

**PARALLEL AND DISTRIBUTED COMPUTING (CSE4001)**

**TITLE: PARALLEL WORD SEARCH USING OPENMP**

**FINAL REPORT**

**Faculty : Venkatraman.S**

|  |  |
| --- | --- |
| NAME | REGISTRATION NUMBER |
| Sai Vignesh | 19BCE1389 |
| Yeshwanth P | 19BCE1862 |

**ABSTRACT :**

Word search is used everywhere from local page search (CTRL + F) to searching words on document viewer like “reader” in windows. In fact a whole branch called Information Retrieval was developed for this.

It has a lot of application in real world. As the name suggests “word search” is about searching words in documents parallely using openmp. This is not just a simple word search but also displays the frequency of a particular word in a file and also find the Relevance Percentage of the word with the files. Our Project also compares and displays the time for parallel and series word search and makes a comparison between them.

**PROBLEM STATEMENT :**

Everything has become digital nowadays and every information is stored as document and pdf nowadays. While Readingthese documentswe use word searching often. While using this sometimes it takes some time while finding the occurrence and position of the word. To solve this problem in our project we have introduced the conecept of OPEN-MP where we can search the words parllely with the help of multiple threads and find their occurrences. This reduce the time of series search to a large extent and is also also very much effective when compared to it. We will Implement a model which uses this concept for word search and compare its efficiency with normal series search.

**ADDITIONAL LITERATURE SURVEY :**

**Citation 1 :**

**Parallel String Matching for Urdu Language Text**

Mirza Baber Baig(&) and Taoshen S. Li Guangxi University, 100, Daxue Road, Nanning 530004, Guangxi, People’s Republic of China.

**SUMMARY:**

This paper is based on pattern identification from sequences of Urdu language text and evaluating the parallelization performance using OpenMP

Urdu language is the language used in Pakistan. Their sacred documents kuran is written in the language of Urdu. Here they use the following algorithms:

1. **Boyer-Moore string matching**
2. **Knuth-Morris-Pratt string matching**
3. **Sunday string matching**

These 3 algorithms are used for pattern matching and then the time taken for executing and matching the pattern is found. This process is repeated with increase in the number of threads and for each increment of 2 threads the experiment results are noted. Then this is repeated for different file sizes and the results are noted. Then speedup is calculated with number of threads and file size ad the results are plotted as a bar graph

 From the results, the following inferences are drawn:

The experimental comparative results on multi-core computer show that the number of running threads impacts the required time of parallelized BM, BF, Sunday and KMP string matching algorithms, the required time of parallelized BM and Sunday algorithms is much less than that of the parallelized BF and KMP algorithm.

Totally, the parallelized BM string matching algorithm is the fastest one among the four parallel string matching algorithms

Citation 2 :

**Performance evaluation of counting words from files using OpenMP**

**Authors** : Prof. Smita Agrawal 1,3 , Prof. Preeti Kathiria 1,3 , Atul Patel 4,5 , GUNJAN ASWANI(12MCA02) 2, 3 and BINDIBEN BHATT(12MCA05) 2,3

**Summary :**

Here the authors have planned to implement a word count engine using Open-MP. After Implementing this they try to compare it with the sequential word count model and compute how efficient their model is.

This Implementation is done in the language C using #include<omp.h> header file.

The Observation the authors in the evaluation time of both the models is as given below

The parallel code execution and time details:

Real 0m0.021s

User 0m0.052s

Sys 0m0.000s

Sequential code Execution and time details:

Real 0m0.030s

User 0m0.041s

Sys 0m0.000s

We can clearly observe how efficient is the parallel code using open mp when compared to serial code is.And they also observe that as the no of threads increases the execution time decreases.

As the Future Scope they state that they will also compute the frequency of all the words in a particular file and fix some of the limitations present due to open MP.

**CITATION 3:**

**Parallel Quicksort Algorithm using OpenMP**

- Sinan Sameer Mahmood Al-Dabbagh1 , Nawaf Hazim Barnouti2 ¹Software Engineering and Information Technology Department, Al-Mansour University College, Baghdad, Iraq ² Al-Mansour University College, Baghdad, Iraq

## SUMMARY OF THE PAPER:

This paper mainly aims to parallelize quicksort algorithm using the OpenMP and do analysis on it. The algorithm that they followed for it is:

Creating many additional temporary sub-arrays according to a number of characters in each word, the sizes of each one of these sub-arrays are adopted based on a number of elements with the exact same number of characters in the input array. The elements of the input datasets is distributing into these temporary sub-arrays depending on the number of characters in each word

So then the efficiency of the algorithm is tested with sorting the words of two text files which has words in a random order and for each iteration the number

of threads are increased and the sort timings are noted

So in the end they have tried to calculate speed up and efficiency with respect to the number of threads. So from the experiments they have concluded that with the increase in the number of threads and cores the efficiency and speed up parameters also increase and they have concluded that for sorting those 2 files 4 threads with 4 cores gave the best speedup and efficiency

## CITATION 4:

**Performance Analysis of Parallel Algorithms on Multi-core System Using OpenMP**

- Sanjay Kumar Sharma1 , Dr. Kusum Gupta2

Departemtne of Computer Science, Banasthali University, Rajasthan, India

## SUMMARY OF THE PAPER:

In this paper they try to evaluate the performance of parallel algorithms compared to the sequential execution with the help of linear equations. They take some number of linear equations and try to solve it in Gauss elimination as well as with integration in both sequential and parallel manner and have tried to note down the readings of the execution times.

Algorithm for gauss elimination:

1. Input: a [1: n, 1: n+1] // Read the matrix data
2. Output: x [1: n]
3. Set the numbr of threads
4. Strat clock
5. // Traingularization process
6. for k = 1 to n-1
7. Insert #pragma directive to parallelize
8. for i = k+1 to n
9. mi,k = ai,k / ak,k
10. for j = k to n+1
11. ai,j = ai,j – mi,k \* ak,j
12. End loop [j]
13. End loop [i]
14. End loop [k]
15. // Back substitution process
16. xn = an,n+1 / an,n
17. for i = n-1 to 1 step -1 do
18. sum = 0
19. for j = i+1 to n do
20. sum = sum + ai,j \* xj
21. End loop [j]
22. xi = ( ai,n+1 - sum )/ai,i
23. End loop[i]
24. Stop clock
25. Display time taken and solution vector

So with this they calculate the solution for the linear equations in both linear and parallel manner and try to calculate the speed up

They tabularize the results and from that they try to draw conclusions. The conclusions that they draw from the experiment are:

1. They see that parallelizing serial algorithm using OpenMP has increased the performance.
2. For multi-core system OpenMP provides a lot of performance increase and parallelization can be done with careful small changes.
3. The parallel algorithm is approximately twice faster than the sequential and the speedup is linear.

They have also mentioned that there is huge scope for this in future as OpenMP is getting popular to learn parallelism and stuff and its also easy to implement and use

## CITATION 5:

**Optimizing Multi-Core Algorithms for Pattern Search**

Veronica Gil-Costa1,2, Cesar Ochoa1 and Marcela Printista1,2 1 LIDIC, Universidad Nacional de San Luis, Ejercito de los Andes 950, San Luis, Argentina 2 CONICET, Argentina

## SUMMARY OF THE PAPER:

This paper deals with the way now search engines work where the user is required to enter the words or sentence or phrases that they need to search. The documents are stored in full text model where abstracts are there for each and every document.

Then the words or phrases are searched based on the following algo:

They divide the pattern query processing operation into two steps. In the former, all threads work to classify queries into four groups. In the last step, threads search the queries using the Suffix Array index. Both steps are separated by a barrier synchronization to avoid data corruption.

The proposed static scheduling algorithm which aims to prune those parts of the index which will not be accessed by any query. Then four local queues assigned to different parts of the index. The partition criterion is set according to the SA centers. Then, all threads compare the incoming queries with the centers (three centers) of the suffix array. According to the comparison result, each query is assigned to a local queue. Pattern queries matching any of the three centers are considered solved. Therefore, they are not included in any local queue for processing in the second step. After all incoming queries are pre- processed, we assign threads to each part of the index. The basic idea is to allocate threads according to the workload. To this end, we compute the minimum number of threads required by each partition:

minThreads[i] = (Local\_queue[i].size()/totalQ)\*NT if ( tid < minThreads[i] ) idGroup = i

where Local\_queue[i].size() is the number of queries assigned to the index partition i, with i={1,2,3,4}. totalQ is the total number of queries that have to be

processed in the second step, and NT is the number of threads. Then, each thread determines its index partition (idGroup) using its thread identifier (tid). Notice that in Fig. 5 the fourth index partition has no query assigned to the local queue. Therefore, no thread is allocated to this partition.

So this algorithm is applied and the efficiency for each increment on the number of threads are calculated. Then graphs are plotted and the results are noted.

The inference they got is that the scheduler algorithm presents a balance workload in the first step, as all threads reports an efficiency of 99%. In the second step, efficiency goes down to 74% in average. Namely, some threads support a workload 30% higher. The baseline suffix array parallel algorithm also presents an efficiency of 99%. In other words, all threads perform almost the same amount of comparisons. This is due to all queries are evenly processed by threads

**Limitations in Research Paper :**

CITATION 1 :

* The Algorithm is not generalised to all the languages
* They compare the speedup that they got with the help of the number of threads but they stopped with 10 so we are not able to see how much impact the fast processing makes
* They have used single core for string pattern matching whereas there are multiple cores in the system which they could have used

CITATION 2 :

* The Algorithm is not generalised to all the languages
* Using this word search technique one cannot find the frequency of all the words in a file.
* The frequency of the words is not correct while using open mp in some cases.

CITATION 3 :

* It is recursive. Especially, if recursion is not available, the implementation is extremely complicated.
* It requires quadratic (i.e., n2) time in the worst-case.
* It is fragile, i.e. a simple mistake in the implementation can go unnoticed and cause it to perform badly.

CITATION 4 :

* Matrix Multiplication is not Commutative
* Matrix multiplication even when executed in parallel is a time consuming process

CITATION 5 :

* The scheduling algorithm uses static scheduling which gives less efficiency use compared to the dynamic,auto and guided
* The algorithm uses only 2 threads to seperate the total array as right and left array
* There are more overheads as depending on the number of elements there is a possibility that the queue may get full

**Implementation :**

We Implemented this project in C++ Language. We first got the path of the file which needed to be searched and declared a 2d string array which stores the words from the file to the array. Then we compare the working of series search and parallel search and observe how efficient the parallel search is.

**Screenshots :**

**Code :**

#include<iostream>

#include<bits/stdc++.h>

#include <omp.h>

using namespace std;

int main()

{

    map<pair<int,string>, int> m;

    bool b = true;

    short choice;

    while(b)

    {

    cout<<endl;

    cout<<"1.Parllel Search"<<endl;

    cout<<"2.Series Search"<<endl;

    cout<<"3.Exit"<<endl;

    //cout<<"4 = quit "<<endl;

    cin >> choice;

    if (choice == 1)

    {

         string str="";

  // ofstream fo("/var/www/html/pagerank/display.php");

         //ofstream fo2("/var/www/html/pagerank/insert.php");

              //omp\_set\_nested(1);

              //omp\_set\_dynamic(0);

  string list[1000][6];

  std::fill(list[0], list[0] + 1000 \* 6, "0");

  int count = 0;

  //map<string,int>m;

  int temp;

  int numthreds;

  //fo2 << "<?php include('config.php'); "<<endl;

  //fo2<< "DELETE FROM list"<<endl;

    cout<<"Enter no of threads"<<endl;

    cin>>numthreds;

    double start\_time = omp\_get\_wtime();

    #pragma omp parallel for private(temp)  num\_threads(numthreds)

        for(int j=1;j<=5;j++)

           {

              string temp3;

              temp3 = "file" + to\_string(j) + ".txt";

              ifstream fi(temp3);   //"file" + temp2 + ".txt"

                  while (fi >> str)

                  {

                    //cout<<str<<endl;

                    int flag=0;

                    int i;

                    int temp23;

                    int flag2 = 1;

                       #pragma omp parallel for  num\_threads(numthreds)

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

                              {

                                  if(list[i][0]==str&&flag2==1)

                                  { //int b = atoi(a.c\_str());

                                    //m[list[i][0]]++;

                                    flag=1;

                                    temp = atoi(list[i][j].c\_str());//(int)list[i][j];

                                    temp++;

                                                                 //fo2<<str<<"$var=" << temp;

                                        //fo2<<"$query = \"UPDATE `list` SET `"<<j<<"`="<<temp<<" WHERE `string`='"<<str<<"';\";"<<endl;

                                        //fo2<<"mysqli\_query($con,$query);"<<endl;

                                                stringstream convert;

                                                convert << temp;

                                                list[i][j] = convert.str();

                                                flag2 = 0;

                                                //break;

                                  }

                             }

                if(flag==0)

                  {

                                            //count++;

                       //#pragma omp critical

                           {

                              list[count][0] = str;

                              //m[str]++;

                              list[count][j] = '1';

                              count++;

                              //fo2 << "$var=\""<<str<<"\";"<<"\n";

                              //fo2 << "$query = \"INSERT INTO `list`(`string`,`"<<j<<"`)VALUES ('\".$var.\"',1);\";"<<endl;

                              //fo2 << "mysqli\_query($con,$query);"<<endl;

                           }

                  }

    }

  }

//fo2<<"header(\"Location:display.php\");?>";

  /\*for (int i = 0; i < count; ++i)

  {

    //cout<<left<<setw(15)<<list[i][0]<<setw(4)<<list[i][1]<<setw(4)<<list[i][2]<<setw(4)<<list[i][3]<<endl;

    for (int j = 0; j < 4; ++j)

    {

      cout<<left<<setw(5)<<list[i][j]<<"    ";

    }

    cout<<endl;

  }\*/

 //cout<<count;

string str1;

double time = omp\_get\_wtime() - start\_time;

cout<<"Enter the term to be searched:"<<endl;

cin>>str1;

int flag1=0;

int result;

int freq=0;

int tmp=0;

//#pragma omp parallel for

for (int i = 0; i < count; ++i)

{

  //cout<<list[i][0]<<endl;

  //cout<<"Frequency of the word"<<m[list[i][0]]<<endl;

  if(list[i][0] == str1)

  {

    flag1=1;

    result = i;

    freq++;

    //tmp++;

    //break;

  }

  else

  {

    //tmp++;

  }

  tmp++;

}

//cout<<"Frequency of the word is"<<freq/tmp<<endl<<"total"<<tmp<<endl;

if(flag1==0)

  cout<<"No Results Found!!!"<<endl;

else

{

  vector< pair<int, int> > v;

  vector< pair<int, int> >::iterator it;

  for (int i = 0; i < 5; ++i)

  {

    v.push\_back(make\_pair(atoi(list[result][i+1].c\_str()),i+1));

      //atoi(list[result][i+1].c\_str())

  }

  sort(v.rbegin(),v.rend());

  int index=1;

  //cout<<"Rank"<<"\t"<<"Document"<<"\t"<<"frequency"<<endl;

for(it=v.begin();it!=v.end();it++)

{

  //cout<<"frequency=>"<<it->first << "  " <<" doc.ID=>"<<it->second<<endl;

  //cout<<index++<<"=> "<<setw(10)<<"file"<<it->second<<".text"<<setw(10)<<it->first<<endl;

}

  v.clear();

}

//list[count][i+1]

int totalcount=0;

for(int j=1;j<=5;j++)

  {

    cout<<"the occurence of the word"<<str1 <<" in file"<<j<<" is :"<<" "<<m[{j,str1}]<<endl;

    totalcount+=m[{j,str1}];

  }

  cout<<"Total count :"<<totalcount<<endl;

  cout<<"Size :"<<m.size()<<endl;

  double percent=(totalcount\*100/m.size());

  cout<<"The Relationship between the files and the word is"<<" "<<percent<<endl;

    //cout<<m.size()<<endl;

  cout<<"Time for search"<<" ";

  printf("%lf",time);

  //cout<<endl;

  //return 0;

}

    else if (choice == 2)

    {

        string str;

  //ofstream fo("/var/www/html/pagerank/display.php");

  //ofstream fo2("/var/www/html/pagerank/insert.php");

  string list[10000][6];

  std::fill(list[0], list[0] + 10000 \* 6, "0");

  int count = 0;

  int temp;

  //fo << "hi";

  //fo2 << "<?php include('config.php'); "<<endl;

  //fo2<< "DELETE FROM list"<<endl;

    double start\_time = omp\_get\_wtime();

  //map<pair<int,string>, int> m;

  for(int j=1;j<=5;j++)

  {

    string temp3;

    temp3 = "file" + to\_string(j) + ".txt";

    ifstream fi(temp3);   //"file" + temp2 + ".txt"

    while (fi >> str)

    {

      int flag=0;

      int i;

      int temp23;

                        int flag2 = 1;

      for (i = 0; i < count; ++i)

      {

        if(list[i][0]==str&&flag2==1)

        { //int b = atoi(a.c\_str());

            m[{j,list[i][0]}]++;

          flag=1;

          temp = atoi(list[i][j].c\_str());//(int)list[i][j];

          temp++;

          //fo2<<str<<"$var=" << temp;

          //fo2<<"$query = \"UPDATE `list` SET `"<<j<<"`="<<temp<<" WHERE `string`='"<<str<<"';\";"<<endl;

          //fo2<<"mysqli\_query($con,$query);"<<endl;

          stringstream convert;

          convert << temp;

          list[i][j] = convert.str();

                                         flag2 = 0;

        }

      }

      if(flag==0)

        { //count++;

          list[count][0] = str;

          m[{j,str}]++;

          list[count][j] = '1';

          count++;

          //fo2 << "$var=\""<<str<<"\";"<<"\n";

          //fo2 << "$query = \"INSERT INTO `list`(`string`,`"<<j<<"`)VALUES ('\".$var.\"',1);\";"<<endl;

          //fo2 << "mysqli\_query($con,$query);"<<endl;

        }

    }

}

//fo2<<"header(\"Location:display.php\");?>";

  /\*for (int i = 0; i < count; ++i)

  {

    //cout<<left<<setw(15)<<list[i][0]<<setw(4)<<list[i][1]<<setw(4)<<list[i][2]<<setw(4)<<list[i][3]<<endl;

    for (int j = 0; j < 4; ++j)

    {

      cout<<left<<setw(5)<<list[i][j]<<"    ";

    }

    cout<<endl;

  }\*/

 //cout<<count;

string str1;

cout<<"Enter the term to be searched:"<<endl;

cin>>str1;

int flag1=0;

int result;

for (int i = 0; i < count; ++i)

{

  //cout<<list[i][0]<<endl;

  //cout<<"Frequency of the word"<<m[list[i][0]]<<endl;

  if(list[i][0] == str1)

  {

    flag1=1;

    result = i;

    //break;

  }

}

if(flag1==0)

  cout<<"No Results Found!!!"<<endl;

else

{

  vector< pair<int, int> > v;

  vector< pair<int, int> >::iterator it;

  for (int i = 0; i < 5; ++i)

  {

    v.push\_back(make\_pair(atoi(list[result][i+1].c\_str()),i+1));

      //atoi(list[result][i+1].c\_str())

  }

  sort(v.rbegin(),v.rend());

  int index=1;

  //cout<<"Rank"<<"\t"<<"Document"<<"\t"<<"frequency"<<endl;

for(it=v.begin();it!=v.end();it++)

{

  //cout<<"frequency=>"<<it->first << "  " <<" doc.ID=>"<<it->second<<endl;

  //cout<<index++<<"=> "<<setw(10)<<"file"<<it->second<<".text"<<setw(10)<<it->first<<endl;

}

  v.clear();

}

//list[count][i+1]

double time = omp\_get\_wtime() - start\_time;

    for(int j=1;j<=5;j++)

  {

    cout<<"the occurence of the word"<<str1 <<" in file"<<j<<"is :"<<" "<<m[{j,str1}]<<endl;

  }

    cout<<"The time for search is"<<" ";

     printf("%lf",time);

     cout<<endl;

    }

     else if (choice == 3)

    {

        b=false;

    }

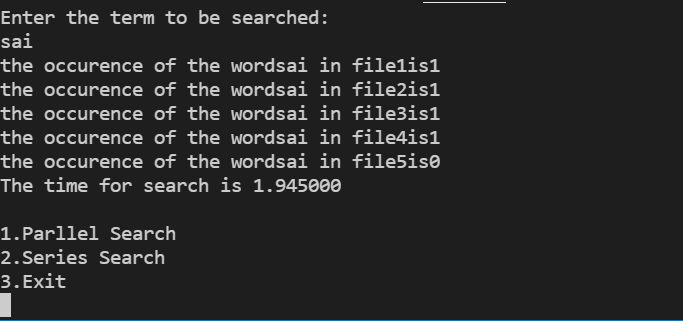
    }

    return 0;

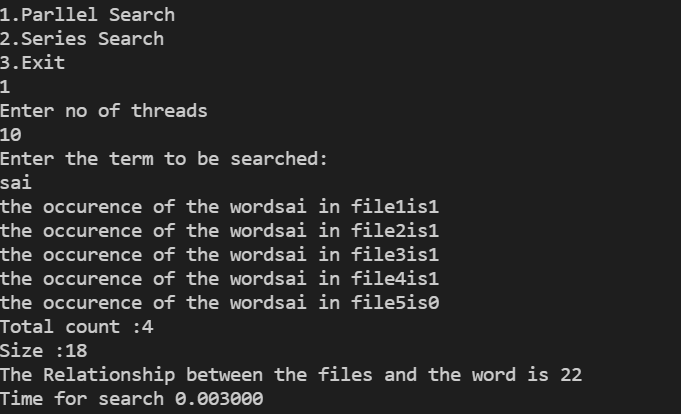
}

**Output Screenshots :**

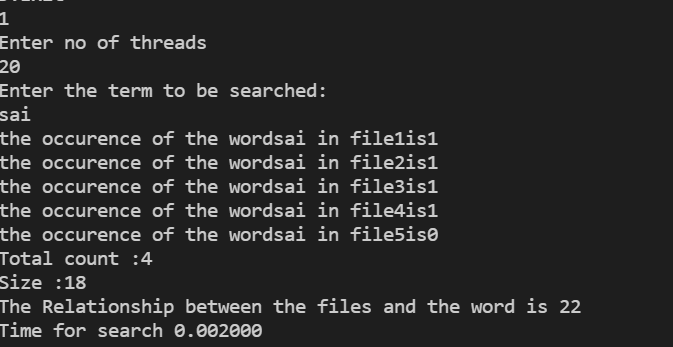
(Here in the given 5 files the word “sai” is searched in series search. Note the Return time)



(Now if you observe the same word “sai” when searched in parallel search takes much less time when compared to series search)



(When the no of threads increases the search time decreases)



**Observations :**

* Here as the no of threads increases the time decreases in parallel search
* While series search do not have any effect
* If we keep on increasing the number of threads we can see that the time in parallel search becomes stagnant/constant. This means that increasing the threads further more does not have any effect.
* As the no of words in the files increases the search time increases.
* Our Project apart from word search also records the relationship of the words occurrences with the file.
* Which is calculated by (Occurence of that word)/(Total no of words in all files) \* 100

**Limitations :**

* This Parllel search program collapses sometimes when run continuosly as the openmp is exhausted of threads.
* The frequency calculation is random sometimes.
* Cannot tell the position of the words in the file.

**Future Scope :**

The Limitations mentioned above can be solved by defining proper critical section for the pragma omp loops. And to find the position of the word in the file we can use data structures like map to store the index of the occurrence. The model developed in this project can be developed like Adobe file reader and pdf viewer which is one of the leading companies in this field. And the time for searching a file in a large file can be imensly reduced.

**REFERENCES:**

* 1. **Parallel Quicksort Algorithm using OpenMP -**Sinan Sameer Mahmood Al-Dabbagh1 , Nawaf Hazim Barnouti2 ¹Software Engineering and Information Technology Department, Al-Mansour University College, Baghdad, Iraq ² Al-Mansour University College, Baghdad, Iraq
  2. **Performance Analysis of Parallel Algorithms on Multi-core System Using OpenMP-**

Sanjay Kumar Sharma1 , Dr. Kusum Gupta2 Departemtne of Computer Science, Banasthali University, Rajasthan, India

* 1. **Optimizing Multi-Core Algorithms for Pattern Search** -

Veronica Gil-Costa1,2, Cesar Ochoa1 and Marcela Printista1,2 1 LIDIC, Universidad Nacional de San Luis, Ejercito de los Andes 950, San Luis, Argentina 2 CONICET, Argentina

* 1. **Performance and Power Analysis of Parallelized Implementations on an MPCore Multiprocessor Platform –**

H. Blume, J. v. Livonius, L. Rotenberg, T. G. Noll

* 1. **Parallel String Matching for Urdu Language Text -**

Mirza Baber Baig(&) and Taoshen S. Li