Parallel String Matching Algorithm

Implementation using OpenMP and MPI

Sindhuri Rayavaram
UC, Irvine
California, US
srayavar@uci.edu

Ramkumar Rajabaskaran
UC, Irvine
California, US
rrajabas@uci.edu

***Abstract*—This project implements the famous Boyer-Moore Horspool string matching algorithm in parallel using OpenMP and MPI. The Boyer-Moore-Horspool algorithm is widely used in string matching of genetics related data. The time complexity greatly reduces with the increase of the length of the pattern being matched. The object of the project is to a) Implement the sequential Boyer-Moore Horspool algorithm and record the time taken, b) Implement the algorithm in parallel using OpenMP and MPI, c) Provide analysis of the speed up produced and bottlenecks. In conclusion, we found that the parallel algorithm implementation of both OpenMP and MPI is better than the sequential algorithm.**

Keywords—Boyer-moore Horspool algorithm;string matching; parallel algorithm; OpenMP; MPI

I. INTRODUCTION

String matching are an important class of algorithms to find one or more occurrences of a pattern in a text. The native algorithm tries to match each letter in the pattern to the text, when a mismatch occurs, the pattern is shifted by one letter and algorithm continues. This algorithm is highly inefficient. The Boyer-Moore-Horspool algorithm was proposed in 1980, is an improvement over the Boyer-Moore algorithm. This preprocesses the pattern present and the shifting of the pattern happens according to the values obtained from the preprocessing table. The parallel implementation of the string matching algorithms greatly reduces the time taken for searching the total number of occurrences. The speedup produced for the OpenMP implementation was more than 10 times when compared to the sequential algorithm. The MPI performance is better on less processors with a bottleneck in reading the file.

The paper is organized as follows. After the introduction section, section 2 contains information regarding the Boyer-Moore-Horspool sequential algorithm explanation and implementation. Section 3 contains the parallel implementation of the algorithm using OpenMP and analysis of the running time. Section 4 contains implementation of the parallel algorithm in MPI and analysis. Section 5 contains the conclusion and the performance achieved.

II. BOYER MOORE HORSPOOL ALGORITHM

The Boyer-Moore-Horspool algorithm observes that when a mismatch occurs at a position then instead of just moving the pattern by one word, we can move by the number of letters already seen and saving the time. The algorithm proposes the construction of a Bad match table.

Bad character Rule:

When a mismatch occurs in between the pattern and the text, the character at which the mismatch occurred is a bad character and the number of places of the pattern to be shifted to match with the text is found out and is shifted accordingly.
A bad match character table is formed using the characters in the pattern. When a mismatch occurs, the value of the character of the text in the Bad match table is found and the pattern is shifted accordingly.

Bad Match Table:

There are so many methods to form the bad match table. Generally, the table is indexed by the letter of the pattern and the corresponding value is found using the formula *length-index-1*. The index here is the position of the letter in the pattern. The last character value is equal to the value of the length of the string if not already defined. All other characters have a value equal to the length of the string. Whenever a mismatch occurs the corresponding value of the letter from the text is seen from the bad match table and the pattern is shifted accordingly. Here we are finding out by how many letters the pattern should be shifted. The look up into the table usually takes O (1) time.

Searching:

The pattern is searched from the right to the left. Whenever a mismatch occurs, the value of the letter is found from the table and the pattern is shifted accordingly. The worst case running time is usually quadratic $O(n * m)$. The best and average case running times are much better and is usually $\Omega(n/m)$, where n is length of the text and m is the length of the pattern.

Example:

Suppose the Pattern is ‘EATER’ and text is ‘PETERTHEANTEATER’. In the preprocessing stage, the Bad match table will be,

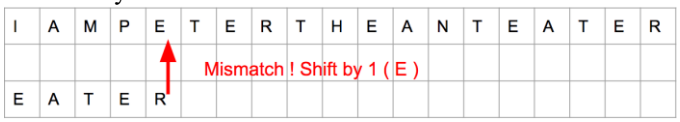
Letter	E	A	T	R
Index	0,3	1	2	4
Value	1	3	2	5

Figure 1. Bad Match table

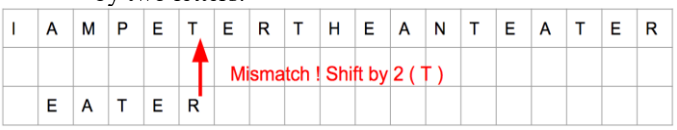
In the Figure 1, the Bad match table, the value for the letters in the pattern are found using the formula given above. The last letter takes the value of the length of the pattern if not mentioned *already*. The value of the remaining letters is equal to the length of the pattern.

While searching, the pattern is matched is from right to left. The following figures illustrate how we can search for a substring in a text.

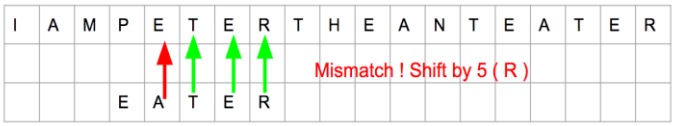
1. The letter 'R' is matched with 'E' in the text. A mismatch occurs and the value for 'E' is found from the Bad match table i.e. is '1'. The pattern is shifted by one letter.



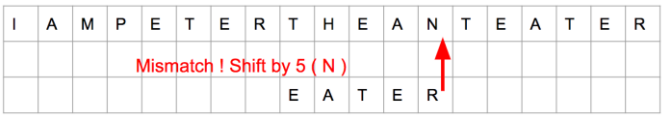
2. The letter 'R' is matched with 'T' in the text. A mismatch occurs and the value for 'T' is found from the Bad match table i.e. is '2'. The pattern is shifted by two letters.



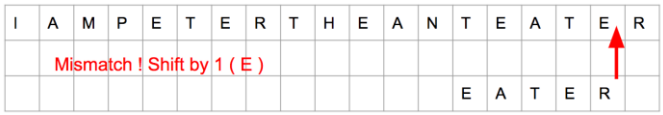
3. The letter 'R' is matched with 'R' in the text. It's a match. The next two letters are matched and a mismatch occurs at the letter 'A'. The value of 'R' is found from the table and the pattern is shifted by 5.



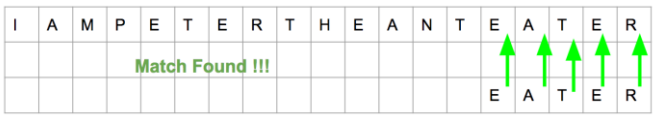
4. The letter 'R' is matched with 'N' in the text. A mismatch occurs and the value for 'N' is found from the Bad match table i.e. is '5'. The pattern is shifted by five letters.



5. The letter 'R' is matched with 'E' in the text. A mismatch occurs and the value for 'E' is found from the Bad match table i.e. is '1'. The pattern is shifted by one letter.



6. All the letters in the pattern match with the text.



Analysis:

The worst case running time of the algorithm takes $O(n * m)$ time where 'n' is the size of the text and 'm' is the size of the pattern. The Best case occurs when the pattern is present at the end of the text. The time taken in this scenario is $\Omega(n/m)$. The Average case time is better when it comes to this algorithm is $O(n/m)$.

Sequential	
Length	Time
50	2.35282s
20	2.40953s
10	2.43091s
7	2.5389s
3	3.24121s

Table 1. Running time of sequential algorithm on different pattern lengths

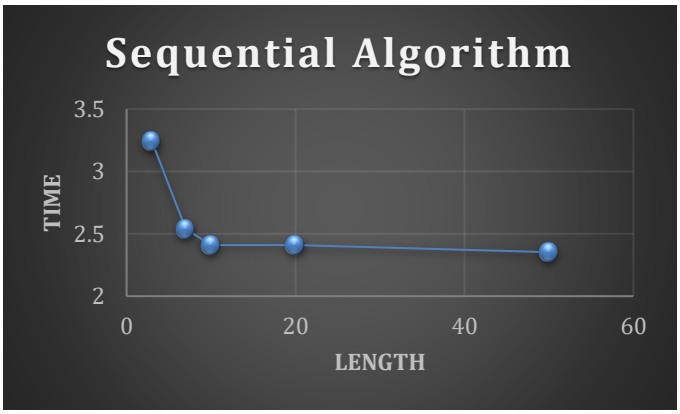


Figure 2 Chart diagram showing variation of running times in sequential algorithm

III. PARALLEL IMPLEMENTATION USIN OPENMP

OpenMP is an application programming interface (API) that facilitates the multi-programming on multiple platforms. We are using C++ to code using OpenMP.

Parallel algorithm:
The create-table function is used to create the Bad match table. The text is divided into blocks based on the number of threads available. Each thread works on a block. Based on the ID of the thread, the start and the end points are specified for the data to read from the input text. Each thread reads the data separately and performs the string matching on it.

```
#pragma omp parallel shared(str,str_len,tab1) num_threads(num)
{
        FILE *in;
        in = fopen(filename,"r");
        int tid = omp_get_thread_num();
        string text;
        char buf[size];
        int temp = 0;
        fseek(in,(size*tid)+str_len-1,SEEK_SET);
        fgets(buf,size,in);
        text = buf;
        int rem = size - text.length();
        int pos = ftell(in);
        while((pos<=size*(tid+1))&&rem>0)
          {
            fgets(buf,size-text.length(),in);
            text+=buf;
            rem = size - text.length();
            pos = ftell(in);
          }
        size_t text_len = text.length();
        fclose(in);
        transform(text.begin(),text.end(),text.begin(), ::tolower);
        temp = boyer_moore_horsepool_sequential(&text[0],text_len,str,str_len,tab1);
        #pragma omp atomic
        count+=temp;
}
```

Figure 3 Code sample showing the parallel implementation of the algorithm in OpenMP

To avoid the cases where the pattern to be matched is cut in between the blocks, each thread is given an offset equal to pattern-length-1. There is a chance where the thread might not read the entire block specified by the limits. We run a while loop till the thread reads the entire data assigned to the thread.

The threads are sharing the table created and the Pattern to be matched. The value of the count is maintained to get the count of the occurrences of the pattern in the text. The "pragma omp reduce" is used to get the sum of the counts from all the threads.

Analysis:
The OpenMP code is run on the Queue "Class64-amd" on a HPC cluster. The cluster has 1 nodes with 64 cores per node. The analysis done by changing the number of threads and the number of processors.

OpenMP (64 Threads)	
Length	Time
3	0.194489s
7	0.17152s
10	0.162964s
20	0.159753s
50	0.157213s

Table 2. Time taken by OpenMP for various pattern lengths

As the length of the pattern increases, the time taken by the OpenMP program decreases. As the time taken is inversely proportional to the length of the pattern, this conclusion is proved.

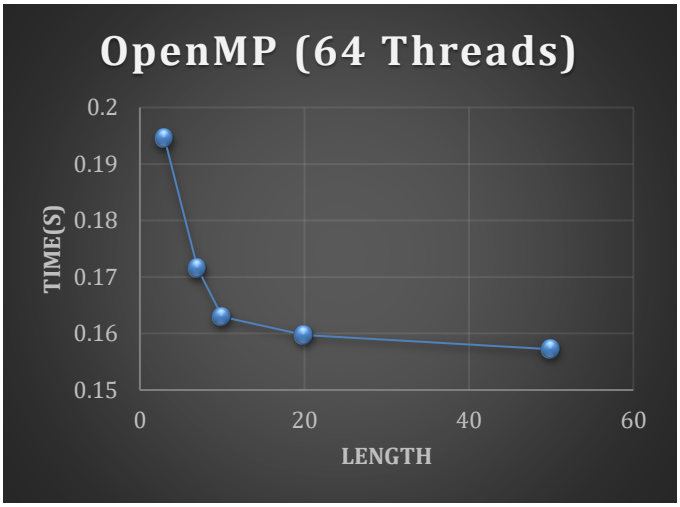


Figure 4 Chart showing variation of time with the length of the pattern

We have taken the OpenMP code and testes it on various threads. As the number of threads are reduced the time taken by the OpenMP code increases.

OpenMP (7 Character Pattern)	
Threads	Time
64	0.17152s
32	0.250249s
16	0.334387s
8	0.609418s

Table 3. Time taken by OpenMP for various thread

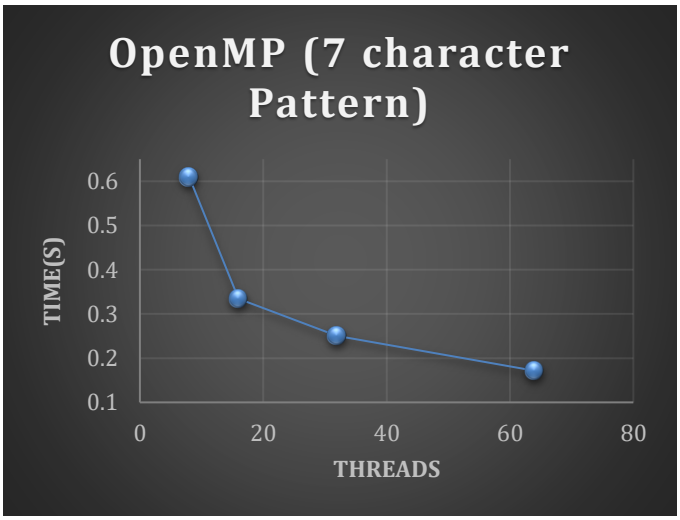


Figure 5 Graph showing the variation of time with the threads

The Speed up is calculated by taking the time taken by the sequential code and dividing it by the OpenMP code. As the number of threads increases, the speed up increases.

Speedup (7-character Pattern)	
50	14.96581072
20	15.08284664
10	14.79412631
7	14.80235541
3	16.66526127

Table 4. Speedup on various character lengths

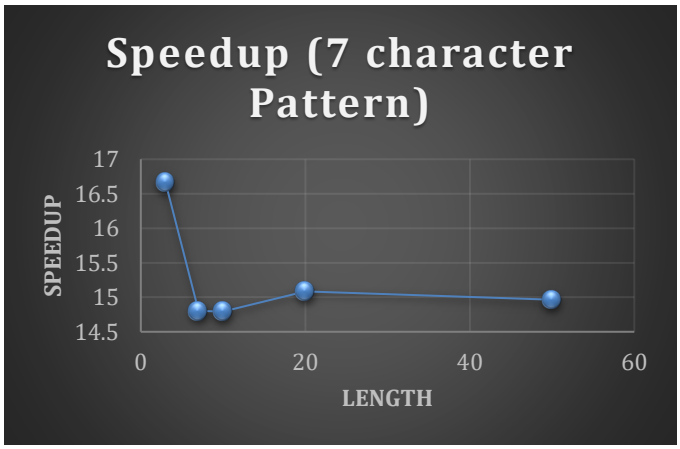


Figure 7 Graph showing Speedup on various lengths

IV. PARALLEL IMPLEMENTATION USING MPI

MPI is a message-passing system that is designed to function on a parallel-computing architecture. The code is done in C++. On a 64-processor machine, each processor takes a block of data in the text and does the computation that. 'MPI_File' File handle is given to each process. MPI_File_reat_at_all is used to read the data simultaneously from the file. This is a blocking read and waits for the other processes to finish the reading. Once every processor finishes its work, to get the total count of the occurrences, MPI_Reduce is used. Essentially, we are dividing the text between the various processes.

Analysis:

The MPI code has the bottleneck in reading the data from the same file. Though different collectives are used, the speed up was not as high as the OpenMP. The computation alone is finishing much better than the sequential time.

```
//Reading data of file into blocks
MPI_File_read_at(file, start, block, blocksize, MPI_CHAR, MPI_STATUS_IGNORE);

block[blocksize] = '\0';

    if(rank == 0){
       cout<<"String '"<<str<<"' is being searched in file '"<<filename<<"'"<<endl;
}

    //cout<<"Length of file "<<filesize<<" is divided into blocksize "<<blocksize<<" for node "<<rank<<endl;

    // Making pointer point to start of block
    text =&block[0];

    while(text_pos <= (blocksize - str_len ))
      {
            // Selecting charachter at position equal to pattern length . -1 is to nullify
            // the 0th element.
            occ_char = text[text_pos + str_len -1];

            // If last charachter of pattern matches curent character in text and
            // if charachters ahead of current in text contains pattern , we have a match.
            // memcmp compares charachters ahead of current with the pattern in blocks of
            // str_len -1.

            if (occ_char == str[str_len-1] && (memcmp(str, text+text_pos, str_len - 1) == 0))
            {
              temp_count++;
            }

            // Look at occurrence table and decide how to increment text pointer
            text_pos += occ1[occ_char];
}
// Adding the tempsum of each thread to the total sum using mpi_reduce

MPI_Reduce(&temp_count,&final_count,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
```

Figure 6 Code sample for MPI

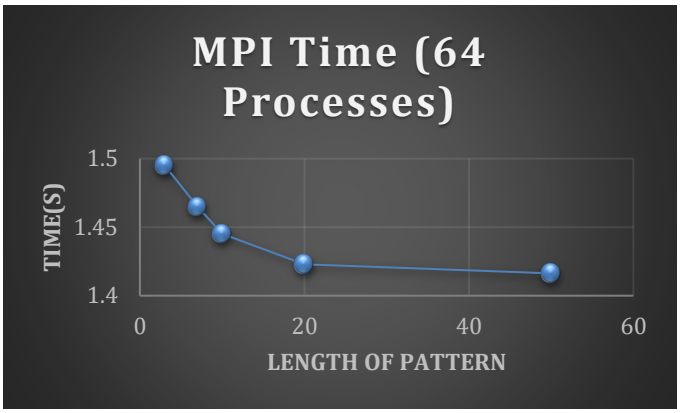


Figure 8 Graph showing MPI Time for various pattern lengths

MPI (7-character Pattern)	
Processes	Time
64	1.46486s
32	1.29218s
16	1.21587s
8	1.23072s

Table 5. Time taken by MPI execution on various processors

MPI (64 Processes)	
Length	Time
3	1.49516s
7	1.46486s
10	1.44504s
20	1.4228s
50	*1.4163s*

Table 6. Time taken by MPI for various Pattern lengths

This code is executed on a queue "class64-amd" on a HPC cluster. This queue has 1 node with 64 processors. As the number of processors are reduced, the speedup increased in MPI. This might be because of the blocking read collective. The Speed up is calculated by dividing the time taken by the sequential time and the time taken by the MPI code. The speed up is less than the OpenMP. As the length of the pattern increased, the time taken by the MPI slightly reduced.

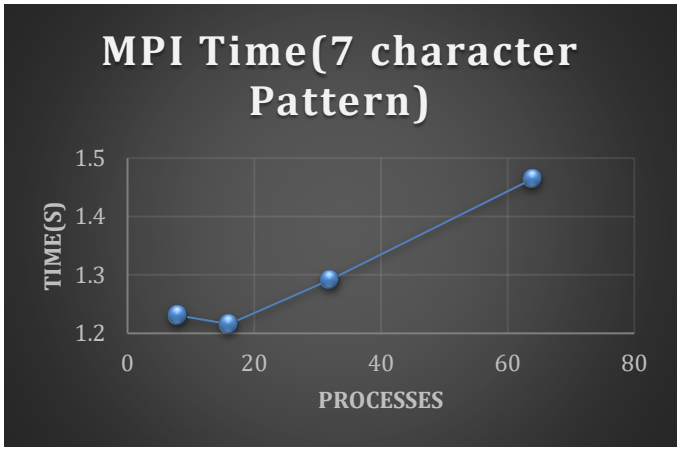


Figure 9 Graph showing MPI Time for 7-character pattern length

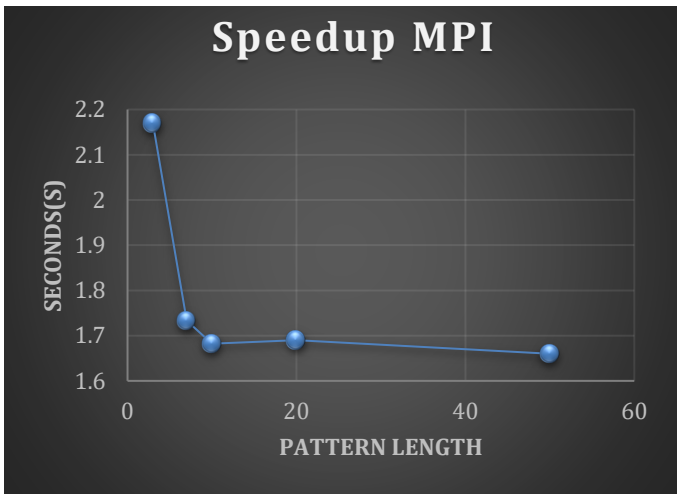


Figure 10 Graph showing Speed MPI for various Pattern lengths

Speedup	
Length	Speedup
50	1.66
20	1.69
10	1.682
7	1.733
3	2.17

Table 7. Speedup obtained for various pattern lengths

V. CONCLUSION:

The parallel algorithm showed sufficient speed up on both OpenMP and MPI. The OpenMP implementation showed better speedup than the MPI. With better architecture and much larger texts, parallel is proved to be much faster than the sequential implementation of the algorithm.

VI. REFERENCES:

[1] Average running time of the Boyer-More-Horspool algorithm
Ricardo A. Baeza-Yates

[2] A new approach to text searching
Ricardo A. Baeza-Yates, Gaston H. Gonnet

[3] https://en.wikipedia.org/wiki/Message_Passing_Interface

[4] https://en.wikipedia.org/wiki/OpenMP

[5] OpenMPI: Goals, Concept, and Design of a next generation MPI Implementation

[6] OpenMP: An industry standard API for shared memory programming.