ISBN 978-981-11-7861-0
Proceedings of 2018 the 8th International Workshop on Computer Science and Engineering (WCSE 2018)
Bangkok, 28-30 June, 2018, pp.6 85-690
doi: 10.18178/wcse.2018.06.113

Dual Parallel Partition Sorting Algorithm

Apisit Rattanatranurak[+]

Department of Computer Engineering, Faculty of Industrial Technology, Suan Sunandha Rajabhat University, Bangkok, Thailand

Abstract. Sorting is the important algorithm which is widely used and implemented in many applications. This paper presents an efficient parallel sorting algorithm called Dual Parallel Partition sorting (DPPSort). The DPPSort partitions the data recursively and then sorts in parallel. The partitioning phase divides the data into two parts in parallel. In sorting phase, the Standard Template Library Sorting function (STLSort) and GNU sorting function (qsort) are integrated in this work. This work is developed in C/C++ language and linked with OpenMP library. In our experiments, the 4-core Intel i7-3770 with Ubuntu Linux systems is implemented. Our work is faster than qsort and STLSort function up to 5.95× and 4.70×, respectively.

Keywords: multithreading, parallel processing, partitioning algorithm, sorting, OpenMP.

1. Introduction

Sorting is the important algorithm for biological, scientific application including Big data. The well-known sorting algorithm is quick sort [1], [2] which is divide and conquer technique to divide the data in smaller size and sort them. It consists of partitioning and sorting step. Partitioning is the step which divides the data using pivot into two or more partitions. This step divides the data into two partitions recursively in quick sort algorithm. If the data size of those partitions are smaller than sorting cutoff size, that partition is sorted in the sorting step.

This paper looks into the simple technique by parallel partitioning in two parts. Then, those parts are merged using simple algorithm by swapping the data to the right place. Finally, the pivot is moved to its new position and run this algorithm in parallel recursively.

In this paper, we focus on partitioning step in divide and conquer sorting algorithm. It is developed and can be run in parallel using OpenMP library. We then show its run time and Speedup compared with original algorithm. Moreover, we compare sorting cutoff size that affects run time of this algorithm on multicore CPU.

This paper is organized as follows: Section 2 shows Background and Related Works. Sections 3 proposes our Dual Parallel Partition Sorting Algorithm. In section 4, the experiment results are compared with related work. Finally, sections 5 shows conclusion and future work.

2. Background and Related Works

2.1. Standard Sorting Algorithm Library

qsort is a standard library for sorting the data. To implement this function, <stdlib.h> directive must be declared in C language. It can be implemented as follows:

void qsort (void ****base*, size_t *num*, size_t *size*, int (****compare***)(void const*, void const*));**

base is a pointer to the array, *num* is the number of elements in array, *size* is the size in bytes of each element, and *compare* is a pointer to a function that compares the elements.

[+] Corresponding author.
E-mail address: apisit.ra@ssru.ac.th

STLSort is a sorting standard library function that can sort the data. This function can be implemented in C++ by declared <algorithm> directive. It can be implemented as follows:

void sort(RandomAccessIterator *first*, RandomAccessIterator *last*);

first and *last* are the range of sorting elements.

2.2. OpenMP Library

OpenMP [3] is the library which can be developed program in parallel for multicore CPUs. It can be implemented using complier directives, environment variable, and its functions in C/C++ and Fortran. The execution model of this library is fork-join model. It runs the master thread in sequential area, fork threads in parallel area, then join threads when finished. The memory can be shared with less overhead between CPU cores compared with other methods.

There are several constructs in OpenMP such as single program multiple data (SPMD) constructs, tasking constructs, device constructs, work sharing constructs, and synchronization constructs. Tasking construct can be used in recursion function. A task unit is run by thread in parallel. It can be implemented in C/C++.

2.3. Related Works

Heidelberger et al. [4] proposed parallel quick sort on an ideal Parallel Random Access Machine. Its average complexity is O(logN). Tsigas and Zhang [5] presented the quick sort algorithm named PQuicksort that used fine-grain parallelism concept. The neutralized blocks technique is used in their algorithm. Leopold [6] proposed the parallel quick sort using pthreads and OpenMP 2.0. Tsigas and Zhang's [5] PQuicksort is modified by Rachid et al. [7]. psort algorithm is developed by Duhu Man et al. [8], [9]. This algorithm splits the data into groups and sorts them locally. Then, those groups are merged and sorted again. Speedup of 11× is achieved on 24 cores CPU. Kim et al. [10] run their Introspective quick sort algorithm on an embedded dual core OMAP-4430. Speedup of their work is 1.47×. Mahafzah [11] run their sorting algorithm which divides the input array with multi-pivot/thread into partitions. The partitions are sorted in parallel up to 8 threads. Saleem et al.[12] used Intel Cilk Plus and estimated Speedup both quick sort and merge sort algorithm. Ranokpanuwat and Kittitornkun [13] proposed Parallel Partition and Merge Quick sort (PPMQSort) that executes on Shared memory/multicore system with OpenMp 3.0. Speedup of 12.29× can be achieved on 8-core HyperThread Xeon E5520 for sorting 200 million 32-bit randomly unsigned integer data. Recently, Taotiamton and Kittitornkun [14] presented parallel Hybrid Dual Pivot Sort (HDPSort). Their algorithm used both Lomuto and Hoare par- titioning with two pivots in parallel using OpenMP. Speedup of 3.02× and 2.49× can be achieved on AMD FX-8320 and Intel Core i7-2600 systems.

3. Dual Parallel Partition Sorting

There are four algorithms in this paper. Firstly, DPPSort is the partitioning function Next, LPartition and RPartition are the sequential partitioning function. LPartition is traversed from left to right. On the other hand, RPartition is traversed from right to left. Finally, MultiSwap is merge function used to select the new position of that level of partitioning.

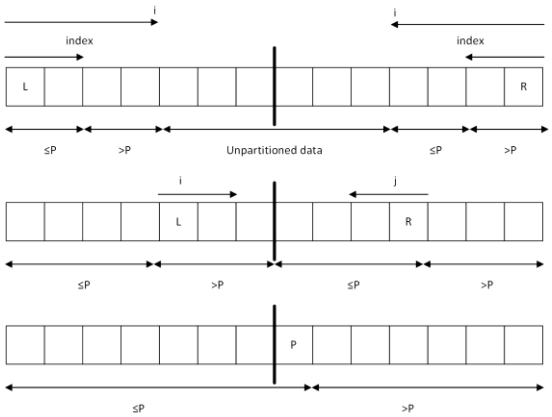


Fig. 1: DPPSort algorithm.

We have declared the notation in this paper as follows: *arr* is array of data, *L* is left position, *R* is right position, *C* is Sorting cutoff size, and *P* is pivot position.

The algorithm starts with compares size of array with sorting cutoff. While it is larger than sorting cutoff, Median of five (MO5) function is executed to select a pivot (line 1, Algorithm 1). The pivot is selected from 5 data randomly in the unsorted array, sort them and choose the 3rd data as selected pivot. The selected pivot is used to partition in LPartition and RPartition function. Both functions divide the data using the selected pivot in parallel using omp task and return the new position of the pivot (line 8 and 10, Algorithm 1). After that, MultiSwap function is executed to swap the data and return new pivot position of this level (line 12, Algorithm 1). Finally, this function is recursively executed on the left and right partition in parallel using omp task (line 14 and 16, Algorithm 1). Note that, if partition is smaller than sorting cutoff, sorting function is executed in parallel (line 3, Algorithm 1).

The algorithm starts with compares size of array with sorting cutoff. While it is larger than sorting cutoff, Median of five (MO5) function is executed to select a pivot (line 1, Algorithm 1). The selected pivot is used to partition in LPartition and RPartition function. Both functions divide the data using the selected pivot in parallel using omp task and return the new position of the pivot (line 8 and 10, Algorithm 1). After that, MultiSwap function is executed to swap the data and return new pivot position of this level (line 12, Algorithm 1). Finally, this function is recursively executed on the left and right partition in parallel using omp task (line 14 and 16, Algorithm 1). Note that, if partition is smaller than sorting cutoff, sorting function is executed in parallel (line 3, Algorithm 1).

Algorithm 1 DPPSort Input: *arr*,*L*,*R*

1: if(*R*−*L*<*C*)then
2: omp task nowait
3: qsort() or STLSort()
4: end if
5: M = *L* + (*R* − *L*)/2
6: MO5(*arr*,*L*,*R*)
7: omp task shared(new mid)
8: new mid=LPartition(*arr*,*L*,M−1,M)
9: omp task shared(new mid2)
10: new mid2=RPartition(*arr*,M+1,*R*,M)
11: omp taskwait
12: new mid3 = MultiSwap(*arr*,new mid,new mid2,M)
13: omp task
14: DPPSort(*arr*,*L*,new mid3−1)
15: omp task
16: DPPSort(*arr*,new mid3+1,*R*)

Algorithm 2 MultiSwap

Input: *arr*,*L*,*R*,*P* Output: i or j

1: i=*L*
2: j=*R*
3: while i<j and i<*P* and j >*P* do
4: swap(*arr*[i], *arr*[j])
5: i=i+1
6: j=j−1
7: end while
8: if i>*P* then
9: swap(arr[j], arr[*P*])
10: return j
11: else
12: swap(arr[i], arr[*P*])
13: return i
14: end if

In this algorithm, OpenMP nested is set to enable using omp_set_nested() function. The new team of DPPSort function (line 14 and 16, Algorithm 1) can be spawned and the new team consists of two thread in every recursion.

3.1. Dual Partitioning Phase

The concept of dual partitioning is very simple. LPartition and RPartition are partitioning using two pointers to split the data and traverse in the same direction. They are traversed from left to the middle and right to the middle, respectively. There are indices which divide the data that less than and greater than pivot (line 2, Algorithm 3, 4) and i which is used to separate partitioned data and unpartitioned data (line 3, Algorithm 3, 4). This algorithm compares *arr*[i] and pivot (line 4, Algorithm 3, 4). While data at i position is less than or equal to pivot in LPartition or greater than pivot in RPartition, data is swapped and index is increased by 1 (line 5-6, Algorithm 3, 4). The iteration will be run until it reaches middle position of that

partition. Finally, it returns index to DPPSort function as new mid and new mid2 in LPartition and RPartition, respectively (line 8 and 10, Algorithm 1). This phase is shown in the first line of Fig. 1.

3.2. Multi-Swap Phase

In Dual Partitioning Phase, the partition is divided into two parts. There are data which greater than pivot in the left part and data which less than pivot in the right part. The data are only swapped until index is at the pivot position.

Algorithm 3 LPartition

Input: *arr,L,R,P* Output: *index*

1: val = *arr*[*P*]
2: *index* = *L*
3: for i=*L*;i<=*R*;i=i+1 do
4: if *arr*[i] <= val then
5: swap(*arr*[i], *arr*[*index*])
6: *index* = *index* + 1
7: end if
8: end for
9: return *index*

Algorithm 4 RPartition

Input: *arr,L,R,P* Output: *index*

1: val = *arr*[*P*]
2: *index* = *R*
3: for i=*R*;i>=*L*;i=i-1 do
4: if *arr*[i] > val then
5: swap(*arr*[i], *arr*[*index*])
6: *index* = *index* - 1
7: end if
8: end for
9: return *index*

This phase starts with initial i and j to the left position and right position, respectively (line 1-2, Algorithm 2). Then, the data at position i and j are swapped and move to the next position until i is greater than pivot position or j is less than pivot position (line 3-6, Algorithm 2). Finally, if i is greater than pivot position, the data at j position is swapped with pivot and return j as pivot position (line 8-10, Algorithm 2). On the other hand, the data at i position is swapped with pivot and return i as pivot position (line 11-13, Algorithm 2). This phase is illustrated as shown in the second and third line of Fig. 1.

3.3. Sorting Phase

In previous phase, the data which is partitioned successfully and smaller than sorting cutoff size is sorted by sorting function (qsort or STLSort) in parallel. The OpenMP Parallel task is used to sort the data by forking thread without blocking (line 2-3, Algorithm 1). This thread is joined with the master thread automatically after the data are sorted.

4. Experiments, Results, and Discussions

4.1. Experiments Setup

We compare DPPSort function with qsort ($DPPSort_{qsort}$) and STLSort ($DPPSort_{STL}$) as Sorting cutoff Algorithms, qsort function, and STLSort function. The random 32-bit unsigned integer are sorted in this experiment. The 50, 100, 200 million random data *N* are generated for every experiment. The sorting cutoff *C* are *N*/2, *N*/4, *N*/8 and *N*/16. Each parameter is tested for 10 times and averaged as Run time in seconds.

We use Intel core i7-3770 3.4 GHz which consists of 4 cores with 8 threads. There are DDR3-1600 32 GB main memory and run on Ubuntu 16.04 LTS Operating System.

4.2. Results

There are two metrics which are used to measure the performance of $DPPSort_{qsort}$ and $DPPSort_{STL}$. 1) Run Time is the basic metric which can be used to measure the performance of the algorithm. 2) Speedup is the metric which can be used to compare the performance of algorithm and the original one.

1) Run Time: DPPSort is faster than other algorithms and $DPPSort_{STL}$ is the fastest. Its run time is only 5.47 and 3.34 seconds to sort 200 million Uint32 data using qsort and STLSort, respectively as Sorting cutoff algorithm. Run time of qsort and STLSort function which are the standard library are 32.56 and 15.68 seconds, respectively. It can be noticed that $DPPSort_{STL}$ is faster compared with $DPPSort_{qsort}$ since run time of qsort is greater than STLSort.

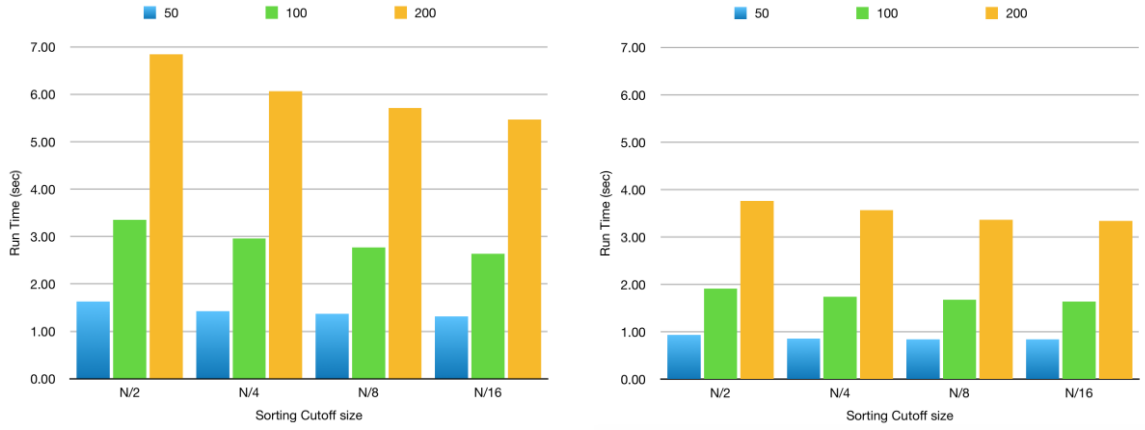
Sorting cutoff size is very important parameter in DPPSort. It is proportional to run time and affects with run time complexity. The results show that run time of $DPPSort_{qsort}$ and $DPPSort_{STL}$ are the fastest at $C = N/16$ and $C = N/16$, respectively. Run time of $DPPSort_{qsort}$ and $DPPSort_{STL}$ are illustrated in Fig.2(a) and 2(b). DPPSort run time slightly falls while the sorting cutoff size is smaller in any sorting cutoff algorithm. It is very significant while the data size is larger. We can notice that the best sorting cutoff size is smallest size. The Dual partitioning phase should be run until its partitions are small enough. Then, sort the partitions sorting function in parallel.

2) Speedup: The metric which can be used to measure the performance of DPPSort algorithm in this paper is Speedup. Note that, Speedup is the fraction of run time of original vs DPPSort algorithm.

The best Speedup of $DPPSort_{STL}$ is up to 4.70× at 200 million data with $C = N/16$. Moreover, the best Speedup of $DPPSort_{qsort}$ is 5.95× at 200 million data with $C = N/16$.

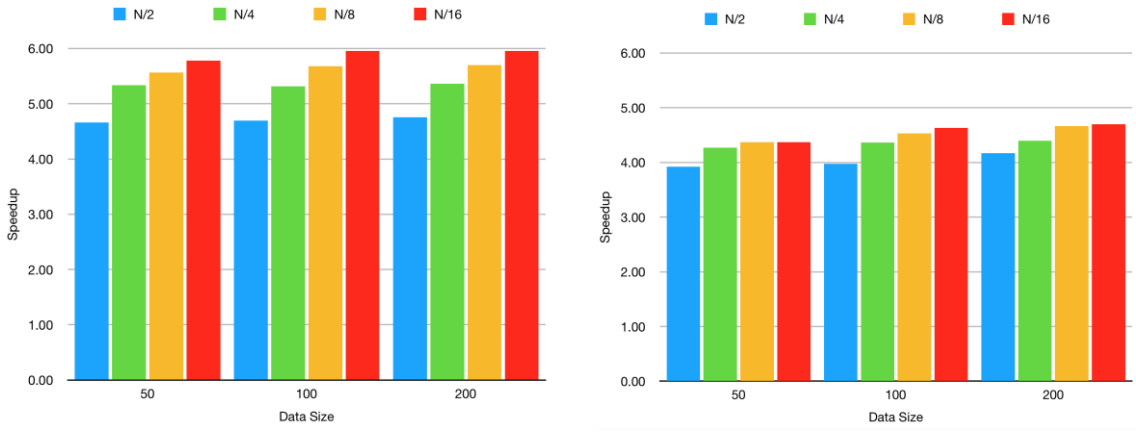
Average Speedup of $DPPSort_{qsort}$ and $DPPSort_{STL}$ are illustrated in Fig.3(a) and 3(b). It can be noticed that Speedup significantly depends on data size. While data size is larger, the sequential part of the algorithm is smaller compared with parallel part. Therefore, Speedup is increased significantly proportional to data size. Furthermore, its speedup depends on sorting cutoff algorithm. Speedup of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$ significantly.

It can be due to partitioning phase run time of both $DPPSort_{qsort}$ and $DPPSort_{STL}$ are equal. Sorting phase run time of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$ significantly. Therefore, Speedup of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$.



(a) $DPPSort_{qsort}$ (b) $DPPSort_{STL}$

Fig. 2: Run time of DPPSort algorithm vs sorting cutoff size on any data size ($\times 10^6$).



(a) $DPPSort_{qsort}$ (b) $DPPSort_{STL}$

Fig. 3: Speedup of DPPSort algorithm vs Data size ($\times 10^6$) with any Sorting Cutoff size.

5. Conclusion and Future Work

This paper proposes a Dual Parallel Partition Sorting (DPPSort) algorithm. The concept of DPPSort is to partition the data into two parts. Then, run partitioning algorithm in parallel and merge them with MultiSwap algorithm. This algorithm is run recursively until it is smaller than sorting cutoff size. The partition is sorted using standard sorting function in parallel.

DPPSort is applied and run on Intel core i7-3770 with Linux system. It is faster than other standard sorting algorithms like qsort and STLSort. Speedup is up to 5.95× and 4.70×, respectively. Its performance depends on data size and the sorting cutoff algorithm and its size.

DPPSort can be improved the performance in the future work. We can apply this algorithm to the larger machines and analyze the parameter which proportional to run time. Moreover, we can implement this algorithm to the heterogeneous system to achieve Speedup of the algorithm.

6. 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] OpenMP 4.0 specification, June 2013.

[4] P. Heidelberger, A. Norton, and J. T. Robinson, "Parallel quicksort using fetch-and-add," IEEE Transactions on Computers, vol. 39, no. 1, pp. 847–857, January 1990.

[5] P. Tsigas and Y. Zhang, "A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000," in 11th Euromicro Conference on Parallel Distributed and Network based Processing (PDP 2003), Genoa, Italy, February 5th-7th 2003, pp. 372– 381.

[6] M. Su ̈ß and C. Leopold, "A users experience with parallel sorting and openmp," in Proceedings of the Sixth European Workshop on OpenMP- EWOMP04, 2004, pp. 23–38.

[7] L. Rashid, W. M.Hassanein, and M. A.Hammad, "Analyzing and enhancing the parallel sort operation on multithreaded architectures," J of Supercomputing, vol. 53, pp. 293–312, 2010.

[8] D. Man, Y. Ito, and K. Nakano, "An efficient parallel sorting compatible with the standard qsort," in International Conference on Parallel and Distributed Computing, Applications and Technologies, Hiroshima, Japan, December 8-11 2009, pp. 512 – 517.

[9] D. Man, Y. Ito, and K. Nakano, "An efficient parallel sorting compatible with the standard qsort," International Journal of Foundations of Computer Science, vol. 22, no. 05, pp. 1057–1071, 2011.

[10] K. J. Kim, S. J. Cho, and J.-W. Jeon, "Parallel quick sort algorithms analysis using openmp 3.0 in embedded system," in 11th International Conference on Control, Automation and Systems, KINTEX, Gyeonggido, Korea, October 26-29 2011, pp. 757–761.

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

[12] S. Saleem, M. I. Lali, M. S. Nawaz, and A. B. Nauman, "Multi-core program optimization: Parallel sorting algorithms in intel cilk plus," International Journal of Hybrid Information Technology, vol. 7, no. 2, pp. 151–164, 2014.

[13] R. Ranokphanuwat and S. Kittitornkun, "Parallel partition and merge quicksort (ppmqsort) on multicore cpus," The Journal of Supercomputing, vol. 72, no. 3, pp. 1063–1091, 2016.

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