# PARALLEL AND DISTRIBUTED COMPUTING (CSE4001)

## J COMPONENT

## REVIEW-3

**TITLE: PARALLEL WORD SEARCH USING OPENMP**

**SUBMITTED TO: DR. DEEBAK B. D.**

**GROUP MEMBERS:**

| NAME | REGISTRATION NUMBER |
|---|---|
| Aman Goel | 18BCE0895 |
| Agnim Chakraborty | 18BCE2186 |
| Sri Siddarth Chakaravarthy | 18BCE2240 |

## *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 .

## *Introduction*

Word search searches for words in multiple text files parallely 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. Word search searches for words in multiple text files parallely 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. 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. 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 an important problem in Text Mining, Information Retrieval and Natural Language Processing.

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

## *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. 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 parallelly 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.

**Type Of Research:** Experimental and Quantitative

## *Literature Review*

Based on our thorough research on the working and limitations of parallel we were able to find that 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. By applying parallelism, it was noticed that the running time significantly reduces, and this helps to process heavy-weight tasks faster with the help of parallelism. We were also able to find more uses of the parallel search algorithms in other fields such bio-informatics in order to analyse the genome material(genome material of organisms is converted to strings and parallel search techniques is used on a large data to find similar genome material. This is used to find organisms that have similar traits. Other uses are in the field of Natural language processing where parallel programming can be used to calculate the TF-IDF (term frequency- inverse document frequency) frequency of words (in bag of words model) this is later used for sentimental analysis, fake review detection, etc. Another interesting research based on parallel programming is the use of parallel approach in the cloud. It was a parallel approach for discovering interacting data entities in data-intensive cloud workflows. Searching for data on the cloud faster and saving resources and time is another observation we were able to review.

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.

Word Searching has been the subject of a large body of previous work. The motivation behind the Teiresias algorithm was to create a novel algorithm that generates all maximal patterns as specific as possible within a set of input sequences without searching the entire solution space [1]. The suffix tree is a data structure that allows for the storage of all the suffixes of a given string in a manner that is more efficient and easier to analyze than other comparable data structures [2]. The main advantage of using a suffix tree to perform operations on a given string is the speed of the operations. Suffix trees have the property to allow for operations to run in linear time in respect

to the length of the given string [1]. However, the increase in speed for string operations cames at the expense of a significant amount of memory allocation to be devoted to storing the suffix tree for a given string [1]. Other word searching approaches include radix sort [3] and the Burrows-Wheeler transform [4] (also called block-sorting compression).
Unlike these special purpose algorithms, our approach attempts to create a very high-speed algorithm that builds a generic representation of the vocabulary of genomic data that gives a flexible way of applying further strategies for Word Scoring and Word Selection that themselves can be parallelized in a way very similar to the algorithm proposed in this paper. .
Several other previous relevant papers that discuss issues related to our presented work are worth noting here. This issue of cache-aware data structures and data layout has been the context of some existing work [5-10]. Strategies for careful data layout by performing cache-conscious memory allocation were discussed in [6] and [7]. These strategies aimed at improving the cache usage of pointer-intensive data structures by improving the access locality of algorithms. Rao & Ross and other researchers developed cache-sensitive search tree techniques that aim at improving the locality of references by changing the data layout in order to reduce pointer chasing operations [8-15].

While much of the aforementioned work is focused on efficient and space-optimized storage of strings in memory the focus of our work is on developing an efficient parallel word search algorithm by improving the access locality and cache-usage of the generic word search algorithm, while largely avoiding the need for locking. In our approach, we exploit the special correlation between input sequence data and output data to develop a cache-conscious parallelization of word storage and retrieval in the context of the particular nature of building a vocabulary for genome sequences.

## *Significance*

Word searches involve pattern matching. Word search searches for words in multiple text files parallely 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.
Applications of parallel word search:

- Parallel Word Search can be used fasten the process of  bags of words generation and thereby we can produce TFIDF (Term Frequency-Inverse Document Frequency) values which will help us in Page Rank Algorithms.
- Word search is used to determine the set of biological words of an input DNA sequence (complex genome sequence).
- Instagram uses parallel word search as in when you search "computer" you get the result for all the fields such as accounts, hashtags, places and tags.
- Parallel word search is also used by google to crawl millions of webpages every day.
  It ranks the documents based on the relevance of the documents with respect to the searched word which is mostly used in search engines. Similarly, when we generally search for something say “apple” we get parallel search results in all, images, videos, etc. in google at the same time.
- Parallel searching is used by amazon to fetch results from billions of products listed on its website within a few milliseconds.
- Ctrl-F and Command -F are used by Windows and Mac users respectively as shortcuts in your browser or operating system that allows you to find words or phrases quickly.
- Notable applications for parallel search (or parallel computing) include computational astrophysics, geoprocessing (or seismic surveying), climate modelling, agriculture estimates, financial risk management, video color correction, computational fluid dynamics, medical imaging and drug discovery.
- AI systems require a large amount of parallel computing for which they are used i.e. image processing , expert systems , natural language processing, pattern recognition, etc.
- Web 3.0 will be more connected, open, and intelligent, with semantic Web technologies, distributed databases, natural language processing, machine learning, machine reasoning, and autonomous agents. As we move towards the development of web 3.0 which primarily focuses semantic databases and semantic searches the use of parallel searches will play an important role. The future of web also depends on faster and meaningful searches. During the next generation of web the development of intelligent systems will take place on the web and the need for distributed searches to take place where the use of parallel word search would be more significant.

## *Methodology*

Word search or Pattern matching has been one of the most extensively studied problems in computer engineering since it performs important tasks in many applications like information retrieval systems (IRS), web search engines (SE), error correction and several other fields. Especially with the introduction of search engines dealing with tremendous amount of textual information presented on the World Wide Web, so this problem deserves special attention and any improvements to speed up the process will benefit these important applications. Word searches involve pattern matching. 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.

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. We design parallel agorithms in the following model: p synchronized processors (RAMs) share a common memory. Any subset of the processors can simultaneously read from the same memory location. We sometimes allow simultaneous writing in the weakest sense: any subset of processors can write the value 0 into the same memory location.

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 betweennodes. 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.

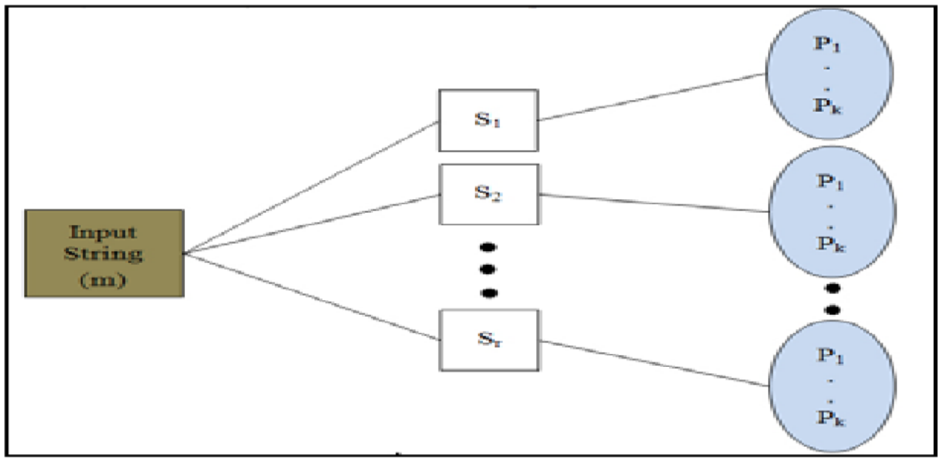


**Fig. 1.** String processing using parallel processing

The exact string-matching problem can achieve data parallelism with data partitioning technique. We decompose the text into r subtexts, where each subtext contains (T/p)+m-1 successive characters of the complete text. There is an overlap of m-1 string characters between successive subtexts, i.e., a redundancy of r(m-1) characters. Alternatively it could be assumed that the database of an information retrieval system contains r independent documents. Therefore, in both the cases all the above partitions yield a number of independent tasks each comprising some data (i.e. a string and a large subtext) and a sequential string matching procedure that operates on that data. Further, each task completes its string matching operation on its local data and returns the number of occurrences

In this model data on processors have been organized such that they represent the m sets of length of n-m+1 of the text string with m* n-m+1 matrix plus, the first processor of each row segment holding the first element of each set also carries an element of pattern. The process is similar as per above for the remaining m-1 rows. First show how to find the occurrences of pattern P in text string T on Butterfly model with m*(n-m+1) in constant time O (1). We designed an algorithm, which is designed based on the Hamming distance. Hamming distance error correction. So it is best of known for finding of differences between two strings. The Hamming distance is a measure of distance between two strings equal is the number of positions for which the corresponding symbols may be different, that need to be changed to obtain one from the other

## *Algorithm used (Pattern matching)*

1. Begin
2. **Initially:** m elements of pattern initially distributed to the m processors on the first column, one processor and Li,j distributed to the m*(n-m+1) processors on the row and column. Where 1≤i≤m and m≤j≤m*(n-m+1)
3. **First processors broadcast elements on row buses:** each first processor pi,1, where 1≤i≤m, broadcasts the element ci,1 to every processor in the ith row using only row buses .After this multiple row broadcast communication operations each processor Pi,j saves received element as ri,j.
4. **Compare and Set results:** Each processor Pi,j compares ri,j with Xi,j, if ri,j = Xi,j if sets result=1 otherwise, result=0
5. **Sum up 1's:** perform a one-dimensional binary prefix sum operation on each column simultaneously for the value of result. Each processor Pi,j where 1≤ (i,j) ≤ m stores the binary prefix sums to bi,j.
6. **Based on Hamming Distance:** If Pm,j =0 then the string matching(i.e. exact string matching) otherwise approximate string matching with k mismatches.
7. End.

## *Implementation*

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 parallelizes 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 ooccurrence.

## **Programs: -**

### **Sequential Code:**

```
#include<bits/stdc++.h>
#include <stdlib.h>
#include <omp.h>
#include <iostream>
#include <ctime>
#include <string>
#include <fstream>

using namespace std;

int find_all(string sen, string word_to_search) {
    int count = 0;
    // Used to split string around spaces.
    istringstream ss(sen);
    // Traverse through all words
    do {
        // Read a word
        string word;
        ss >> word;
        if(word == word_to_search){
            count++;
        }
    } while (ss);
    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 word_to_search = "";
        cout << "Enter a word to search: ";
        cin >> word_to_search;
        int i = 1;

        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;
}
```

**Parallel Code:**

```
#include <stdlib.h>
#include<bits/stdc++.h>
#include <omp.h>
#include <iostream>
#include <string>
#include <fstream>
using namespace std;

int find_all(string sen, string word_to_search) {
    int count = 0;
    // Used to split string around spaces.
    istringstream ss(sen);
    // Traverse through all words
    do {
        // Read a word
        string word;
        ss >> word;
        if(word == word_to_search){
            count++;
        }
    } while (ss);
    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;
}
```

## *Challenges*

The issues of this project are 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.

## *System Requirements*

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, including desktop applications (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.

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++ .

## Tools Description

- Ubuntu Version 20.04
- Windows 10 64 bit
- C++ compiler
- Some header files
- File Handling Concepts

## Specifications

### Headers Used: -

- **stdlib.h:** This header defines several general-purpose functions, including dynamic memory management, random number generation, communication with the environment, integer arithmetic, searching, sorting and converting.
- **omp.h:** OpenMP is a library for parallel programming in the SMP (symmetric multi-processors, or shared-memory processors) model. When programming with OpenMP, all threads share memory and data. OpenMP supports C, C++ and Fortran. The OpenMP functions are included in a header file called omp.h .
- **iostream.h:** Header that defines the standard input/output stream objects
- **String:** This header file defines several functions to manipulate C strings and arrays.
- **Fstream:** Header providing file stream classes.

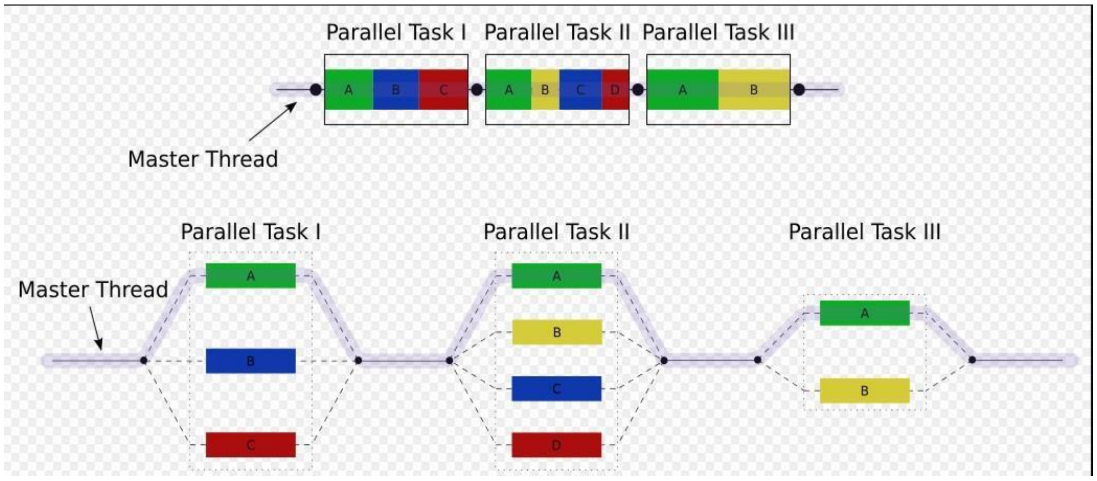


**Fig. 2.** Fork join model

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. 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.

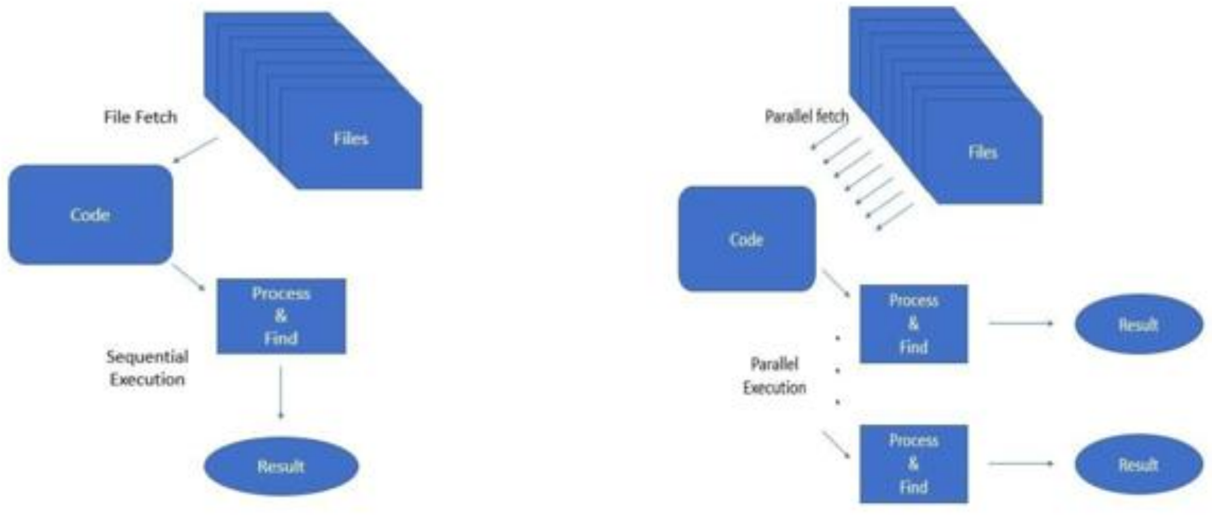
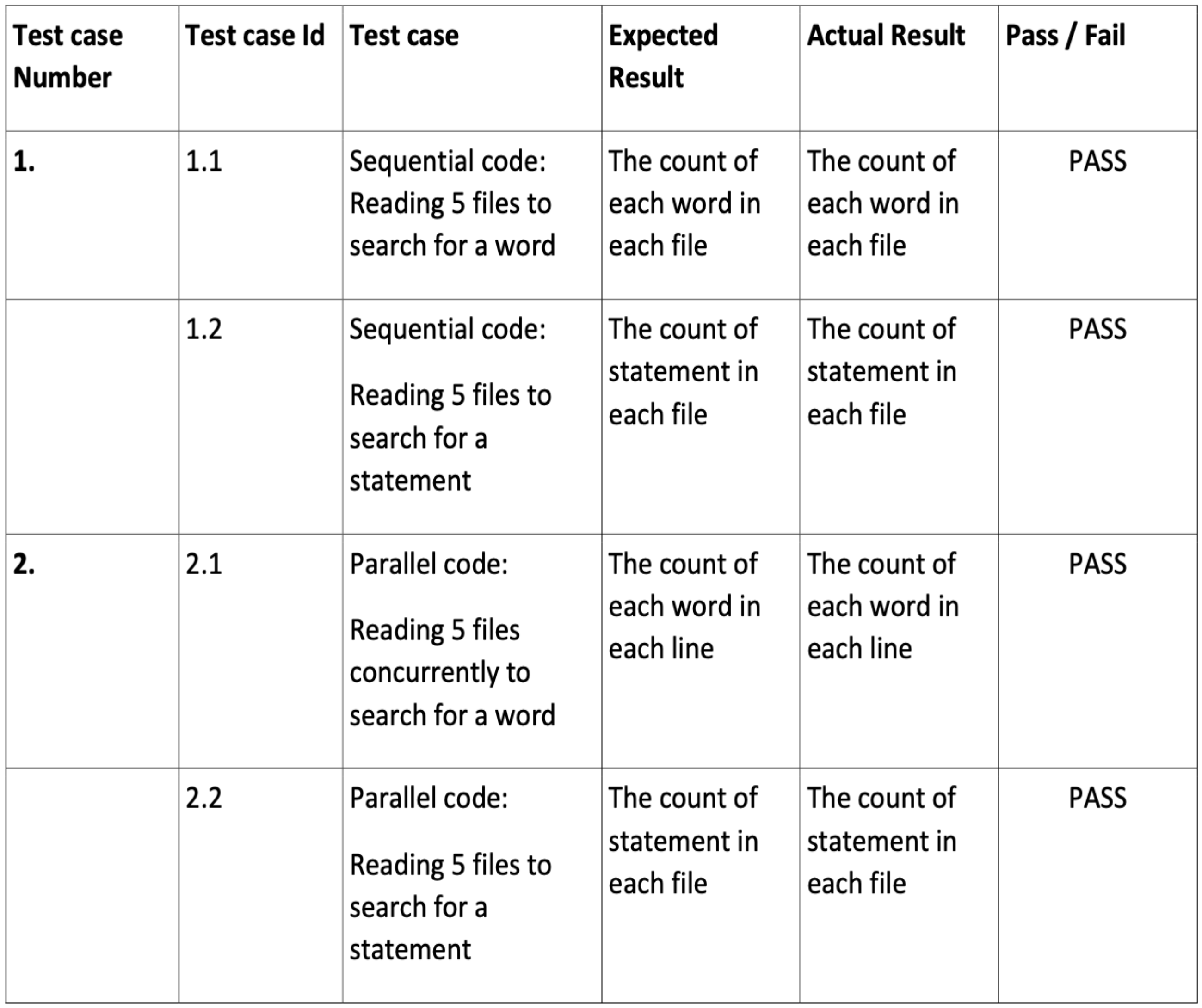


**Fig. 3.** Sequential (left) and Parallel (right) model

## *Testing and Result Analysis*



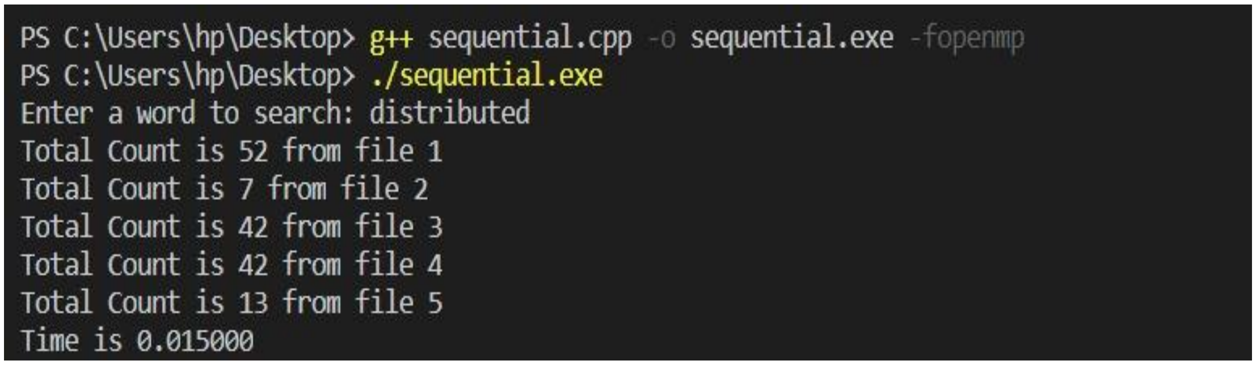
| Test case Number | Test case Id | Test case | Expected Result | Actual Result | Pass / Fail |
|---|---|---|---|---|---|
| **1.** | 1.1 | Sequential code: Reading 5 files to search for a word | The count of each word in each file | The count of each word in each file | PASS |
|  | 1.2 | Sequential code: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

## *Observations*

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 find the count of all the word in the files and sums it all.The output will be count of occurences of statement in the files.

## Output Screenshots

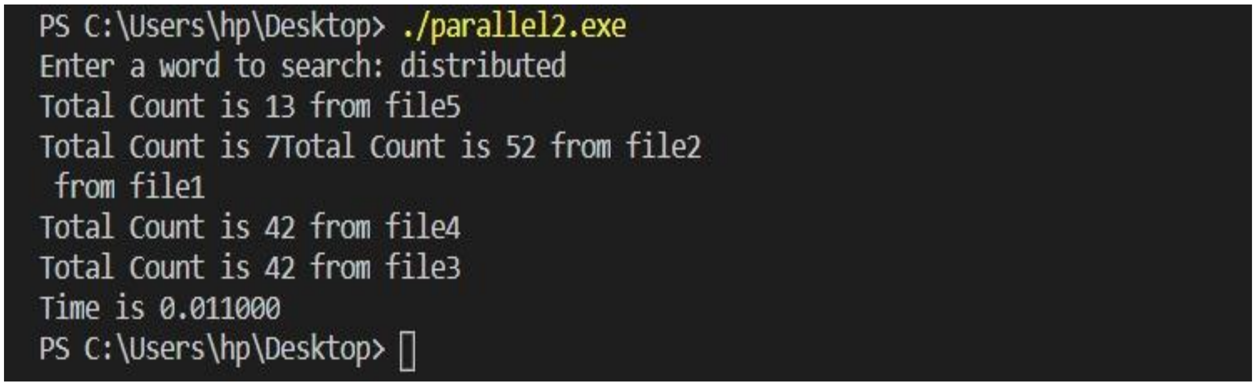
### Sequential code



```
PS C:\Users\hp\Desktop> g++ sequential.cpp -o sequential.exe -fopenmp
PS C:\Users\hp\Desktop> ./sequential.exe
Enter a word to search: distributed
Total Count is 52 from file 1
Total Count is 7 from file 2
Total Count is 42 from file 3
Total Count is 42 from file 4
Total Count is 13 from file 5
Time is 0.015000
```

**Sequential search time = 0.015 seconds or  15 milliseconds**

### Parallel code

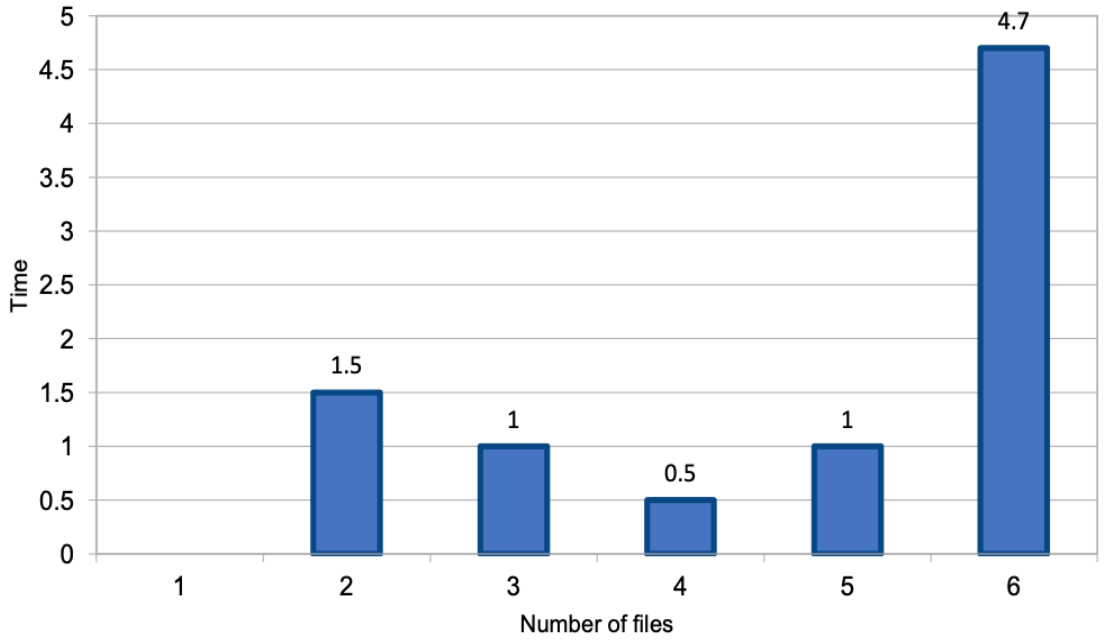


```
PS C:\Users\hp\Desktop> ./parallel2.exe
Enter a word to search: distributed
Total Count is 13 from file5
Total Count is 7Total Count is 52 from file2
 from file1
Total Count is 42 from file4
Total Count is 42 from file3
Time is 0.011000
PS C:\Users\hp\Desktop> ▯
```

**Parallel search time = 0.011 seconds or 11 milliseconds**

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.



Some sample examples were recorded with the time taken in sequential and parallel mode. We can observe a slight improvement in the parallel mode. Thus, we can conclude that the parallel mode would definitely be more superior and takes less time to execute. If we increase the number the documents we can even observe more significant difference between the two modes of execution.

| 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*

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: -

a. Proper design interface can be made which will be useful for general users who do not have general experience with coding platforms.
b. 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.
c. 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.
d. 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.
The issues 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.

## *Future Scope*

The use of word search has been drawing attention recently. There seems to be a need for a new way of searching words, pattern or data that does not only return the matched string but also returns the meaning of the searched or importance of the word in the result. The use of semantic searches have been preferred by many in the field of medicine, commerce, and stock market. With the improvement in the web as in the change in the web generations from web 2.0 to **WEB 3.0** semantic searches play a major role. In the future the searching will become distributed searches unlike the present version of search engines where the searched are executed on a single fixed server, in the case of distributed search the search engine model performs tasks of Web crawling, indexing and query processing are distributed among multiple computers and networks. In the field of medicine searching plays a major for both descriptive and analysis statistics where we can search for patient details and also find similarities between patients. This can we very useful during the covid-19 pandemic when we deal with a large pool of patients. As we can see there is a lot of scope for the field of parallel word search.

## *References*

### Base Paper

"Scalable Parallel Word Search in Multi-Core/Multiprocessor Systems", Drews, F., Lichtenberg, J., & Welch, L. (2010). Scalable parallel word search in multicore/multiprocessor systems. The Journal of Supercomputing, 51(1), 58-75.

### Reference Papers

i. Rigoutsos, A. Floratos, "Combinatorial pattern discovery in biological sequences: The TEIRESIAS algorithm," Bioinformatics, Vol. 14, No. 1, 55-67, 1998.

ii. D. Gusfield. Algorithms on strings, trees and sequences: computer science and computational biology. Cambridge University Press, Cambridge, UK, 1997.

iii. Askitis, Nikolas & Sinha, Ranjan (2007), "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings", Proceedings of the 30th Australasian Conference on Computer science 62: pp. 97-105.

iv. Glen L. Beane, "The effects of Microprocessor Architecture on Speedup in Distributed Memory Supercomputers" (M.S. thesis, The University of Maine, 2004) http://www.umcs.maine.edu/~beaneg/docs/thesis.pdf

v. Drews, F., Lichtenberg, J., & Welch, L. (2010). Scalable parallel word search in multicore/multiprocessor systems. The Journal of Supercomputing, 51(1), 58-75.

vi. García, L. L., Arellano, A. G., & Cruz-Santos, W. (2020). A parallel path-following phase unwrapping algorithm based on a top-down breadth-first search approach. Optics and Lasers in Engineering, 124, 105827.

vii. Hallberg, J., Palm, T., & Brorsson, M. (2003, June). Cache-conscious allocation of pointer-based data structures revisited with HW/SW prefetching. In Second Annual Workshop on Duplicating, Deconstructing, and Debunking. San Diego, California, United States.The design and analysis of parallel algorithms.

viii. Aki, S. G. (1989). The design and analysis of parallel algorithms.

ix. Gusfield, D. (1997). Algorithms on stings, trees, and sequences: Computer science and computational biology. Acm Sigact News, 28(4), 41-60.

x. Huang, Y., Huang, J., Liu, C., & Zhang, C. (2020). PFPMine: A parallel approach for discovering interacting data entities in data-intensive cloud workflows. Future Generation Computer Systems.

xi. 1. García-López, Félix, et al. &quot;The parallel variable neighborhood search for the p-median problem.&quot; Journal of Heuristics 8.3 (2002): 375-388.

xii. Süß, Michael, and Claudia Leopold. &quot;Implementing irregular parallel algorithms with OpenMP.&quot; European Conference on Parallel Processing. Springer, Berlin, Heidelberg, 2006.

xiii. Drews, Frank, Jens Lichtenberg, and Lonnie Welch. &quot;Scalable parallel word search in multicore/multiprocessor systems.&quot; The Journal of Supercomputing 51.1 (2010): 58-75.

xiv. Rabenseifner, Rolf, Georg Hager, and Gabriele Jost. &quot;Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes.&quot; Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on. IEEE, 2009.

xv. Lee, Seyong, and Rudolf Eigenmann. &quot;OpenMPC: Extended OpenMP programming and tuning for GPUs.&quot; Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society, 2010.