Modern Applied Science; Vol. 12, No. 4; 2018
ISSN 1913-1844 E-ISSN 1913-1852
Published by Canadian Center of Science and Education

Parallel Processing of Sorting and Searching Algorithms Comparative Study

Sa'ad Al-Azzam[1] & Mohammad Qatawneh[1]

[1] Computer Science Department, University of Jordan, Amman, Jordan

Correspondence: Sa'ad Abdalkarim Al-Azzam, University of Jordan, Amman, Jordan. E-mail: sa3d_al3zam@hotmail.com

Received: January 8, 2018 Accepted: January 12, 2018 Online Published: March 30, 2018

doi:10.5539/mas.v12n4p143 URL: https://doi.org/10.5539/mas.v12n4p143

Abstract

Recently, supercomputers structure and its software optimization have been popular subjects. Much of the software recently consumes a long period of time both to sort and search datasets, and thus optimizing these algorithms becomes a priority. In order to discover the most efficient sorting and searching algorithms for parallel processing units, one can compare CPU runtime as a performance index. In this paper, Quick, Bubble, and Merge sort algorithms have been chosen for comparison, as well as sequential and binary as search algorithms. Each one of the sort and search algorithms was tested in worst, average and best case scenarios. And each scenario was applied using multiple techniques (sequential, multithread, and parallel processing) on a various number of processors to spot differences and calculate speed up factor.

The proposed solution aims to optimize the performance of a supercomputer focusing one-time efficiency; all tests were conducted by The IMAN1 supercomputer which is Jordan's first and fastest supercomputer.

Keywords: MPI, message passing interface, sorting, parallel searching, parallel computing, bubble sort, quicksort, IMAN1, supercomputer

1. Introduction

In previous years, a high interest in building supercomputers with high performance and ability to perform up to quadrillions FLOPS (floating point operation per second) has been displayed. Building those computers usually require special processing units that are not widely available, and are usually very costly especially in terms of connecting them together and cooling them down (Markidisa, et al., 2016; Mohammed, 2005; 2011; 2011; 2016).

Structuring the computer hardware alone is not an efficient solution as one will need to find optimized software and optimal number of processors for that software to provide best results. For example, increasing number of processors will increase overhead and will not always result in faster process. This paper will dispute several sorting and searching algorithms performance on IMAN1 supercomputer using message passing interface (MPI).

Building a powerful computer needs a lot of resources that may not be available. In this case, creativity in building is required to achieve needed results. For example, IMAN1 supercomputer has been built using 2250 ready playstation3 boards (http://www.iman1.jo/iman1/). Those have been chosen as they are easily available and can communicate with each other easily through internet or special cables, beside the powerful advantages of the processing board itself such as having 8 powerful processors, the ability of accepting Linux OS, and having a very low failure rate (Parrya, et al., 2014).

Each supercomputer is built differently, and thus has a different running OS, internal structure and communication method between processors. This means that each supercomputer will have its own processing method for a given application. Thus, an optimal application for a supercomputer is not necessarily to be optimal for another supercomputer. This leads to think that a case study shall be made for every different supercomputer structure to be able to determine the optimal method to use in programming applications for this specific supercomputer.

A High-performance machine such as IMAN1 has no issues with memory space or computation power (for most applications) and the only resource which matters is the time efficiency as the main aim in building a supercomputer is to perform tasks fast.

One good method in comparing applications performance on a supercomputer is choosing an application that has

multiple and easy to access algorithms such as sorting and searching. Those can be performed in multiple ways and on various numbers of processors and the result of each run can be compared with other runs in order to find the most optimal algorithm and number of processors combination in terms of processing time required for IMAN1 supercomputer.

Sorting is a mechanism that puts elements of a dataset into a certain order, which in this case it is the numerical order, this is a commonly used process in which the input is a group of data elements and the output is the same group but organized in a particular order. While searching is the process of finding a single element or a group of elements with a specific description in set of data (Mohammeda, et al., 2017; Saadeh, et al., 2016).

There are many Algorithms that can perform sorting and searching processes, however the main issue is to find the most efficient algorithm for a specific device (IMAN1 supercomputer in this research), considering a Supercomputer is not concerned with Data size nor processing power but the performance index which becomes the actual time needed to process the information, and how it compares to its sequential counterpart (speed up).The choice of which algorithm to be used and the number of processors becomes an important issue to optimize the performance and free the resources.

MPI (Message Passing Interface) is standardized messages exchanging between multiple processors and processing distribution, it is the most commonly used paradigm in writing parallel programs (Gropp, et al., 1998). Since MPI is a high-level concept for parallel programming, programmers are able to simply build applications with parallel and distributed processing without extensive knowledge of the process production and synchronization exact mechanism. In order to exploit multi-processors, MPI processes could be organized to have multiple threads within them. MPI-based programs can be executed on a single multi-core device or a cluster (Jost, et al., 2003).

Furthermore, the unique contribution in this paper is stated in the obtained results, which discusses the most efficient sorting algorithm for a given number of processors, and then compares sequential search and binary search algorithms together, and states what sorting algorithm is needed to be used with binary search to optimize it depending on given number of processors. This information is very important in case we need to run multiple programs on a supercomputer, where each application, and depending on its priority has specific number of processors to run on. Theoretically, when designing an application to run on specific supercomputer, if the application has multiple sorting/searching algorithms programmed in it, and knows number of processors it will run on, then, and depending on this study, it will be able to choose the most optimal algorithm to run. And thus, no matter what conditions or load on supercomputer we have, the application will always run in minimum time.

This paper is organized as follows. Section 2 is related works and research. In section 3 the precise methodology is discussed with details; Section 4 presents the algorithms design use in the simulation of both sorting and searching algorithms respectively, besides the divide and conquer method of parallel algorithms, section 5 discusses the results obtained from the evaluation of sorting and searching algorithms and section 6 concludes the paper.

2. Related Work

(Balaji, et al., 2011) Shows an examination to the issue MPI scalability, it first, examines the MPI specification and how it could be overcome. Furthermore, it studied issues that MPI implementation must address in order to be scalable. To Show the arguments, a number of simple operations were made in order to measure MPI memory consumption at scale up to 131,072 processes, which is 79%, of the IBM Blue Gene/P system at Argonne National Laboratory. As such, by reviewing the results, non-scalable properties of the MPI implementation have been realized. And techniques to modify it in order to decrease memory allocations have been developed.

Due to some sorting problems, research into the matter has intrigued a large number of studies and as such many sorting algorithms have been developed with increasingly improved efficiency. Several algorithms are efficient for only a handful of applications; an illustration of such is (Bharadwaj, et al., 2013).

(Jeon & Kim, 2002) Shows a load-balanced merge sort algorithm was introduced which employs the total of the processors throughout the computation. It evenly distributes data to all processors for every stage. Thus, every processor is working in all phases.

(Adikari, 2007) Compares various performance properties amongst selection sort & shell sort algorithm determines that shell sort provides an excellent performance; however, both algorithms should not be used for large Datasets.

(Pasetto & Akhriev, 2010) Gives a general study of the execution of parallel sorting algorithms on modern multi-core hardware, various well-known algorithms were examined. These authors offered an insight as to which of the algorithms is best suited for a particular objective, as well as the deficiencies & benefits for each of the algorithms.

(Aliyu & Zirra, 2013) The authors examine both selection sort and quicksort algorithm had been compared for integer and string Datasets. Furthermore, the algorithms are examined on random data & results showed that selection sort performed better than quicksort and string Datasets are processed faster than Integer Datasets.

(Kumari & Chakraborty, 2015) a statistical comparative study of sorting algorithms, viz. Quick sort, Heap sort and K sort with a study of time complexity for each optimal average case complexity.

(Dominik, et al., 2013) Described the implementation results for some parallel sorting algorithms using GPU cards and multi-core processors. The author presented the hybrid algorithm and executed on both platforms CPU and GPU. The comparison of many core and multi-core is lacking. The threads are grouped into blocks and the blocks are grouped in grids. This work shows most advantages in terms of sharing resources between processors which effectively increases performance, but it also increases the overhead when passing messages which will waste processor time in doing processing that is not directly related to the application, and thus this method will not be efficient when considering a large number of processors such as 18000 processors like in IMAN1 supercomputer.

(Mukul, et al., 2014) Used the GPU architecture to solve the sorting problem. The highly parallel computing nature of GPU architecture is utilized for sorting purposes. The author considered the input dataset in the form of a 2D matrix which issued for sorting. The modified version of merge sort is applied in that matrix. This research work performed much efficient sorting algorithm with reduced complexity, but it is not always possible to modify input matrix (which is usually generated by another application) into a 2D matrix form, and even when it is possible, doing this will require a 3rd party application which will use processing power and need execution time to prepare the needed matrix.

(Garcia, et al., 2014) Presented the fast data parallel implementation of radix sort using the Direct Compute software development kit(SDK). The author also discussed in details the optimization strategies used to increase radix sort performance. The paper share the insights should be used in GPGPU (General Purpose Graphics Processing Unit). Testing results of this research showed great performance in sorting random datasets, but also gave very low performance in sorting both all zero and descending sorted datasets.

3. Methodology

This research aims at finding the best data-sort-search combination in a parallel processing environment. Thus comparing the performance of the device on suggested datasets was done over the Sorting and searching of the same dataset.

3.1 Dataset Creation

In the purpose of testing the sort algorithms a Dataset of 10 Million random generated Elements datasets was created by using a C++ code into a Text File beforehand, furthermore, this file was duplicated twice and once sorted in an Ascending order and another in Descending order to create all best case scenario, average scenario and worst case scenario to provide more accurate simulation for what might encounter the sorting algorithms.

3.2 Sort Comparison

The sorting Efficiency was measured through two performance indices: Elapsed time and speed up time.

The Elapsed Time:

Which the Algorithm ran for (Runtime) which was ran for four times for each dataset each of the 5 times onto 1, 8, 16, 32 and 64 processors respectively, yielding 20 results for each algorithm for 3 algorithms as such we obtain 150 variations which describe the variety of timings possible and the general performance of the algorithm.

The number of processors where chosen depending on data collection from IMAN1 server, where technicians orally stated that most applications do not get permission for more than 64 processors (thus it will be useless to apply case study on a larger number of processors), and that most permissions are usually for 16 or 32 processors (so that it was a must to include those numbers in the study). Choosing 1 processor is very helpful as it represents sequential runtime which will later help in calculating both speedup time and efficiency factors, while choosing 8 processors was to simulate multithreading (because a single PlayStation3 board like those used in IMAN1 supercomputer contains 8 processors on it, and this will give us good indication on time necessary to process the application internally without parallel computing function, and thus, later on, it can be very useful to calculate network traffic and overhead parameters in parallel computing).

Speed up Time:

The Ratio of Elapsed time of the algorithms running in parallel to its conventional sequential counterpart, which were obtained by running it onto one processor.

$$Speedup\ time = \frac{sequential\ runtime}{parallel\ runtime}$$

3.3 Search Comparison

The Searches were compared through elapsed runtime and were performed for elements in the top middle and bottom of the searched dataset and this was done for 1, 8, 16, 32 and 64 processors.

4. Algorithm Walkthrough

The proposed research performed the two basic operations of dataset management: Sorting and searching, in the multiprocessing environment (IMAN 1). Applied three basic sorting algorithms: Bubble sort, Quick sort, and merge sort. And also applied two basic searching algorithms: sequential search and Binary Search.

4.1 Sorting Algorithms

Through the appearance of parallel processing, parallel sorting has developed into an important area for algorithm research. A vast quantity of parallel sorting algorithms has been introduced.

The current trends of hardware development and innovations are oriented towards extensive usage of high-performance computations based on multicomputer and multiprocessor computer systems. Grid and cloud computing also pose the requirement for distributed data processing.

Bubble Sort:

Bubble sort is the simplest sorting algorithm available; its mechanism is as such:

1- Steps through the Dataset to be sorted.

2- Compares each pair of adjacent items.

3-swaps them if they are in the wrong order.

4-repeatfor n times, where n is dataset length.

Worst-case performance $\Theta(n^2)$

Best-case performance $\Theta(n)$

Average performance $\Theta(n^2)$

Every processor will execute $\frac{n/2}{p}$ comparisons for each inner loop. In this case, the complexity level of the algorithm will be $\Theta(\frac{n/2}{p})$

Quick Sort:

Quicksort is performed as follows:

1-Choosing a value is called as the pivot value

2- Partitioningthe input data set to two subsets

3- One is assigned data lesser than the pivot value and the other contains data greater than the pivot value.

4-Thesteps are repeated for each dataset till the dataset is sorted.

Worst-case performance $\Theta(n^2)$

Best-case performance $\boldsymbol{\Theta(n)}$

Average performance $\boldsymbol{\Theta(n \log n)}$

Merge sort

Merge sort is performed as follows:

1- Divide: split the Dataset into two, each having half the elements.

2- Conquer: recursively sorts the two subsets.

3- Combine: merging the two sorted subsets.

Worst-case performance $\boldsymbol{\Theta(n \log n)}$

Best-case performance $\boldsymbol{\Theta(n \log n)}$

Average performance $\boldsymbol{\Theta(n \log n)}$

In parallel Merge sort, input elements are split into equal segments and each segment is assigned to a processor after segmentation, all processors will perform sequential Merge sort to sort their assigned data

4.2 Search Algorithms

Sequential Search

A sequential search is a method for finding a target value within a Dataset. It sequentially checks each element of the Dataset for the target value until a match is found or until all the elements have been searched.

Worst-case performance $\Theta(n)$

Best-case performance $\Theta(1)$

Average performance $\Theta(n)$

A Sequential search is performed on each processor, each looking through a portion of the dataset and checking each value encountered.

Binary Search

A binary search or half-interval search algorithm finds the position of a specified input value (the search "key") within and a set sorted by key value (Oommen, 2014).

Worst-case performance $\boldsymbol{\Theta(\log n)}$

Best-case performance $\boldsymbol{\Theta(1)}$

Average performance $\boldsymbol{\Theta(\log n)}$

Binary search is one of the most common search algorithms Binary search is a more specialized algorithm than sequential search as it exploits sorted dataset. It splits the sorted dataset into multiple equal parts and to inspect the dataset at the partition point.

4.3 Parallel Algorithms

Sorting methods mentioned above are turned parallel through the divide and conquer technique. This Divides the Dataset into equal subsets and sends one to each of the processors; each Performs The sequential Algorithm on its given subset; Combines the results by passing messages between processors.

Initialize MPI environment.

Determine the number of MPI processes (p) and their id's.

%Divide Phase

If (id=master)

Read Dataset

Calculate partition size $S = \frac{N\ (Data\ Size)}{P\ (Number\ of\ processors)}$

Send S & N to all processors

%Conquer Phase

Invoke Algorithm

%Combine Phase

While (id < P &>0)

if(id is even)

Send even-Subset to process id + 1

Receive odd-Subset from processor id + 1

Merge even-Subset and odd-Subset into sorted-dataset

Replace even-Subset by the first half of sorted-dataset

else

Receive even-Subset from process id – 1

Send odd-Subset to process id- 1

Merge even-Subset and odd-Subset into sorted-dataset

Replace odd-Subset by the second half of sorted-dataset

end

End while

Finalize MPI environment

End

5. Results

Datasets were tested on the aforementioned device with the following technical specifications:

IMAN1 Zaina cluster is used in all calculations. Zaina Cluster in an Intel Xeon based computing cluster with 1 Gbit Ethernet interconnects. It compounds of two Dell PowerEdge R710 and Five HP ProLiant DL140 G3 servers (http://www.iman1.jo/iman1/):

Server: 7 Servers (Two Dell PowerEdge R710 and five HP ProLiant DL140 G3)

CPU per server: Dell (2 X 8 cores Intel Xeon) HP (2 X 4 cores Intel Xeon)

RAM per server: Dell (16 GB) HP (6 GB)

Total storage (TB): 1 TB NFS Share, and OS: Scientific Linux 6.4

5.1 Sorting Results

Bubble sort is the slowest of the three though of its high scalability however it takes on average between 300×10^3 and 700×10^3 µs. However, bubble sort, takes much more time than other sort algorithm as Quick Sort takes between 7389~3117 µs and 8329 ~ 2315 µs for merge Sort. Figure 1 shows Sorting scalability which is based on number of processors used in testing vs. speed up value.

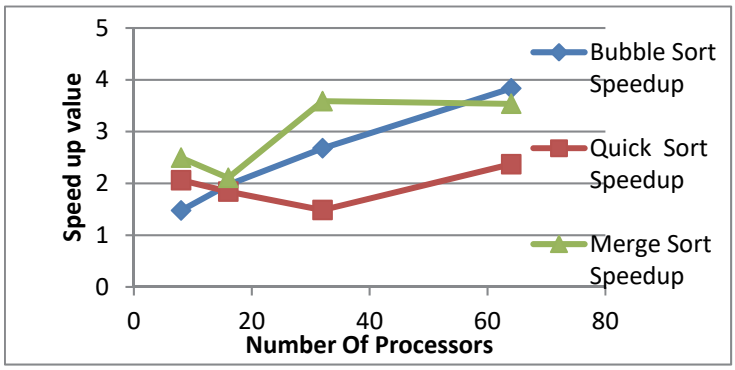

Figure 1. Sorting Scalability

Though Bubble sort can be faster, Merge sort is more stable as such it would be more efficient for most uses and its high scalability makes it a better algorithm for parallel processing if there is a high number of processors (to a limit).

In order to make a smart application that will always perform sorting in IMAN1 supercomputer in less time possible, one shall program both bubble sort and merge sort in it, and put a condition to check number of processors available for application to use. If this number is 32 or less, then the application shall activate merge sort function, else it shall activate bubble sort function.

5.2 Search Results

The Binary search method's time is highly dependent on the Sorting algorithm used, as sequential search on average is Faster than The Sorting Algorithms, as such Binary search is only faster if there are multiple searches within the same dataset. Figure 2 shows Searching scalability which is based on number of processors used in testing vs. speed up value.

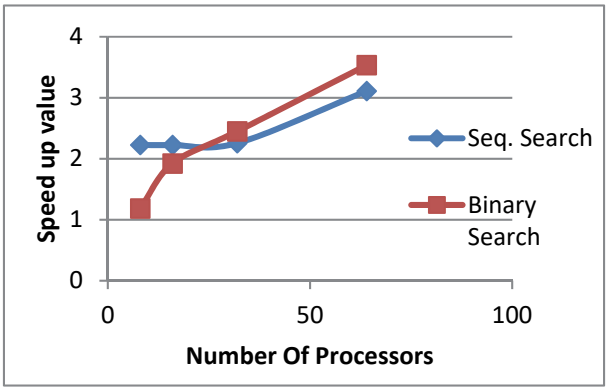

Figure 2. Search Algorithm Scalability

In order to make a smart application that will always perform searching in IMAN1 supercomputer in less time possible, one shall program both sequential search and binary search in it, and put a condition to check number of processors available for application to use, and a condition to check if the application target is to search for a single or multiple values in a data set. If application is searching for multiple values, then no matter what number of processors available, it shall activate binary search function. Else if it is searching for a single value, the application will consider number of available processors for it to use. If this number 16 or less, it shall activate sequential search algorithm, else it shall activate binary search algorithm.

6. Conclusion

The comparing of parallel Quicksort, parallel Merge sort, and parallel Bubble sort algorithms with varying number of processors in terms of running time and speed up showed that Merge sort is the most efficient of the three if number of processors available for the application is 32 or less, and that bubble sort becomes most efficient for higher number of processors. This indicates that a smart application shall have both algorithms programmed in it, and it can select which algorithm to activate depending on available number of processors for application to use.

Binary Search is much more efficient than sequential search if the dataset is to be reused to multiple searches due to binary search's requirement of a Sorted Dataset. But, sequential search is still more efficient under condition of having 16 processors or less, and searching for a single value in the dataset. This indicates that a smart application shall have both algorithms programmed in it, and it can select which algorithm to activate depending on number of values we need to search for and available number of processors for application to use.

The unique contribution in this paper can be used efficiently to increase applications efficiency by reducing runtime, and it can be used to schedule more applications to run on IMAN1 super-computer at same time without worrying about runtime needed for each application as those applications will calculate and use most efficient algorithms for given number of processors by the scheduling app. Of course, less operating time means faster results (which is the main goal of having a supercomputer in 1st place), besides saving more electricity power (as runtime becomes shorter), and the server can handle more applications in saved time. The only limitation for this, is making sure that all applications that will run on IMAN1 supercomputer are following the introduced study in their core code when needed.

References

Adikari, P. (2007). Review on Sorting Algorithms: A comparative study on two sorting algorithms. Mississippi state university, Mississippi.

Aliyu, M., & Zirra, P. B. (2013). A Comparative Analysis of Sorting Algorithms on Integer and Character Arrays. *The International Journal of Engineering and Science (IJES), 2*(7), 25-30.

Balaji, P., Buntinas, D., Goodell, D., Gropp, W., Hoefler, T., Kumar, S., Lusk, E., Thakur, R., & Traff, J. L. (2011). MPI on Millions of Core, ‖Parallel Processing Letter, 21(1).

Bharadwaj, Ashutosh, & Mishra, S. (2013). Comparison of Sorting Algorithms based on Input Sequences. *International Journal of Computer Applications, 78*(14), 7-10

Dominik, Z. et al. (2013). The comparison of parallel sorting algorithms implemented on different hardware platforms. *Computer Science, 14*(4), 679-691.

Garcia, A., Flores, J. O., Pinto, U. O., & Ramos, F. (2014). Fast Data Parallel Radix Sort Implementation in DirectX 11 Compute Shader to Accelerate Ray Tracing Algorithms. EURASIA GRAPHICS: International Conference on Computer Graphics, Animation and Gaming Technologies, 27-36.

Gropp, W., Huss-Lederman, S., & Lumsdaine, A. et al. (1998). MPI: The Complete Reference, the MPI-2 Extensions, vol. 2, The MIT Press.

Jeon, M., & Kim, D. (2002). Load-Balanced Parallel Merge Sort on Distributed Memory Parallel Computers. Proceedings of the IEEE International Parallel and Distributed Processing Symposium, IPDPS.02.

Jost, G., Jin, H., Mey, D., & Hatay, F. (2003). Comparing the OpenMP,MPI, and hybrid programming paradigm on an SMP cluster, in Proceedings of the 5th European workshop on OpenMP(EWOMP'03).

Kumari, N. K. S., & Chakraborty, S. (2015). A statistical comparative study of some sorting algorithms. *International Journal in Foundations of Computer Science & Technology (IJFCST), 5*(4).

Markidisa, S., Penga, I. B., Iakymchuka, R., Laurea, E., Kestorb, G., & Gioiosab, R. (2016). A Performance Characterization of Streaming Computing on Supercomputers. *Procedia Computer Science, 80*, 98-107.

Mohammad, Q. (2011). Embedding Binary Tree and Bus into Hex-Cell Interconnection Network. *Journal of American Science, 7*(12).

Mohammad, Q. (2011). Multilayer Hex-Cells: A New Class of Hex-Cell Interconnection Networks for Massively Parallel Systems. *International journal of Communications, Network and System Sciences, 4*(11).

Mohammad, Q. (2016). New Efficient Algorithm for Mapping Linear Array into Hex-Cell Network. *International Journal of Advanced Science and Technology, 90*.

Mohammed, Q. (2005). Embedding Linear Array Network into the tree-hypercube Network. *European Journal of Scientific Research, 10*(2), 72-76.

Mohammeda, A. S., Amrahovb, S. E., & Çelebic, F. V. (June 2017). Bidirectional Conditional Insertion Sort algorithm; An efficient progress on the classical insertion sort. *Future Generation Computer Systems, 71*, 102-112.

Mukul, P., Kumar, M., & Bhargava, S. (2014). GPU Matrix Sort (An Efficient Implementation of Merge Sort). *International Journal of Computer Applications, 89*(18), 9-11.

Oommen, A. (2014). Chanchal Pal Student, Dronacharya College of Engineering, Gurgaon, BINARY SEARCH ALGORITHM. *IJIRT, 1*(5). ISSN: 2349-6002.

Parrya, I., Carbullidob, C., Kawadab, J., Bagleyb, A., Senc, S., Greenhalghc, D., & Palmieric, T. (August 2014). Keeping up with video game technology: Objective analysis of Xbox Kinect™ and PlayStation 3 Move™ for use in burn rehabilitation. *Burns, 40*(5), 852-859.

Pasetto, D., & Akhriev, A. (2010). Comparative Study of Parallel Sort Algorithms. IBM Dublin Research Lab, Mulhuddart, Dublin 15, Ireland.

Saadeh, M., Saadeh, H., & Mohammad, Q. (2016). Performance Evaluation of Parallel Sorting Algorithms on IMAN1 Supercomputer. *International Journal of Advanced Science and Technology, 95*, 57-72.

Wesite (2017). Retrieved March 11, 2017, from http://www.iman1.jo/iman1/