Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X
IMPACT FACTOR: 5.258

IJCSMC, Vol. 5, Issue. 6, June 2016, pg.372 – 382

Parallel Quicksort Algorithm using OpenMP

Sinan Sameer Mahmood Al-Dabbagh[1], Nawaf Hazim Barnouti[2]

[1]Software Engineering and Information Technology Department, Al-Mansour University College, Baghdad, Iraq

[2] Al-Mansour University College, Baghdad, Iraq

[1] sinan.aldabbagh815@gmail.com, [2] nawafhazim1987@gmail.com

Abstract— In this paper we aims to parallelization the Quicksort algorithm using multithreading (OpenMP) platform. The proposed method examined on two standard dataset (File 1: Hamlet.txt 180 KB and File 2: Moby Dick.txt 1.18 MB) with different number of threads. The fundamental idea of the proposed algorithm is to creating many additional temporary sub-arrays according to a number of characters in each word, the sizes of each one of these sub-arrays are adopted based on a number of elements with the exact same number of characters in the input array. The elements of the input datasets is distributing into these temporary sub-arrays depending on the number of characters in each word. As a conclusion, the experimental results of this study reveal that the performance of parallelization the proposed Quicksort algorithm has shown improvement when compared to the sequential Quicksort algorithm by delivering improved Execution Time, Speedup and Efficiency.

Keywords— Quicksort, Sorting Algorithms, Parallelism, OpenMP, Parallel Sorting Algorithm, Parallel Computing, Multi-Core, Speedup, Parallel Programing

I. INTRODUCTION

One of the most critical problem in computer science is the sorting process, for that reason many sorting algorithms have been developed, such as Quicksort, Merge sort, Bubble sort, Insertion sort and Selection sort...etc. [1]. In computer science sorting algorithm is an algorithm that arranges the components of a list in a specific order. Sorting algorithms are taught in some fields such as Computer Science and Mathematics. There are numerous sorting algorithms applied in the field of computer science. They vary in their performance, functionality, resource usage and applications [2]. Parallel sorting algorithms also have been investigated to improve the sorting operation. Many parallel sorting algorithms such as column sort [3], parallel radix sort [4], bitonic sort [5], sample sort [6], and parallel merge sort [7] had been produced. Parallel sorts generally need a substitute of a fixed number of data between merging process and processing elements.

Hoare's Quicksort algorithm, It is one of the most intensively studied problems in computer science. It utilizes the "divide and conquers" strategy by reducing the sorting problem into several easier sorting problems and solve each of them [8]. Due to the good performance in practise the Quicksort algorithm considered one of the most popular sorting algorithm. The fundamental algorithm can be briefly identified as follows [8, 9, and 10]

One value is selected form the input data which normally called the pivot value, this pivot use to partitioning the input dataset into two subsets that one contains input data smaller in size compared to the pivot value and the

other contains input data higher than the pivot value. In every single step these divided datasets are sub-divided selecting pivots from each set. The recursive operation is no longer occur when there is no sub division is possible. It employs the "divide and conquers" technique by minimizing the sorting problem into several simpler sorting problems and solve each of them [11]. Figure 1 shows the recursive steps of Quicksort.

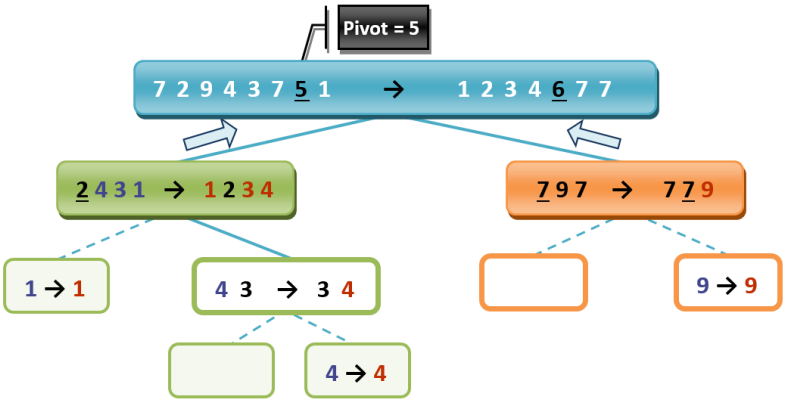

Fig. 1 Quicksort recursive steps

The initial step of the divide process is selecting a comparison element X. All the elements of the list which have been lower than X are put in the first sub list, all of elements higher than X are put in the second sub list. For elements equivalent to X it does not matter into which sub lists they are presented. It could possibly also occur that an element equivalent to X remains between the two sub list [12, 13]. Figure 2 show the sequential Quicksort implementation outline.

```
void quicksort(int Array[],int first,int last){
    int pivot,j,temp,i;

     if(first<last){
         pivot=first;
         i=first;
         j=last;

         while(i<j){

    while(Array[i]<=Array[pivot]&&i<last)
                 i++;
            while(Array[j]>Array[pivot])
                 j--;
            if(i<j){
                 temp=Array[i];
                  Array[i]=Array[j];
                  Array[j]=temp;
            }
         }

         temp=Array[pivot];
         Array[pivot]=Array[j];
         Array[j]=temp;

         quicksort(Array,first,j-1);
         quicksort(Array,j+1,last);
    }
```

Fig. 2 Quicksort implementation outline

The fundamental algorithm experiences the list in three noticeable stages:

1. Search and switch until i and j meet.
2. Create two sub-lists with values lower than X and higher than X.
3. Then recursive with the two sub-lists.

With recent modern technology increasing the number of transistors does not improve the computing ability of a computer system. One of the modern approaches applied to improve the overall performance of a computer system is the creation of multi-core processors. In multi-core processors environment, two or more processors are employed in order to enhance performance and improve efficiency [14].

A program functionality operating on a single-core can be improved over multi-core by splitting the entire program into several threads which can simultaneously work on multiple processors. Because of the growing popularity of multi-core technologies, a lot of efforts to parallelism have been recorded in literature. Away from current techniques, there is yet enough space for creating effective technique for better parallelism [14]. In this paper we will focus on Quicksort algorithm, in precisely the parallel implementation of Quicksort algorithm using OpenMP.

II. PROGRAMS DEVELOPMENT USING INTEL VTUNE

Fundamental problem in parallel programming is to guarantee that the sequential program its effective and working right manner. Minimizing the workload of the program along with the resources it requires as much as possible of effort. The VTune performance analyser is an analyser program used as a tool to identify the most computational region of the sequential program [15]. VTune Call Graph is an extension to the VTune performance analysis program help the developer to illustrate program functions overall performance as well as to determine consumption time for each function. Programs that using multithreaded techniques its also could be analysis using thread checker and profiling that offering from Intel Company [15]. The Following is the summary of the tools provided by The VTune performance analyser program:

- Analyser: tool that help to identify the most computational region (Hotspot) to be parallelize using OpenMP directive.
- Debug for correctness to check the multithreaded version of our matrix multiplication program we used Intel Thread Checker.
- Tool for performance finely check the performance of the multithreaded program, checking the overhead issues and bottlenecks using Intel Thread Profiler.

In the next subsection more explanations about these tools and their usages in this project.

1. *Checking the Efficiency of Sequential Program*

In this section discuss the tools along with its advantages and capabilities that have been used in developing the parallel Quicksort algorithm, so this is considered as the first step in our multithread program developments.

A) VTune (Sampling): Assists to detect and characterize performance issues such as Hotspot region. Discovering Hotspots region assist programmer to identify the region in the program that require to speed up using the parallel methods [15].

B) VTune (Call Graph): VTune Call graph gathers information regarding the program flow of an application, an example of this information is how many function that calls other function as well as how much time consuming for each function to executing its code and/or calling other functions. The Call Graph tool offer to the programmer an overall view of program flow to determine the critical functions and call sequences. Determining the functions which have been called frequently throughout the program. The most independent frequently function it's a suitable part to be parallelized [15].

2. *Performance Experiment using VTune (Sampling and Call Graph)*

First, VTune sampling is used to check whatever there is a potential part of code that takes the most CPU cycles from the total process CPU_CLK. Figure 3 show the Intel VTune Performance Environment.

Events	Total
CPU_CLK_UNHALTED.CORE samples	24.00
INST_RETIRED.ANY samples	14.00
Clocks per Instructions Retired - CPI	1.71
CPU_CLK_UNHALTED.CORE %	21.43
INST_RETIRED.ANY %	23.33
CPU_CLK_UNHALTED.CORE events	63,984,000.00
INST_RETIRED.ANY events	37,324,000.00

Fig. 3 Intel VTune Performance Environment.

The sequential program takes CPU_CLK 24 samples totally. Figure 4 shows the CPU clock for QSseq.exe.

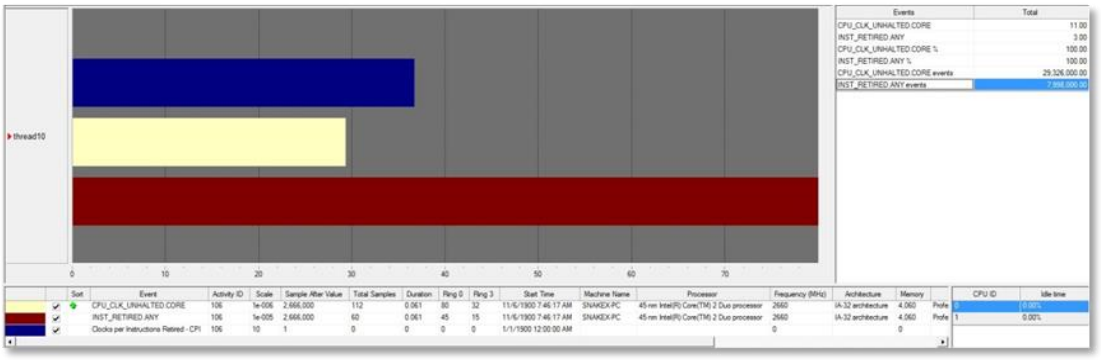

Fig. 4 Intel Graph representation for CPU Clock

Figure 5 showing the Quicksort recursive function that takes the most CPU cycles and instructions.

```
void QuickSort(int p, int r)
{
    if (p < r)
    {   int q = Partition(p, r);
        QuickSort (p, q-1);
        QuickSort (q+1, r);
    }
}
```

Fig. 5 Represent the recursive function for Quicksort

Using VTune sampling we could go deeply and check the exact part of code that consuming the CPU_CLK and frequently instructions calls.

III.PROPOSED METHOD

The proposed parallel Quicksort algorithm initially, starting with pre-processing of the input dataset to remove the special characters from the input text file, after that the fundamental idea is to creating many additional temporary sub-arrays according to a number of characters in each word, the sizes of each one of these sub-arrays are adopted based on a number of elements with the exact same number of characters in the input array. Finally, the elements of the input datasets is distributing into these temporary sub-arrays depending on the number of characters in each word. The datasets chosen in this paper are consist of two types of text file

(Hamlet and Moby Dick), these dataset are different in size and length. The size of the first dataset is equal to (180 KB) the second one is equal to (1.18 MB) which are downloaded from (http://www.loyalbooks.com/).

A) Pre-Processing

The pre-processing phase of the input dataset using the Quicksort algorithm is performed in three stages. The first stage, the algorithm remove / ignore all the special characters from the input text file. In the second stage, the algorithm transform the text file to array of list (vectors of string) according to the length of characters, the shorter words come first and the longer words comes last. The third stage, we sort every single vector of string by arranging in the alphabetic order using Quicksort algorithm. Table I shows the execution time of pre-processing phase. The sequential and parallel programs of the proposed algorithm have been implemented using laptop (ASUS) with Intel Core(TM) i7-3630QM @ 2.4GHZ and 8 GB of RAM. The operating system used is Windows 8.1 Pro edition 64- bit, Microsoft Visual Studio 2010 is the editor that is used as an environment to write the codes and the visual C++ compiler compiles and runs the programs.

TABLE I
EXECUTION TIME OF PRE-PROCESSING PHASE

Function	Dataset 1 (Hamlet) (seconds)	Dataset 2 (Moby Dick) (seconds)
Pre-processing to remove the special characters from a text file	3.6414	29.6906
Reading a text file	4.5668	43.6574

B) Parallel Quicksort algorithm using OpenMP directives

OpenMP is an application programming interface (API) for shared parallel programming systems. It offer high level programming constructs by utilizes a set of directives which are written within a code, that coded by one of the high-level programming language such as C/C++ or FORTRAN [16]. In the OpenMP model, the implementation is usually began by a master thread. Whenever the code comes into a parallel region the master thread creates several slave threads that work together to carry out the calculations. The slaves thread shares information with other threads, owning additionally a private memory for auxiliary and temporary variables. Figure 6 shows the typical execution model of a simple OpenMP program [16].

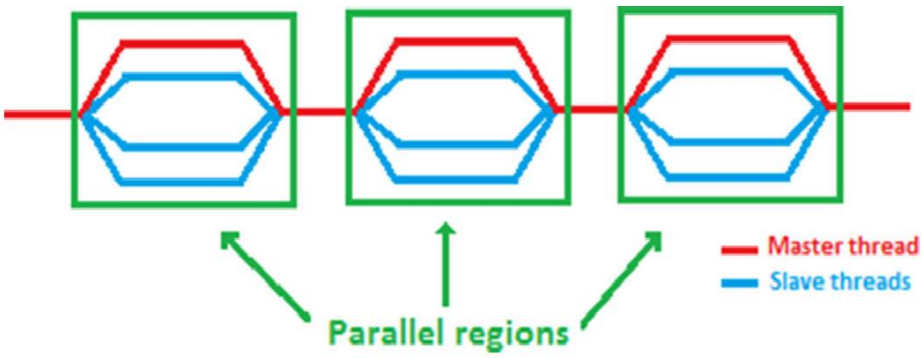

Fig.6 Program flow in an OpenMP execution model [16]

Based on the problem given in the previous section, showed that the algorithm creating many additional temporary sub-arrays according to a number of characters in each word, based on length of word this array distribute into a sub arrays to sort each array using the Quicksort algorithm. The VTune performance analyser result shown in previous section that the Quicksort recursive function analysis that takes the most CPU cycles and instructions. The Quicksort recursive function considered the hotspot of the program, for each sub array we assign a set portion of data to each thread. Since Quicksort algorithm is based on recursive calls for two partitions of dataset in a level, we assign each call and each partition to a thread in order the partitions will be arranged indecently as shown in Figure 7.

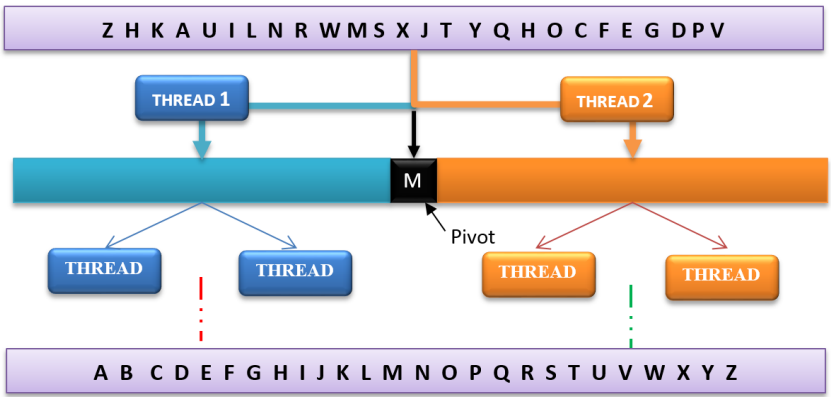

Fig.7 Quicksort Algorithm using OpenMP for each array

After apply the VTune performance analyser tools that have been explained in (section II), several useful information helps us to detect various issues that helped in parallelize the high computational region. From the first step, extracted information shows that the recursive function its take the most program time and CPU_CLK. Based on the previous information we decide to parallelize each call for the recursive function to minimize the consuming time and speedup the program. Figure 8 show the implementation code of the proposed parallel Quicksort recursive function.

```
void parQuickSort(int p, int r,vector<string> &B)
{
    if (p < r)
    {   int q = Partition(p, r,B );
        #pragma omp parallel sections
        {
                #pragma omp section
                parQuickSort (p, q-1,B);

                #pragma omp section
                parQuickSort (q+1, r,B);
        }
    }
    }
```

Fig.8 Proposed Quicksort recursive function

Parallel sections: executing each section concurrently is the same as executing each section sequentially. In the proposed parallel recursive function the algorithm call itself twice, the OpenMP sections directive used to Parallelization the recursive (see Figure 8).

IV. RESULT ANALYSIS

In this section, experimental results of the sequential Quicksort algorithm as well as the parallel Quicksort algorithm are compared with each other. We performed the experiments on the sequential and parallel Quicksort algorithm several times on different dataset size (File 1: Hamlet.txt 180 KB and File 2: Moby Dick.txt 1.18 MB). During the experiment, the program is executed 5 times, and the average time for all the reading is chosen as the sequential execution time for the Quicksort algorithms shown in Table II.

TABLE III
SEQUENTIAL TIME FOR DIFFERENT DATA SIZE

Reading No.	**File 1 (180 KB)**	**File 2 (1.18 MB)**
Reading 1	0.040012	2.401233
Reading 2	0.041692	2.396601
Reading 3	0.045342	2.400768
Reading 4	0.045127	2.399871
Reading 5	0.044149	2.411457
Average	0.0432644	2.401986

The second experiment is performed using OpenMP directive, the program is executed 5 times, and the average time for all the reading is chosen as the parallel execution time for the Quicksort algorithms. The experiment is done using different dataset size. The OpenMP program is scalable since it could sort huge dataset size and also the number of thread can be changed depend on the computer architectures. In this experiment, OpenMP program is performed in different number of threads, starting with only 1 thread and double the thread number to 2 threads and 4 threads, as shown in Table III.

TABLE IIIII
OPENMP PARALLEL TIME FOR DIFFERENT DATA SIZE AND DIFFERENT NUMBER OF THREADS

Number of Threads		**File 1 (180 KB)**	**File 2 (1.18 MB)**
1	Reading 1	0.041054	2.401157
	Reading 2	0.040092	2.390131
	Reading 3	0.044315	2.411054
	Reading 4	0.039951	2.400722
	Reading 5	0.039818	2.388016
Average		0.041046	2.398216
Number of Threads		**File 1 (180 KB)**	**File 2 (1.18 MB)**
2	Reading 1	0.041054	2.401157
	Reading 2	0.040092	2.390131
	Reading 3	0.044315	2.411054
	Reading 4	0.039951	2.400722
	Reading 5	0.039818	2.388016
Average		0.029341	1.97757828

Number of Threads		File 1 (180 KB)	File 2 (1.18 MB)
4	Reading 1	0.015838	1.40058
	Reading 2	0.015899	1.400137
	Reading 3	0.015812	1.399865
	Reading 4	0.015814	1.399069
	Reading 5	0.015784	1.400533
Average		0.0158294	1.4000368

Figure 9 illustrates the execution times of the OpenMP Parallel program for different dataset size using two datasets (File1 180 KB and File2 1.18 MB). The Figure show that the performance is increased to twice when using 2 threads instead of only 1 thread, as well as the result show when using 4 threads the performance is almost double the result when using 2 threads.

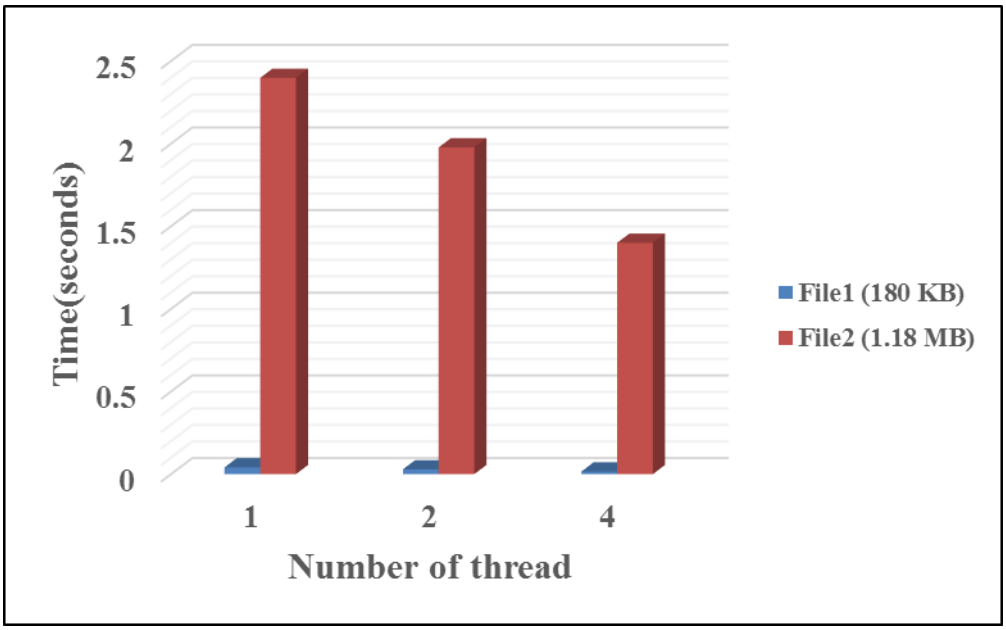

Fig.9 OpenMP parallel execution time

A parallel program is intrinsically much more complex compared to its sequential program. To write a powerful and scalable parallel program one must figure out the behaviour and performance of the sequential program, to achieve that we will measure some important factors such as: Speedup and Efficiency.

1) Speedup: The first factor taken into account when it comes to the performances of the parallel systems are assessed is the speedup utilized to show how many times a parallel program works faster when compare to the sequential program, wherever the two programs are solving the same problem. The primary purpose of parallelization a sequential program is to run the program faster [17]. The speedup Equation is shown below:

$$\textbf{Speedup } (S) = {}^{T_s}\!/_{T_p} \text{ ..... (1)}$$

Whereas Ts is the execution time of the best sequential program that solves the problem, and Tp is the execution time of the parallel program used to solve the same problem, as shown in Equation 1.

2) Efficiency: The parallel efficiency quantifies the quantity of the useful works performed by the processors throughout the execution of the parallel program [17]. The parallel efficiency are expressed as the following Equation:

$$\textbf{Efficiency } (E) = {}^{S}/_{C} \text{ ..... (2)}$$

Where S represents the parallel speedup and C corresponds to the number of processors or threads, as shown in Equation 2.

The experimental results of the parallel performance metrics using different dataset size (File 1: Hamlet.txt 180 KB and File 2: Moby Dick.txt 1.18 MB) is shown in Table IV.

TABLE IVV
OPENMP SPEED UP AND EFFICIENCY FOR DIFFERENT DATA SIZE

Number of Threads	File 1 (180 KB) Sequential	File 1 (180 KB) Parallel	File2 (1.18 MB) Sequential	File 2 (1.18 MB) Parallel	Speedup1	Speedup2	Efficiency1	Efficiency2
1	0.0432644	0.041046	2.401986	2.398216	1.054047	1.001572	0.131756	0.125197
2	0.057682	0.029341	3.739857	1.97757828	1.965918	1.89113	0.24574	0.236391
4	0.051559	0.0158294	3.929974	1.4000368	3.257167	2.807051	0.407146	0.350881

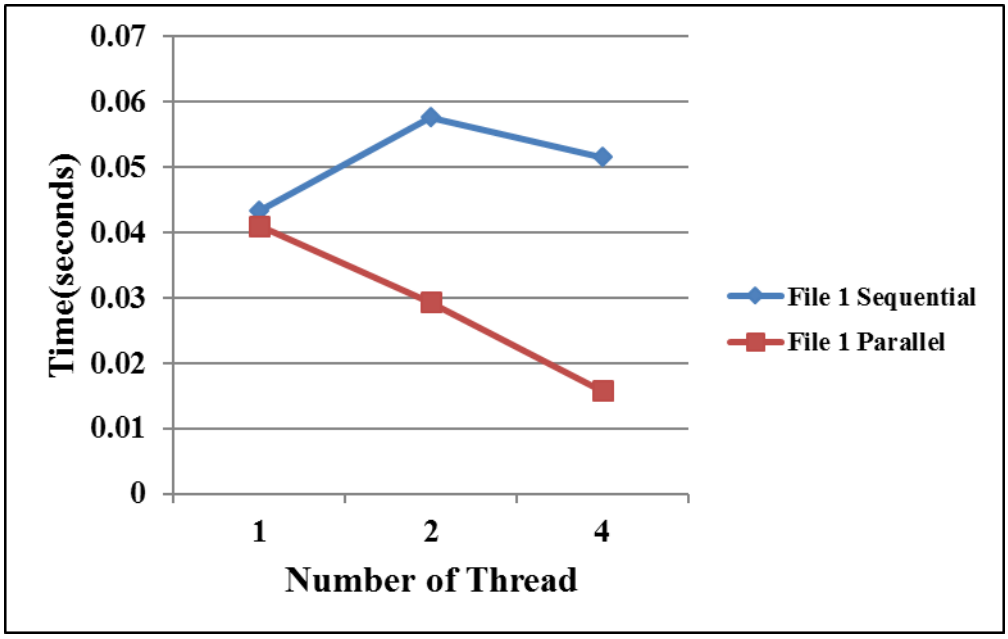

Fig.10 Sequential and parallel execution time of the Quicksort algorithm using File1

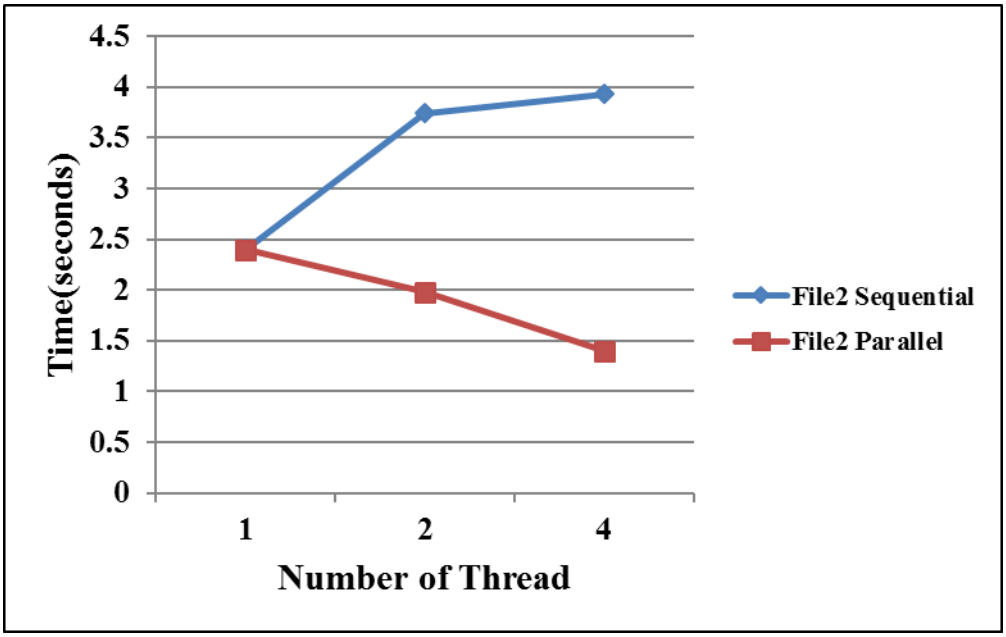

Fig.11 Sequential and parallel execution time of the Quicksort algorithm using File2

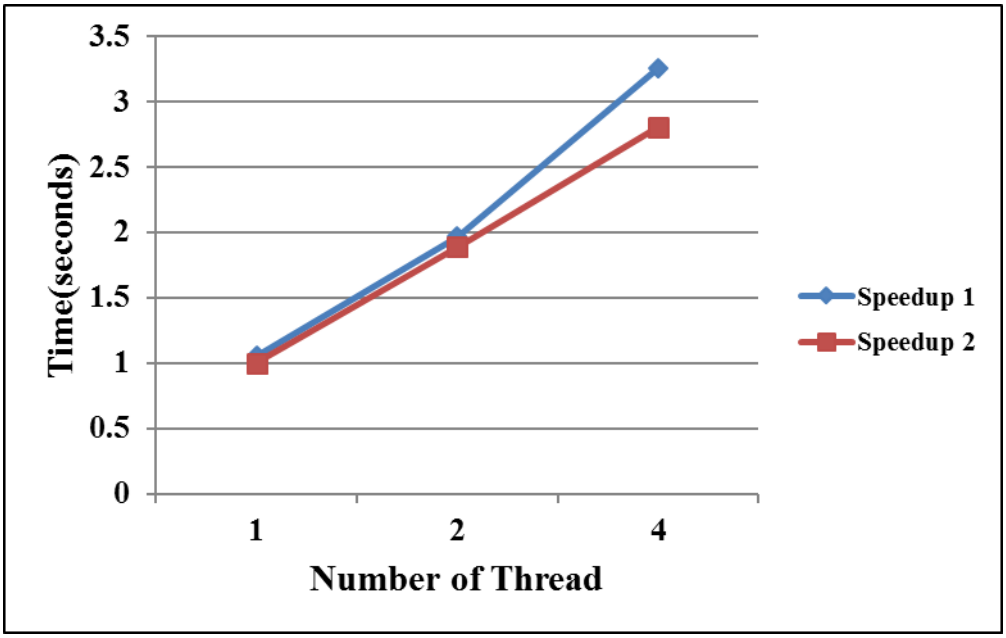

Fig.12 The speedup of File1 and File2 using OpenMP

The results of using (File 1: Hamlet.txt 180 KB) shows higher speedup when running the parallel algorithm compared to the sequential algorithm in different number of threads. The ratio of the efficiency using 4 cores with 4 threads in the parallel method is close by the optimal result of utilizing such number of cores. It could be viewed that the ratio of the efficiency increases when the number of cores increases, (see Figure 10). Moreover, when using (File 2: Moby Dick.txt 1.18 MB) also shows higher speedup when running the parallel program compared to the sequential program with different number of threads. The ratio of the efficiency increases when the number of cores increases, (see Figure 11). Figure 12 show the speedup when using File1 and File2 data types with different number of threads.

V. CONCLUSIONS

This study aims to parallelization the Quicksort algorithm using multithreading (OpenMP) platform. The proposed method examined on two standard dataset (File 1: Hamlet.txt 180 KB and File 2: Moby Dick.txt 1.18 MB) with different number of threads. Experimental results of this study, which have been explained carefully in the previous sections, reveal that the performance of parallelization the proposed Quicksort algorithm has shown improvement when compared to the sequential Quicksort algorithm by delivering improved Execution Time, Speedup and Efficiency.

It is found that when using File1 and File2 dataset shows higher speedup when running the parallel algorithm compared to the sequential algorithm in different number of threads. The ratio of the efficiency using 4 threads with 4 cores in the parallel method is close by the optimal result of utilizing such number of cores. This lead to conclusion when the number of cores increase the ratio of the efficiency increase too. We planned in the future to implement the Quicksort algorithm using Message Passing Interface (MPI) and compare its results with OpenMP method.

REFERENCES

[1] Umeda, Takayuki, and Shuhei Oya. "Performance comparison of open-source parallel sorting with OpenMP." 2015 Third International Symposium on Computing and Networking (CANDAR). IEEE, 2015.
[2] Alyasseri, Zaid Abdi Alkareem, Kadhim Al-Attar, and Mazin Nasser. "Parallelize Bubble Sort Algorithm Using OpenMP." arXiv preprint arXiv: 1407.6603 (2014).
[3] A. C. Dusseau, D. E. Culler, K. E. Schauser, R. P. Martin, Fast Parallel Sorting under Log P : Experience with the CM-5, IEEE Trans. Parallel Distributed Systems, 7, 791–805 (1996).
[4] S. J. Lee, M. Jeon, D. Kim, A. Sohn, Partitioned parallel radix sort, J. Parallel Distr. Comput., 62, 656–668 (2002).
[5] K. Batcher, Sorting networks and their applications, Proc.AFIPS Spring Joint Computer Conference, 32, pp.307–314 (1968).
[6] J. S. Huang, Y. C. Chow, Parallel Sorting and Data Partitioning by Sampling, Proc. 7th Computer Software and Applications Conference, pp.627–631 (November 1983).
[7] M. Jeon, D. Kim, Parallel merge sort with load balancing, Inter. J. Parallel Prog., 31, 21–33 (2003).
[8] Knessl, Charles, and Wojciech Szpankowski. "Quicksort Algorithm Again Revisited." Discrete Mathematics & Theoretical Computer Science 3.2, 43-64 (1999).
[9] D. Knuth, the Art of Computer Programming. Sorting and Searching, Addison-Wesley (1973).
[10] H. Mahmoud, Evolution of Random Search Trees, John Wiley & Sons, New York (1992).
[11] Aumüller, Martin, and Martin Dietzfelbinger. "Optimal Partitioning for Dual-Pivot Quicksort." ACM Transactions on Algorithms (TALG) 12.2, 18 (2015)
[12] Edelkamp, Stefan, and Armin Weiß. "BlockQuicksort: How Branch Mispredictions don't affect Quicksort." arXiv preprint arXiv: 1604.06697 (2016).
[13] Neininger, Ralph. "Refined quicksort asymptotics." *Random Structures & Algorithms* 46.2, 346-361, (2015)
[14] Sabahat Saleem1, M. I, Multi-Core Program Optimization: Parallel Sorting Algorithms in Intel Cilk Plus. International Journal of Hybrid Information Technology Vol.7, No.2, pp 151-164 (2014).
[15] (2016) The intel Developer Zone website. [Online]. Available: https://software.intel.com/en-us/intel-vtune-amplifier-xe
[16] Petaccia, G., F. Leporati, and E. Torti. "OpenMP and CUDA simulations of Sella Zerbino Dam break on unstructured grids." Computational Geosciences, 1-10 (2016).
[17] Grama, Ananth. *Introduction to parallel computing*. Pearson Education, (2003).

