A Parallel Triple-Pivot Sorting (PTPSort) Algorithm: Preliminary Results

Apisit Rattanatranurak
Dept. of Computer Eng., Faculty of Industrial Technology
Suan Sunandha Rajabhat University
Bangkok, Thailand
Email: apisit.ra@ssru.ac.th

Surin Kittitornkun
Dept. of Computer Eng., Faculty of Engineering
King Mongkuts Institute of Technology Ladkrabang
Bangkok, Thailand
Email: surin.ki@kmitl.ac.th

***Abstract*—Parallel or multithreaded sorting algorithms can be useful for data science/analytics and other applications for manycore CPU systems. A Parallel Triple Pivot Sort (PTPSort) is devised based on the Hoare's partition algorithm. The input array is initially partitioned with two pivots, P_{Lo} and P_{Hi} in parallel with two threads. Subsequently, the middle pivot, P_{Mi}, is applied resulting in approximately two halves. Finally, four subarrays can be obtained from two independent threads partitioning each leftover half with P_{Lo} and P_{Hi}, respectively. The process is recursively forked as threads until the subarray is shorter than a cutoff threshold and then STLSorted in parallel. However, its preliminary execution time is a bit longer than that of our benchmark, a parallel original Hoare's algorithm, on a 24-thread AMD ThreadRipper 2920x Linux system.**

Keywords-Triple Pivot, Sort, parallel, OpenMP, multithread

I. Introduction

Sorting is one of the most fundamental algorithms in computing until the Data Science and Big Data era. The Single-thread Single-pivot Quicksort algorithms are based on the well known Hoare's patition algorithm since 1962 [1], [2]. In addition to Hoare's, the so called Lomuto's partition algorithm [3] is considered single pivot. Other than these single-pivot in-place quicksort algorithms, a multipivot SampleSort [4] algorithm was dated back in 1970.

In this paper, we have devised a parallel triple pivot Sort (PTPSort) algorithm on manycore CPUs. Our main contributions are as follows.

1) The parallel partition algorithm of PTPSort can obtain four subarrays based on three pivot values.
2) The proposed partition algorithm of PTPSort can achieve about the same run time as the parallel Hoare partition function on AMD TR-2920x, a CPU with 12 cores 24 threads.

This paper can be formatted as follows. Section 2 reviews some important multi-pivot sorting algorithms and related work to PTPSort. Section 3 elaborates how PTPSort performs based on Hoare's. Preliminary experiment results are presented and discussed in Section 4. Finally, Section 5 concludes and suggests future work.

II. Background and Related Work

Important sequential and parallel multi-pivot sorting algorithms are reviewed in addition to the STLSort (Standard Template Library Sort) .

A. Multi-Pivot Sorting Algorithms

Since 1962, single-pivot QuickSort algorithms have been devised based on the so-called Hoare's partition algorithm [1]. The multipivot Samplesort was later on proposed in 1970 [4]. Since then, multi-pivot sorting algorithms have been proposed including [5], [6], [7], [8], [9], [10], [11], [12]

In 2009, a dual-pivot QuickSort was proposed and written in Java by Yaroslavskiy [5]. It is quite similar to Lomuto's but rather sequential. Two years later in 2011, Solehria and Jadoon presented their empirical results of a triple-pivot sorting algorithm [6]. In 2014, Kushagra et al. [7] reported a triple pivot algorithm with better results than a single-pivot one. It was clamied to have fewer number of comparisons and cache misses. Bodiman et al. [8] presented a number of Python based QuickSort algorithms supporting up to five pivots in 2017. As a parallel multi-pivot bucket sort algorithm, Mahafzah [9] proposed one in 2013. The input array can be classified by m buckets and sorted by m threads in parallel based on $m-1$ pivot values. In addition, Axtmann et al [10] published and opened the source code of an in-place parallel multipivot bucket-based sorting algorithm. Lastly, the Lomuto's algorithm was applied to divide a data array from both ends to the middle [11] and [12] as a parallel dual-pivot partition algorithm.

B. Standard Template Library Sort (STLSort)

Among the sorting algorithms, STLSort and its parallel modes are standardized and efficient to handle any data type. As such, the Standard Template Library $(STL)Sort$ is a comparison-based. It is available in most C++ compilers and declared as follows.

void sort(RandomAccessIterator first, RandomAccessIterator last);

The parameters $first$ and $last$ are begin and end indices of an array to be sorted, respectively. STLSort supports user-define comparison function for flexibility. Moreover, a GNU libstdc++ parallel mode sorting function, Balanced

978-1-7281-6486-1/20/$31.00 ©2020 IEEE

QuickSort[13] is a parallel QuickSort function with single-pivot partition algorithm.

III. A Parallel Triple-Pivot QuickSort (PTPSort)

The PTPSort's triple pivot partition algorithm is based on the single-pivot Hoare's due to its simplicity and good performance. Its c-style pseudocode is depicted in Algorithm 1. Recursive calls are made and forked as OpenMP tasks (threads).

At line 13 of Alg. 1, three pivot indices, iP_{Lo}, iP_{Mi} and iP_{Hi} are randomly selected between beg and end of input array a. Pivot values P_{Lo}, P_{Mi} and P_{Hi} corresponding to iP_{Lo}, iP_{Mi} and iP_{Hi} indices can be obtained and reordered such that $P_{Lo} < P_{Mi} < P_{Hi}$. The lowest pivot value P_{Lo} is applied with Hoare function as an OpenMP Task (thread) (Line 15) on the left half. As a consequence, i_L is returned pointing to the boundary where $a[i] \geq P_{Lo}$ and $i \geq i_L$. On the other end, the largest pivot value P_{Hi} is applied with the right half partition (Line 17). Similarly, i_H is returned pointing to the boundary where $a[i] \geq P_{Hi}$ and $i \geq i_H$.

Once both threads finish and synchronize with **OpenMP TaskWait** at line 18. Two undecided middle subarrays corresponding to $a[\cdot] \geq P_{Lo}$ and $a[\cdot] < P_{Hi}$ are then partitioned with P_{Mi} from the i_L to i_H-1 to achieve $a[\cdot] < P_{Mi}$ and $a[\cdot] \geq P_{Mi}$ partitions on the left and right, respectively. The middle-left partition ($P_{Lo} \leq a[\cdot] < P_{Mi}$) and the middle-right one ($P_{Mi} \leq a[\cdot] < P_{Hi}$ partition) can finally be obtained at lines 21 and 26, respectively. Both threads can be executed in parallel resulting in four partitions. The next recursive calls are invoked at lines 22 and 23 of Alg. 1 as two nested threads right after line 21 finishes. Similarly, the next recursive calls are forked at lines 27 and 28. These four subarrays shall be further recursively divided until the subarrays are shorter than U_{stl} elements and then STLSorted.

Figure 1 depicts the parallel partition step of PTPSort. Note that the arrow lines correspond to Hoare's partion algorithm with pivot value in the middle. In addition, these four subarrays can be recursively partitioned in parallel as four independent threads.

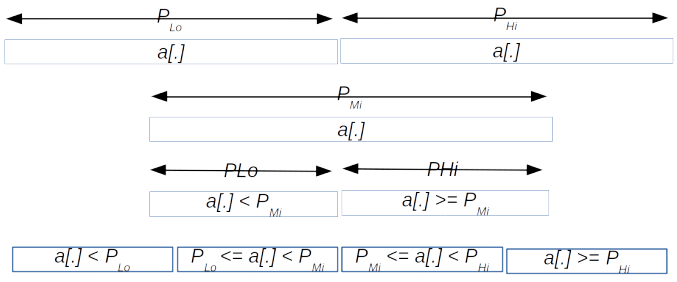

Fig. 1: Parallel partition of PTPSort algorithm (Note that the figure is not drawn to scale.)

ALGORITHM 1: Pseudocode of PTPSort Algorithm

```
1  Function main()
2      PTPSort(a, 0, N − 1)
3  EndFunction
4  Function PTPSort(a, beg, end)
5      if end − beg + 1 < U_stl then
6          if !PartitionOnly then
7              OpenMP Task
8              STLSort(a + beg, a + end + 1)
9          end
10         return;
11     end
12     len = (beg − end)/2
13     iP_Lo, iP_Mi, iP_Hi = Sampling(a, beg, end)
14     OpenMP Task
15     i_L=Hoare(a, beg, beg + len − 1, P_Lo)
16     OpenMP Task
17     i_H=Hoare(a, beg + len, end, P_Hi)
18     OpenMP Taskwait
19     i_M=Hoare(a, i_L, i_H − 1, P_Mi)
20     OpenMP Task {
21     i_lm=Hoare(a, i_L, i_M − 1, P_Lo)
22     PTPSort(a, beg, i_lm − 1)        // new thread
23     PTPSort(a, i_lm, i_M − 1)        // new thread
24     }
25     OpenMP Task {
26     i_mh=Hoare(a, i_M, i_H − 1, P_Hi)
27     PTPSort(a, i_M, i_mh − 1)        // new thread
28     PTPSort(a, i_mh, end)            // new thread
29     }
30 EndFunction
31 Function Hoare(a, b, e, P)
32     i = b, j = e
33     while true do
34         while a[i] < P do
35             i = i+1
36         end
37         while a[j] ≥ P do
38             j = j-1
39         end
40         if i > j then
41             return j − 1
42         end
43         swap(i, j)
44     end
45 EndFunction
```

IV. Experiments and Discussions

This section describes how the experiments are prepared and conducted. Preliminary results are presented and discussed

based on performance indicators such as Execution Time, Data TLB Loads/Misses and Branch Loads/Misses.

A. Experiment Setup

Both PTPSort and the original Hoare's algorithms are evaluated on the same AMD ThreadRipper TR-2920x as shown in Table I. According to AMD, AMD TR-2920x is composed of 4 identical dies. Each die hosts 3 CPU cores with a shared 8MB L3 cache. In total, an AMD TR-2920x chip comprises of 12 CPU cores. In addition, each core is capable of simultaneous multhreading up to two threads that amounts to 24 threads. It is also equiped with a private 512KB L2 cache, 64KB L1 instruction cache, and 32KB L1 data cache.

The operating system is 64-bit Ubuntu Linux 18.04.1 kernel 4.15.0-43. The GCC optimization flag is -O2 to get faster execution time. Both algorithms are compiled with G++ 7.3.0 and linked with OpenMP 4.5 library. To control the amount of parallelism, the total threads is fixed at 24 threads for every experiment.

Only uniform 32-bit unsigned integer (Uint32) data type is experimented. The array a ranges from N=100M to 500M elements where M=10^6. The STLSort cutoffs U_{stl} are 100k, 200k, 500k, 1M and 2M elements.

TABLE I: Specifications of AMD ThreadRipper 2920x

System	TR-2920x
Name	ThreadRipper
Architecture	Zen+
Clock	3.50 GHz
Cores vs. Threads	12 vs. 24
RAM	8x8GB
Memory Channels	4
Technology	DDR4-2400
L1 I-Cache	12x64KB 4-way
L1 D-Cache	12x32KB 8-way
L2 Cache	12x512KB 8-way
L3 Cache	4x8MB 16-way

B. Performance Indicators

In order to investigate its performance on a manycore CPU the following indicators should be measured and analyzed.

1) Execution Time T: Execution time is the key performance indicator to compare the new PTPSort algorithm with the original one because of their similarities. Both T_{ptp} and T_{org} are measured with the same data files excluding the overheads such as opening/closing file and data initialization. In addition to Sorting Time, the Partition Only Time can be investigated by disabling the STLSort function call at line 6 of Alg. 1. It can indicate how well the proposed/original partition algorithm performs.

2) Data TLB Loads D and Data TLB Load Misses D': The perf -stat utility version 4.15.18 is invoked by *perf stat -r 10 -e* to profile the algorithm repeatedly 10 times to get the average results. Users can specify several indicators including Data TLB Loads D, Data TLB Load Misses D', Branch Loads B and Branch Load Misses B'.

TLB stands for Translation Look Aside Buffer. TLB is a hardware cache of RAM-based page table responsible for translating virtual page numbers to physical frame numbers. Every instruction fetch incurs an instruction TLB access. Similarly, whenever a program reads/writes data, the CPU initiates Data TLB loads/stores accordingly. Therefore, the number of Data TLB loads D amounts to the complexity of the algorithm. In addition, Data TLB Load Misses D' can slow down the execution just like data cache misses.

3) Branch Loads B and Branch Load Misses B': Branches correspond to comparison or conditional statements. The more comparisons the higher the number of Branch Loads B done at the Branch Prediction unit. Inside the loops, a branch prediction unit play an important role to predict the outcome of each branch/condition instruction. The correct predicted outcomes can fill up the instruction pipeline resulting in faster execution time. On the other hand, the wrong prediction outcomes result in more Branch Load Misses and longer execution time as penalties. The worse branch prediction algorithm, the higher Branch Load Misses B'.

C. Results and Discussions

The preliminary results of PTPSort are presented and discussed as follows.

1) Execution Time: Sort vs. Partition Only: In order to compare the PTPSort with the original one, we share the same set of data files. Similarly, the sizes of work load vary from 100M to 500M. Each Excution Time is averaged by100 trials.

TABLE II: Sort vs. Partition Only: Average Execution Time T_{ptp} vs. T_{org} of N data array at various U_{stl} on TR-2920x system (Note that Part.=Partition Only, 100 Trials

N	U_{stl}	T_{ptp}		T_{org}	
		Sort	Part.	Sort	Part.
100M	2.0M	1.659	**1.345**	1.621	1.331
	1.0M	1.591	1.380	1.544	**1.322**
	0.5M	1.566	1.359	1.516	1.352
	0.2M	**1.550**	1.361	1.506	1.364
	0.1M	1.551	1.405	**1.496**	1.360
200M	2.0M	3.310	**2.741**	3.292	**2.835**
	1.0M	3.261	2.791	3.199	2.851
	0.5M	3.292	2.843	3.159	2.889
	0.2M	**3.188**	2.878	**3.129**	2.940
	0.1M	3.250	2.922	3.147	2.971
300M	2.0M	5.063	**4.193**	4.932	**4.339**
	1.0M	4.938	4.257	4.956	4.409
	0.5M	4.907	4.377	4.886	4.447
	0.2M	**4.897**	4.325	**4.870**	4.484
	0.1M	4.972	4.514	4.872	1.465
400M	2.0M	**6.513**	**5.696**	6.417	**5.603**
	1.0M	6.555	5.712	6.420	5.671
	0.5M	6.557	5.799	6.388	5.762
	0.2M	6.593	5.829	6.368	5.920
	0.1M	6.573	5.943	**6.362**	5.911
500M	2.0M	8.690	**7.187**	8.272	**7.353**
	1.0M	8.541	7.255	8.297	7.387
	0.5M	8.528	7.315	8.303	7.602
	0.2M	**8.486**	7.622	**8.245**	7.528
	0.1M	8.663	7.704	8.292	7.678

The best Execution Time is highlighted in each data size and column of Table II. For Partition Only, it is also noticeable that the bigger U_{stl} cutoff, the lower Execution Time. PTPSort is a bit faster than the original one except at N=100M and 400M.

For Sorting, PTPSort is slightly slower than the original one at every data size N. U_{stl} at either 0.1M or 0.2M elements can achieve the fastest Execution Time except at N=400M. Note that each element of Uint32 is 4 bytes. Therefore, the size of U_{stl}=0.1M or 0.2M corresponds to L2 cache (512 MB) of each core. On the other hand, U_{stl}=2M corresponds to L3 cache (8MB) of each ThreadRipper die.

2) Partition Only: Data TLB Loads vs. Data TLB Load Misses: Table III tabulates the Data TLB and Data TLB Misses of Partition function of PTPSort vs. Original at U_{stl}=2M on TR-2920x system. It can be observed that the bigger data size N the higher Data TLB Loads. This can be due to the fact that PTPSort accesses more data to gain higher parallelism. As a result, more Data TLB Load Misses D' can occur. However, the D'/D of PTPSort is about the same as that of Org.

TABLE III: Data TLB Loads D and Data TLB Load Misses D' of Partition Only N data array to be less than U_{stl}=2M on TR-2920x system with PTPSort vs. Parallel Original one, Remark xey=$x \times 10^y$

N	PTPSort			Org.		
	D	D'	D'/D	D	D'	D'/D
100M	7.92e09	1.78e6	0.022%	6.14e09	1.59e6	0.026%
200M	1.75e10	3.58e6	0.020%	1.39e10	3.41e6	0.025%
300M	2.87e10	6.37e6	0.022%	2.24e10	5.29e6	0.024%
400M	4.01e10	9.17e6	0.023%	3.08e10	6.85e6	0.022%
500M	5.15e10	1.11e7	0.021%	3.99e10	8.97e6	0.023%

TABLE IV: Branch Loads B and Branch Load Misses B' of Partition Only N data array to be less than U_{stl}=2M on TR-2920x system with PTPSort vs. Parallel Original one, Remark xey=$x \times 10^y$

N	PTPSort			Org.		
	B	B'	B'/B	B	B'	B'/B
100M	4.59e09	2.70e8	5.89%	3.22e09	2.55e8	7.92%
200M	1.02e10	6.09e8	5.97%	7.15e09	5.90e8	8.25%
300M	1.66e10	9.99e8	6.03%	1.16e10	9.66e8	8.28%
400M	2.28e10	1.38e9	6.07%	1.60e10	1.37e9	8.60%
500M	2.93e10	1.79e9	6.11%	2.12e10	1.76e9	8.33%

3) Partition Only: Branch Loads vs. Branch Load Misses: Table IV tabulates the Branch Loads B and Branch Load Misses B' of Partition Only of PTPSort vs. Original at U_{stl}=2M on TR-2920x system. It can be observed that the bigger data size N the higher Branch Loads. This can be due to the fact that PTPSort accesses more data to compare and swap at each recursion level. As a result, more Branch Load Misses B' can occur. However, the B'/B of PTPSort is much lower that of Org. That means it is easier to predict branch outcomes. Lower B'/B and D'/D may result in better partition time than the Original algorithm in general.

In summary, our PTPSort demands about 25% Data TLB Loads D and 40% Branch Loads B more than those of original one in order to gain more parallelism. This may be due to selection and skew of pivot values. More sampling is needed to obtain proper and balanced pivot values to eliminate unnecessary data TLB Loads D, data TLB load Misses D', Branch Loads B and Branch Load Misses B'. That may result in faster execution time.

V. Conclusions and Future Work

The Parallel Triple-Pivot Sort is a multithreaded algorithm for manycore CPU systems. Based on the well-known Hoare's partition algorithm, the input array is initialy partitioned with two pivots, P_{Lo} and P_{Hi} in parallel with two threads resulting in four subarrays. The leftmost one is less than P_{Lo} while the rightmost one is greater than or equal to P_{Hi}. Two undecided middle ones are further combined and partitioned by P_{Mi}, P_{Lo} and P_{Hi} respectively, resulting in four subarrays. The process recursively continues to partition and finally sort subarrays in multithread. To be fair, PTPSort is benchmarked against the original Hoare's in terms of execution time, Data TLB Load Misses and Branch Load Misses. As a conclusion, the PTPSort algorithm can be a bit slower than the original PTPSort algorithm on an 12-core 24-thread AMD TR-2920x running Ubuntu Linux 18.04.1 LTS system.

The PTPSort shall be investigated to improve its performance further. For example, five and seven pivots can be applied as a parallel multipivot sorting algorithm for future work with more cores/threads.

References

[1] C. A. R. Hoare, "Quicksort," *ACM*, vol. 4, p. 321, 1962.

[2] R. Sedgewick, "Implementing quicksort program," *Communications of the ACM*, vol. 21, no. 10, pp. 847–857, October 1978.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[4] W. D. Frazer and A. C. McKellar, "Samplesort: A sampling approach to minimal storage tree sorting," *J. ACM*, vol. 17, no. 3, p. 496507, Jul. 1970. [Online]. Available: https://doi.org/10.1145/321592.321600

[5] V. Yaroslavskiy, "Dual-pivot quicksort algorithm," 2009. [Online]. Available: http://codeblab.com/wp-content/uploads/2009/09/DualPivotQuicksort.pdf

[6] S. F. Solehria and S. Jadoon, "Multiple pivot sort algorithm is faster than quick sort algorithms: An empirical study," *International Journal of Electrical & Computer Sciences*, vol. 11, no. 3, pp. 14–18, 2011.

[7] S. Kushagra, A. López-Ortiz, J. I. Munro, and A. Qiao, "Multi-pivot quicksort: Theory and experiments," in *Proceedings of the Meeting on Algorithm Engineering & Expermiments*. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2014, pp. 47–60. [Online]. Available: http://dl.acm.org/citation.cfm?id=2790174.2790180

[8] M. Budiman, E. Zamzami, and D. Rachmawati, "Multi-pivot quicksort: an experiment with single, dual, triple, quad, and penta-pivot quicksort algorithms in python," in *1st Annual Applied Science and Engineering Conference*, 2017, pp. 1–6.

[9] B. A. Mahafzah, "Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture," *J of Supercomputing*, vol. 66, no. 1, pp. 339–363, 2013.

[10] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place parallel super scalar samplesort (ipsssso)," *CoRR*, vol. abs/1705.02257, 2017. [Online]. Available: http://arxiv.org/abs/1705.02257

[11] S. Taotiamton and S. Kittitornkun, "Parallel hybrid dual pivot sorting algorithm," in *Proceedings of 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, July 2017, pp. 377–380.

[12] ——, "A parallel dual-pivot quicksort algorithm with lomuto partition," in *Proceedings of The 21st International Computer Science and Engineering Conference*, November 2017, pp. 164–167.

[13] J. Singler, P. Sanders, and F. Putze, "Mcstl : The multi-core standard template library," *Euro-Par 2007 Parallel Processing. Springer Berlin Heidelberg*, pp. 682–694, 2007.