PARALLEL WORD SEARCH USING OPENMP

Under the guidance of

Professor NARAYANAMOORTHI M

Submitted by

Raghav Jindal 18BCE2080

Prepared For

CSE4001 - PARALLEL AND DISTRIBUTED COMPUTING

VIT®

Vellore Institute of Technology

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

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 parallel 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 3 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 parallel 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: Open MP, Parallelization, Multithreaded, Word search, Sequential.

The word frequency effect refers to the remark that high-frequency words are managed more efficiently than low-frequency words. Although the effect was first talked about over eighty years ago, in recent years it has been understood and studied in more detail. It has become obvious that substantial quality alterations exist between frequency approximations and that we need a new uniform frequency measure that does not misinform users. Research also points to steady individual differences in the word frequency effect, denoting that the result will be present at different word frequency ranges for people with different grades of language acquaintance. Finally, a few ongoing expansions point to the importance of semantic diversity rather than mere differences in the number of times words have been encountered and to the importance of taking into account word commonness in addition to word frequency. Frequency tells us about the relevance of the term and those terms can be used for greater user coverage and attraction.This is a technique used by websites to become an important page on searches. The more frequently used words are put into the content and there are more hits on the page because of this. So creating a project for increasing its efficiency gives good future scope.

Our project focuses on creating a parallel word search mechanism from a dictionary of words. Parallelisation if such a task is helpful in reducing the complexity of the task. Each task is subdivided into smaller tasks and are computed in parallel on different cores. Algorithms are applied to make these implementations. TAU is a performance tool which will analyse why the parallel implementation is better and on what scale it is better or not based on predefined criteria.

The packages we use for the performance analysis are:

TAU Portable Profiling Package

The model that TAU uses to profile parallel, multi-threaded programs maintains performance data for each thread, context, and node in use by an application. The profiling instrumentation needed to implement the model captures data for C++ functions, methods, basic blocks, and statement execution at these levels. The instrumentation is complicated, however, by advanced features in the C++ language, such as templates and namespaces. All C++ language features are supported in the TAU profiling instrumentation, which is available through an API at the library or application level. The API also provides selection of profiling groups for organizing and controlling instrumentation. ACTS software layers have been instrumented and support for thread profiling has been recently added.

From the profile data collected, TAU's profile analysis procedures can generate a wealth of performance information for the user. It can show the exclusive and inclusive time spent in each function with nanosecond resolution. For templated entities, it shows the breakup of time spent for each instantiation. Other data includes how many times each function was called, how many profiled functions did each function invoke, and what the mean inclusive time per call was. Time information can also be displayed relative to nodes, contexts, and threads. Instead of time, hardware performance data can be shown. Also, user-level profiling is possible.

TAU's profile visualization tool, paraprof, provides graphical displays of all the performance analysis results, in aggregate and per node/context/thread form. The user can quickly identify sources of performance bottlenecks in the application using the graphical interface. In addition, TAU can generate event traces that can be displayed with the Vampir trace visualization tool.

TAU Code Analysis Package

The TAU static analysis tools are based on PDT, a Fortran, and C/C++ code analysis system built using a combination of Edison Design Group's C++ front end, edgcpfe, which produces an intermediate language (IL) representation, an IL converter tool, taucpdisp, which generates a more descriptive C++ program database (PDB), and a C++ library, ductape, for processing and merging PDB files, among other interesting things. Other parsers included in PDT are GNU gfortran, Cleanscape Flint Fortran, and Mutek Fortran 90 parser. With these tools, TAU provides tools that support sophisticated views of program structure, incorporating the latest C++ language features such as templates, namespaces, and exceptions. Currently, the code analysis systems have been used to analyze C++ source to automatically generate TAU profiling instrumentation.

PURPOSE

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.

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 (multicore) 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.

PROBLEM STATEMENT

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.

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,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 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.

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++

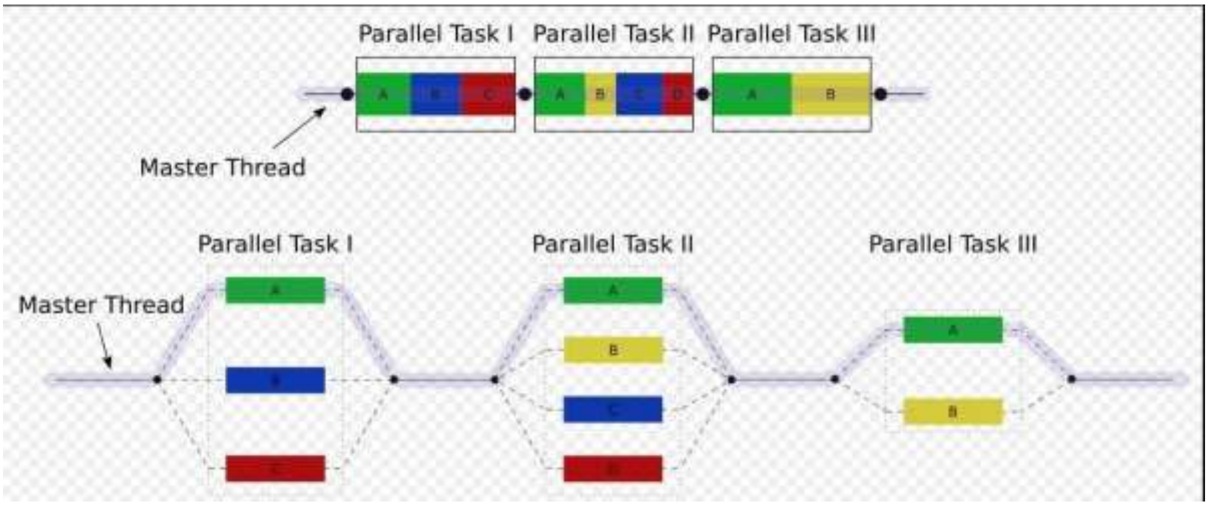


Fig 1:Fork Join model

DESIGN ARCHITECTURE

3.1 Sequential model:

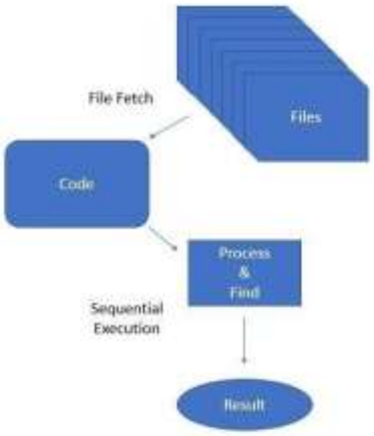


Fig 2:Sequential model

3.2 Parallel model:

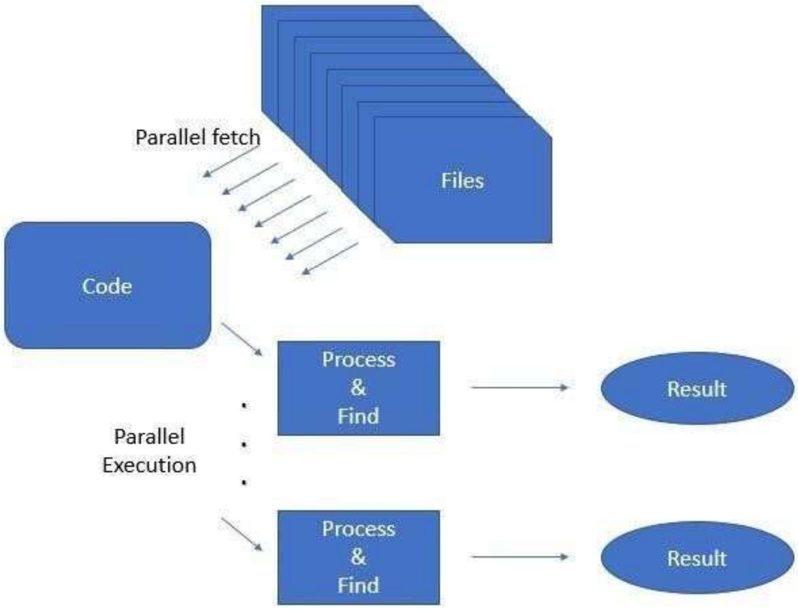


Fig 3:Parallel model

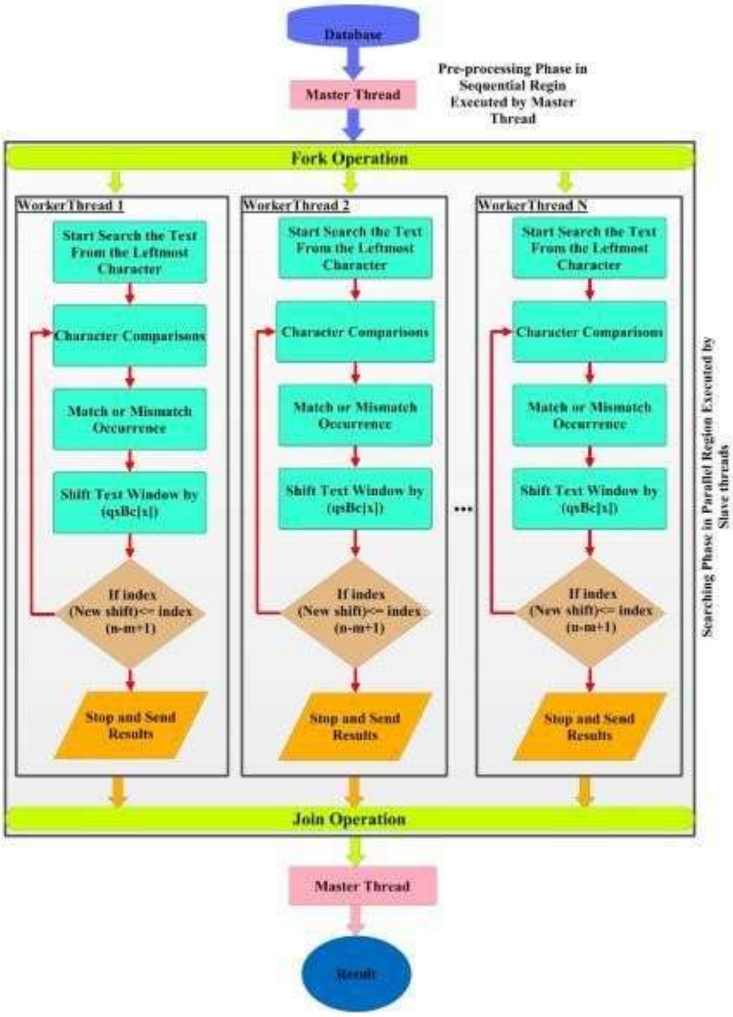
SYSTEM IMPLEMENTATION

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.

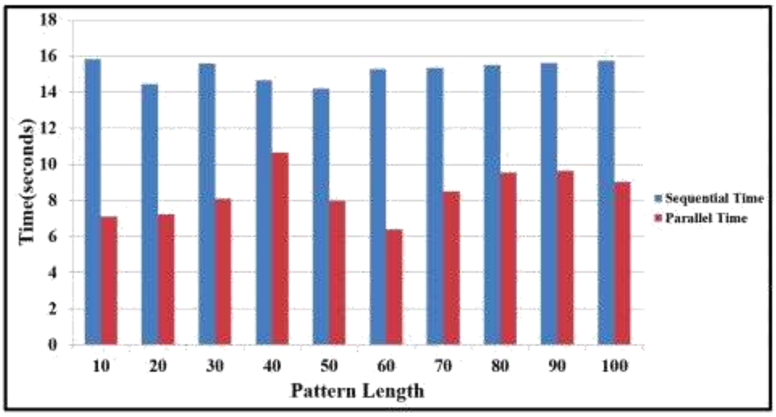
LITERATURE SURVEY

In this paper “Parallel Quick Search Algorithm for Exact String Matching Problem” AlDabbagh,S.S.M.,Barnouti,N.H Naser,M.A.S and Ali,Z.G, have used the Quick Search string matching algorithm which is implemented in the multi-core environment using OpenMP. The use of OpenMP is to lessen the execution time of the entire program. The data types like Proteins, English text and DNA are used to scrutinize the out come of parallelization and implementation of the algorithm on multi-core environment. The main idea of parallelising the algorithm is to divide one task into subtasks that are computed parallely or say simultaneously on many CPUs. Here, even the sub-task is further divided into instructions. Finally only one program is carried out at one particular time.


Database
Master Thread
Pre-processing Phase in Sequential Regin Executed by Master Thread
Fork Operation
WorkerThread 1
Start Search the Text From the Leftmost Character
Character Comparisons
Match or Mismatch Occurrence
Shift Text Window by (qsBc[x])
If index (New shift)<= index (n-m+1)
Stop and Send Results
WorkerThread 2
Start Search the Text From the Leftmost Character
Character Comparisons
Match or Mismatch Occurrence
Shift Text Window by (qsBc[x])
If index (New shift)<= index (n-m+1)
Stop and Send Results
...
WorkerThread N
Start Search the Text From the Leftmost Character
Character Comparisons
Match or Mismatch Occurrence
Shift Text Window by (qsBc[x])
If index (New shift)<= index (n-m+1)
Stop and Send Results
Searching Phase in Parallel Region Executed by Slave threads
Join Operation
Master Thread
Result

The Quick Search algorithm shows irregular behavior in comparison to data types of DNA and proteins, due to the alphabet size. The English text has 100 characters or more. The instability of behavior is seen in the pattern length of 40 and 60, which gives the worst time and best time respectively. Thus, the average time of parallel program shows better performance compared to the sequential program

The execution time of the sequential and parallel of Quick Search algorithm using English text data type.



For future work the parallel Quick Search algorithm could be enhanced by parallelizing the preprocessing phase with the searching phase.

The paper "Scalable Parallel Word Search in Multi-Core/Multiprocessor Systems " written by F. Drews1 , E. Petri , J. Lichtenberg , and L. Welch

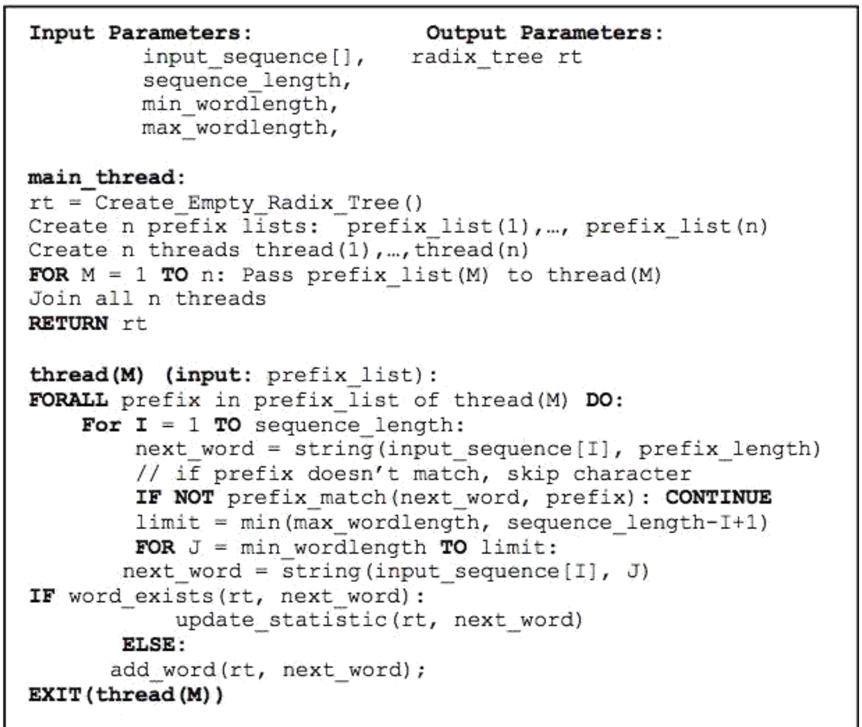
Electrical Engineering and Computer Science, Ohio University, Athens, OH, USA talks about a parallel algorithm for fast word searching. The aim of this paper is to ascertain an input sequence of DNA which is a set of biological words which works best for larger inputs in multiprocessor/multicore systems. The paper aims at determining words in genomic data.

This paper tries to overcome the following drawbacks:

(i) to inflict a high degree of cache locality which tends to showcase non-local access patterns

(ii) to reduce the need for data access locking

(iii) to allow an even distribution of the overall processing load among multiple threads.



```
Input Parameters:                 Output Parameters:
         input_sequence[],    radix_tree rt
         sequence_length,
         min_wordlength,
         max_wordlength,

main_thread:
rt = Create_Empty_Radix_Tree()
Create n prefix lists:  prefix_list(1),…, prefix_list(n)
Create n threads thread(1),…,thread(n)
FOR M = 1 TO n: Pass prefix_list(M) to thread(M)
Join all n threads
RETURN rt

thread(M) (input: prefix_list):
FORALL prefix in prefix_list of thread(M) DO:
    For I = 1 TO sequence_length:
        next_word = string(input_sequence[I], prefix_length)
        // if prefix doesn't match, skip character
        IF NOT prefix_match(next_word, prefix): CONTINUE
        limit = min(max_wordlength, sequence_length-I+1)
        FOR J = min_wordlength TO limit:
       next_word = string(input_sequence[I], J)
IF word_exists(rt, next_word):
           update_statistic(rt, next_word)
       ELSE:
      add_word(rt, next_word);
EXIT(thread(M))
```

Figure 4: Parallelized Word Search Algorithm (PWSA)

Here, the implementation and performance evaluation of the proposed algorithm on the DNA sequence for the human chromosome 7 on a dual processor quad-core system with a total of 8 cores is done. The word search can be divided into different phases:

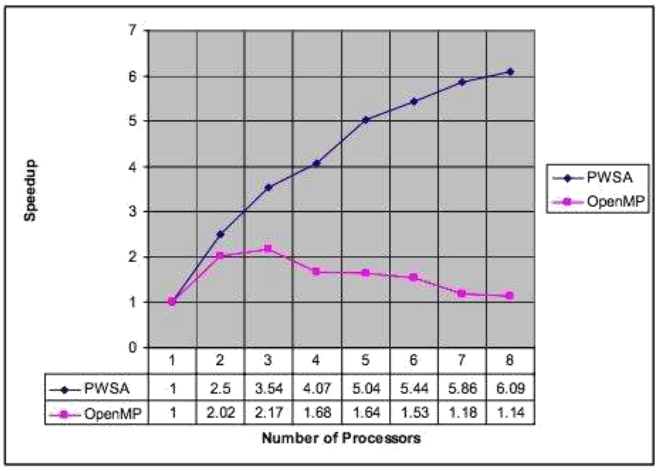
(1) Specify input sequence that need to be searched.

(2) Give parameters about the min and max length of words

(3) Generate some internal representation (hash tables to tree representations) . Along with that, statistical information regarding the repeat is stored

(4) Perform “Word Scoring”.

(5) The “Word Selection” associate words as trivial or non-trivial by examining the nucleotide organization of each word.



	1	2	3	4	5	6	7	8
PWSA	1	2.5	3.54	4.07	5.04	5.44	5.86	6.09
OpenMP	1	2.02	2.17	1.68	1.64	1.53	1.18	1.14

Figure 8: OpenMP Implementation vs. PWSA;
input size = 10,000,000;
max. word length = 50

Hence, we understood from the above result that the algorithm that they proposed here works amazing for large input sequences on a multiprocessor/multicore machine

The paper Alan Morris et al which is **"Supporting Nested OpenMp parallelism in the TAU performance system"** states that OpenMP allows nested parallel codes which a parallel codes can be present inside a code segment which is itself parallel creating a thread of parallel codes at once.To know the overall performance of this parallelism TAU performance tool is used which checks the speed and accuracy of each section. This helps us to determine the overall time complexity of the system which maybe a necessary criteria in a lot of evaluations and applications.It uses tracing and profiling of the data. For profiling it uses statistics,atomic profiles,entry and exit profiles ,mapping,profile I/O,sampling profiles. All these profilings are done by separate interfaces for each of them.Trace includes trace buffer,record creation,trace I/O,timestamps generation,trace filtering. The instrumentation is on different levels like Virtual machine bases,multi level and selective.If TAU is being used for flat profiling, performance measurements are kept for interval events only. We can also use call path profiling based on the code gravity and even phase profiling which can be put

when there are phases of a code in the execution or the cod e is divided into phased segments.

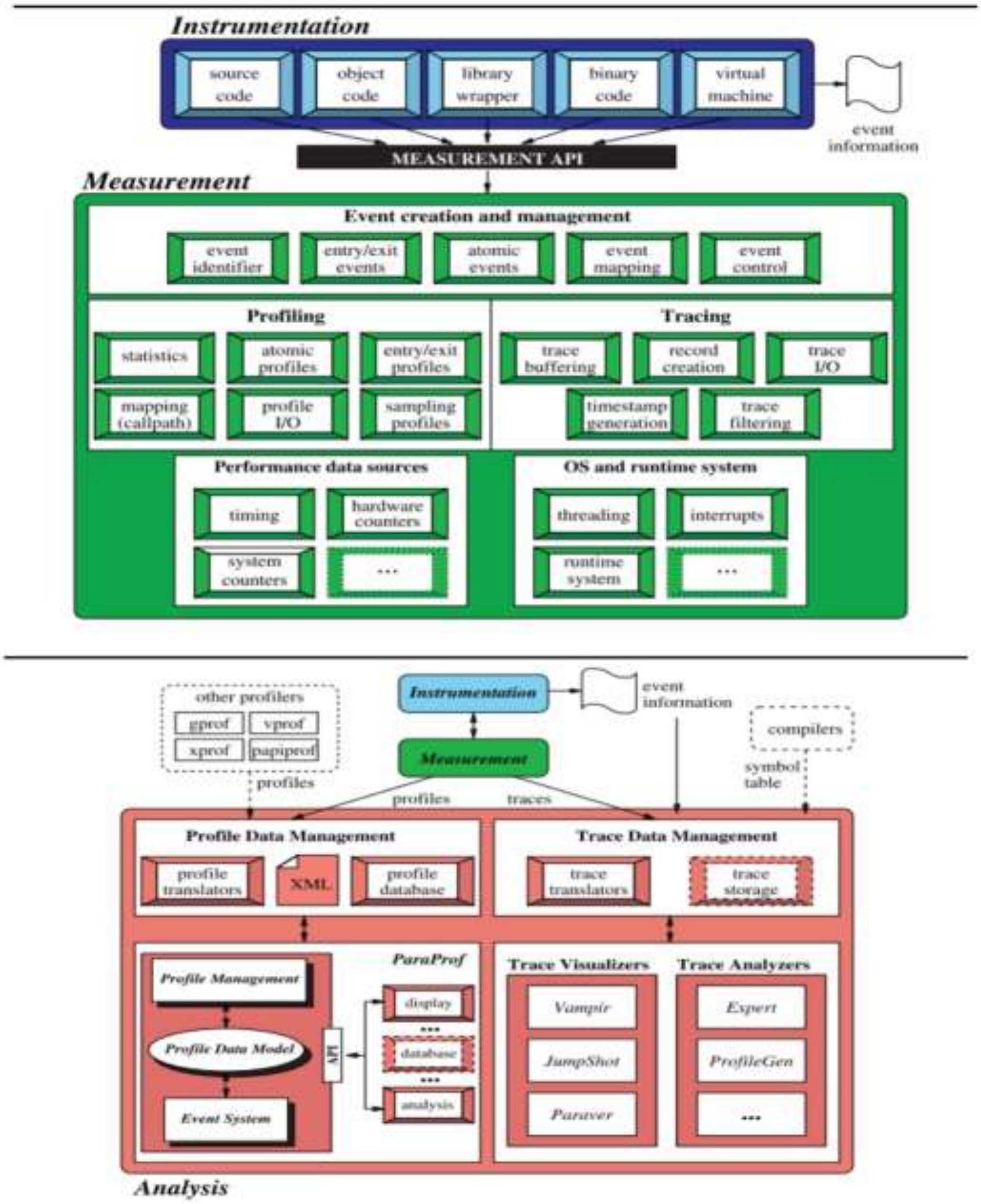


Fig. 3 Architecture of TAU Performance System – Analysis and Visualization.

For a normal searching for small text files, or strings.We could preprocess the data if memory is not an issue. After preprocessing, we could make string representation of the text file and search in row major order or vice or versa. But these basic algorithms have horrible time complexities ranging from O(4n) if we are searching from row wise and column wise.Time complexity would be O(8n) if we add the diagonals for searching as well.

For searching large text files or dictionaries, we need more advanced algorithm such as Boyer-Moore,Knuth-Morris-Pratt(KMP), and Robin-Karp. For our project in word search, we are going to be using KMP algorithm since time complexity of KMP is only O(n) in the worst case. Microsoft word uses Boyer-Moore searches the strings from the back of the words and has a complexity of O(mn) in the worst case. KMP is relatively easy to understand and implement that boyer-moore and preprocesses the data relatively faster than the other two algorithms. KMP doesn't work that well in cases of many matching characters followed by mismatching ones but, it also uses degenerating property(patterns having the same sub patterns appearing in the pattern) to hold the time complexity to O(n).

```
#include <bits/stdc++.h>


void computeLPSArray(char* pat, int M, int* lps);


// Prints occurrences of txt[] in pat[]

void KMPSearch(char* pat, char* txt)

{

int M = strlen(pat);

int N = strlen(txt);


        // create lps[] that will hold the longest prefix suffix

        // values for pattern int lps[M];
// Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lps);


        int i = 0; // index for txt[]           int j = 0; // index for pat[]          while (i <
N) {

        if (pat[j] == txt[i]) {          j++;     i++;

}


        if (j == M) {

        printf("Found pattern at index %d ", i - j);

        j = lps[j - 1];

}


// mismatch after j matches

        else if (i < N && pat[j] != txt[i]) {

        // Do not match lps[0..lps[j-1]] characters,     // they will match anyway
        if (j != 0)

        j = lps[j - 1];

        else

        i = i + 1;

        }

        }

}


// Fills lps[] for given patttern pat[0..M          -1] void computeLPSArray(char*
pat, int M, int* lps)

{
```

```
// length of the previous longest prefix suffix
int len = 0;

lps[0] = 0; // lps[0] is always 0

// the loop calculates lps[i] for i = 1 to M-1
int i = 1;
    while (i < M) { if (pat[i] == pat[len]) {
len++; lps[i] =
len; i++;
    }
    else // (pat[i] != pat[len])
    {
    // This is tricky. Consider the example.
    // AAACAAAA and i = 7. The idea is similar    // to search step.
    if (len != 0) {
len = lps[len - 1];

    // Also, note that we do not increment
    // i here
    }
    else // if (len == 0)
    {
    lps[i] = 0;
i++;
    }
```

```
        }
        }
}

// Driver program to test above function int main()
{
char txt[] = "ABABDABACDABABCABAB";
char pat[] = "ABABCABAB";
KMPSearch(pat, txt);        return 0;
}
```

Reference:

“KMP Algorithm for Pattern Searching.   ” GeeksforGeeks, May 20, 2019. preprocessing is an entegral part of the KMP algorithm as it makes the file easier to search. Preprocessing is done in KMP to know the number of characters to be skipped. To know the no. of characters to be skipped, we preprocess data in string format in pat[] and assign an integer array lps[] to count how many characters to skip. lps is abbreviation for longest proper prefix which is also suffix. the document is searched in sub-patterns using the lps focusing on sub-strings having patterns of prefix or suffix.

Example of preprocessing:

For the pattern “AAAA”,

lps[] is [0, 1, 2, 3]

For the pattern “ABCDE”,

lps[] is [0, 0, 0, 0, 0]

For the pattern “AABAACAABAA”,

lps[] is [0, 1, 0, 1, 2, 0, 1, 2, 3, 4, 5]

Searching algorithm for KMP is:

1. Compare of pat[i] with i=0 with characters of text.
2. we continuously match txt[j] and pat[i] and increment i,j till they keep matching.
3. When we see a mismatch: -

-We know that characters pat[0..i-1] match with txt[j-i…j-1].

-lps[i-1] is count of characters of pat[0…j-1] that are both proper prefix and suffix.

From above two points, we can conclude that we do not need to match these lps[i-1] characters with txt[j-i…j-1] because we know that these characters will anyway match.

TAU PERFORMANCE ANALYSIS TOOL

TAU is a performance analysis tool for analysing the overall functionality and complexity of the programs in C or C++. It has the power of gathering performance data through instrumentation of functions, methods, basic blocks, and statements. All C++ language features are supported including templates and namespaces.We would like to include this tool to analyse our parallel code and the serial code and show a comparitive analysis between the two and then would hopefully conclude that the parallel code is much more efficient.

Data Analysis and The Visualisation of the data

- Profiling the data: TAU's profile visualization tool, ParaProf, gives us a wide range of graphical output and displays for profiling the data available to help users to rapidly identify sources of performance bottlenecks. The word based pprof tool is also provided for analyzing the profiled data.
- Tracing the data: TAU gives us the JumpShot trace visualization tool for graphical visualisation of the tracing data.TAU also gives us with a lot of utilities to convert the tracing data into formats for viewing with Vampir, Paraver and other performance analysis tools.

Programming models and platforms available: TAU supports most commonly used parallel hardware and programming models, including Intel, Cray, IBM, Sun, Apple, SGI, GPUs/Accelerators, HP, NEC, Fujitsu, MS Windows, using MPI, OpenMP, Pthreads, OpenCL, CUDA and Hybrid.

What do we need from the analysis to draw a conclusion?

The tool can show graphical representation of data for better understanding on one look.

1.Profiling tells you how much time is needed for a particular task.

2.Tracing tells you which part exists in which segment of the code on a timeline

Instrumentation which would be direct and indirect performance analysis. This invokes performance analysis.Direct measuring is with probes whereas indirect measuring is from periodic sampling of the data.

User defined events are interval,atomic and measurement events which have inclusive and exclusive and duration where inclusive checks the internal loops whereas exclusive checks for all the external loops which have integrated internal loops. Each loop will take an amount of time depending on the conditions and the number of executions.

Types of Performance Profiles

Flatprofiles •Metric (e.g., time) spent in an event

•Exclusive/inclusive, # of calls, child calls, ...

Callpathprofiles •Time spent along a calling path (edges in callgraph) •“main=> f1 => f2 => MPI_Send”

•Set the TAU_CALLPATH and TAU_CALLPATH_DEPTHenvironment variables

Callsiteprofiles •Time spent along in an event at a given source location •

Set the TAU_CALLSITE environment variable Phaseprofiles

•Flat profiles under a phase (nested phases allowed) •Default “main” phase
•Supports static or dynamic (e.g. per-iteration) phases
http://tau.uoregon.edu/tau_lln

TAU can be used to trace actions during a program's execution. Unlike profiling, which combines the time taken in each routine, loop, etc, tracing allows you to see the events as they related to each other in terms of a timeline. One can work at tracing however, is that for tracing files quickly which can grow to be very large, which makes tracing problematic or intolerable for long running, many process jobs.

As with profiling, the most relaxed and rapid way to trace an application is to use the tau_exec command. It automatically gadgets your executables at run time, and needs no unusual gathering or alterations to source code. All that we need to do is make sure your TAU environment is setup properly.

1. Setup your TAU atmosphere by loading the TAU dotkit package. Also, make sure the TAU_TRACE environment variable is set to "1". If we want to postulate a almanac where the tracefiles should be written (default is the working directory), use the TRACEDIR environment variable.

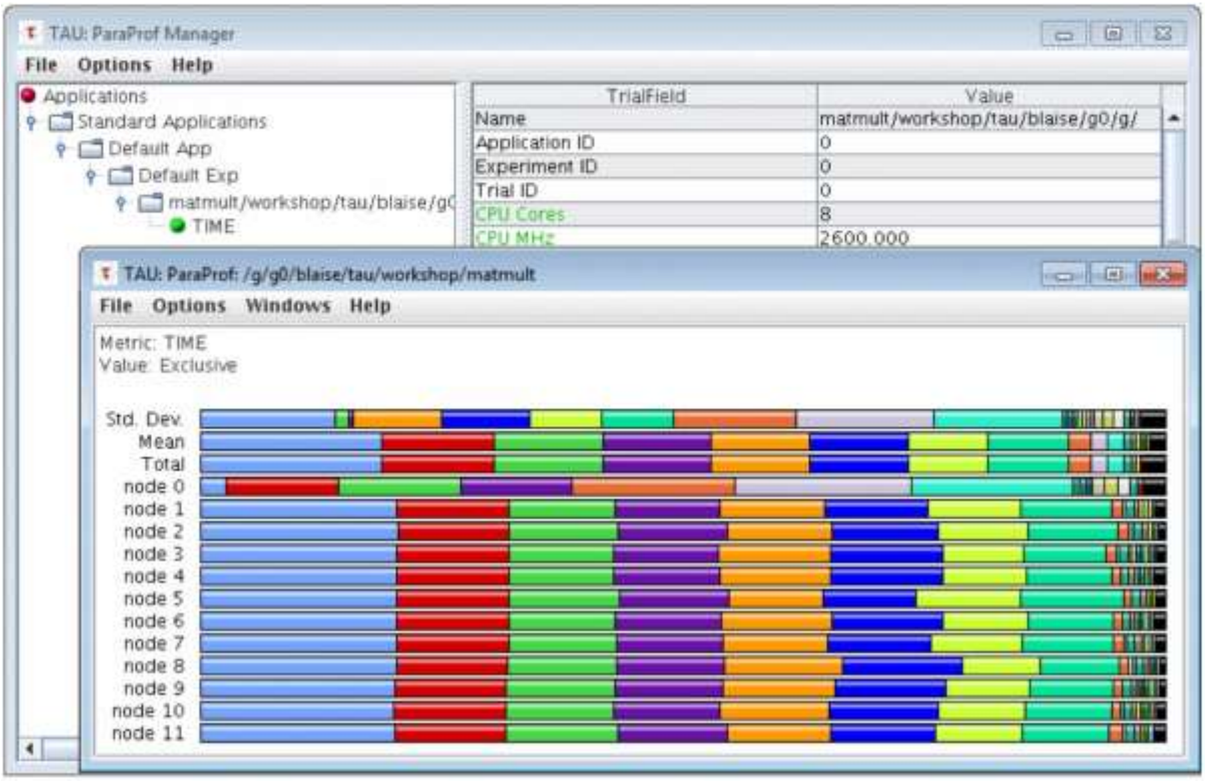
```
% use tau

Prepending: tau (ok)

% setenv TAU_TRACE 1

% setenv TRACEDIR /p/lscratche/joesmith/matmultTracefiles
```

Note: If you want to comprise TAU profiling at the same period as tracing, set the TAU_PROFILE environment variable to "1". By default, it is turned off when tracing.

2. Run your program using the tau_exec command. For example, launching a 64 task MPI job in the pdebug partition:

```
% srun -n64 -ppdebug tau_exec matmult
```

3. Following accomplishment of your job, you will have a two groups of files named tautrace.#.*.trc and events.#.edfwhere # denotes the MPI rank.

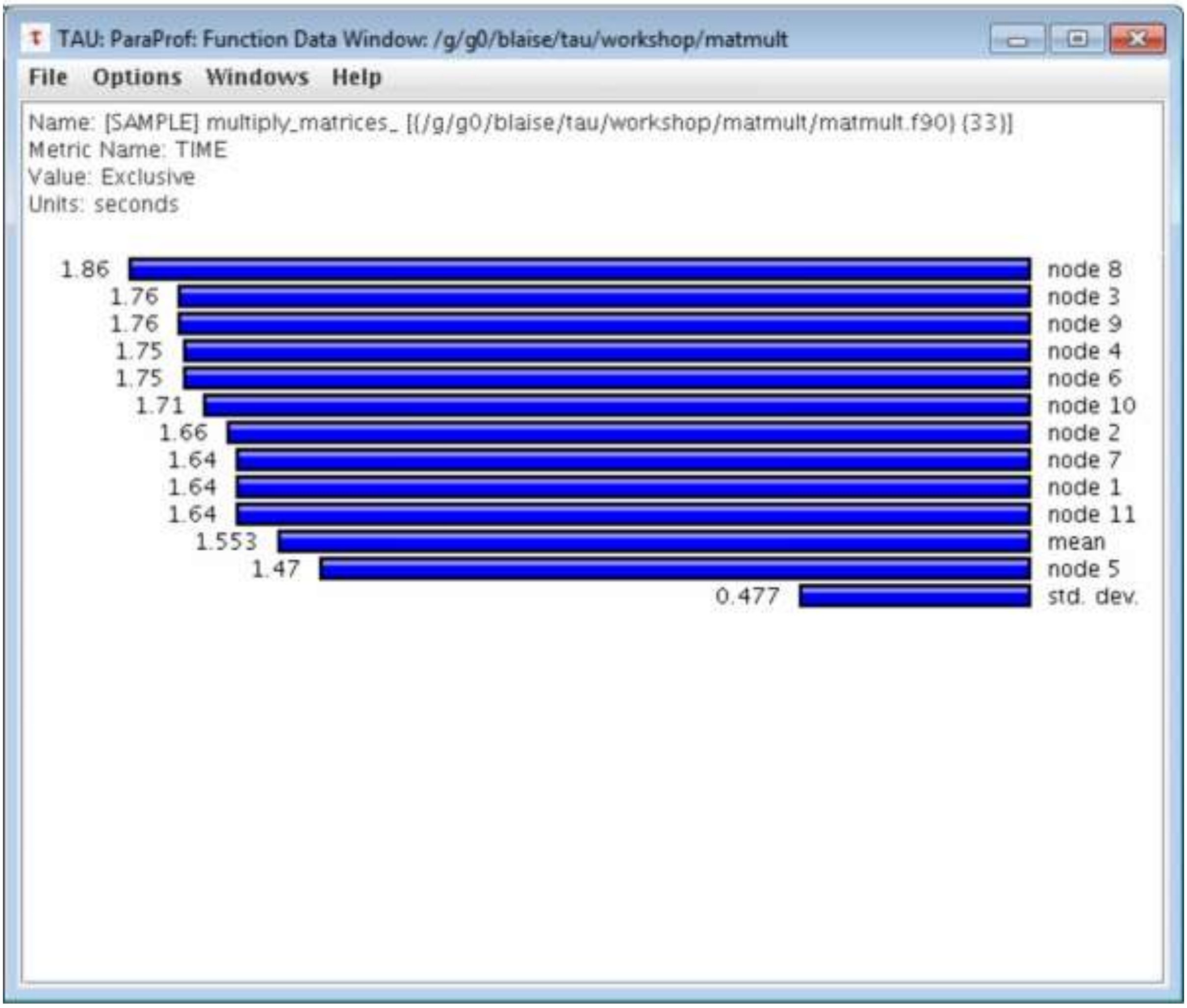


TAU: ParaProf Manager

TrialField	Value
Name	matmult/workshop/tau/blaise/g0/g/
Application ID	0
Experiment ID	0
Trial ID	0
CPU Cores	8
CPU MHz	2600.000

TAU: ParaProf: /g/g0/blaise/tau/workshop/matmult

Metric: TIME
Value: Exclusive



TAU: ParaProf: Function Data Window: /g/g0/blaise/tau/workshop/matmult

Name: [SAMPLE] multiply_matrices_ [{/g/g0/blaise/tau/workshop/matmult/matmult.f90} {33}]
Metric Name: TIME
Value: Exclusive
Units: seconds

FRONT END

For word searching algorithm display, it would be better if we have a front end connected to a database for the betterment of the users. This is mainly to give our project the platform or say a user interface so that the users can use the word search for various documents at one time.

The front end, is basically a website that we are making for a better user interface. The need for a front end is that, we have an algorithm but when the users want to use it, that is only possible when there is an interface/ platform for it. To display the output for the same, we need a front end. The same way, we also require a backend and a database to store the algorithm and the output generated from it.

Talking about the front end, we will be developing the UI with the help of HTML5 and CSS (Cascading style sheets) and JavaScript(if needed). The backend is Php (Hypertext Preprocessor) and the database is (MySQLi).

A few HTML tags that help us achieve this task are <head>, <a> - anchor tag, <legend>, <label>, <input>, <p>, <em>, <strong> etc.

The reason why it's better to use MySQLi is because it's an improved version of relational database management system (RDBMS) and it uses a structured query language. MySQL and MySQLi are php database extensions required to write php codes for accessing DBs. It's compatible with php. The main advantage of using MySQLi DB is because it provides Data security, gives high performance also its pretty flexible. MySQLi is nothing but an "improved" extension. It provides OOI (Object oriented Interface) and procedural both.

```
p {
font-family: verdana;
font-size: 20px;
}
```

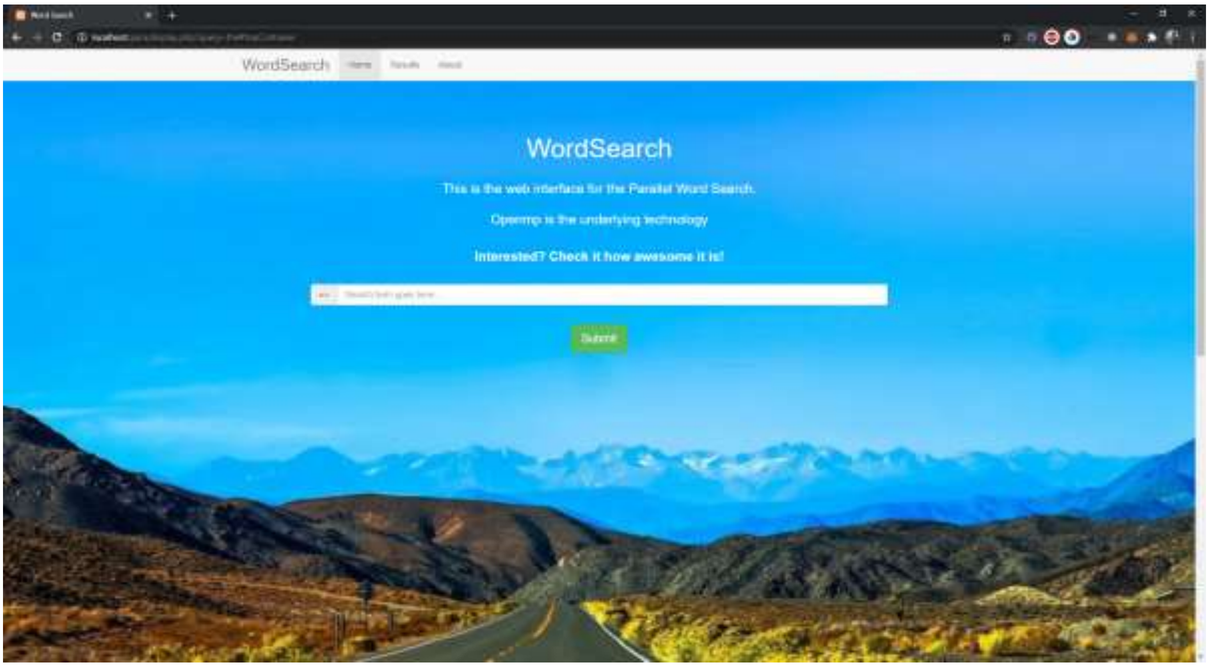
The sample code for connection of php is as follows:

```
<?php
$servername ="localhost";
$username ="username";
$password ="";
//Create connection
$conn = mysql_connect($servername);
//Check for connection
if†(!$conn)
{
die("Connection failed: ". mysql_connect_error());
}
echo"Connected successfully";
?>
```

How it works?

The above code is a sample for the connection of the database via PHP using MySQL. We design the frontend with HTML and CSS by making one file which includes the styling along with the input tags.

The user can actually browse through his files and upload in the drop box depending on however many files they want. There will be a search option where the user can type the word that needs to be searched and press “enter”. Then the algorithm runs and searches for the particular word in the given documents. The output is displayed on the front end and stored in the Database.

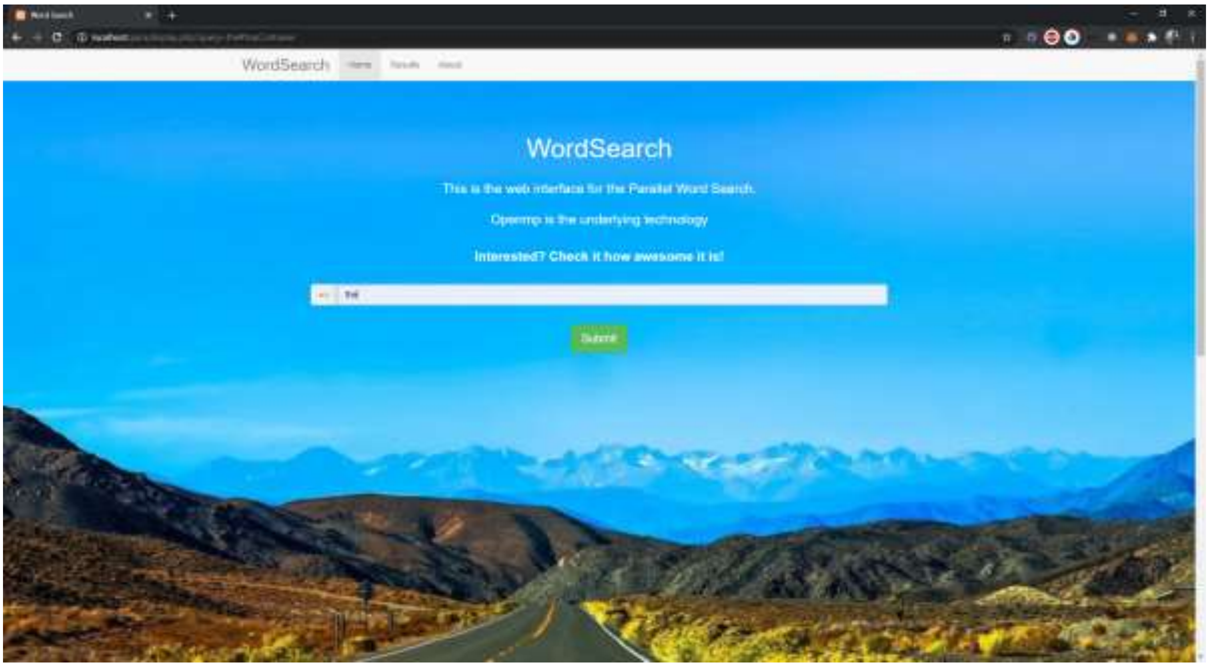


WordSearch

This is the web interface for the Parallel Word Search.

Openmp is the underlying technology

Interested? Check it how awesome it is!

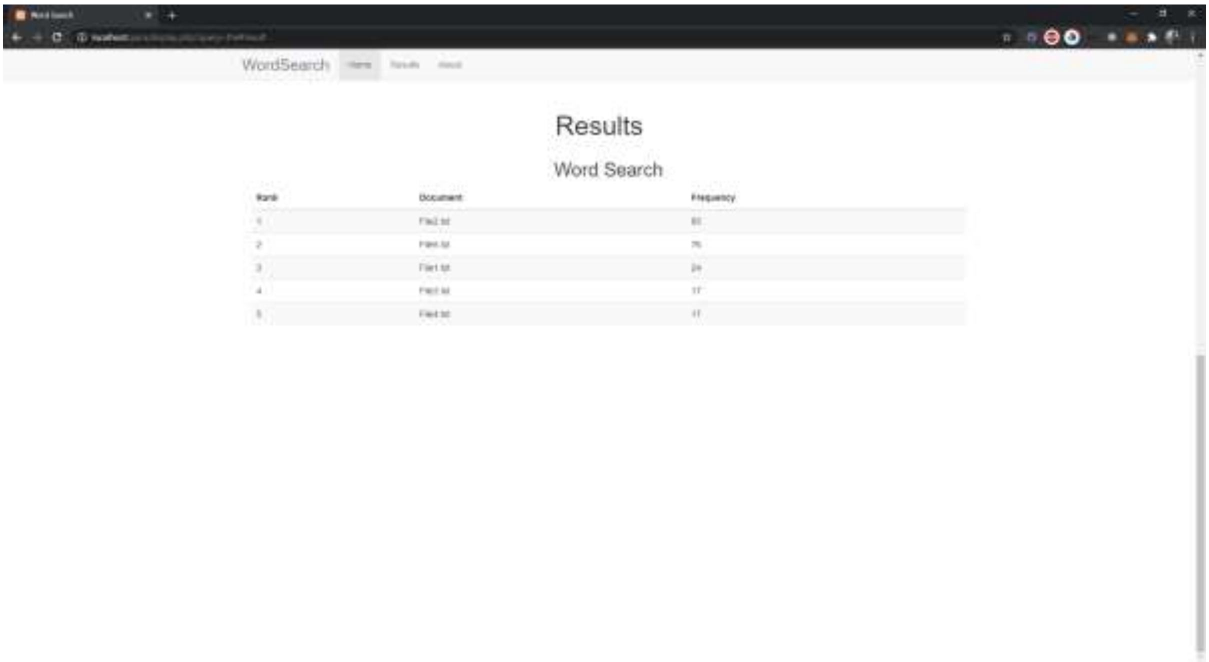


WordSearch

This is the web interface for the Parallel Word Search.

Openmp is the underlying technology

Interested? Check it how awesome it is!



Results

Word Search

Rank	Document	Frequency
1	File2.txt	83
2	File4.txt	75
3	File1.txt	24
4	File3.txt	17
5	File5.txt	17

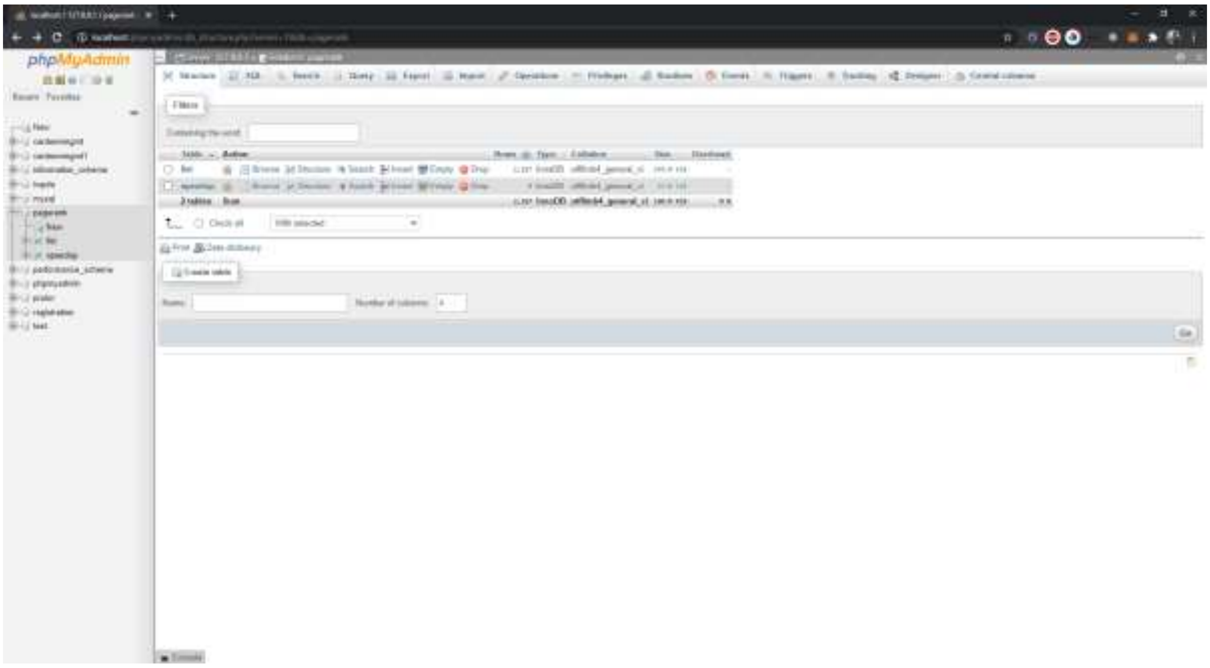
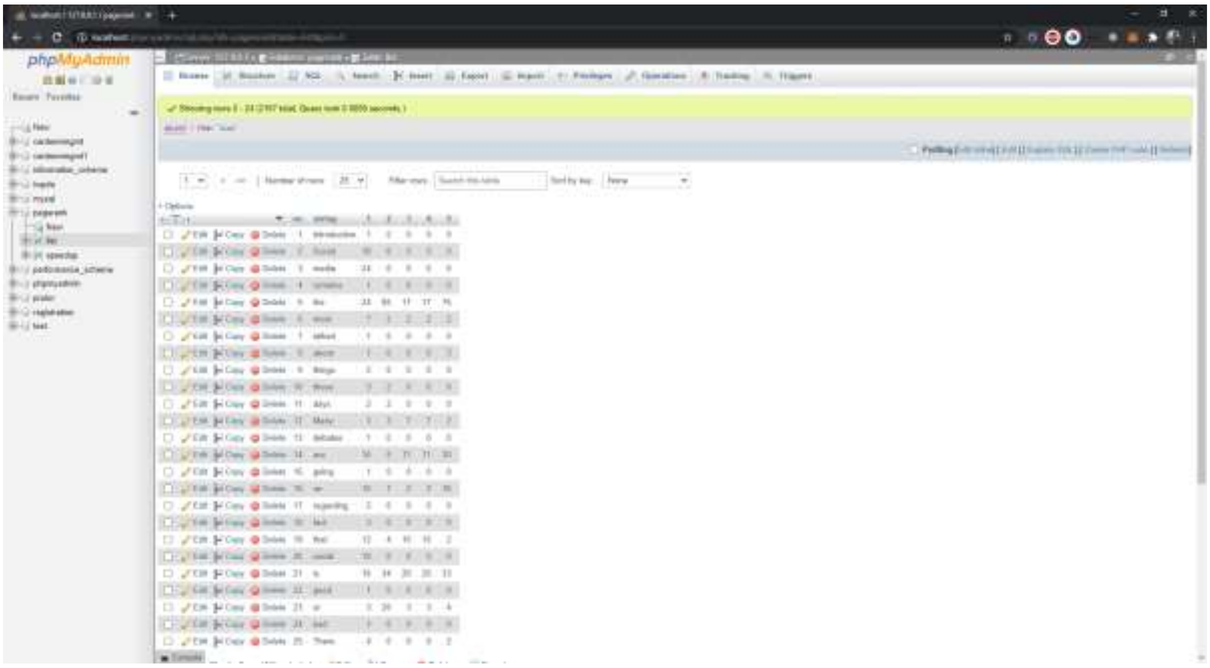


Table	Action	Rows	Type	Collation	Size	Overhead
[illegible]	Browse Structure Search Insert Empty Drop	[illegible]	InnoDB	utf8mb4_general_ci	[illegible]	
[illegible]	Browse Structure Search Insert Empty Drop	[illegible]	InnoDB	utf8mb4_general_ci	[illegible]	
2 tables	Sum	[illegible]	InnoDB	utf8mb4_general_ci	[illegible]	[illegible]



✓ Showing rows 0 - 24 (2197 total, Query took 0.0000 seconds.)

id	string	1	2	3	4	5
1	introduction	1	0	0	0	0
2	[illegible]	10	0	0	0	0
3	media	24	0	0	0	0
4	[illegible]	1	0	0	0	0
5	the	23	84	17	17	75
6	[illegible]	7	1	2	2	2
7	[illegible]	1	0	0	0	0
8	[illegible]	1	0	0	0	0
9	things	2	0	0	0	0
10	those	3	2	0	0	0
11	days	2	2	0	0	0
12	Many	1	1	7	7	2
13	[illegible]	1	0	0	0	0
14	are	34	9	71	71	30
15	going	1	0	0	0	0
16	we	10	1	2	7	15
17	regarding	2	0	0	0	0
18	[illegible]	3	0	0	0	0
19	that	12	4	10	10	2
20	social	10	0	0	0	0
21	is	16	14	20	20	11
22	good	1	0	0	0	0
23	or	3	20	3	3	4
24	bad	1	0	0	0	0
25	There	4	0	0	0	2

CODE IMPLEMENTATION :

PARALLEL CODE:

```
//Parallel processing Assignment

//to compile : g++ pagerank.cpp -std=c++0x

//parallel code

#include <omp.h>

#include <bits/stdc++.h>


using namespace std;

int main(int argc, char const *argv[])

{

  //int i;

  string str;

  // ofstream fo("/opt/lampp/htdocs/para/display.php");

  ofstream fo2("/opt/lampp/htdocs/para/insert.php");
```

```
      //omp_set_nested(1);
      //omp_set_dynamic(0);

string list[10000][6];
std::fill(list[0], list[0] + 10000 * 6, "0");
int count = 0;
int temp;

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

 double start_time = omp_get_wtime();
 #pragma omp parallel for private(temp)  num_threads(2)
   for(int j=1;j<=5;j++)
    {
      string temp3;
      temp3 = "File" + to_string(j) + ".txt";
      ifstream fi(temp3);   //"file" + temp2 + ".txt"

        while (fi >> str)
        {
         int flag=0;
         int i;
         int temp23;
         int flag2 = 1;
```

```
#pragma omp parallel for
  for (i = 0; i < count; ++i)
    {
      if(list[i][0]==str&&flag2==1)
      { //int b = atoi(a.c_str());
       flag=1;
       temp = atoi(list[i][j].c_str());//(int)list[i][j];
       temp++;

                     //fo2<<str<<"$var=" << temp;
         fo2<<"$query = \"UPDATE `list` SET `"<<j<<"`="<<temp<<" WHERE `string`='"<<str<<"';\";"<<endl;
         fo2<<"mysqli_query($con,$query);"<<endl;




             stringstream convert;


             convert << temp;


             list[i][j] = convert.str();
             flag2 = 0;
             //break;


      }
```

```
                     }




           if(flag==0)
             {
                               //count++;
                //#pragma omp critical
                   {
                     list[count][0] = str;
                     list[count][j] = '1';
                     count++;
                     if(j==1)
                     {
                                                                   fo2 <<
"$var=\""<<str<<"\";"<<"\n";
                                                                   fo2 << "$query = \"INSERT
INTO `list`(`string`,`"<<j<<"`,`2`,`3`,`4`,`5`)VALUES
('\".$var.\"',1,0,0,0,0);\";"<<endl;
                                                                   fo2 <<
"mysqli_query($con,$query);"<<endl;
                                                           }
                                                           else if(j==2)
                                                           {
                                                                   fo2 <<
"$var=\""<<str<<"\";"<<"\n";
```

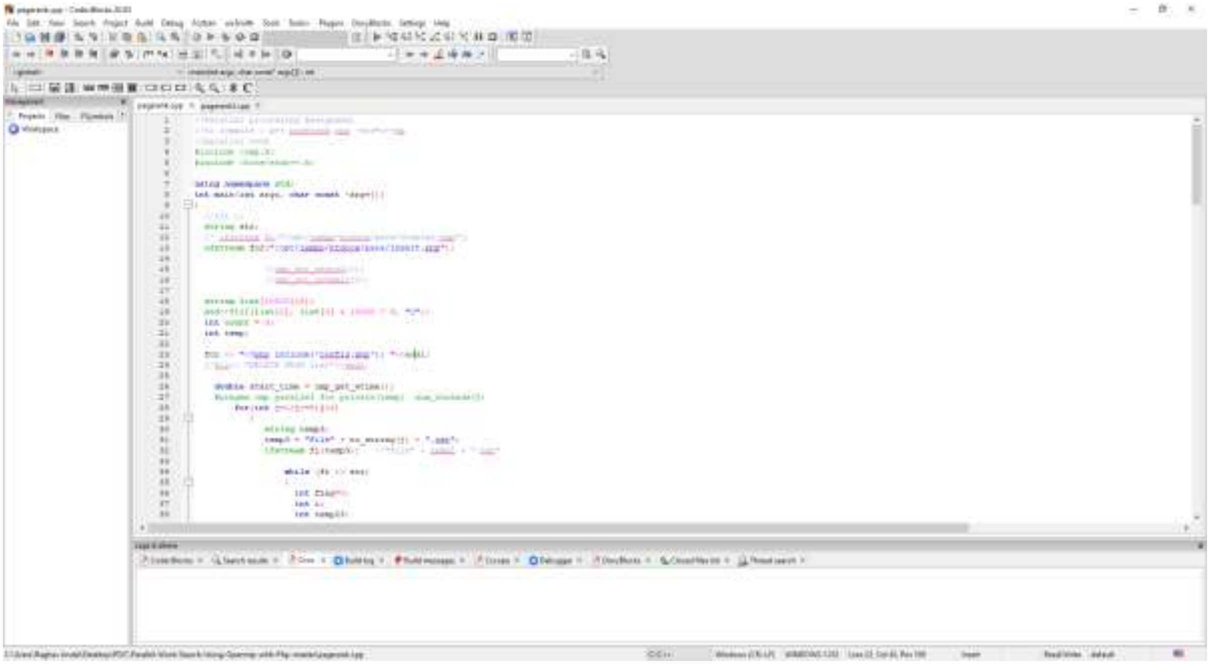
```
                                                  fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`"<<j<<"`,`3`,`4`,`5`)VALUES
('\".$var.\"',0,1,0,0,0);\";"<<endl;
                                                  fo2 <<
"mysqli_query($con,$query);"<<endl;
                                          }
                                          else if(j==3)
                                              {
                                                  fo2 <<
"$var=\""<<str<<"\";"<<"\n";
                                                  fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`2`,`"<<j<<"`,`4`,`5`)VALUES
('\".$var.\"',0,0,1,0,0);\";"<<endl;
                                                  fo2 <<
"mysqli_query($con,$query);"<<endl;
                                          }
                                          else if(j==4)
                                              {
                                                  fo2 <<
"$var=\""<<str<<"\";"<<"\n";
                                                  fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`2`,`3`,`"<<j<<"`,`5`)VALUES
('\".$var.\"',0,0,0,1,0);\";"<<endl;
                                                  fo2 <<
"mysqli_query($con,$query);"<<endl;
                                          }
                                          else
                                              {
                                                  fo2 <<
"$var=\""<<str<<"\";"<<"\n";
```

```
                                                                    fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`2`,`3`,`4`,`"<<j<<"`)VALUES
('\".$var.\"',0,0,0,0,1);\";"<<endl;

                                                                    fo2 <<
"mysqli_query($con,$query);"<<endl;

                                                        }



                            }




                      }






   }

  }









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





  /*for (int i = 0; i < count; ++i)

  {


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

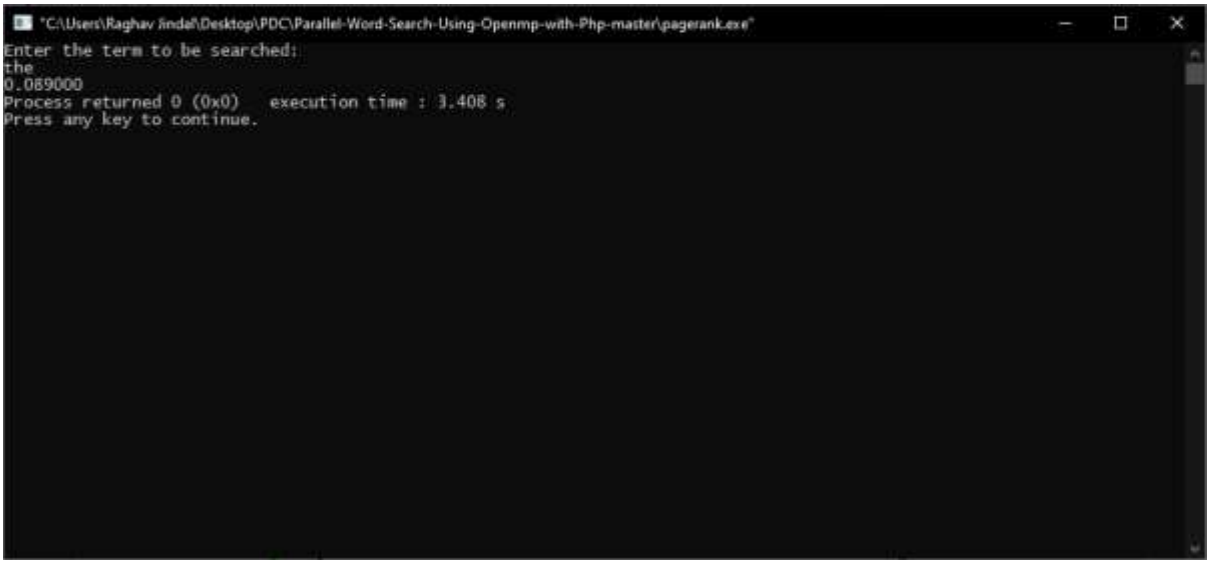
```
    for (int j = 0; j < 4; ++j)
    {
      cout<<left<<setw(5)<<list[i][j]<<"    ";
    }
    cout<<endl;
  }*/
 //cout<<count;

string str1;
double time = omp_get_wtime() - start_time;

cout<<"Enter the term to be searched:"<<endl;
cin>>str1;
int flag1=0;
int result;

//#pragma omp parallel for
for (int i = 0; i < count; ++i)
{
  if(list[i][0] == str1)
  {
    flag1=1;
    result = i;
    break;
  }
}
```

```
if(flag1==0)
  cout<<"No Results Found!!!"<<endl;

else
{

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

  for (int i = 0; i < 5; ++i)
  {
    v.push_back(make_pair(atoi(list[result][i+1].c_str()),i+1));

      //atoi(list[result][i+1].c_str())
  }
  sort(v.rbegin(),v.rend());
  int index=1;
  //cout<<"Rank"<<"\t"<<"Document"<<"\t"<<"frequency"<<endl;
for(it=v.begin();it!=v.end();it++)
{
  //cout<<"frequency=>"<<it->first << "  " <<" doc.ID=>"<<it->second<<endl;
  //cout<<index++<<"=> "<<setw(10)<<"file"<<it->second<<".text"<<setw(10)<<it->first<<endl;
}
```

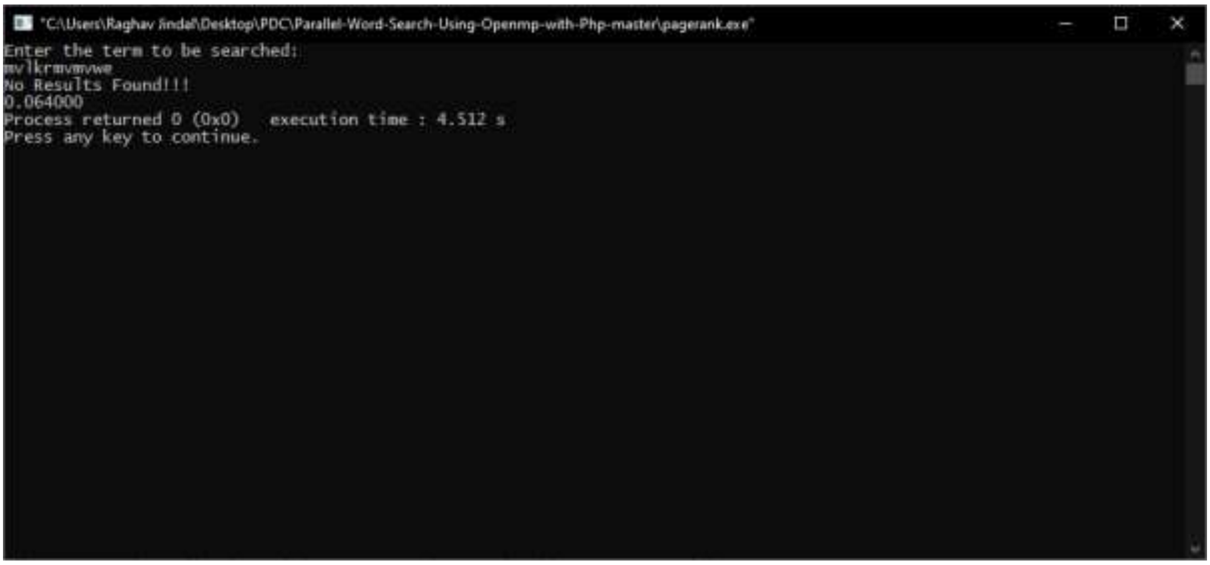
```
  v.clear();


}
//list[count][i+1]





  printf("%lf",time);
  return 0;
}
```



This is the execution of the parallel code


"C:\Users\Raghav Jindal\Desktop\PDC\Parallel-Word-Search-Using-Openmp-with-Php-master\pagerank.exe"

```
Enter the term to be searched:
the
0.089000
Process returned 0 (0x0)   execution time : 3.408 s
Press any key to continue.
```


"C:\Users\Raghav Jindal\Desktop\PDC\Parallel-Word-Search-Using-Openmp-with-Php-master\pagerank.exe"

```
Enter the term to be searched:
mvlkrmvmvwe
No Results Found!!!
0.064000
Process returned 0 (0x0)   execution time : 4.512 s
Press any key to continue.
```

SERIAL CODE:

```
//Parallel processing Assignment
//to compile : g++ pagerank.cpp -std=c++0x
//serial code
#include <omp.h>
#include <bits/stdc++.h>
using namespace std;
int main(int argc, char const *argv[])
{
    //int i;
    string str;
    //ofstream fo("/opt/lampp/htdocs/para/display.php");
    ofstream fo2("/opt/lampp/htdocs/para/insert.php");



    string list[10000][6];
    std::fill(list[0], list[0] + 10000 * 6, "0");
    int count = 0;
    int temp;

    //fo << "hi";

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

double start_time = omp_get_wtime();
    for(int j=5;j<=5;j++)
    {
          string temp3;
          temp3 = "File" + to_string(j) + ".txt";
          ifstream fi(temp3);   //"file" + temp2 + ".txt"

          while (fi >> str)
          {
            int flag=0;

            int i;

            int temp23;

        int flag2 = 1;

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

            {

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

                {       //int b = atoi(a.c_str());

                        flag=1;

                        temp = atoi(list[i][j].c_str());//(int)list[i][j];

                        temp++;


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

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

                        fo2<<"mysqli_query($con,$query);"<<endl;






                        stringstream convert;


                        convert << temp;


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

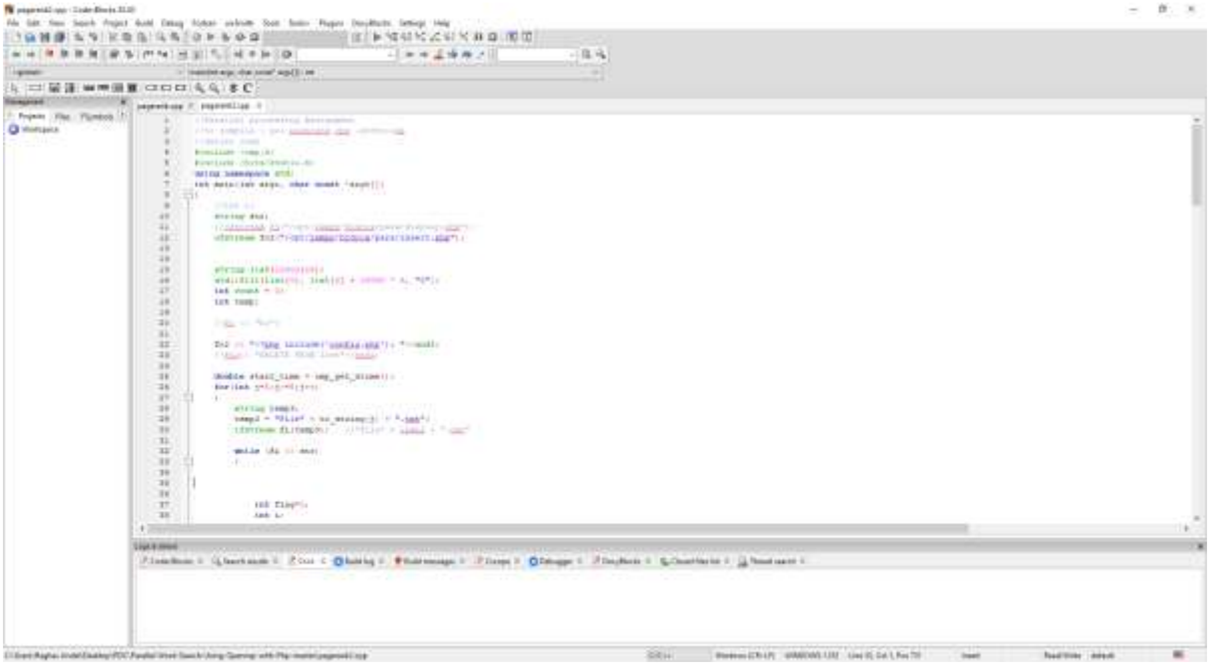
```
              flag2 = 0;

                }



            }



            if(flag==0)
                {       //count++;

          list[count][0] = str;
          list[count][j] = '1';
          count++;
          if(j==1)
          {
                                fo2 << "$var=\""<<str<<"\";"<<"\n";
                                fo2 << "$query = \"INSERT INTO `list`(`string`,`"<<j<<"`,`2`,`3`,`4`,`5`)VALUES ('\".$var.\"',1,0,0,0,0);\";"<<endl;
                                fo2 << "mysqli_query($con,$query);"<<endl;
                            }
                            else if(j==2)
                            {
                                fo2 << "$var=\""<<str<<"\";"<<"\n";
```

```
                                        fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`"<<j<<"`,`3`,`4`,`5`)VALUES
('\".$var.\"',0,1,0,0,0);\";"<<endl;
                                        fo2 <<
"mysqli_query($con,$query);"<<endl;
                                }
                                else if(j==3)
                                    {
                                        fo2 <<
"$var=\""<<str<<"\";"<<"\n";
                                        fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`2`,`"<<j<<"`,`4`,`5`)VALUES
('\".$var.\"',0,0,1,0,0);\";"<<endl;
                                        fo2 <<
"mysqli_query($con,$query);"<<endl;
                                }
                                else if(j==4)
                                    {
                                        fo2 <<
"$var=\""<<str<<"\";"<<"\n";
                                        fo2 << "$query = \"INSERT
INTO `list`(`string`,`1`,`2`,`3`,`"<<j<<"`,`5`)VALUES
('\".$var.\"',0,0,0,1,0);\";"<<endl;
                                        fo2 <<
"mysqli_query($con,$query);"<<endl;
                                }
                                else
                                    {
                                        fo2 <<
"$var=\""<<str<<"\";"<<"\n";
```

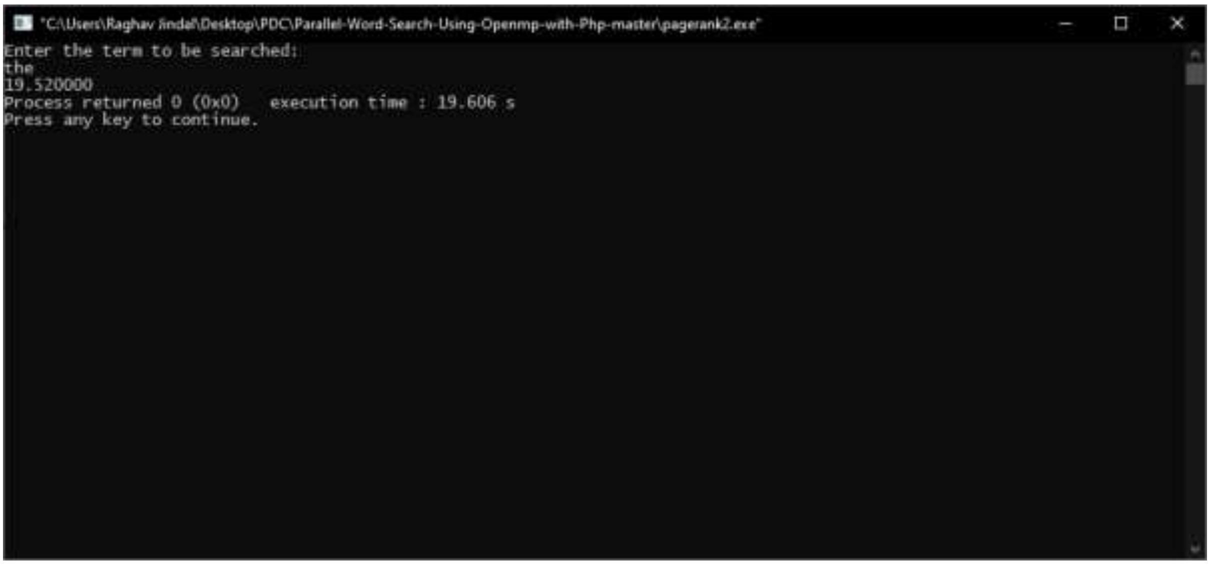
```
                                fo2 << "$query = \"INSERT INTO `list`(`string`,`1`,`2`,`3`,`4`,`"<<j<<"`)VALUES ('\".$var.\"',0,0,0,0,1);\";"<<endl;

                                fo2 << "mysqli_query($con,$query);"<<endl;

                            }



        }



}



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



    /*for (int i = 0; i < count; ++i)

    {


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

        for (int j = 0; j < 4; ++j)

        {

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

        }

        cout<<endl;

    }*/
```

```
 //cout<<count;

string str1;


cout<<"Enter the term to be searched:"<<endl;
cin>>str1;
int flag1=0;
int result;

for (int i = 0; i < count; ++i)
{
        if(list[i][0] == str1)
        {
                flag1=1;
                result = i;
                //break;
        }
}

if(flag1==0)
        cout<<"No Results Found!!!"<<endl;

else
{
	vector< pair<int, int> > v;
	vector< pair<int, int> >::iterator it;

	for (int i = 0; i < 5; ++i)
	{
		v.push_back(make_pair(atoi(list[result][i+1].c_str()),i+1));

		  //atoi(list[result][i+1].c_str())
	}
	sort(v.rbegin(),v.rend());
	int index=1;
	//cout<<"Rank"<<"\t"<<"Document"<<"\t"<<"frequency"<<endl;
for(it=v.begin();it!=v.end();it++)
{
	//cout<<"frequency=>"<<it->first << "  " <<" doc.ID=>"<<it->second<<endl;
	//cout<<index++<<"=> "<<setw(10)<<"file"<<it->second<<".text"<<setw(10)<<it->first<<endl;
}

	v.clear();

}
//list[count][i+1]

double time = omp_get_wtime() - start_time;
```

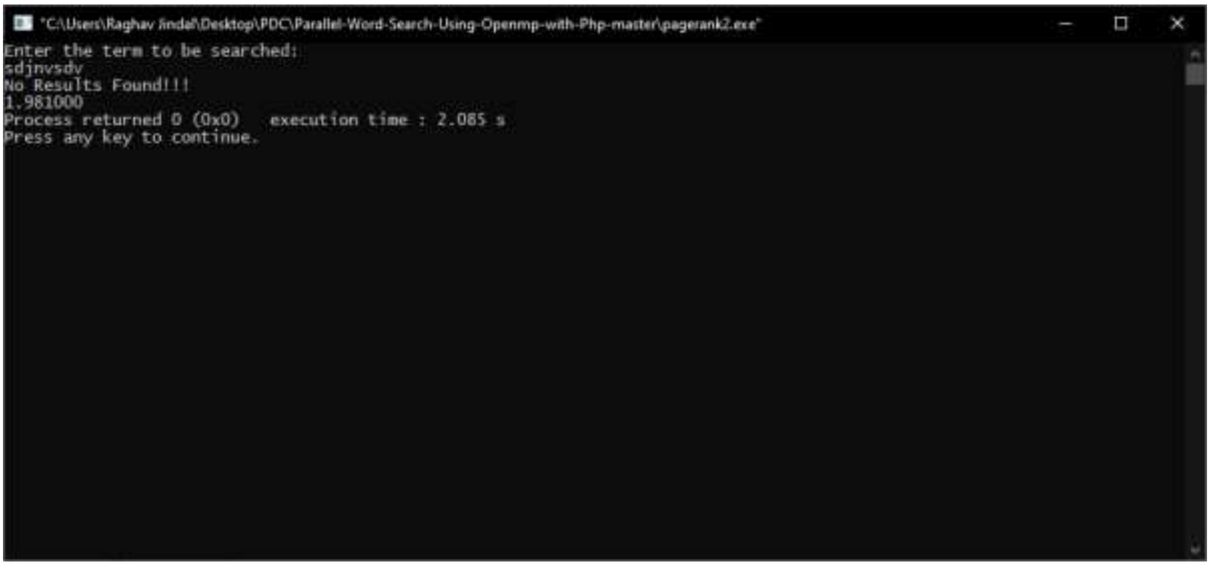
```
    printf("%lf",time);


        return 0;
}
}
```



This serial implementation uses openmp only to find the time of execution and not for parallel implementation.


"C:\Users\Raghav Jindal\Desktop\PDC\Parallel-Word-Search-Using-Openmp-with-Php-master\pagerank2.exe"

```
Enter the term to be searched:
the
19.520000
Process returned 0 (0x0)   execution time : 19.606 s
Press any key to continue.
```


"C:\Users\Raghav Jindal\Desktop\PDC\Parallel-Word-Search-Using-Openmp-with-Php-master\pagerank2.exe"

```
Enter the term to be searched:
sdjnvsdv
No Results Found!!!
1.981000
Process returned 0 (0x0)   execution time : 2.085 s
Press any key to continue.
```

TESTING AND RESULT ANALYSIS

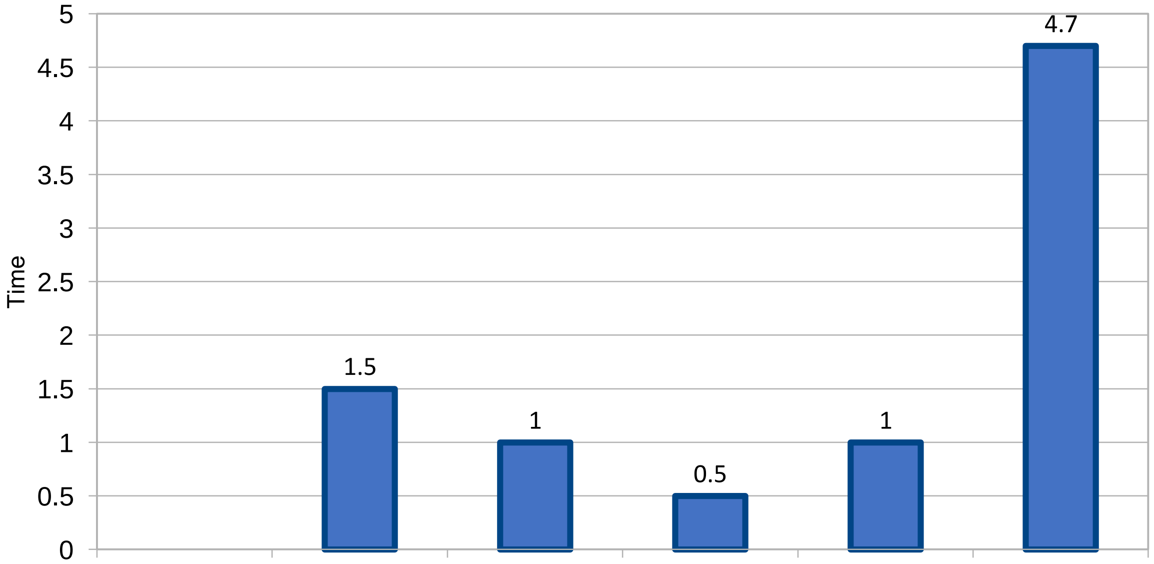
Test case Number	**Test case Id**	**Test case**	**Expected Result**	**Actual Result**	**Pass / Fail**
1.	1.1	Sequential code: Reading 5 files to search for a word	The count of each word in each file	The count of each word in each file	PASS
	1.2	Sequential code: Reading 5 files to search for a statement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

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.

NOVELTY OF THE PROPOSED WORK

Typically a computer scientist will divide a composite task into many parts with a software tool and allocate each part to a processor, then each processor will resolve its part, and the data is collected by a software tool to read the solution or perform the task.

Typically each processor will work normally and will perform executions in parallel as trained, getting data from the computer’s memory. Processors will also rely on software to communicate and interact with one another so that they can stay in sync relating to changes in data points. Assuming all the processors remain in sync with each another, at the end of a task, software will fit all the data parts together.

Computers that do not have multiple processors can still be used in parallel processing if they are networked together to form a cluster.

Parallel computing is of importance in every field and can be seen in day to day instances. Talking about the future of the word searches it is more than obvious that this is used in many technologies. A simple search on google opens up pages which have the keywords typed by you which implies that such a big organization also uses this basic algorithm for their implementation of work.

Just like our project searches for a word in a number of documents websites like google also search for the keyword searched by the user in the webpages available to them. It can only be anticipated that parallelization in something so basic is going to fetch faster results and will be even more satisfactory for the users.

Word searches is also applied in the field of data sciences where you can visualise a chart with all the important words of a particular topic. The visualisation is referred as WORDCLOUD. Word searches allow us to find these relevant words that are to be used in the representation of the data. Tracking the data of the words that are searched most frequently can be used in the system as well. Parallelization of the both the processes would give faster and better results.

Word searches help in create recognition and understanding of the topic. An application of parallel word search can be in the field of education.

Young students many a times like doing crosswords and puzzles where they have to find words thus creating a system where the program searches for the most relevant words for the young students knowledge and education at that level and then inputs it into the crossowords and puzzles online will provide a more efficient and all rounded way of learning for the kids. The student will now have more knowledge and infact that knowledge will be relevant .

Word searches can be extended to frequency searches which is used in data mining or text mining to find which pages or text on the internet are more relevant than other pages available and then they are presented sequentially. Imagine increasing the efficiency of this system on the internet and getting better results by simple parallelization.

RESULTS AND DISCUSSIONS

We implemented a basic word search algorithm with parallel processing using C++ on openmp and what we have concluded from the following is :

- Parallelization of word search increases the efficiency of the word search and makes it faster and some cases helps in increasing the relevance.
- Parallel programming goes beyond the limits imposed by sequential computing, which is often constrained by physical and practical factors that limit the ability to construct faster sequential algorithms.
- The word searches can also be applied further for word frequency search in parallel which is used in many different fields for gathering relevant information.

- We studied the different charts and graphs shown by the performance tool and it can be concluded on various different graphical as well as data representations that this algorithm works better when it is parallelized.

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.