International Journal of Mathematics and Computer Research

ISSN: 2320-7167

Volume 12 Issue 11 November 2024, Page no. – 4577-4585

Index Copernicus ICV: 57.55, Impact Factor: 8.316

DOI: 10.47191/ijmcr/v12i11.05

Parallel Sorting of Randomly Generated Grid Jobs on a Single-Processor System

Dr. Abraham Tomvie Goodhead[1], Kikile Vincent Samuel [2]

[1]Computer Science Department, Faculty of Science, Niger Delta University, Wilberforce Island, Bayelsa State, Nigeria.

[2]Mechanical Department, Faculty of Engineering, Federal University Otuoke, Bayelsa State, Nigeria

ARTICLE INFO

Published Online:
20 November 2024

Corresponding Author:
Dr. Abraham Tomvie Goodhead

ABSTRACT

Improvements in computer technologies continue to shape the presence and the future of modernization, driven by the need for faster and more efficient processing, most chip manufacturers have abandoned the single-processor system and turned attention to other hardware technologies like the multicore system. However, should the baby (single-processor system) be thrown away with the bathwater? Parallelization which defines the era of the multicore if properly exploited on single-processor systems can improve performance. This work exploits thread-level parallelism on the single processor system. This work uses thread-level parallelism to sort randomly generated grid jobs. The method randomly generates grid jobs which are then sorted into groups based on the computing requirements of the job. Using fuzzy rules, the sorting is done with a range of threads from one to eight in steps of two. For each set of sorting, the time of completion is recorded. The analysis shows that increases in the thread improve performance on the single processor system. However, as the number of jobs increases, the execution time also increases for all threads – indicating a general performance decline. The analysis also showed a steady improvement in performance as the number of threads increased from one to two and between two and four threads. However, the improvement leveled off at four threads and six threads and degraded between six threads and eight threads. This indicates that as the number of threads increases, the single processor system poses a bottleneck to performance due to context switches and other overheads. We therefore recommend that for thread-level parallelization on the single-processor systems, the number of threads should not be more than four.

KEYWORDS: Grid Computing, Parallel sorting, Single-processor system, Threads, Multi-threading.

1.0 INTRODUCTION

The overarching goal in computing has been dominated by speed-up and increased performance or throughput. Speed-up has been exploited using various computing designs while different computing models and programming paradigms have been exploited to achieve increased performance. Threads offer a platform for enhanced throughput and have been exploited for parallelism on different platforms.

The single-processor systems have long been touted as haven reached their performance threshold, and system designers and manufacturers have turned to more performance-promising models like the multicore technology which offers parallel processing. However, parallelisation using threads presents another window of performance increase sought after by computer enthusiasts, scientists and researchers.

Implementation of threads in performance speed-up can also be challenging as too many threads can throw up communication bottlenecks and overheads while too few threads would lead to the underutilization of resources.

This work examines the impact of threads-level parallelism on the performance of a single processor system. The work implements an algorithm that uses threads to sort randomly generated grid jobs

1.1 Background

Speculations about the inadequacies of single-processor systems meeting the computing needs of a fast-changing and data-driven world have long been discussed with the conclusion that the single-processor system has reached its performance threshold (Amdahl, 1967; Bridges et al., 2008; Sirhan, 2020) prompting system designers and manufacturers to adopt a better performancepromising model (Kalla, 2004; Wang et al., 2007b, Pandey, 2019). However, interests in performance enhancement in computing continue to dominate research interests (El-Moursy, 2005; Ungerer, 2002; & Wellein, 2006) and parallelization using thread presents another window of increased performance and presents a lifeline to single processor systems. The use of threads has severally been exploited (Madriles et al., 2009; Pandey, 2019). Threads provide a platform for enhanced throughput and have been exploited for parallelism in different research (Chen, 2002; Haji, 2021; Sharif, 2020) this trend is expected to continue as new parallelization effort emerges (Vachharajani et al., 2005).

This work exploits a varying range of threads to sort a randomly generated grid job. The aim is to ascertain the performance-improving effect of threads on a single processor system.

2.0 REVIEW OF LITERATURE

Performance Improvement Measures in Single-Processor Design

The exponential demand for performance improvement in computing doesn't only necessitate hardware improvement but also an improved programming practice (Eck, 2021; McCool, 2008) However, it has been demonstrated that sequential programs do not optimize system utilization during execution (Stone, 2010) necessitating a rethink in the programming paradigm (Debbi, 2019). Parallel programming has been exploited variously to attain system optimization (Zhong, 2007; Asanovic, 2009; Debbi, 2019; & Valencia, 2019;). This is the motivation for the use of threads in this work.

The need for processor speed, efficiency and optimal performance has motivated several researchers in computing (Debbi, 2019; Kalla, 2004; Schauer, 2008; Valencia, 2019; & Wang et al., 2007). Researchers had sought an increase in gate density on the integrated circuit technology (Lacoe, 2008; Nair, 2002) thereby increasing component and total costs. Therefore, the need to adequately utilize the capabilities of the technology became imperative. This prompted (Olukotun et al., 1996) to undertake a study that revealed the possibility of implementing a single-chip multiprocessor on the same area as a superscalar, the work also revealed that where the need for parallelism is negligible, performance between the two technologies almost evens out.

Another effort to improve processor performance led (Sprangle et al., 2002; Sprangle & Carmean, 2002) to exploit the deep pipelining method on a Pentium ® 4 processor-like architecture. They observed that branch misprediction inhibited performance and stressed the need for branch prediction and fast branch recovery. Before this, (Bae et al., 2003) designed the Single Chip Programmable Platform (SPP) that employed multithreading on the RISC processor to improve concurrent processing and communication in the platform-based design of microprocessors. The platform integrates hardware blocks used in the design of embedded system chips.

Similarly, to achieve a fine-grain thread-level parallelization, Thread-level speculation (TSL) was exploited by (Prabhu, 2003) to enhance performance on seven spec CPU2000 benchmark applications. The work presented various methods for manual parallelization and attained a speedup of 110% and 70% respectively for TSL parallelization on four floating points applications and three integer applications.

Noting the importance of optimization of the serial code in improving performance (Wellein, 2006) evaluated the performance of the single processor systems and its impact on architecturedependent implementations on the Lattice Boltzmann kernel, the researchers also shed light on the relevance of commodity off-the-shelf (COTS) architecture and Vector processors. Likewise, (Inoue, 2007) implemented the aligned-access (AA-sort) parallel sorting algorithm, the method eliminates unaligned memory access to increase performance. However, this work was targeted at the multi-core systems.

Following the success of previous researchers on parallelization using threads, (Sharif, 2020) proposed and designed a professional integrated operating system performance measuring system to measure processes and threads. The system was capable of controlling the CPU, measuring execution time, CPU time, user time, kernel time and context switches, the system was applied to several processing environments.

2.1.0 The GRID

The Grid is a computing platform that delivers computing services from various independent computing sites to users in disparate locations, the paradigm requires large-scale sharing and proper service delivery to meet user's needs. To attain these requirements, the Grid encourages the integration and aggregation of different federating computing units to create a virtual organization of grid networks so that it can deliver the Quality of Service required by customers (Foster, 2000; Foster & Kesselman, 1999; Wieczorek, 2009).

2.1.1 Sorting Requirement on the Grid

Sorting is a fundamental activity in computing, improving efficiency in sorting will directly improve performance and throughput. Researchers have exploited several methods aimed at improving sorting. (Debbi, 2019) assessed the possibility of parallelizing sorting algorithms and compared improvements between various sorting algorithms. (Kristo, 2020) exploited a distributed sort algorithm that uses a learning model of the CDF of data empirically, then applied a deterministic sorting algorithm to establish a total sort order and achieved a 3.38 improvement. Other researchers have exploited other methods to improve sorting (Norollah, 2019;

& Vasanth et al. 2019). This work aims at exploiting parallelism in sorting grid jobs.

The importance of sorting is fundamental to the computing grid and has inspired researchers to exploit various methods to improve efficiency (Barthel, 2023; Rajeswari, 2019). Improvement in grid sorting is a necessity (Coleman, 2022) for the grid to attain its large-scale aggregation, sharing and quality of services. This demand necessitates the exploitation of thread for efficiency in the execution of processes on the grid.

2.2.0 Grid Scheduling Algorithms

Jobs on the computer Grid are of disparate requirements, they are submitted by users with varying needs and grid sites for processing jobs are also made up of varying computing capacities. These requirements may include heuristics, network bandwidth, blind matching and QoS (Agarwal, 20011; Kobra, 2007; Nasir et al., 2012). Several parallelisation attempts have been made to parallelize grid scheduling (Abraham, 2016; Abraham et al., 2015a, 2015b, 2021b, 2021a). In consideration of these factors, this implementation seeks to improve the performance of grid jobs on a single processing system using a random sorting algorithm.

The remainder of this article is as follows: section 3 discusses the experimental design, and the flowchart detailing the sort criteria for the work. Section 4 discusses the experiments, Tests, Data and Analysis of data on the single processor system using varying numbers of threads. Section 5 discusses the conclusion.

3.0.0 EXPERIMENTAL DESIGN

This experiment was designed to sort jobs into various files based on some attributes of the job. Sorting was done in parallel using Java threads - java threads offer basic programming parallelism and also possess the ability to scale up with an increased number of threads and processors. The system executes the primary process of sorting different numbers of jobs using one thread, two threads, four threads, six threads and eight threads.

In the execution, a defined range of jobs are generated with random attributes, these sets of jobs are then sorted into various files based on the attributes of the jobs. The various attributes of a job are ID, SIZE, PLATFORM and PRIORITY.

Sorted jobs are written into four different files representing each set of jobs grouping (i.e. priority group). The four output files for sorted jobs hold jobs with VERY HIGH PRIORITY, HIGH PRIORITY, MEDIUM PRIORITY and LOW PRIORITY.

The ranges of jobs generated and used for the experiment are from 10000 to 800000 (i.e. from ten thousand to eight hundred thousand) in steps of 50000 (except the first step which was 40000). For each step in the range of jobs generated, the program used one thread, two threads, four threads, six threads and eight threads to do the sorting in parallel.

Hence, the numbers of jobs randomly generated and sorted are 10000, 50000, 100000, 150000, 200000, 250000, 300000, 350000, 400000,450000, 600000, 650000, 700000, 750000, 800000 and the numbers of threads used for each execution are one, two, four, six and eight threads.

The number of jobs generated and sorted in each execution is equally divided by a division algorithm among the number of threads.

For each set of executions, the corresponding time of execution (computer system time in Milliseconds) is written to a file. Every step of the number of jobs in the experiment is executed ten times by each set of threads, and then the average for each set of jobs by each set of thread(s) is calculated.

The output file for timing has the following: Number of Jobs, Number of Thread, Time of execution by each thread, Total time of execution- which is a total of all the time used by each thread in the execution.

CONFIGURATION OF MACHINES USED FOR THE EXPERIMENT

The configuration of the computer on which the experiment was executed is as follows:

A Single-processor system, Pentium4, CPU speed 3.00GHz, RAM size 1.5 GB, running on Microsoft Windows XP 2002 version.

Sorting Criteria

The sorting criteria are based on Job Size, Computational Requirement and Deadline. Various combinations of these attributes are used to determine the group to sort jobs into. very large jobs with high computational requirements and critical deadlines are sorted into group A. Large jobs with high computational requirements but with medium critical deadlines are sorted to group B, Medium size jobs with medium computational requirements without critical deadlines are sorted to group C while the rest are sorted to group D. Fig.1 shows the stripped-down version flowchart for the Job Sorting criteria.

Sorting algorithm

Job sorting algorithm

Check size_of_job, Computational requirement and Deadline requirement of jobs

```
3.      If Job.Size =Very_Large & Comp.Req = Very_High and
Deadline=Critical
4.      Then sort the job to GroupA
5.      End_if
6.      If Job.Size =Large & Comp.Req=Medium &Deadline= MediumCritical
7.      Then sort the job to GroupB
8.      End_if
9.      If Job.Size=Medium & Comp.Req =Medium & Deadline=Not_Critical
```

```
10.        Then sort the job to GroupC
11.        Else
 13.   sort job to group D
14. End_while
 16. Stop.
```

3.1.0 Flow Chart for the Job Sorter

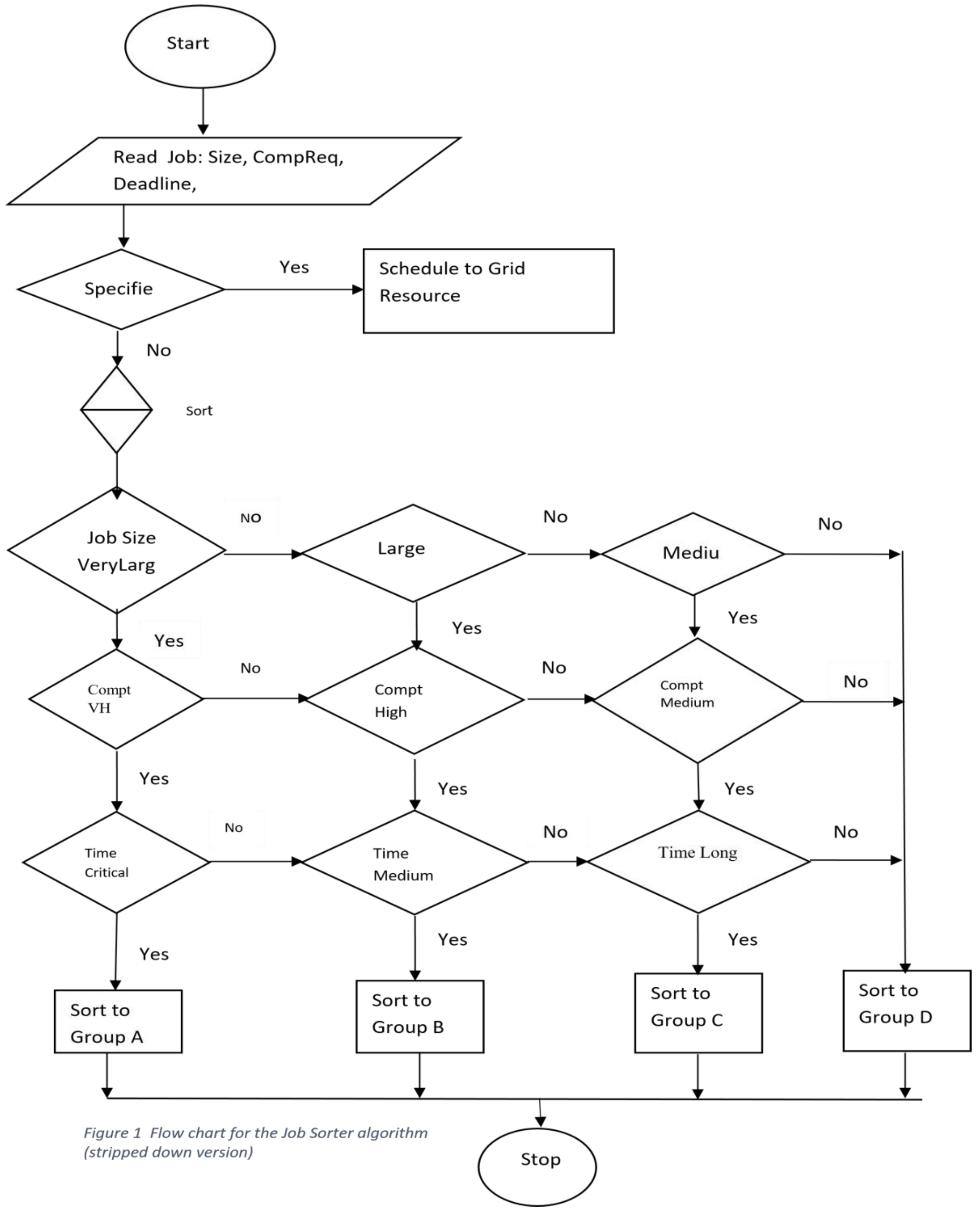

Figure 1 Flow chart for the Job Sorter algorithm (stripped down version)

4.0 EXPERIMENTS, TESTS, DATA AND ANALYSIS OF THE EFFECTS OF PARALLELISM USING DIFFERENT THREADS

This set of tests was aimed at how an increase in threads affects the performance of the sorting algorithm on a single processor system.

(i) **The effects of threading on the** Single **processor machine**

Result: There was a very strong correlation coefficient between the results among all the threads. the result for single processor systems reveals a minimal improvement as the number of threads increases. The trend levelled up at four threads. Table 1 shows the number of jobs sorted and the average completion time by the different threads.

Table 1 Thread Completion times for the Single Processor system

Jobs coun	OneThrea	TwoThrea	FourThre	ı SixThread	EightThread	Averag
10000	284.09	167.55	109.55	116.55	133.55	
50000	211.45	196.09	201.91	212.82	198.82	
100000	470.18	535.36	400.55	430.45	394.91	
150000	751.36	816.73	619.45	644.91	671.91	
200000	1211.64	941.82	934.64	821.09	776.82	
250000	1477.36	1134.91	1175	998.55	995.55	
300000	1534.09	1477.18	1321	1337.91	1379.09	
350000	1982.91	1662	1451.82	1378.91	1569.36	
400000	2083.91	2154.82	1707.36	1998.45	1652	
450000	2261.36	2276.91	2022.91	2018.55	1850.82	
500000	2500	2195.91	2022.82	2294.09	2186.18	
550000	2671.91	2308.27	2340.73	2276.82	2225.55	
600000	3004.18	3068.09	2515.64	2444.64	2505.73	
650000	3195.91	3277	3092.27	2808.18	2765.45	
700000	3541.09	3375.09	2964.36	3123.45	3076.82	
750000	3799.82	3481.45	3461.55	3412.09	3410.27	
800000	3950.27	3836.82	3643.64	3701.64	3437.55	

This test shows that for each set of jobs sorted, one thread takes more average time in sorting. This is followed by two threads, four threads then six threads. From the bar graph Also, from the line graph in Figure 3, all the threads exhibited the same characteristics, it shows a steady increase in execution time as the number of jobs increases for all the threads. Though this was to be expected, the difference (Figure 2) much cannot be deciphered between four threads, six threads and eight threads.

between the number of threads and the execution was not decipherable – indicating that the impact of threading on the single processor systems is minimal using this method of analysis. Hence, we shall employ another analytic measure.

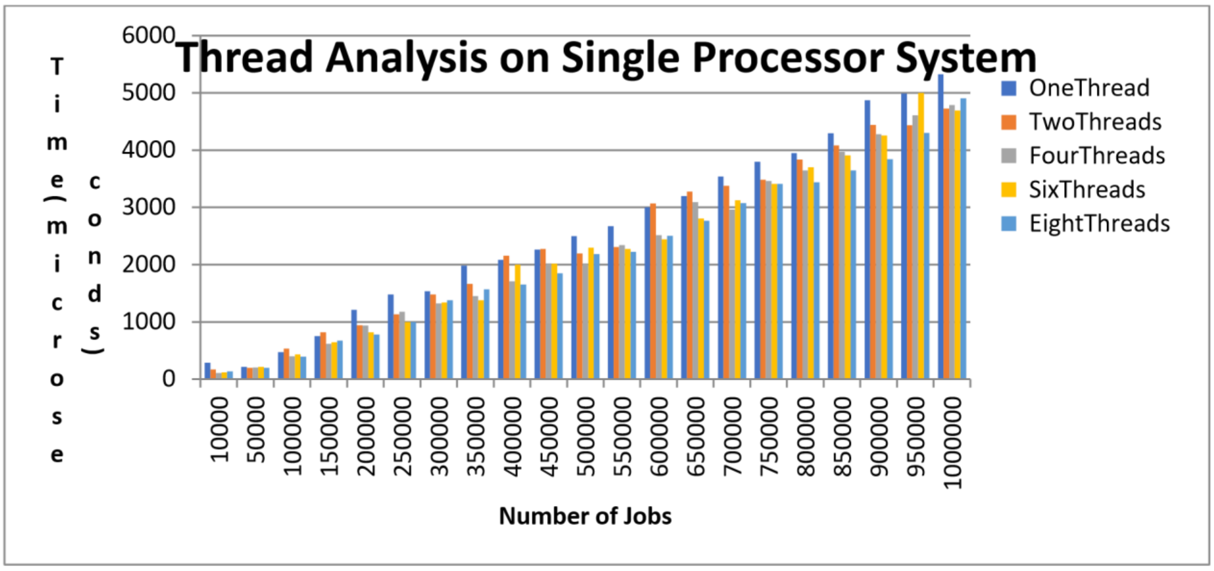

Figure 2 Bar chat representing completion times

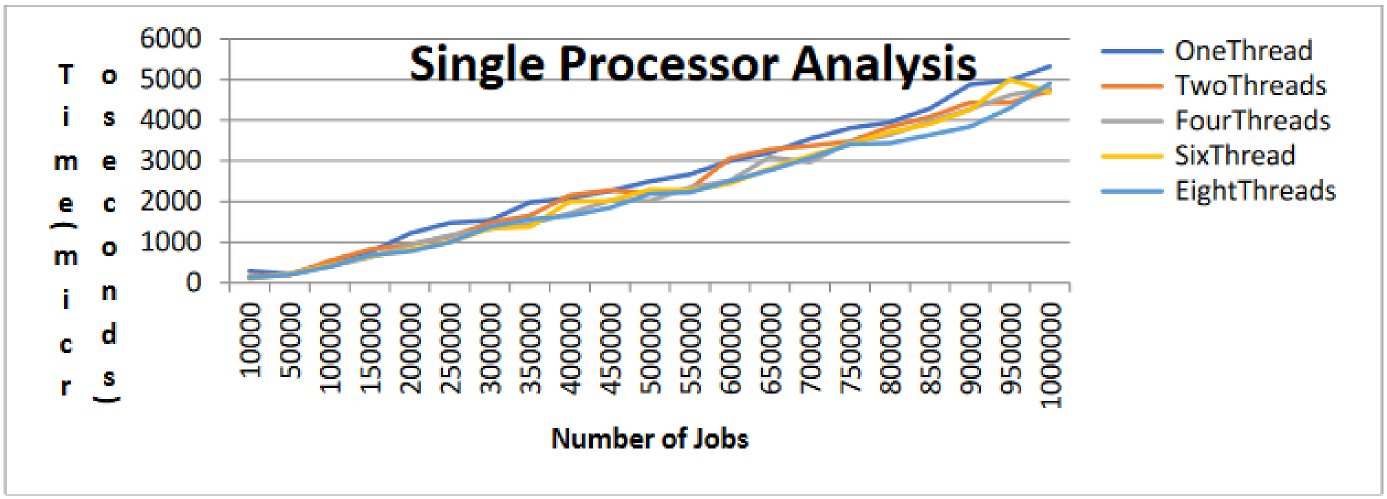

Figure 3 Line chat representing completion time by threads

(ii) **Performance improvement by threads on the Single processor machine using percentage measure**

Since the line graph could not indicate much difference in performance between the various threads, we computed the average performance in percentage. Figure 4 indicates the performance of each thread in percentage. One thread used 35% average time to sort the range of jobs. Two threads used a 21% average time to sort the range of jobs. Four threads used 14% average time to sort the range of jobs. Six threads used 14% average time to sort the range of jobs while eight threads used 16% average time to sort the range of jobs.
This shows that there was a significant performance gain between one thread and two threads, and there was also a significant performance improvement between two threads and four threads. But there was no performance improvement between four threads and six threads and finally, there was a performance degradation between six threads and eight threads. This indicates that beyond
4 threads, there was no longer any improvement. Also, as the number of threads increases, the single processor system becomes a bottleneck due to context switches and other overheads. We therefore propose that for thread-level parallelisation in single processor systems, the number of threads should not be more than four.

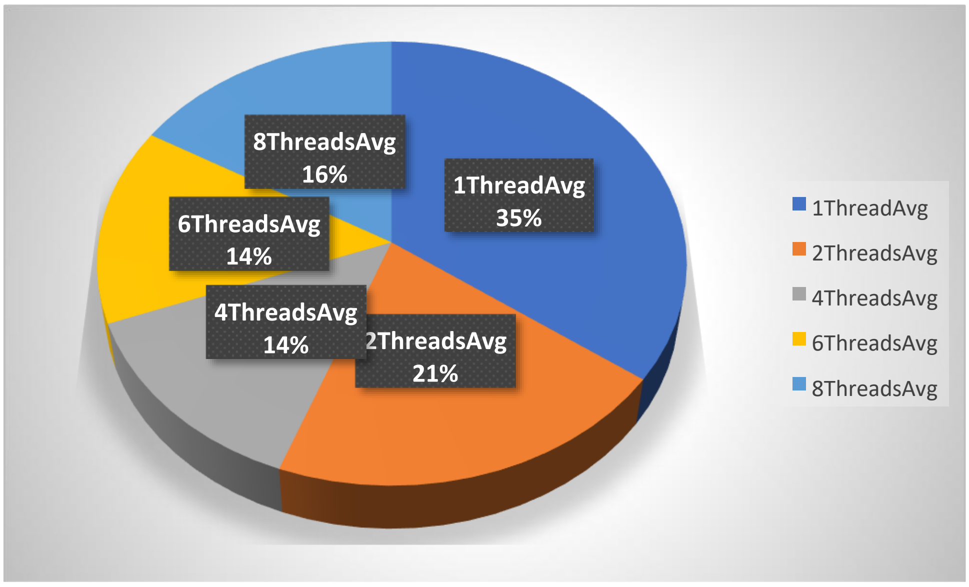

Figure 4. Percentage performance by different threads

Performance Difference between Successive Threads in Percentage

We computed the percentage difference (improvement) between successive threads. See Table 2. There was a 5.7% difference in improvement between two threads and one thread. There was an 8.8% difference in improvement between four threads and two threads. The percentage difference in improvement between six threads and four threads was negative; this is because the general performance degraded from this point. For 8 threads, we computed the performance improvement of 4 threads over 8 threads and that of 6 threads over 8 threads because both performed better. For this, the improvements were approximately the same (See Figure 5). This indicates that beyond 4 threads, there was no longer improvement in performance.

Table 2 Performance Difference between successive Threads in Percentage

1ThrdAggAvg	2Thrd&1Thrd	4Thrd&2Thrd	6Thrd&4Thrd	8Thrd&6Thrd	8Thrd&4Thrd
34931.53	32906	29985.2	30019.1	29230.38	
	5.7986	8.8762	-0.1131	2.6274	2.5173

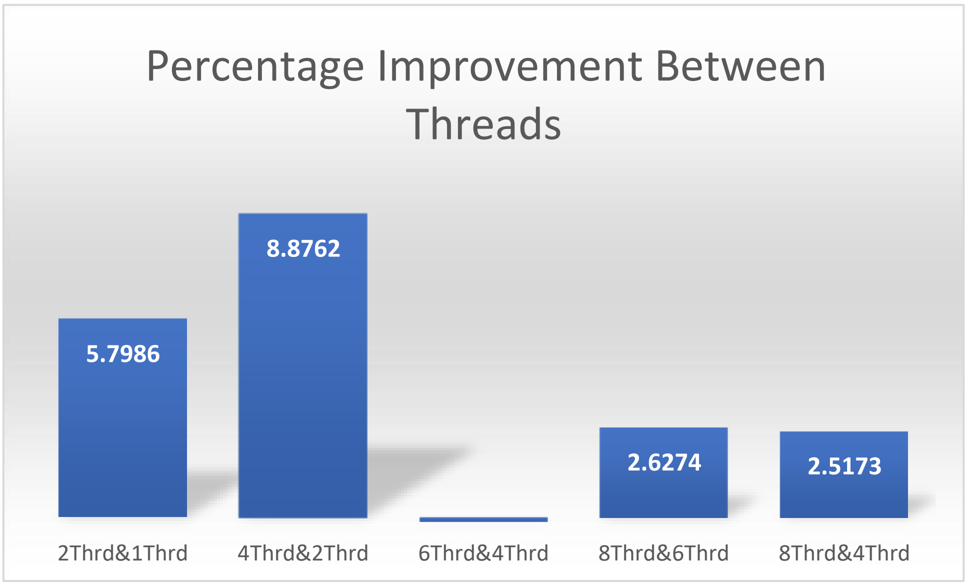

Figure 5 Difference in percentage improvement between successive threads

5.0 CONCLUSION

The result and analysis show that increases in the thread improve performance on the single processor system. However, the rate of increase in execution time against the number of jobs increased for all the threads – indicating that as the number of jobs increased, the general performance decreased.

The analysis also showed a steady improvement in performance as the number of threads increased from one to two and between two and four threads.

However, the improvement leveled off at four threads and six threads and degraded between six threads and eight threads.

This indicates that as the number of threads increases, the single processor system poses a bottleneck to performance due to context switches and other overheads.

We recommend that for thread-level parallelization in single processor systems, the number of threads should not be more than four for the sorting of grid jobs.

Future Thoughts

This work concentrated on the single processor system, in the future, we will exploit the same method on different sets of multicore systems, and then compare the performance against the single processor system.

REFERENCES

1. Abraham, G. T. (2016). Group-based parallel multi-scheduling methods for grid computing. Coventry University.
2. Abraham, G. T., James, A., & Yaacob, N. (2015a). Group-based Parallel Multi-scheduler for Grid computing. Future Generation Computer Systems, 50, 140–153. https://doi.org/10.1016/j.future.2015.01.012
3. Abraham, G. T., James, A., & Yaacob, N. (2015b). Priority-grouping method for parallel multi-scheduling in Grid. Journal of Computer and System Sciences, 81(6), 943–957. https://doi.org/10.1016/j.jcss.2014.12.009
4. Abraham, G. T., Osaisai, E. F. and, & Dienagha, N. (2021a). Parallel Scheduling of Grid Jobs on Quadcore Systems Using Grouping Methods. Asian Journal of Research in Computer Science, 8(4), 21–34. https://doi.org/10.9734/ajrcos/2021/v8i430207
5. Abraham, G. T., Osaisai, E. F., & Dienagha, N. S. (2021b). Parallel scheduling of grid jobs on quadcore systems using grouping methods. Asian Journal of Research in Computer Science, 8(4), 21–34.https://doi.org/10.9734/AJRCOS/2021/v8i430207
6. Agarwal, Amit. , & K. Padam. (20011). Multidimensional Qos oriented task scheduling in

grid environments. International Journal of Grid Computing & Applications (IJGCA), 2(1), 28–37.

7. Amdahl, G. M. (1967). Validity of the single processor approach to achieving large-scale computing capabilities. AFIPS Conference Proceedings - 1967 Spring Joint Computer Conference, AFIPS 1967, 483–485.
8. Asanovic, K., B. R., D. J. , K. T. , K. K. , K. J. , and Y. K. (2009). A View of the Parallel Computing Landscape. Communications of the ACM, 52(10), 56–67.
9. Bae, Y., Park, S., Circuits, I. P.-I. J. of S.-S., & 2003, undefined. (2003). A single-chip programmable platform based on a multithreaded processor and configurable logic clusters. Ieeexplore.Ieee.OrgYD Bae, SI Park, IC ParkIEEE Journal of Solid-State Circuits,2003•ieeexplore.Ieee.Org,38(10). https://doi.org/10.1109/JSSC.2003.817259
10. Barthel, K. U., H. N., J. K., & S. K. (2023). Improved evaluation and generation of grid layouts using distance preservation quality and linear assignment sorting. Computer Graphics Forum, 42(1), 261–276.
11. Bridges, M. J., Vachharajani, N., Zhang, Y., Jablin, T., & August, D. I. (2008). Revisiting the sequential programming model for the multicore era. IEEE Micro, 28(1), 12–20. https://doi.org/10.1109/MM.2008.13
12. Chen, Y. K., D. E., L. R., H. M. J., & Y. M. M. (2002, December). Evaluating and improving performance of multi-media applications on simultaneous multithreading architectures. IEEE International Conference on Parallel and Distributed Systems.
13. Coleman, J., & M.-P. O. (2022). Robotic Sorting on the Grid. In Proceedings of the 23rd International Conference on Distributed Computing and Networking, 26–30.
14. Debbi, A. E., H. A. F., & B. H. (2019). Would it be Profitable Enough to Re-adapt Algorithmic Thinking for Parallelism Paradigm? 2nd International Conference on New Trends in Computing Sciences (ICTCS), 1–6.
15. Eck, D. J. (2021). Introduction to programming using Java. https://biblioteca.unisced.edu.mz/handle/123456789/1574
16. El-Moursy, A., G. R., A. D. H., & D. S. (2005). Partitioning multi-threaded processors with a large number of threads. In IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005. . 112–123.
17. Foster, I. (2000). Internet computing and the emerging grid. Nature Web Matters, 7.
18. Foster, I., & Kesselman, C. (1999). "The Grid: Blueprint for a new computing infrastructure." Morgan Kaufmann.
19. Haji, L. M. , Z. S. R. , A. O. M. , S. M. A. , S. H. M. , & A. A. (2021, June). Performance Monitoring for Processes and Threads Execution-Controlling. In (pp. 161-166). IEEE. International Conference on Communication & Information Technology (ICICT).
20. Inoue, Hiroshi. , T. Moriyama. , H. K. and T. N. (2007). AA-sort: A new parallel sorting algorithm for multicore SIMD processors. 16th International Conference on Parallel Architecture and Compilation Techniques (PACT 2007), 189–198.
21. Kalla, R. , S. B. , & T. J. M. (2004). IBM Power5 chip: A dual-core multithreaded processor. IEEE Micro, 24(2), 40–47.
22. Kobra, E. & N. P. M. (2007). A min-min max-min selective algorithm for grid task scheduling. In 2007 3rd IEEE/IFIP International Conference, 1–7.
23. Kristo, A. , V. K. , Ç. U., M. S., & K. T. (2020). The case for a learned sorting algorithm. . 2020 ACM SIGMOD International Conference on Management of Data, 1001–1016.
24. Lacoe, R. C. (2008). Improving integrated circuit performance through the application of hardness-bydesign methodology. IEEE Transactions on Nuclear Science, 55(4), 1903–1925.
25. Madriles, C., López, P., Codina, J. M., Gibert, E., Latorre, F., Martinez, A., Martinez, R., & Gonzalez, A. (2009). Boosting single-thread performance in multi-core systems through fine-grain multithreading. ACM SIGARCH Computer Architecture News, 37(3), 474–483. https://doi.org/10.1145/1555815.1555813
26. McCool, M. D. (2008). Scalable programming models for massively multicore processors. . Proceedings of the IEEE, 96(5), 816–831.
27. Nair, R. (2002). Effect of increasing chip density on the evolution of computer architectures. IBM Journal of Research and Development, 46(2.3), 223–234.
28. Nasir, S., Shah, M., Mahmood, A. K., Kamil, A., Mahmood, B., Oxley, A., & Zakaria, M. N. (2012). QoSbased performance evaluation of grid scheduling algorithms. Ieeexplore.Ieee.OrgSNM Shah, AKB Mahmood, A Oxley, MN Zakaria2012 International Conference on Computer & Information Science, 2012•ieeexplore.Ieee.Org. https://doi.org/10.1109/ICCISci.2012.6297118
29. Norollah, A. , D. D. , B. H. , & F. M. (2019). RTHS: A low-cost high-performance real-time hardware sorter, using a multidimensional sorting algorithm.

IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 27(7), 1601–1613.

30. Olukotun, K., Nayfeh, B. A., Hammond, L., Wilson, K., & Chang, K. (1996). The case for a single-chip multiprocessor. SIGPLAN Notices (ACM Special Interest Group on Programming Languages), 31(9), 2–11. https://doi.org/10.1145/248209.237140
31. Pandey, R., & B. N. (2019, March). Understanding the role of parallel programming in multi-core processor based systems. In Proceedings of 2nd International Conference on Advanced Computing and Software Engineering (ICACSE).
32. Prabhu, M. K., & O. K. (2003). Using thread-level speculation to simplify manual parallelization. The Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 1–12.
33. Rajeswari, D. , P. M. , & S. J. (2019). Computational grid scheduling architecture using MapReduce model-based non-dominated sorting genetic algorithm. Soft Computing, 23(18), 8335–8347.
34. Schauer, B. (2008). Discovery Guides Multicore Processors - A Necessity. In ProQuest discovery guides. http://www.netrino.com/node/91
35. Sharif, K. H. , Z. S. R. , H. L. M. , & Z. R. R. (2020). Performance measurement of processes and threads controlling, tracking and monitoring based on shared-memory parallel processing approach. 2020 3rd International Conference on Engineering Technology and Its Applications (IICETA), 62–67.
36. Sirhan, N. N. (2020). Multi-core processors: concepts and implementations. Available at SSRN 3628131.
37. Sprangle, E., & Carmean, D. (2002). Increasing processor performance by implementing deeper pipelines. Conference Proceedings - Annual International Symposium on Computer Architecture, ISCA, 25–34. https://doi.org/10.1145/545214.545219
38. Sprangle, E., News, D. C.-A. S. C. A., & 2002, undefined. (2002). Increasing processor performance by implementing deeper pipelines. Dl.Acm.Org. https://dl.acm.org/doi/abs/10.1145/545214.545219
39. Stone, J. E., G. D., & S. G. (2010). OpenCL: A parallel programming standard for heterogeneous computing systems. Computing in Science & Engineering, 12(3), 66.
40. Ungerer, T. , R. B. , & Š. J. (2002). Multithreaded processors. The Computer Journal, 45(3), 320–348.
41. Vachharajani, N., Iyer, M., Ashok, C., Vachharajani, M., August, D. I., & Connors, D. (2005). Chip multiprocessor scalability for single-threaded applications. ACM SIGARCH Computer Architecture News, 33(4), 44–53. https://doi.org/10.1145/1105734.1105741
42. Valencia, D., & A. A. (2019). A real-time spike sorting system using parallel OSort clustering. . IEEE Transactions on Biomedical Circuits and Systems, 13(6), 1700–1713.
43. Valencia, D., biomedical, A. A.-I. transactions on, & 2019, undefined. (n.d.). A real-time spike sorting system using parallel OSort clustering. Ieeexplore.Ieee.Org. Retrieved November 14, 2023, fromhttps://ieeexplore.ieee.org/abstract/document/8869918/
44. Vasanth, K., Sindhu, E., Microsystems, R. V.-M. and, & 2019, undefined. (n.d.). VLSI architecture for Vasanth sorting to denoise image with minimum comparators. Elsevier. Retrieved November 14, 2023,fromhttps://www.sciencedirect.com/science/article/pii/S0141933119302522
45. Wang, P. H., Collins, J. D., Chinya, G. N., Jiang, H., Tian, X., Girkar, M., Yang, N. Y., Lueh, G. Y., & Wang, H. (2007). EXOCHI: Architecture and programming environment for a heterogeneous multi-core multithreaded system. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 156–166. https://doi.org/10.1145/1250734.1250753
46. Wellein, G. , Z. T. , H. G. , & D. S. (2006). On the single processor performance of simple lattice Boltzmann kernels. Computers & Fluids, 35(8–9), 910–919.
47. Wieczorek, M. , H. A. , P. R. (2009). Towards a general model of the multi-criteria workflow scheduling on the grid. Future Generation Computer Systems, 25(3), 237–256.
48. Zhong, H., L. S. A., & M. S. A. (2007). Extending multicore architectures to exploit hybrid parallelism in single-thread applications. IEEE 13th International Symposium on High-Performance Computer Architecture, 25–36.