

**PARALLEL AND DISTRIBUTED
COMPUTING (CSE4001) J COMPONENT**

**REVIEW 3
SUBMITTED TO: NARAYANMOORTHY M**



TITLE: PARALLEL WORD SEARCH USING OPENMP

GROUP MEMBERS:

| NAME | REGISTRATION NUMBER |
|-----------------|---------------------|
| Abhishek Kandel | 19BCE2629 |
| Suman Tandan | 19BCE2651 |
| Atul Koirala | 19BCE2666 |

ABSTRACT

Word search is used everywhere from local page search (Cntrl + F) to searching words on document viewer like “reader” in windows. In fact a whole branch called Information Retrieval was developed for this. This project was actually inspired by Information Retrieval. It has a lot of application in real word.

As the name suggests “word search” is about searching words in documents parallely using openmp. This is not just a simple word search but it also ranks the documents based on the relevance of the documents with respect to the searched word. The documents are scanned word by word and a dictionary is maintained so that words can be searched. Since documents can have several thousands of terms scanning each term can take a lot of time and hence there is a lot of scope for parallelization.

This project focuses on the principle of multithreaded systems. It uses multiple threads to read multiple files and perform the action. The action here being searching the word through the files, and doing so in very less time as compared to the sequential system. The parameters are similar to that of the sequential method, but the processors are used to do the sequential process on multiple files at the same time. Here we explain how the sequential and multithreaded search works.

Sequential method: We’ve spoken about updates like Panda and Hummingbird and highlighted how important semantics in search is, in modern SEO strategies. We expanded on how Search Engines are looking past exact keyword matching on pages to providing more value to end users through more conceptual and contextual results in their service. While the focus has moved away from exact keyword matching, keywords are still a pivotal part of SEO and content strategies, but the concept of a keyword has changed somewhat. Search strings are more conversational now, they are of the long-tail variety and are often, context rich. Traditionally, keyword research involved building a list or database of relevant keywords that we hoped to rank for. Often graded by difficulty score, click through rate and search volume, keyword research was about finding candidates in this list to go create content around and gather some organic traffic through exact matching. We are using simple method of sequential method to search the file of the given file. The Method is quite simple of input the file, read the file, input the word we are looking for, and search the file, and give output that it is found or not. Multithreaded method: The method here is using multithreaded library and declare multiple threads to handle each file. This system has perks as well as cons. Multithreading support was introduced in C++11. Prior to C++11, we had to use

POSIX threads or p threads library in C. While this library did the job the lack of any standard language provided feature-set caused serious portability issues. C++ 11 did away with all that and gave us **std::thread**. The thread classes and related functions are defined in the **thread** header file. **Std::thread** is the thread class that represents a single thread in C++. To start a thread we simply need to create a new thread object and pass the executing code to be called (i.e. a callable object) into the constructor of the object. Once the object is created a new thread is launched which will execute the code specified in callable.

INTRODUCTION

Word search searches for words in multiple text files parallelly using openmp concepts. This reduces the execution time of the code as compared to the sequential word search. The documents are scanned word by word and a dictionary is maintained so that words can be searched. Since documents can have several thousands of terms scanning each term can take a lot of time and hence there is a lot of scope for parallelization.

KEYWORDS: OpenMP, Parallelization, Multithreaded, Wordsearch, Sequential.

PURPOSE

Word search searches for words in multiple text files parallelly using openmp concepts. This reduces the execution time of the code as compared to the sequential word search. The documents are scanned word by word and a dictionary is maintained so that words can be searched. Since documents can have several thousands of terms scanning each term can take a lot of time and hence there is a lot of scope for parallelization. The purpose of this project to analysis which word search method is efficient to handle the multiple files. The conclusion of this project will give us the sufficient reasons to choose the method. This could be helpful for future word searing engines.

SCOPE

The scope of this project is that the word search engines in the world are efficient enough for the current level of processing speed. But, as the technology evolves, the speed has to be greater than the present. The present method is sequential method. The metrics of sequential method are much inferior compared to that of the multithreaded search. The scope as of now is the files saved in the memory. Later developments can be made to make it suitable .

PREVIOUS WORKS

Word Searches and Computational Thinking

Solve word search puzzles and learn about computational thinking and search algorithms. Puzzles are a good way of developing computational thinking. Word searches involve pattern matching. To solve them quickly an algorithm helps – essentially an adapted linear search for individual letters by scanning the grid, followed by searching round that point for the second letter, then continuing the search in that direction. Students can be set word searches and encouraged to reflect on and write down the way they are trying to solve them. They can then try to find improvements and heuristics that help solve them more quickly, encouraging the development of algorithmic thinking skills. As an alternative it can be used as a practical use of linear search.

Program search jigsaws are a variation where you must search for fragments of program syntax, rather than words, then put them together to make a working program.

Keyword extraction is tasked with the automatic identification of terms that best describe the subject of a document.

Key phrases, key terms, key segments or just keywords are the terminology which is used for defining the terms that represent the most relevant information contained in the document. Although the terminology is different, function is the same: characterization of the topic discussed in a document. Keyword extraction task is important problem in Text Mining, Information Retrieval and Natural Language Processing.

EXISTING WORK AND TECHNOLOGIES USED

Word searches involve pattern matching. To solve them quickly an algorithm helps – essentially an adapted linear search for individual letters by scanning the grid, followed by searching round that point for the second letter, then continuing the search in that direction. There is a need to find improvements and heuristics that help solve them more quickly, encouraging the development of algorithmic thinking skills. As an alternative it can be used as a practical use of linear search. Program search jigsaws are a variation where you must search for fragments of program syntax, rather than words, then put them together to make a working program. Keyword extraction is tasked with the automatic identification of terms that best describe the subject of a document. Key phrases, key terms, key segments or just keywords are the terminology which is used for defining the terms that represent the most relevant information contained in the document. Although the terminology is different, function is the same: characterization of the topic discussed in a document. Keyword extraction task is important problem in Text Mining, Information Retrieval and Natural Language Processing.

The parallel part will be implemented using OpenMP. 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. An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems. OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

PROBLEM STATEMENT

Word search searches for words in multiple text files parallelly using openmp concepts. This reduces the execution time of the code as compared to the sequential word search. The documents are scanned word by word and a dictionary is maintained so that words can be searched. Since documents can have several thousands of terms scanning each term can take a lot of time and hence there is a lot of scope for parallelization. The purpose of this project to analysis which word search method is efficient to handle the multiple files. The conclusion of this project will give us the sufficient reasons to choose the method. This could be helpful for future word searing engines.

OBJECTIVE

The scope of this project is that the word search engines in the world are efficient enough for the current level of processing speed. But, as the technology evolves, the speed has to be greater than the present. The present method is sequential method. The metrics of sequential method are much inferior compared to that of the multithreaded search. The scope as of now is the files saved in the memory. Later developments can be made to make it suitable for a wide variety of platforms.

REQUIREMENTS ANALYSIS

Programming Languages and Web Technologies Used:

C++ (using Standard Template Library) : C++ is a general-purpose programming language. It has imperative, object-oriented and generic programming features, while also providing facilities for low-level memory manipulation.

It was designed with a bias toward system programming and embedded, resource-constrained and large systems, with performance, efficiency and flexibility of use as its design highlights. C++ has also been found useful in many other contexts, with key strengths being software infrastructure and resource-constrained applications,^[6] including desktop applications, servers (e.g. e-commerce, Web search or SQL servers), and performance-critical applications (e.g. telephone switches or space probes). C++ is a compiled language, with implementations of it available on many platforms. Many vendors provide C++ compilers, including the Free Software Foundation, Microsoft, Intel, and IBM.

C++ introduces object-oriented programming (OOP) features to C. It offers classes, which provide the four features commonly present in OOP (and some non-OOP) languages: abstraction, encapsulation, inheritance, and polymorphism. One distinguishing feature of C++ classes compared to classes in other programming languages is support for deterministic destructors, which in turn provide support for the Resource Acquisition is Initialization (RAII) concept.

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.

An application built with the hybrid model of parallel programming can run on a computer cluster using both OpenMP and Message Passing Interface (MPI), such that OpenMP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run OpenMP on software distributed shared memory systems, to translate OpenMP into MPI and to extend OpenMP for non-shared memory systems.

OpenMP is an implementation of multithreading, a method of parallelizing whereby a master thread (a series of instructions executed consecutively) forks a specified number of slave threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.

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

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

The runtime environment allocates threads to processors depending on usage, machine load and other factors. The runtime environment can assign the number of threads based on environment variables, or the code can do so using functions. The OpenMP functions are included in a header file labelled `omp.h` in C/C++

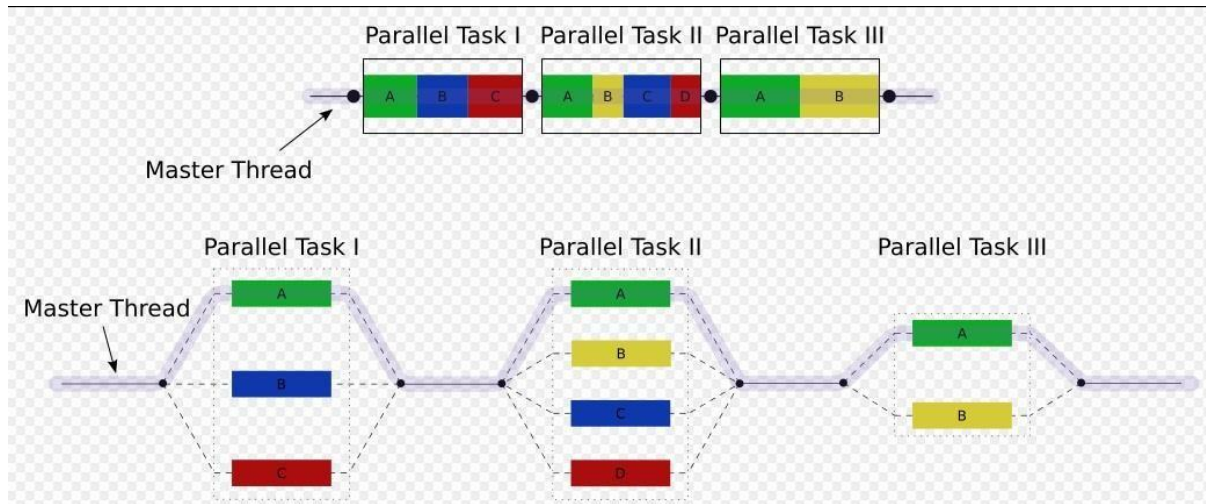


Fig 1:Fork Join model

DESIGN ARCHITECTURE

3.1 Sequential model:

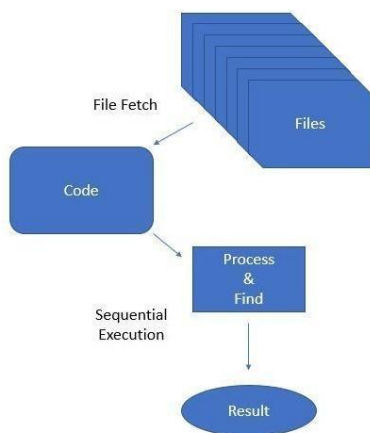


Fig 2:Sequential model

3.2 Parallel model:

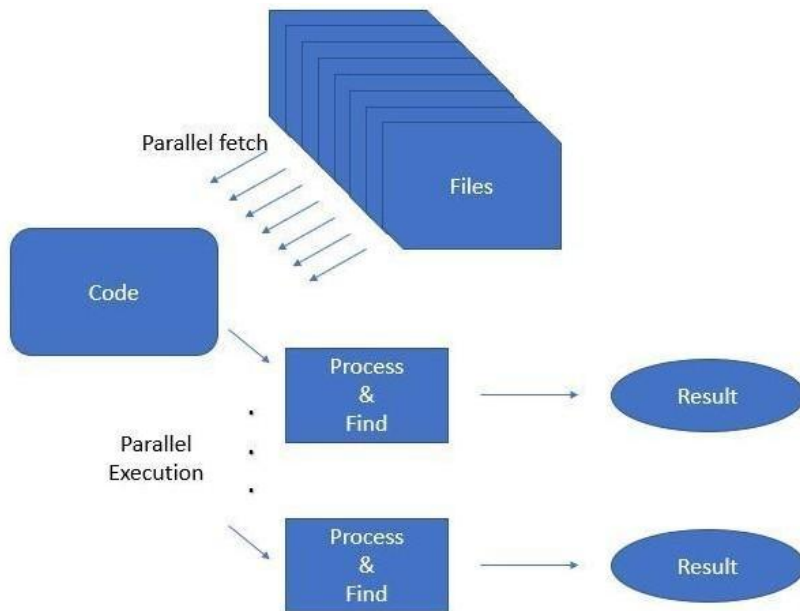


Fig 3:Parallel model

SYSTEM IMPLEMENTATION

The basic code is written in C++ which uses openmp to parallelize the code.

1. The documents are located in a folder and file names are given as some collection
2. The files are taken in a parallel way and all the data is collectively stored in a vector
3. After this the vectors are sent to a search method that parallizes the search and concurrently browses through the working documents
4. After this, the documents are ranked based on highest word searches.
5. A mapping is done and the text files are displayed accordingly based on those with highest occurrence.

1. Sequential Code:

```
#include <stdlib.h>
```

```
#include <omp.h>
```

```
#include <iostream>
```

```
#include <string>
```

```
#include <fstream>
```

```
using namespace std;
```

```
string substring(int start, int length, string strword) {
```

```
    string a = "";
```

```
    for (int i = start; i < length; i++) {
```

```
        a += strword[i];
    }
    return a;
}

int find_all(string sen, string word) {

    int wordLen =
word.length();    int start = 0;
    int endword = wordLen;
    int finLength = sen.length();
    int count = 0; string
senWord = "";

    for (int i = 0; i <= sen.length(); i++) {
        if (endword > finLength) {
            break;
        }
        senWord = substring(start, endword, sen);
        if ((senWord.compare(word)) == 0)
        {
            count++;
            if (endword > finLength) {
                break;
            }
        }
    }
}
```

```
        }

    }

    start += 1;

    endword += 1;

}

return count;

}

int display(string path, string word_to_search) {

    int  totalCount  =  0;
    string line;

    int count = 1;
    ifstream myfile(path);
    if (myfile.is_open())
    {
        while (getline(myfile, line)) {
            totalCount += find_all(line, word_to_search);
            count++;
        }
    }
    else {
        cout << "File not open\n" << endl;
```

```

    }

    return totalCount;
}

int main() {
    string Path = "C:\\xampp\\htdocs\\parallel";

    string word_to_search = "";  cout
<< "Enter a word to search: ";    cin
>> word_to_search;

    double time = omp_get_wtime();

    for (int i = 1; i <= 5; i++) {
        string npath = "File" + to_string(i) + ".txt";
        cout << "Total Count is " + to_string(display(npath, word_to_search)) <<
" from file " + to_string(i) << endl;
    }

    time = omp_get_wtime() - time;
    cout << "Time is " + to_string(time);

    return 0;
}

```

2. Parallel Code:

```
#include <stdlib.h>
#include <omp.h>
#include <iostream>
#include <string>
#include <fstream>
```

```
using namespace std;
```

```
string substring(int start, int length, string strword) {
```

```
    string a = "";
```

```
    for (int i = start; i < length; i++) {
```

```
        a += strword[i];
```

```
    }
```

```
    return a;
```

```
}
```

```
int find_all(string sen, string word) {
```

```
    int wordLen =
```

```
    word.length();    int start = 0;
```

```
int endword = wordLen;

int finLength = sen.length();

int count = 0; string
senWord = "";

for (int i = 0; i <= sen.length(); i++) {
    if (endword > finLength) {
        break;
    }
    senWord = substring(start, endword, sen);
    if ((senWord.compare(word)) == 0)
    {
        count++;
        if (endword > finLength) {
            break;
        }
    }
    start    +=    1;
    endword += 1;

}

return count;

}
```

```

int display(string path, string word_to_search) {

    int totalCount = 0;    string line;    int count = 1;

    ifstream myfile(path);    if (myfile.is_open()) {
        while (getline(myfile, line)) {
            totalCount += find_all(line, word_to_search);

            count++;
        }
    }
    else {
        cout << "File not open\n" << endl;
    }

    return totalCount;
}

int main() {

    string Path = "";

    string word_to_search = "";    cout
    << "Enter a word to search: ";    cin
    >> word_to_search;

    double time = omp_get_wtime();

    #pragma omp parallel for    for
    (int i = 1; i <= 5; i++) {
        string npath = "File" + to_string(i) + ".txt";

```



```
        // #pragma omp critical  
        cout << "Total Count is " + to_string(display(npath, word_to_search)) << " from file  
        "+to_string(i)<<endl;  
    }  
  
    time = omp_get_wtime() - time;  
    cout << "Time is " + to_string(time);  
  
    return 0;  
}
```

5.TESTING AND RESULT ANALYSIS

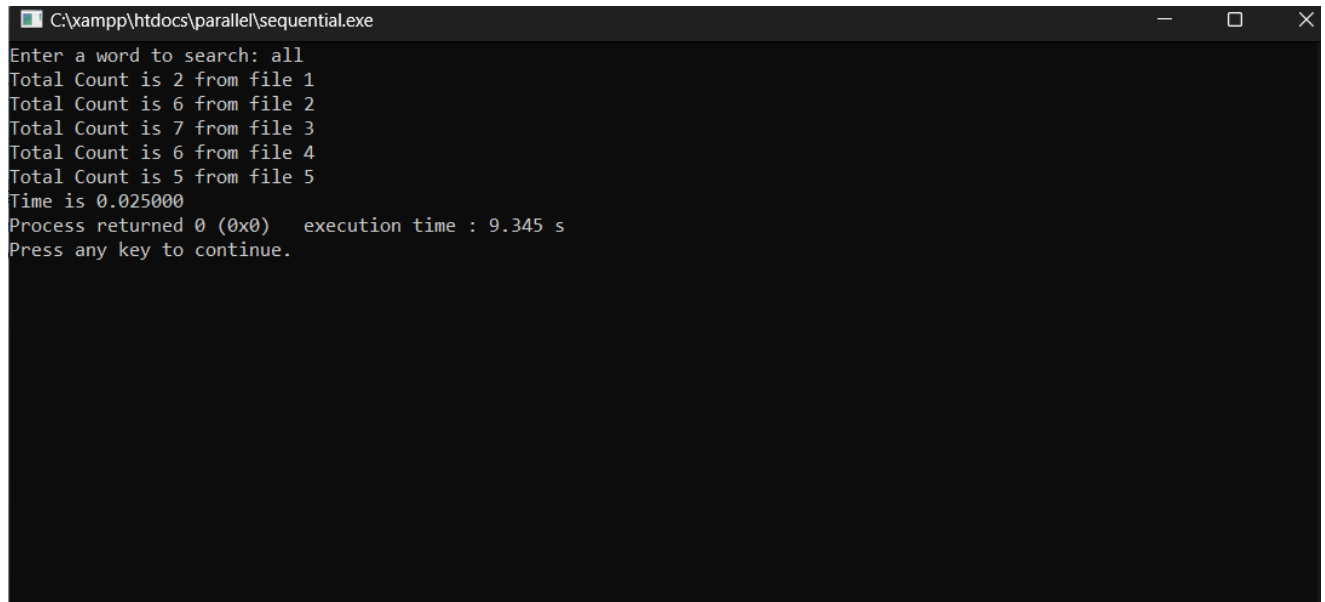
| Test case Number | Test case Id | Test case | Expected Result | Actual Result | Pass / Fail |
|------------------|--------------|---|-------------------------------------|-------------------------------------|-------------|
| 1. | 1.1 | Sequential code: Reading 5 files to search for a word | The count of each word in each file | The count of each word in each file | PASS |
| | 1.2 | Sequential code: Reading 5 files to search for a statement | The count of statement in each file | The count of statement in each file | PASS |
| 2. | 2.1 | Parallel code: Reading 5 files concurrently to search for a word | The count of each word in each line | The count of each word in each line | PASS |
| | 2.2 | Parallel code: Reading 5 files to search for a statement | The count of statement in each file | The count of statement in each file | PASS |

RESULTS (OUTPUT SCREENSHOTS)

We have taken a sample of 65 files each containing around 1500 words and compared the search time of both parallel and sequential codes. The code can search for statements as well as words. If the user gives a statement as input our code will break the statement into an array of words and search for each word in the files sequentially. For this we have created a method called substring which takes the starting index, ending index and the string as input parameters. At last it finds the count of all the words in the files and sums it all. The output will be count of occurrences of statement in the files.

Output Screenshots

Sequential code



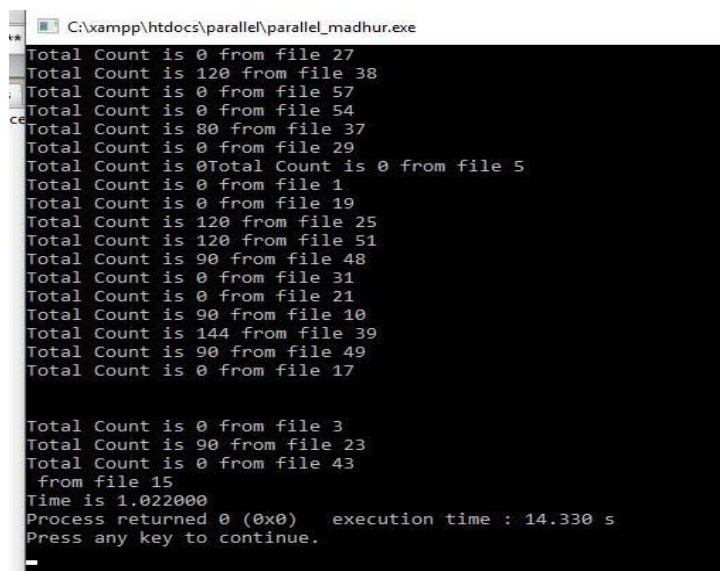
```

C:\xampp\htdocs\parallel\sequential.exe
Enter a word to search: all
Total Count is 2 from file 1
Total Count is 6 from file 2
Total Count is 7 from file 3
Total Count is 6 from file 4
Total Count is 5 from file 5
Time is 0.025000
Process returned 0 (0x0)   execution time : 9.345 s
Press any key to continue.

```

Sequential time = 0.025000 seconds

Parallel code



```

C:\xampp\htdocs\parallel\parallel_madhur.exe
Total Count is 0 from file 27
Total Count is 120 from file 38
Total Count is 0 from file 57
Total Count is 0 from file 54
Total Count is 80 from file 37
Total Count is 0 from file 29
Total Count is 0Total Count is 0 from file 5
Total Count is 0 from file 1
Total Count is 0 from file 19
Total Count is 120 from file 25
Total Count is 120 from file 51
Total Count is 90 from file 48
Total Count is 0 from file 31
Total Count is 0 from file 21
Total Count is 90 from file 10
Total Count is 144 from file 39
Total Count is 90 from file 49
Total Count is 0 from file 17

Total Count is 0 from file 3
Total Count is 90 from file 23
Total Count is 0 from file 43
from file 15
Time is 1.022000
Process returned 0 (0x0)   execution time : 14.330 s
Press any key to continue.

```

Parallel search time(when number of threads is equal to the number of files to be searched i.e 65)=1.022 seconds

```

C:\xampp\htdocs\parallel\parallel_madhur.exe
Total Count is 0 from file 54
Total Count is 0 from file 33
Total Count is 120 from file 25
Total Count is 0 from file 59
Total Count is 144Total Count is 80 from file 63
Total Count is 0 from file 47
Total Count is 144 from file 13
Total Count is 0 from file 17
from file 18
Total Count is 0 from file 45
Total Count is 0 from file 55
Total Count is 0 from file 53
Total Count is 0 from file 2
from file 39
Total Count is 90 from file 48
Total Count is 0 from file 31
from file 41
Total Count is 0 from file 19
Total Count is 0 from file 7
Total Count is 0 from file 56
Total Count is 0Total Count is 80 from file 11
from file 27
Total Count is 0 from file 3
Total Count is 90 from file 49
Total Count is 0 from file 57
Time is 0.533000
Process returned 0 (0x0)   execution time : 9.095 s
Press any key to continue.

```

Parallel search time(when number of threads is equal to half the number of files to be searched I.e 32)=0.533 seconds

```

C:\xampp\htdocs\parallel\parallel_madhur.exe
Total Count is 80 from file 63
Total Count is 0 from file 14
from file 9
Total Count is 0 from file 18
Total Count is 0 from file 56
Total Count is 90Total Count is 0 from file 41
Total Count is 0Total Count is 0 from file 19
Total Count is 0 from file 44Total Count is 90 from file 10
Total Count is 0 from file 33 from file 49Total Count is 0 from file 57
from file 15
Total Count is 120 from file 64
Total Count is 0 from file 3
Total Count is 144Total Count is 80 from file 24
from file 52
Total Count is 0 from file 45
Total Count is 0 from file 20
Total Count is 0 from file 4
Total Count is 0 from file 53
Total Count is 144 from file 65
Total Count is 120 from file 25
Total Count is 0 from file 5
Time is 0.926000
Process returned 0 (0x0)   execution time : 7.982 s
Press any key to continue.

```

Parallel search time(when number of threads is equal to one fourth the number of files to be searched I.e 15)=0.926000 seconds

```

C:\xampp\htdocs\parallel\parallel_madhur.exe
Total Count is 0 from file 55
Total Count is 0 from file 44
Total Count is 0 from file 42
  from file 36
Total Count is 90 from file 10
Total Count is 0 from file 45
  from file 32
Total Count is 120 from file 51
Total Count is 120 from file 64
Total Count is 0 from file 54
Total Count is 0 from file 43
Total Count is 0 from file 29
Total Count is 0 from file 59
Total Count is 120 from file 38
Total Count is 90 from file 61Total Count is 80 from file 63Total Count is 120 from file 25
Total Count is 0 from file 60
Total Count is 90 from file 48
Total Count is 0 from file 19
Total Count is 90 from file 22
Total Count is 0 from file 1
Time is 4.699000
Process returned 0 (0x0)   execution time : 14.590 s
Press any key to continue.

```

Parallel search time(when number of threads is far greater than number of files to be searched I.e 10000)=4.699000 seconds

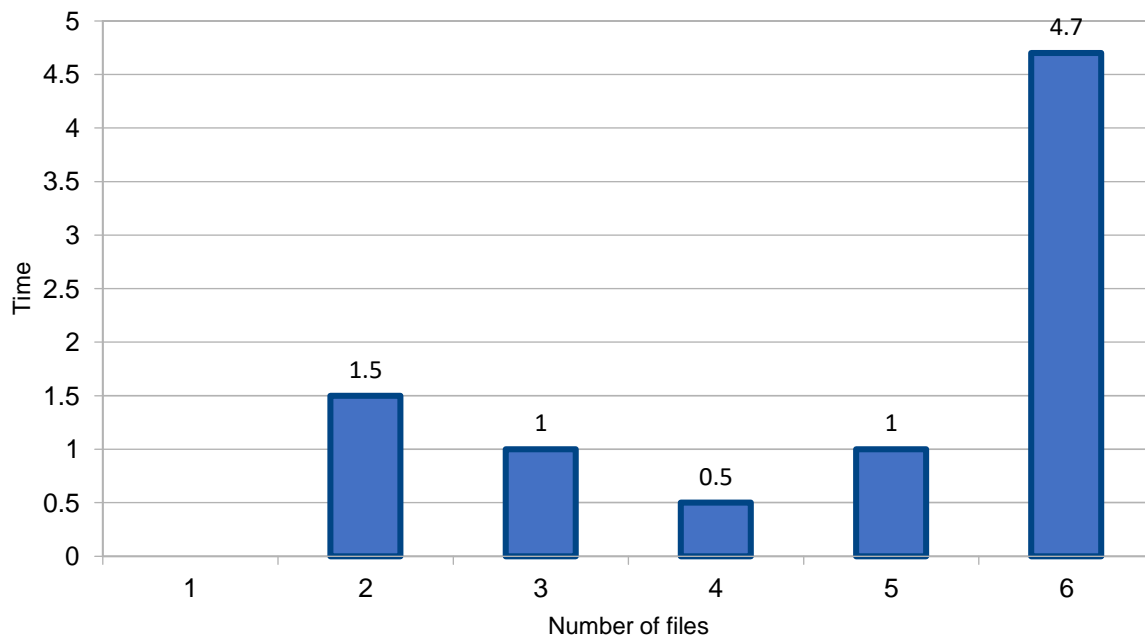
EXPLANATION FOR THE ABOVE RESULTS:

As we know open mp follows for join concept. Thus additional overheads incur when we create threads for a program. Thus the number of threads chosen have to such that the parallel code gives the best possible result. When we used number of threads to be 65 we were getting access time which was less than the sequential time but due to the additional overhead of fork and join of threads the result was not the most optimum. When we took the number of threads to be too small, again we were getting getting the search time to be better than the sequential but not most optimum. When we took the number of threads to be 10000, the search time came out to be 4 seconds which was far worse then the sequential search time. Thus after detailed analysis of the code and taking different number of files and trying out with different number of threads we came to conclusion that for the search time to be least, the number of threads should be around half of the number of files used for less than 100 files to be read.

Result is depicted in the form of following graph for easy visualization.

X-Axis shows the number of threads.

Y-Axis shows the number search time in seconds.



| Word to be searched. | Time taken in sequential code | Time taken in parallel code |
|----------------------|-------------------------------|-----------------------------|
| boat | 1.790317 | 1.594218 |
| book | 1.839465 | 1.491440 |
| milk | 1.791273 | 1.474439 |
| zone | 1.862333 | 1.413917 |
| xerox | 1.877263 | 1.524952 |

CONCLUSION AND FUTURE WORKS

The Code was executed successfully and applying parallelism reduces running time significantly. Word search is used everywhere from local page search (Cntrl + F) to searching words on document viewer like “reader” in windows. Infact a whole branch called Information Retrieval

was developed for this. This project was actually inspired by Information Retrieval. It has a lot of application in real word.

In the following project, we can inculcate certain updates like :

1. Proper design interface can be made which will be useful for general users who do not have general experience with coding platforms.
2. After developing a proper interface we can host this online and connect it with a database (backend). Database can be connected to cloud (local server) which can be used to aggregate, visualise and process the data in order to draw conclusions.
3. When we are using the project, it can store around 60 to 65 thousand words but after connecting it to database the limit can be extended to a large extent.
4. We can extend this idea to implement Page Rank algorithm followed by Google.

PageRank (PR) is an algorithm used by Google Search to rank websites in their search engine results. PageRank was named after Larry Page, one of the founders of Google. PageRank is a way of measuring the importance of website pages. According to Google:

PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is. The underlying assumption is that more important websites are likely to receive more links from other websites.

It is not the only algorithm used by Google to order search engine results, but it is the first algorithm that was used by the company, and it is the best-known.

The above centrality measure is not implemented for the multi-graphs.

Algorithm

The PageRank algorithm outputs a probability distribution used to represent the likelihood that a person randomly clicking on links will arrive at any particular page. PageRank can be calculated for collections of documents of any size. It is assumed in several research papers that the distribution is evenly divided among all documents in the collection at the beginning of the computational process. The PageRank computations require several passes, called “iterations”, through the collection to adjust approximate PageRank values to more closely reflect the theoretical true value. Cartoon illustrating the basic principle of PageRank. The size of each face is proportional to the total size of the other faces which are pointing to it.

REFERENCES

1. García-López, Félix, et al. "The parallel variable neighborhood search for the p-median problem." *Journal of Heuristics* 8.3 (2002): 375-388.
2. Süß, Michael, and Claudia Leopold. "Implementing irregular parallel algorithms with OpenMP." *European Conference on Parallel Processing*. Springer, Berlin, Heidelberg, 2006.
3. Drews, Frank, Jens Lichtenberg, and Lonnie Welch. "Scalable parallel word search in multicore/multiprocessor systems." *The Journal of Supercomputing* 51.1 (2010): 58-75.
4. Rabenseifner, Rolf, Georg Hager, and Gabriele Jost. "Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes." *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*. IEEE, 2009.
5. Ballard, Clinton L. "Query refinement method for searching documents." *U.S. Patent* No. 5,987,457. 16 Nov. 1999.
6. Fisk, Arthur D., and Walter Schneider. "Category and word search: generalizing search principles to complex processing." *Journal of Experimental Psychology: Learning, Memory, and Cognition* 9.2 (1983):
7. Lee, Seyong, and Rudolf Eigenmann. "OpenMPC: Extended OpenMP programming and tuning for GPUs." *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society, 2010.