Comparing Performance of KMP-Algorithm in Serial and Parallel Environments

Joel John Thomas
USN: 1MS18CS053

Abstract

In order to make the most out of the CPU's resources, we must find different ways to implement the algorithms. KMP (the Knuth–Morris–Pratt) algorithm is used **to find a "Pattern" in a "Text"**. This algorithm compares character by character from left to right. But whenever a mismatch occurs, it uses a preprocessed table called "Prefix Table" to skip characters comparison while matching. To activate parallel programming, we will use MPI clustering programming. MPI is a message-passing model of distributed memory. To assess parallel programming's performance, we compare it to a sequential KMP algorithm, which does not make efficient use of the CPU's resources. To determine whether an algorithm is more efficient, we evaluate the performance of sequential and parallel KMP algorithms.

Introduction

The exponential growth of processing and network speeds means that parallel architecture isn't just a good idea; it's necessary. Big data and the IoT will soon force us to crunch trillions of data points at once. Nowadays new applications always need faster processors in today's world to operate efficiently. Parallel systems are employed in a lot of business applications nowadays. Parallel computing uses multiple computer cores to attack several operations at once. Unlike serial computing, parallel architecture can break down a job into its component parts and multi-task them. Parallel computer systems are well suited to modeling and simulating real-world phenomena. Without parallel computing, performing digital tasks would be tedious, to say the least. The advantages of parallel computing are that computers can execute code more efficiently, which can save time and money by sorting through "big data" faster than ever.

Parallel systems, divide the programs into multiple fragments and process them all at the same time in order to reduce the execution time of the program. By constructing a parallel processing bundle or a combination of both entities, a parallel system can deal with many processors, machines, computers, or CPUs, among other things.

String searching is one of the most common problems where it tries to find whether a string is a sub-string of another particular string or not. There are several algorithms to solve this problem. These algorithms are mostly used in text edit processing, image processing, literature retrieval, natural language recognition, spell checking, and many other fields.
The most common string searching algorithm is Knuth–Morris–Pratt (KMP) algorithm. It is a prefix based searching algorithm faster than normal string searching algorithm like 'Naive String Searching Algorithm'. The algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n).

We can make this algorithm faster if we make it parallelly executable. As it is linear time complexity algorithm, it will consume more time if the text data is big. If we run the KMP algorithm on multiple computers to handle larger data, it will be less time-consuming. By doing that, it can be used in large-scale applications. To achieve parallelism in KMP algorithm, we will use MPI clustering programming. MPI is a message-passing model of distributed memory. The aim of the MPI is to establish an efficient and flexible standard for message passing which is used for writing message passing programs. As such, MPI is the vendor independent and having message passing library. The benefit of developing message passing software using MPI helps to achieve design goals like portability, efficiency and flexibility. MPI performs their operations by using the methods such as MPI_send, MPI_receive, MPI_COMM_WORLD, MPI_Init and many more by Using these kind of functionality the processes communicate with themselves.

The steps of parallel implementations using MPI are given below:

1) Read the input file of Snort rules, prepare the input stream of that file, obtain the file size, allocate memory to contain whole file, copy the file into the buffer and divide it across processes using MPI_Bcast and MPI_Barrier.

2) Prepare the input stream of pattern file and get the number of lines.

3) Divide the input file among the processors.

4) Call the KMP search function and compute the longest proper suffix and at last show the matching patterns as output.

Code:

```c

#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <malloc.h>

#define master 0
#define MAX(a,b) ((a) > (b) ? a : b)
#define MILLION 1000000L

#define bug printf("trust\n");

int* kmp(char* target, char* pattern, int* table, int myrank);
int* kmptable(char* pattern, int len);
void print(char* arr, int id, int len);
void fillup(char** matrix, int rankid, char* msg, int len, int offset);
void freeDouble(char** matrix, int nproc);
void printMatrix(char **matrix, int nproc, int id, int length);
int* getRealIdx(int* answer, int len, int myrank, int x, int y, int* length);
void pinpoint(int* result, int* msg, int shortlen);


void pinpoint(int* result, int* msg, int shortlen){
        int j = 0;
        for(j=0; j<shortlen; j++){
                if(j > 0 && msg[j] == 0)
                        break;
                else if(j == 0 && msg[j] == 0)
                        continue;
                else{
                        result[msg[j]] = 1;
                }
        }
}

// get the exact idx(es) in each process
// return back an int pointer
int* getRealIdx(int* answer, int len, int myrank, int x, int y, int* length){
        int* package = (int *) calloc(len, sizeof(int));
        int index;
        int real_len = 0;
        int i;
        for(i=0;i < len; i++){
                if(i > 0 && answer[i] == 0)
                        break;
                else if(i == 0 && answer[i] == 0)
                        continue;
                else{
                        if(myrank == 0){
                                index =  answer[i];
                                package[real_len++] = index;
                                // printf("This is REAL index: %d\n", index);
                        }else{
```

```c
                        // printf("This is REAL index: %d\n", index);
                }else{
                        index = x * myrank - y + answer[i];
                        package[real_len++] = index;
                        // printf("This is REAL index: %d\n", index);
                }

        }
    }
    int* result = (int*)malloc((real_len) * sizeof(int));

    for(i =0 ; i<real_len; i++){
        if(i > 0 && package[i] == 0)
            break;
        else{
            result[i] = package[i];
        }
    }
    *length = real_len;
    free(package);
    return result;
}

void printMatrix(char **matrix, int nproc, int id, int length){
    // printf("This matrix at %d row.\n", id);
    int i ,j;
    for(i=0 ;i < nproc; i++){
        if(i == id){
            for(j=0;j<length;j++){
                printf("%c", matrix[i][j]);
                break;
            }
            printf("\n");
        }
    }
    printf("\n");
}


// free two-pointer matrix
void freeDouble(char** matrix, int nproc){
    for (int i=0; i<nproc; ++i) {
        free(matrix[i]);
    }
    free(matrix);
}


void fillup(char** matrix, int rankid, char* msg, int len, int offset){
    int i;
    for(i=0;i<len;i++)
        matrix[rankid][offset + i] = msg[i];
}

```

```c
// print out one pointer list
void print(char* arr, int id, int len){
    int j;

    for(j=0; j<len; j++){
        printf("%c", arr[j]);
    }

}

// key func kmp
int* kmp(char* target, char* pattern, int* table, int myrank){
    int n = strlen(target);
    int m = strlen(pattern);
    int* answer = (int*) calloc(n-m+1,sizeof(int));
    int j=0;
    int i=0;
    int index = 0;
    while(i<n){
        if(pattern[j]== target[i])
        {
            j++;
            i++;
        }
        if(j == m){
            printf("Start Point: %d.\n", i-j);
            answer[index++] = i-j;
            j = table[j-1];

        }else if(i < n && pattern[j] != target[i]){
            if(j!=0)
                j = table[j-1];
            else
                i++;
        }
    }
    return answer;
}


int main(int argc, char** argv){
    int i,j;
    int n;
    int m;

    struct timespec start1, end1;
    double diff;

    char target[] =
"zztzjjsharzzfuasssssssssssssssssssssssssssssssssssssssssssssssssssssssssssssaaaaaaaaaaaajjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjjaaaaaaaaaaaaaaaaaaa
k-wlew-l;;;;ppppdopsdspdspdpsdpsdspdpsdpspdpsdpspdpsdpdpspdspdpspdpsdpspdspdpsdpspdpsdpsdpspdsdpspdspdpspdspdpspdspap-wq=l-
lk,ldlslldslddlsldsldlslslsldlsldlsdlsdllsdldlsldslsldlsdslld[lsdldsll[dl[sld[sd[[sl[lds[l[lds[l[dl[sl[ls[dl[sd[sd[s[dsl[dls[d[s[dls[dl[sld[ls[dls[l[s[ewpjjjjoodgnnogndngdongodngod
    char pattern[] = "pryyaogqzekpumxktopubehtpzeoqlkfcifewmwysgnotkbmyqhzuqujpayflinpsuvqisobabrtnsinjrofhkpzkpgvxtmpqifycubcxvwnngayxpegbvfmkm";
```

```
    char pattern[] = "pryyaogqzekpumxktopubehtpzeoqlkfcifewmwysgnotkbmyqhzuqujpayflinpsuvqisobabrtnsinjrofhkpzkpgvxtmpqifycubcxvwnngayxpegbvfmkm";

    n = strlen(target);
    m = strlen(pattern);

    int tag = 1;
    int tag2 = 2;

    int* kmp_table = kmptable(pattern, m);



    int myrank, nproc;
    MPI_Init(&argc, &argv);                                         //initialize the MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);                 //number of processes that given in command line
    MPI_Comm_size(MPI_COMM_WORLD, &nproc);                  //ranks the process rank




    MPI_Status status;

    int start_pos = n/nproc;
    int str_per_proc = n/nproc;
    int end = str_per_proc + n % nproc;


    char** matrix = (char**)malloc(nproc*sizeof(char*));
    for(i=0;i<nproc;i++){
            matrix[i] = (char*)malloc((end+m-1) * sizeof(char));
    }
    printMatrix(matrix, nproc, myrank, 10);

  char send_msg[MAX(m-1, end)]; // 4-1 = 3
    char recv_msg[MAX(m-1, end)];
  char end_msg[MAX(m-1, end)];

  double start, stop;


    if(myrank == master){
            printf("Result KMP Sequential\n");
            start = MPI_Wtime();
            for(i=1;i< nproc;i++){
                    if(i == nproc-1){
                            strncpy(send_msg, target + i*start_pos, end);
                            MPI_Send(send_msg, end, MPI_CHAR, i, tag, MPI_COMM_WORLD);
                    }else{
                            strncpy(send_msg, target + i*start_pos, str_per_proc);
                            MPI_Send(send_msg, str_per_proc, MPI_CHAR, i, tag, MPI_COMM_WORLD);
                    }


            }
```

```

            }
            strncpy(send_msg, target + master * start_pos, str_per_proc);
            fillup(matrix, myrank, send_msg, str_per_proc, 0);

    // The last process
    }else if(myrank == nproc-1){
            MPI_Recv(recv_msg, end, MPI_CHAR, master, tag, MPI_COMM_WORLD, &status);

            fillup(matrix, myrank, recv_msg, end, m-1);
    }
    else{
            MPI_Recv(recv_msg, str_per_proc, MPI_CHAR, master, tag, MPI_COMM_WORLD, &status);
            fillup(matrix, myrank, recv_msg, str_per_proc, m-1);
    }

    MPI_Barrier(MPI_COMM_WORLD);

    if(myrank == nproc-1){

            MPI_Recv(end_msg, m-1, MPI_CHAR, myrank-1, tag, MPI_COMM_WORLD, &status);

            fillup(matrix, myrank, end_msg, m-1, 0);
            int* answer = kmp(matrix[myrank], pattern, kmp_table, myrank);

            int len;
            // ---------------------
            int* result = getRealIdx(answer, end-m+1, myrank, str_per_proc, m-1, &len);
            // ---------------------
            free(answer);

            MPI_Send(result, len, MPI_INT, master, tag2, MPI_COMM_WORLD);
            free(result);

    }else{
            strncpy(send_msg, target + str_per_proc-m + 1 + myrank * str_per_proc, m-1);
            MPI_Send(send_msg, m-1, MPI_CHAR, myrank+1, tag, MPI_COMM_WORLD);

            /* Processes other than master one
             * Re-recv more msg from the previous process whose rank is myrank-1
             * kmp implementation
             * Send the result back to master process
             */

            if(myrank!=master){
                    MPI_Recv(end_msg, m-1, MPI_CHAR, myrank-1, tag, MPI_COMM_WORLD, &status);
                    fillup(matrix, myrank, end_msg, m-1, 0);
                    // implement kmp to get the matching result
                    int* answer = kmp(matrix[myrank], pattern, kmp_table, myrank);
                    int len;
                    int* result = getRealIdx(answer, end-m+1, myrank, str_per_proc, m-1, &len);
                    free(answer);
                    MPI_Send(result, len, MPI_INT, master, tag2, MPI_COMM_WORLD);
                    free(result);
```

```c
                        int* answer = kmp(matrix[myrank], pattern, kmp_table, myrank);
                        int len;
                        int* result = getRealIdx(answer, end-m+1, myrank, str_per_proc, m-1, &len);
                        pinpoint(final_result,result, len);
                        free(answer);
                        free(result);
                        int j;
                        for (j = 0; j < end+m-1; j++)
                        {
                                recv[j] = 0;
                        }

                        for(j = 1; j <nproc; j++){
                                MPI_Recv(recv, end+m-1, MPI_INT, j, tag2, MPI_COMM_WORLD, &status);
                                pinpoint(final_result,recv, end+m-1);
                        }
                        stop = MPI_Wtime();
                        free(recv);
                        double timeDifference = (stop-start) * MILLION;
                        printf("Target Length: %d\n", n);
                        printf("Pattern Length: %d\n", m);
                        printf("Time Elapsed: %0.05f\n",timeDifference);
                        // for(j=0;j<n;j++){
                        //      if(final_result[j] == 1)
                        //              printf("Find a matching substring starting at: %d.\n", j);
                        // }
                        printf("\n");
                        free(final_result);
                }
        }

        MPI_Barrier(MPI_COMM_WORLD);
        freeDouble(matrix, nproc);
        MPI_Finalize();

        return 0;
}

int* kmptable(char* pattern, int len){
        int k = 0;
        int i = 1;
        int* table = (int*)malloc(len * sizeof(int));
        table[0] = k;
        for(i=1;i<len;i++){
                while(k > 0 && pattern[i-1] != pattern[i]){
                        table[i] = k-1;
                        k = table[i];
                }
                if(pattern[i] == pattern[k])
                        k++;
                table[i] = k;
        }
        return table;
}
```

Output:

On a Computer without using MPI

Result KMP_MPI
Start Point: 49521.
Target Length: 99044
Pattern Length: 122
Time Elapsed: 2581.42600

On both Computer using MPI Clustering

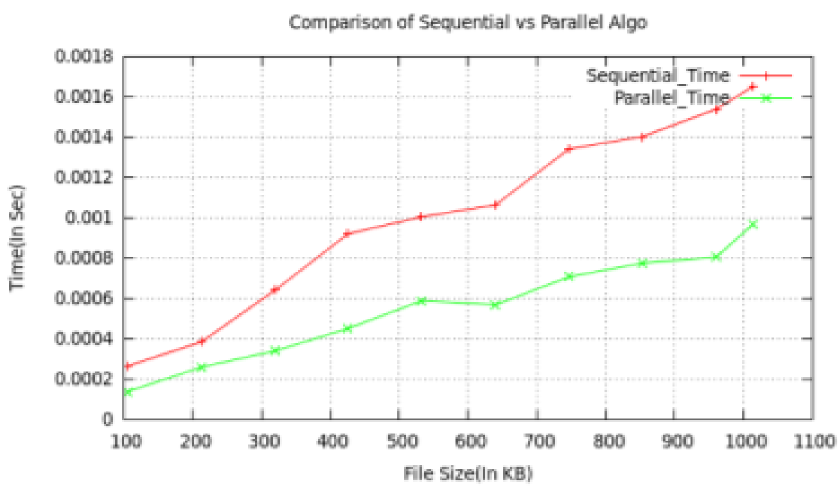
Result KMP_MPI
Start Point: 49521.
Target Length: 99044
Pattern Length: 122
Time Elapsed: 698.22000

```
data10000.txt     kmp_MPI.c              pattern.txt       text.txt
data1000.txt      kmp_MPI.out            Temporary.c
joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpicc -g -Wall -o test kmp_MPI.c
kmp_MPI.c: In function ‘main’:
kmp_MPI.c:153:9: warning: unused variable ‘diff’ [-Wunused-variable]
  153 |  double diff;
      |         ^~~~
kmp_MPI.c:152:27: warning: unused variable ‘end1’ [-Wunused-variable]
  152 |   struct timespec start1, end1;
      |                           ^~~~
kmp_MPI.c:152:19: warning: unused variable ‘start1’ [-Wunused-variable]
  152 |   struct timespec start1, end1;
      |                   ^~~~~~
kmp_MPI.c:148:8: warning: unused variable ‘j’ [-Wunused-variable]
  148 |  int i,j;
      |        ^
joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpiexec -n 2 ./test
Invalid MIT-MAGIC-COOKIE-1 key



Result KMP Sequential
Start Point: 49521.
Target Length: 99044
Pattern Length: 122
Time Elapsed: 2581.42600

joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpicc -0 test kmp_MPI.c
gcc: error: unrecognized command line option ‘-0’
joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpicc -o test kmp_MPI.c
joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpirun -np 2 ./test
Invalid MIT-MAGIC-COOKIE-1 key

Result KMP Sequential


Start Point: 49521.
Target Length: 99044
Pattern Length: 122
Time Elapsed: 698.22000

joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ ls
data1000000.txt   HowToRun.txt           KMPSequentical    test
data100000.txt    intrductionReport.pdf  KMPSequentical.c  testmake.py
data10000.txt     kmp_MPI.c              pattern.txt       text.txt
data1000.txt      kmp_MPI.out            Temporary.c
joel@joel-Inspiron-3576:~/Downloads/KMP-Parallel-Using-MPI-master$ mpicc -g -Wall -o test KMPSequentical.c
KMPSequentical.c: In function ‘main’:
KMPSequentical.c:17:8: warning: unused variable ‘j’ [-Wunused-variable]
   17 |  int i,j;
      |        ^
KMPSequentical.c:17:6: warning: unused variable ‘i’ [-Wunused-variable]
   17 |  int i,j;
      |      ^
KMPSequentical.c: In function ‘kmptable’:
```

Result

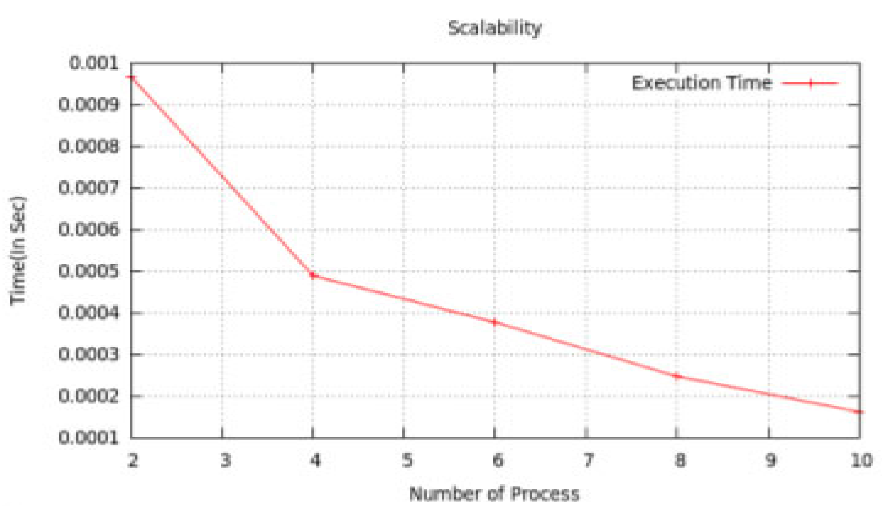
The most time consuming part of KMP search is the LPS that is Longest Prefix Suffix. The following results show the Compute time of this algorithm.

Sequential Vs Parallel Time Comparison graph:



The above graph shows us the comparison of sequential and parallel time in which we have taken the input file in KB on X-axis and Time(s) to compute in Y-axis by keeping the pattern file constant. So the above graph shows that the parallel computation time is more efficient than the sequential time and the performance of parallel is completely outplayed the sequential performance.

Scalability graph:



The above shows us the number of processes on X-axis and the time taken by that process on Y-axis. In the above graph 10 processes are taken and computed the time taken by them in seconds. It shows that if we go on increasing the number of processes, then the performance gets better. In this way the scalability is achieved.

By viewing the outputs and the graphs we can conclude by saying we implemented the Knuth-Morris-Pratt algorithm to match the Snort pattern and the performance of the parallel algorithm is far better than the sequential algorithm for multiple patterns.

References

[1] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. In Field-Programmable Custom Computing Machines, 2001. FCCM'01.The 9th Annual IEEE Symposium on, pages 227-238. IEEE, 2001.

[2] Christopher R Clark and David E Schimmel. Scalable pattern matching for high speed networks. In Field-Programmable Custom Computing Machines, 2004.FCCM 2004. 12th Annual IEEE Symposium on, pages 249-257. IEEE, 2004.

[3] Sarang Dharmapurikar and John Lockwood. Fast and scalable pattern matching for content _ltering. In Architecture for networking and communications systems, 2005. ANCS 2005. Symposium on, pages 183-192. IEEE, 2005.

[4] Alfred V Aho and Margaret J Corasick. E_cient string matching: an aid to bibliographic search. Communications of the ACM, 18(6):333-340, 1975.

[5] Nen-Fu Huang,Hsien-Wei Hung, ShengHung Lai, Yen-Ming Chu, and Wen-Yen Tsai. A gpu-based multiple-pattern matching algorithm for network intrusion detection systems. Advanced Information Networking and Applications-Workshops, 2008. AINAW 2008. 22nd International Conference on, pages 62-67. IEEE, 2008