OIRT Journal of Information Technology

Available at: <https://journals.otuirt.com/index.php/OJIT>

ORIGINAL RESEARCH ARTICLE

Performance Optimization of Sorting Algorithm using Processor Affinity in Java

Abbas Muhammad Rabi^{1*}, Etemi Josua Garba², Benson Yusuf Baha³, Yusuf Musa Malgwi², Mu'azu Dauda¹

¹Department of Computer Science, Federal University Dutse, Nigeria

²Department of Computer Science, Modibbo Adama University Yola, Nigeria,

³Department of Information Technology, Modibbo Adama University, Yola, Nigeria

Corresponding Author: Abbas Muhammad Rabi, Department of Computer Science, Federal University Dutse, Nigeria.
Email: mambas86@fud.edu.ng

Copyright: © the author (s), publisher, and licensee OIRT. It is an open-access article distributed under the terms of the Creative Commons Attribution Non-Commercial License, which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided there is a proper citation for the original work.

ABSTRACT

Background:

The processor affinity library in Java can switch ON all the processing cores available in a multi-processing environment. Using this feature will enable concurrent programmers to fully utilize the benefits of processing power available in multi-core processors. Therefore, this study aims to use processor affinity to develop four different frameworks that could optimize the efficiency of quick sorting algorithms on multi-core platforms.

Methods:

Benchmarking is the method used to carry out all the experiments and test the developed algorithm's efficiencies using all four frameworks. An Octa-core machine with eight (8) processing cores was used to develop and run the algorithms to measure their running times and compare their performances. JDK 12.0 is the version of Java used for development. An array data structure containing One Million Elements (1,000,000) was used as the preferred data structure.

Results:

The results obtained show that processor affinity can improve the performance of quick sorting algorithms by ensuring no processing core is idled during the computation of results. The results also show that processor affinity and Work-Stealing-Action perform similar functions by ensuring that available cores in a machine are fully utilized to handle tasks and improve performance. It was further found that the algorithm developed using the Executor Services framework outperformed all the three other frameworks implemented using Naïve, Fork-Join, and Sequential implementations.

Conclusion:

It was concluded that processor affinity improves the performances of all the four different implementations of quick sorts. It was also concluded that in the fork-join framework, Work-Stealing-Action performed by the worker-threads has an effect similar to that of affinity by ensuring that all the processing cores are fully utilized to improve performance. Further investigations can be carried out using machines with more processing cores in their CPU using a different data structure such as a Link List Array of the same or different data size to see whether the same or different conclusions can be drawn.

Keywords: Affinity, Algorithms, Efficiency, Performance, Processing core

Article History

Received: January 14, 2022

Revised: February 15, 2022

Accepted: February 19, 2022

INTRODUCTION

The processor affinity library in Java can switch ON all the processing cores available in a multi-processing environment. This feature will enable concurrent programmers to fully utilize the benefits of processing power available in multi-core processors (Nsour & Fasha, 2020). The nature of algorithms determines the performance benefit derived from multi-processor and how algorithms are being implemented. However, the performance improvement that could be achieved is limited by the fractions of the codes that can run parallel or concurrently on the machines. This relationship has been defined by Amdahl's law (Anil & Himansh, 2019; Ionel & Vasile, 2019; Techopedia, 2018; Thomas, 2004). Threads are an unavoidable feature of the Java programming language. They can be used to simplify the development of some complex programs; this can be done by changing the asynchronous codes into some more direct straight line of codes (Ginanjar, 2021; Göetz et al., 2006).

Additionally, threads are the easiest way to tap the computing power and enjoy the benefit of multi-core processing machines (Ganesh & Sondhi, 2018; Göetz et al., 2006; Rabiou et al., 2021). Therefore, as the power of multi-processors increases, developing concurrent software will become more critical. The same need for resource utilization, convenience, and fairness that motivated the development of processes to ensure the program's efficiency is also the same concern that motivated the development of threads; (Ganesh & Sondhi, 2018; Rabiou et al., 2018). They allow multiple streams of program control flow to occur within a given process. Threads often share some broad resources of a process, such as file handles and memory, with each thread having its stack, program counter, and local variable variables. They also provide a natural way of decomposition that enables the exploitation of hardware parallelism on multi-processor machines. Multiple threads within the same program can be scheduled to run simultaneously on multiple processing cores of CPUs (Göetz et al., 2006; Rabiou et al., 2018).

Java was developed by Gosling James and was released in early 1995 as one of the Object-Oriented Programming (OOP) Languages. Since then, Java has offered synchronized and multithreading methods for developers. The earliest version of Java involved some low-level primitives that help synchronize and monitor threads. After almost a decade, in 2004, a new

version 5 of Java was released, including some concurrency utilities. These utilities were designed to serve as the building block for creating concurrency classes and related applications. These classes offered many advantages for the developers. The Java language is being updated and released as the newer versions, with the latest one being Java JDK 12, which was released in March 2021 (Oracle, 2020; Rabiou et al., 2021). An improved version of Circle sort was proposed by anonymous (2020). The experimental approach was the main method used to benchmark the algorithm. The performance improvement was achieved by reducing the number of iterations in the original version. However, the complexity of this algorithm remains undetermined.

Five different versions of merge sorts were analyzed and compared based on the ability of these algorithms to utilize resources, delay, and the space covered by each of the algorithms. Field Programmable Gate Array (FPGA) was used to experiment. The results indicated that each of the algorithms could best fit the area of application in which it is found suitable (Lobo & Kuwelkar, 2020). Bezemer and Moses (2016) proposed improved shell sorts and insertions sorts by Bezemer and Moses (2016). An Experimental approach was used to measure and compare these algorithms' performances on the MDIS system by considering both the best and the worst-case scenarios. The results exposed the strength and weaknesses of each algorithm in different situations. However, it was concluded that the type of list to be sorted would determine the applicability of the algorithms. A Graphical User Interface (GPU) based sorting algorithms were developed using the experimental methods, and their performances were compared. Parallelization was achieved by utilizing the GPU. It was revealed that count sort was the most efficient of all the algorithms tested on the GPU. It was further revealed that the future must be carried out to refine the performance of these algorithms on GPU (Faujdar & Ghreya 2016).

With the continual growth in microprocessor processing cores, developers will need to focus more on concurrent programming to get the desired performance increase of their concurrent programs that could not be achieved in the past using older programming approaches or logically increasing the CPU clock rate. Today, different libraries and mechanisms are readily available in Java that can be

used to optimize the performance of algorithms and concurrent applications. However, despite the availability of these tools, studies that used processor affinity and recent concurrency tools to improve the performance of algorithms and contemporary applications are lacking in literature as most of the studies carried out to enhance their performance uses a different approach. Therefore, this paper used the processor affinity library and some of the concurrency mechanisms provided by Java to optimize the performance of four other implementations of quick sorting algorithms using octa-core machines.

METHODOLOGY

Data Structure:

The data structure specified for this paper's sorting algorithm is the array data structure. The data structure contains positive integers of 10 million elements used to carry out sorting and testing overheads in this paper. Several test runs were varied to try different array sizes and reduce the effects of background programs on the measured time.

Looping:

Having defined our data structure, the next step after specifying the data size of the array is creating a loop that takes two different parameters: The "seed" for generating random values and the required array size. After each iteration, the size of the "seed" remains the same, but the array size is varied.

Hardware Specification:

The following hardware specifications were used for experimentation and benchmarking in this paper. An octa-core computer with 2.0 GHz, CPU processing cores was used to measure the overhead costs on Windows 10, 64 bits (OS).

Software Specifications:

All concurrency mechanisms and affinity libraries used to develop frameworks and algorithms in this paper were provided by Java. They are thread pools used to create and manage threads, and some frameworks are used to synchronize threads and carry out task executions. Others are locks, atomic operations, counting semaphores, and condition variables.

Running times Measurements:

This paper used the `system.nanoTime()` method to measure running times during benchmarking. `System.nanoTime()` method returns a measured time in nanoseconds. this is then converted to microseconds for easy comparison.

RESULTS AND DISCUSSIONS

Running times of Affinity Quicksort-Sequent:

Here, running times of different quick-sorts developed using four frameworks were measured and compared using an octa-core machine. The results obtained are presented in Figure 1.

Figure1. Running times of Affinity-Quicksort-Sequent and Quicksort-Sequent

From Figure 1, it is noticeable that Affinity-Quicksort-Sequent performs better than Quicksort-Sequent tested on an octa-core machine. Considering the smallest array size (i.e. 1,000 elements) $0.03\mu s$ and $0.06\mu s$ were recorded. Therefore, Affinity-Quicksort-Sequent was two times faster than Quicksort-Sequent tested without affinity applied. This indicates that, out of eight processing cores available on quad-core, only four were utilized at the initial stage when the sorting started. At the end of the sorting processes, when the array size reaches 1,000,000 elements, running times of $49.95\mu s$ and $67.78\mu s$ were recorded with and without affinity, respectively. A difference of $67.78\mu s$

was obtained. This is considered significant improvements achieved with affinity applied.

Test runs of Affinity-Quicksort-ParNai and Quicksort-ParNai:

Similarly, here, the performance of Affinity-Quicksort-ParNai was tested with affinity and without affinity enabled. Again, the running times were measured and compared on an octa-core machine, as described in figure 2.

Figure 2. Running times of Affinity-Quicksort-ParNai and Quicksort-ParNai Sequent

From Figure 2, considering the smallest array size, running times of $0.14\mu s$ and $0.17\mu s$ were obtained on octa-core machines. A decrease of $0.03\mu s$ was recorded, making Affinity-Quicksort-ParNai 3% faster than Quicksort-ParNai when affinity was applied. When the array increased to 1,000,000 elements, a reduction in the running times of $8.14\mu s$ was obtained at the end of the sorting process, making Affinity-Quicksort-ParNai $8.14\mu s$ times faster than Quicksort-ParNai tested without affinity.

Test runs of Affinity-Quicksort-ForkJo and Quicksort-ForkJo:

Another parallel algorithm called Affinity-Quicksort-ForkJo was tested and compared with Quicksort-ForkJo without affinity being applied. Figure 3 describes the running times obtained.

Figure 3. Running times of Affinity-Quicksort-ForkJo and Quicksort-ForkJo

The results described in Figure 3 are different from the results obtained in Figure 1 and Figure in such a way that the running times of the two algorithms obtained in Figure 3 are slightly different in values. The slight difference in the running times is because work-stealing and affinity actions performed similar functions. However, they are used to fully utilize the processing power available in the multi-core machine. Work-Stealing-Action is done so that a worker-thread executes its task in a FIFO order. However, when it has no tasks left in its deque, it will try "stealing" tasks from other worker threads in a LIFO order. This idea of pooping its tasks in a FIFO order but stealing other tasks for pooping in a LIFO order makes the Fork/Join framework more efficient for sorting by ensuring that no core is left unused during the computation

processes. This same function is performed by multi-core affinity. Therefore, the combined effects of affinity and work-stealing action make the running times obtained in Figure 3 look similar.

Test runs of Affinity-Quicksort-Executive and Quicksort-Execut:

The results described in Figure 4 were obtained using a Quicksort-Execut framework tested on an octa-core machine. The framework used to build quick sort using executor solves problems differently from previously mentioned frameworks. For example, the programmer does not handle the creation and termination threads done during Naïve framework implementation.

Figure 4. Running times of Affinity-Quicksort-Execut and Quicksort-Execut

From Figure 4, it can be observed that the effects of processor affinity have improved the running times of Affinity-Quicksort-Execut. At the beginning of the sorting process, Affinity-Quicksort-Execut recorded $0.03\mu s$ as against $0.034\mu s$ recorded by Quicksort-Execut without affinity enabled. Therefore, it can be concluded that the effects of affinity at the beginning of the processes are not significant as compared with the effects measured with the previously mentioned algorithms. This is because the framework developed using the executor framework is very efficient on smaller array sizes regardless of the presence of affinity being applied. However, affinity effects become noticeable as the array size increases to 500,000 elements. At this point, running times of $14.12\mu s$ and $15.83\mu s$ were recorded with and without affinity, respectively. Therefore, Affinity-Quicksort-Execut is found to be $1.71\mu s$ times faster than Quicksort-Execut without affinity enabled. At the end of the computation process, when the array size increases to 1,000,000 elements, running times of $21.78\mu s$ and $24.47\mu s$ were obtained with and without using affinity, respectively. Thus, Affinity-Quicksort-Execut is found to be $2.69\mu s$ times faster than Quicksort-Execut when the process completes. This is considered a significant improvement obtained with affinity enabled.

Performance comparison of all the four frameworks:

To determine the best performing framework tested on octa-core machines when affinity was enabled, the running times of all the four algorithms developed using each of the optimizing frameworks was compared (Figure 5).

As seen in Figure 5, all the four algorithms developed using the four corresponding framework implementations were compared. Quick-sort-Executor outperforms the other three algorithms. Its performance becomes more noticeable, especially when the array sizes reach 60,000 elements. This good performance remained unchanged till the end of the sorting process. Its good performance is because the framework developed using executor-service can automatically handle the creations and assignments of threads without additional effort from the programmer's side. Thus, the executor handles threads creation, task splitting, and the processor core assignments in collaboration with OS and JVM. Quicksort-ForkJoin was found to be the second-best performing framework. Thus, its good performance became noticeably observed when the array size reached 300,000 elements. The good performance might not be unconnected because this framework uses a divide and conquer approach to sort a given array size. Another good performance is the work-stealing action between the worker threads that ensure no processing core remains idle during results computations. Quicksort-Par-Naïve was better than Quick-sort-Sequential, especially when the array gets to 70,000 elements. This is also because Quicksort-Par-Naïve solves a given problem in parallel, unlike the two previously mentioned algorithms. Thread-pool utilization is also another reason for its good performance. However, Quick-sort-Sequential performance becomes noticeably good with the smaller array of elements. As the array size increases, its performance starts falling till the end of the computation process.

Table 5. Performance comparison of all the four algorithms with an affinity enabled

CONCLUSION

Four different frameworks were developed using other concurrency mechanisms. Each of the frameworks was used to implement different versions of quick sorts. Their running times were measured and compared. It was concluded that processor affinity could optimize performances of quick sorts. It was further concluded that in a fork-join framework, Work-Stealing-Action performed by the worker-threads can perform similar functions to that of processor affinity by ensuring that no processing core remained idled during the computation of results to improve performance. Finally, it can be concluded that Affinity-Quicksort-Execut outperformed Affinity-Quicksort-ForkJo, Affinity-Quicksort-ParNai, and Affinity-Quicksort-Sequent when multi-core affinity was applied. Further investigations can be carried out using machines with more processing cores in their CPU using a different data structure such as a Link-List Array of the same or different data size to see whether the same or different conclusions can be drawn.

Conflict of Interest: None

Funding/Support: None

REFERENCES

- Anonymous (2018). Exact sorting algorithm. Available online at: <http://www.geocities.ws/p356spt/>, Retrieved on 23/07/2021.
- Bezemer, H., & Moses, O. (2016). A Variant of Modified Diminishing Increment Sorting: Circle sort and its Performance Comparison with some Established Sorting Algorithms. *International Journal of Experimental Algorithms (IJEa)*, 6(2), 2016, 1-8.
- Faujdar, N. & Ghrera, S. (2016). Performance analysis of parallel sorting algorithms using GPU computing. *International Journal of Computer Applications*, 12(2), 0975 – 8887.
- Ganesh, & Sondhi. (2018). An overview of multi-core. *International Journal of Innovative Science and Research Technology* 3(4), 261-263.
- Ginanjar, A. (2021). Visual Sorting: Visual paradigm implementation for the tree-based sorting algorithm. *International Journal of Computer Application* 8(18), 183-193.
- Götz, B., Peierls, T., Bloch, J., Bowbeer, J., David, H., & Lea, D. (2006). *Java concurrency in practice*. Addison: Wesley Professional, 1-10.
- Ionel, Z. & Vasile G. G. (2019). Improving the performance of CPU architectures by reducing the Operating System overhead. *Electrical, Control and Communication Engineering* 10(1), 1-25. DOI:10.1515/ecce-2016-0002
- Lobo, J. & Kuwelkar, S. (2020). Performance analysis of merge sort algorithms. *Proceedings of the International Conference on Electronics and Sustainable Communication Systems (ICESC 2020)*, 1-6.
- Nsour, E. & Fasha, M. (2020). Implementing parallel quick Sort algorithm on OTIS Hyper Hexa-Cell (OHHC) interconnection network. 1-33.
- Oracle. (2019). *Java SE 7 API specification document*. Retrieved from: <http://docs.oracle.com/javase/7/docs/api/>.
- Rabiu, A.M., Garko, A.B. & Abdullahi, A.M. (2018). Effects of multi-core processors on linear and binary sorting algorithms. *Dutse Journal of Pure and Applied Sciences* 4(2), 130-140. Available at <http://fud.edu.ng/journals.php>
- Rabiu, A.M., Garko, A.B., Abdullahi, A.M, Umar, H.A., & Babagana, M. (2018). Performance evaluation of three quick-sorting algorithms on single and multi-core processors. *Dutse Journal of Pure and Applied Sciences* 4(2), 254-263. Available at <http://fud.edu.ng/journals.php>
- Rabiu et al., (2021). Comparative Analysis Between selection sort and merge sort algorithms. *Nigerian Journal of Basic and Applied Sciences (NJBAS)*, 29(1), 48-53.
- Techopedia. (2017). *Amdahl's law*. Retrieved from: <https://www.techopedia.com/definition/17035/amdahls-law>.