

PARALLEL WORD SEARCH USING OPENMP

Submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology in Computer Science and Engineering

by

**FRANCIS ALEX KUZHIPALLIL
18BCE2325**

Under the guidance of

Mrs Preetha

Evangeline D

School of Computer Science and Engineering

VIT, Vellore.



October, 2020

TABLE OF CONTENTS

SNO	CONTENT	PG NO
1	ABSTRACT	3
2	INTRODUCTION	3
3	LITERATURE SURVEY	5
4	PROPOSED METHODOLOGY	7
5	DESIGN ARCHITECTURE	9
6	MODULE WISE DESCRIPTION	10
7	RESULTS	12
8	CODE AND SCREENSHOTS	14
9	REFERENCES	32

ABSTRACT

Word search searches for words in multiple text files in parallel using open MP 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 as well as handling multiple tables of the database. The conclusion of this project will give us an idea about whether parallel execution is able to enhance the performance of the word search as compared to serial execution. This could be helpful for future word searching engines, word puzzles and lot more real life applications.

INTRODUCTION

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 in parallel using open MP. 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 pthreads 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.

LITERATURE SURVEY

[1] Peng Xu states that Cloud assisted Internet of Things (IOT) is a popular system that combines the advantage of both cloud and IOT. Further upon receiving a keyword search trapdoor one can disclose all corresponding relationships in parallel followed by finding all matching cipher text. Hence the new relationship allows a keyword search task to be performed in parallel. This new instance enhances the efficiency and reduces time and communication cost.

[2] Frank Drew's proposes a parallel algorithm technique to determine the sets of biological word of an input DNA sequence. The algorithm is designed to scale well on state-of-the-art multiprocessor/multicore systems for large inputs and large maximum word sizes. The proposed algorithm's performance was compared to the sequential implementation. The result was drastically improved on using parallel execution.

[3] Sinan adopts quick search string matching algorithm technique implemented under multi-core environment using Open MP directive which is used to parallelize the code and thereby reduce the overall execution time of the program. There was a significant improvement in performance of parallel execution as compared to serial execution.

[4] Oleksij uses parallelization to increase the efficiency of different kinds of word search algorithms interconnected with an SQL relational database. Results of this will provide valuable information for the design of an efficient helping system for solving word games.

[5] Podhorszki states that information volumes are increasing at a rate of 25%-35% and transaction rate is also increasing by a factor of 10%. Hence it is very important for increasing the performance of retrieval of data from database. Hence the data flow approach to database is very important. The dataflow approach requires message based client server OS to interconnect parallel processes executing relational operators. Better dataflow mechanism yields improves the performance of the system.

[6] Fernando states that data of internet of things twenty times more than the planet earth. The aim of performing the research was to showcase the importance of data and database systems which is a necessity to satisfy the current application needs. Parallelization plays an enormous roles in increasing the responsiveness of the database.

[7] David explains that Parallel database machine architectures have evolved from the use of exotic hardware to a software parallel dataflow architecture based on conventional shared-nothing hardware. These new designs provide impressive speedup and scale up when processing Relational database queries.

[8] Zoe explains about the importance of file systems to store data. Parallelization of file systems helps improvise to faster storage of data. Zoe further reveals two prominent modern parallel file systems,PVFS2 and Lustre, and compare them experimentally on a range of benchmark-driven scenarios,modeling specific real-world applications.

[9] Wright tries to investigate the performance gains attributed to the Parallel Log-structured File System (PLFS) being developed by EMC Corporation and the Los Alamos National Laboratory. Our evaluation of PLFS involves two HPC systems with contrasting I/O backplanes and illustrates the varied improvements to I/O that result from the deployment of PLFS (ranging from up to 25 x speed-up in I/O performance on a large I/O installation to 2 x speed-up on the much smaller installation at the University of Warwick).

[10] Gerard does a comparison of recently proposed parallel text search methods to alternative available search strategies that use serial processing machines suggests parallel methods do not provide large-scale gains in either retrieval effectiveness or efficiency.

PROPOSED METHODOLOGY

PROBLEM STATEMENT

Word search searches for words in multiple text files in parallel using open MP 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.

Open MP 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 Open MP and Message Passing Interface (MPI), such that Open MP is used for parallelism within a (multi-core) node while MPI is used for parallelism between nodes. There have also been efforts to run Open MP on software distributed shared memory systems, to translate Open MP into MPI and to extend Open MP for non-shared memory systems.

Open MP 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 Open MP 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++.

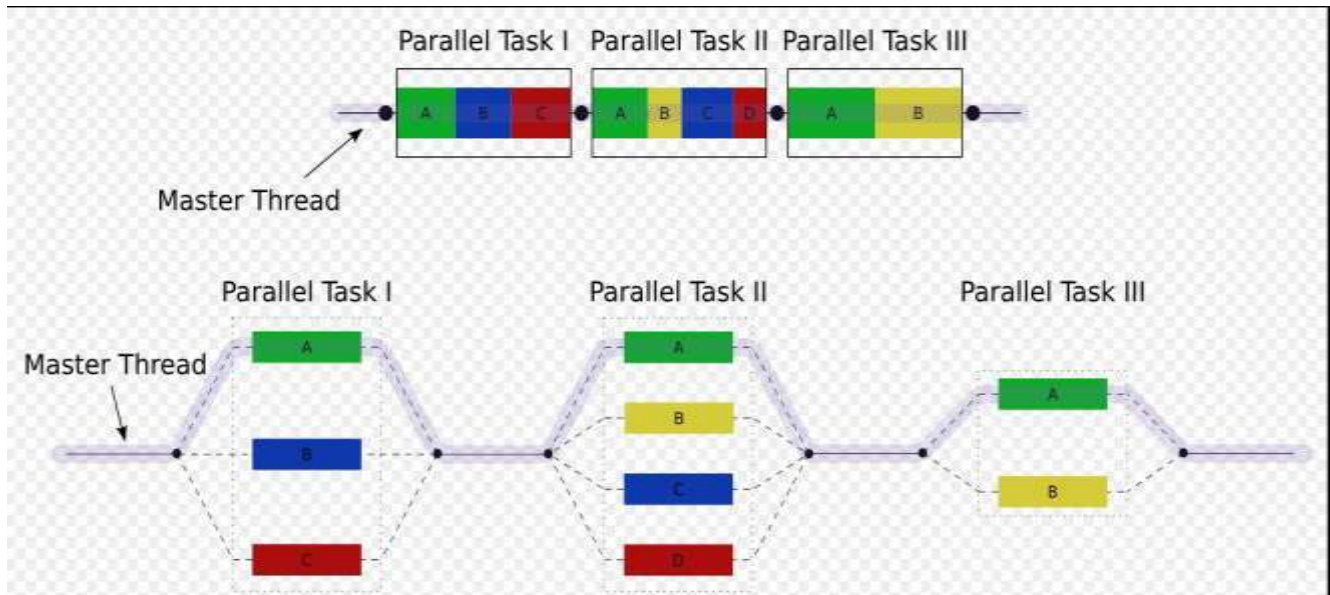


FIG: FORK JOIN MODEL

DESIGN ARCHITECTURE

1) Sequential Model

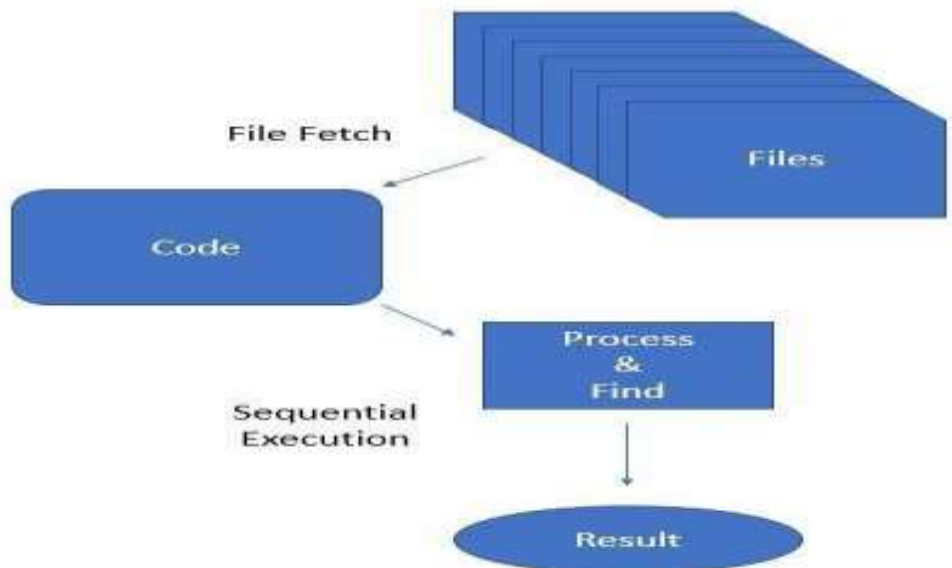


FIG: Sequential Model

2) Parallel Model

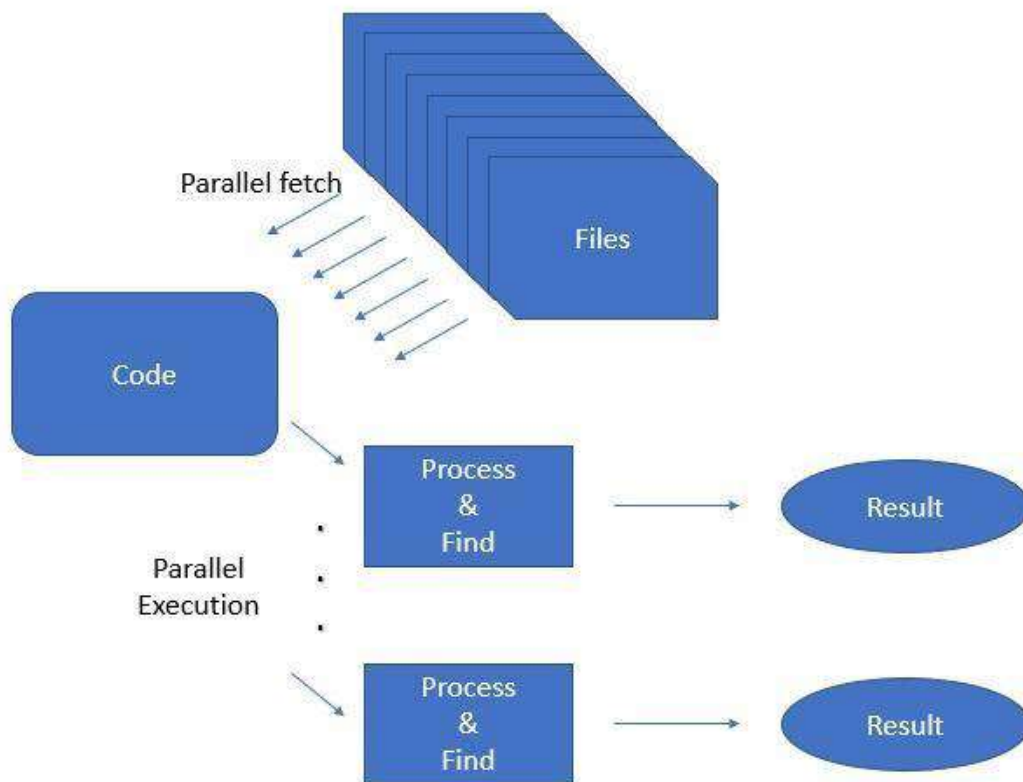


FIG: PARALLEL MODEL

MODULE WISE DESCRIPTION

CASE1: ANALYSING SERIAL VS PARALLEL EXECUTION WITH FILE SYSTEM

i) Module 1: Analyzing serial execution of word search in text files

- ➔ Here almost 5 text files are loaded with data. Then a particular word is entered as input and the word is being checked in each of the file's data one after another.

ii) Module 2: Analyzing parallel execution of word search in text files

- ➔ Here almost 5 text files is loaded with data. Then a particular word is entered as input and the word is being checked in each of the file.
- ➔ #pragma omp enables to execute the word search in parallel.
- ➔ The concept used is fork join. Each sub thread executes file search for all the 5 files.

Hence printing the count of words in each file

iii) Module 3: Analyzing serial execution of word search in pdf files

Here almost 5 pdf files is loaded with data. Then a particular word is entered as input and the word is being checked in each of the file's data one after another.

iv) Module 4: Analyzing parallel execution of word search in pdf files (not optimal)

- ➔ Here almost 5 text files is loaded with data. Then a particular word is entered as input and the word is being checked in each of the file.
- ➔ #pragma omp enables to execute the word search in parallel.
- ➔ The concept used is fork join. Each sub thread executes word search for all the 5 files.
- ➔ Hence printing the count of words in each file.

v) Module 5: Analyzing parallel execution of word search in pdf files (optimal)

- ➔ Here almost 5 text files is loaded with data. Then a particular word is entered as input and the word is being checked in each of the file.
- ➔ #pragma omp enables to execute the word search in parallel.
- ➔ #pragma for parallelizes the for loop.
- ➔ The concept used is fork join. Each sub thread executes word search for each file and not the entire file.
- ➔ Hence printing the count of words in each file.

CASE2: ANALYSING SERIAL VS PARALLEL EXECUTION WITH SQL DATABASE

i) Module 6: Analyzing serial execution of word search in SQL database

- ➔ Here almost 5 tables are created and loaded with data. Then a particular word is entered as input and the word is being checked in with each table of the database.

ii) Module 7: Analyzing parallel execution of word search in SQL database

- ➔ Here almost 5 tables are created and loaded with data. Then a particular word is entered as input and the word is being checked in each of the tables in SQL database.
- ➔ #pragma omp enables to execute the word search in parallel.
- ➔ The concept used is fork join. Each sub thread executes word search for all the 5 tables with one thread does word search for all the tables.
- ➔ Hence printing the count of words in each table of database.

v) Module 8: Analyzing parallel execution of word search in SQL database (optimal)

- ➔ Here almost 5 tables of database are loaded with data. Then a particular word is entered as input and the word is being checked in each of the tables.
- ➔ #pragma omp enables to execute the word search in parallel.
- ➔ #pragma for parallelizes the for loop.
- ➔ The concept used is fork join. Each sub thread executes word search for each table and not the entire 5 tables at a time. Hence printing the count of words in each table.

RESULTS

Our aim was to analyze how serial execution was performing as compared to parallel execution.

ANALYSIS 1: FOR SMALLER FILES (TXT FILES)

FILE	SERIAL EXECUTION TIME	PARALLEL EXECUTION TIME
TXT	0.483 Seconds	0.149 Seconds

ANALYSIS 2: FOR LARGER FILES (PDF FILES)

FILE	SERIAL EXECUTION TIME	PARALLEL EXECUTION TIME	OPTIMIZED PARALLEL EXECUTION
PDF	33.42 Seconds	49.06 Seconds	16.20 Seconds

ANALYSIS 3: FOR SQL DATABASE

DATABASE	SERIAL EXECUTION TIME	PARALLEL EXECUTION TIME	OPTIMIZED PARALLEL EXECUTION
PHP MYADMIN	77.00 milli seconds	77.99 milli seconds	48.99 milli seconds

FINALLY WE CONCLUDE THAT PARALLEL EXECUTION DOES TASK EFFICIENTLY WITH LESSER AMOUNT OF TIME.

CODE AND SCREENSHOTS

CASE1: ANALYSING SERIAL VS PARALLEL EXECUTION WITH FILE SYSTEM

1) Serial Execution with text file


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:\\Users\\DELL\\Desktop\\Parallel-word-Search-using-Openmp-and-PHP-master";
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;
}

```

 "C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\Pagerank.exe"

```

Enter a word to search: of
Total Count is 49 from file 1
Total Count is 159 from file 2
Total Count is 32 from file 3
Total Count is 170 from file 4
Total Count is 517 from file 5
Time is 0.483000
Process returned 0 (0x0)   execution time : 4.878 s
Press any key to continue.

```

2) Parallel Execution with text file

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 (int i = 1; i <= 5; i++) {
        string npath = "File" + to_string(i) + ".txt";
        // #pragma omp critical
        cout << "\nTotal 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;
}

```


"C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\Pagerank_parallel.exe"

Enter a word to search: of

Total Count is 49 from file 1

Total Count is 49 from file 1

Total Count is 49 from file 1

Total Count is 49 from file 1

Total Count is 159 from file 2

Total Count is 159 from file 2

Total Count is 159 from file 2

Total Count is 159 from file 2

Total Count is 32 from file 3

Total Count is 32 from file 3

Total Count is 32 from file 3

Total Count is 32 from file 3

Total Count is 170 from file 4

Total Count is 170 from file 4

Total Count is 32 from file 3

Total Count is 170 from file 4

Total Count is 170 from file 4

Total Count is 170 from file 4

Total Count is 170 from file 4

Total Count is 517

Total Count is 517 from file 5
from file 5

Total Count is 517 from file 5

Total Count is 517 from file 5

Time is 0.149000

Process returned 0 (0x0) execution time : 4.765 s

Press any key to continue.

3) Serial Execution of Pdf files

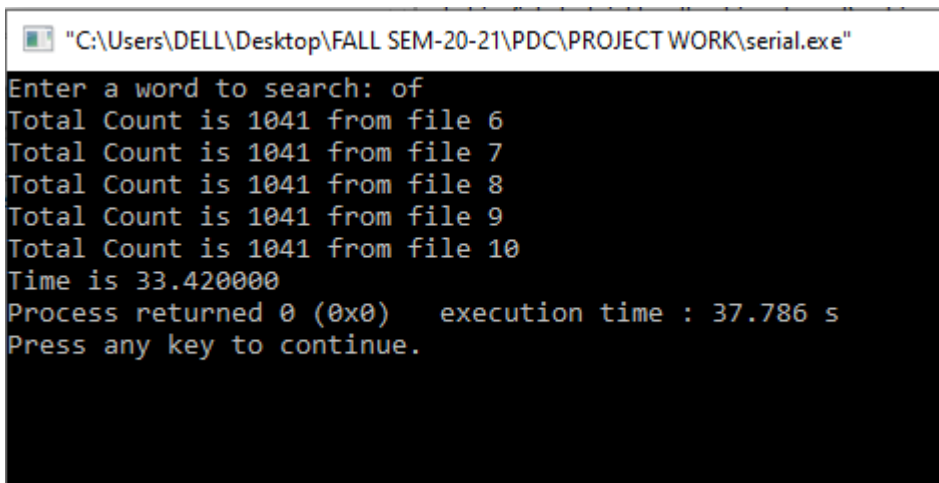
Code:

```
#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:\\Users\\DELL\\Desktop\\Parallel-word-Search-using-Openmp-and-PHP-master";
    string word_to_search = ""; cout << "Enter a word to search: "; cin >> word_to_search;
    double time = omp_get_wtime();
    for (int i = 6; i <= 10; i++) {
        string npath = "File" + to_string(i) + ".pdf";
```

```

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;
}

```



```

"C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\serial.exe"
Enter a word to search: of
Total Count is 1041 from file 6
Total Count is 1041 from file 7
Total Count is 1041 from file 8
Total Count is 1041 from file 9
Total Count is 1041 from file 10
Time is 33.420000
Process returned 0 (0x0)   execution time : 37.786 s
Press any key to continue.

```

2) PARALLEL EXECUTION OF PDF FILES

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 (int i = 6; i <= 10; i++) {
string npath = "File" + to_string(i) + ".pdf";
//#pragma omp critical
cout << "\nTotal 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;
}

```

"C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\

Enter a word to search: of

Total Count is 1041 from file 6

Total Count is 1041 from file 6

Total Count is 1041 from file 6

Total Count is 1041 from file 6

Total Count is 1041 from file 7

Total Count is 1041 from file 7

Total Count is 1041 from file 7

Total Count is 1041 from file 7

Total Count is 1041 from file 8

Total Count is 1041 from file 8

Total Count is 1041 from file 8

Total Count is 1041 from file 8

Total Count is 1041 from file 9

Total Count is 1041 from file 9

"C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\normal_s

Total Count is 1041 from file 9

Total Count is 1041 from file 9

Total Count is 1041 from file 9

Total Count is 1041 from file 9

Total Count is 1041 from file 10

Total Count is 1041 from file 10

Total Count is 1041 from file 10

Total Count is 1041 from file 10

Time is 49.060000

Process returned 0 (0x0) execution time : 52.132 s

Press any key to continue.

5) OPTIMIZED PARALLEL EXECUTION OF PDF FILES

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);
            }
        }
    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

//#pragma omp single nowait


#pragma omp for

for (int i = 6; i <= 10; i++) {
string npath = "File" + to_string(i) + ".pdf";
cout << "\nTotal Count is " + to_string(display(npath, word_to_search)) << " from file " + to_string(i) << endl;
}

time = omp_get_wtime() - time; cout << "\nTime is " + to_string(time);
return 0;

}

```

 "C:\Users\DELL\Desktop\FALL SEM-20-21\PDC\PROJECT WORK\parallel.exe"

```

Enter a word to search: of

Total Count is 1041 from file 8
Total Count is 1041 from file 6
Total Count is 1041 from file 9
Total Count is 1041 from file 10
Total Count is 1041 from file 7

Time is 16.204000
Process returned 0 (0x0)   execution time : 19.021 s
Press any key to continue.

```

CASE2: ANALYSING SERIAL VS PARALLEL EXECUTION WITH SQL DATABASE

SCREENSHOTS OF DATABASE:

Server: 127.0.0.1 » Database: pdc

Structure SQL Search Query Export Import Operations Privileges Routines Events

Filters

Containing the word:

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> table1	★ Browse Structure Search Insert Empty Drop	3,281	InnoDB	utf8mb4_general_ci	192.0 KiB	-
<input type="checkbox"/> table2	★ Browse Structure Search Insert Empty Drop	6,234	InnoDB	utf8mb4_general_ci	336.0 KiB	-
<input type="checkbox"/> table3	★ Browse Structure Search Insert Empty Drop	11,871	InnoDB	utf8mb4_general_ci	1.5 MiB	-
<input type="checkbox"/> table4	★ Browse Structure Search Insert Empty Drop	22,951	InnoDB	utf8mb4_general_ci	1.5 MiB	-
<input type="checkbox"/> table5	★ Browse Structure Search Insert Empty Drop	45,098	InnoDB	utf8mb4_general_ci	2.5 MiB	-
5 tables	Sum	89,347	InnoDB	utf8mb4_general_ci	6.1 MiB	0 B

Server: 127.0.0.1 » Database: pdc » Table: table1

Browse Structure SQL Search Insert

+ Options

PID	PNAME	PCOMPANY	QTY
1	lamb	australia	100
2	lamb	india	100
3	lamb	ethiopia	100
4	lamb	pakistan	100
5	lamb	saudi	100
6	lamb	sudan	100
7	lamb	georgia	100
8	mutton	india	100
9	mutton	pakistan	100
10	mutton	australia	100
11	mutton	ethiopia	100
12	mutton	sudan	100
13	mutton	england	100
14	chicken	india	100
15	chicken	pakistan	100
16	chicken	australia	100
17	chicken	ethiopia	100
18	chicken	sudan	100
19	chicken	england	100
20	beef	india	100
21	beef	pakistan	100
22	beef	australia	100
23	beef	ethiopia	100
24	beef	sudan	100
25	beef	england	100

1 > >> | Number of rows: 25 | Filter rows: S

Server: 127.0.0.1 » Database: pdc » Table: table2

Browse Structure SQL Search Insert

⚠ Current selection does not contain a unique column. Grid edit, checkbox

✓ Showing rows 0 - 24 (6234 total, Query took 0.0010 seconds.)

```
SELECT * FROM `table2`
```

1 > >> | Number of rows: 25 Filter rows

+ Options

PID	PNAME	PCOMPANY	QTY
1	shirt	max	100
2	shirt	londonbridge	100
3	shirt	oxo	100
4	shirt	addidas	100
5	shirt	nike	100
6	shirt	amer	100
7	shirt	giordano	100
8	pant	londonbridge	100
9	pant	oxo	100
10	pant	addidas	100
11	pant	nike	100
12	pant	amer	100
13	pant	giordano	100
14	kurti	londonbridge	100
15	kurti	oxo	100
16	kurti	addidas	100
17	kurti	nike	100
18	kurti	amer	100
19	kurti	giordano	100
20	leggings	londonbridge	100

Server: 127.0.0.1 » Database: pdc » Table: table3

Browse Structure SQL Search Insert Export

⚠ Current selection does not contain a unique column. Grid edit, checkbox, Edit, Copy a

✓ Showing rows 0 - 24 (11871 total, Query took 0.0010 seconds.)

`SELECT * FROM 'table3'`

1 > >> Number of rows: 25 Filter rows: Search this

+ Options

PID	PNAME	PCOMPANY	QTY
1	tv	sony	100
2	tv	samsung	100
3	tv	onida	100
4	tv	lg	100
5	tv	panasonic	100
6	tv	apple	100
7	mobile	samsung	100
8	mobile	onida	100
9	mobile	lg	100
10	mobile	panasonic	100
11	mobile	apple	100
12	fridge	samsung	100
13	fridge	onida	100
14	fridge	lg	100
15	fridge	panasonic	100
16	fridge	apple	100
17	ac	samsung	100
18	ac	onida	100
19	ac	lg	100
20	ac	panasonic	100

Server: 127.0.0.1 » Database: pdc » Table: table4

Browse Structure SQL Search Insert Edit

⚠ Current selection does not contain a unique column. Grid edit, checkbox, Edit, Copy

✓ Showing rows 0 - 24 (22951 total, Query took 0.0010 seconds.)

```
SELECT * FROM table4
```

1 > >> | Number of rows: 25 | Filter rows: Search

+ Options

PID	PNAME	PCOMPANY	QTY
1	icecream	kwaliti	100
2	icecream	igloo	100
3	icecream	waldilal	100
4	icecream	london diary	100
5	icecream	baskin	100
6	icecream	johns	100
7	chips	lays	100
8	chips	kurkure	100
9	chips	bingo	100
10	chips	pringles	100
11	chips	go	100
12	chocolate	dairymilk	100
13	chocolate	galaxy	100
14	chocolate	merci	100
15	chocolate	toblerone	100
16	chocolate	lindtt	100
17	milk	dairymilk	100
18	milk	galaxy	100
19	milk	merci	100
20	milk	toblerone	100

Server: 127.0.0.1 » Database: pdc » Table: table5

Browse Structure SQL Search Insert

⚠ Current selection does not contain a unique column. Grid edit, checkbox, Edit.

✓ Showing rows 0 - 24 (45090 total. Query took 0.0010 seconds.)

SELECT * FROM 'table5';

1 > >> | Number of rows: 25 | Filter rows: Se

+ Options

PID	PNAME	PCOMPANY	QTY
1	pen	inoxrom	100
2	pen	cello	100
3	pen	zebra	100
4	pen	octane	100
5	pen	linx	100
6	pen	pilot	100
7	pen	pentel	100
8	pencil	steadler	100
9	pencil	nataraj	100
10	pencil	cello	100
11	pencil	inoxrom	100
12	pencil	zebra	100
13	pencil	pentel	100
14	pencil	octane	100
15	eraser	steadler	100
16	eraser	nataraj	100
17	eraser	cello	100
18	eraser	inoxrom	100
19	eraser	zebra	100
20	eraser	pentel	100

1) Serial Execution

Code:

```
#include<iostream>
#include <winsock.h>
#include<windows.h>
#include<mysql.h>
#include<omp.h>
```

```
using namespace std;
```

```
void psql(int i,string txt)
{
    int qstate;
    MYSQL* conn;
```

```

conn=mysql_init(0);
conn=mysql_real_connect(conn,"localhost","root","", "pdc",0,NULL,0);
MYSQL_ROW row;
MYSQL_RES* res;
if(conn)
{

    if(i==1)
    {

        std::string word = txt;
        std::string tmp = "select count(PNAME) from table1 where PNAME=\"\" + word+\"\"";

        qstate=mysql_query(conn,tmp.c_str());
        if(!qstate)
        {
            res=mysql_store_result(conn);

            while(row=mysql_fetch_row(res))
            {

                cout<<"Count from table1 is "<<row[0]<<endl;

            }
        }

        else if(i==2)
        {

            std::string word = txt;
            std::string tmp = "select count(PNAME) from table2 where PNAME=\"\" + word+\"\"";

            qstate=mysql_query(conn,tmp.c_str());
            if(!qstate)
            {
                res=mysql_store_result(conn);

                while(row=mysql_fetch_row(res))
                {

                    cout<<"Count from table2 is "<<row[0]<<endl;
                }
            }
        }
    }
}

```

```

    }

    else if(i==3)
    {

std::string word = txt;
std::string tmp = "select count(PNAME) from table3 where PNAME=\'" + word+"\';

qstate=mysql_query(conn,tmp.c_str());
if(!qstate)
{
    res=mysql_store_result(conn);

    while(row=mysql_fetch_row(res))
    {

        cout<<"Count from table3 is "<<row[0]<<endl;
    }
}

}

    else if(i==4)
    {

std::string word = txt;
std::string tmp = "select count(PNAME) from table4 where PNAME=\'" + word+"\';

qstate=mysql_query(conn,tmp.c_str());
if(!qstate)
{
    res=mysql_store_result(conn);

    while(row=mysql_fetch_row(res))
    {

        cout<<"Count from table4 is "<<row[0]<<endl;
    }
}

}

    else if(i==5)
    {

std::string word = txt;
std::string tmp = "select count(PNAME) from table5 where PNAME=\'" + word+"\';

qstate=mysql_query(conn,tmp.c_str());
if(!qstate)

```

```

{
    res=mysql_store_result(conn);

    while(row=mysql_fetch_row(res))
    {

        cout<<"Count from table5 is "<<row[0]<<endl;

    }
}

}

else{
    cout<<"end"<<endl;
}

}
else
{
    cout<<"Not connected"<<endl;
}

}

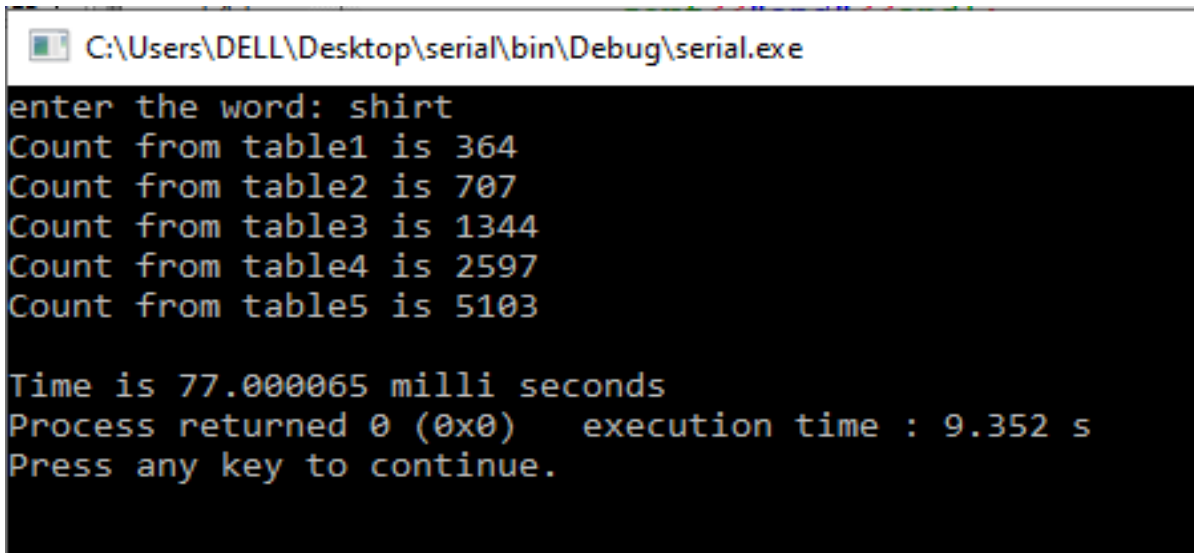
int main()
{

    int i=0;
    string txt;
    cout<<"enter the word: ";
    cin>>txt;
    double time = omp_get_wtime();
    for(i=1;i<6;i++)
    {

        psql(i,txt);
    }
    time = (omp_get_wtime() - time)*1000;
    cout << "\nTime is " + to_string(time)+" milli seconds";

    return 0;
}

```



```
C:\Users\DELL\Desktop\serial\bin\Debug\serial.exe
enter the word: shirt
Count from table1 is 364
Count from table2 is 707
Count from table3 is 1344
Count from table4 is 2597
Count from table5 is 5103

Time is 77.000065 milli seconds
Process returned 0 (0x0)   execution time : 9.352 s
Press any key to continue.
```

2) Parallel Execution

Code:

```
#include<iostream>
#include <winsock.h>
#include<windows.h>
#include<mysql.h>
#include<omp.h>

using namespace std;

string psql(int i,string txt)
{
    int qstate;
    MYSQL* conn;
    conn=mysql_init(0);
    conn=mysql_real_connect(conn,"localhost","root","","pdc",0,NULL,0);
    MYSQL_ROW row;
    MYSQL_RES* res;
    if(conn)
    {

        std::string word = txt;
        std::string iter = to_string(i);

        std::string tmp = "select count(PNAME) from table"+iter+" where PNAME='"+word+"'";

        qstate=mysql_query(conn,tmp.c_str());
        if(!qstate)
        {
            res=mysql_store_result(conn);

            while(row=mysql_fetch_row(res))
```



```

        {
            cout<<"\nCount from Table "<<i<< " is "<<row[0]<<endl;

        }
    }

}
else
{
    cout<<"Not connected"<<endl;

}

}

int main()
{

    int i=0;
    string txt;
    cout<<"enter the word: ";
    cin>>txt;
    cout<<" ";
    double time = omp_get_wtime();
    #pragma omp parallel
    for(i=1;i<6;i++)
    {
        psql(i,txt);

    }

    time = (omp_get_wtime() - time)*1000;
    cout << "\nTime is " + to_string(time)+" milli seconds";

    return 0;
}

```

```
C:\Users\DELL\Desktop\normal_parallel\bin\Debug\normal_parallel.exe
enter the word: shirt
Count from Table
Count from Table 11 is 364
is 364
Count from Table 1 is 364
Count from Table 1 is 364
Count from Table 2 is 707
Count from Table 3 is 1344
Count from Table 4 is 2597
Count from Table 5 is 5103
Time is 77.999973 milli seconds
Process returned 0 (0x0)   execution time : 7.200 s
Press any key to continue.
```

3) Optimized Parallel Execution

Code:

```
#include<iostream>
#include <winsock.h>
#include<windows.h>
#include<mysql.h>
#include<omp.h>

using namespace std;

void psql(int i,string txt)
{
    int qstate;
    MYSQL* conn;
    conn=mysql_init(0);
    conn=mysql_real_connect(conn,"localhost","root","","pdc",0,NULL,0);
    MYSQL_ROW row;
    MYSQL_RES* res;
    if(conn)
    {

        std::string word = txt;
        std::string iter = to_string(i);
```

```

std::string tmp = "select count(PNAME) from table"+iter+" where PNAME=\""+word+"\"";

qstate=mysql_query(conn,tmp.c_str());
if(!qstate)
{
    res=mysql_store_result(conn);
    //cout<<"Email id"<<" "<<"name"<<" "<<"combo"<<endl;
    while(row=mysql_fetch_row(res))
    {
        //cout<<row[0]<<" "<<row[1]<<" "<<row[2]<<endl;
        cout<<"\nCount from Table "<<i<<" is "<<row[0]<<endl;

    }
}

}
else
{
    cout<<"Not connected"<<endl;

}

}

int main()
{

    int i=0;
    string txt;
    cout<<"enter the word: ";
    cin>>txt;
    cout<<" ";
    double time = omp_get_wtime();
    #pragma omp parallel
    #pragma omp for
    for(i=1;i<6;i++)
    {

        psql(i,txt);
    }
    time = (omp_get_wtime() - time)*1000;
    cout << "\nTime is " + to_string(time)+" milli seconds";

    return 0;
}

```

C:\Users\DELL\Desktop\less_parallel\bin\Debug\less_parallel.exe

enter the word: shirt

Count from Table 1 is 364

Count from Table 2 is 707

Count from Table 3 is 1344

Count from Table 4 is 2597

Count from Table 5 is 5103

Time is 48.999912 milli seconds

Process returned 0 (0x0) execution time : 6.905 s

Press any key to continue.

REFERENCES

- 1) P. Xu, X. Tang, W. Wang, H. Jin and L. T. Yang, "Fast and Parallel Keyword Search Over Public-Key Ciphertexts for Cloud-Assisted IoT," in *IEEE Access*, vol. 5, pp. 24775-24784, 2017, doi: 10.1109/ACCESS.2017.2771301.
- 2) Drews, F., Lichtenberg, J. & Welch, L. Scalable parallel word search in multicore/multiprocessor systems. *J Supercomput* 51, 58–75 (2010). <https://doi.org/10.1007/s11227-009-0308-3>
- 3) Sinan Sameer Mahmood Al-Dabbagh, Nawaf Hazim Barnouti, Mustafa Abdul Sahib Naser, Zaid G. Ali. Parallel Quick Search Algorithm for the Exact String Matching Problem Using OpenMP (2016). DOI: 10.410.4236/jcc.2016.413001236/jcc.2016.413001
- 4) Oleksij Volkv, Simona Ramanauskaite. Research of word search algorithm based on relational database (2013) ISSN 1648-8776.
- 5) P. Kacsuk and N. Podhorszki, "Dataflow parallel database systems and LOGFLOW," *Proceedings of the Sixth Euromicro Workshop on Parallel and Distributed Processing - PDP '98* -, Madrid, Spain, 1998, pp. 382-388, doi: 10.1109/EMPDP.1998.647223.
- 6) Kao B., Garcia-Molina H. (1994) An Overview of Real-Time Database Systems. In: Halang W.A., Stoyenko A.D. (eds) *Real Time Computing*. NATO ASI Series (Series F: Computer and Systems Sciences), vol 127. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-88049-0_13
- 7) David J. DeWitt , Jim Gray , *Parallel Database Systems: The Future of High Performance Database Processing* (1992)
- 8) Sebeopou, Z., Magoutis, K., Marazakis, M., & Bilas, A. (2008). A Comparative Experimental Study of Parallel File Systems for Large-Scale Data Processing. *LASCO*.
- 9) S. A. Wright et al., "Parallel File System Analysis Through Application I/O Tracing," in *The Computer Journal*, vol. 56, no. 2, pp. 141-155, Feb. 2013, doi: 10.1093/comjnl/bxs044.
- 10) Gerard Salton, Chris Buckley (1988). *Parallel Text Search Methods*. DOI: 10.1145/42372.42380

