VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

CSE4001 - Parallel and Distributed Computing J-Component Project

Slot: E2

Faculty Name: Prof. Dr Harini S

Project Title: RABIN KARP PARALLELIZATION AND COMPARATIVE ANALYSIS

Team Members:

1. M. Hanuman Sai - 19BPS1066
2. N Charan Reddy - 19BCE1594
3. P Varshith - 19BCE1585

Abstract:

Correct string coordinating alludes to the pursuit of each and any events of a string in another string. These days, this issue presents itself in different portions in a lot, beginning from standard schedules for the correct hunt, which schedules are executed into projects for word processing and preparing, through databases and the distance to their different applications in different sciences. The errand of discovering strings that match to a given example is of enthusiasm for an assortment of useful applications, including DNA sequencing and message looking. The endeavour of finding strings that match to a given model is of excitement for a grouping of sensible applications, including DNA sequencing and message looking. Owing to its essentialness, decisions to revive the precedent planning task have been extensively investigated in the composition. The guideline responsibility of this work is to present a parallel type of Accelerated Rabin-Karp.

The Rabin-Karp algorithm is an implementation of exact string matching that uses a rolling hash to find any ‘one’ set of pattern strings in a text. The Rabin-Karp algorithm is used in detecting plagiarism because, given a pattern and a source of texts, the algorithm can quickly search through papers for patterns from the source material. We have parallelized this algorithm using MPI, Distributed Python – disPy.

Introduction:

A rolling hash (also known as recursive hashing) is a hash function where the input is hashed in a window that moves through the input. A few hash functions allow a rolling hash to be computed very quickly—the new hash value is rapidly calculated given only the old hash value, the old value removed from the window, and the new value-added to the window— similar to the way a moving average function can be computed much more quickly than other low-pass filters.

The Rabin–Karp algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any 'one' of a set of pattern strings in a text. For the text of length n and p patterns of combined length m, its average and best-case running time is O(n+m) in space O(p), but its worst-case time is O(nm).

This project aims to improve the process of hashing by parallelizing the process using MPI. A given set of input will be tested first by the serial method and then by parallel and we will see the advantage of parallelism.

In light of this significant assignment and step by step expanding research in various fields, industry, institute individuals requesting such programming to identify whether submitted articles, books, national or global papers are certified or not.

One of the more productive calculations is Rabin-Karp calculation, whose multifaceted nature is direct. This work furnishes us with one approach to parallelize this calculation for execution on multiprocessor frameworks.

The Rabin-Karp algorithm is used in detecting plagiarism because, given a pattern and a source of texts, the algorithm can quickly search through papers for patterns from the source material.

Problem Statement:

The Rabin–Karp algorithm is a string searching algorithm created by Richard M. Karp and Michael O. Rabin that uses hashing to find any 'one' of a set of pattern strings in a text. For the text of length n and p patterns of combined length m, its average and best-case running time is O(n+m) in space O(p), but its worst-case time is O(nm).

This project aims to improve the process of hashing by parallelizing the process using MPI. A given set of input will be tested first by the serial method and then by parallel and we will see the advantage of parallelism.

Literature Survey:

---

1. **' Parallel Rabin Karp Algorithm Implementation on GPU**'.
   Here (Bordim & Nakano, 2018) they see that the undertaking of discovering strings that match to a given example is of enthusiasm for an assortment of reasonable applications, including DNA sequencing and message looking. Attributable to its significance, choices to quicken the example coordinating assignment have been broadly explored in the writing. The principal commitment of this work is to introduce a parallel form of the observed Rabin-Karp calculation. Given an example P of size m and a content string T of size n, the Rabin-Karp calculation discovers all events of the example P in T with high likelihood. The proposed plan can contrast k diverse examples with the information message simultaneously.
   The proposed, parallelized, a variant of the Rabin-Karp calculation has been actualized on the GeForce GTX 960 GPU.

2. **A Comparative Analysis of Single Pattern Matching Algorithms in Text Mining.**

Here (Sheshayee &Thailambal 2015) have suggested that content Mining is a developing zone of research where the important data of clients should be given from extensive measure of data. The client needs to discover a content

P in the hunt box from the gathering of content data T. A match should be found in the data then just the hunt is effective. Many String coordinating calculations accessible for this hunt. This paper talks about three calculations in novel example seeking in which just a single event of the example is looked. Knuth Morris Pratt, Naive and Boyer Moore calculations actualized in Python and looked at their execution time for various Text length and Pattern length. This paper likewise gives you a concise thought regarding time Complexity, Characteristics given by different creators. The paper is finished up with the best calculation for increment in content length and example length.

3. In '**Plagiarism Detection by using Karp-Rabin and String Matching Algorithm Together**' By Sonawane Kiran Shivaji, Prabhudeva S Here (Shivaji & Sonawane, 2015) they saw that the today world is replicating something from different sources and asserting it as a possess commitment is wrongdoing. We have additionally observed it is the real issue in scholastic understudies of UG, PG or even at PhD level duplicating some piece of unique records and distributing on possess name without taking appropriate authorization from creator or engineer. Numerous product apparatuses exist to discover and help the dreary and tedious errand of following written falsification because distinguishing the proprietor of that entire content is troublesome and unthinkable for market.

Methodology:

We have used the master-slave model in MPI to parallelize the Rabin Karp algorithm. Files with a large amount of text and a pattern to be searched within them, are given as input. The master process takes this input and divides the files among the slave processes. Each slave process performs the Rabin Karp algorithm on its input file and returns the results to the master. The master process compiles the results and displays them.

Algorithm:

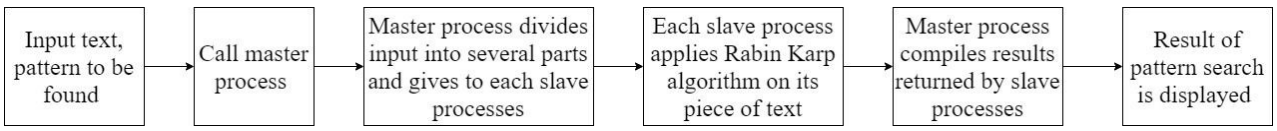
Rabin-Karp algorithm is an algorithm used for searching/matching patterns in the content utilizing a hash function. Unlike Naive string matching algorithm, it doesn't go through each character in the underlying stage. It channels the characters that don't match and then perform the comparison.

```
n = t.length
m =
p.length
 h = dm-1 mod
 q p = 0
 t0 = 0
 for i = 1 to m
    p = (dp + p[i]) mod
    q t0 = (dt0 + t[i])
    mod q
 for s = 0 to n -
    m if p = ts
       if p[1.....m] = t[s + 1.s + m]
          print "pattern found at
    position" s If s < n-m
       ts + 1 = (d (ts - t[s + 1]h) + t[s + m + 1]) mod q
```

Complexity:
The normal case and best case intricacy of Rabin-Karp calculation is O(m + n) and the most pessimistic scenario unpredictability is O(mn).

The most pessimistic scenario intricacy happens when false hits happen a number for all the windows.

Block Diagram:



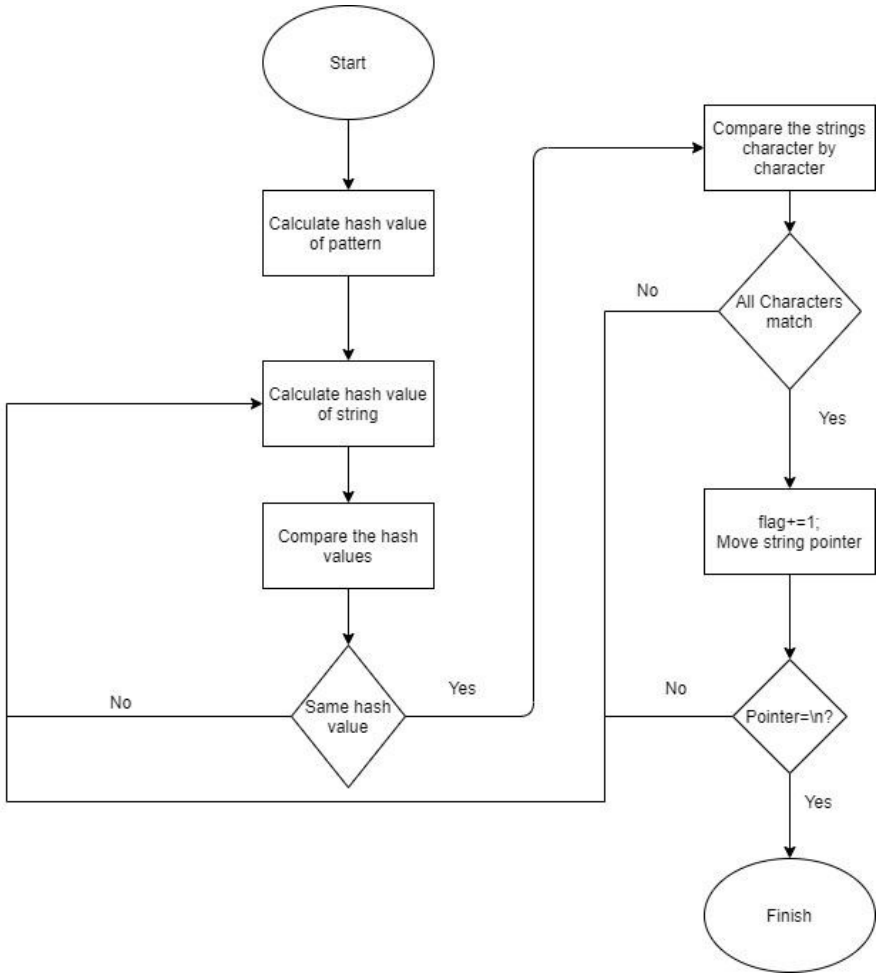
Hardware Specifications:

Processor: Intel(R) Core
i5 RAM : 8GB

Software specifications:

OS: Windows 10
Python 3.8
Microsoft MPI 10.1.12498.18

Flowchart:


Start
Calculate hash value of pattern
Calculate hash value of string
Compare the hash values
Same hash value
No
Yes
Compare the strings character by character
All Characters match
No
Yes
flag+=1;
Move string pointer
Pointer=\n?
No
Yes
Finish

Code Snippets and Implementation

RabinKarpSerial.py

```
import
time
import
string
from sys import argv

#d value for the rolling
hash d = 26

# splits pattern into specified pattern size before
searching def splitCount(s, count):
   return[s[i:i+count] for i in range(0,len(s),count)]

# removes punctuation and capitalizes letters to prep for
processing def prep_text(text):
   exclude = set(string.punctuation)
   return ''.join(x.upper() for x in text if x not in exclude)

#runs the rka of a piece of the pattern on
piece of text def
sub_search(txt,pat,q,matchlist):
   patlen =
 len(pat) #print
 pat
   txtlen =
   len(txt)
   hashpat = 0
   hashpat2 = 0
   hashtxt = 0
   hashtxt2 = 0
   h = 1
   h2 = 1
   tuple_array = []
```

```
for i in
   range(0,patlen-1):
   # print "creating
   h\n" h = (h*d)%q
   h2 = (h2*d)%q2

for i in range(0,patlen):
   # print "creating hash values\n"
```

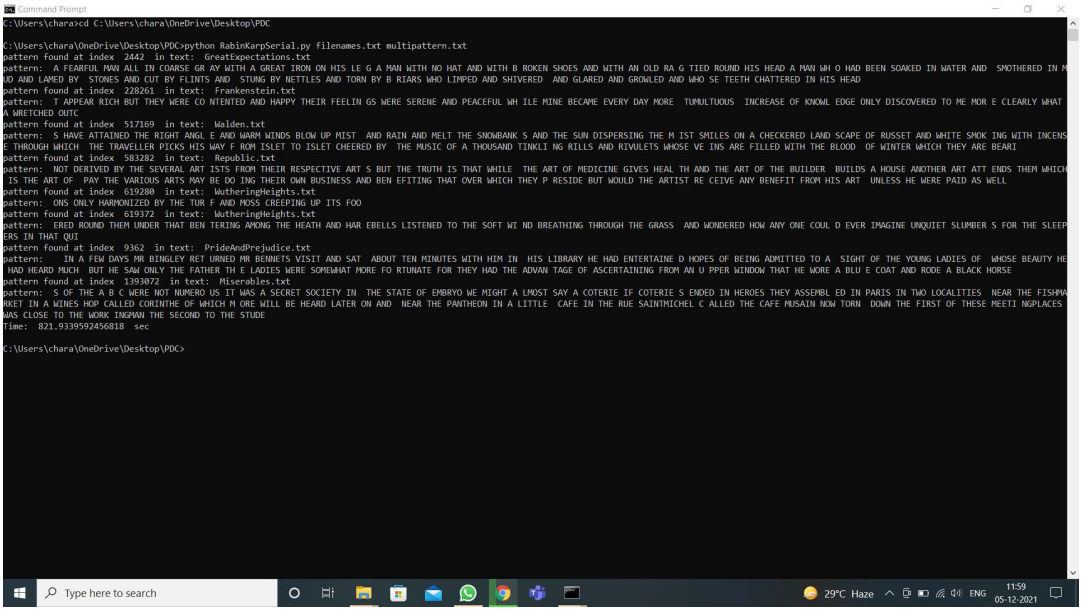
```
        hashpat = (d*hashpat +
        ord(pat[i]))%q hashpat2 =
        (d*hashpat2 + ord(pat[i]))%q2
        hashtxt = (d*hashtxt +
        ord(txt[i]))%q hashtxt2 =
        (d*hashtxt2 + ord(txt[i]))%q2


    for i in range(0,txtlen-
        patlen+1): if (hashpat ==
        hashtxt):
            for j in range
                (0,patlen): if
                (txt[i+j] != pat[j]):
                    break
                if j == patlen-1:
                    matchlist.append((i,txt[i:i+pat
                    len]))


        if (i < txtlen-patlen):
            # print "shifting pat in txt\n"
            hashtxt = (d*(hashtxt - ord(txt[i])*h) +
            ord(txt[i+patlen]))%q hashtxt2 = (d*(hashtxt2 -
            ord(txt[i])*h2) + ord(txt[i+patlen]))%q2 if (hashtxt < 0):
                hashtxt = hashtxt
            + q if (hashtxt2 < 0):
                hashtxt2 = hashtxt2 + q2

# splits pattern into pieces and calls sub_search on
each piece def full_search(txt,pat,q,patsize):
    splitpat =
    splitCount(pat,patsize)
    matchlist = []
    for subpat in range(0,len(splitpat)):
        sub_search(txt,splitpat[subpat],q,mat
        chlist)
    return matchlist
```

```
# combines consecutive matches for
entire match def
post_process(patlen,recv_result):
        match_len =
        len(recv_result) result =
        []
        curr = 0
        offset = 1

        if match_len == 1:
```

```
                result.append(recv_result[c
                urr]) return result
        else
        :       index,string = recv_result[curr]

        while curr < match_len:
                nextcurr = curr+offset

                if nextcurr < match_len and index + offset*patlen ==
                        recv_result[nextcurr][0] : string = string + " " +
                        recv_result[nextcurr][1]
                        offset += 1
                else:
                        result.append((index,stri
                        ng)) curr = nextcurr
                        if curr < match_len:
                                index,string = recv_result[curr]

        return result

###################################----MAIN
 ------------------------------------------------- ######################################
##
#########

if __name_== '__main_

        ': if len(argv) !=

        3:
                print ("Usage: mpiexec -n [# of processors] python", argv[0], "[corpus
                filenames]
[input text]")
                exit()
        q = 1079
        q2 = 1011
        patsize = 30;

        # opening many files
```

```
filenames, pattxt =
argv[1:] with open
(pattxt,"r") as patfile:
        pat=patfile.read().replace('\n',' ')

#gets rid of all
punctuation pat =
prep_text(pat)
```

```
#starts rka
start = time.time()
files =
open(filenames).readlines()
for i in files:
        filename = i.replace('\n','')
        with open (filename,"r") as txt:
                txt =

        txt.read().replace('\n',' ') txt =

        prep_text(txt)

        pre_results = full_search(txt,pat,q,patsize)

        if len(pre_results) == 0:
                results = pre_results
        else:
                results = post_process(patsize,pre_results)

        for index,match in results:

                print ("pattern found at index %d in text: %s"
                %(index,filename)) print ("pattern: %s" %match)

end = time.time()
print ("Time: %f sec" %(end-start))
```


Command Prompt
C:\Users\chara>cd C:\Users\chara\OneDrive\Desktop\PDC
C:\Users\chara\OneDrive\Desktop\PDC>python RabinKarpSerial.py filenames.txt multipattern.txt
pattern found at index 2442 in text: GreatExpectations.txt
pattern: A FEARFUL MAN ALL IN COARSE GR AY WITH A GREAT IRON ON HIS LE G A MAN WITH NO HAT AND WITH B ROKEN SHOES AND WITH AN OLD RA G TIED ROUND HIS HEAD A MAN WH O HAD BEEN SOAKED IN WATER AND SMOTHERED IN M
UD AND LAMED BY STONES AND CUT BY FLINTS AND STUNG BY NETTLES AND TORN BY B RIARS WHO LIMPED AND SHIVERED AND GLARED AND GROWLED AND WHO SE TEETH CHATTERED IN HIS HEAD
pattern found at index 228261 in text: Frankenstein.txt
pattern: T APPEAR RICH BUT THEY WERE CO NTENTED AND HAPPY THEIR FEELIN GS WERE SERENE AND PEACEFUL WH ILE MINE BECAME EVERY DAY MORE TUMULTUOUS INCREASE OF KNOWL EDGE ONLY DISCOVERED TO ME MOR E CLEARLY WHAT
A WRETCHED OUTC
pattern found at index 517169 in text: Walden.txt
pattern: S HAVE ATTAINED THE RIGHT ANGL E AND WARM WINDS BLOW UP MIST AND RAIN AND MELT THE SNOWBANK S AND THE SUN DISPERSING THE M IST SMILES ON A CHECKERED LAND SCAPE OF RUSSET AND WHITE SMOK ING WITH INCENS
E THROUGH WHICH THE TRAVELLER PICKS HIS WAY F ROM ISLET TO ISLET CHEERED BY THE MUSIC OF A THOUSAND TINKLI NG RILLS AND RIVULETS WHOSE VE INS ARE FILLED WITH THE BLOOD OF WINTER WHICH THEY ARE BEARI
pattern found at index 583282 in text: Republic.txt
pattern: NOT DERIVED BY THE SEVERAL ART ISTS FROM THEIR RESPECTIVE ART S BUT THE TRUTH IS THAT WHILE THE ART OF MEDICINE GIVES HEAL TH AND THE ART OF THE BUILDER BUILDS A HOUSE ANOTHER ART ATT ENDS THEM WHICH
IS THE ART OF PAY THE VARIOUS ARTS MAY BE DO ING THEIR OWN BUSINESS AND BEN EFITING THAT OVER WHICH THEY P RESIDE BUT WOULD THE ARTIST RE CEIVE ANY BENEFIT FROM HIS ART UNLESS HE WERE PAID AS WELL
pattern found at index 619280 in text: WutheringHeights.txt
pattern: ONS ONLY HARMONIZED BY THE TUR F AND MOSS CREEPING UP ITS FOO
pattern found at index 619372 in text: WutheringHeights.txt
pattern: ERED ROUND THEM UNDER THAT BEN TERING AMONG THE HEATH AND HAR EBELLS LISTENED TO THE SOFT WI ND BREATHING THROUGH THE GRASS AND WONDERED HOW ANY ONE COUL D EVER IMAGINE UNQUIET SLUMBER S FOR THE SLEEP
ERS IN THAT QUI
pattern found at index 9362 in text: PrideAndPrejudice.txt
pattern: IN A FEW DAYS MR BINGLEY RET URNED MR BENNETS VISIT AND SAT ABOUT TEN MINUTES WITH HIM IN HIS LIBRARY HE HAD ENTERTAINE D HOPES OF BEING ADMITTED TO A SIGHT OF THE YOUNG LADIES OF WHOSE BEAUTY HE
HAD HEARD MUCH BUT HE SAW ONLY THE FATHER TH E LADIES WERE SOMEWHAT MORE FO RTUNATE FOR THEY HAD THE ADVAN TAGE OF ASCERTAINING FROM AN U PPER WINDOW THAT HE WORE A BLU E COAT AND RODE A BLACK HORSE
pattern found at index 1393072 in text: Miserables.txt
pattern: S OF THE A B C WERE NOT NUMERO US IT WAS A SECRET SOCIETY IN THE STATE OF EMBRYO WE MIGHT A LMOST SAY A COTERIE IF COTERIE S ENDED IN HEROES THEY ASSEMBL ED IN PARIS IN TWO LOCALITIES NEAR THE FISHMA
RKET IN A WINES HOP CALLED CORINTHE OF WHICH M ORE WILL BE HEARD LATER ON AND NEAR THE PANTHEON IN A LITTLE CAFE IN THE RUE SAINTMICHEL C ALLED THE CAFE MUSAIN NOW TORN DOWN THE FIRST OF THESE MEETI NGPLACES
WAS CLOSE TO THE WORK INGMAN THE SECOND TO THE STUDE
Time: 821.9339592456818 sec
C:\Users\chara\OneDrive\Desktop\PDC>
Type here to search
29°C Haze
ENG
11:59
05-12-2021

RabinKarpParallel
.py import numpy
as np from sys
import argv import
string

set up communication
world comm =
MPI.COMM_WORLD
rank =
comm.Get_rank()
size =
comm.Get_size()

#d value for the rolling
hash d = 26

#splits pattern into specified pattern size before
searching def splitCount(s, count):
  return[s[i:i+count] for i in range(0,len(s),count)]

removes punctuation and capitalizes letters to prep for
processing def prep_text(text):
  exclude = set(string.punctuation)
  return ''.join(x.upper() for x in text if x not in exclude)

#runs the rka of a piece of the pattern on
piece of text def
sub_search(txt,pat,q,matchlist):
  txtlen =
  len(txt) patlen
  = len(pat)
  hashpat = 0
  hashtxt = 0
  h = 1

  assert txtlen > patlen

```
for i in
   range(0,patlen-1):
   # print "creating
   h\n" h = (h*d)%q

for i in range(0,patlen):
   hashpat = (d*hashpat +
   ord(pat[i]))%q hashtxt =
   (d*hashtxt + ord(txt[i]))%q
for i in range(0,txtlen-patlen+1):
```

```
        # print "running through txt\n"
        # check all the letters if the hashes
        match if (hashpat == hashtxt):
            for j in range
                (0,patlen): if
                (txt[i+j] != pat[j]):
                    break
                if j == patlen-1:
                    matchlist.append((i,txt[i:i+pat
                    len]))

        if (i < txtlen-patlen):
            hashtxt = (d*(hashtxt - ord(txt[i])*h) +
            ord(txt[i+patlen]))%q if (hashtxt < 0):
                hashtxt = hashtxt + q

# splits pattern into pieces and calls sub_search on
each piece def
full_search(txt,pat,q,patsize,filecount):
    splitpat=splitCount(pat,patsize)
    matchlist = []
    for subpat in range(0,len(splitpat)):
        sub_search(txt,splitpat[subpat],q,mat
        chlist)
    comm.send(matchlist,dest=0,tag=filecount)

# combines consecutive matches for
entire match def
post_process(patlen,recv_result):
    match_len =
    len(recv_result) result =
    []
    curr = 0
    offset = 1
    if match_len == 1:
        result.append(recv_result[c
        urr]) return result
    else:
```

```
    index,string = recv_result[curr]

while curr <
    match_len:
    nextcurr =
    curr+offset
    if nextcurr < match_len and index + offset*patlen ==
        recv_result[nextcurr][0] : string = string + " " +
        recv_result[nextcurr][1]
        offset +=
    1 else:
```

```
        result.append((index,stri
        ng)) curr = nextcurr
        if curr < match_len:
          index,string                =

  recv_result[curr] return result

# divides files into equal pieces for each slave processor and sends that
part of each # file to processors until no more files need to be
processed and calculates absolute # index once each file returns its
matches
def
  master(filenames,patle
  n): status =
  MPI.Status() text_list
  = []
  files =
  open(filenames).readlines()
  for i in files:
    filename = i.replace('\n',")
    with open (filename,"r") as
      txt: txt =
      txt.read().replace('\n',' ')
    text_list.append((filename,prep_text(txt)))

    #print "txtlen %d"
  %txtlen # run
  numfiles =
  len(text_list) k =
  size-1
  #print 'numfiles: %d' %(numfiles)
  #keeps a count of which file the slave processor is on and if there
  are any left #for it to process
  count = [0]*k

  #counter of received processed file parts per
  processor received = 0
  total = numfiles*k
```

```
#print 'total %d'
%(total)

filecount = 0

#initialization of first file in all
processors name,txt = text_list[0]
txtlen = len(txt)
#print 'txtlen %d' %txtlen
```

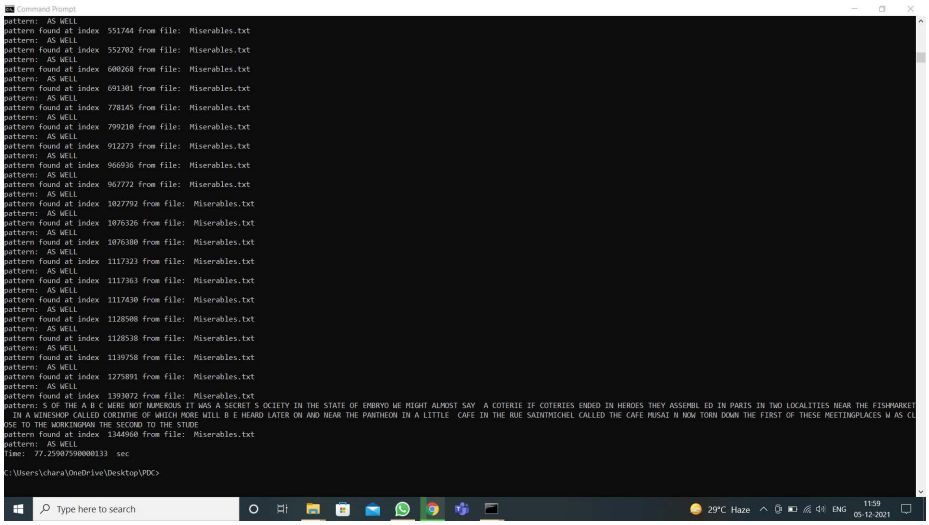
```
for i in range(1,size):
   start = int(round((i-1)*(txtlen-patlen+1)/k))
   end = int(round(i*(txtlen-
   patlen+1)/k)+(patlen-1)) #print 'start %d'
   %start
   #print 'end %d' %end
   send_data =
   txt[start:end]
   #need to keep track of start so we know the
   absolute index
   comm.send(send_data,dest=i,tag=filecount)

while received <
   total: for i in
   range(1,size):
      recv_result =
      comm.recv(source=i,tag=MPI.ANY_TAG,status=status)
      slave = i
      name,txt =
      text_list[filecount] txtlen
      = len(txt)
      start = int(round(((slave-1)*(txtlen-patlen+1)/k)))
      #post processing for combining consecutive
      matches result = []
      match_len =
      len(recv_result) if
      match_len > 0 :
         result =
      post_process(patlen,recv_result)
      else:
         result = recv_result
      #print out results received from that
      slave for index, match in result:
         abs_index = start+index
         print ("pattern found at index %d from file: %s"
         %(abs_index,name)) print ("pattern: %s" %match)
            #calculate absolute index
```

```
#update total number of received
parts received += 1
#print 'received %d' %received

#update to next file to
process currfile =
filecount+1
#send the next file part to processor when it's
finished #each slave must do a part in each
file
if currfile < numfiles:
```

```
            #print 'filecount sending out to slave %d: %d'
            %(i,currfile) name,txt = text_list[currfile]
            txtlen = len(txt)
            #print 'txtlen: %d' %txtlen
            start = int(round(((slave-1)*(txtlen-patlen+1))/k))
            end = int(round(slave*(txtlen-
            patlen+1)/k)+(patlen-1)) send_data =
            txt[start:end]
            comm.send(send_data,dest=slave,tag=currfile)
        filecount += 1

#stop all processes when everyone
   is done for s in range(1,size):
      comm.send(-1,dest=s,tag=100)


#conducts the rka on its piece of
the text def slave(pat,q,patlen):
   status =
   MPI.Status()
   while True:
      local_data =
      comm.recv(source=0,tag=MPI.ANY_TAG,status=status)
      currfile = status.Get_tag()
      #print 'filecount in slave %d: %d'
      %(rank,currfile) #break out of the while loop
      if all processes are done if local_data == -1:
         break
      full_search(local_data,pat,q,patlen,c
      urrfile)

##################################----MAIN
 ------------------------------------------------ ####################################
##
#########

if __name_== '__main_

   ': if len(argv) != 3:
```

```
    print ("Usage: mpiexec -n [# of processors] python", argv[0], "[corpus
filenames] [input text]")
    exit()

  # distribute data to other processes to do
  computations filenames, pattxt = argv[1:]
  with open (pattxt,"r") as patfile:
```

```
    pat=patfile.read().replace('\

n',' ') pat = prep_text(pat)

#size of pattern we want to
match patsize = 50

q = 1079

if rank == 0:
    start = MPI.Wtime()
    master(filenames,pats
    ize) end =
    MPI.Wtime()
    print ("Time: %f sec" %(end-
start)) else:
    slave(pat,q,patsize)
```



Result observations:

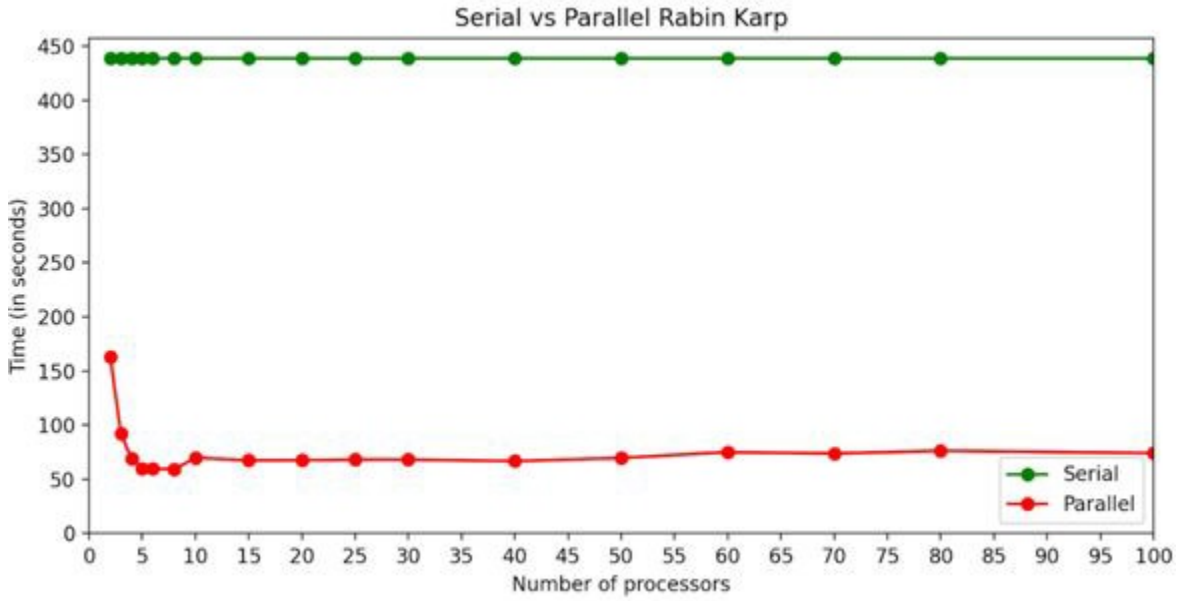
We have noted the time taken by the parallel algorithm when the number of processors is varied. Time taken by the serial algorithm is also noted and the two have been compared. The results are given below.

No. of processors (p)	Serial execution time (ts) (in seconds)	Parallel execution time (tp) (in seconds)	Speedu p

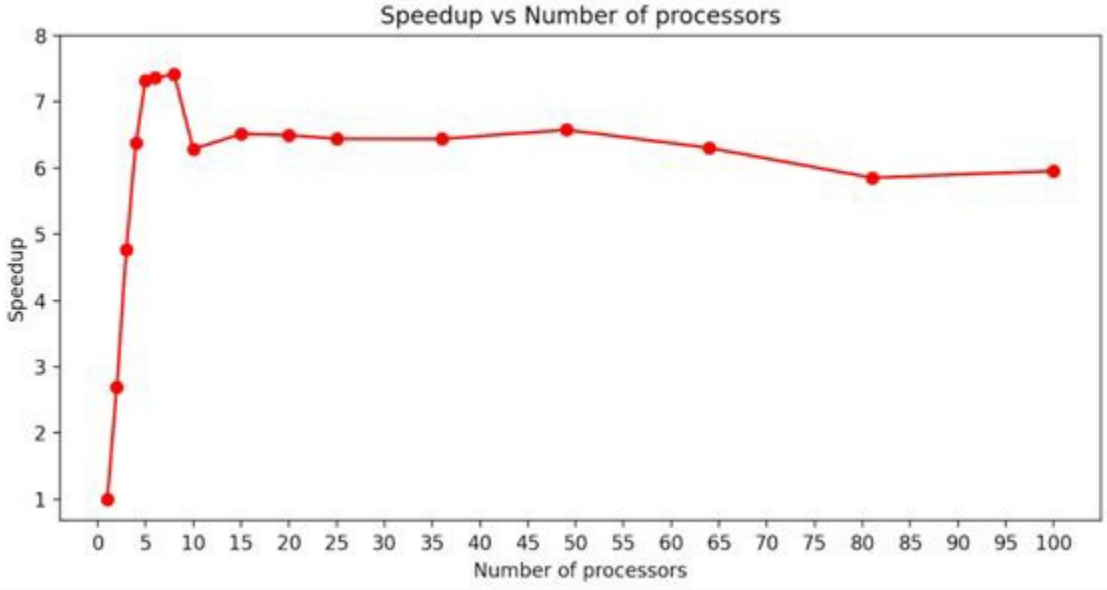
			(ts/tp)

2	438.611527	163.031782	2.690343
3	438.611527	91.901379	4.772632
4	438.611527	68.720557	6.382537
5	438.611527	59.958593	7.315240
6	438.611527	59.580127	7.361708
8	438.611527	59.144965	7.415872
10	438.611527	69.781191	6.285526
15	438.611527	67.347746	6.512638
20	438.611527	67.523261	6.495710
25	438.611527	68.100446	6.440655
30	438.611527	68.138343	6.437073
40	438.611527	66.698307	6.576051
50	438.611527	69.598556	6.302020
60	438.611527	74.943374	5.852572
70	438.611527	73.662477	5.954341
80	438.611527	76.355119	5.744363

100	438.611527	74.121631	5.917456



From the above graph, we can observe that while the serial algorithm took nearly 450 seconds, the parallel algorithm took only about 60 seconds for execution. We can also observe that initially, we were able to reduce the parallel execution time by increasing the number of processors. However, the time remained unchanged, at around 75 seconds, when the number of processors was increased beyond 10.



From the above graph, we can observe that a speedup of more than 6 times was achieved by parallelizing the algorithm.

Kaggle Link:

---

https://www.kaggle.com/charan2206/rabinkarp

Conclusion:

Rabin Karp algorithm is a pattern matching algorithm using a rolling hash. We have successfully parallelized the Rabin Karp algorithm using MPI. We observed a significant reduction in the time taken by the parallel algorithm as compared to the serial algorithm. A speedup of more than 6 times was achieved by the parallel algorithm.

Future Enhancements:

Methods to further reduce the time taken by the parallel algorithm can be studied. Application of the parallel algorithm in plagiarism detection can also be explored.

References:

[1] Maurice Herlihy, Nir Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann Publisher, 2008

[2] M. C. Schatz, C. Trapnell, Fast Exact String Matching on the GPU, Technical report

[3] S. Viswanadha Raju, A. Vinayababu, Optimal Parallel Algorithm for String Matching on Mesh Network Structure, International Journal of Applied Mathematical Sciences, Vol.3 No.2(2006), pp. 167-175

[4] T. F. Smith, M. S. Waterman, Identification of common molecular subsequences, J Mol Biol, vol. 147, pp. 195-197, 1981

[5] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, Introduction to Algorithms, The MIT Press

[6] Y. Utan, M. Inagi, S. Wakabayashi, and S. Nagayama, "A GPGPU implementation of approximate string matching with regular expression operators and comparison with its FPGA implementation," in Proc. Int. Conf. Parallel and Distributed Processing Techniques and Applications, 2012.

[7] W.-m. W. Hwu, GPU Computing Gems Emerald Edition, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2011.

[8] NVIDIA Corporation, "NVIDIA CUDA C Programming guide," http: //docs.nvidia.com/cuda/cuda-c-programming-guide/, April 2015.s, 2005.