Algorithms for String Searching on A Beowulf Cluster

by

Kishore Ramakrishna Kattamuri

A thesis
submitted to the Department of Computer Science
at Florida Institute of Technology
in partial fulfillment of the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
December, 2003

We the undersigned committee hereby recommend
that the attached document be accepted as fulfilling in
part the requirements for the degree of
Masters of Science in Computer Science

Algorithms for String Searching on A Beowulf Cluster

by
Kishore Ramakrishna Kattamuri

_____

William D Shoaff, Ph.D.
Associate Professor
Program Chair, Department of Computer Sciences

_____

Marius Calin Silaghi, Ph.D.
Assistant Professor
Department of Computer Sciences

_____

David Carroll, Ph.D.
Assistant Professor
Department of Biological Sciences

Abstract

Title:
Algorithms for String Searching on A Beowulf Cluster

Author:
Kishore Ramakrishna Kattamuri

Principle Advisor:
William D Shoaff, Ph.D.

String matching is a subject of both theoretical and practical interest in computer science. Theoretically, time and space complexity of the algorithms and on the practical side searching for keywords, names, phrases etc., is common. Searching for patterns in DNA (Deoxyribose Nucleic Acid) is one such application and is the ultimate area for this study.

This study resulted in efficient parallel way to search for patterns, the program being operated on a parallel computer. This research provides basis for a further study and help in selecting the search algorithms for broader use.

The algorithms proposed in this research try to solve the problem in both the single pattern and multiple pattern search areas. The outcome of this study was very encouraging and paved path for the possible future study. The algorithms were written in C using the MPI model on the Beowulf cluster of Florida Tech

(Bluemarlin). The timings were compared against each other. The test data set used was kept uniform throughout the experiments.

Though several ideas were formulated and were tried to be implemented in the search, only three algorithms (KTV, KTV2 and KPrime) were designed. Of these, KTV and KTV2 showed good results where as the KPrime took lot more time than expected. Investigation on the algorithm revealed a further tweaking which is left as a part for the further study.

Table of Contents

List of Figures

List of Tables

Acknowledgement

I greatly appreciate the generous help and guidance provided by my advisor, Dr. William D Shoaff, throughout the development of this thesis.

I also thank the committee members Dr. David Carroll and Dr. Marius Silaghi for their constructive suggestions and insightful comments. Daniel Vaz and Hugues Treiber need a very special mention for their help during the implementation and testing of the KTV algorithm.

I also would like to thank Dr. Ryan Stansifer, Dr. Ronaldo Menezes, Dr. Cem Kaner, Madhan Thirukonda, Sujit Raghavan and Anusha Madhavan for their help at various stages of this thesis.

Dedication

To My parents, Chalam and Kumari, and to my sisters, Shyamala and Kiranmai, for their continued support and encouragement throughout my educational and professional career.

Chapter 1
Introduction

1.1 Purpose of study

The purpose of this study was to use a parallel computer to find effective ways of searching for occurrences of finite strings in a given long string. Several known algorithms were studied: Brute Force [9], Knuth Morris Pratt (KMP) [10], Kim's Fast Multiple Pattern Matching [12] and Aho Corasick [11] of which Brute Force, KMP were implemented. Real data along with simulated samples with sizes ranging from 40MB to 3GB were used as the input to the above algorithms and searchs were performed with patterns of different length ranging from 5 to 100. Processors used ranged from 1 to 40. The outcome of the study was to compare the results of these algorithms in terms of time taken to search for these patterns. In the course of this study I developed two new search techniques KTV and KPrime. These algorithms were designed and tested against the same set of data.

1.2 Background and rationale

String matching can be defined as *"The problem of finding occurrence(s) of a pattern string within another string or body of text"* [1] String matching is a subject of both theoretical and practical interest in

computer science. Theoretically, time and space complexity of the algorithms is of interest. On the practical side, searching for keywords, names, phrases etc., is common in web-based and bibliographic searches which exemplify the above definition. Searching for patterns in DNA has interesting scientific implications and is the ultimate use for this study.

With the advent of computers into the bio-informatics field, searching and string matching was no longer difficult. DNA sequences coded were ranging from somewhere below one mega byte (1,000,000 bases) to somewhere around 6 mega bytes (6,000,000 bases) and the storage constraints and the computing power were enough for those range of files. These DNA sequences belonged to the lower ranked living organisms (e.g. Salmonellae). Every good thing comes complimented with a problem of its own. As the study was extended to higher ranked organisms like mouse and *Homo sapiens*, the base count exceeded 3 Giga Bytes in size (3,000,000,000 bases). The first and foremost problem with this is the storage space. Secondly the memory and the CPU speed available to compute the search and other required operations. At this juncture the power of parallel computing has come to the rescue. This piece of work uses the power of parallel computer (Beowulf) to do the same with a large data on hand. This study is not to do the language specific (MPI specific) implementations but to make use of the storage and the computation efficiency and arrive at the

results. The basis for this study is an implementation proposed in a class project by Sowmya Padmanabhan using the message passing programming model [2]. Implementation of this kind is the first on the Beowulf cluster at Florida Tech.

1.3 Terminology

In this section we list and provide short definitions for major terms and ideas used in the thesis. In most cases these ideas will be explained upon in later chapters.

1.3.1 Computer Memory and Storage

A bit is a fundamental storage unit in computing. A bit records an on or off state; high or low voltage; or the more common interpretation of 1 or 0. From historical implementations, bits are often grouped in sets of 8 called as byte. Bytes are collected into a larger group, for example a kilobyte (2^{10} bytes) which in turn make up a Mega Byte (2^{20} bytes) or a Giga Byte (2^{30} bytes)

1.3.2 Deoxyribose Nucleic Acid

DNA is a nucleic acid that constitutes the genetic material of all cellular organisms. “The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G’s, A’s, T’s and C’s. This string is the root data structure of an organism’s biology” [3]. RNA is another nucleic acid that is part of the mechanism of

cellular organisms. The alphabet used in genetics for nucleic acid is presented in Table 1-1: The Alphabet used in Genetics for Nucleic Acids.

Code	**Nucleotides**
A	Adenine
C	Cytosine
G	Guanine
T	Thymine
U	Uracil
R	Purine (A or G)
Y	Pyrimidine (C or T/U)
M	A or C
W	A or T/U
S	C or G
K	G or T/U
D	A, G or T/U
H	A, C or T/U
V	A, C, or G
B	C, G, or T/U
N	A, C, G, or T/U No Base

Table 1-1: The Alphabet used in Genetics for Nucleic Acids

1.3.3 String Searching

String searching is the problem of finding occurrence(s) of a pattern within text. Precisely it can be termed as checking for equality

(contiguously) of all the characters in the pattern against the set of characters of equal length in a text string.

1.3.4 String Searching Algorithms

- Single Pattern
    - Brute Force is the naive algorithm used to find patterns. It continually checks to see if the pattern occurs at text positions 1, 2, 3…., until the end of the text is reached. (Refer 3.1 for algorithm)
    - KMP (Knuth-Morris-Pratt) is a string matching algorithm which uses a finite state machine generated from the pattern to avoid restarting the search from the next text position. It then runs the machine with the string to be searched as the input string (Refer 3.2 for algorithm)
    - KTV (Refer 3.3 for algorithm)
- Multiple Pattern
    - Kim's Fast Multiple String-Pattern Matching is a simple and efficient multiple string matching algorithm based on a compact encoding scheme[12]
    - Aho Corasick is a simple, efficient algorithm to locate all occurrences of any of a finite number of keywords in a string of text [11]

- KTV2 (Refer 4.3 for algorithm)
- KPrime (Refer 4.4 for algorithm)

1.3.5 Parallel Computing

- Parallel Computer can be termed as a computer that has multiple arithmetic units or logic units that are used to accomplish parallel operations or parallel processing [7]
- Parallel processing pertains to the concurrent or simultaneous execution of two or more processes in a single unit [8]
- Beowulf is a high-performance, massive parallel computer that performs similarly to a supercomputer for a fraction of the price. The computer is made up of a cluster of nodes connected by a high-speed network that perform intense computing tasks. The system is connected to the external world through a single head node [4]
- Bluemarlin, Florida Tech's Beowulf cluster is a distributed memory supercomputer cluster that is in the MIMD (Multiple Instruction Multiple Data) paradigm. It contains 47 compute nodes and one head node running Red Hat Linux 6.2 [4]

1.4 Research question

What search algorithm is advisable in a parallel environment while searching for patterns in a DNA?

1.5 Study design

The algorithms were written in C using the MPI model on the Beowulf cluster of Florida Tech (Bluemarlin). The timings are taken from the output file and compared against each other. The data set used is kept uniform throughout the experiments with different algorithms.

1.6 Significance of study

The study will find efficient parallel way to search for patterns, the program being operated on a parallel computer. This study will help us to achieve more work in the given time. Also this study is aimed at providing basis for a future study and help in selecting the search algorithms for real data search.

1.7 Study limitations

- The data being used for search is the DNA data of *Homo sapiens*'s chromosomes.
- Complete MPI based implementation (data scattering, gathering) is avoided to remove the factor of extra time consuming processes
- The study is not a comparison of string search algorithms but rather the outcome of application of these algorithms BruteForce, KMP, KTV, KPrime and KTV2 in the case of DNA alphabet.

Chapter 2
Background Study

2.1 Parallel Computing

"With ability of being able to perform a few hundred arithmetic computations each second, clamor for more power increased which currently reached about quadrillions of operations per second (10^5). The basic paradigm of observe, theorize and test through experiment is being followed very closely. Simulations help us avoid building high costing prototypes. Apart from this another closely attached problem is that of the development of vastly greater storage requirements. Hence greater computational power is in fact a combination of both greater speed and storage". [21]

"Consider an analogous problem described in Peter, a Roman contractor, has laborers capable of excavating 1000 cubic feet a day. He could solve the problem of having to excavate 100,000 cubic feet a day by having 100 men on his workforce. Comparing the same with the processors and memory, we should obtain more processors and memory to solve the computation problem. Buying more processors will not solve this since there should be a way to have them communicate amongst themselves. If the job of building the physical machine and the design of the software is avoided in the discussion, we are left with the communication problem".

2.2 Programming Models

Data Parallelism is a way of applying same operation to multiple elements of a data structure. A data-parallel program is made up of sequences of such operations. Each operation on data element can be thought of as an independent task, computation is small, and the concept of locality does not arise. Due to this, data-parallel compilers often require information about how data are to be distributed over processors and how data is to be partitioned into tasks. The compiler translates this data-parallel program into an SPMD formulation, and generates communication code. [2]

Message passing is most widely used parallel programming model. Message-passing programs create multiple tasks with each task encapsulating local data which are identified by a unique name. Tasks interact by sending and receiving messages to and from named tasks. Message passing differ from the task/channel model only in the mechanism used for data transfer. This model avoids dynamic creation of tasks, execution of multiple tasks per processor, or the execution of different programs by different tasks. In practice most message-passing systems create a fixed number of identical tasks at program startup and do not allow tasks to be created or destroyed during program execution. These systems are said to implement a single program multiple data (SPMD) programming model because each task executes the same program but operates on different data. SPMD model is

sufficient for a wide range of parallel programming problems but does hinder some parallel algorithm developments. [2]

In a *shared-memory* model, tasks share a common address space, which they read and write asynchronously by using locks and semaphores control access to the shared memory. An advantage of this model from the programmer's point of view is that there is no need to specify explicitly the communication of data from producers to consumers. This model can simplify program development. However, understanding and managing locality hinders the most shared-memory architectures. It is also more difficult to write deterministic programs [2]

2.3 Beowulf

The first Beowulf was built with DX4 processors and 10Mbit/s Ethernet. The processors were too fast for a single Ethernet and Ethernet switches were still too expensive. To balance the system Don Becker rewrote his Ethernet drivers for Linux and built a "channel bonded" Ethernet where the network traffic was striped across two or more Ethernets. As 100Mbit/s Ethernet and 100Mbit/s Ethernet switches have become cost effective, the need for channel bonding has diminished (at least for now). In late 1997, a good choice for a balance system was 16, 200MHz P6 processors connected by Fast Ethernet and a Fast Ethernet switch. The exact network configuration of a balanced cluster will continue to change and will remain dependent on

the size of the cluster and the relationship between processor speed and network bandwidth and the current price list for each of components. An important characteristic of Beowulf clusters is that these sorts of changes---processors type and speed, network technology, relative costs of components---do not change the programming model. Therefore, users of these systems can expect to enjoy more forward compatibility then we have experienced in the past.

A Beowulf class cluster computer is distinguished from a Network of Workstations by several subtle but significant characteristics. First, the nodes in the cluster are dedicated to the cluster. This helps ease load balancing problems, because the performance of individual nodes is not subject to external factors. Also, since the interconnection network is isolated from the external network, the network load is determined only by the application being run on the cluster. [25]

Florida Tech's Beowulf cluster is a supercomputer cluster that is in the MIMD paradigm. It contains 47 compute nodes and one head node. Head node has Red Hat Linux 6.2. Each node has Pentium III 866 MHz processor and 512Mb of RAM which are interconnected and channel bonded. The portable batch system (PBS) has been implemented to ensure priority queuing system. Argonne's version of mpich [23] is installed for MPI [24] usage [22]

2.4 Search Algorithms

Following algorithms were implemented and tested -

- Brute Force [Refer 3.1]
- KMP [Refer 3.2]
- KTV [Refer 3.3]
- KPrime [Refer 4.4]
- KTV2 [Refer 4.3]

Following algorithms were studied -

- Kim's Multiple String-Pattern Matching Algorithm [Refer 4.1]
- Aho & Corasick Algorithm [Refer 4.2]
- Baker's Boyer Moore-type Algorithm [37]
- Sunday's Substring Search Algorithm [38]

2.5 DNA

Deoxyribose Nucleic Acid is a polymer. The monomer units of DNA are nucleotides, and the polymer is known as a polynucleotide. Each nucleotide consists of a 5-carbon sugar (Deoxyribose), a nitrogen containing base attached to the sugar, and a phosphate group. There are four different types of nucleotides found in DNA, differing only in the nitrogenous base. The four nucleotides are given one letter abbreviations as shorthand for the four bases.

A (adenine), G (guanine), C (cytosine) and T (thymine). Adenine and guanine are purines. Purines are the larger of the two types of bases found in DNA. Structures are shown below:

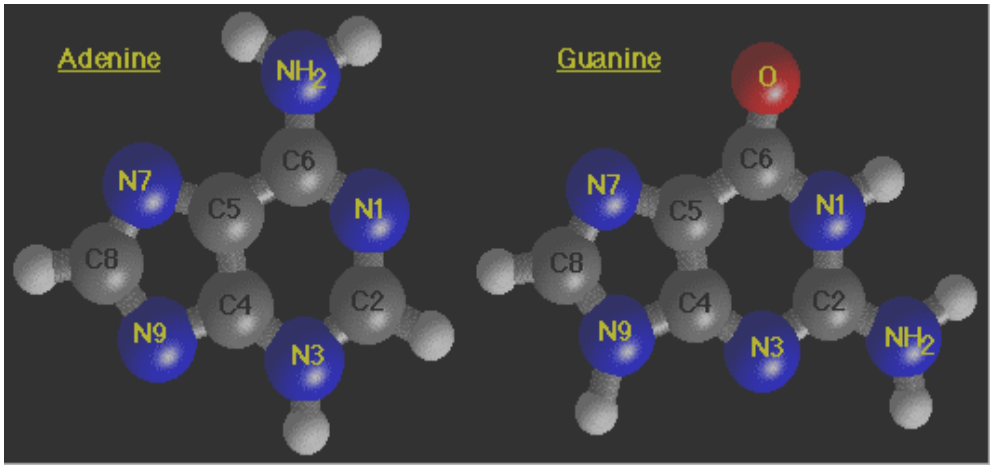

Figure 2-1 : Adenine and Guanine Structures [source: unknown]

The 9 atoms that make up the fused rings (5 Carbon, 4 Nitrogen) are numbered 1-9. All ring atoms lie in the same plane. Cytosine and thymine are pyrimidines. The 6 stoms (4 carbon, 2 nitrogen) are numbered 1-6. Like purines, all pyrimidine ring atoms lie in the same plane.

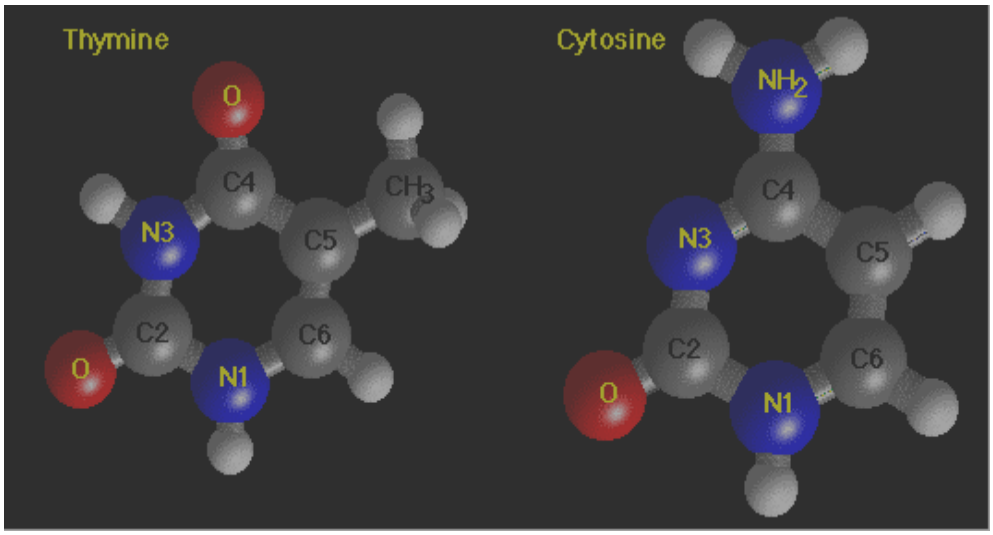

Figure 2-2 : Thymine and Cytosine Structures [source: unknown]

2.6 GenBank

GenBank® is the NIH (National Institutes of Health) genetic sequence database, an annotated collection of all publicly available DNA sequences (Nucleic Acids Research 2002 Jan 1; 30(1):17-20). There are approximately 22,617,000,000 bases in 18,197,000 sequence records as of August 2002. As an example, you may view the record for a *Saccharomyces cerevisiae* gene. GenBank is part of the International Nucleotide Sequence Database Collaboration, which comprises the DNA DataBank of Japan (DDBJ), the European Molecular Biology Laboratory (EMBL), and GenBank at NCBI. These three organizations exchange data on a daily basis.

Chapter 3
Single Pattern Algorithms

3.1 Brute Force Algorithm

3.1.1 Description

In this naive method, we align the left end of P [pattern] with the left end of T [text] and then compare the characters of P and T left to right until either two unequal characters are found or until P is exhausted, in which case an occurrence of pattern is noted. Using n to denote the length of pattern and m to denote the length of text, at most n(m-n+1) number of comparisons are made. [18]. This matching procedure can be interpreted graphically as a sliding "template" containing the pattern over the text, noting for which shifts all of the characters on the template equal the corresponding characters in text. [27]

3.1.2 Algorithm Listing

Compare the size n pattern with the first m characters of the text.
REPEAT
If a match is found, note the shift position and increment the count.
Move the pattern right one position.
Compare the size n pattern with the next m chars of the string.
UNTIL there are no more characters to compare

3.1.3 Algorithm Implementation

For a complete program listing used for the testing please refer to Appendix B

```
long int BruteForce(char *text, char *pattern)
{

   int i, j, k;
   long int count = 0, occurences = 0;
   int first = 0;

   const long int length_of_pattern = strlen(pattern);
   const long int length_of_text = strlen(text);
   const long int limit = length_of_text - length_of_pattern;

   for (i = 0; i < limit; i++ )
   {
      count = 0;

      for(j = i, k = 0; k < (length_of_pattern) ; j++, k++)
      {
         if(*(text + j) != *(pattern + k) )
         {
            break;
         }
         else
         {
            count++;
         }

         if(count == length_of_pattern )
         {
            occurences++;
         }

      }
   }

return occurences;
}
```

3.2 Knuth Morris Pratt Algorithm

3.2.1 Description

KMP is a string matching algorithm which turns the search string into a finite state machine, then runs the machine with the string to be searched as the input string [10]. The design of the Knuth-Morris-Pratt algorithm follows a tight analysis of the Morris-Pratt algorithm. The major problem with the brute-force search is that characters in the text may be re-examined multiple times and this can lead to poor performance in some cases. The algorithm of Knuth, Morris and Pratt provides a way to alleviate the repeated accesses to the text and, as a result, it gives us a guaranteed linear time searching algorithm [26]. The key aspect of the Knuth-Morris-Pratt(KMP) algorithm is that a failed attempt to find a match yields useful information to be used on the next attempt. Specifically, if a mismatch is detected when considering the characters *pat*[*j*] and *text*[*k*], we do not need to start the next attempt at *text*[*k-j*+1] as we know the characters text[k–j], text[k-j+1] … text[k-1] are identical to the prefix of the pattern, pat[0], pat[1]…pat[j-1]. By using this information we can access the text characters sequentially and alleviate the need to *back-up* the text. The KMP algorithm is essentially the brute-force algorithm with a more intelligent re-initialisation of pointers when a mismatch is detected. In most practical situations the running time for KMP is not much better than for brute-force,

however KMP *guarantees* a linear bound and it is well suited to extensions for more difficult problems.

3.2.2 Algorithm Listing

```
KMP (T,P)
n ← Length[T]
m ← Length[P]
?  ← Compute-Prefix-Funtion(P)
q ← 0
for i ← 1 to n
  do while q > 0 and P[q+1] ? T[i]
           do q ← ? [q]
     if P[q+1] = T[i]
           then q ← q + 1
     if q = m
       then print "Pattern occurs with shift" I – m
           q ← ? [q]

Compute-Prefix-Function(P)
M ← length[P]
? [1] ← 0
k ← 0
for q ← 2 to m
  do while k > 0 and P[k+1] ? P[q]
           do k ← ? [k]
    if P[k+1] = P[q]
           then k ← k+1
  ? [q] ← k
return ?
```

3.2.3 Algorithm Implementation

For a complete program listing used for the testing please refer to Appendix B

```
void preKmp(char *string, int string_len, int kmpNext[])
{
   int i, j;
   i = 0;
   j = kmpNext[0] = -1;
   while (i < search_string_len)
```

```
    {
        while (j > -1 && string[i] != string[j])
            j = kmpNext[j];
        i++;
        j++;
        if (string[i] == string[j])
            kmpNext[i] = kmpNext[j];
        else
            kmpNext[i] = j;
    }
}

long int KMP(char *string, int string_len, char *text, int text_len)
{
     int i, j, k, kmpNext[SIZE];
    preKmp(string, string_len, kmpNext);
    i = j = k = 0;
    while (j < string_len)
    {
        while (i > -1 && string[i] != text[j])
            i = kmpNext[i];
        i++;
        j++;
        if (i >= string_len)
        {
            k++;
            i = kmpNext[i];
        }
    }
return k;
}
```

3.3 Kishore Treiber Vaz (KTV) Algorithm

3.3.1 Description

This algorithm has been designed for speed in searching a pattern. The efficiency of this algorithm can be seen when the starting character of the pattern being searched does not exist in the text. The important part of the algorithm is the preprocessing phase. Following explanations of the

algorithms use an example with pattern being searched as ‘ggg’ and the text in which it is being searched is ‘agcggg’.

a) Pre processing part :

This is the main part where the complete text is scanned and the KTV structure is created. A KTV structure is made up of two pointers. The first one is a character pointer and the next one is a KTV structure pointer. The algorithm also takes help of two other character arrays called as ‘Position’ and ‘Alpha’. ‘Alpha’ is initialized with the 26 alphabets. All the elements of the ‘Position’ array are initialized to NULL. The first occurrence of characters in the ‘Alpha’ is stored in ‘Position’ with first position (index 0) being ‘a’ and last position being ‘z’ (index 25). An array of KTV structures is declared whose size is equal to the size of the text. For each character in the text the character pointer of the structure array is initialized so as that it points to the correct character in the ‘Alpha’ array. If the index of the character being processed in the ‘Position’ is Null then the KTV pointer at index is updated in such a way that it points to this structure. Also at the same time using a temporary pointer array the position of this character is stored. When another character same as this is encountered then, using the temporary pointer array the previous structure’s KTV pointer is updated so that it points to this new character. This process

is followed until all the characters of the text are processed. Following is the illustration with respect to the above example.

- Initial Stage:

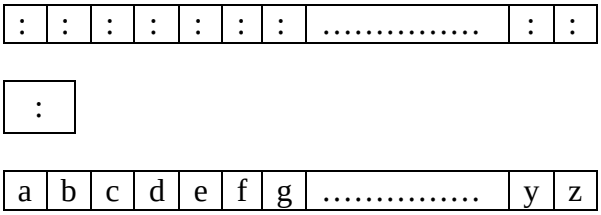

Figure 3-1 : Position, KTV Structure and Alpha arrays

- After first character is processed

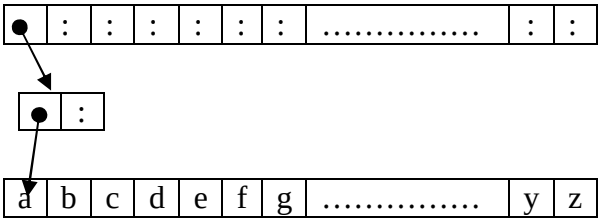

Figure 3-2 : Position, KTV Structure and Alpha arrays

- After all characters are processed

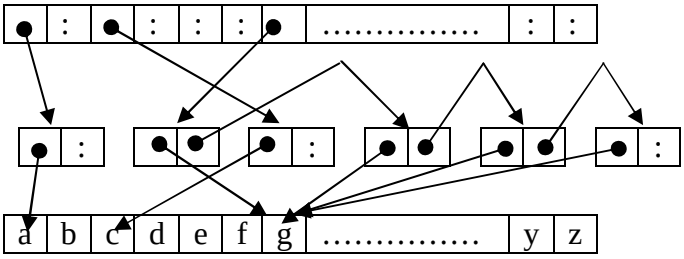

Figure 3-3 : Position, KTV Structure and Alpha arrays

b) Searching :

Once the whole text is processed and the KTV structure array is created the first character of the search string is considered. Using the 'Position' array, the search is started from the position which is

indicated by the pointer. From this point onwards the rest of the characters of the pattern are compared with the consecutive character pointers of the KTV structure. If all the characters of the pattern are completed then the occurrences counter is incremented by one. In the case of failure of the comparison, using the ‘Next’ pointer of the KTV structure where the previous search started, the current position is updated and the search is started again. Using the example above we have the following. First attempt starts at position two. The search fails since the next character is not equal to ‘g’.

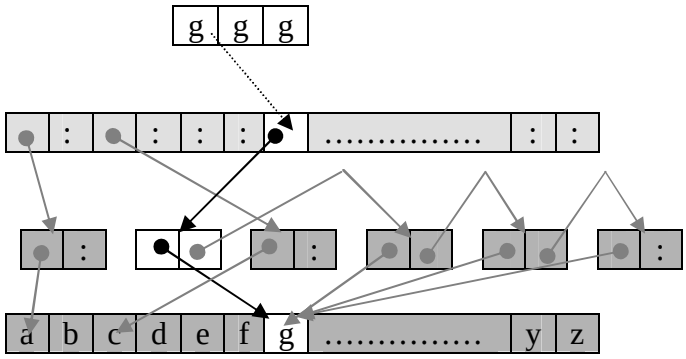

Figure 3-4 : Illustrating search for 'ggg' - failure

Using the second position’s KTV structure pointer the search is started again at position four. This search will result in occurrences to be incremented by one since the next two characters are equal to the consecutive characters in the KTV structure.

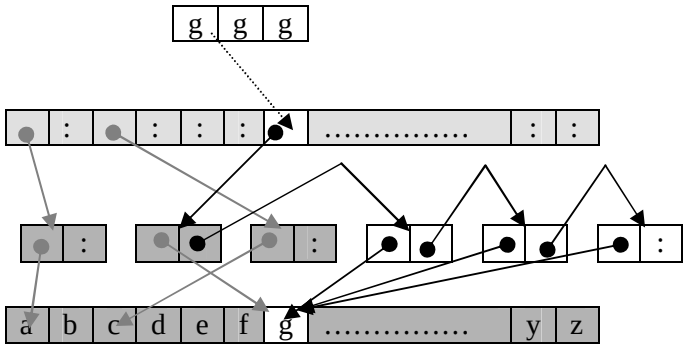

Figure 3-5 : Illustrating search for 'ggg' - success

3.3.2 Algorithm Implementation

For a complete program listing used for the testing please refer to Appendix B

```
struct aa
{    char *c;
   struct aa *next; };

struct aa *p;
struct aa positions[26];
struct aa *prev[26];
char *alpha[26]={"a","b","c","d",NULL,NULL,"g","h",NULL,
                  NULL, "k",NULL,"m","n",NULL,NULL,NULL,
                 "r","s","t", "u","v","w",NULL,"y",NULL};
long int totaloccurances;

void preKTV(char *data, long int length)
{
   long int i;
   int pos;

   p = (struct aa *) malloc (sizeof(struct aa)*length);
   for(i = 0;i<length;i++)
   {
        pos = data[i] - 97;
       if(pos<0 || pos>26)
            break;
        p[i].c = alpha[pos];
        p[i].next = NULL;
        if(positions[pos].c != NULL)
            prev[pos]->next = &p[i];
        else
```

```
            {
                positions[pos].c = alpha[pos];
               positions[pos].next = &p[i];
            }
            prev[pos] = &p[i];
    }
    p[i].c=NULL;
    p[i].next = NULL;
}
long int KTV(char *data, char *pattern)
{
     struct aa *current;
     long int occurances=0;
     long int index;
     int plength = strlen(pattern);

     preKTV(data,strlen(data));
    current = positions[pattern[0]-97].next;
    while(current != NULL && (current+plength-1)->c !=NULL)
    {
       for(index = 1; index<plength;index++)
      {
            if(pattern[index] != *(current+index)->c) break;
      }
      if (index == plength) occurances++;
            current = current->next;
    }
   free(p);
  return occurances;
}
```

Chapter 4
Multiple Pattern Algorithms

4.1 Kim's Multiple String-Pattern Matching Algorithm

4.1.1 Description

This algorithm scans text from left to right while encoding characters in the text based on the alphabet that occurs in the input patterns. The simple scanning algorithm demonstrates the ability to handle a very large number of input patterns simultaneously. The compact encoding scheme can be summarized as follows:

1) Scan input pattern P and determine how many bits E are needed for compact encoding
2) Define the encoding function ENCODE for each symbol in P and any symbol that does not occur in P.
3) Encode each symbol in P and T by function ENCODE.

The multiple string pattern matching algorithm was summarized as below.

1) Scan the input patterns to determine the number in bits, E, for each character encoding and define the ENCODE function.
2) Encode each pattern P_i and set the associated mask $PMASK_i$
3) Set the hash mask HMASK according to the hash table size.
4) Initialize the text scanning variable T to 0.

5) While scanning the text character by shifting T E bits left, perform the pattern testing procedure for all patterns at the hash entry position computed by logically ANDing T and H. If the hash entry at the position is empty, skip the pattern testing procedure and scan the next text character.

4.1.2 Partial Code Listing

```
struct hash_entry
{
        PAT                 P;
        PATMASK             pmask;
        struct hash_entry   * next;
};
PAT   HMASK;

for(i = 1; i <= n; i++)
    insert_pattern_into_hash_table(P[i]);

T = encode_ncharacters(text, S);
i = S +1;

while ( i <= Tlen)
{
    if (HTBL[T&HMASK] != NULL)
    {
    candidate = HTBL[T&HMASK];
        while(candidate)
        {
            if (((T & candidate -> pmask) ^ candidate -> p) = = 0)
                report_pattern_match(candidate);

            candidate = candidate -> next;
    }
}
T = T << E | ENCODE(text[i]);
i ++;
}
```

4.1.3 Algorithm Limitations

The algorithm relies on hashing techniques, was made it essential to reduce the number of collisions in the hash entries to speed up the pattern searching. The test cases discussed in the paper are based on text sizes less than 13 MB in size [for English pattern searching] and 19MB [for DNA] with patterns of lengths 3 and up until 20000. There are no test results discussed in the paper for sizes other than those indicated above. The Adaptive string matching technique as discussed in the paper when implemented will increase the runtime complexity as the function will take a bit more time to adapt to the character set. This algorithm was used as reference in understanding the logic behind applying string searching techniques to multiple patterns. The encoding technique was also considered for the algorithm KPrime and KTV2 but due to time complexity the ideas was dropped.

4.2 Aho & Corasick Algorithm

4.2.1 Description

The algorithm consists of constructing a finite state pattern matching machine from the keywords and then using the pattern matching machine to process the text string in a single pass. The finite state pattern matching algorithm was used in a library bibliographic search program. The purpose of the program was to enable a bibliographer to find in a

citation index all titles satisfying some Boolean function of keywords and phrases. This algorithm resulted in running times which are fifth to a tenth of the original straightforward string matching algorithm. Construction of the pattern matching machine takes time proportional to the sum of the lengths of the patterns. The number of state transitions made by the pattern matching machine in processing the text string is independent of the number of patterns. The algorithm is divided into three parts as the pattern matching machine, goto function and failure function.

4.2.2 Algorithm Listing

- **Algorithm 1: Pattern Matching Machine**

```
Input.
   A text string x = a I a 2 - - • a n where each a i is an
   input symbol and a pattern matching machine M with goto
   function g, failure function f, and output function output, as
   described above.
Output.
   Locations at which keywords occur in x.
Method.
begin
   state ~ 0
   for i ~ 1 until n do
   begin
            while g (state, a i ) = fail do state ~ f ( s t a t e )
            state ~ g (state, a i )
            if output (state) ;~ empty then
            begin
                     print i
                     print output (state)
            end
   end
end
```

- **Algorithm 2: Goto Function**

Input.
Set of keywords K = {Yl, Y2 . . . . . Yk}.
Output.
Goto function g and a partially computed output function output.
Method.
We assume output(s) is empty when state s is first created, and g (s , a) = f a i l if a is undefined or if g(s, a) has not yet been defined. The procedure enter(y) inserts into the goto graph a path that spells out y.

```
begin
  newstate ← 0
  for i ← 1 until k do enter(y i )
  for all asuch that g(O, a) = fail  do g(O, a) ← 0
end

procedure enter(a 1 a 2 • • • a m ):
begin
  state ← 0; j ← 1
  while g (state, aj ) ?  fail do
  begin
          state ← g (state, a) )
          j ← j + l
  end
  for p ←  j until m do
  begin
          newstate ← newstate + 1
          g (state, ap ) ← newstate
          state ←  newstate
  end
  output(state) ← { a I a 2 . . . a m}
end
```

- **Algorithm 3: Failure Function**

Input.
Goto function g and output function output from Algo 2.

```
Output.
  Failure function fand output function output.
Method.
begin
  queue ← empty
  for each a such that g(O, a ) = s ← 0 do
  begin
        queue ← queue LI {s }
        f ( s )← 0
  end
  while queue ← empty do
  begin
        let r be the next state in queue
        queue ← queue - {r}
        for each a such that g(r, a ) = s ? fail do
        begin
              queue ← queue t2 {s }
              state ← f ( r )
              while g (state, a)= fail do state ← f (state)
              f ( s ) ← g(state, a)
             output(s) ← output(s) U output(f(s))
        end
  end
end
```

4.2.3 Algorithm Limitations

This pattern matching scheme is well suited for applications in which we are looking for a large numbers of keywords in text strings. According to [28] the text used for testing is of 10^7 size which is equal to roughly 10MB. For this size of text the algorithm took 0.18 hrs (roughly 11mins) for 15 keywords and 0.21 hrs (roughly 13mins) which are quite longer times than anticipated. The main construction of the pattern matching machine is the part where there is more consumption of time is involved.

4.3 KTV2 Algorithm

4.3.1 Description

This is the modified version of the algorithm described in 3.3. This algorithm takes advantage of the already created KTV structure to search for all the patterns instead of re-creating the whole structure. The search is the same but is repeated with all the different patterns and the results are stored in an array.

4.3.2 Algorithm implementation

```
struct aa *p;
struct aa positions[26];
struct aa *prev[26];
char *alpha[26]={"a","b","c","d",NULL,NULL,"g","h",NULL,
                 NULL, "k",NULL,"m","n",NULL,NULL,NULL,
                 "r","s","t","u","v","w",NULL,"y",NULL};
long int *totaloccurances;

void preKTV(char *data, long int length)
{
   long int i;
   int pos;

   p = (struct aa *)malloc(sizeof(struct aa)*length);
   for(i = 0;i<length;i++)
  {
      pos = data[i] - 97;
      if(pos<0 || pos>26)
     {
         break;
     }
     p[i].c = alpha[pos];
     p[i].next = NULL;
     if(positions[pos].c != NULL)
     prev[pos]->next = &p[i];
     else
```

```
    {
        positions[pos].c = alpha[pos];
        positions[pos].next = &p[i];
     }
     prev[pos] = &p[i];
   }
   p[i].c=NULL;
   p[i].next = NULL;
}

long int * KTV(char *data)
{
   struct aa *current;
   long int *occurances;
   long int index;
   int plength;
   int i;
   preKTV(data,strlen(data));
   occurances =(long int *)malloc(sizeof(long int)*pCount);
   for(i=0;i<pCount;i++)
   {
        occurances[i]=0;
        plength = strlen(patterns[i]);
        current = positions[patterns[i][0]-97].next;
        while(current !=NULL &&(current+plength-1)->c!=NULL)
        {
                for(index = 1; index<plength;index++)
                {
                        if(patterns[i][index] != *(current+index)->c)
                                break;
                }
                if (index == plength)
                        occurances[i]=occurances[i]+1;
                current = current->next;
        }
   }
   free(p);
   return occurances;
}
```

4.4 KPrime Algorithm

4.4.1 Description

This algorithm has been designed to handle multiple patterns and search for their occurrences in the text by using the concept of prime numbers. The algorithm has both preprocessing and actual search phases. They are explained below with an example where patterns being searched are ‘gc’, ‘aa’, ‘a’ & ‘gcgg’ and the text in which it is being searched is ‘gcggga’.

a) Pre processing part :

This is the main part where all the patterns are read into the patterns array called as ‘Patterns’. Using the pre assigned primes array called as ‘Primes’, the respective values are stored in the values array called as ‘pValue’. The algorithm also takes help of the array ‘pRowOccurances’ which is of the size 26 by length of the largest pattern. This array is used as a Boolean array to check whether the character indicated by the index occurs at the first position or not. Also the patterns are checked to get the maximum length and stored in ‘pLength’. Using a neutral character, a character which we know for sure doesn’t exist in the text [in the test case I used character “ ` ”], pad the rest of the patterns so that every one of them are of the same size. This character’s prime value is 1. The Occurrences array which is being used to store the

number of occurrences is also initialized with zeros. Every element of mValue array is also initialized to 1. This array is used to store the multiplied values of the values of the patterns.

g	c	`	`
a	a	`	`
a	`	`	`
g	c	g	g

Figure 4-1 : Patterns array

17	5	1	1
2	2	1	1
17	1	1	1
17	5	17	17

Figure 4-2 : pValue array

0	1	1	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
1	0	0	1
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

Figure 4-3 : pRowOccurances array

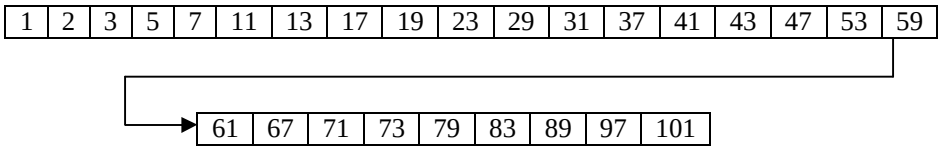

Figure 4-4 : Primes array

1	1	1	1

Figure 4-5 : mValue array

b) Searching:

The first step of searching is to call the pre processing method. The main text is also padded using the character of the pre processing phase so that the text is a multiple of the pLength. The search takes place as follows. Using the values in the pValue array, the product of all the values in the same column positions is stored in mValue array. The multiplication is done in such a way that multiple values are avoided. To illustrate the mValue array after the initial multiplication:

17	5	1	1
2	2	1	1
17	1	1	1
17	5	17	17

Figure 4-6 : Active patterns in the search

34	10	17	17

Figure 4-7 : mValue array

At first a character is read from the text. The value of the character from the primes array is used to divide the first value in the mValue

array. If the remainder of the operation is zero then the character is existing in any of the patterns at the first position. Prime numbers are those which are divisible by one or by itself. This algorithm uses this property to check the existence of the character in the patterns. Once the reminder is zero, the multiply function is called but this time with only those values of the patterns whose first character is equal to the character from the text. This process is followed with the rest of the characters of the text until pLength is reached. After that comparison, the patterns that are still a valid candidate for search are the strings that are equal to the characters of the text read or a substring of the text. Hence occurrences of these patterns are updated by one. The search is restarted again from the second character onwards. This procedure is followed until the end of text is reached. Following is the illustration of search starting at position one.

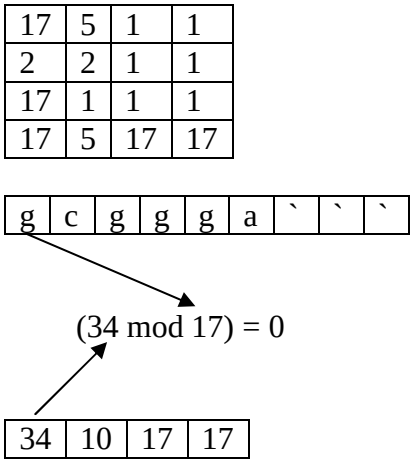

Figure 4-8 : Illustration of Search

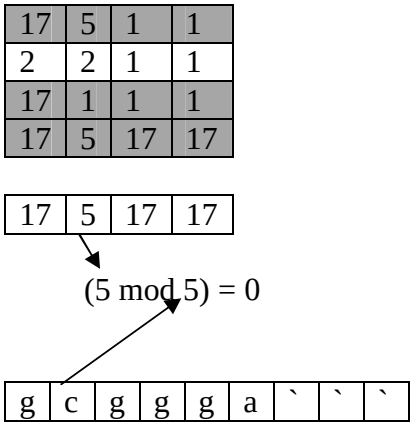

Figure 4-9 : Illustration of Search

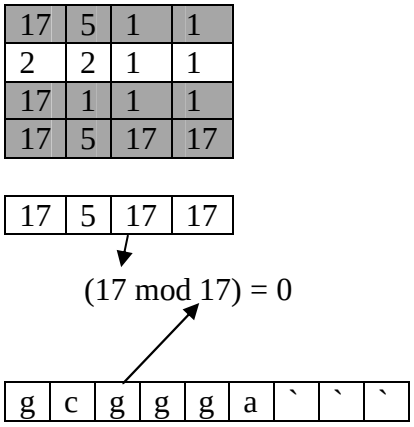

Figure 4-10 : Illustration of Search

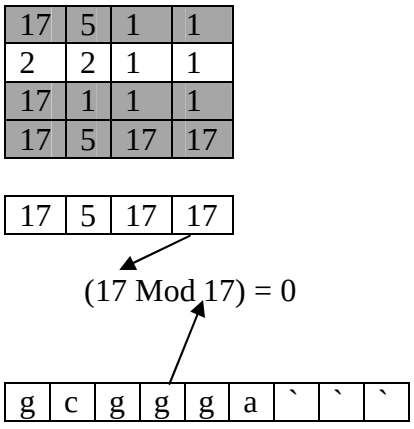

Figure 4-11 : Illustration of Search

17	5	1	1
2	2	1	1
17	1	1	1
17	5	17	17

1	0	1	1

Figure 4-12 : Patterns, Occurrences at end of Search

As it can be observed the characters read from the text were ‘gcgg’. The patterns found at the end of the first cycle of search are ‘gc’, ‘g’ & ‘gcgg’. The search is stopped when sizeof(text -1) is reached. This position doesn’t give error with regards to memory since we already padded the text with the neutral character. At the end of the complete search for the example in discussion the resultant occurrences array is {1, 0, 4, 1}.

4.4.2 Algorithm Implementation

```
int primes[]=
            {1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,
              47,53,59,61,67,71,73,79,83,89,97,101};
int pCount;
int pLength;
long int **pValue;
int **pRowOccurances;
long int *mValue;
int *tmpValue;
char **patterns;

long int* preKPrime()
{
   int i=0;
   int j=0;
   long int *Occurances;
```

```
    pValue =(long int **)malloc(sizeof(long int *)*pCount);
    pRowOccurances = (int **)malloc(sizeof(int *)*27);
    Occurances =
            (long int *)malloc(sizeof(long int *)*pCount);
    for(i=0;i<27;i++)
    {
       pRowOccurances[i] = (int *)malloc(sizeof(int)*pLength);
       Occurances[i]=0;
       for(j=0;j<pCount;j++)
          pRowOccurances[i][j]=0;
    }
    mValue =(long int *)malloc(sizeof(long int)*pLength);
    for(i=0;i<pCount;i++)
    {
       pValue[i] = (long int *)malloc(sizeof(long int)*pLength);
       pRowOccurances[patterns[i][0]-96][i]=1;
       for(j=0;j<pLength;j++)
       {
          pValue[i][j]=primes[patterns[i][j]-96];
         mValue[j] = 1;
       }
    }
    return Occurances;
}

long int* KPrime(char *str,int mode)
{
   char *mainString;
    long int i=0, j=0, k=0, tmp1=0, adjustment=0;
    long int count=0;
    long int Value,Start;
   long int stringlen=0;
    long int *occurances;

    occurances = preKPrime();
    tmpValue = (int *)malloc(sizeof(int)*pCount);
    stringlen= strlen(str);

    if (mode == 0)
    {
       mainString = (char *)malloc(sizeof(char)*(stringlen));
```

```
   strcpy(mainString,str);
 }
 else
 {
    mainString =
          (char *)malloc(sizeof(char)*(stringlen+pLength-1));
   strcpy(mainString,str);
    for(i=0;i<(pLength-1);i++)
         strcat(mainString,"`");
 }
stringlen= (long int)strlen(mainString);
while((k+(pLength)-1) < stringlen)
{
    for(j=0;j<pCount;j++)

         tmpValue[j]=
                    pRowOccurances[mainString[k]-96][j];

         for(j=0;j<pLength;j++)
              mValue[j]=1;
          multiplyValues();

          for(i=0,count=0;i<pLength && count < 2;i++)
                count += (tmpValue[i] == 1) ? 1 : 0;
         switch(count)
         {
              case 1:
                   for(i=0;i<pLength;i++)
                   if (mValue[i] != 1)
                   if ((mValue[i] %
                        primes[mainString[k+i]-96]) != 0)
                      break;
                   if (i == pLength)
                      for(i=0;i<pCount;i++)
                        if (tmpValue[i]==1)
                        {
                            occurances[i] += 1;
                            break;
                        }
              break;

              case 0:
```

```
                    break;

                    default:
                        for(i=0;i<pLength-1;i++)
                        {
                            if (mValue[i] != 1)
                               if ((mValue[i] %
                                  primes[mainString[k+i]-96]) != 0)
                                  break;
                              Value = mainString[k+i+1]-96;
                              Start = i+1;
                              for(j=0;j<pCount;j++)
                                    tmpValue[j] =
                                    (tmpValue[j] == 0)?
                                     0 :(((pValue[j][Start] ==
                                    primes[Value])|| pValue[j][Start] ==
                                    1 )? 1 : 0);
                             multiplyValues();
                        }
                        if (i == pLength-1)
                          for(i=0;i<pCount;i++)
                                if (tmpValue[i]==1)
                                    occurances[i] += 1;
                                break;
                }
                k++;
  }
 return occurances;
}
```

Chapter 5
Experiments and Results

5.1 Hardware

The experiments were conducted on a parallel computer named Beowulf. The idea was to see that all the test cases to get almost the same set of resources each time. The different nodes were used to act as single computer. Settings for each node:

- x330 series with 1 PIII 866 MHz processor
- 512 MB SDRAM RDIMM2 of which 500MB was available
- 18.2 GB Ultra 160 HDD

The text files used as the text are copied to all the nodes. The programs were written in such a way that these files are read from the nodes in which the program is being executed. While executing the test cases the number of nodes used was in the range of 1 to 20 [in the case of single patterns] and 40 [in the case of multiple patterns].

5.2 Text

The text used for the experiments of the ASCII text version of the chromosome data of Human Genome. These are listed in the following table with sizes indicated in bytes.

Sl.No	Chromosome	Size (bytes)
1	Chromosome 01	245,203,898
2	Chromosome 02	243,315,028
3	Chromosome 03	199,411,731
4	Chromosome 04	191,610,523
5	Chromosome 05	180,967,295
6	Chromosome 06	170,740,541
7	Chromosome 07	158,431,299
8	Chromosome 08	145,908,738
9	Chromosome 09	134,505,819
10	Chromosome 10	135,480,874
11	Chromosome 11	134,978,784
12	Chromosome 12	133,464,434
13	Chromosome 13	114,151,656
14	Chromosome 14	105,311,216
15	Chromosome 15	100,114,055
16	Chromosome 16	89,995,999
17	Chromosome 17	81,691,216
18	Chromosome 18	77,753,510
19	Chromosome 19	63,790,860
20	Chromosome 20	63,644,868
21	Chromosome 21	46,976,537
22	Chromosome 22	49,476,972
23	Chromosome X	152,634,166
24	Chromosome Y	50,961,097

Table 5-1: Text Data for the experiments

By using different number of nodes various sizes were tested for the algorithms. For example using Chromosome1 file of size 245MB and 10 nodes, each node gets a text of size 24.5 MB and hence the results from all these nodes can be averaged for 24.5MB size, whereas the total can be used for checking for a 245Mb size. For the multiple patterns the nodes used were fixed at 40, which means that the maximum size tested for the algorithms is approximately 6MB (245,203,898 / 40).

5.3 Patterns

Patterns of different sizes were used in the testing of the algorithms. The patterns used in different searches are listed below:

a) Single Pattern Search

1. z
2. a
3. atattaggt
4. atattaggtatatta
5. ccatttattcacctgttatcaattacaggcattgtatttaaagatcagatgttttatattta tttcttcaaatttcattcatggtgccataagtgaaggt

b) Multiple Pattern Search

1. • a
   • atattaggt
   • atattaggtatatta
   • catttattcacctgttatcaattacaggcattgtatttaaagatcagatgttttata tttatttcttcaaatttcattcatggtgccataagtgaaggt

2. • atattaggtatattaatatt
   • ccatttattcacctgttatc
   • aattacaggcattgtattta
   • aagatcagatgttttatatt
   • tatttcttcaaatttcattc

3. All the patterns in 5.3 b 2
   - atggtgccataagtgaaggt
   - ccattgagtcgtagcttaat
   - ggtatatcactattactatt
   - cagtatattctagtcagtac
   - attatagcattatgattaga

4. All the patterns in 5.3 b 3
   - tttgtagtatagtgatgata
   - catgatcgtactgatcgtac
   - tagatagctagacatcgaat
   - aaataggagcagcgactaga
   - aggatcaggcagctagacta

5. All the patterns in 5.3 b 4
   - ggaggatcattcaggagcta
   - gagtattatgattaggtatg
   - ttatattgagacaggagaga
   - ccgcgattaggcccgaggat
   - ttttaggaggattggggata

6. - a
   - at
   - ata
   - atat
   - atatt
   - atatta
   - atattag
   - atattagg
   - atattaggt
   - atattaggta
   - atattaggtat
   - atattaggtata
   - atattaggtatat
   - atattaggtatatt
   - atattaggtatatta
   - atattaggtatattaa
   - atattaggtatattaat
   - atattaggtatattaata
   - atattaggtatattaatat
   - atattaggtatattaatatt

7. All patterns in 5.3 b 5 repeated twice
   Repeat the first 10 lines of pattern in 5.3 b 5

8. All patterns in 5.3 b 7 repeated twice.

9. All patterns in 5.3 b 7 repeated thrice.

5.4 Single Pattern Search

The test cases designed for single pattern search algorithms include a pattern which does not exist in the text. This we know for sure because the test case adopted ‘z’ in this case, is not part of the DNA alphabet. Other test cases were pseudorandom generated strings of the DNA alphabet. These strings were randomly taken from the DNA data of zebra fish and also from the actual text in which they are searched. The maximum size of the pattern which was tested was 100 characters in length. The other test cases include patterns in length of 1, 9 and 15 characters.

5.4.1 Results for Pattern 5.3 a 1

This test was done in order to check the efficiency built into the algorithms under test to check for a character that does not exist in the text being searched. As it can be seen from Figure 5-1 Brute Force and KMP were almost linearly increasing in time along with the size of the text whereas the KTV algorithm acted differently. It showed a little increase in time as compared to the other two algorithms. It almost averaged at 0.3 seconds. In the preprocessing stage as the Position array was being updated the corresponding character that did not exist was pointing to nothing.

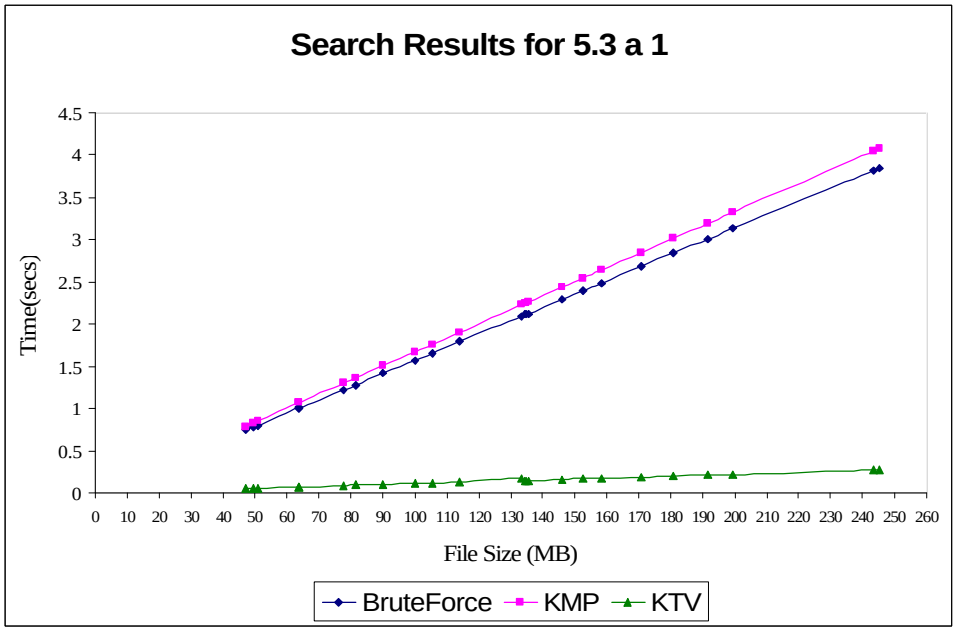

Figure 5-1 : Search Results for 5.3 a 1

S.No	Sample Size	Occurrences	Search Time in Seconds		
			BF	KMP	KTV
1	46,976,537	-	0.757481	0.788041	0.053924
2	49,476,972	-	0.785963	0.829355	0.058255
3	50,961,097	-	0.803733	0.857521	0.058848
4	63,644,868	-	1.015911	1.067487	0.071989
5	63,790,860	-	1.008220	1.070992	0.072092
6	77,753,510	-	1.225231	1.302336	0.086881
7	81,691,216	-	1.283535	1.365692	0.096492
8	89,995,999	-	1.418662	1.505741	0.099034
9	100,114,055	-	1.572478	1.674647	0.112859
10	105,311,216	-	1.655221	1.756888	0.119492
11	114,151,656	-	1.793934	1.905149	0.128979
12	133,464,434	-	2.095320	2.228942	0.178677
13	134,505,819	-	2.114270	2.245151	0.149774
14	134,978,784	-	2.124924	2.250289	0.151805
15	135,480,874	-	2.125514	2.262828	0.150584
16	145,908,738	-	2.288132	2.434612	0.164225
17	152,634,166	-	2.394981	2.547484	0.171306
18	158,431,299	-	2.487196	2.640976	0.176680
19	170,740,541	-	2.681511	2.850161	0.192104
20	180,967,295	-	2.838816	3.019265	0.203536
21	191,610,523	-	3.007935	3.198592	0.213308
22	199,411,731	-	3.129303	3.324192	0.220759
23	243,315,028	-	3.815656	4.054258	0.271210
24	245,203,898	-	3.845615	4.086006	0.272053

Table 5-2: Search Results for 5.3 a 1

5.4.2 Results for Pattern 5.3 a 2

This test uses a pattern of one character length which is one of the DNA alphabet that exists in the text we are searching. This test case actually revealed that the algorithms were dependent not only on the size but also number of occurrences. But as it can be seen from Figure 5-2 KTV does perform well with average time way below the other two. The dips in the results are due to less occurrences of the pattern being searched. The sudden dip in the time when the size reaches 245MB is due to the fact that the corresponding text was containing unknown value for most part of it. This observation holds for all the test cases that follow.

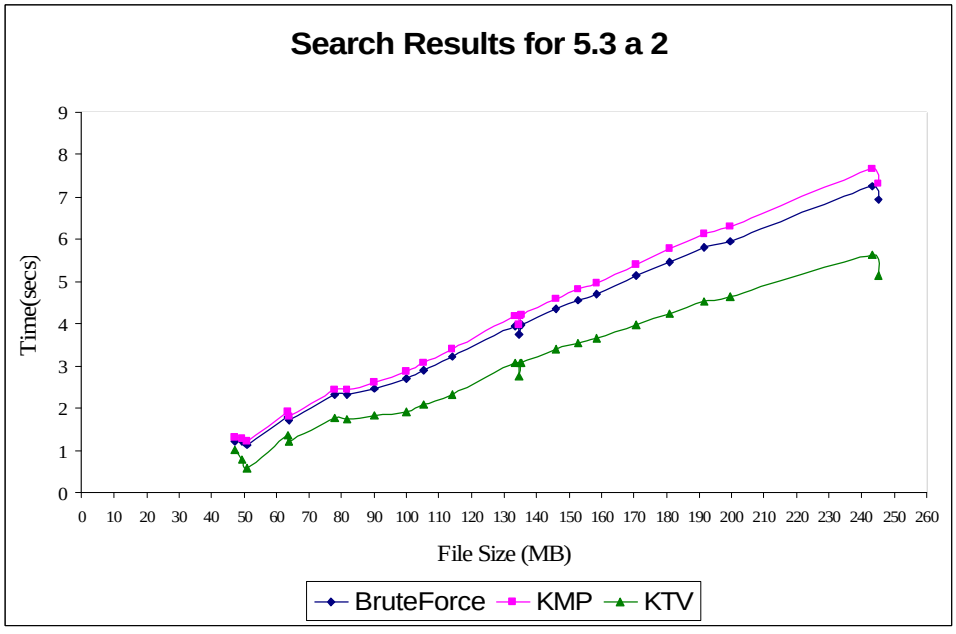

Figure 5-2 : Search Results for 5.3 a 2

S.No	Sample Size	Occurrences	Search Time in Seconds		
			BF	KMP	KTV
1	46,976,537	10,062,440	1.226420	1.295841	1.021599
2	49,476,972	8,978,002	1.210903	1.280126	0.774751
3	50,961,097	6,890,018	1.141688	1.212898	0.590703
4	63,644,868	16,503,721	1.807034	1.904953	1.361593
5	63,790,860	14,381,985	1.701668	1.786864	1.220484
6	77,753,510	22,427,279	2.316282	2.441205	1.785455
7	81,691,216	21,082,323	2.308541	2.429316	1.743744
8	89,995,999	22,007,159	2.481774	2.615588	1.819688
9	100,114,055	23,458,690	2.711950	2.868382	1.914170
10	105,311,216	25,670,202	2.903965	3.063273	2.079309
11	114,151,656	29,324,966	3.214664	3.396818	2.328671
12	133,464,434	38,292,633	3.955090	4.171520	3.080859
13	134,505,819	33,807,672	3.755813	3.963801	2.744183
14	134,978,784	38,183,681	3.975309	4.193568	3.086076
15	135,480,874	38,156,953	3.981836	4.215943	3.069523
16	145,908,738	42,448,402	4.350663	4.592824	3.395131
17	152,634,166	44,668,508	4.561584	4.826400	3.543843
18	158,431,299	45,788,455	4.708755	4.966777	3.666394
19	170,740,541	50,409,420	5.126437	5.409791	3.990255
20	180,967,295	53,602,345	5.443963	5.764939	4.250215
21	191,610,523	57,657,548	5.803383	6.123233	4.527539
22	199,411,731	58,359,484	5.963724	6.307639	4.636781
23	243,315,028	70,765,429	7.249469	7.668920	5.635446
24	245,203,898	63,719,743	6.942533	7.329491	5.145736

Table 5-3: Search Results for 5.3 a 2

5.4.3 Results for Pattern 5.3 a 3

This test case uses a pattern of length nine characters. As it can be seen in Figure 5-3, KTV again performed well. The results show a variance in time from the previous results. Both Brute Force and KMP almost recorded same time while KTV averaged at lower seconds. Also it can be inferred that KTV almost averaged less than a linear increase where as the other two are almost linear with the size of the text.

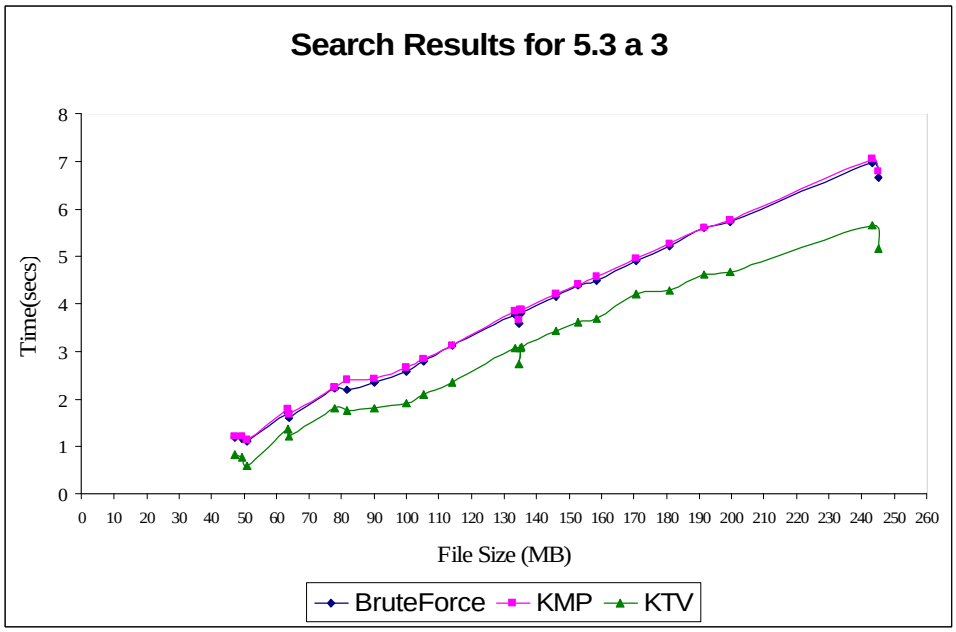

Figure 5-3 : Search Results for 5.3 a 3

S.No	Sample Size	Occurrences	Search Time in Seconds		
			BF	KMP	KTV
1	46,976,537	172	1.183336	1.204694	0.827198
2	49,476,972	93	1.154213	1.206738	0.763175
3	50,961,097	112	1.110685	1.148062	0.597393
4	63,644,868	206	1.713512	1.768929	1.363157
5	63,790,860	142	1.601199	1.670222	1.209462
6	77,753,510	390	2.215396	2.239184	1.817822
7	81,691,216	218	2.180977	2.402149	1.760086
8	89,995,999	269	2.357198	2.435740	1.814971
9	100,114,055	300	2.591508	2.655136	1.912107
10	105,311,216	473	2.784181	2.835398	2.095868
11	114,151,656	559	3.128077	3.126510	2.361092
12	133,464,434	661	3.780210	3.833279	3.078556
13	134,505,819	549	3.595887	3.667474	2.745531
14	134,978,784	633	3.798156	3.858948	3.080993
15	135,480,874	606	3.813668	3.864854	3.084623
16	145,908,738	723	4.167585	4.212731	3.427952
17	152,634,166	830	4.384217	4.412222	3.607223
18	158,431,299	763	4.500066	4.556501	3.698874
19	170,740,541	831	4.909691	4.955164	4.202163
20	180,967,295	967	5.218560	5.270413	4.294843
21	191,610,523	1,154	5.605319	5.590453	4.617117
22	199,411,731	1,061	5.723980	5.766655	4.661800
23	243,315,028	1,189	6.960146	7.042328	5.663650
24	245,203,898	1,028	6.655266	6.778885	5.152020

Table 5-4: Search Results for 5.3 a 3

5.4.4 Results for Pattern 5.3 a 4

This test case uses a pattern of length fifteen characters. As it can be inferred from **Table 5-5**, though the occurrences are few, still the time taken is almost in the same range as that in the previous test case. Investigation on this regard revealed that the test case in 5.4.1 contains a non existing character at the first position. KTV works more efficiently on the time factor if the first character is a not existing. But still as it can be seen its performance is better than the other two.

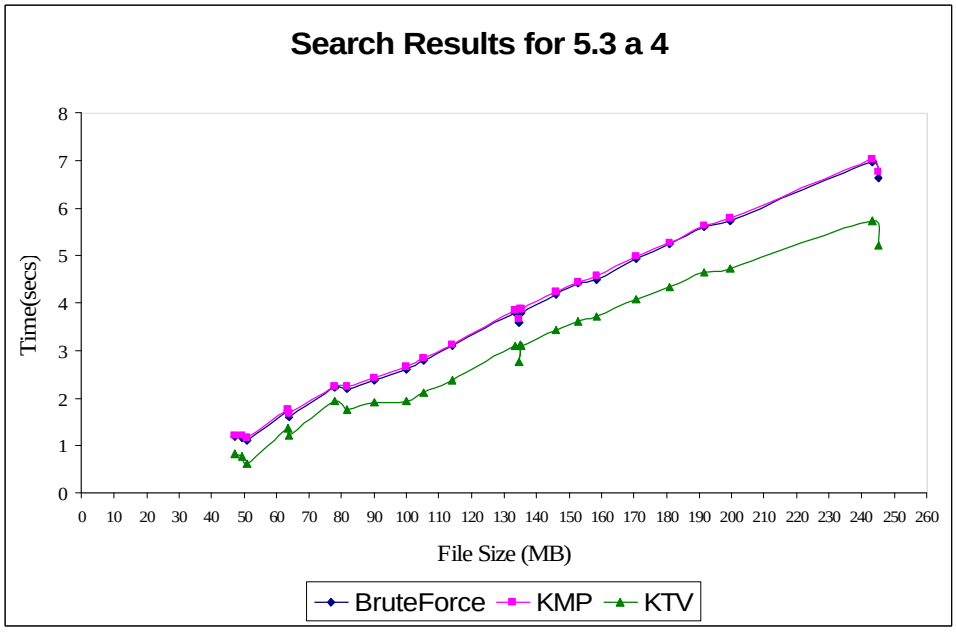

Figure 5-4 : Search results for 5.3 a 4

S.No	Sample Size	Occurrences	Search Time in Seconds		
			BF	KMP	KTV
1	46,976,537	-	1.184113	1.208396	0.835451
2	49,476,972	-	1.155488	1.207899	0.771346
3	50,961,097	-	1.118051	1.149814	0.607726
4	63,644,868	-	1.718562	1.765926	1.371414
5	63,790,860	-	1.602760	1.673810	1.221291
6	77,753,510	-	2.221112	2.240561	1.926467
7	81,691,216	1	2.183906	2.254807	1.748245
8	89,995,999	-	2.371620	2.432553	1.919472
9	100,114,055	1	2.598303	2.658129	1.936921
10	105,311,216	2	2.785693	2.836850	2.105441
11	114,151,656	-	3.106957	3.127952	2.383976
12	133,464,434	-	3.782394	3.838906	3.106516
13	134,505,819	1	3.596987	3.667718	2.768048
14	134,978,784	-	3.797974	3.862060	3.114173
15	135,480,874	1	3.798754	3.869809	3.107110
16	145,908,738	-	4.169155	4.222570	3.437215
17	152,634,166	-	4.403023	4.426241	3.606982
18	158,431,299	1	4.502660	4.563108	3.711432
19	170,740,541	1	4.923796	4.974983	4.065468
20	180,967,295	1	5.242719	5.274171	4.331305
21	191,610,523	3	5.606663	5.615917	4.638695
22	199,411,731	2	5.732169	5.788630	4.712458
23	243,315,028	2	6.962249	7.024011	5.733794
24	245,203,898	2	6.637968	6.771017	5.216083

Table 5-5: Search Results for 5.3 a 4

5.4.5 Results for Pattern 5.3 a 5

This test case produced interesting results than expected depending on the performances acquired in the previous test cases. The results indicate that the results might be dependent on three factors – size of text, size of pattern and the number of occurrences. But as it can be seen the number of occurrences doesn't make much of a difference in the resulting times. But the performance of the algorithms did get affected by the size of pattern and the size of text. Also it can be observed the dip in the time at size 245MB is much lower than the previous test cases. Investigation in this aspect revealed that the difference was due to the size of the pattern and fact that there were more of non existing characters in the pattern.

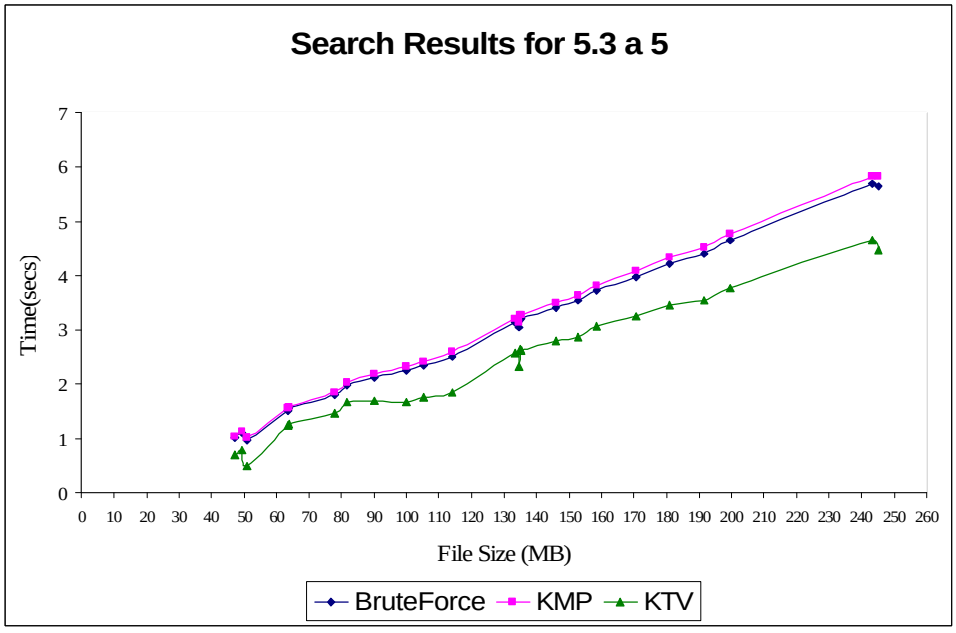

Figure 5-5 : Search results for 5.3 a 5

S.No	Sample Size	Occurrences	Search Time in Seconds		
			BF	KMP	KTV
1	46,976,537	-	1.013712	1.045499	0.694154
2	49,476,972	-	1.110987	1.138230	0.781607
3	50,961,097	-	0.980296	1.027381	0.488876
4	63,644,868	-	1.521961	1.560839	1.252262
5	63,790,860	1	1.547399	1.580038	1.257569
6	77,753,510	-	1.804601	1.853631	1.463302
7	81,691,216	-	1.995354	2.033904	1.664766
8	89,995,999	-	2.128197	2.180330	1.701525
9	100,114,055	-	2.250391	2.315806	1.676751
10	105,311,216	-	2.354057	2.425747	1.752925
11	114,151,656	-	2.516179	2.588573	1.846552
12	133,464,434	-	3.137305	3.213536	2.568236
13	134,505,819	-	3.052809	3.135511	2.323556
14	134,978,784	-	3.196125	3.274228	2.631125
15	135,480,874	-	3.198427	3.279073	2.629903
16	145,908,738	-	3.409637	3.500014	2.804972
17	152,634,166	-	3.544420	3.637510	2.872177
18	158,431,299	-	3.726988	3.822464	3.064220
19	170,740,541	-	3.977433	4.080839	3.254164
20	180,967,295	-	4.221601	4.331382	3.447777
21	191,610,523	-	4.397034	4.518192	3.555599
22	199,411,731	-	4.647971	4.761495	3.781932
23	243,315,028	-	5.692548	5.836513	4.650336
24	245,203,898	1	5.655801	5.832116	4.481596

Table 5-6: Search Results for 5.3 a 5

5.4.6 Comment

As it can be seen from the results KTV performed well for all the test cases. Using it for larger pattern lengths could give us decent results as compared to the regularly used simple search algorithms. Also as it can be seen from the test cases that follow in section 5.5, KTV proves more helpful less times by repeating the search for different patterns.

5.5 Multiple Pattern Search

The test cases were designed so as to test for the performance of the algorithms KPrime and KTV2. The first test case has the patterns discussed in 5.4.2 through 5.4.5. This was to ensure that we get the same results. Other test cases include patterns of fixed size of twenty characters, but the number of patterns were kept increasing except for the test case discussed in 5.5.6. These strings were randomly taken from the DNA data of zebra fish and also from the actual text in which they are searched. The maximum number of patterns which were tested was hundred. Also the patterns were searched individually in KTV2 where as they were handled as one in KPrime.

5.5.1 Results for Pattern 5.3 b 1

This test case was designed to see if the search algorithms were giving correct results as the single pattern search algorithms in terms of occurrences found.

KPrime results were pretty discouraging as it can be seen from Figure 5-6. Investigation in this regard revealed that this was due to the repetitive multiplication function which the search algorithm uses. Since this is the most important part of the algorithm, this function could not be eliminated.

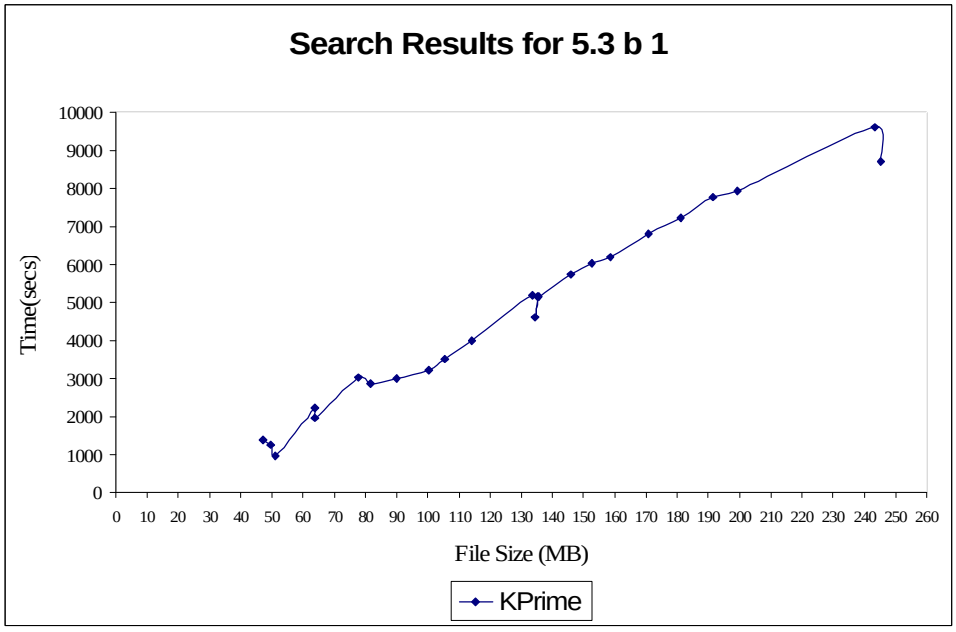

Figure 5-6 : KPrime results for 5.3 b 1

As far as KTV2 was concerned, it gave good results with respect to time. Though the shape of the graphs are almost the same but were on a different time scale. This is mainly due to the fact that the algorithm takes help of the Position array, which keeps track of the first occurrence of the first character in the pattern being searched.

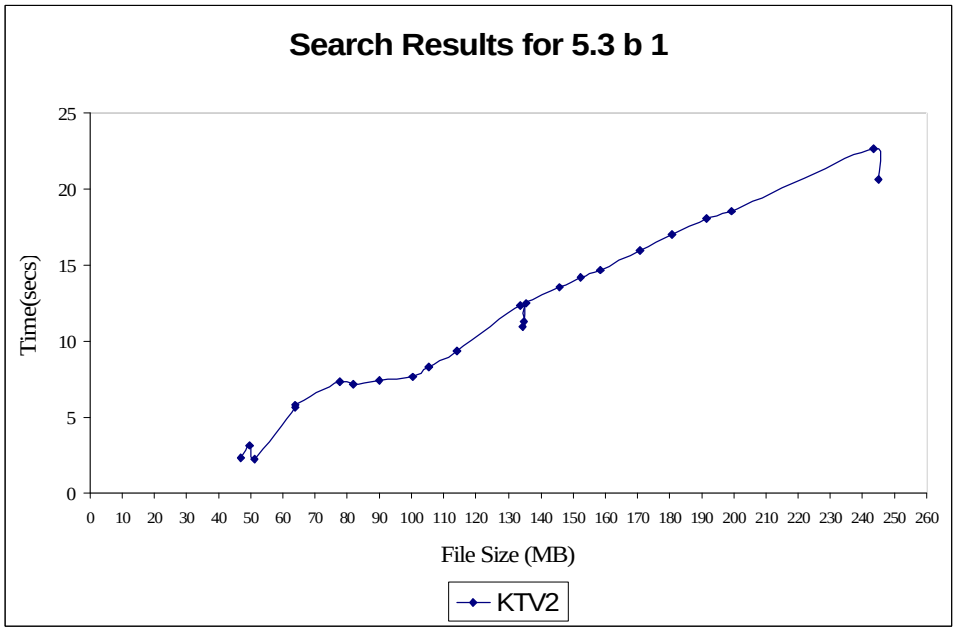

Figure 5-7 : KTV2 results for 5.3 b 1

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	1381.803843	2.367941
2	49,476,972	1255.205689	3.160540
3	50,961,097	972.571558	2.260309
4	63,644,868	2240.740123	5.647738
5	63,790,860	1972.967295	5.804579
6	77,753,510	3029.333463	7.343389
7	81,691,216	2873.009802	7.159451
8	89,995,999	3009.579697	7.421235
9	100,114,055	3220.801205	7.667375
10	105,311,216	3508.995002	8.300159
11	114,151,656	3996.323747	9.342560
12	133,464,434	5192.469819	12.304571
13	134,505,819	4604.669653	10.946519
14	134,978,784	5167.210499	11.261290
15	135,480,874	5172.714954	12.522197
16	145,908,738	5732.925406	13.533954
17	152,634,166	6022.705938	14.174241
18	158,431,299	6183.687670	14.701045
19	170,740,541	6798.711792	15.958586
20	180,967,295	7228.347241	17.008667
21	191,610,523	7769.102322	18.027493
22	199,411,731	7932.631668	18.529769
23	243,315,028	9614.849818	22.683024
24	245,203,898	8711.546778	20.682601

Table 5-7: Search Results for 5.3 b 1

5.5.2 Results for Pattern 5.3 b 2

This test case used five different patterns of twenty characters in length. This was designed to see how well the algorithms handle the patterns when they are different in data but same in length. The effect can be seen in the KPrime. The search time drastically reduced. The assumption that the pattern length has effect on the search time in the previous test results was proved correct.

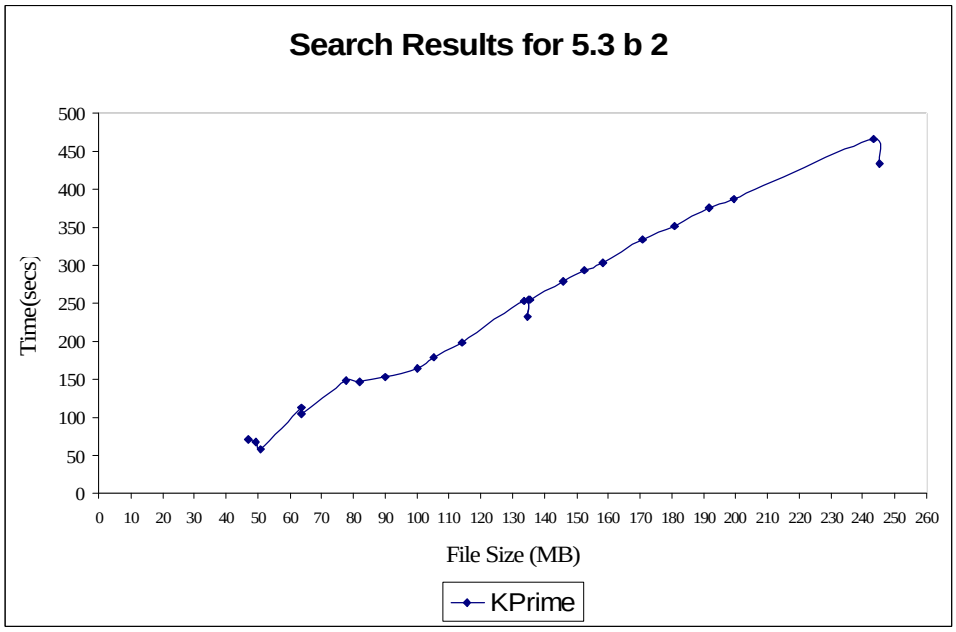

Figure 5-8 : KPrime results for 5.3 b 2

As far as KTV2 was concerned the results given were decent. The number of occurrences or the size of the largest pattern did not make much of a difference. The test results were almost the same as compared to the previous test case.

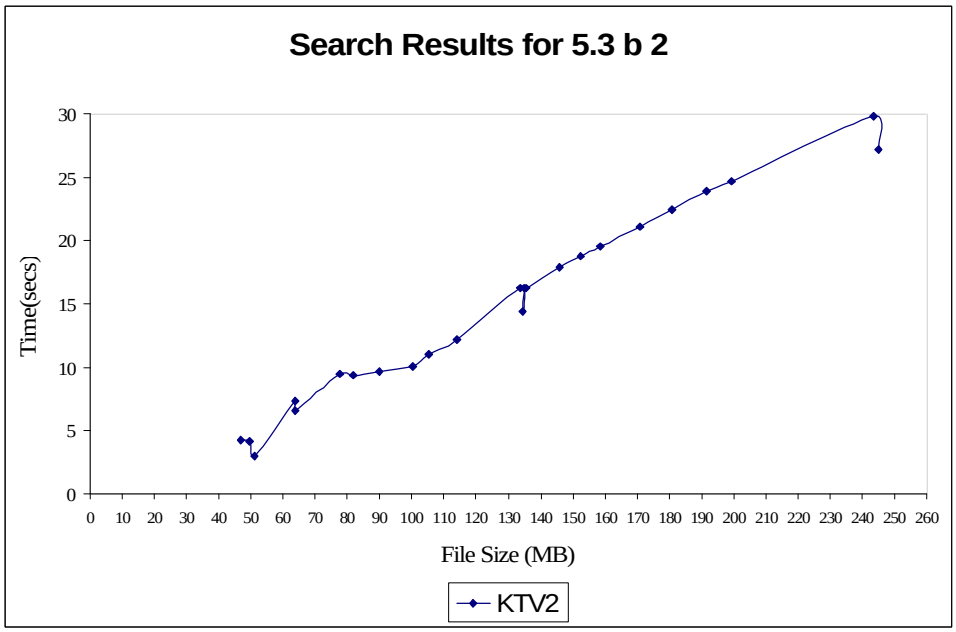

Figure 5-9 : KTV2 results for 5.3 b 2

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	70.843180	4.292034
2	49,476,972	67.889560	4.192291
3	50,961,097	57.972309	2.970446
4	63,644,868	113.221785	7.334186
5	63,790,860	104.272518	6.560908
6	77,753,510	149.042988	9.463691
7	81,691,216	146.363336	9.347251
8	89,995,999	153.725761	9.694050
9	100,114,055	165.281883	10.079919
10	105,311,216	178.371709	11.045695
11	114,151,656	198.805587	12.237198
12	133,464,434	253.955965	16.227524
13	134,505,819	232.000821	14.417742
14	134,978,784	254.177998	16.299633
15	135,480,874	254.400557	16.292190
16	145,908,738	278.856510	17.866517
17	152,634,166	293.532634	18.735664
18	158,431,299	303.746398	19.506384
19	170,740,541	333.356197	21.132365
20	180,967,295	351.981832	22.490933
21	191,610,523	375.623941	23.906428
22	199,411,731	386.703268	24.662541
23	243,315,028	466.386033	29.832875
24	245,203,898	433.375599	27.219134

Table 5-8: Search Results for 5.3 b 2

5.5.3 Results for Pattern 5.3 b 3

This test case used ten different patterns of twenty characters in length. This was designed to see how well the algorithms handle the patterns when they are different in data but same in length. The following comments hold true for the test cases discussed in 5.5.4 through 5.5.9 except for 5.5.6.

This test induced a steep increase in the times for the KPrime algorithm. The increase in the number of patterns did have an effect on the algorithm though the number of occurrences did hardly make any difference.

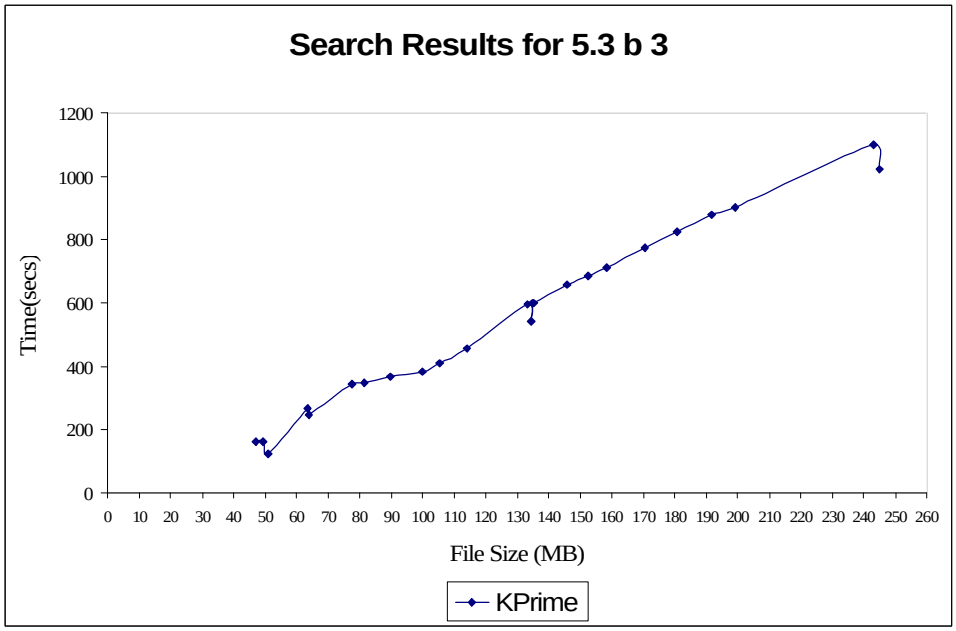

Figure 5-10 : KPrime results for 5.3 b 3

The KTV2 algorithm did show some increase in times but not a much variation. It was quite acceptable due to the fact that the number of patterns was doubled. As in the previous test cases the graphs retained the same shape but the scales are quite different.

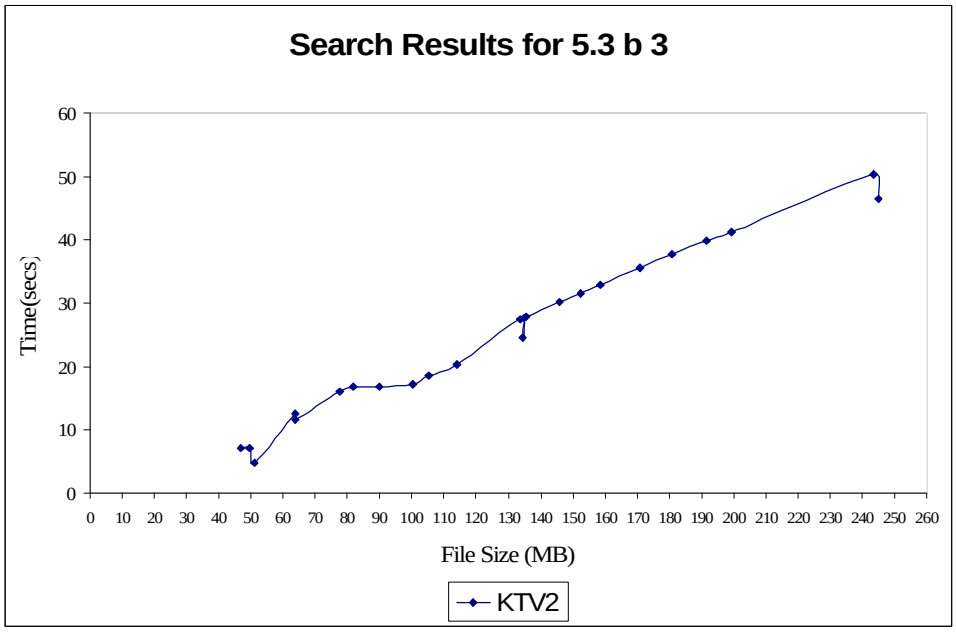

Figure 5-11 : KTV2 results for 5.3 b 3

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	164.317820	7.253664
2	49,476,972	162.015770	7.251675
3	50,961,097	124.810282	4.930707
4	63,644,868	268.694970	12.572023
5	63,790,860	249.396011	11.696838
6	77,753,510	345.927683	16.061259
7	81,691,216	349.752210	16.761641
8	89,995,999	366.220840	16.870338
9	100,114,055	381.432416	17.239325
10	105,311,216	411.127467	18.546939
11	114,151,656	457.523157	20.409496
12	133,464,434	597.348787	27.502861
13	134,505,819	542.617557	24.510082
14	134,978,784	601.743237	27.770173
15	135,480,874	601.678641	27.794656
16	145,908,738	657.393412	30.166849
17	152,634,166	687.087129	31.492380
18	158,431,299	713.798132	32.852983
19	170,740,541	773.081986	35.530958
20	180,967,295	824.016922	37.825344
21	191,610,523	876.987297	39.873452
22	199,411,731	901.978398	41.305512
23	243,315,028	1101.062432	50.416683
24	245,203,898	1023.634967	46.434099

Table 5-9: Search Results for 5.3 b 3

5.5.4 Results for Pattern 5.3 b 4

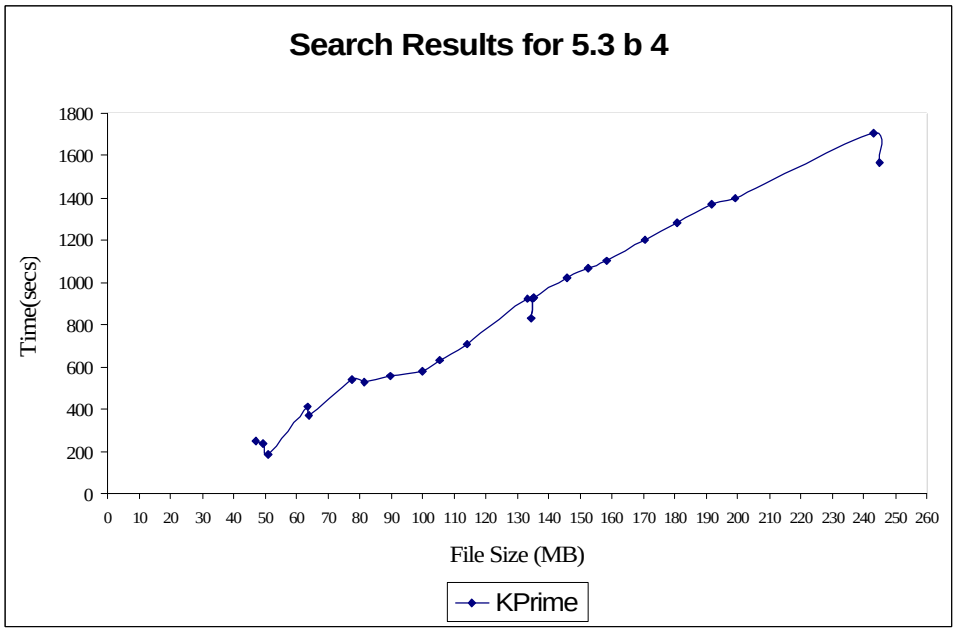

Figure 5-12 : KPrime results for 5.3 b 4

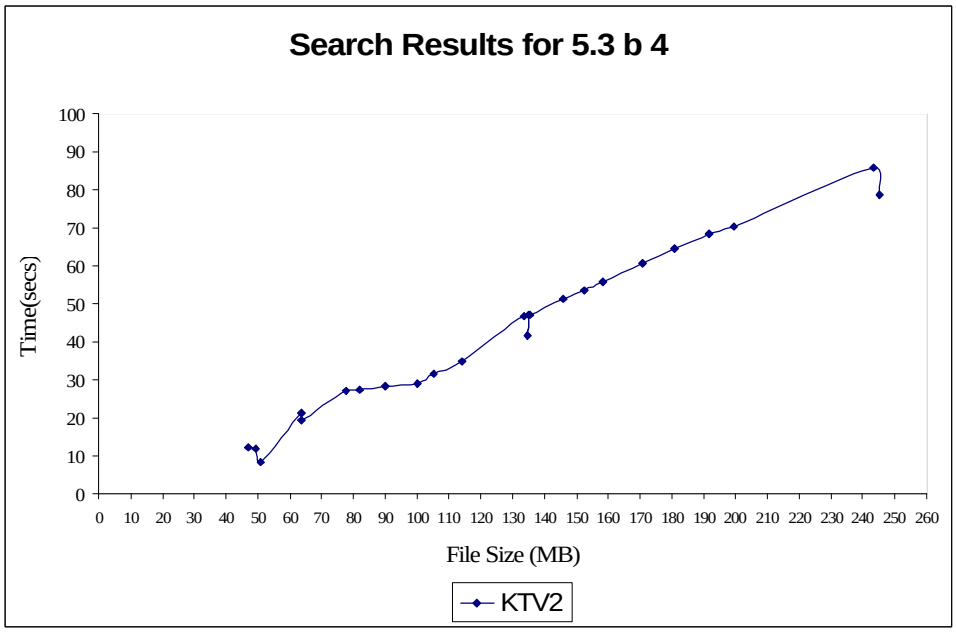

Figure 5-13 : KTV2 results for 5.3 b 4

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	249.938868	12.295118
2	49,476,972	237.465761	12.047602
3	50,961,097	184.466109	8.347660
4	63,644,868	410.392144	21.159558
5	63,790,860	370.428253	19.427896
6	77,753,510	542.342084	27.066946
7	81,691,216	530.510131	27.387178
8	89,995,999	556.417115	28.332412
9	100,114,055	580.327464	29.158087
10	105,311,216	632.781179	31.492105
11	114,151,656	705.729336	34.876788
12	133,464,434	921.637252	46.723866
13	134,505,819	830.479819	41.548884
14	134,978,784	924.451361	47.072879
15	135,480,874	927.080994	47.058871
16	145,908,738	1019.372926	51.340945
17	152,634,166	1070.145230	53.687677
18	158,431,299	1104.270223	55.835056
19	170,740,541	1201.156834	60.574243
20	180,967,295	1285.004529	64.481658
21	191,610,523	1367.577783	68.227217
22	199,411,731	1401.562513	70.341474
23	243,315,028	1706.473388	85.851043
24	245,203,898	1566.962922	78.717328

Table 5-10: Search Results for 5.3 b 4

5.5.5 Results for Pattern 5.3 b 5

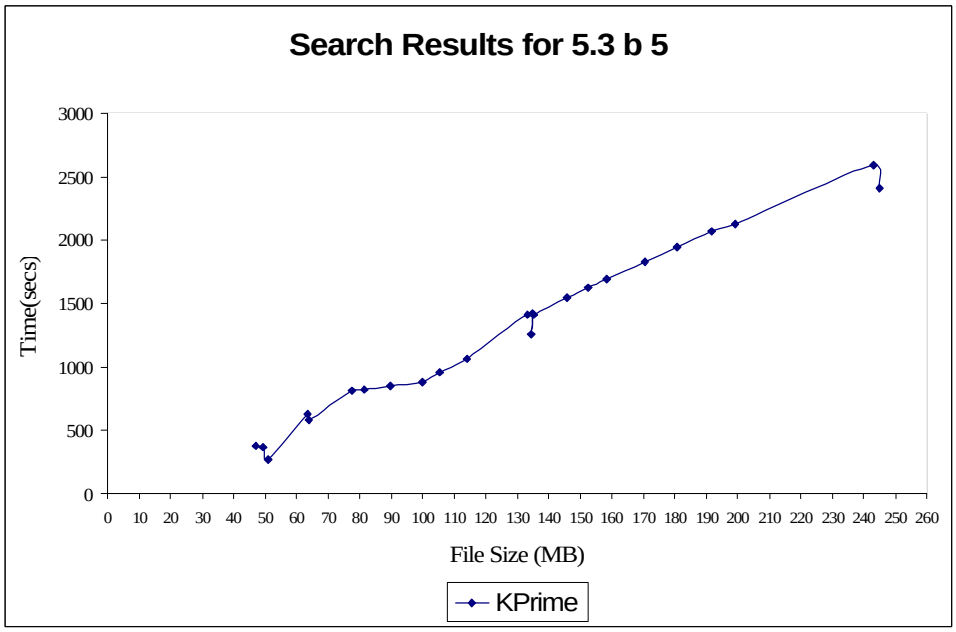

Figure 5-14 : KPrime results for 5.3 b 5

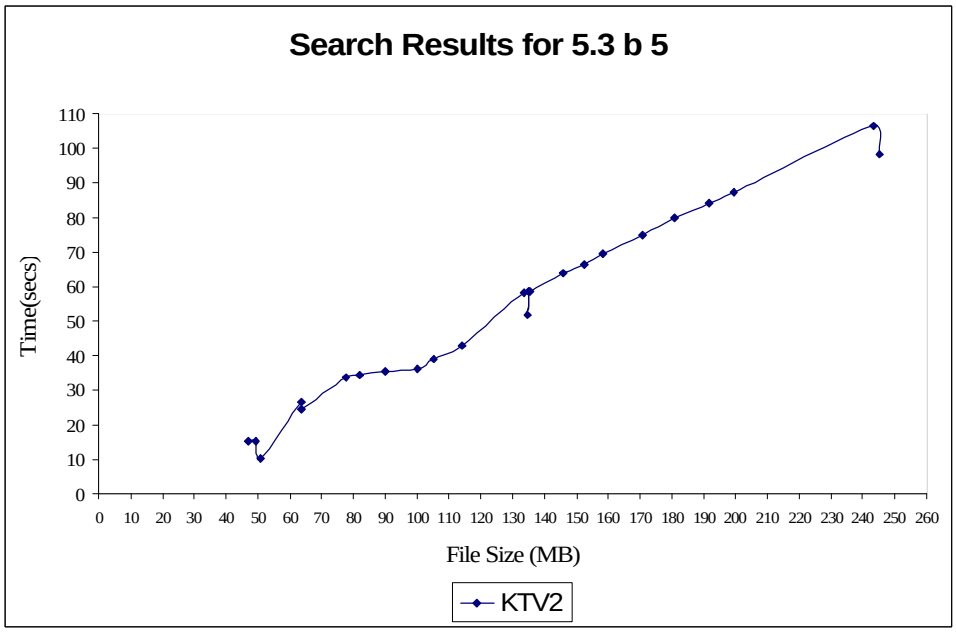

Figure 5-15 : KTV2 results for 5.3 b 5

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	374.646187	15.252259
2	49,476,972	367.997563	15.197695
3	50,961,097	273.564777	10.313441
4	63,644,868	633.849308	26.478501
5	63,790,860	581.474922	24.560212
6	77,753,510	814.176035	33.538428
7	81,691,216	821.717676	34.390308
8	89,995,999	854.757586	35.571729
9	100,114,055	882.926216	36.318339
10	105,311,216	959.846299	39.129896
11	114,151,656	1068.723946	43.087808
12	133,464,434	1411.583747	58.029516
13	134,505,819	1260.248818	51.645791
14	134,978,784	1420.033986	58.563558
15	135,480,874	1416.240376	58.554648
16	145,908,738	1548.870139	63.717285
17	152,634,166	1626.844657	66.529301
18	158,431,299	1688.983883	69.544777
19	170,740,541	1833.714744	75.044808
20	180,967,295	1941.458411	79.914697
21	191,610,523	2069.842370	84.238497
22	199,411,731	2133.535634	87.140797
23	243,315,028	2597.011803	106.491267
24	245,203,898	2414.369072	98.145369

Table 5-11: Search Results for 5.3 b 5

5.5.6 Results for Pattern 5.3 b 6

This test case was designed to check the effect of increasing pattern length on the algorithms. The patterns were used ranged from lengths one to twenty. This would give us occurrences which are decrementing in number. This test case quite useful to check the various assumptions made during the previous test cases.

KPrime did show the effect on the time recorded but still retained the shape.

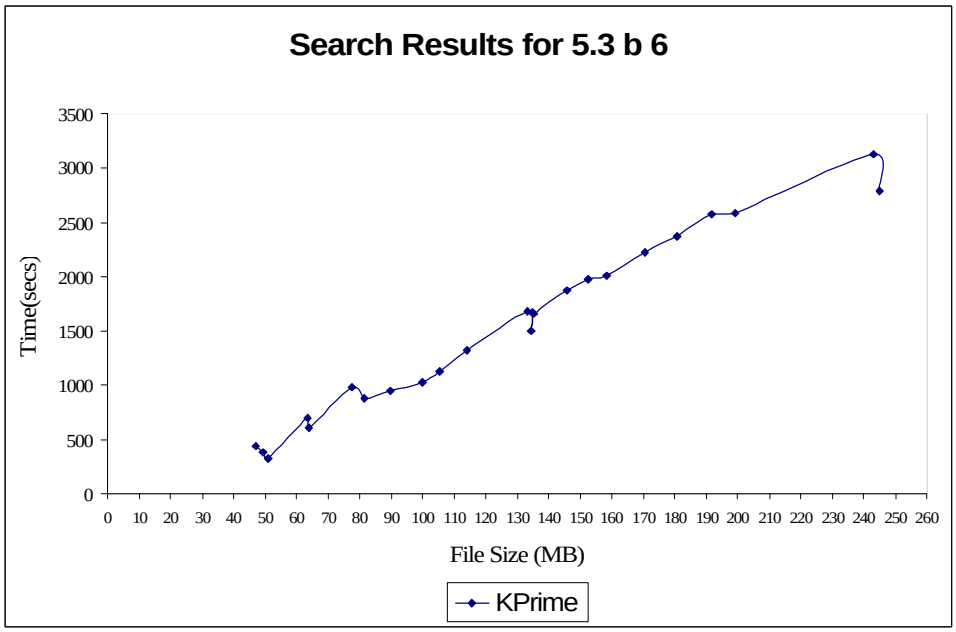

Figure 5-16 : KPrime results for 5.3 b 6

KTV2 on the other hand did work as expected. Number of occurrences did not have much effect on the time recorded. But the number of patterns and size of them did increase the time factor.

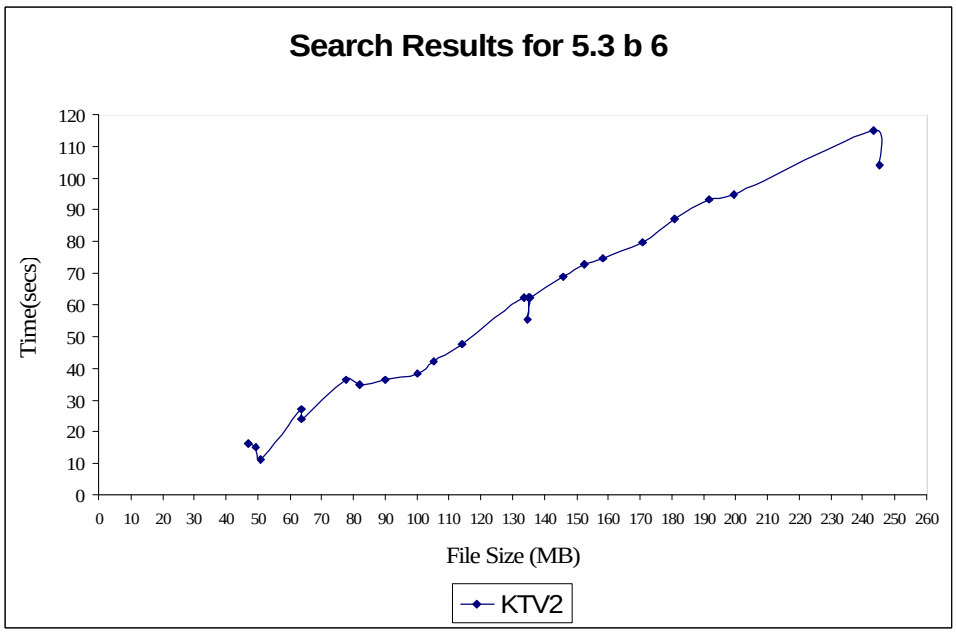

Figure 5-17 : KTV2 results for 5.3 b 6

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	445.081425	16.407512
2	49,476,972	385.393059	14.991219
3	50,961,097	326.371686	11.288010
4	63,644,868	700.706340	27.261644
5	63,790,860	605.452300	24.051447
6	77,753,510	982.438214	36.458769
7	81,691,216	885.964361	34.822973
8	89,995,999	947.340884	36.351625
9	100,114,055	1024.208292	38.310050
10	105,311,216	1133.670682	42.041212
11	114,151,656	1320.171226	47.613044
12	133,464,434	1679.196913	62.350967
13	134,505,819	1496.604366	55.183839
14	134,978,784	1668.703294	62.392889
15	135,480,874	1662.930480	62.271020
16	145,908,738	1870.743890	68.994909
17	152,634,166	1975.448803	72.668177
18	158,431,299	2006.043498	74.580830
19	170,740,541	2225.897266	79.562140
20	180,967,295	2371.913484	87.109532
21	191,610,523	2573.696959	93.378465
22	199,411,731	2581.022843	94.778641
23	243,315,028	3123.458211	115.108887
24	245,203,898	2787.274100	104.066357

Table 5-12: Search Results for 5.3 b 6

5.5.7 Results for Pattern 5.3 b 7

This test case was not used for testing the KPrime algorithm because of the times recorded were way larger than those recorded in 5.5.1. This comment holds true for the other test cases that follow.

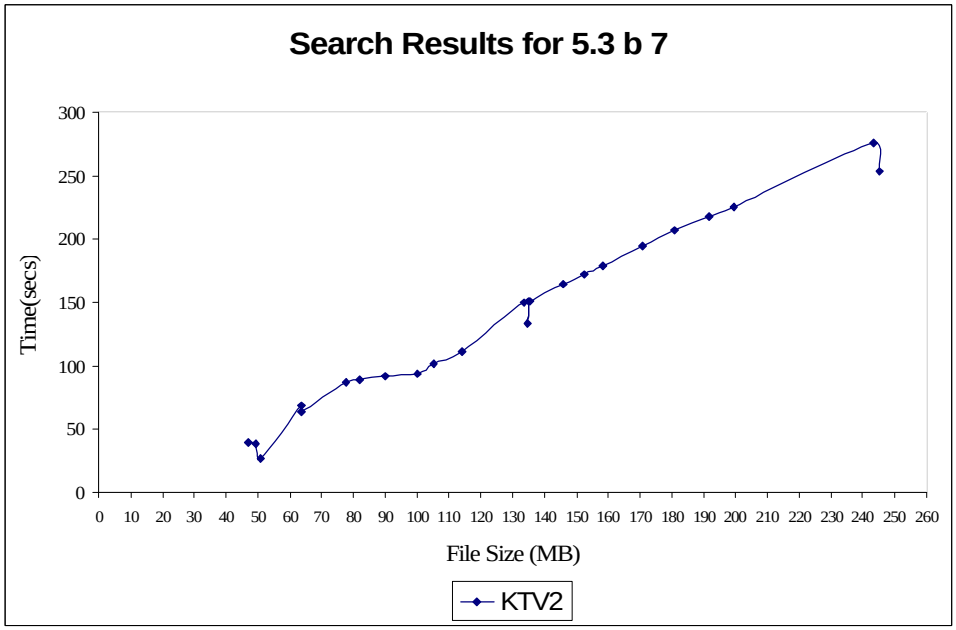

Figure 5-18 : KTV2 results for 5.3 b 7

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	-	39.379163
2	49,476,972	-	39.105183
3	50,961,097	-	26.613804
4	63,644,868	-	68.384062
5	63,790,860	-	63.692960
6	77,753,510	-	86.771678
7	81,691,216	-	88.756190
8	89,995,999	-	91.697530
9	100,114,055	-	93.847029
10	105,311,216	-	101.172617
11	114,151,656	-	111.543413
12	133,464,434	-	150.099079
13	134,505,819	-	133.532575
14	134,978,784	-	151.408488
15	135,480,874	-	151.381402
16	145,908,738	-	164.734304
17	152,634,166	-	172.044750
18	158,431,299	-	179.335775
19	170,740,541	-	194.179371
20	180,967,295	-	206.697578
21	191,610,523	-	218.010220
22	199,411,731	-	225.374151
23	243,315,028	-	275.477268
24	245,203,898	-	253.194739

Table 5-13: Search Results for 5.3 b 7

5.5.8 Results for Pattern 5.3 b 8

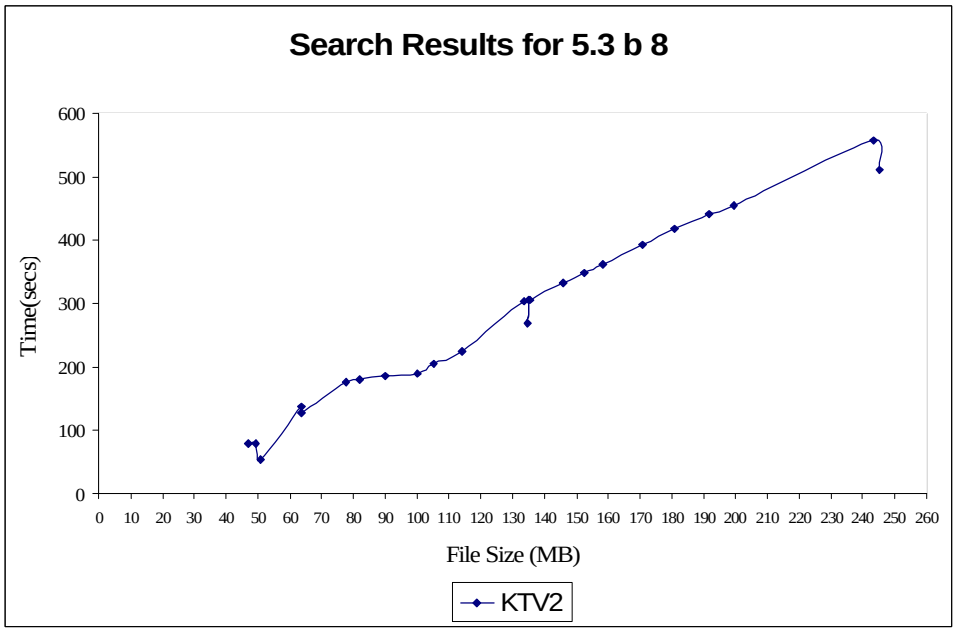

Figure 5-19 : KTV2 results for 5.3 b 8

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	-	79.592837
2	49,476,972	-	78.944370
3	50,961,097	-	53.758858
4	63,644,868	-	137.947255
5	63,790,860	-	127.677649
6	77,753,510	-	175.260865
7	81,691,216	-	179.089369
8	89,995,999	-	185.037572
9	100,114,055	-	189.496933
10	105,311,216	-	204.298518
11	114,151,656	-	225.386368
12	133,464,434	-	303.248940
13	134,505,819	-	269.728069
14	134,978,784	-	305.869150
15	135,480,874	-	306.611373
16	145,908,738	-	332.890987
17	152,634,166	-	347.693934
18	158,431,299	-	362.289322
19	170,740,541	-	392.499012
20	180,967,295	-	417.679800
21	191,610,523	-	440.648249
22	199,411,731	-	455.461220
23	243,315,028	-	556.485166
24	245,203,898	-	511.372055

Table 5-14: Search Results for 5.3 b 8

5.5.9 Results for Pattern 5.3 b 9

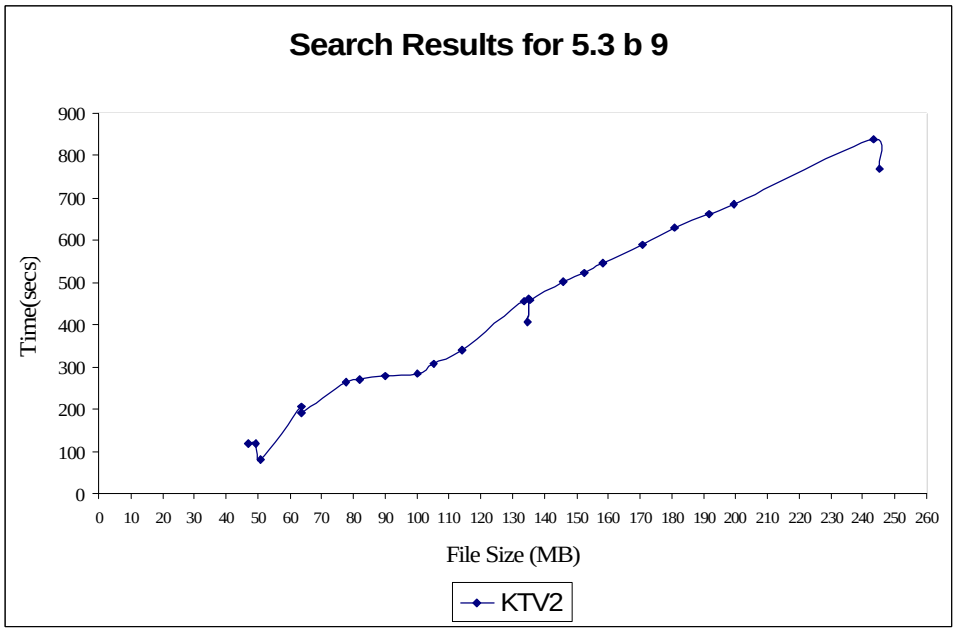

Figure 5-20 : KTV2 results for 5.3 b 9

S.No	Sample Size	Search Time in Seconds	
		KPrime	KTV2
1	46,976,537	-	119.742555
2	49,476,972	-	118.742539
3	50,961,097	-	80.713170
4	63,644,868	-	207.554101
5	63,790,860	-	192.094629
6	77,753,510	-	263.836822
7	81,691,216	-	269.526463
8	89,995,999	-	278.480827
9	100,114,055	-	285.192685
10	105,311,216	-	307.505523
11	114,151,656	-	339.971031
12	133,464,434	-	456.406343
13	134,505,819	-	405.932011
14	134,978,784	-	460.663512
15	135,480,874	-	460.154998
16	145,908,738	-	501.054100
17	152,634,166	-	523.299532
18	158,431,299	-	545.284831
19	170,740,541	-	590.791370
20	180,967,295	-	628.688668
21	191,610,523	-	663.349130
22	199,411,731	-	685.532280
23	243,315,028	-	837.675736
24	245,203,898	-	769.694385

Table 5-15: Search Results for 5.3 b 9

5.5.10 Comment

As is evident from the test cases used the KPrime algorithm requires a lot of tweaking and implementation changes so as to be able to get good results on par with the KTV2 algorithm. KPrime algorithm recorded high times due to the fact of multiplication operation over the pattern. Future work will include replacing this function and finding best replacement. Further during the future work the conceptualized ideas may be implemented and used in the redesign of the algorithm.

Chapter 6
Future Work

As it can be seen from the results KTV gives good results both in the case of single pattern and multiple patterns. It can also be seen that it is more efficient in the multiple pattern because the time complexity is greatly reduced due to the fact that the regeneration of KTV structure is not necessary. KPrime algorithm on the other hand works fine with fewer patterns of a small length. A more efficient implementation than the one used for testing can be used and tested for the same patterns as above.

The algorithms and the test cases used to test them were DNA specific. These algorithms can be modified so as to handle the entire English alphabet along with the Arabic numerical.

During the research of the algorithms described in 4.1 and 4.2, many new theories were formulated but were discarded since they were out of scope of this thesis. The future work would also include implementation of these ideas and applying them to the above said algorithms.

References

[1] National Institute of Standards and Technology (2003, April). S**tring matching.** Retrieved on May 25, 2003, from URL: http://www.nist.gov/dads/HTML/stringMatching.html

[2] Ian Foster (1995). **A Parallel Programming Model**. Retrieved on May 25, 2003, from URL: http://www-unix.mcs.anl.gov/dbpp/text/node9.html

[3] M.V. Olson, 1995. **A time to sequence**, Science, 270:394-396.

[4] Florida Institute of Technology (2002, September). **The Beowulf Project at Florida Tech**. Retrieved on February 02, 2003 from URL: http://my.fit.edu/beowulf/

[5] [Unknown]. **Pattern Matching**. Retrieved on May 01, 2003 from URL: http://www.dcs.shef.ac.uk/~u0rf/pattern.htm

[6] James S. Huggins (2003). **How Much Data Is That?** Retrieved on May 25, 2003 from URL: http://www.jamesshuggins.com/h/tek1/how_big.htm

[7] Telecommunications: Glossary of Telecommunication terms (2003). **Parallel Computer**. Retrieved on March 10, 2003 from URL: http://glossary.its.bldrdoc.gov/fs-1037/dir-026/_3841.htm

[8] Telecommunications: Glossary of Telecommunication terms (2003). **Parallel Processing**. Retrieved on March 10, 2003 from URL: http://glossary.its.bldrdoc.gov/fs-1037/dir-026/_3843.htm

[9] National Institute of Standards and Technology (2003, April). **Brute Force String Search**. Retrieved on March 10, 2003 from URL: http://www.nist.gov/dads/HTML/bruteForceStringSearch.html

[10] National Institute of Standards and Technology (2003, April). **Knuth-Morris-Pratt algorithm**. Retrieved on March 10, 2003 from URL: http://www.nist.gov/dads/HTML/knuthMorrisPratt.html

[11] ACM Journal, 1975, **Communications of the ACM**, Vol. 18, No 6, Association of Computing Machinery, Inc.

[12] Sun Kim, Yanggon Kim, 1997, **A Fast Multiple String-Pattern Matching Algorithm**, ACM Journal of Experimental Algorithmics.

[13] David A Bader (2002, October) **Para Scope: A Listing of Parallel Computing Sites**. Retrieved on March 10,2003 from URL: http://www.computer.org/parascope/

[14] Al Kelley, Ira Pohl, 1987. **C by Dissection: The Essentials of C programming**, Benjamin/Cummings Publishing Co.

[15] Chandra Rohit, 2001. **Parallel Programming in OpenMP**, Morgan Kaufmann Publishers.

[16] Kai Zhang, 1993. **A comparative study of fast full-text search algorithms**, Thesis (M.S)--Florida Institute of Technology.

[17] Maxime Crochemore, Wojciech Rytter, 1994. **Text algorithms**, Oxford University Press

[18] Dan Gusfield, 1999. **Algorithms on Strings, Trees and Sequences**, **Computer science and computational Biology**, Cambridge University Press.

[19] Ellis Horowitz, Sartaj Sahni, Sanguthevar Rajasekaran, 1997. **Computer algorithms / C++**, Computer Science Press.

[20] Gary Bronson, Stephen J. Menconi, 1991. **A first book of C: Fundamentals of C programming**, West Publishing. Co.

[21] Peter S. Pacheco, 1997. **Parallel Programming with MPI**, Morgan Kaufmann Publishers, Inc.

[22] Florida Institute of Technology (2002, September). **The Beowulf Project at Florida Tech**. Retrieved on February 02, 2003 from URL: http://my.fit.edu/beowulf/papers/userguide.txt

[23] Mathematics and Computer Science Division, Argonne National Laboratory (2003, March). **MPICH-A Portable Implementation of MPI**. Retrieved on March 10, 2003 from URL: http://www-unix.mcs.anl.gov/mpi/mpich/index.html

[24] Mathematics and Computer Science Division, Argonne National Laboratory (2003, March). **The Message Passing Interface (MPI) standard**. Retrieved on March 10, 2003 from URL: http://www-unix.mcs.anl.gov/mpi/index.html

[25] Beowulf.org -- Beowulf cluster information (2000). **Beowulf History**. Retrieved on March 10,2003 from URL: http://www.beowulf.org/beowulf/history.html

[26] Michael John LIDDELL (1997, December). **Knuth, Morris and Pratt**. Retrieved on March 10, 2003 from URL: http://www.cs.mu.oz.au/~mjl/thesis/node5.html

[27] Alfred V. Aho, Margaret J. Corasick, 1975. **Efficient String Matching: An Aid to Bibliographic Search**, Association of Computing Machinery, Inc.

[28] Michael E Smith, 2003. **Digital Signal Processing Techniques In The Prediction Of Protein Secondary Structures**, Thesis(MS), Florida Institute of Technology

[29] Baldi, P., Brunak S, 2001**. Bioinformatics**, The Machine Learning Approach. MIT Press

[30] Bourne, Philip E., Weissig, Helge, 2003. **Structural Bioinformatics**. John Wiley & Sons.

[31] Brandon C., Tooze J., 1999. **Introduction to Protein Structure**. Garland Publishing.

[32] Dunbrack, R.. 1999. **Comparative Modeling of CASP3 Targets Using PSI-BLAST and SCWRL**. PROTEINS: Structure, Function and Genetics 3:81-87

[33] Person, W.R., Wood, T., Zhang, Z., and Miller, W., 1997. **Comparision of DNA sequences with protein sequences**, Genomics 46: 24-36

[34] Setubal J., Meidanis J., 1997. **Introduction to Computational Molecular Biology**. PWS Publishing Company.

[35] Westbrook, J., Feng, Z., Jain, S., Bhat, T., Thanki, N., Ravichandran, V., Gilliland, G., Bluhm, W., Weissig, H., Greer, D., Bourne, P., Berman, H. 2002. **The Protein Data Bank: unifying the archive**. Nucleic Acids Research vol. 30 no. 1:245-248

[36] Zhang, C., Lin, Z.S., Zhang, Z.D., Yan, M., 1998. **Prediction of the helix/strand content of globular proteins based on their primary sequences**. Protein Engineering, Vol. 11, No. 11, 971-979

[37] Brenda S. Baker, 1995. **Parameterized Pattern Matching by Boyer-Moore-type Algorithms.** ATA&T Bell Laboratories.

[38] Daniel M. Sunday, 1990. **Avery Fast Substring Search Algorithm**. ACM 0001-0782/90/0300-0132.

Appendix A
Brute Force Program Listing

```
// BruteForce_01.cpp - Kishore R Kattamuri
// MS(CS) Thesis - Florida Tech

/* Required include files
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

/* This is the text in which the search is performed
 */
#define inFile "Chromosome01.txt"
/* This is the file which contains patterns
 */
#define serFile "patterns.txt"
/* This is the text file which is stored on the nodes
 */
#define inFile2 "/scr/Chromosome01.txt"

/* Predeclaration of the Bruteforce
 */
long int BruteForce(char *,char *);

/* This array will hold number of Occurances
 */
long int totaloccurances;

/* Various variables required for timing the search
 */
double start,end,startmain,endmain,swatch;
```

```
/* Function to get the filesize of the passed parameter file
 */
long int getFileSize(char file[])
{
   long int size;
   FILE *filename = fopen(file,"r");
   fseek(filename,0,SEEK_END);
   size = ftell(filename);
   fclose(filename);
   return size;
}

/* Function to calculate the size to be read by each node
  along with the overlap
 */
long int* getOptimumSizeToRead(long int mainFileSize,
                        int processors,
                        long int overlap)
{
   long int* optimumSize;
   long int tempSize;
   long int estimate;
   long int tmpProcs;
   int counter;

   if (processors < 1) return 0;
   tempSize = mainFileSize;
   estimate = (tempSize % processors == 0)
           ? (tempSize / processors)
           : (tempSize / processors) + 1;
   tmpProcs = tempSize / estimate;
   optimumSize = (long int *)malloc(sizeof(long int) * processors);
   tempSize = tempSize - estimate;
   optimumSize[0] = estimate;
   for(counter=1;counter<processors;counter++)
   {
     optimumSize[counter] = (counter<=tmpProcs)
                        ? estimate + overlap : 0;
   }

   return optimumSize;
}
```

```
/* Function to caluclate the overlap
 */
long int getOverlap(long int size)
{
   return size - 1;
}

/* The main function of the program
 */
main(int argc, char* argv[])
{
   /* Required variables
    */
   int my_rank;
   int p;
   int source;
   int dest;
   int tag=0;
   char message[1000];
   char num[1000];
   MPI_Status status;
   FILE *fp;
   char *searchstring;
   char *docstring;
   long int occur=0;
   long int sizefile;
   long int overlap;
   long int *optimumSize;

   /* Initialising the MPI environment variables
    */
   MPI_Init(&argc,&argv);
   /* Start Timer for the complete program
    */
   startmain=MPI_Wtime();
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   sizefile = getFileSize(serFile);
   searchstring = (char *)malloc((sizefile)*sizeof(char));
```

```
    fp=fopen(serFile,"r");
    fgets(searchstring,sizefile,fp);
    fclose(fp);

    overlap = getOverlap(sizefile);
    sizefile = getFileSize(inFile);

    optimumSize = getOptimumSizeToRead(sizefile,p-1,overlap);

/* If the process is not being run on parent Node
    */
    if(my_rank != 0)
    {
       /* Open the text file and allocate memory to docstring
        */
       fp = fopen(inFile2,"r");
       docstring = (char *)malloc
                ((optimumSize[my_rank-1]+1)*sizeof(char));
       /* If the node is first in the current node set the
         filepointer to position 0
        */
       if (my_rank == 1)
       {
          fseek(fp,0,SEEK_SET);
       }
       /* Otherwise set it to Optimum size time the node rank
        */
       else
       {
          fseek(fp,(((my_rank-1)*optimumSize[0])-(overlap)),SEEK_SET);
       }
       /* Read the text data into docstring
        */
       fgets(docstring,optimumSize[my_rank-1]+1,fp);
       /* Do the search and catch the results in Occurances array
        */
       occur = BruteForce(docstring,searchstring);
       /* Store the values in the message array
        */
       sprintf(message,"%ld %f",occur,end-start);
       /* Set the destination to parent node
        */
```

```
        dest=0;
        fclose(fp);
        /* Send the message array to the parent node
         */
        MPI_Send(message,strlen(message)+1,
                        MPI_CHAR,dest,tag,MPI_COMM_WORLD);
    }
    /* If the process is being run on parent Node
     */
    else
    {
        /* For all the nodes in the node set except the parent node
           recieve the message array sent by them and process them
           accordingly
         */
        for(source=1;source<p;source++)
        {
            MPI_Recv(message,1000,
                            MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
            printf("Processor with Rank %2d : %s\n",source,message);
            totaloccurances += atol(strtok(message," "));
            swatch += atof(strtok(NULL," "));
        }
        /* Stop the timer for the complete program
         */
        endmain=MPI_Wtime();
        /* Print the results to the standard output
         */
        printf("Total Occurances : %ld\n",totaloccurances);
        printf("Total Time for Searching : %f\n",swatch);
        printf("Total Time for Execution : %f\n",endmain-startmain);
    }
    /* Close the MPI environment
     */
    MPI_Finalize();
}

/* Implementation of Brute Force
 */
long int BruteForce(char *string, char *search_string)
{
    int i, j, k;
```

```
    long int count = 0, occurences = 0;
    int first = 0;

    const long int len_search_string = strlen(search_string);
    const long int len_given_string = strlen(string);
    const long int limit = len_given_string - len_search_string;
    start = MPI_Wtime();
    for (i = 0; i <= limit; i++ )
    {
       count = 0;
       for(j = i, k = 0; k < (len_search_string) ; j++, k++)
       {
          if(*(string + j) != *(search_string + k) )
          {
             break;
          }
          else
          {
             count++;
          }
          if(count == len_search_string )
          {
             occurences++;
          }
      }
    }
    end = MPI_Wtime();
return occurences;
}
```

Appendix B
KMP Program Listing

```
// KMP_01.cpp - Kishore R Kattamuri
// MS(CS) Thesis - Florida Tech

/* Required include files
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

/* This is the text in which the search is performed
 */
#define inFile "Chromosome01.txt"
/* This is the file which contains patterns
 */
#define serFile "patterns.txt"
/* This is the text file which is stored on the nodes
 */
#define inFile2 "/scr/Chromosome01.txt"
/* Maximum size of the pattern
 */
#define MAX_PAT_SIZE 255

/* Predeclaration of the preKMP
 */
void  preKmp(char *,long int,long int *);
/* Predeclaration of the KMP
 */
long int KMP(char *,long int, char *,long int);

/* This array will hold number of Occurances
 */
long int totaloccurances;
```

```
/* Various variables required for timing the search
 */
double start,end,startmain,endmain,swatch;

/* Function to get the filesize of the passed parameter file
 */
long int getFileSize(char file[])
{
   long int size;
   FILE *filename = fopen(file,"r");
   fseek(filename,0,SEEK_END);
   size = ftell(filename);
   fclose(filename);
   return size;
}

/* Function to calculate the size to be read by each node
  along with the overlap
 */
long int* getOptimumSizeToRead(long int mainFileSize,
                        int processors,
                        long int overlap)
{
   long int* optimumSize;
   long int tempSize;
   long int estimate;
   long int tmpProcs;
   int counter;

   if (processors < 1) return 0;
   tempSize = mainFileSize;
   estimate = (tempSize % processors == 0)
           ? (tempSize / processors)
           : (tempSize / processors) + 1;
   tmpProcs = tempSize / estimate;
   optimumSize = (long int *)malloc(sizeof(long int) * processors);
   tempSize = tempSize - estimate;
   optimumSize[0] = estimate;
   for(counter=1;counter<processors;counter++)
   {
     optimumSize[counter] = (counter<=tmpProcs) ? estimate + overlap : 0;
   }
```

```
   return optimumSize;
}

/* Function to caluclate the overlap
 */
long int getOverlap(long int size)
{
   return size - 1;
}

/* The main function of the program
 */
main(int argc, char* argv[])
{
   /* Required variables
    */
   int my_rank;
   int p;
   int source;
   int dest;
   int tag=0;
   char message[1000];
   char num[1000];
   MPI_Status status;
   FILE *fp;
   char *searchstring;
   char *docstring;
   long int occur=0;
   long int sizefile;
   long int overlap;
   long int *optimumSize;

   /* Initialising the MPI environment variables
    */
   MPI_Init(&argc,&argv);
   /* Start Timer for the complete program
    */
   startmain=MPI_Wtime();
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
sizefile = getFileSize(serFile);
searchstring = (char *)malloc((sizefile)*sizeof(char));

fp=fopen(serFile,"r");
fgets(searchstring,sizefile,fp);
fclose(fp);

overlap = getOverlap(sizefile);
sizefile = getFileSize(inFile);

optimumSize = getOptimumSizeToRead(sizefile,p-1,overlap);

/* If the process is not being run on parent Node
 */
if(my_rank != 0)
{
   /* Open the text file and allocate memory to docstring
    */
   fp = fopen(inFile2,"r");
   docstring = (char *)malloc
            ((optimumSize[my_rank-1]+1)*sizeof(char));
   /* If the node is first in the current node set the
     filepointer to position 0
    */
   if (my_rank == 1)
   {
      fseek(fp,0,SEEK_SET);
   }
   /* Otherwise set it to Optimum size time the node rank
    */
   else
   {
      fseek(fp,(((my_rank-1)*optimumSize[0])-(overlap)),SEEK_SET);
   }
   /* Read the text data into docstring
    */
   fgets(docstring,optimumSize[my_rank-1]+1,fp);
   /* Do the search and catch the results in Occurances array
    */
   occur = KTV(docstring,searchstring);
   /* Store the values in the message array
    */
```

```
        sprintf(message,"%ld %f",occur,end-start);
        /* Set the destination to parent node
         */
        dest=0;
        fclose(fp);
        /* Send the message array to the parent node
         */
        MPI_Send(message,strlen(message)+1,
                        MPI_CHAR,dest,tag,MPI_COMM_WORLD);
    }
    /* If the process is being run on parent Node
     */
    else
    {
        /* For all the nodes in the node set except the parent node
           recieve the message array sent by them and process them
           accordingly
         */
        for(source=1;source<p;source++)
        {
            MPI_Recv(message,1000,
                            MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
            printf("Processor with Rank : %2d %s\n",source,message);
            totaloccurances += atol(strtok(message," "));
            swatch += atof(strtok(NULL," "));
        }
        /* Stop the timer for the complete program
         */
        endmain=MPI_Wtime();
        /* Print the results to the standard output
         */
        printf("Total Occurances : %ld\n",totaloccurances);
        printf("Total Time for Searching : %f\n",swatch);
        printf("Total Time for Execution : %f\n",endmain-startmain);
    }
    /* Close the MPI environment
     */
    MPI_Finalize();
}

/* Implementation of the function preKMP
 */
```

```
void preKmp(char *search_string,long int search_string_len,long int kmpNext[])
{
   long int i, j;
   i = 0;
   j = kmpNext[0] = -1;
   while (i < search_string_len)
   {
      while (j > -1 && search_string[i] != search_string[j])
         j = kmpNext[j];
      i++;
      j++;
      if (search_string[i] == search_string[j])
         kmpNext[i] = kmpNext[j];
      else
         kmpNext[i] = j;
   }
}

/* Implementation of the function KMP
 */
long int KMP(char *search_string, long int search_string_len,
                  char *string, long int string_len)
{
   long int i, j, k, kmpNext[105];
   preKmp(search_string, search_string_len, kmpNext);
   start = MPI_Wtime();
   i = j = k = 0;
   while (j < string_len)
   {
      while (i > -1 && search_string[i] != string[j])
         i = kmpNext[i];
      i++;
      j++;
      if (i >= search_string_len)
      {
         k++;
         i = kmpNext[i];
      }
   }
   end = MPI_Wtime();
return k;
}
```

Appendix C
KTV Program Listing

```
// KTV_01.cpp - Kishore R Kattamuri
// MS(CS) Thesis - Florida Tech

/* Required include files
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

/* This is the text in which the search is performed
 */
#define inFile "Chromosome01.txt"
/* This is the file which contains patterns
 */
#define serFile "patterns.txt"
/* This is the text file which is stored on the nodes
 */
#define inFile2 "/scr/Chromosome01.txt"
/* Maximum size of the pattern
 */
#define MAX_PAT_SIZE 255

/* Predeclaration of the preKTV
 */
void preKTV(char *, long int);

/* Predeclaration of the KTV
 */
long int KTV(char *, char *);
```

```
/* Definition of KTV structure
 */
struct aa
{
        char *c;
   struct aa *next;
};

/* The KTV2 structure array which is built in the preKTV
 */
struct aa *p;

/* Positions array
 */
struct aa positions[26];

/* Temporary array to hold the previous occurance of
  the character
 */
struct aa *prev[26];

/* Alpha array which holds all the available characters
  in the text under tes
 */
char *alpha[26]={
    "a","b","c","d",NULL,NULL,"g","h",NULL,NULL,"k",
    NULL,"m","n",NULL,NULL,NULL,"r","s","t","u","v",
    "w",NULL,"y",NULL};

/* This array will hold number of Occurances
 */
long int *totaloccurances;

/* Various variables required for timing the search
 */
double start,end,startmain,endmain,swatch;

/* Number of patterns used for search
 */
int pCount;
```

```
/* Patterns used for search
 */
char **patterns;

/* Maximum pattern length read
 */
int pLength;

/* Function to read patterns from the file passed
  as argument into the variable Patterns
 */
void readPatterns(char file[])
{

   FILE *filename = fopen(file,"r");
   int i = 0,j,k,tpLength;

   /* Allocating enough memory
    */
   patterns = (char **)malloc(sizeof(char *)*1);
   patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);

   /* Reading pattern one and storing it
    */
   fgets(patterns[i],MAX_PAT_SIZE,filename);
   patterns[i][strlen(patterns[j])-1]='\0';

   /* Intialise the pLength
    */
   pLength = strlen(patterns[i]);

   /* Reading rest of the patterns
    */
   while(!feof(filename))
   {
      i++;
      patterns = (char **)realloc(patterns,sizeof(char *)*(i+1));
      patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);
      fgets(patterns[i],MAX_PAT_SIZE,filename);
      patterns[i][strlen(patterns[i])-1]='\0';
```

```
        /* Updating pLength depending on the length of the current
           pattern read
         */
        pLength
          = (pLength < strlen(patterns[i]))
            ? strlen(patterns[i]) : pLength;
    }
    pCount = i;
    patterns[pCount]=NULL;
    fclose(filename);
}

/* Function to get the filesize of the passed parameter file
 */
long int getFileSize(char file[])
{
    long int size;
    FILE *filename = fopen(file,"r");
    fseek(filename,0,SEEK_END);
    size = ftell(filename);
    fclose(filename);
    return size;
}

/* Function to calculate the size to be read by each node
   along with the overlap
 */
long int* getOptimumSizeToRead(long int mainFileSize,
                               int processors,
                               long int overlap)
{
    long int* optimumSize;
    long int tempSize;
    long int estimate;
    long int tmpProcs;
    int counter;

    if (processors < 1) return 0;
    tempSize = mainFileSize;
    estimate = (tempSize % processors == 0)
              ? (tempSize / processors)
              : (tempSize / processors) + 1;
```

```
    tmpProcs = tempSize / estimate;
    optimumSize = (long int *)malloc(sizeof(long int) * processors);
    tempSize = tempSize - estimate;
    optimumSize[0] = estimate;
    for(counter=1;counter<processors;counter++)
    {
      optimumSize[counter] = (counter<=tmpProcs)
                        ? estimate + overlap : 0;
    }
    return optimumSize;
}

/* Function to caluclate the overlap
 */
long int getOverlap(long int size)
{
    return size - 1;
}

/* The main function of the program
 */
main(int argc, char* argv[])
{
    /* Required variables
     */
    int my_rank;
    int p;
    int source;
    int dest;
    int tag=0;
    char message[1000];
    MPI_Status status;
    FILE *fp;
    char *searchstring;
    char *docstring;
    long int occur=0;
    long int sizefile;
    long int overlap;
    long int *optimumSize;
    long int *Occurances;
```

```
/* Initialising the MPI environment
 */
MPI_Init(&argc,&argv);
/* Start Timer for the complete program
 */
startmain=MPI_Wtime();
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* Read the patterns and calculate filesizes
 */
readPatterns(serFile);
sizefile = pLength;
overlap = getOverlap(sizefile);
sizefile = getFileSize(inFile);

/* Estimate the optimum size for each node
 */
optimumSize = getOptimumSizeToRead(sizefile,p-1,overlap);

/* If the process is not being run on parent Node
 */
if(my_rank != 0)
{
   /* Open the text file and allocate memory to docstring
    */
   fp = fopen(inFile2,"r");
   docstring = (char *)malloc
            ((optimumSize[my_rank-1]+1)*sizeof(char));
   /* If the node is first in the current node set the
     filepointer to position 0
    */
   if (my_rank == 1)
   {
      fseek(fp,0,SEEK_SET);
   }
   /* Otherwise set it to Optimum size time the node rank
    */
   else
   {
      fseek(fp,(((my_rank-1)*optimumSize[0])-(overlap)),SEEK_SET);
   }
```

```
   /* Read the text data into docstring
    */
   fgets(docstring,optimumSize[my_rank-1]+1,fp);
   /* Do the search and catch the results in Occurances array
    */
   occur = KTV(docstring,searchstring);
   /* Store the values in the message array
    */
   sprintf(message,"%ld %f",occur,end-start);
   /* Set the destination to parent node
    */
   dest=0;
   fclose(fp);
   /* Send the message array to the parent node
    */
   MPI_Send(message,strlen(message)+1,
                  MPI_CHAR,dest,tag,MPI_COMM_WORLD);
}
/* If the process is being run on parent Node
 */
else
{
   /* For all the nodes in the node set except the parent node
      recieve the message array sent by them and process them
      accordingly
    */
   for(source=1;source<p;source++)
   {
      MPI_Recv(message,1000,
                     MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
      printf("Processor with Rank : %2d %s\n",source,message);
      totaloccurances += atol(strtok(message," "));
      swatch += atof(strtok(NULL," "));
   }
   /* Stop the timer for the complete program
    */
   endmain=MPI_Wtime();
   /* Print the results to the standard output
    */
   printf("Total Occurances : %ld\n",totaloccurances);
   printf("Total Time for Searching : %f\n",swatch);
   printf("Total Time for Execution : %f\n",endmain-startmain);
```

```
    }
    /* Close the MPI environment
     */
    MPI_Finalize();
}

/* Impelementation of Function preKTV
 */
void preKTV(char *data, long int length)
{
    long int i;
    int pos;

    /* Allocate memory to the KTV2 structure variable p
     */
    p = (struct aa *)malloc(sizeof(struct aa)*length);

    /* Start building the KTV2 structure
     */
    for(i = 0;i<length;i++)
    {
        /* Get the position of the character at i
         */
        pos = data[i] - 97;

        /* Invalid character, exit the function
         */
        if(pos<0 || pos>26)
        {
            break;
        }
        /* Store the character
         */
        p[i].c = alpha[pos];
        /* Point the next pointer to NULL
         */
        p[i].next = NULL;

        /* If this character is not NULL in the postions array
           then store the address of this KTV nodes in the prev
           array
         */
```

```
        if(positions[pos].c != NULL)
        {
           prev[pos]->next = &p[i];
        }
        /* Otherwise assign the address to the corresponding
          character position index of the positions array
         */
        else
        {
           positions[pos].c = alpha[pos];
           positions[pos].next = &p[i];
        }
        /* Update the prev position
         */
        prev[pos] = &p[i];
    }
    p[i].c=NULL;
    p[i].next = NULL;
}

/* Implementation of the Function KTV
 */
long int KTV(char *data, char *pattern)
{
          struct aa *current;
          long int occurances=0;
          long int index;
    int plength = strlen(pattern);

    /* Call the preKTV function with the text and length of it
     */
    preKTV(data,strlen(data));
    /* Start the timer for the search
     */
    start = MPI_Wtime();
    /* Using the postions array go to the first incidence of the
      first character in the pattern
     */
    current = positions[pattern[0]-97].next;
          while(current != NULL && (current+plength-1)->c !=NULL)
          {
```

```
  /* check the remaining characters of the pattern with the
        adjacent KTV structures
     */
     for(index = 1; index<plength;index++)
                {
       if(pattern[index] != *(current+index)->c) break;
                }
     /* If all the characters have been matched increment the
       number of occurances of the pattern
     */
     if (index == plength) occurances++;
     /* Move to the next incidence of the first character of the
       current pattern
     */
     current = current->next;
  }
  /* Stop the timer for the search
   */
  end = MPI_Wtime();
  /* Destroy the KTV2 strucutre
   */
  free(p);
  /* Return the occurances array
   */
  return occurances;
}
```

Appendix D
KPrime Program Listing

```
// KPrime_01.cpp - Kishore R Kattamuri
// MS(CS) Thesis - Florida Tech

/* Required include files
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

/* This is the text in which the search is performed
 */
#define inFile "Chromosome01.txt"
/* This is the file which contains patterns
 */
#define serFile "patterns.txt"
/* This is the text file which is stored on the nodes
 */
#define inFile2 "/scr/Chromosome01.txt"
/* Maximum size of the pattern
 */
#define MAX_PAT_SIZE 255

/* This array will hold number of Occurances
 */
long int *totaloccurances;

/* Various variables required for timing the search
 */
double start,end,startmain,endmain,swatch;

/* prime numbers for characters with ascii code from 96 to 122
 */
int primes[]={1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,
          59,61,67,71,73,79,83,89,97,101};
```

```
/* number of patterns
 */
int pCount;
/* length of the largest pattern
 */
int pLength;
/* values of all the characters in the pattern
 */
long int **pValue;
/* following array has the first occuring pattern number
 */
int **pRowOccurances;
/* holds the multiplied values of the patterns
 */
long int *mValue;
/* temporary values holder
 */
int *tmpValue;

/* Stores the patterns
 */
char **patterns;

/* Function used to multiply the prime values of the characters
  in the pattern
 */
void multiplyValues()
{
   int i;
   int j;

   for(i=0;i<pCount;i++)
      if (tmpValue[i]==1)
                    for(j=0;j<pLength;j++)
                                mValue[j]= (i==0) ? pValue[i][j] :
                    (((mValue[j] % pValue[i][j]) == 0)
                     ? mValue[j] : (mValue[j] * pValue[i][j]));

}
```

```
/* Function to read patterns from the file passed
   as argument into the variable Patterns
 */
void readPatterns(char file[])
{

    FILE *filename = fopen(file,"r");
    int i = 0,j,k,tpLength;

    /* Allocating enough memory
     */
    patterns = (char **)malloc(sizeof(char *)*1);
    patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);

    /* Reading pattern one and storing it
     */
    fgets(patterns[i],MAX_PAT_SIZE,filename);
    patterns[i][strlen(patterns[j])-1]='\0';

    /* Intialise the pLength
     */
    pLength = strlen(patterns[i]);

    /* Reading rest of the patterns
     */
    while(!feof(filename))
    {
       i++;
       patterns = (char **)realloc(patterns,sizeof(char *)*(i+1));
       patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);
       fgets(patterns[i],MAX_PAT_SIZE,filename);
       patterns[i][strlen(patterns[i])-1]='\0';

       /* Updating pLength depending on the length of the current
          pattern read
        */
       pLength
          = (pLength < strlen(patterns[i]))
            ? strlen(patterns[i]) : pLength;
    }
    pCount = i;
    patterns[pCount]=NULL;
```

```
   fclose(filename);
}

/* Implementation of the function preKPrime
 */
long int* preKPrime()
{
   int i=0;
   int j=0;
   long int *Occurances;

   /* Allocate memory
    */
   pValue =(long int **)malloc(sizeof(long int *)*pCount);
   pRowOccurances = (int **)malloc(sizeof(int *)*27);
   Occurances = (long int *)malloc(sizeof(long int *)*pCount);
   /* Intialise the second dimension of the pRowOccurances
    */
   for(i=0;i<27;i++)
   {
      pRowOccurances[i] = (int *)malloc(sizeof(int)*pLength);
      Occurances[i]=0;
      for(j=0;j<pCount;j++)
         pRowOccurances[i][j]=0;
   }
   /* Intiailise the mValue array
    */
   mValue =(long int *)malloc(sizeof(long int)*pLength);

   /* Store the first Occurances of the characters in all the
     patterns we are searching
    */
   for(i=0;i<pCount;i++)
   {
      pValue[i] = (long int *)malloc(sizeof(long int)*pLength);
      pRowOccurances[patterns[i][0]-96][i]=1;
      for(j=0;j<pLength;j++)
      {
         pValue[i][j]=primes[patterns[i][j]-96];
                        mValue[j] = 1;
      }
   }
```

```
   return Occurances;
}

/* Implementation of the KPrime function
 */
long int* KPrime(char *str,int mode)
{
        char *mainString;
   long int i=0, j=0, k=0, tmp1=0, adjustment=0;
   long int count=0;
   long int Value,Start;
        long int stringlen=0;
   long int *occurances;

   /* Call the preKprime function
    */
   occurances = preKPrime();

   tmpValue = (int *)malloc(sizeof(int)*pCount);

   stringlen= strlen(str);

   /* if mode = 0  allocate the memory and store the
     text in the mainString
    */
   if (mode == 0)
   {
      mainString = (char *)malloc(sizeof(char)*(stringlen));
            strcpy(mainString,str);
   }
   /* else pad the aminString to make it multiple of the lagest pattern size
    */
   else
   {
      mainString = (char *)malloc(sizeof(char)*(stringlen+pLength-1));
            strcpy(mainString,str);
      for(i=0;i<(pLength-1);i++)
                  strcat(mainString,"`");
   }
        stringlen= (long int)strlen(mainString);
```

```
/* while not the end is reached continue the search
 */
      while((k+(pLength)-1) < stringlen)
      {
   for(j=0;j<pCount;j++)
                        tmpValue[j]= pRowOccurances[mainString[k]-96][j];

                for(j=0;j<pLength;j++)
                        mValue[j]=1;

                multiplyValues();

   /* Check how many patterns are active in the search
    */
   for(i=0,count=0;i<pLength && count < 2;i++)
      count += (tmpValue[i] == 1) ? 1 : 0;

                switch(count)
                {
                   /* If the patterns left is one
                    */
                  case 1:
                       for(i=0;i<pLength;i++)
                           if (mValue[i] != 1)
                               if ((mValue[i] % primes[mainString[k+i]-96]) != 0)
                                   break;
                            if (i == pLength)
                                for(i=0;i<pCount;i++)
                                    if (tmpValue[i]==1)
                                    {
                                        occurances[i] += 1;
                                        break;
                                     }
                                break;
                  /* If no patterns then go to next character
                   */
                  case 0:
                       break;
```

```
                    /* if more than one pattern is active in the search
                     */
                    default:
                        for(i=0;i<pLength-1;i++)
                        {
                            if (mValue[i] != 1)
                                if ((mValue[i] % primes[mainString[k+i]-96]) != 0)
                                    break;
                                Value = mainString[k+i+1]-96;
                                Start = i+1;
                                for(j=0;j<pCount;j++)
                                    tmpValue[j] = (tmpValue[j] == 0) ? 0
                                    : (((pValue[j][Start] ==
                                      primes[Value])|| pValue[j][Start] == 1)
                                      ? 1 : 0);
                                multiplyValues();
                        }
                        if (i == pLength-1)
                            for(i=0;i<pCount;i++)
                                if (tmpValue[i]==1)
                                    occurances[i] += 1;
                            break;

                    }
                    k++;
            }
    return occurances;
}

/* Function to get the filesize of the passed parameter file
 */
long int getFileSize(char file[])
{
    long int size;
    FILE *filename = fopen(file,"r");
    fseek(filename,0,SEEK_END);
    size = ftell(filename);
    fclose(filename);
    return size;
}
```

```
/* Function to calculate the size to be read by each node
  along with the overlap
 */
long int* getOptimumSizeToRead(long int mainFileSize,
                        int processors,
                        long int overlap)
{
   long int* optimumSize;
   long int tempSize;
   long int estimate;
   long int tmpProcs;
   int counter;

   if (processors < 1) return 0;
   tempSize = mainFileSize;
   estimate = (tempSize % processors == 0)
           ? (tempSize / processors)
           : (tempSize / processors) + 1;
   tmpProcs = tempSize / estimate;
   optimumSize = (long int *)malloc(sizeof(long int) * processors);
   tempSize = tempSize - estimate;
   optimumSize[0] = estimate;
   for(counter=1;counter<processors;counter++)
   {
     optimumSize[counter] = (counter<=tmpProcs)
                        ? estimate + overlap : 0;
   }

   return optimumSize;
}

/* Function to caluclate the overlap
 */
long int getOverlap(long int size)
{
   return size - 1;
}

/* The main function of the program
 */
main(int argc, char* argv[])
{
```

```
/* Required variables
 */
int my_rank;
int p;
int source;
int dest;
int tag=0;
char message[1000];
char num[1000];
MPI_Status status;
FILE *fp;
char *searchstring;
char *docstring;
long int occur=0;
long int sizefile;
long int overlap;
long int *optimumSize;

long int *Occurances;
int i;

/* Initialising the MPI environment variables
 */
MPI_Init(&argc,&argv);
/* Start Timer for the complete program
 */
startmain=MPI_Wtime();
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* Read the patterns and calculate filesizes
 */
readPatterns(serFile);

sizefile = pLength;
overlap = getOverlap(sizefile);
sizefile = getFileSize(inFile);

/* Estimate the optimum size for each node
 */
optimumSize = getOptimumSizeToRead(sizefile,p-1,overlap);
```

```
/* If the process is not being run on parent Node
 */
if(my_rank != 0)
{
   fp = fopen(inFile2,"r");
   docstring = (char *)malloc
            ((optimumSize[my_rank-1]+1)*sizeof(char));
   /* If the node is first in the current node set the
     filepointer to position 0
    */
   if (my_rank == 1)
   {
      fseek(fp,0,SEEK_SET);
   }
   /* Otherwise set it to Optimum size times the node rank
    */
   else
   {
      fseek(fp,(((my_rank-1)*optimumSize[0])-(overlap)),SEEK_SET);
   }
   /* Read the text data into docstring
    */
   fgets(docstring,optimumSize[my_rank-1]+1,fp);

   /* if the node is the last one in the set
    */
   if((p-1) == my_rank)
   {
      start=MPI_Wtime();
      /* Do the search in mode 1 and catch the results in Occurances array
       */
      Occurances = KPrime(docstring,1);
      end=MPI_Wtime();
   }
   /* otherwise
    */
   else
   {
      start=MPI_Wtime();
      /* Do the search in mode 0 and catch the results in Occurances array
       */
      Occurances = KPrime(docstring,0);
```

```
        end=MPI_Wtime();
      }

      /* Store the values in the message array
       */
      sprintf(message,"%ld",Occurances[0]);
      for(i=1;i<pCount;i++)
         sprintf(message,"%s %ld",message, Occurances[i]);
      /* Append the time taken to the message array
       */
      sprintf(message,"%s %f",message, end-start);
      /* Set the destination to parent node
       */
      dest=0;
      fclose(fp);
      /* Send the message array to the parent node
       */

MPI_Send(message,strlen(message)+1,MPI_CHAR,dest,tag,MPI_COMM_WORL
D);
   }
   /* If the process is being run on parent Node
    */
   else
   {
      /* Allocate memory to the totaloccurances array
       */
      totaloccurances = (long int *)malloc(sizeof(long int)*pCount);
      /* Initialise all the values to 0
       */
      for(i=0;i<pCount;i++)
         totaloccurances[i] = 0;

      /* For all the nodes in the node set except the parent node
        recieve the message array sent by them and process them
        accordingly
       */
      for(source=1;source<p;source++)
      {
         MPI_Recv(message,1000,
                        MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
         printf("Processor with Rank : %2d %s\n",source,message);
```

```
            /* Update the total occurances
             */
            totaloccurances[0] += atol(strtok(message," "));
            for(i=1;i<pCount;i++)
              totaloccurances[i] += atol(strtok(NULL," "));
            /* Store the time taken in swatch
             */
            swatch += atof(strtok(NULL," "));
        }
        /* Stop the timer for the complete program
         */
        endmain=MPI_Wtime();
        /* Print the results to the standard output
         */
        for(i=0;i<pCount;i++)
            printf("Total Occurances of %s : %ld\n",
                  patterns[i],totaloccurances[i]);
        printf("Total Time for Searching : %f\n",swatch);
        printf("Total Time for Execution : %f\n",endmain-startmain);
    }
    /* Close the MPI environment
     */
    MPI_Finalize();
}
```

Appendix E
KTV2 Program Listing

```
// KTV2_01.cpp - Kishore R Kattamuri
// MS(CS) Thesis - Florida Tech

/* Required include files
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <math.h>
#include "mpi.h"

/* This is the text in which the search is performed
 */
#define inFile "Chromosome01.txt"
/* This is the file which contains patterns
 */
#define serFile "patterns.txt"
/* This is the text file which is stored on the nodes
 */
#define inFile2 "/scr/Chromosome01.txt"
/* Maximum size of the pattern
 */
#define MAX_PAT_SIZE 255

/* Predeclaration of the preKTV
 */
void preKTV2(char *, long int);

/* Predeclaration of the KTV2
 */
long int KTV2(char *, char *);
```

```
/* Definition of KTV2 structure
 */
struct aa
{
          char *c;
   struct aa *next;
};

/* The KTV2 structure array which is built in the preKTV
 */
struct aa *p;

/* Positions array
 */
struct aa positions[26];

/* Temporary array to hold the previous occurance of
  the character
 */
struct aa *prev[26];

/* Alpha array which holds all the available characters
  in the text under tes
 */
char *alpha[26]={
    "a","b","c","d",NULL,NULL,"g","h",NULL,NULL,"k",
    NULL,"m","n",NULL,NULL,NULL,"r","s","t","u","v",
    "w",NULL,"y",NULL};

/* This array will hold number of Occurances
 */
long int *totaloccurances;

/* Various variables required for timing the search
 */
double start,end,startmain,endmain,swatch;

/* Number of patterns used for search
 */
int pCount;
```

```
/* Patterns used for search
 */
char **patterns;

/* Maximum pattern length read
 */
int pLength;

/* Function to read patterns from the file passed
   as argument into the variable Patterns
 */
void readPatterns(char file[])
{

    FILE *filename = fopen(file,"r");
    int i = 0,j,k,tpLength;

    /* Allocating enough memory
     */
    patterns = (char **)malloc(sizeof(char *)*1);
    patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);

    /* Reading pattern one and storing it
     */
    fgets(patterns[i],MAX_PAT_SIZE,filename);
    patterns[i][strlen(patterns[j])-1]='\0';

    /* Intialise the pLength
     */
    pLength = strlen(patterns[i]);

    /* Reading rest of the patterns
     */
    while(!feof(filename))
    {
       i++;
       patterns = (char **)realloc(patterns,sizeof(char *)*(i+1));
       patterns[i] = (char *)malloc(sizeof(char)*MAX_PAT_SIZE);
       fgets(patterns[i],MAX_PAT_SIZE,filename);
       patterns[i][strlen(patterns[i])-1]='\0';
```

```
        /* Updating pLength depending on the length of the current
           pattern read
         */
        pLength
          = (pLength < strlen(patterns[i]))
            ? strlen(patterns[i]) : pLength;
    }
    pCount = i;
    patterns[pCount]=NULL;
    fclose(filename);
}

/* Function to get the filesize of the passed parameter file
 */
long int getFileSize(char file[])
{
    long int size;
    FILE *filename = fopen(file,"r");
    fseek(filename,0,SEEK_END);
    size = ftell(filename);
    fclose(filename);
    return size;
}

/* Function to calculate the size to be read by each node
   along with the overlap
 */
long int* getOptimumSizeToRead(long int mainFileSize,
                               int processors,
                               long int overlap)
{
    long int* optimumSize;
    long int tempSize;
    long int estimate;
    long int tmpProcs;
    int counter;

    if (processors < 1) return 0;
    tempSize = mainFileSize;
    estimate = (tempSize % processors == 0)
             ? (tempSize / processors)
             : (tempSize / processors) + 1;
```

```
   tmpProcs = tempSize / estimate;
   optimumSize = (long int *)malloc(sizeof(long int) * processors);
   tempSize = tempSize - estimate;
   optimumSize[0] = estimate;
   for(counter=1;counter<processors;counter++)
   {
     optimumSize[counter] = (counter<=tmpProcs) ? estimate + overlap : 0;
   }

   return optimumSize;
}

/* Function to caluclate the overlap
 */
long int getOverlap(long int size)
{
   return size - 1;
}

/* The main function of the program
 */
main(int argc, char* argv[])
{
   /* Required variables
    */
   int my_rank;
   int p;
   int source;
   int dest;
   int tag=0;
   char message[1000];
   MPI_Status status;
   FILE *fp;
   char *searchstring;
   char *docstring;
   long int occur=0;
   long int sizefile;
   long int overlap;
   long int *optimumSize;
   long int *Occurances;
```

```
/* Initialising the MPI environment
 */
MPI_Init(&argc,&argv);
/* Start Timer for the complete program
 */
startmain=MPI_Wtime();
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);

/* Read the patterns and calculate filesizes
 */
readPatterns(serFile);
sizefile = pLength;
overlap = getOverlap(sizefile);
sizefile = getFileSize(inFile);

/* Estimate the optimum size for each node
 */
optimumSize = getOptimumSizeToRead(sizefile,p-1,overlap);

/* If the process is not being run on parent Node
 */
if(my_rank != 0)
{
   /* Open the text file and allocate memory to docstring
    */
   fp = fopen(inFile2,"r");
   docstring = (char *)malloc ((optimumSize[my_rank-1]+1)*sizeof(char));
   /* If the node is first in the current node set the
      filepointer to position 0
    */
   if (my_rank == 1)
   {
      fseek(fp,0,SEEK_SET);
   }
   /* Otherwise set it to Optimum size time the node rank
    */
   else
   {
      fseek(fp,(((my_rank-1)*optimumSize[0])-(overlap)),SEEK_SET);
   }
```

```
   /* Read the text data into docstring
    */
   fgets(docstring,optimumSize[my_rank-1]+1,fp);
   /* Do the search and catch the results in Occurances array
    */
   Occurances = KTV2(docstring);
   /* Store the values in the message array
    */
   sprintf(message,"%ld",Occurances[0]);
   for(i=1;i<pCount;i++)
      sprintf(message,"%s %ld",message, Occurances[i]);
   /* Append the time taken to the message array
    */
   sprintf(message,"%s %f",message, end-start);
   /* Set the destination to parent node
    */
   dest=0;
   fclose(fp);
   /* Send the message array to the parent node
    */
  MPI_Send(message,strlen(message)+1,
                MPI_CHAR,dest,tag,MPI_COMM_WORLD);
}
/* If the process is being run on parent Node
 */
else
{
   /* Allocate memory to the totaloccurances array
    */
   totaloccurances = (long int *)malloc(sizeof(long int)*pCount);
   /* Initialise all the values to 0
    */
   for(i=0;i<pCount;i++)
      totaloccurances[i] = 0;
   /* For all the nodes in the node set except the parent node
     recieve the message array sent by them and process them
     accordingly
    */
   for(source=1;source<p;source++)
   {
      MPI_Recv(message,1000,
      MPI_CHAR,source,tag,MPI_COMM_WORLD,&status);
```

```
            printf("Processor with Rank : %2d %s\n",source,message);

            /* Update the total occurances
             */
            totaloccurances[0] += atol(strtok(message," "));
            for(i=1;i<pCount;i++)
              totaloccurances[i] += atol(strtok(NULL," "));
            /* Store the time taken in swatch
             */
            swatch += atof(strtok(NULL," "));
        }
        /* Stop the timer for the complete program
         */
        endmain=MPI_Wtime();
        /* Print the results to the standard output
         */
        for(i=0;i<pCount;i++)
            printf("Total Occurances of %s : %ld\n",
                  patterns[i],totaloccurances[i]);
        printf("Total Time for Searching : %f\n",swatch);
        printf("Total Time for Execution : %f\n",endmain-startmain);
    }
    /* Close the MPI environment
     */
    MPI_Finalize();
}

/* Impelementation of Function preKTV2
 */
void preKTV2(char *data, long int length)
{
    long int i;
    int pos;

    /* Allocate memory to the KTV2 structure variable p
     */
    p = (struct aa *)malloc(sizeof(struct aa)*length);

    /* Start building the KTV2 structure
     */
    for(i = 0;i<length;i++)
    {
```

```
        /* Get the position of the character at i
         */
        pos = data[i] - 97;

        /* Invalid character, exit the function
         */
        if(pos<0 || pos>26)
        {
           break;
        }
        /* Store the character
         */
        p[i].c = alpha[pos];
        /* Point the next pointer to NULL
         */
        p[i].next = NULL;

        /* If this character is not NULL in the postions array
           then store the address of this KTV nodes in the prev
           array
         */
        if(positions[pos].c != NULL)
        {
           prev[pos]->next = &p[i];
        }
        /* Otherwise assign the address to the corresponding
           character position index of the positions array
         */
        else
        {
           positions[pos].c = alpha[pos];
           positions[pos].next = &p[i];
        }
        /* Update the prev position
         */
        prev[pos] = &p[i];
    }
    p[i].c=NULL;
    p[i].next = NULL;
}
/* Implementation of the Function KTV2
 */
```

```
long int KTV2(char *data, char **pattern)
{
   struct aa *current;
   long int *occurances;
   long int index;
   int plength;

   /* Call the preKTV2 function with the text and length of it
    */
   preKTV(data,strlen(data));
   /* Start the timer for the search
    */
   start = MPI_Wtime();
   /* Allocate memory to the occurances array
    */
   occurances =(long int *)malloc(sizeof(long int)*pCount);
   /* For each pattern in the patterns array do the search
    */
   for(i=0;i<pCount;i++)
   {
      occurances[i]=0;
      plength = strlen(pattern[i]);
      /* Using the postions array go to the first incidence of the
        first character in the current pattern
       */
      current = positions[pattern[i][0]-97].next;

      /* while the current is not NULL and not reached the end of the
        KTV structure
       */
      while(current != NULL && (current+plength-1)->c !=NULL)
      {
         /* check the remaining characters of the pattern with the
           adjacent KTV structures
          */
         for(index = 1; index<plength;index++)
         {
            pattern[i][index] != *(current+index)->c) break;
         }
         /* If all the characters have been matched increment the
           number of occurances of the current pattern
          */
```

```
            if (index == plength)
               occurances[i]=occurances[i]+1;
            /* Move to the next incidence of the first character of the
              current pattern
             */
            current = current->next;
          }
       }
       /* Stop the timer for the search
        */
       end = MPI_Wtime();
       /* Destroy the KTV2 strucutre
        */
       free(p);
       /* Return the occurances array
        */
       return occurances;
}
```