Implementation of Rabin Karp String Matching Algorithm Using MPI

Nupur Kohli
Computer Science department
San Jose State University
San Jose, CA 95112
858-564-3210
nupurkohli2014@gmail.com

Rushikesh Joshi
Computer Science Department
San Jose State University
San Jose, CA 95112
408-618-9632
rushikesh.joshi@sjsu.edu

ABSTRACT

Searching for occurrences of string patterns is a common problem in many applications. One of the easiest approaches is to search a pattern in a text character by character. But this method becomes very slow when we deal with long patterns. Various good solutions like Naïve string Matcher, Knuth-Morris-Pratt (KMP), Rabin Karp (RK), etc have already been presented for string matching and we selected Rabin Karp for this project. However, the main focus of this project is to apply the concepts of parallelism to further improve the performance of the algorithm. There are lots of parallel processing languages and API's available like OpenMP, MPI, MapReduce, etc and we chose MPI and OpenMP to achieve parallelism for this project. We started with analyzing the algorithm in serial mode to get a better understanding of the nature of the algorithm and then further design it for achieving parallelism. Later comparing the results of both serial and parallel implementation gave us insights into how performance and efficiency is achieved through various techniques of Parallelism.

1. INTRODUCTION

We used **C** programming language to implement Rabin Karp algorithm in both serial and parallel modes. We implemented this algorithm by designing an efficient Rolling Hash Function by selecting unique base and mod values. First we implemented the algorithm using Serial mode. Later we used 3 parallelization techniques for Rabin Karp algorithm. Dividing the string into various sub strings and allocating each thread to work on independent part of string was our approach for all parallelization techniques. First one is Forking technique. Second approach is achieving parallelism using Message Passing Interface (MPI) technique and the third one is OpenMP technique. Each technique that we used gave us unique behavior in terms of performance of the algorithm. We plan to discuss all our observations in the sections below.

2. String Matching Algorithm

In computer science, pattern matching is the act of checking a given sequence of tokens for the presence of the constituents of some pattern In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence.

String matching algorithm is nowadays used in various fields like Genetic computing where the algorithm is used in looking for similarities of two or more DNA sequences. It is also applied to the problem of plagiarism detection where documents are being compared based on these string matching algorithms and find count of matched patterns. In other words, this algorithm deals with matching patterns between various texts. The pattern here can be single pattern or multi pattern. The Naive string matching approach searches this pattern character by character. For example suppose we have a string A of say 'm' characters and another string B of 'n' characters , each of the m characters are searched though length(B) one by one. This generates a time complexity of O (m*(m-n-1)). This becomes very high when we talk about large DNA sequences used in genome projects or large documents in detecting plagiarism. Thus other techniques like Rabin Karp, Knuth-Morris-Pratt, etc are used to enhance the performance of native string matching algorithm. We will discuss about Rabin Karp algorithm in next section.

Pseudo code for Naïve String Matching

```
for i := 0 to len(B)-len(A) {
 match =true
 for j := 0 to len(A) {
 if A[j] != B[i+j]
  match =false
}
  if match ==true
  R.append(i)
}
```

3. Rabin Karp String Matching

As we observed above, the time complexity of naïve string matching algorithm is *O(m*(m-n-1)).*The running time becomes very large in case of bigger values of lengths of String A and String B.

To improve the performance of above algorithm, there is another algorithm called Rabin Karp algorithm which has a Linear Time complexity of *O(m-n-1).* It uses the technique of hashing.

Let us see the working of Rabin Karp Algorithm in detail.

The Rabin-Karp algorithm groups characters into substrings, produces a unique value for the substring and compares these unique values. This unique value is called a Hash Value. Hash values are computed using appropriate Hash functions. The algorithm exploits the fact that if two strings are equal, their hash values are also equal. Thus, all we have to do is compute the hash value of the substring we're searching for, and then look for a substring with the same hash value. The key to the Rabin–Karp algorithm's performance is the efficient computation of hash values of the successive substrings of the text. One popular and effective hash function treats every substring as a number in some base which is usually a prime number.

Pseudo code:

```
FunctionRabinKarp(string s[1..n], string
pattern[1..m])
hpattern := hash(pattern[1..m]);
hs := hash(s[1..m])
for i from 1 to n-m+1
  if hs = hpattern
    if s[i..i+m-1] = pattern[1..m]
      return i
  hs := hash(s[i+1..i+m])
return not found
```

The above pseudo code calculates the new hashing value each time it does not find the substring ‘pattern’ in main string ‘S’.

```
  hs := hash(s[i+1..i+m])
```

Following hash function is used in Rabin Karp Implementation to find first hash value of main string.

$C_1 * B^{k-1} + C_2 * B^{k-2} * \ldots + C_{k-1} * B + C_k$

where $C_{1,2....k}$ represents ascii value of kth character in the main string,

B represents large prime base selected by programmer.

The above hashing a string of ‘m’ characters takes running time of *O(m)* because it traverses the string ‘S’ character by character to compute hash value. This adds extra cost to running time and is equivalent to naïve string matching algorithm we discussed above. To overcome this problem, the concept of rolling hashing comes into picture. We will discuss about Rolling hashing in next section

4. Rolling Hashing

This is a technique that computes new hash values based on old hash values. This way time complexity reduces because it does not iterate through each character to add their numeric values.

The following formula explains why:

```
s[i+1..i+m]=s[i..i+m-1]-s[i]+s[i+m]
```

It subtracts the ascii value of character at position ‘i’ and adds the ascii value of character at position ‘i+m’. Lets explain this with an example:

Suppose we have text "abracadabra" and we are searching for a pattern of length 3, the hash of the first substring, "abr", using 101 as base is:

Hash("abr") = (97 × 101^2) + (98 × 101^1) + (114 × 101^0) = 999,509

Now we compute next substring:

Hash("bra") = [101 × (999,509 - (97 × 101^2))] + (97 × 101^0) = 1,011,309

This way Rolling hashing improves the performance of the algorithm remarkably.

Refer to figure 1 for more details of Rolling Hashing technique.

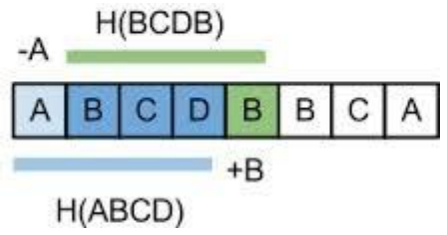

Figure 1: Rolling Hashing technique

5. Parallelism through various techniques

We started with implementing serial mode of Rabin Karp Algorithm with Rolling Hashing technique. In the later sections, we tend to show the speedup ratios between serial and parallel modes. Lets discuss the various parallelization techniques that we followed.

5.1 Fork

C programming language provides very efficient API’s for achieving parallelism. One of these functions that we used is *fork()* API.

The system call fork( ) is called without any arguments and returns an integer process identification number (pid). It causes the OS kernel to create a new process which is an exact duplicate of the calling process. The new process is termed to be a child of the parent process. The new child process is an exact clone of the parent. It has the same data and variable values as the parent at the time fork was executed. It even shares the same file descriptors as the parent. The child process does not start its execution from the first instruction in the source code, but continues with the next statement after the call to fork. That is, after the call, the parent process and its newly created offspring execute concurrently with both processes resuming execution at the statement immediately after the call to fork.

In order to distinguish between parent and child process , the only way is to have each process immediately examine the return value of the fork call. In the parent, a successful fork returns the process identifier (PID) of the new child. The pid is set to a unique, non-zero, positive integer identifying the newly created child process for the parent. In the child, fork returns a nominal value of 0. The value of the pid enables a programmer to distinguish a child from its parent and to specify different actions for the two processes, usually via an IF or CASE statement. A process can obtain its own pid and that of its parent using the getpid( ) and getppid( ) system calls respectively. The typical method of spawning processes is as follows. The main (parent) program executes a fork. If the fork is successful, each process must now determine its identity (parent or child) by checking the value returned by fork. Then, a branch in execution paths occurs as a function of the process type (parent or child) through a simple test of the return value of the fork system call.

Refer to Figure 2 for detailed view:

In a multiprogrammed system where several processes can be active concurrently, kernel functions are needed to enable user processes to create (fork) other processes coordinate (via wait and sleep) their execution sequences, and communicate with (signal) each other. Processes may also need to be aware of time, and react to time-based events.

The role of interprocess communication (IPC) is twofold. First, it transmits information which may be necessary between processes. Second, it provides a capability for process synchronization. Although processes can always pass information through named files, facilities are provided to enable the processes to communicate more directly.

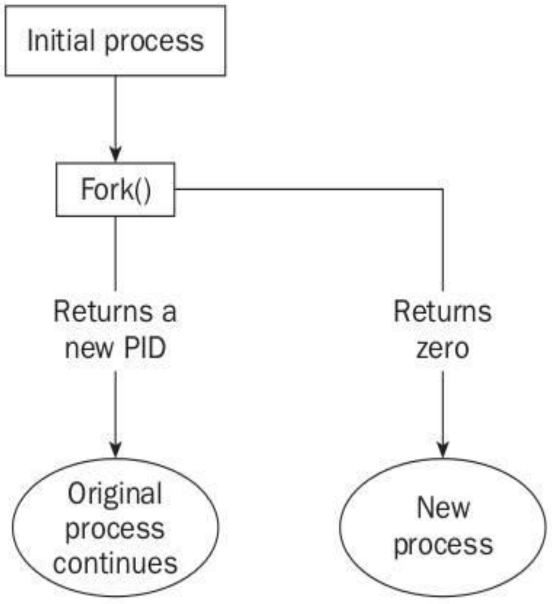

Figure 2: Fork Technique

The approach that we followed is : We created a child process out of our parent process. The main design approach we followed was to divide the main String 'S' in to two equal parts. These 2 subparts are concurrently searched through by each of our process. Thus same text which was earlier being searched by 1 process is now being searched by 2 processes concurrently. Both the processes then use pipes to read and write their share of count of pattern they found. Later we do summation of the count variables from each of these processes. We found out this count variable to match the count that we found in our serial implementation and the timing also reduced significantly. We would like to point out here that we dealt with creation two threads here.

Design Flow(refer figure 3):

5.2 MPI (Message Passing Interface)

MPI stands for "Message Passing Interface". As the name suggests it is used for passing messages. MPI represents message-passing parallel programming model. It mainly moves data from one process to another process with the co-operation of each process. Distributed Architecture was the main reason for invention of MPI programming model. It was designed with the concept of distributed architecture. You can see general architecture of programming model for MPI in given figure*:*

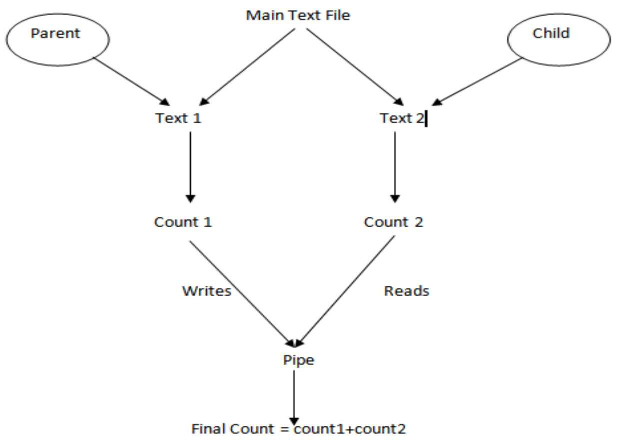

Figure 3: Design flowchart for parallelism through fork()

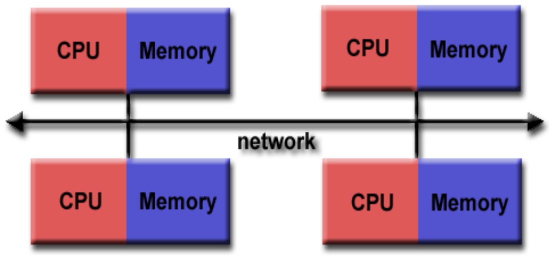

Figure 4: Programming Model For MPI in Distributed Architecture.

MPI libraries are available for different programming languages. There are numerous compilers available in the market for MPI programming model. We have used Open-MPI in our project in order to apply parallelism in Rabin-Karp algorithm.

5.2.1 Basic MPI communication routines

There are few basic MPI communication routines which we will be using for our Rabin-Karp algorithm. The basic two routines which are important to understand to work with MPI are given below:

- MPI_Send
- MPI_Recv

5.2.1.1 MPI_Send

MPI_Send routine is used to send a message to another process. The signature of MPI_Send is as follow.

int MPI_Send(void *data_to_be_sent, int sizeof_data, MPI_Datatype send_data_type, int destination_ID, int tag, MPI_Comm comm)

- data_to_be_sent: data which you want send over the network
- sizeof_data: number of single element to be sent
- send_data_type: data type of the variable to be sent
- destination_ID: process ID of destination
- tag: message tag
- comm: communicator handle

5.2.1.2 MPI_Recv

MPI_Send routine is used to receive a message from another process. The signature of MPI_Recv is as follow.

Int MPI_Recv(void *data_tobe_received, int sizeof_rec_data, MPI_Dataype data_typeof_received, int sender_ID, int tag, MPI_Comm comm., MPI_Status *status)

- data_tobe_received: variable in which sent data will be received
- sizeof_rec_data: size of data to be received
- data_typeof_received: data type of data to be received
- sender_ID: id of the sender process
- tag: message tag
- comm.: communicator handle
- status: status struct (MPI_Status)

5.2.1.3 More routines

There are other few routines which we will use to work with MPI. Their short descriptions are given below:

MPI_Init: it initializes MPI environment. This function can be considered as a stepping stone for MPI program. No MPI program can be executed without calling this function.

MPI_Comm_size: It returns the total number of processes after initializing MPI execution environment.

MPI_Comm_rank: It returns the rank of the calling MPI process within the specified communicator.

5.2.2 Designing Concepts of MPI

It is challenging task to design and inject MPI programming model in any serial algorithm. General structure of the MPI program goes like this (refer figure 5):

By taking reference of this MPI programming model there were multiple approaches to design this algorithm in parallel mode.

1) Divide the main string between multiple threads and allow each individual thread to search for pattern in allocated part of main string
2) Assign individual thread to individual pattern (in case of multi pattern search) and then search for that single pattern in the main string

In our implementation we have taken only single pattern for string matching for sake of simplicity. Considering this limitation first option was better suited for our algorithm. We have single pattern to search in main string. We can divide main string into multiple

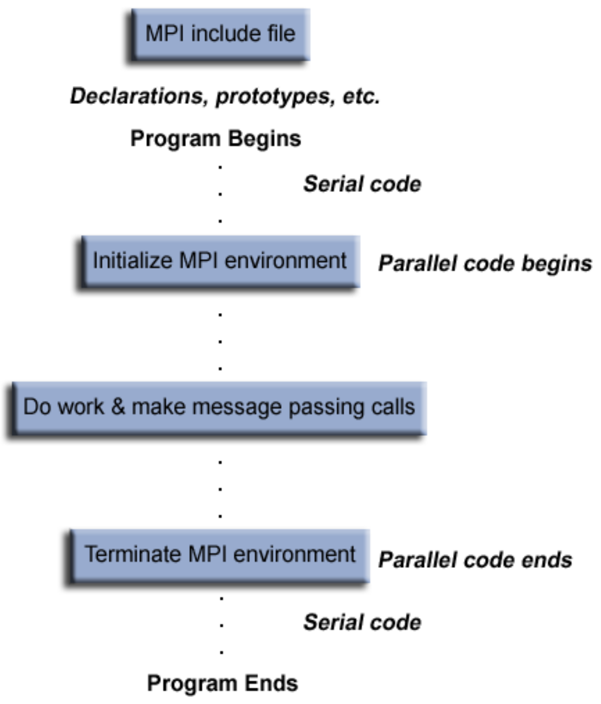
MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

Figure 5: Programming Model for MPI

parts and assign smaller string to different threads which will search given pattern in that part of string. Once all calculations are done, we combine their result to get final count of that pattern. This approach seems more promising in two ways as compared to second approach. First reason is we are not taking multiple patterns here. Second reason is if we would have taken second approach then MPI libraries would not have been used efficiently. As discussed above, Second approach divides main pattern into individual patterns which does not require communication between threads, as each individual thread would count their own pattern in the given main string. MPI program structure was major factor for rejection of second approach.

5.2.3 Final Design of Program using MPI

Considering first approach we have designed our program in parallel mode which divides the main string between multiple threads and searches given pattern.

Pseudo code for Rabin-Karp algorithm in parallel mode

```
Rabin-karp-Parallel (String, Pattern){
  initialize MPI environment
  get rank of specified communicator
  get number of thread for the program
  if (process == root process) {
    allocate its own part to calculate
    from main string
```

```
      for(i=0;i<num_proc;i++) {
        calculate the starting index
        and ending index for each process
        send those two values to each process
      }
      sum = rabin-karp(substring,pattern)
      for(i=0;i<num_proc;i++) {
        receive total number of occurrences
        from each processes
        add those values to final sum
      }
    }
    else { //if processes are slave
      receive starting index and ending index
      from root process
      temp_sum = rabin-karp(substring,pattern)
      send the temp_sum to root process
    }
  }
```

5.3 OpenMP

One of the very famous parallel processing API's is OpenMP (Open Multi-Processing). It is a multi platform API that uses 'shared memory model' in programming languages like C, C++ and Fortran. It runs on various operating systems like Linux, MacOS, Windows, etc. OpenMP consists of a set of complier directives, library routines and environment variables which provides a simple and flexible interface for developing parallel applications for platforms like desktop computer, supercomputers, etc.

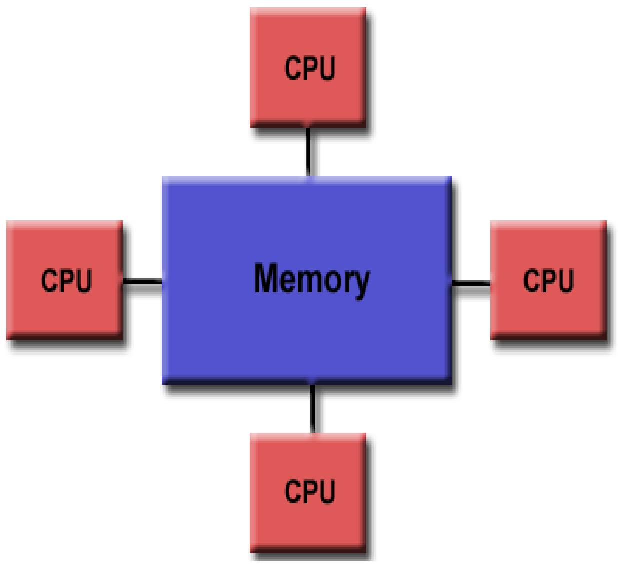

Figure 6: Shared Memory Model for OpenMP

5.3.1 Introduction

Open MP is an implementation of multithreading , a method of parallelizing whereas a master thread forks specified number of slave threads and scheduler schedules tasks among them. These threads then run concurrently achieving parallelism

The section of code that is meant to run in parallel is marked accordingly, with a preprocessor directive that will cause the threads to form before the section is executed. Each thread has an *id* attached to it which can be obtained using a function. The thread id is an integer, and the master thread has an id of *0*. After the execution of the parallelized code, the threads *join* back into the master thread, which continues onward to the end of the program.

By default, each thread executes the parallelized section of code independently. Work-sharing constructs can be used to divide a task among the threads so that each thread executes its allocated part of the code. Both task parallelism and data parallelism can be achieved using OpenMP in this way.

5.3.2 Some important routines

5.3.2.1 Omp_get_thread_num()

This method returns the unique id associated by the system to a thread. For master thread , this id is 0.

5.3.2.2 Omp_get_num_threads()

This method returns the number of threads that are being executed concurrently.

5.3.2.3 Omp_set_num_threads(n)

This method is used to set the level of multithreading where n stands for the number of threads you want to allocate to your program. For example, omp_set_num_thread(4) allocates 4 threads to a master thread.

5.3.3 Compilation

For compiling a c program using openMP and using above mentioned routines, we need to include corresponding header file.

<omp.h>

Compilation is done using following command [for Unix platform]

#gcc –fopenmp filename.c

#EXPORT OMP_NUM_THREADS = n

Run your program

#. /a.out

5.3.4 Thread creation

The pragma *omp parallel* is used to fork additional threads to carry out the work enclosed in the construct in parallel. The original thread will be denoted as *master thread* with thread ID 0.

Consider the following snippet:

```
#Pragma omp parallel
   {
   Thread 1 executes…
```

```
    Thread 2 executes...
    .
    .
    .
    Thread n executes…
     }
```

Pragma omp parallel is a predefined compiler directive that is the main entry point for execution of multiple threads. All the threads within this structured block run independently of all other threads. These threads have their own local environment variables within this block and do not share their variables. Once all threads execute their code, they all wait for each other before exiting the block. Figure 3 illustrates this process very clearly.

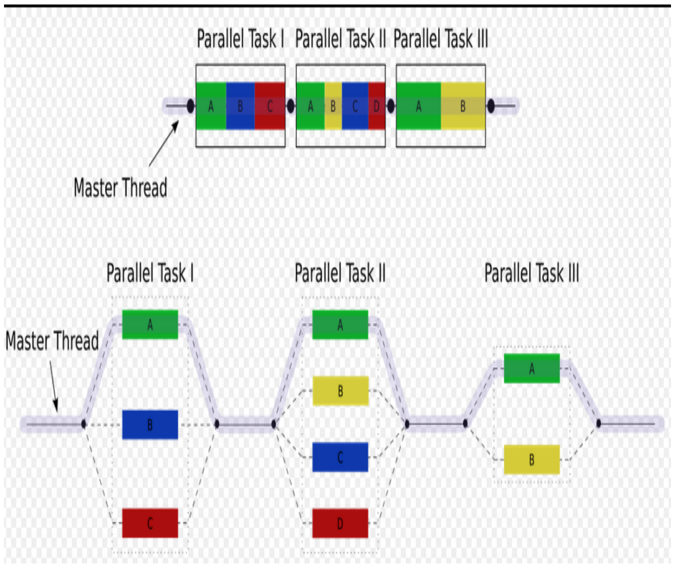

Figure 7: Illustrating the forking and join process for OpenMP

5.3.5 Work Sharing Constructs

A parallel construct by itself creates an SPMD or "Single Program Multiple Data" program i.e each thread **redundantly** executes the same code. A workaround for this is to use Work sharing Constructs provided by OpenMP. They are used to specify how to assign independent work to one or all of the threads.

Some common constructs that are used are:

- omp for or omp do: used to split up loop iterations among the threads, also called loop constructs.
- sections: assigns consecutive but independent code blocks to different threads
- single: specifies a code block that is executed by only one thread, a barrier is implied in the end
- master: similar to single, but the code block will be executed by the master thread only and no barrier implied in the end.

Example where an array is being assigned values is divided among various threads is shown below:

```
#pragma omp parallel
#pragma omp for
for(i=0;I<N;i++) { a[i] = a[i] + b[i];}
```

5.3.6 OpenMP clauses

Since OpenMP is a shared memory programming model, most variables in OpenMP code are visible to all threads by default. But sometimes private variables are necessary to avoid race conditions and there is a need to pass values between the sequential part and the parallel region (the code block executed in parallel), so data environment management is introduced as *data sharing attribute clauses* by appending them to the OpenMP directive. The different types of clauses are:

5.3.6.1 Data sharing attribute clauses

- shared: the data within a parallel region is shared, which means visible and accessible by all threads simultaneously. By default, all variables in the work sharing region are shared except the loop iteration counter.
- private: the data within a parallel region is private to each thread, which means each thread will have a local copy and use it as a temporary variable. A private variable is not initialized and the value is not maintained for use outside the parallel region. By default, the loop iteration counters in the OpenMP loop constructs are private.
- default: allows the programmer to state that the default data scoping within a parallel region will be either shared,or none forC/C++,or shared, firstprivate, p rivate, or none for Fortran. The noneoption forces the programmer to declare each variable in the parallel region using the data sharing attribute clauses.
- firstprivate: like private except initialized to original value
- lastprivate: like private except original value is updated after construct.
- reduction: a safe way of joining work from all threads after construct.

5.3.6.2 Synchronization clauses

- critical: the enclosed code block will be executed by only one thread at a time, and not simultaneously executed by multiple threads. It is often used to protect shared data from race conditions.
- atomic: the memory update (write, or read-modify-write) in the next instruction will be performed atomically. It does not make the entire statement atomic; only the memory update is atomic. A compiler might use special hardware instructions for better performance than when using critical.
- ordered: the structured block is executed in the order in which iterations would be executed in a sequential loop
- barrier: each thread waits until all of the other threads of a team have reached this point. A work-sharing

construct has an implicit barrier synchronization at the end.

- nowait: specifies that threads completing assigned work can proceed without waiting for all threads in the team to finish. In the absence of this clause, threads encounter a barrier synchronization at the end of the work sharing construct.

6. Experimental Results

For the purpose of testing of this algorithm, we worked on some random files of different sizes. To compare the speeds, testing was performed by implementing serial Rabin-Karp algorithm , parallelism through forking ,MPI and OpenMP. For fork technique, we limited our thread count to 2. We observed that there was significant speed up when compared to serial R-K. For MPI technique, we created 4 threads and found out that results vary depending on length (n) of file. For smaller n, serial R-K performs better than MPI based algorithm however as the length increases MPI technique tends to perform better than serial R-K algorithm. For OpenMP technique, we observed that there was no significant speed up when compared to serial version of the R-K algorithm and we could not find consistent results for different thread count. Therefore, we have not recorded the observations for OpenMP below. All these algorithms were made to run on 4GB RAM memory and 4 core processor. Also, the scope of this project deals with single pattern searching algorithm. The results for different sizes of main string are displayed in the following tables:

Length of file (n)	**Serial Rabin Karp**	**Rabin Karp using MPI (t =4))**
1533764	0.25	1.445
9170538	1.444	2.204
19588015	3.068	3.343
58764046	9.197	7.861
235056237	39.731	28.573

Table 1: Comparison of speed/performance of Rabin Karp Search Algorithm by MPI Approach

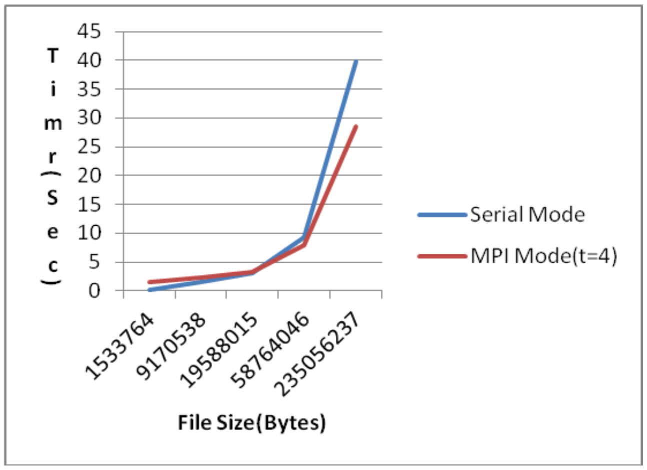

Figure 8: Line Graph for comparision between Serial and MPI based Rabin Karp

Length of file (n)	**Serial Rabin Karp**	**Rabin Karp using Fork(t =2)**
1533764	0.25	0.152
9170538	1.444	0.836
19588015	3.068	1.801
58764046	9.197	5.622
235056237	39.731	21.867

Table 2: Comparison of speed/performance of Rabin Karp Search Algorithm by Forking Approach

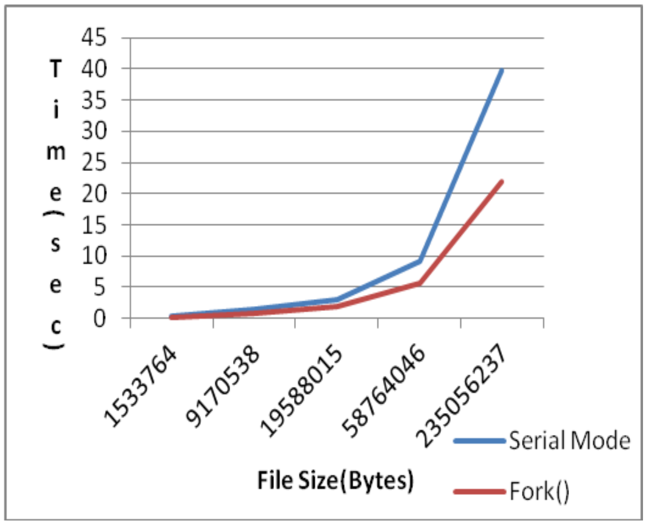

Figure 9: Line Graph for comparision between Serial and fork based Rabin Karp

7. CONCLUSION

In this paper we have applied different types of parallel framework, libraries and approaches. We wanted to achieve maximum performance gain in string matching algorithm while processing large file. While applying different techniques to gain performance, we observed several things in parallel processing paradigm.

It is very hard to decide the fix amount of running time for program which is running with multiple processes. Our all experimental results which we have shown above is average. You cannot get same fixed amount of time for same program with same inputs while working with multiple processes. It depends on number memory available, number of processes running on that machine at time of running program. Execution of the program highly depends on the environment of operating system kernel. Kernel is responsible for scheduling the processes. This environment gets changed on every mili-second. It also depends on number of cores on the machine. The more number of cores you have, more performance gain you achieve. Best practice is to average the running time of program while running with multiple threads.

While designing parallel program architecture you must consider all the parameters of the program. Parameters like how threads are going to communicate. Whether you want to share memory or pass messages through network. Such parameters are highly dependent on the design parallel architecture of program. Let’s assume your program is divided in multiple threads in such a way that they would have their own environment and would do their task independently. They just pass their final output of task to root process or any other process. Probably MPI libraries would give you best performance for your program rather than OPEN-MP. On the other hand if your program is highly dependent on each other thread. It would need lot of message passing calls during execution of program. It is not best idea to use MPI for that program. OPEN-MP would give you better performance for that program. In such case, sharing memory would be better option to get better performance.

In our program, every thread works independently. Therefore MPI seemed to be a suitable option to get better performance. At very beginning we designed our program using OPEN-MP. That was degrading our performance. Reason looked to be that every thread had to pass most of its time waiting for other process to complete its execution within the critical section. This factor degraded the performance of the program. So, we shifted towards MPI. We observed better performance while working with MPI.

In summary, we can conclude that every framework and library has its own pros and cons. It all depends on the nature and design of your program that you want to execute. Some libraries may give you best performance while some may not. So the better approach would be to analyze the algorithm or program on which you want to achieve parallelism. Thorough analysis gives you better idea about which parallel approach will give you best performance

8. REFERENCES

[1] Dharani, Nikhat, and Pin-Wen Wang. "Parallel Rabin-Karp Algorithm." Parallel Rabin-Karp Algorithm. Harvard University, Dec.-Jan. 2013. Web. 17 Nov. 2014.

[2] Broanac, Predrag, Leo Budin, and Domagoj Jakobovic. "Parallelized Rabin-Karp Method for Exact String Matching." *Proceedings of the ITI 2011 33rd Int. Conf. on Information Technology Interfaces* (2011): 585-90. Web. 2 Oct.2014.

[3] Snir, Marc. *MPI--the Complete Reference*. Cambridge, MA: MIT, 1998. *MPI--the Complete Reference*. Web. 5 Nov. 2014.

[4] Quinn, Michael J. *Parallel Programming in C with MPI and OpenMP*. Dubuque, IA: McGraw-Hill, 2004. Print.

[5] "Rabin–Karp Algorithm." *Wikipedia*. Wikimedia Foundation, 24 Oct. 2014. Web. 2 Oct. 2014.

[6] "Rolling Hash." *Wikipedia*. Wikimedia Foundation, 24 Oct. 2014. Web. 2 Oct. 2014.

[7] "MPI Examples." *MPI Examples*. Golden Energy Computing Organization, n.d. Web. 6 Nov. 2014.

[8] Barney, Blaise. "Message Passing Interface (MPI)." *Message Passing Interface (MPI)*. Lawrence Livermore National Laboratory, n.d. Web. 8 Nov. 2014.

[9] The University of Kansas. "An Introduction to the Message Passing Interface (MPI) Using C." *Introduction to the Message Passing Interface (MPI) Using C*. The University of Kansas, n.d. Web. 8 Nov. 2014.

[10] Mattson, Tim, and Larry Meadows. "A “Hands-on” Introduction to OpenMP." (n.d.): n. pag. *A “Hands-on” Introduction to OpenMP*. Intel. Web. 1 Nov. 2014.

[11] "OpenMP." *Wikipedia*. Wikimedia Foundation, n.d. Web. 3 Nov. 2014.

[12] "Difference between MPI and Open MP." *Question 5*. Wiki.utep.edu, n.d. Web. 8 Nov. 2014.

[13] Blaise Barney. "OpenMP Example - Hello World." *Open MP Examples*. N.p., May 1999. Web. 1 Nov. 2014.

[14] POSTECH. "Introduction to MPI and OpenMP." (n.d.): n. pag. *Introduction to MPI and OpenMP*. POSTECH. Web. 2 Nov. 2014.

[15] Stackoverflow. "Rolling Hash in Rabin-Karp." *C++*. Stackoverflow, n.d. Web. 5 Oct. 2014.

[16] "Compiling and Running MPI Programs." *TAMU Supercomputing Facility*. Texas A&M University, 17 Mar. 2013. Web. 8 Nov. 2014.

[17] Katey Cruz. “String Matching Using the Rabin-Karp Algorithm”. Smith College, 12 Dec, 2000. Web 7 Nov. 2014

[18] "Plagiarism Detection." *Wikipedia*. Wikimedia Foundation, 11 Feb. 2014. Web. 14 Nov. 2014.

[19] "Pattern Matching." *Wikipedia*. Wikimedia Foundation, 11 Nov. 2014. Web. 17 Nov. 2014.