seems to be the most important in determining the total search time. This is because databases can take up a lot of memory and also because the BLAST algorithm is an exhaustive search algorithm which examines each and every database DNA sequence for every user query. Further research is being performed to investigate new data indexing and organization methods for these databases. These new methods show promise for improving the BLAST search performances further (Costa & Lifschitz, 2003).

From this paper we have seen that there are various ways to improve the performance of BLAST searches by exploiting parallelism at number of levels. Some of the notable ways include vector instructions, multithreading, replicated databases, distributed databases and optimized batch queries, or a hybrid of any of these.

Some of the future work are as follows:

1. Integration of Existing Approaches: Biologists along with researchers are developing version of BLAST that combine various implementations of parallel BLAST algorithms. pThreads and UMD-BLAST is a first step in this direction.
2. Preprocessing Sequence Databases: Another area that researchers are currently looking into is to analyze and pre-process DNA sequence databases to make them more efficient for BLAST searches. Such pre-processing can take advantage of knowledge of properties of the BLAST algorithm. For example, the BLAST scoring

algorithm may be analyzed to index and partition sequence databases to reduce the portion of the database that must be scanned.

3. Exploiting Grid Resources: For large user query searches, researchers may attempt to take advantage of CPU cycles in large grid systems. Grid computing attempts to provide a common interface for a variety of computing resources, ranging from dedicated computing clusters at supercomputing centers to spare CPU cycles harvested from pools of idle computers. Researchers are also looking into the interfaces and infrastructure required for supporting BLAST in a grid environment. Prototype grid-based versions of BLAST attempt to provide transparent user interfaces for performing BLAST searches on a grid. Such system will increase in importance as grid computing becomes even more popular.
4. Alternative Search Algorithms: Lastly, researchers are looking into developing new search algorithms that yield results as precise or even better than BLAST along with better performance. Examples include BLAT, PatternHunter, and MEGABLAST. Although these new algorithms can be quite powerful and precise, BLAST is so frequently and commonly used tool that convincing biologists to use a new search tool will require much more evidence of positive results and also training (Akl, 2014).

References

AG BLAST-UNIX. (2018). *VCU wiki*. Retrieved October 7, 2018, from wiki.vcu.edu/display/unix/AGBlast.

Akl, S. G. (2014). *Parallel sorting algorithms*. Academic Press.

Aluru, S (2003, September). *Computational biology on parallel computers*. European Control Conference, pp. 3370-3377.

Costa, R., & Lifschitz, S. (2003). Database allocation strategies for parallel blast evaluation on clusters. *Distributed Parallel Databases, 13*, 99-127.

Isa, M. N. (2013). *High performance reconfigurable architectures for biological sequence alignment*.

Julich, A. (1995). Implementations of BLAST for parallel computers. *Computer Application Bioscience, 11*(1), 3-6.

Naruse, A., & Nishinomiya, N. (2002). Hi-per BLAST: High performance BLAST on PC cluster system. *Genome Informatics, 13*, 254-255.

National Center for Biotechnology Information. (2018, October). *BLAST: Basic local alignment search tool.* U.S. National Library of Medicine: National Center for Biotechnology Information.

Zomaya, A. Y. (Ed.). (2006). *Parallel computing for bioinformatics and computational biology: Models, enabling technologies, and case studies*. John Wiley & Sons.