PARALLEL IMPLEMENTATION OF USER QUERY SEARCH

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology

in

COMPUTER SCIENCE

by

DHANANJAY SUNIL MENON	(18BCE2330)
GOKUL RAJ	(18BCE2308)
HRITHIK AHUJA	(18BCE2154)
ROHAN ALLEN	(18BCI0247)

CSE4001_PARALLEL AND DISTRIBUTED COMPUTING

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

October,2020

DECLARATION

I hereby declare that the thesis entitled *Parallel Implementation of User Query Search* is submitted by us, for

the award of the degree of *Bachelor of Technology in Computer Science and Engineering*

to VIT is a record of bonafide work carried out by me under the supervision of Mr Murugan K.

I further declare that the work reported in this thesis has not been submitted and will

not be submitted, either in part or in full, for the award of any other degree or diploma in

this institute or any other institute or university.

Place: Vellore
Date : 25/10/20

Dhananjay Sunil Menon
Gokul Raj
Hrithik Ahuja
Rohan Allen
Signature of the Candidate

ACKNOWLEDGEMENTS

We would like to take the opportunity to thank all the people who helped us. Firstly, we would like to express our sincere gratitude to Mr Murugan K, Senior Professor, SCOPE, Vellore Institute of Technology, for his valuable guidance, his continuous support and understanding throughout the duration of the project. We are highly indebted for his constant supervision and his knowledge in regards to the field. Working with her has proved to be a very good opportunity for both of us.

We would also like to thank the teaching and non-teaching staff of Vellore Institute of Technology for their non-self-centered enthusiasm and provided an environment to work in which further prompted us to complete the project successfully.

Finally, we would also like to thank our family and friends for their constant support and help for the successful completion of the project.

Dhananjay Sunil Menon
Gokul Raj
Hrithik Ahuja
Rohan Allen

Executive Summary

The word search used in windows explorer can find documents based on name. But imagine you wanted to search for a word within a document, and you had to go through an entire file of documents. This would be time consuming manually and serially. In this project, parallelism can be used to find the document with the highest occurrences of the word. Also a table will contain the number of times a word has occurred in each document. Multi-threaded systems will be used for search purposes, and this would be compared to the sequential search of words (in a separate program). The comparison would be made based on time taken for different circumstances for both serial and parallel searches.

Scope Of This Project

- Get User Query and remove stop words and split the words
- Search all the documents based on user search
- Display the table to show occurrences of word
- Sort documents and show the highest frequency word containing document

CONTENTS

INTRODUCTION

Page Ranking is being used everyday in our lives. Google works on getting a user query search and displaying the results almost instantly. Similarly, we can create a program to search all the words in different files and get the document with the highest occurrences of the word. We have found different ways to do this. One by serially searching for the query, and the others by parallel searching for the query. Later we can compare the performance in both systems and derive a conclusion

Objectives of The Project

- Finding documents having the query searched.
- Constructing a table based on occurrences of the word
- Sorting documents based on table
- Comparative study of parallel and linear search

Problem Statement

We have many documents each named file i (where i is the number of the file) saved in a particular folder. The objective of our program is to accept a query from the user, split the query into key words and finally print the occurrence table and sort all the files accordingly.

The time for the word search algorithm to work would be recorded and the performance of both systems would be compared accordingly.

For storing the words onto a table, a 2D vector is used in both cases. In serial implementation, the vector has a dynamic memory. But for parallel implementation, the vector has already allotted space because the vector functions cannot work in a parallel manner. Hence, the parallel implementation model consumes a lot of memory.

While comparing execution time of less number of files, the time taken are almost the same. But for a large number of files, there appears to be a bigger difference.

Literature Survey

OPENMP

OpenMP consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

When we use OpenMP, we use directives to *tell* the compiler details of how our code should be run in parallel.

OpenMP uses a portable, scalable model that gives programmers a simple and flexible interface for developing parallel applications for platforms ranging from the standard desktop computer to the supercomputer.

Linear Search

Linear search is one of the simplest algorithms to implement. It has the worst case complexity of **O(n)**, The algorithm has to scan through the entire list to find the element if the required element isn't in the list or is present right at the end. Each file is accessed one by one and each word of the file is accessed one at a time too.

Parallel Search

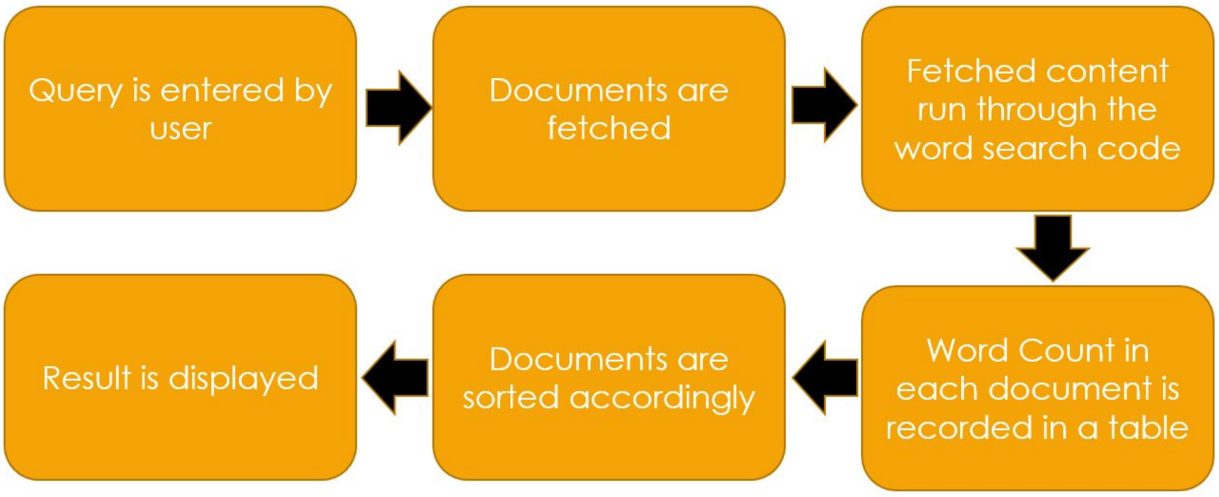
To parallelize the for loop, the openMP directive is: #pragma omp parallel for. This directive tells the compiler to parallelize the for loop. So files are accessed at the same time and each line in a file is accessed in parallel too.

Threads and Processors

Threads share all memory in the program, but there are multiple threads of execution. In OpenMP, 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. Threads are virtual, and help to generate parallelism in a program.

In processes, all memory is local to each processor except memory that is explicitly shared. Processors/cores are physical components.

Proposed Model


Query is entered by user
Documents are fetched
Fetched content run through the word search code
Word Count in each document is recorded in a table
Documents are sorted accordingly
Result is displayed

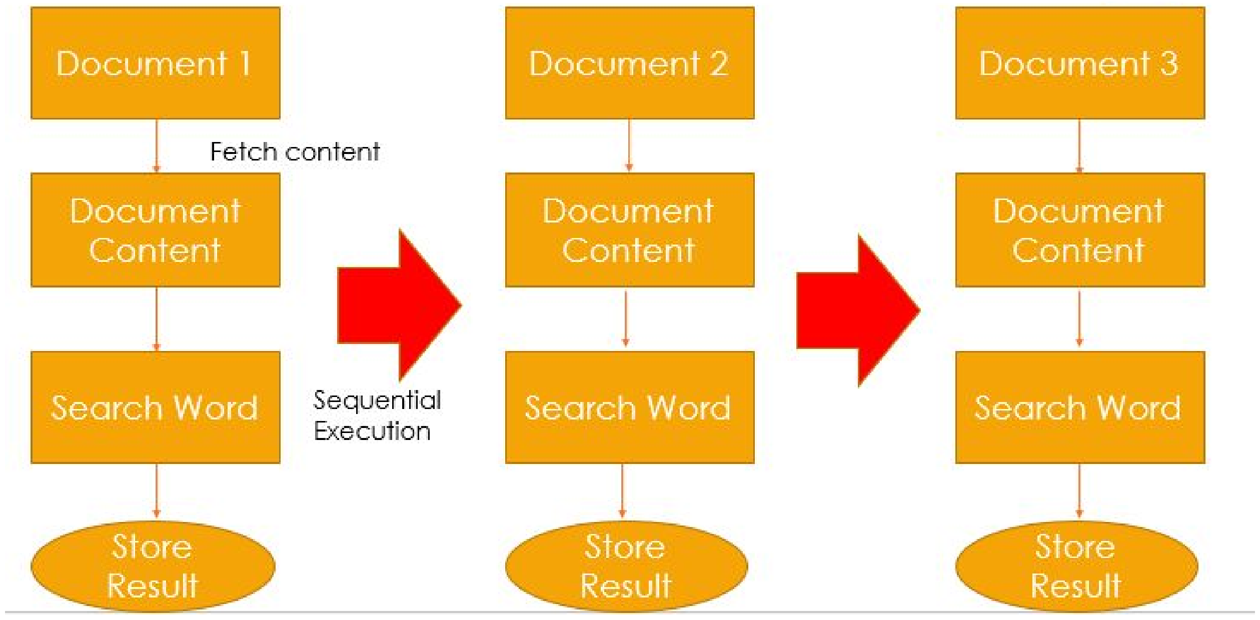
Algorithm

- First words are searched in all the documents and text files in a sequential manner.
- The number of occurrences of each word in the document is stored first. Only then do we go and search for the next document.
- We use linear search to perform this sequential searching of documents
- Now words are searched in all the documents and text files in a parallelized manner, i.e. all the documents are searched for that particular word concurrently.
- The number of occurrences of each word in all the documents as and when they have completed searching them.
- We use OpenMP library in C programming and : #pragma omp parallel for parallelizing the linear search.
- The documents are located in a folder and file names are given as some collection
- The files are taken in a parallel way and all the data is collectively stored in a vector

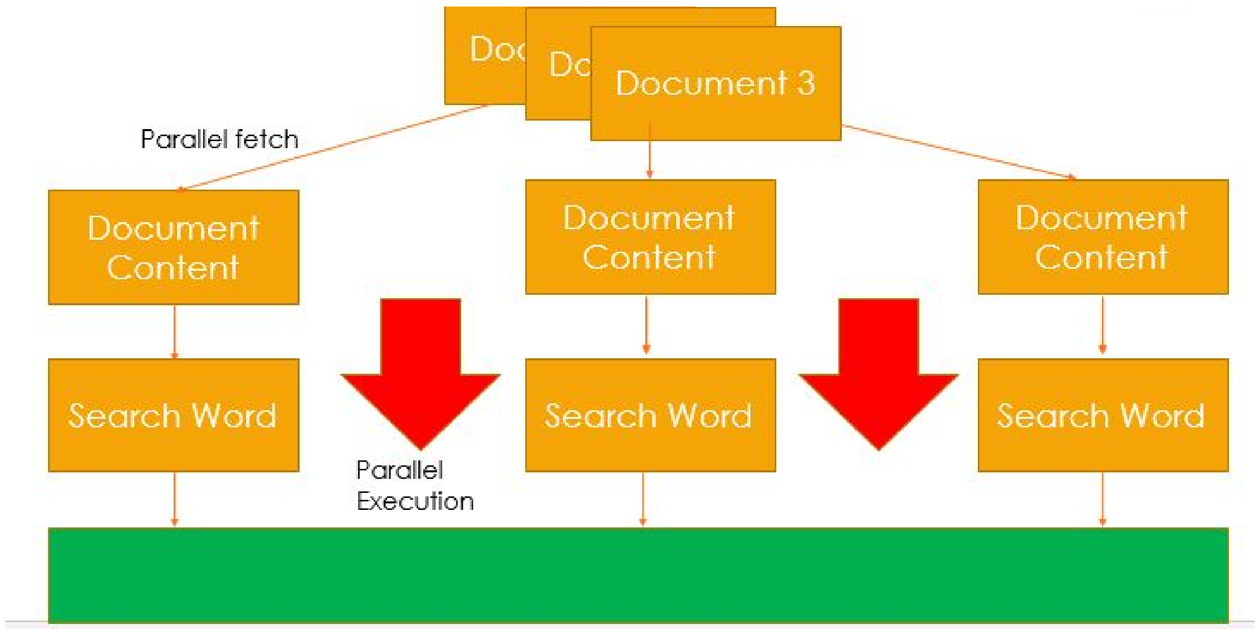
- After this the vectors are sent to a search method that parallelizes the search and concurrently browses through the working documents
- After this, the documents are ranked based on highest word searches.
- A mapping is done and the text files are displayed accordingly based on those with highest occurrence.
- Parallelizing the sequential search will expedite the word search process as there are thousands of words in each file.

Block Diagram

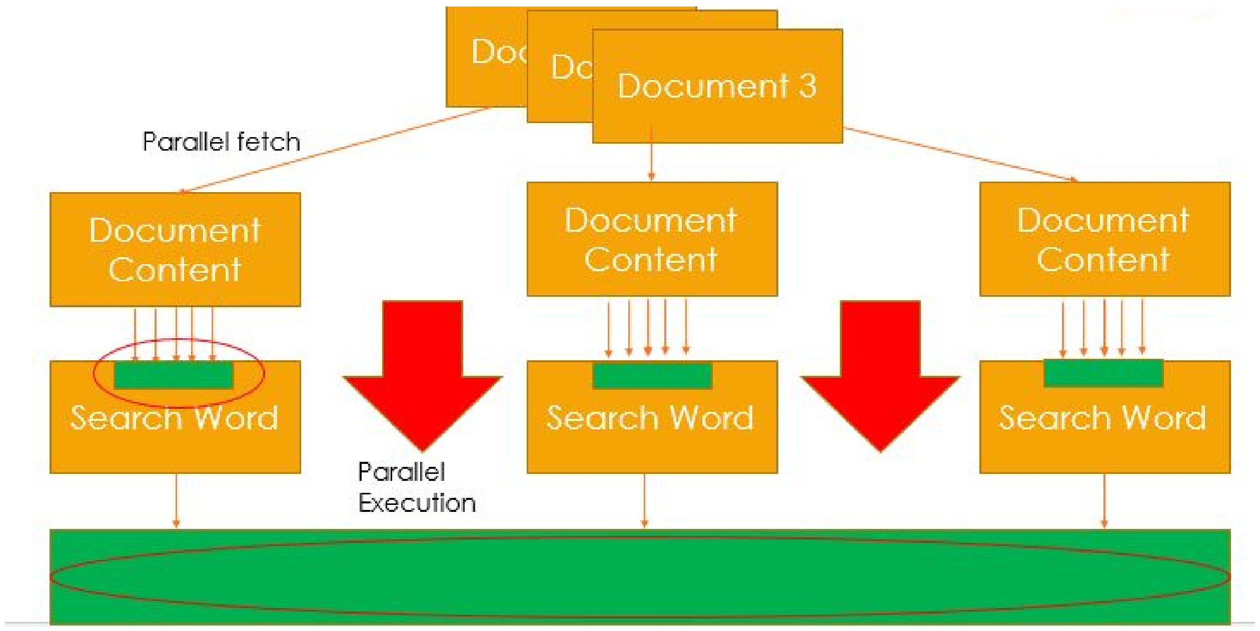
Serial Execution



Parallel Execution 1



Parallel Execution 2



NOTE

In Parallel Execution 1, each line in a file is fetched serially and in parallel execution 2, Each line in a file is fetched in parallel.

Implementation

Serial Code Snippets

Search word function

```
int searchWord (string s, int i)
{

   int count1=0,offset;
   string line;
   string fileName;
   fileName = "file"+to_string(i)+".txt";

   ifstream Myfile;
   Myfile.open(fileName);

   if(Myfile.is_open())
   {
      while (!Myfile.eof())
      {
         getline(Myfile,line);
         transform(line.begin(),line.end(),line.begin(),::tolower);
         if ((offset = line.find(s,0)) != string::npos){
            count1 = count1 + countFreq(s,line);
         }
         //count2++;
      }

      return count1;
   }

   else{
      return -1;
   }
}
```

Main()

```
int main()
{
   string str;
   vector<string> s;
   vector<vector<int>> table;
   vector<int> count1;
   cout<<"Enter a query : ";
   getline(cin, str);

   s = removeDupWord(str);

   for (auto i = s.begin(); i != s.end(); ++i)
      cout << *i << " ";
   cout<<endl;





   auto start1 = chrono::steady_clock::now();





   int n=100;

   for(int j=0;j<=n;j++){

   count1 = {};

   for(int i = 0; i < s.size(); i++)
   {
      count1.push_back(searchWord(s[i],j));
   }
   table.push_back(count1);

   }







  for (int i = 0; i < table.size(); i++)
```

```
    {
        cout<<"file "<<i<<" : ";
        for (int j = 0; j < table[i].size(); j++)
        {
            cout << s[j]<<" ("<<table[i][j] << ") ";
        }
        cout << endl;
    }
    auto end1 = chrono::steady_clock::now();

    double elapsed_time_ns = double (std::chrono::duration_cast <std::chrono::nanoseconds>
(end1-start1).count());

    cout<<" \n Elapsed Time (s) :"<<elapsed_time_ns / 1e9;
}
```

Parallel-1 Code Snippets

Search word function

```
int searchWord (string s, int i)
{
   int count1=0,offset;
   string line;
   string fileName;
   fileName = "file"+to_string(i)+".txt";

   ifstream Myfile;
   Myfile.open(fileName);

   if(Myfile.is_open())
   {
      while (!Myfile.eof())
      {
         getline(Myfile,line);
         transform(line.begin(),line.end(),line.begin(),::tolower);
         if ((offset = line.find(s,0)) != string::npos){
            count1 = count1 + countFreq(s,line);
         }
         //count2++;
      }

      return count1;
   }

   else{
      return -1;
   }
}
```

Main()

```
int main()
{



    string str;
    vector<string> s;
    vector<vector<int>> table = {{0…..

vector<int> count1;
    cout<<"Enter a query : ";
    getline(cin, str);
    s = removeDupWord(str);
    for (auto i = s.begin(); i != s.end(); ++i)
        cout << *i << " ";

    cout<<endl;

    auto start1 = chrono::steady_clock::now();

    int arr[10];
    int n=100;








    #pragma omp parallel num_threads(n)
    {
```

```
    int tid = omp_get_thread_num();


    for(int i = 0; i < s.size(); i++)
    {
        table [tid][i] = searchWord(s[i],tid);
    }


    }


   for (int i = 0; i < n; i++)
    {
        cout<<"file "<<i<<" : ";
        for (int j = 0; j < s.size(); j++)
        {
            cout << s[j]<<" ("<<table[i][j] << ") ";
        }
        cout << endl;
    }
    auto end1 = chrono::steady_clock::now();


    double elapsed_time_ns = double (std::chrono::duration_cast <std::chrono::nanoseconds>
(end1-start1).count());


    cout<<" \n Elapsed Time (s) :"<<elapsed_time_ns / 1e9;
}
```

Parallel-2 Code Snippets

Search word function

```
int searchWord (string s, int i)
{
   int count1=0,offset,countline=0,j=0;
   int arr[100];
   string line[100];
   string fileName;
   fileName = "file"+to_string(i)+".txt";

   ifstream Myfile;
   Myfile.open(fileName);

   if(Myfile.is_open())
   {
      while (!Myfile.eof())
      {
         getline(Myfile,line[j]);
         transform(line[j].begin(),line[j].end(),line[j].begin(),::tolower);
         countline++;
         j++;
      }


      #pragma omp parallel for
      for(int k=0; k<countline; k++)
      {



         arr[k]=0;
         if ((offset = line[k].find(s,0)) != string::npos){
            arr[k] = countFreq(s, line[k]);

         }

      }

      int sum=0;

      for(int k = 0; k<countline ; k++){
      sum+=arr[k];
      }




      return sum;
   }
```

```
    else{
        return -1;
    }
}
```

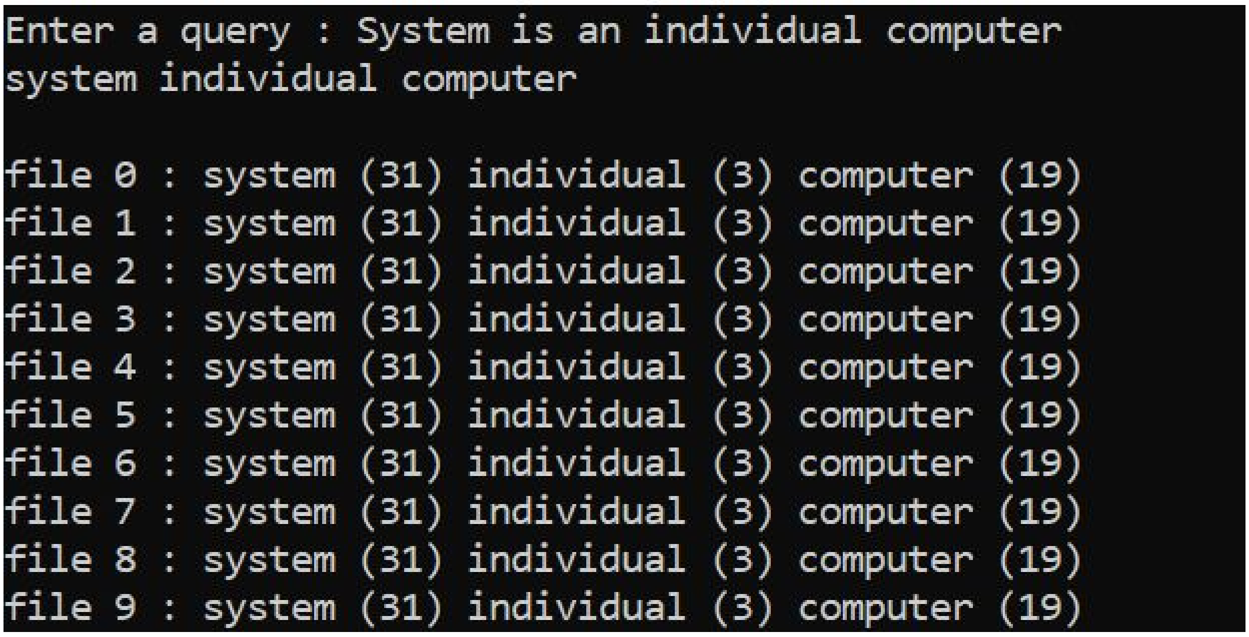
Main()

```
int main()

{

    string str;

    vector<string> s;

    vector<vector<int>> table = {{0,0,0,0,0,0………

    cout<<"Enter a query : ";

    getline(cin, str);

    s = removeDupWord(str);

    for (auto i = s.begin(); i != s.end(); ++i)

        cout << *i << " ";


    cout<<endl;



    auto start1 = chrono::steady_clock::now();




    int arr[10];

    int n=100;


    #pragma omp parallel num_threads(n)

    {


    int tid = omp_get_thread_num();
```

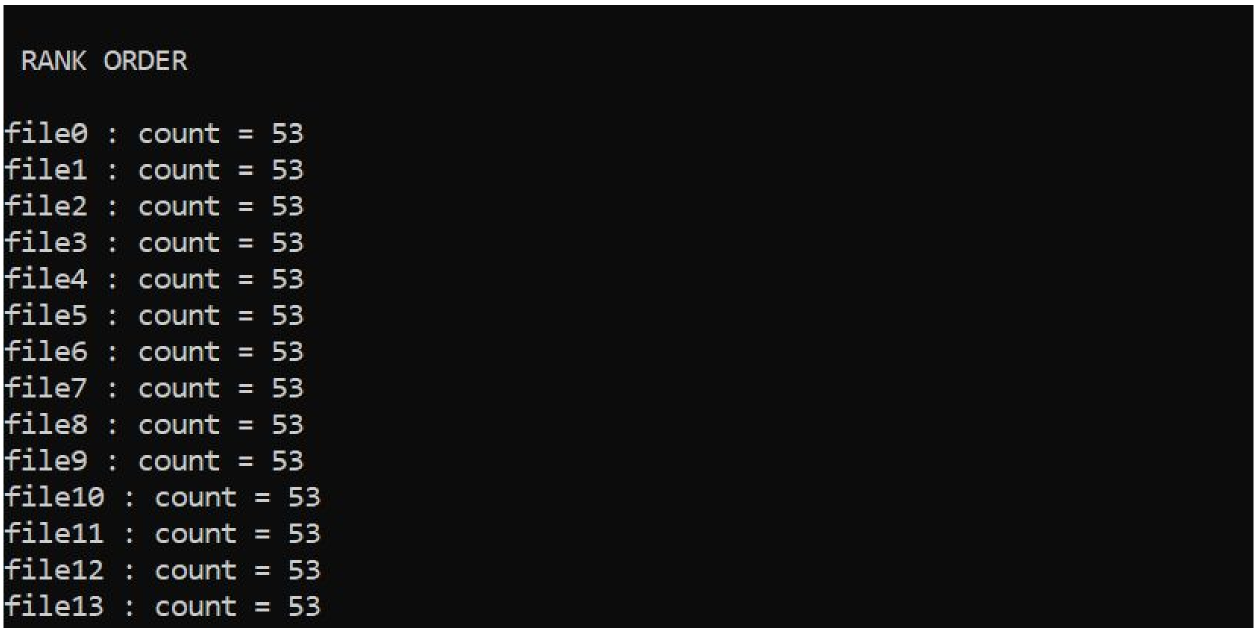
```
   for(int i = 0; i < s.size(); i++)
   {
      table [tid][i] = searchWord(s[i],tid);
   }
   }


  for (int i = 0; i < n; i++)
   {
      cout<<"file "<<i<<" : ";
      for (int j = 0; j < s.size(); j++)
      {
         cout << s[j]<<" ("<<table[i][j] << ") ";
      }
      cout << endl;
   }
   auto end1 = chrono::steady_clock::now();


   double elapsed_time_ns = double (std::chrono::duration_cast <std::chrono::nanoseconds>
(end1-start1).count());


   cout<<" \n Elapsed Time (s) :"<<elapsed_time_ns / 1e9;
```

OUTPUT

Serial Code

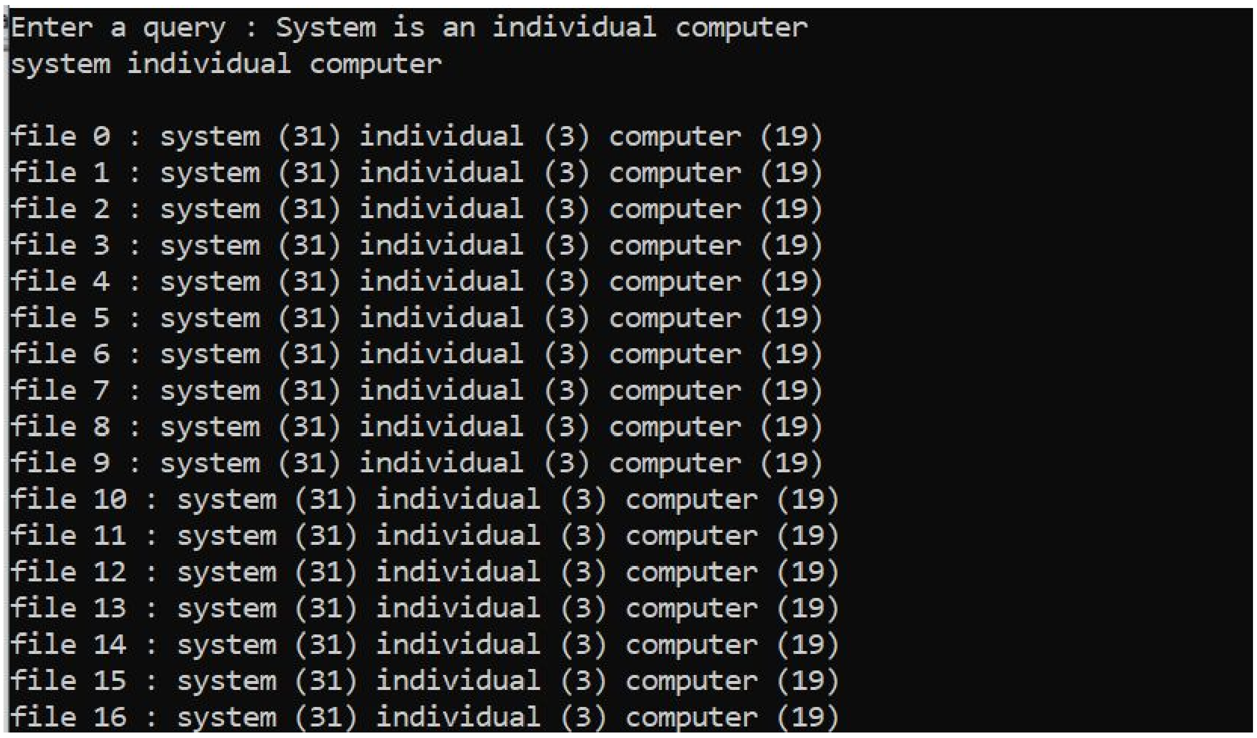


```
Enter a query : System is an individual computer
system individual computer

file 0 : system (31) individual (3) computer (19)
file 1 : system (31) individual (3) computer (19)
file 2 : system (31) individual (3) computer (19)
file 3 : system (31) individual (3) computer (19)
file 4 : system (31) individual (3) computer (19)
file 5 : system (31) individual (3) computer (19)
file 6 : system (31) individual (3) computer (19)
file 7 : system (31) individual (3) computer (19)
file 8 : system (31) individual (3) computer (19)
file 9 : system (31) individual (3) computer (19)
```

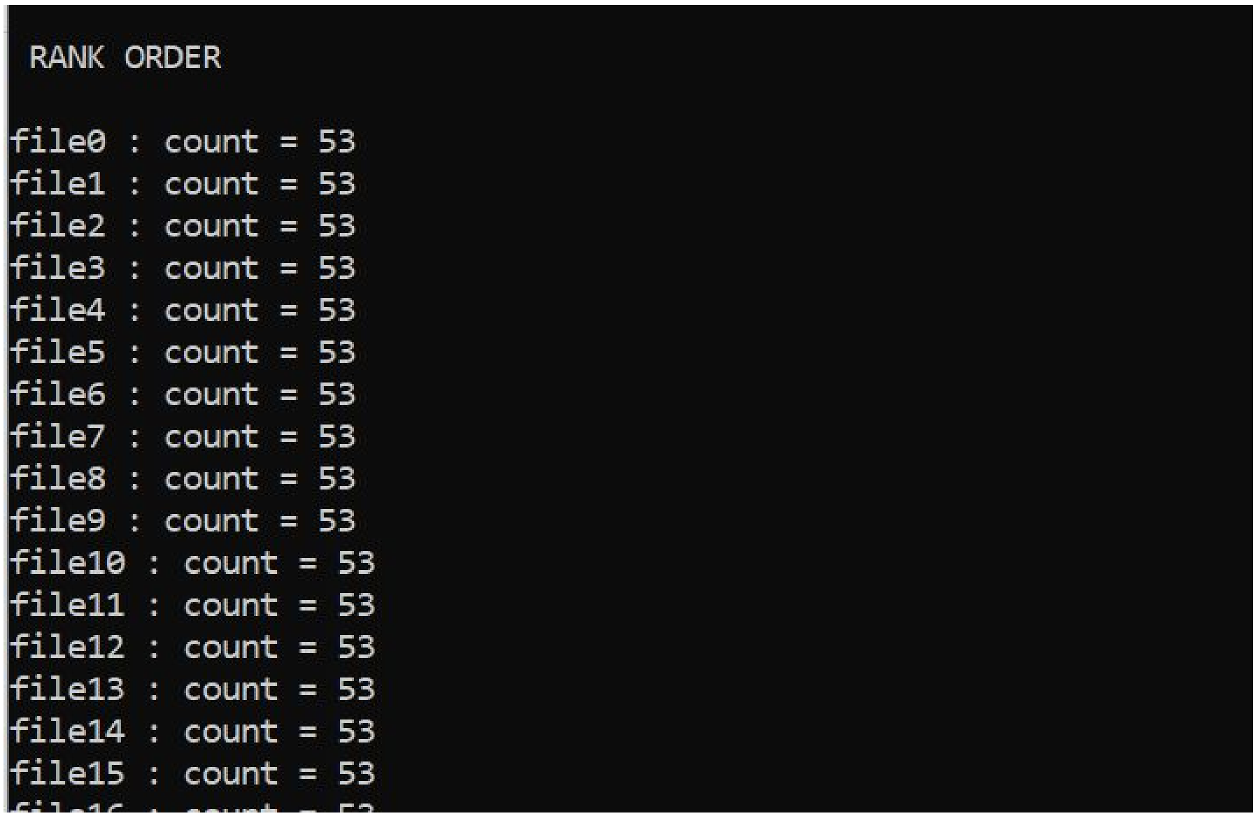


```
 RANK ORDER

file0 : count = 53
file1 : count = 53
file2 : count = 53
file3 : count = 53
file4 : count = 53
file5 : count = 53
file6 : count = 53
file7 : count = 53
file8 : count = 53
file9 : count = 53
file10 : count = 53
file11 : count = 53
file12 : count = 53
file13 : count = 53
```

Parallel – 1

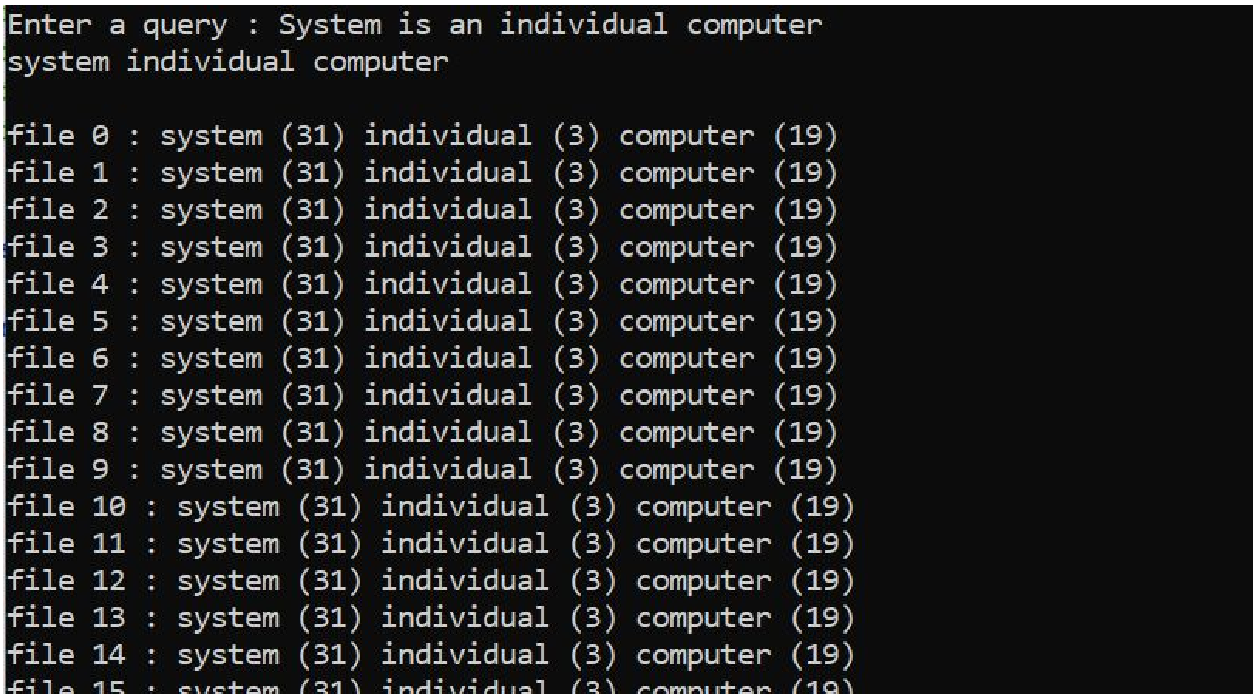


```
Enter a query : System is an individual computer
system individual computer

file 0 : system (31) individual (3) computer (19)
file 1 : system (31) individual (3) computer (19)
file 2 : system (31) individual (3) computer (19)
file 3 : system (31) individual (3) computer (19)
file 4 : system (31) individual (3) computer (19)
file 5 : system (31) individual (3) computer (19)
file 6 : system (31) individual (3) computer (19)
file 7 : system (31) individual (3) computer (19)
file 8 : system (31) individual (3) computer (19)
file 9 : system (31) individual (3) computer (19)
file 10 : system (31) individual (3) computer (19)
file 11 : system (31) individual (3) computer (19)
file 12 : system (31) individual (3) computer (19)
file 13 : system (31) individual (3) computer (19)
file 14 : system (31) individual (3) computer (19)
file 15 : system (31) individual (3) computer (19)
file 16 : system (31) individual (3) computer (19)
```

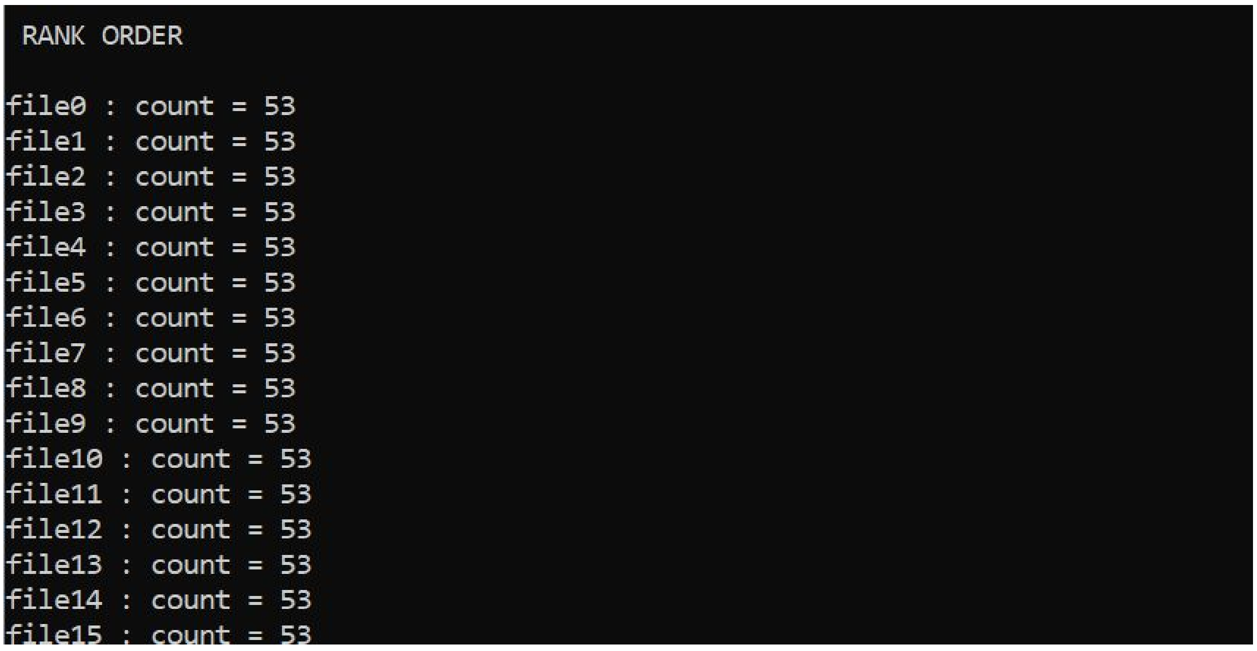


```
 RANK ORDER

file0 : count = 53
file1 : count = 53
file2 : count = 53
file3 : count = 53
file4 : count = 53
file5 : count = 53
file6 : count = 53
file7 : count = 53
file8 : count = 53
file9 : count = 53
file10 : count = 53
file11 : count = 53
file12 : count = 53
file13 : count = 53
file14 : count = 53
file15 : count = 53
```

Parallel - 2



```
Enter a query : System is an individual computer
system individual computer

file 0 : system (31) individual (3) computer (19)
file 1 : system (31) individual (3) computer (19)
file 2 : system (31) individual (3) computer (19)
file 3 : system (31) individual (3) computer (19)
file 4 : system (31) individual (3) computer (19)
file 5 : system (31) individual (3) computer (19)
file 6 : system (31) individual (3) computer (19)
file 7 : system (31) individual (3) computer (19)
file 8 : system (31) individual (3) computer (19)
file 9 : system (31) individual (3) computer (19)
file 10 : system (31) individual (3) computer (19)
file 11 : system (31) individual (3) computer (19)
file 12 : system (31) individual (3) computer (19)
file 13 : system (31) individual (3) computer (19)
file 14 : system (31) individual (3) computer (19)
file 15 : system (31) individual (3) computer (19)
```



```
 RANK ORDER

file0 : count = 53
file1 : count = 53
file2 : count = 53
file3 : count = 53
file4 : count = 53
file5 : count = 53
file6 : count = 53
file7 : count = 53
file8 : count = 53
file9 : count = 53
file10 : count = 53
file11 : count = 53
file12 : count = 53
file13 : count = 53
file14 : count = 53
file15 : count = 53
```

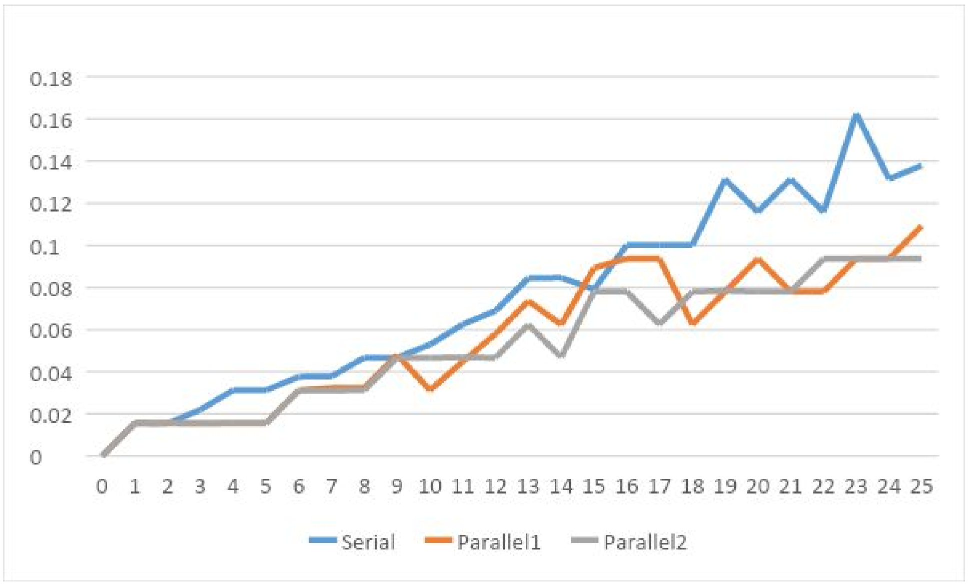
Result

So by running a standard experiment of running the code with 100 files with the same text, and searching the same query, times were collected in all cases and plotted accordingly to see the difference between serial and parallel implementation.

0 – 25 Files

	Serial	Parallel1	Parallel2
0	0	0	0
1	0.015648	0.015437	0.015461
2	0.015463	0.015507	0.015611
3	0.022003	0.015475	0.015476
4	0.03127	0.015469	0.015628
5	0.031239	0.015466	0.015623
6	0.037608	0.031257	0.031132
7	0.037808	0.032337	0.031091
8	0.046716	0.032498	0.031269
9	0.046725	0.047973	0.046713
10	0.052902	0.031249	0.046714
11	0.062515	0.04485	0.046848
12	0.068899	0.05797	0.046724
13	0.084493	0.073591	0.062347
14	0.084661	0.062384	0.046877
15	0.079146	0.089282	0.07811
16	0.100111	0.09375	0.078004
17	0.100116	0.093593	0.062487

18	0.100127	0.062493	0.077962
19	0.131363	0.077968	0.07843
20	0.115852	0.093598	0.077966
21	0.131371	0.07811	0.077969
22	0.115914	0.077963	0.093592
23	0.162563	0.093725	0.093589
24	0.13154	0.093591	0.093589
25	0.137932	0.109238	0.093596



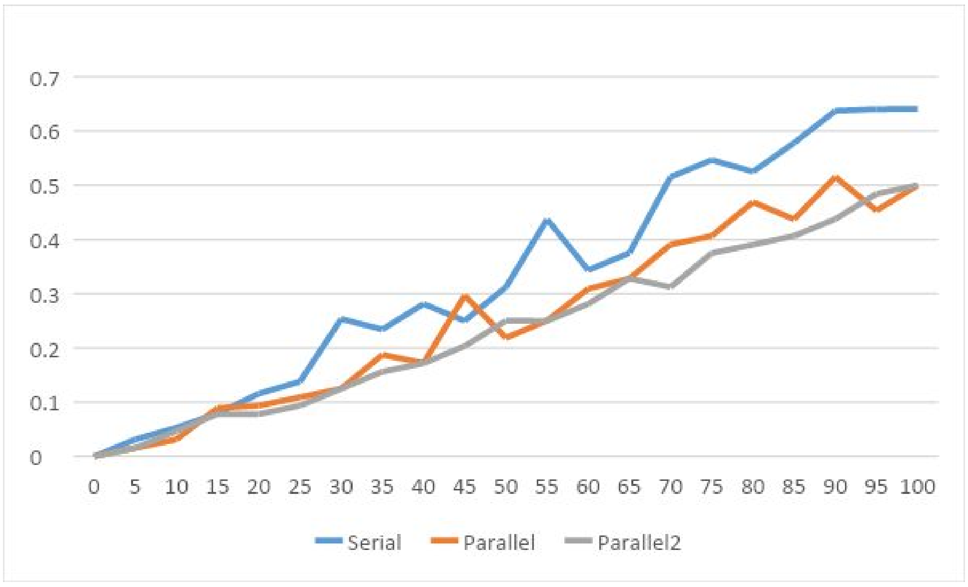
SPEED UP 1 – 1.366340

SPEED UP 2 – 1.413178

0 – 100 Files

	Serial	Parallel	Parallel2
0	0	0	0
5	0.031239	0.015466	0.015623

10	0.052902	0.031249	0.046714
15	0.079146	0.089282	0.07811
20	0.115852	0.093598	0.077966
25	0.137932	0.109238	0.093596
30	0.253474	0.124845	0.125
35	0.234224	0.187376	0.156101
40	0.281111	0.172349	0.171698
45	0.249998	0.296801	0.203503
50	0.312345	0.218601	0.25008
55	0.437329	0.250461	0.249841
60	0.343594	0.309215	0.281156
65	0.375066	0.328084	0.328247
70	0.515462	0.390754	0.312348
75	0.546866	0.406717	0.374994
80	0.525264	0.46893	0.390475
85	0.578277	0.4375	0.406837
90	0.637396	0.515467	0.437815
95	0.640466	0.453582	0.484222
100	0.640655	0.499843	0.5



SPEED UP 1 – 1.365830

SPEED UP 2 – 1.43990

Observation

From the above results we can see that serial implementation takes more time than parallel implementation. However the difference in time increases only when there is an increase in the number of files or an increase in workload. Hence for tasks with a lot of workload, it is essential to use parallelism to save time.

Conclusion

Nowadays, all computers use parallel programming in order to accomplish tasks. The main reason for parallel programming is to execute code efficiently, since parallel programming saves time, allowing the execution of applications in a shorter wall-clock time. As a consequence of executing code efficiently, parallel programming often scales with the problem size, and thus can solve larger problems. In general, parallel programming is a means of providing concurrency, particularly performing simultaneously multiple actions at the same time. Parallel programming goes beyond the limits imposed by sequential computing, which is often constrained by physical and practical factors that limit the ability to construct faster sequential computers.

This program is an introduction to the applications of parallel computing and shows its advantages of serial implementation.

REFERENCES

[1] P. Heidelberger, A. Norton, and J. T. Robinson. Parallel quicksort using Fetch-and-Add. IEEE Transactions

on Computers, 39(1):133–137.

[2] M. Edahiro. Parallelizing fundamental algorithms such as sorting on multi-core processors for EDA

acceleration. In Proceedings of the 2009 Asia and South Pacific Design Automation Conference, pages

230–233, Yokohama, Japan, 2009.

[3] J. Singler, P. Sanders, and F. Putze. MCSTL: The Multi-core Standard Template Library. Lecture Notes in

Computer Science, 4641:682–694, 2007.

[4] Introduction to algorithms by CLRS.

[5] Puneet C Kataria, Parallel quicksort implementation using MPI and Pthreads.

[6] Hanmao Shi Jonathan Schaeffer, Parallel Sorting by Regular Sampling.