Research Journal of Applied Sciences, Engineering and Technology 12(4): 465-472, 2016
DOI: 10.19026/rjaset.12.2386
ISSN: 2040-7459; e-ISSN: 2040-7467

Submitted: September 9, 2015 Accepted: September 25, 2015 Published: February 25, 2016

Research Article

Experimental Study of Parallelizing Breadth First Search (BFS) Algorithm

[1]Hakim Adil Kadhim, [2]Mohammad Ali Sarvghadi and [3]Aymen Hasan AlAwadi
[1]Department of Electronics and Communication, Faculty of Engineering, University of Kufa, Kufa P.O. Box (21), Najaf, Iraq
[2]School of Computer Sciences, Universiti Sains Malaysia, 11800 USM, Penang, Malaysia
[3]Information Technology Research and Development Center, University of Kufa, Kufa P.O. Box (21), Najaf, Iraq

Abstract: Recent years, many applications have exploited graph-based computations for searching massive data through distributed processors and memories. In this study, we present Breadth First Search algorithm as a graph-based algorithm to organize large DNA dataset and parallelize the algorithm using two highly tuned parallel approaches named OpenMP and MPI on a cluster, besides tuning the serial version of the algorithm. OpenMP is implemented using domain decomposition method. MPI is applied after dividing the dataset into equal parts and changing it to a two-dimensional array. Both approaches are tested in Khwarizmi cluster with 8 Intel Xeon Processors at the School of Computer Sciences in USM. The two aforementioned experiments are implemented and evaluated with certain characteristics and the results show high performance is achieved in terms of speedup and efficiency in comparison with the serial version of the algorithm.

Keywords: BFS, DNA searching, MPI, OpenMP, parallel computing

INTRODUCTION

A search algorithm is used to find an item with certain properties among a collection of items, sorted in various manners, like records in a database or an element defined by mathematical procedure of a search space (Donald, 1999). A search algorithm could also be used to solve and evaluate Discrete Optimization Problem (DOP), which is a type of highly computational problems with theoretically and practically interest (Kumar *et al*., 1994). The searching algorithm provides solutions to DOPs from finite or infinite set of solutions. Many problems can be addressed as DOPs like the optimal layout of VLSI chips, robot motion planning, test-pattern generation for digital circuits and logistics and control (Kumar *et al*., 1994). There is variety of data structures for search algorithms like simple linear search, binary search tree, heaps and hash tables that could be used for large or normal databases.

The main existing types of search algorithms are tree and graph traversal, where the tree type imposes all the nodes of data structure to be examined in the search process in a practical way. On the other hand, in the graph traversal all nodes should be examined in a systematic way and may be more than once. So, the tree traversal could be considered as a special case of the graph traversal. There are two main algorithms for graph traversal named Depth First Search (DFS) and Breadth First Search (BFS).

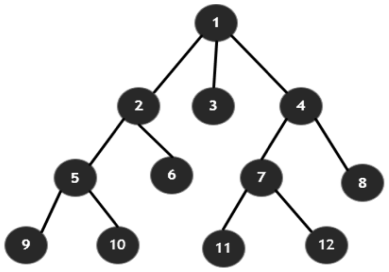

Fig. 1: The order of expanding the nodes in BFS algorithm

Breadth First Search (BFS): BFS expands the whole nodes beginning with the root node and explores all nearest neighbors one by one to find the required item as depicted in Fig. 1.

The algorithm does not use heuristic direct, so all the nodes should be expanded and added to the FIFO (First In First Out) queue. All the unexpanded nodes will be added to the container, which is organized as a Linked list and called open list. Afterwards, the produced list will be added to the other container, which is a closed list after being examined by the algorithm. Therefore, the FIFO queue will hold the final result.

Corresponding Author: Aymen Hasan AlAwadi, Information Technology Research and Development Center, University of Kufa, Kufa P.O. Box (21), Najaf, Iraq

Applications: The main usage of BFS algorithm is based on finding the shortest path between the entry point and the exit point. BFS is employed to solve several problems such as:

- Finding all nodes within one connected component
- Copying Collection, Cheney's algorithm
- Finding the shortest path between two nodes u and v
- Testing a graph for bipartiteness
- (Reverse) Cuthill-McKee mesh numbering
- Ford-Fulkerson method for computing the maximum flow in a flow network
- Serialization and Deserialization of a binary tree vs. Serialization in sorted order, allows the tree to be re-constructed in an efficient way.

In current work, the choice of BFS algorithm did not come because of its fast performance only, but because of the simplicity provided to serve various applications (Süß and Leopold, 2006). The contributions of this study could be listed as below:

- Presents three tunable and scalable forms of BFS and employs them to serve DNA various related search processing
- Employs two highly tuned parallel approaches (OpenMP and MPI) on a cluster
- Proves high performance achieved by means of the cluster and analyzes the result.

LITERATURE REVIEW

This section reviews recent works of BFS parallelization based on current trends of parallel algorithms.

Multithreaded by Madduri *et al*. (2009) offered a faster parallel algorithm for evaluating betweeness centrality and performed a detailed analysis for performance of the proposed algorithm. They implemented the optimized algorithm for the Cary XMT and achieved lower synchronization overhead and memory cache. Later, Mizell and Maschhoff (2009) improved their work depending on (Madura *et al*., 2009) by tuning the algorithm for Cray XMT (an improved 64 processors version of (MTA-2)) and achieved 350x faster running time on the new cluster than an MPI approach on other cluster.

General Purpose Graphical Processor Unit (GPGPU) was adopted to accelerate the graph and data processing due to the huge amount of parallelism that can be attained by current GPUs. Harish and Narayanan (2007) proposed the first implementation to various graph algorithms including BFS by employing Nvidia GPU and CUDA. Hong *et al*. (2011a) reduced branch divergence produced with a warp-centric programming model. Recently, Tran *et al*. (2015) introduced GpSM, which is a GPU massive parallel architecture method for sub-graph matching based on filtering-and-joining techniques.

Distributed memory architecture was used in some researches to address the graph computing problems caused by entity of scalability in graph decomposition implementation as well as high synchronization overhead in MPI (Beamer *et al*., 2013). Edmonds *et al*. (2010) ombined lightweight graph metadata transactions with active messages and proved that this combination can leverage the parallelism of the distributed memory graph applications. By improving the memory access locality, Cong *et al*. (2010) were able to develop fast PGAS for graph algorithms. Checconi *et al*. (2012) proposed an efficient version of BFS on IBM Blue Gene/P and Blue Gene/Q architectures. They achieved high performance on large graph, due to employing various techniques including bitmab storage, efficient graph representations, 1-dilation mapping, removal of redundant predecessor map updates and compression on communication. The Blue Gene*P version of the code was written by MPI for communication and OpenMP was used for on-node parallelism, while the Blue Gene/Q which is written entirely in C used Pthreads for on-node threading and SPI for commination. Beamer *et al*. (2013) used 2D decomposition approach to an earlier purely BFS algorithm in addition to top-down combined with a new bottom-up search steps to enhance edges expands. Amer *et al*. (2015) proposed a hybrid and distributed BFS algorithm using MPI-only and MPI+threads models at scale. In MPI+threads model, communication of threads is independent of remote processes during the process of synchronization with the local computation. The idea behind this type of hybridization comes to enhance the memory usage, node-to-node synchronization and communication performance for what MPI-only model offers by scaling certain parameters such as the size of the target system, where intranode communication and drawbacks of runtime contention in case of multiple threads exist.

Finally, the shared-memory parallelization parallelizes BFS algorithm on multi-core architecture. Hong *et al*. (2011b) proposed a hybrid model for BFS that can dynamically indicate the optimum performance for each BFS-level iteration in order to utilize the performance in both large and small graphs. Yasui *et al*. (2013) applied a column-wise portioning for the adjacency list of edges to Beamer *et al*. (2011) algorithm on Non-Uniform Memory Access architecture (NUMA) using SandyBridge-EP. The authors explained two affinity strategies, (scatter-type and compact-type) for the NUMA architecture. The scatter-type optimized the distribution of OpenMP threads evenly. On the other hand, the compact-type binds the OpenMP threads closely in a free thread context. This optimization could avoid the overhead usage on the remote RAM access, since the local

Table 1: The general information of sequential method

Criteria	Time (sec)
Average Concurrency	0.68
Logical Processors	2
Threads	1
Transitions	10103
Transitions Per Sec	1266.58
Filtered Transitions	10
Filtered Transition Time	0.00
APIs	70789
APIs Per Sec	8874.57
Total critical path time	7.94144 (100.00%)
No thread active	2.55204 (32.14%)
Serial time	5.3894 (67.86 %)
Wait time	2.55204
Active time	5.3894
Total time	10103

threads traverse each adjacency list on local RAM (Yasui *et al*., 2013).

EXPERIMENT SPECIFICATION

Size of the used DNA dataset is about 385 MB and contains 90,000 strings including three characters only. The experiment is implemented on Khwarizmi cluster in the school of Computer Science at USM. Performance of the parallelization over the algorithm is analyzed using special performance analyzers such as Intel VTune Performance, Thread Checker and Thread profiler. The code is written in C using Microsoft Visual Studio 2008 Professional as platform.

Sequential performance: Since BFS is used in the experiment, all nodes should be expanded. Therefore, we expect a big amount of memory to handle an extensive search space. The following table is obtained from the Intel thread profile and shows the general information about the sequential performance.

Intel thread checker is used to obtain information in Table 1. Figure 2 depicts the thread profile chart that shows the algorithm is running serially (single threading) as can be seen by the orange bar with execution time of 5.3894 sec. On the other hand, no threading activity that is shown by the gray bar, took about 2.38254 sec for the whole experiment.

The waiting time is too large and affects the whole performance of the algorithm. Therefore, to overcome this drawback, two parallel algorithms are proposed based on two approaches of OpenMP and MPI.

Parallelization: As shown above, in sequential performance, the total critical path time is large (7.94144 sec) and it takes 100% of the total time. In this experiment, two parallelizing approaches of OpenMP and MPI are employed to achieve better performance on a specific architecture. The parallelization experiment has been analyzed and compiled on Khawarizimi cluster machine in USM.

OpenMP: In shared memory architecture, multiple CPUs are sharing the same amount of memory. OpenMP API is an example of multi-platform shared memory multiprocessing programming where multithreading concept is applied to rewrite the algorithm in a parallel manner. In OpenMP, programmer should think about parallelizing computation parts of the program as possible in successive steps (Harish and Narayanan, 2007) and then the result of each step (thread) will be collected

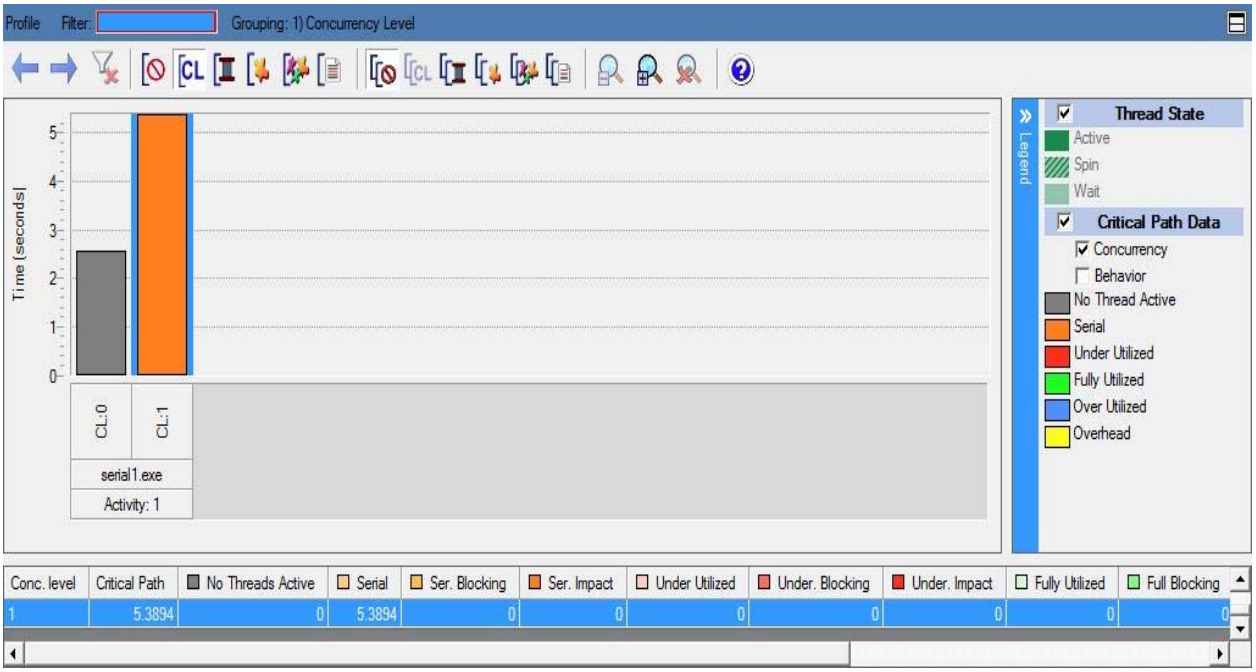

Fig. 2: Thread profile chart

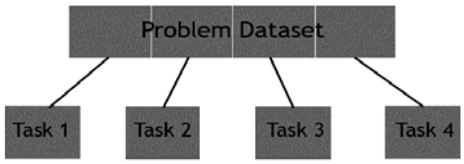

Fig. 3: Domain decomposition

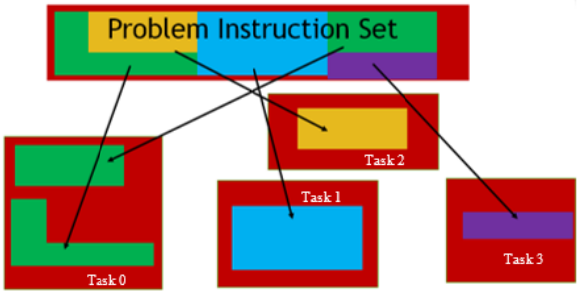

Fig. 4: Functional decomposition

yielding the main function. The threads are running individually, but at the same time on separate processors, where each thread performs its own task and sharing the resources with other threads. Finally, the result of each thread collected in order to form the final result. OpenMP is considered as a library that could be added to C/C++ and FORTRAN. The library is portable and can be implemented in most Operating Systems like Windows NT and Unix/Linux platforms. Now, which tasks should be assigned to threads?

- Independent tasks of the sequential code that can be parallelized and run at the same time
- Functions that have the most computation of the sequential code and can be divided into threads
- Independent functions, which can be parallelized without affecting other functions.

Decomposition: The Decomposition or portioning refers to splitting the problem into multiple sub-problems. There are two types of decomposition: Domain decomposition and functional decomposition. In domain decomposition, the problem data set is partitioned into sub-datasets and each sub-dataset of the function works on certain portion of the data as shown in Fig. 3.

The functional decomposition is dividing the problem or function into different sub-functions with different tasks as shown in Fig. 4.

In this experiment, the main objective is to reduce the waiting time of sequential code of the algorithm as much as possible. The domain decomposition method is used to parallelize the sequential code by splitting the most computational functions into pieces, hence there is a wide range of data which can be processed in parallel using (#pragma omp parallel) number of threads (four

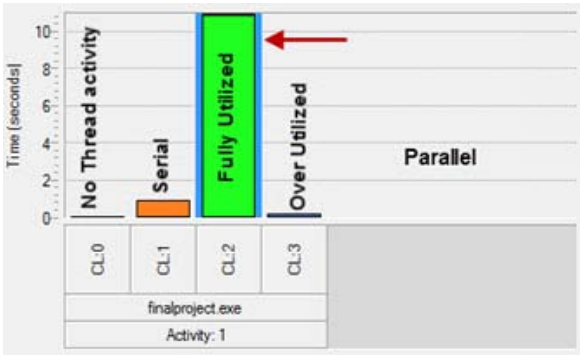

Fig. 5: Comparison between no thread activity, serial optimization, fully utilization and over utilization

or five threads). Choosing the best number of threads is experimentally achieved by trial and error after getting the result for each experiment. We got unsatisfactory results every time we exceeded this number of threads. This is because when the number of threads increases, the amount of fork and join activity among the threads increases too, in addition to the overhead of synchronization is needed to form the final result.

The second portion of the parallelization is performed on the function that changes the one-dimensional array into two-dimensional array, in addition to the function of printing data after changing it into a two-dimensional array. The main computational part of the code is not parallelizable; because of the data-trace that happens between variables. According to the Bernstein condition, only independent functions can be in parallel (Beamer *et al*., 2013) and if there is any overlap between variables, errors will occur with threading synchronization. The data-race error occurred six times because of the dependent variables between the functions. According to the Bernstein condition (Beamer *et al*., 2013), there are three types of data dependency:

- Flow dependence: write -> read
- Anti-dependence: read -> write
- Output dependence: writ -> write

Writing and reading activities are performed with variables within the function. Most of the errors are following the dependence problem and only one of them founded anti-dependence.

RESULTS AND DISCUSSION

The objectives of reducing the total waiting time and utilization of the resources were achieved. The Information related to the “Fully utilization” bar in bar chart of Fig. 5 is shown in Table 2.

It can be seen that “No Thread activity” bar is declined to 6.66741e-005 sec (about 0%). The serial bar is also reduced to 0.00137793 sec because of deploying

Table 2: Analyzed result of Open MP experiment

Criteria	Time (sec)
Total critical path time	12.1019 (90.61%)
No thread active	6.66741e-005
Serial time	0.00137793
Serial blocking	0.896081
Serial impact	0.029866
Under utilize	0
Under blocking	0
Under impact	0
Full utilized	1.03663
Full blocking	9.62859
Full impact	0.246098
Over utilized	0.201813
Over blocking	0
Over impact	0
Overhead	0.0613929

OpenMP of three threads. The "Over Utilized" bar raised by 1.03663 sec and this is an acceptable rising because of the concurrency level that matches the number of the processors. Figure 5 illustrates the comparison between fully utilized, serial optimization, no thread activity and over utilization.

Speedup and efficiency: the speedup refers to how fast parallel computing is in comparison to the sequential computing for the same algorithm (Hennessy and Patterson, 2011). Speedup ratio can be measured by Eq. (1), where T1 is the sequential execution time and Tp is parallel execution time:

$$S_P = \frac{Tl}{T_P} \quad (1)$$

For the whole execution time, on Khwarizmi cluster with four threads the following results are achieved:

$$S_P = \frac{0.07}{0.04} = 1.75$$

For the whole execution time, on a PC with Intel Core2Due 2.20 GHz machine, running windows on it with 5 threads the following results are achieved:

$$S_P = \frac{7.385}{7.163} = 1.0309$$

According to the achieved results, the speedup ratio on Khwarizmi cluster is greater than on regular PC because of the number of the processors on the cluster. The execution time of the parallel or serial may vary according to state of the machine of the experiment (i.e., how much of the process is already done on the machine).

Efficiency can be defined as how well processors are utilized in the parallel execution of the algorithm compared to the wasted time in communication and synchronization between the processors. Efficiency can be calculated by Eq. (2), where Sp is the speedup ratio and P is number of the processors:

$$E_P = \frac{S_P}{P} \quad (2)$$

In Khwarizmi cluster the efficiency of using the processors for 4 threads only:

$$\frac{1.75}{4} = 0.43$$

Efficiency by using PC with five or with four threads are:

$$\frac{1.0309}{5} = 0.206 \text{ And } \frac{1.0309}{4} = 0.257$$

Efficiency value is typically between zero and one. Since we have achieved acceptable speedup on the cluster experiment, the efficiency will be better than the PC experiment. Overall, efficiency of the threads and speedup need to be improved in order for processors to be more utilized.

MPI EXPERIMENT

MPI stands for Message Passing Interface, which is a standard library for message passing interfaces. MPI is a specification not an implementation. The purpose of MPI is to develop a widely used standard for writing message passing programs and achieving practical, portable, efficient and flexible standard for message passing. It can be implemented in C, C++ and FORTRAN languages. MPI is consisted of functions, subroutines and methods and they have a different syntax from one language to another one.

In parallel programming, the problem will be divided to sub-problems or processes and then passed to the processors. The responsibility of MPI is to establish communication between processes. Therefore, it is used for distributed memory not for shared memory.

There are different versions of MPI such as MPI 1.0, MPI 1.1, MPI 1.2, MPI 2.0, MPI 1.3, MPI 2.1 and MPI 2.2 each of which has different facilities and conveniences.

MPI is suitable for networks of workstations even heterogeneous ones, shared memory implementations, such as multi core processors and hybrid structures too.

Types of communications in MPI: There are two types of communication in MPI. First, is point-to-point communication and second is collective communication. In this experiment, point-to-point is used for parallelizing the BFS algorithm. Before starting to explain point-to-point communication, we have to explain four routines that are common in almost every program in MPI. The first one is MPI_Init (int argc, char **argv). This routine is the first routine that must be called in any MPI interface and it must be called only once. Initialization routine has two

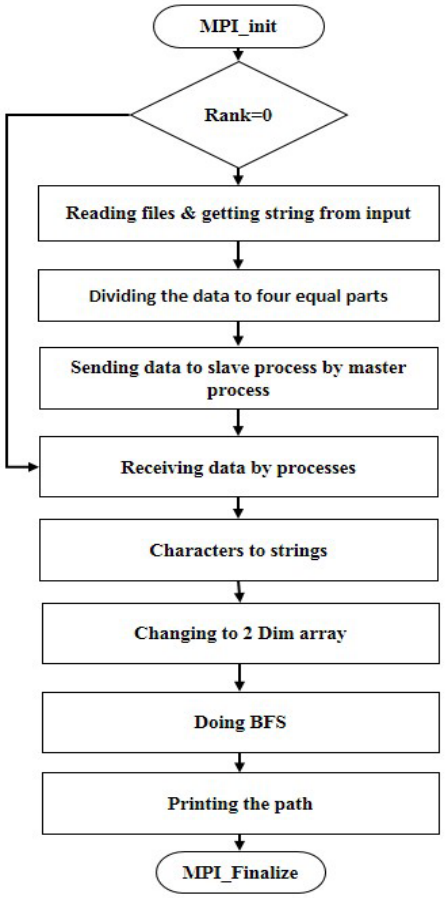

Fig. 6: Flowchart of MPI experiment

parameters; one is argc, which is a pointer to number of arguments and the other one is argv, which is a pointer to the arguments vector. The second routine is MPI_Finalize (). This routine is responsible for terminating the MPI execution environment. This routine cleans up all MPI state. Once this routine is called, no MPI routine (even MPI_INIT) may be called. The user must ensure that all pending communications involving a process are completed before the process calls MPI_Finalize.

The third routine is MPI_Comm_size (MPI_Comm comm, int *size). This routine has two parameters; the first one indicates the type of communicator that the processes employ and the second one indicates the number of processes involved in that specific communicator. The last common routine is MPI_Comm_rank (MPI_Comm comm, int* rank). This routine has two parameters too; the first one is exactly like MPI_Comm_size but the second indicates the rank of the process in the communicator.

Another two routines that will be used in the experiment for BFS are MPI_Send (void *buffer, int count, MPI_Data type data type, int dest, int tag, MPI_Comm *comm) and MPI_Recv (void *buffer, int count, MPI_Data type datatype, int source, int tag, MPI_Comm *comm, MPI_Status *status). As mentioned MPI_Send has six parameters and is responsible of sending data point to point from one process to another one.

Table 3: Analyzed result of OpenMP experiment

T1(sec)	Tp(sec)	Sp	P	Ep
0.07	0.043	1.75	4	0.43
0.07	0.008	8.75	24	0.36
0.07	0.007	10	32	0.31
0.07	0.009	7.77	40	0.19

Table 4: Speedup and efficiency comparison

Type of Parallel method	Speed up	Efficiency	Number of chars
OpenMP (4 threads)	1.75	0.43	270000
MPI (4 proc.)	1.75	0.43	270000
MPI (24 proc.)	8.75	0.36	270000
MPI (32 proc.)	10	0.31	270000
MPI (40 proc.)	7.77	0.19	270000

MPI Experiment: In this section, we will clarify the MPI experiment to the BFS sequential code.

Parallelization scenario: In this scenario, data read from the dataset file by the root process. Afterwards, this process will do the computation to divide the read data, which consists of characters of the same chunks in size and send them to different processes. The root process read the data and stored it in an array, after sending it to other processes in point to point manner. Then, the process of changing the array dimension of the strings began to change the one-dimensional array of the data to an array of 3-character strings and then it converted to a two-dimensional array to be ready for BFS algorithm. Next, each process started to do BFS for each array and compared each one with the requested string. In each event of searching through the dataset, the processes recorded the pass of events and then these events (passes) were combined to represent the final result. Figure 6 shows the flowchart of the MPI scenario.

Speedup and efficiency: The results of running the algorithm on the cluster with different number of processes and the execution time of the last process, which finished its job is summarized in Table 3. According to the sequential code, speedup and efficiency for MPI calculated by Eq. (1) and (2) respectively.

It is clear that the number of processes has affected the efficiency and the speedup ratio. When the number of processes increased, the speedup ratio increased too. However, the efficiency decreased in contrast. The best speedup achieved with 32 processes when the number of processes and processors were identical. When the number of processes grew to 40, both speedup and efficiency decreased. In conclusion, the best number for processes is equal to the number of processors.

Table 4 summarizes the comparison between the two experiments of parallel computing (OpneMP and

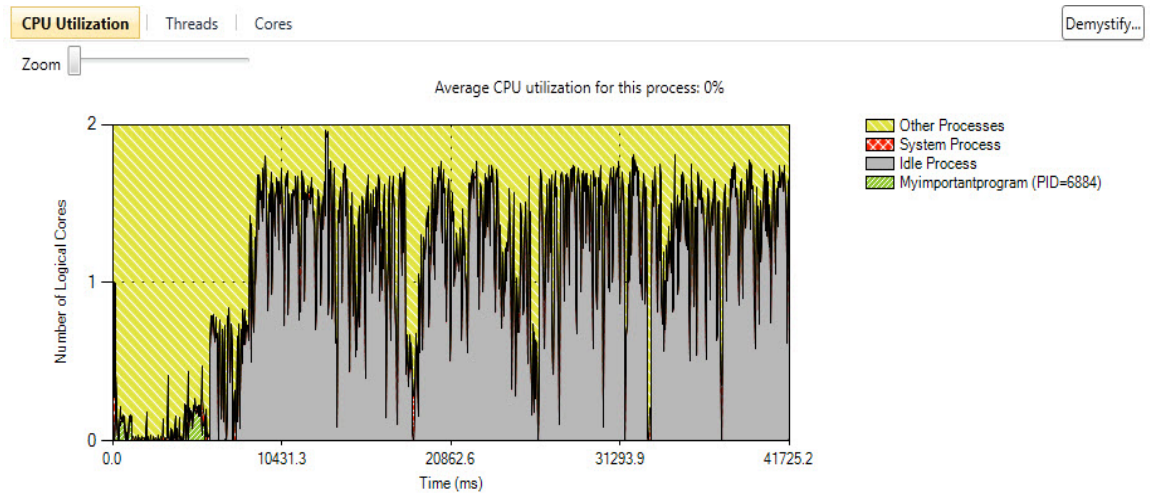

Fig. 7: CPU utilization report

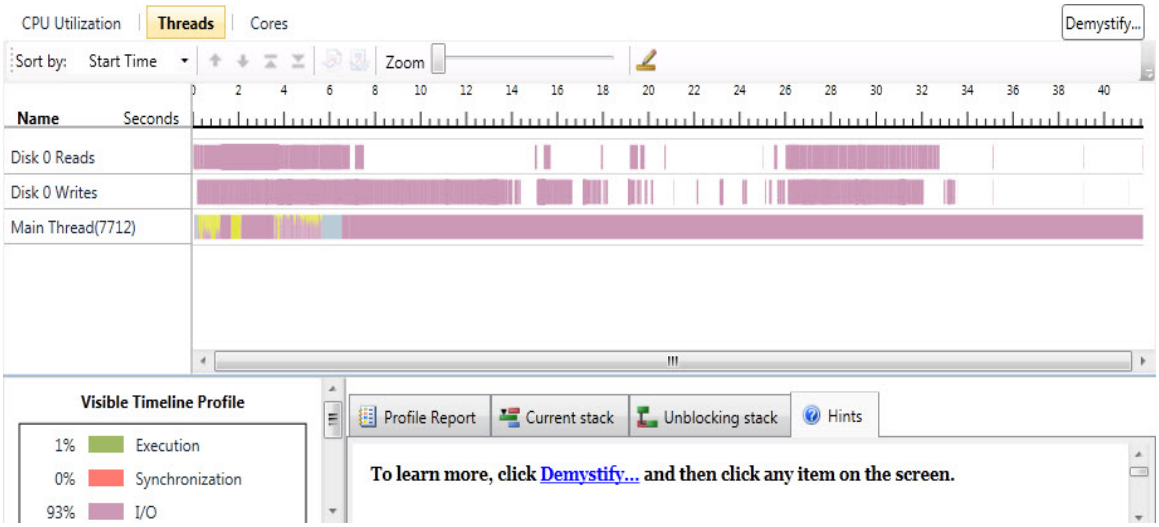

Fig. 8: Threads report

MPI) regarding the achieved speedup, efficiency and number of characters.

PERFORMANCE ANALYZING TOOLS

Performance analyzing tools can be classified into two categories: Profiling and tracing tools. The profiling tools depend on collecting timing summaries while tracing collect a sequence of time-stamped events. Profiling tools produce small amount of data that can be scaled well, on the other hand the tracing tools cannot be scaled properly (Chung *et al*., 2006).

In this study, two different tools from different vendors and developers were used to analyze the parallel performance of the algorithm. Intel Vtune thread checker and Microsoft Visual studio 2010 performance analyzer are the used tools. These tools are employed to produce CPU utilization, threads and Cores reports. Figure 7 and 8 show concurrency profiling reports for analyzing the performance of the BFS algorithm by using Microsoft Visual studio 2010 performance analyzer.

CONCLUSION

This paper studied parallelizing of BFS algorithm in two different approaches of OpenMP and MPI experimentally. Both parallel approaches may produce equal speedup ratio if it is executed in the same conditions (i.e., the number of threads/processes equal to the number of the processors). OpenMP was implemented using domain decomposition method and tested on Khwarizmi cluster with 8 Intel Xeon Processors. MPI was implemented by different number of processes on the same cluster. However, the achieved results (speedup and efficiency) for both

OpenMP and MPI were acceptable to some extent, but still need some enhancement to get better performance.

REFERENCES

Amer, A., H. Lu, P. Balaji and S. Matsuoka, 2015. Characterizing MPI and hybrid MPI+threads applications at scale: Case study with BFS. Proceeding of the IEEE Workshop on Parallel Programming Model for the Masses (PPMM, 2015), pp: 1075-1083.

Beamer, S., K. Asanovic and D.A. Patterson, 2011. Searching for a parent instead of fighting over children: A fast breadth-first search implementation for graph500. Tech. Rep. UCB/EECS-2011-117, EECS Department, University of California, Berkeley.

Beamer, S., A. Buluc, K. Asanovic and D. Patterson, 2013. Distributed memory breadth-first search revisited: Enabling bottom-up search. Proceeding of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW, 2013), pp: 1618-1627.

Checconi, F., F. Petrini, J. Willcock, A. Lumsdaine, A.R. Choudhury and Y. Sabharwal, 2012. Breaking the speed and scalability barriers for graph exploration on distributed-memory machines. Proceeding of the 2012 IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp: 1-12.

Chung, I.H., R.E. Walkup, H.F. Wen and H. Yu, 2006. MPI performance analysis tools on blue gene/L. Proceeding of the ACM/IEEE SC 2006 Conference, pp: 16-16.

Cong, G., G. Almasi and V. Saraswat, 2010. Fast PGAS implementation of distributed graph algorithms. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp: 1-11.

Donald, E.K., 1999. The art of computer programming. Sorting Search., 3: 426-458.

Edmonds, N., J. Willcock, T. Hoefler and A. Lumsdaine, 2010. Design of a large-scale hybrid-parallel graph library. Proceeding of the International Conference on High Performance Computing, Student Research Symposium. Goa, India.

Harish, P. and P.J. Narayanan, 2007. Accelerating large graph algorithms on the GPU using CUDA. Proceeding of the 14th International Conference on High Performance Computing (HiPC'07), pp: 197-208.

Hennessy, J.L. and D.A. Patterson, 2011. Computer architecture: A quantitative approach. Elsevier Science, Burlington.

Hong, S., S.K. Kim, T. Oguntebi and K. Olukotun, 2011a. Accelerating CUDA graph algorithms at maximum warp. ACM SIGPLAN Notices, 46(8): 267-276.

Hong, S., T. Oguntebi and K. Olukotun, 2011b. Efficient parallel graph exploration on multi-core CPU and GPU. Proceeding of the 2011 IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT, 2011), pp: 78-88.

Kumar, V., A. Grama, A. Gupta and G. Karypis, 1994. Introduction to Parallel Computing: Design and Analysis of Algorithms. Benjamin/Cummings Publishing Co., Redwood City, CA.

Madduri, K., D. Ediger, K. Jiang, D. Bader and D. Chavarria-Miranda, 2009. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. Proceeding of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS, 2009), pp: 1-8.

Mizell, D. and K. Maschhoff, 2006. Early experiences with large-scale XMT systems. Proceeding of the Workshop on Multithreaded Architectures and Applications (MTAAP'09), May 2009.

Süß, M. and C. Leopold, 2006. Implementing irregular parallel algorithms with OpenMP. In: Nagel, W.E. (Eds.), Euro-Par 2006. LNCS 4128, Springer-Verlag, Berlin, Heidelberg, pp: 635-644.

Tran, H.N., J.J. Kim and B. He, 2015. Fast subgraph matching on large graphs using graphics processors. In: Renz, M. *et al*. (Eds.), DASFAA, 2015, Part 1, LNCS 9049, Springer International Publishing, Switzerland, pp: 299-315.

Yasui, Y., K. Fujisawa and K. Goto, 2013. NUMA-optimized parallel breadth-first search on multicore single-node system. Proceeding of the IEEE International Conference on Big Data, pp: 394-402.