Analysis and optimization of Dual Parallel Partition Sorting with OpenMP

Sirilak Ketchaya
Faculty of Science and Technology, Suan Sunandha Rajabhat University, Bangkok, Thailand, and
Apisit Rattanatranurak
Department of Computer and Information Sciences, Faculty of Applied Science, King Mongkut's University of Technology North Bangkok, Bangkok, Thailand

Received 20 October 2021
Revised 27 March 2022
9 June 2022
Accepted 18 June 2022

Abstract

Purpose – Sorting is a very important algorithm to solve problems in computer science. The most well-known divide and conquer sorting algorithm is quicksort. It starts with dividing the data into subarrays and finally sorting them.

Design/methodology/approach – In this paper, the algorithm named *Dual Parallel Partition Sorting* (*DPPSort*) is analyzed and optimized. It consists of a partitioning algorithm named *Dual Parallel Partition* (*DPPartition*). The *DPPartition* is analyzed and optimized in this paper and sorted with standard sorting functions named *qsort* and *STLSort* which are quicksort, and introsort algorithms, respectively. This algorithm is run on any shared memory/multicore systems. OpenMP library which supports multiprocessing programming is developed to be compatible with C/C++ standard library function. The authors' algorithm recursively divides an unsorted array into two halves equally in parallel with *Lomuto's partitioning* and merge without compare-and-swap instructions. Then, *qsort/STLSort* is executed in parallel while the subarray is smaller than the sorting cutoff.

Findings – In the authors' experiments, the 4-core Intel i7-6770 with Ubuntu Linux system is implemented. *DPPSort* is faster than *qsort* and *STLSort* up to 6.82× and 5.88× on Uint64 random distributions, respectively.

Originality/value – The authors can improve the performance of the parallel sorting algorithm by reducing the compare-and-swap instructions in the algorithm. This concept can be used to develop related problems to increase speedup of algorithms.

Keywords Multithreading, Parallel processing, Partitioning algorithm, Sorting, OpenMP

Paper type Research paper

1. Introduction

The well-known algorithm for solving biological, scientific applications including big data is sorting. Quicksort [1, 2] is the important sorting algorithm which is a divide and conquer technique. The unsorted array is divided into smaller subarrays and sorted independently. It consists of partitioning and sorting steps in this technique. Initially, the partitioning step divides the unsorted array recursively using pivot(s) into subarrays (divide). It runs until each subarray is shorter than cutoff size. Note that, the popular partitioning algorithm is Hoare's partitioning algorithm [3]. Then, the sorter executes the sorting step independently (conquer).

The partitioning step is very important for sorting the data in parallel. The parallel partitioning which divides the unsorted array into two subarrays is proposed in this paper. Then, the partitioned subarrays on each side are merged to their position. Note that, the data

Applied Computing and Informatics
Emerald Publishing Limited
e-ISSN: 2210-8327
p-ISSN: 2634-1964
DOI 10.1108/ACI-10-2021-0288

are only swapped to their correct positions without compare-and-swap instructions in our merge algorithm. Finally, the pivot is moved to its correct position and this algorithm is executed recursively.

The partitioning step which is the part of divide and conquer concept is focused. OpenMP Task in the OpenMP library [4] is implemented using this algorithm. Run time, Speedup and Speedup per core/thread results compared with their original algorithms are presented. We optimized this algorithm using sorting cutoff size that affects run time on each data distribution. *Perf* profiling tool [5] is used to measure and analyze cache misses, branch misprediction and other metrics. Finally, we compare our proposed partitioning algorithm with *Hoare's partitioning algorithm*.

The contributions of this paper are summarized as follows:

(1) We proposed the parallel sorting algorithm named *Dual Parallel Partition Sorting* (*DPPSort*) algorithm which consists of partitioning and sorting steps using OpenMP.

(2) The parallel partitioning algorithm called *Dual Parallel Partition Phase* which divides the array into 2 subarrays. Then, partition them using *Lomuto's partitioning algorithm* independently. Finally, those two subarrays are merged without compare-and-swap instructions using *Multi-Swap Phase* is proposed.

(3) The performance metrics such as Run time, Speedup, Speedup/core, Speedup/thread, cache misses and branch load misses of *DPPSort* and the others are compared and analyzed.

This paper is organized as follows: Section 2 shows Background and Related work. Section 3 proposes our Dual Parallel Partition for sorting. In section 4, the results are shown and compared with any distributions. Finally, sections 5 shows conclusion and future work.

2. Background and related work

In this section, OpenMP [4] which is a parallel application programming interface is proposed. Moreover, sequential standard sorting algorithms named *qsort* and *STLSort* are proposed. Finally, several parallel sorting algorithms are proposed and compared with standard sorting algorithms in this section.

2.1 OpenMP library

OpenMP [4] is an application programming interface (API) which supports parallel programming on a shared memory system. It consists of complier directives, environment variables and functions that support C/C++ and Fortran. The execution model of OpenMP is the fork-join model. It starts with the master thread in sequential part. Then, worker threads are forked in parallel . Finally, all threads are joined while their works are finished. The overhead between CPU cores of this API is very low compared with other libraries.

The constructs of OpenMP consist of single program multiple data (SPMD) constructs, tasking constructs, device constructs, work sharing constructs and synchronization constructs. The tasking construct is implemented in the recursion function in this paper. A task unit is executed by a thread independently.

2.2 Standard sorting algorithm library

The well-known standard sorting algorithm libraries called *qsort* and *STLSort* are presented in this paper. *qsort* is a standard library for sorting the data. It is a well-known quicksort algorithm which consists of partitioning and sorting steps. < *stdlib.h* > directive must be included in C language to use this function.

STLSort [6] is a sorting standard library function that can sort the data. It consists of 3 algorithms. *Introsort* algorithm which is combined with quicksort and heapsort is performed. Then, insertion sort is executed to sort subarray. To implement this function in C++, < *algorithm* > directive must be declared.

In the quicksort algorithm, there are 2 well-known partitioning algorithms which are used to sort the data. The first algorithm is *Hoare's partitioning algorithm* [3]. It is the most popular algorithm that indices traverse from left to right and right to left to compare and swap data. The second algorithm is *Lomuto's partitioning algorithm*. Its indices traverse in the same direction. The first index is used to scan the array and the second index is used to divide the data that is less than pivot or greater than pivot. These indices run to compare and swap data until the first index is at the last data of the array.

2.3 Related works

There are several parallel quicksort algorithms which can be run on shared memory system. Many algorithm concepts start with dividing the data into blocks and partitioning the data in parallel. Then, the data in each block is merged to the correct position. We classify the related work into 4 categories as follows:

2.3.1 Parallel quicksort using fetch-and-add instruction and block-based techniques. Heidelberger *et al.* [7] proposed parallel quicksort on an ideal Parallel Random Access Machine using Fetch-and-add instruction. Speedup of 400× with 500 processors can be obtained from sorting 2^{20} data. *PQuicksort,* which is a fine-grain parallel quicksort algorithm, was proposed by Tsigas and Zhang [8]. Their algorithm uses neutralized blocks technique in parallel. Speedup of 11× can be obtained with 32 processor cores. Süß and Leopold [9] presented the implementation of pthreads and OpenMP 2.0 library to their parallel quicksort. Its speedup is 3.24× on a 4-core AMD Opteron processor. Traoŕe *et al.* [10] showed work-stealing technique of deque-free parallel introspective sorting algorithm. Speedup of 8.1× on a 16-core processor can be achieved. Ayguade *et al.* [9] presented *MultiSort* which divides the input and sorts them with quicksort. After that, sorted data are merged in parallel. The best speedup is 13.6× on a 32-core CPU. Kim *et al.* [11] developed an *Introspective quicksort* and executed on embedded dual core OMAP-4430. Speedup of their parallel Introspective quicksort is 1.47×. Saleem *et al.* [12] estimated speedup of both quick and merge sort algorithms using Intel Cilk Plus. Ranokpanuwat and Kittitornkun [13] developed Parallel Partition and Merge Quick sort (*PPMQSort*). Speedup of 12.29× is achieved on 8-core Hyperthread Xeon E5520 for sorting 200 million random integer data.

Recently, MultiStack Parallel Partition (*MSP*) which is a block-based partitioning algorithm was proposed [14]. Threads are forked to compare and swap the data in parallel using stacks. *MSPSort* is better than balanced quicksort and multiway merge sort while sorting Uint32 data on i7-2600, R7-1700 and R9-2920 machines.

2.3.2 Parallel sorting algorithms using multi-pivot technique. Man *et al.* [15, 16] developed *psort* which splits the unsorted array into groups of data and sorts them in parallel. After that, those groups of data are merged and sorted again in sequential order. This algorithm is run on a 24-core CPU and Speedup of 11× is achieved. Mahafzah [17] developed their multi-pivot sorting algorithm that divides the unsorted array into partitions in parallel up to 8 threads. Speedup of 3.8× can be obtained with a 2-core HyperThread machine. In 2017, parallel Hybrid Dual Pivot Sort (*HDPSort*) was presented by Taotiamton and Kittitornkun [18]. Both *Lomuto's* and *Hoare's partitioning algorithms* are implemented with two pivots in parallel using OpenMP. Speedup of 2.49× and 3.02× are achieved on Intel Core i7-2600 and AMD FX-8320 systems, respectively.

2.3.3 Parallel partitioning. Chen *et al.* [19, 20] proposed the performance-aware model for Sparse matrix multiplication on the Sunway TaihuLight supercomputer. A multi-level parallelism design for SpGEMM was developed to optimize load balance, coalesced DMA

transmission, data reuse, vectorized computation and parallel pipeline processing. Later, they presented an adaptive and efficient framework for the sparse tensor-vector product kernel on the Sunway TaihuLight supercomputer. The auto-tuner that selects the best tensor partitioning method to improve load balance was proposed. Its maximum GFLOPS is up to 195.69 on 128 processors.

2.3.4 GPU sorting algorithms. The parallel quicksort on GPU was implemented in 2010 [21]. It requires more memory to sort the data because it is not an in-place algorithm. This algorithm contains 2 phases. The first phase divides the data to GPU local memory and partitioning them. The second phase runs a portioning algorithm recursively using stack and sorting them.

Kozakai *et al.* [22] developed an integer sorting algorithm based on histogram and prefix-sums which run on GPU. Their algorithm is faster than the well-known sorting algorithms *Thrust sort* and *CUB sort* on Intel Xeon E5-2620 v3 and NVIDIA Tesla K40c.

3. Dual Parallel Partition Sorting algorithm

This section shows the divide and conquer sorting algorithm named Dual Parallel Partition Sorting (*DPPSort*). There are 5 algorithms which are implemented in this work. Firstly, Dual Parallel Partition function *DPPartition* (Algorithm 1) is the partitioning function. Median of five function or *MO*5 (Algorithm 2) is the pivot selection function before partitioning. *LPar* and *RPar* functions (Algorithms 3 and 4) which are *Lomuto's partitioning algorithms* are implemented in *DPPartition*. Note that, *LPar* and *RPar* are partitioning from left to right and right to left, respectively. Finally, *MSwap* (Algorithm 5) is a merging algorithm which swaps the partitioned arrays which are greater than pivot from left subarray and less than pivot from the right subarray. We have declared the notation in this paper as follows: *arr* is array of data, l is left position index, r is right position index, c is Sorting cutoff size and p is pivot position index.

Algorithm 1. DPPartition

Input: *arr, l, r*
Initialisation:
if ($r - l < c$) **then**
  omp task nowait
  qsort () *or STLSort* ()
end if
$p = l + (r - 1)/2$
MO5(*arr*, *l*, *r*)
omp task shared(*new_midL*)
new_midL = *LPar* (*arr*, *l*, *p* – 1, *p*) //Partitioning algorithm
omp task shared(*new_midR*)
new_midR = *RPar* (*arr*, *p* + 1, *r*, *p*) //Partitioining algorithm
omp taskwait
new_midC = *MSwap* (*arr*, *new_midL*, *new_midR*, *p*)
omp task
DPPartition (arr, *l*, *new_midC* – 1)
omp task
DPPartition (*arr*, *new_midC* + 1, *r*)

Algorithm 2. MO5

```
Input: arr, l, r
    Initialisation:
1: p = l + (r − l)/2
2: q1 = l + (p − l)/2
3: q3 = m + (r − p)/2
4: SORT (arr[l],arr[q1],arr[p],arr[q3],arr[r])
```

Algorithm 3. LPar

```
Input: arr, l, r, p
Output: indexl
    Initialisation:
 1: val = arr[p]
 2: indexl = l
 3: for i = l; i <= r;i = i + 1 do
 4:     if arr[i] <= val then
 5:         swap(arr[i], arr[indexl])
 6:         indexl = indexl + 1
 7:     end if
 8: end for
 9: return indexl
```

Algorithm 4. RPar

```
Input: arr, l, r, p
Output: indexr
    Initialisation:
 1: val = arr[p]
 2: indexr = r
 3: for j = r; j >= l;j = j − 1 do
 4:     if arr[j] > val then
 5:         swap(arr[j], arr[indexr])
 6:         indexr = indexr − 1
 7:     end if
 8: end for
 9: return indexr
```

Algorithm 5. MSwap

```
Input: arr, l, r, p
Output: i or j
    Initialization:
 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
```

The *DPPartition* begins with comparing size of subarray with sorting cutoff size c. While subarray size is larger than c, *MO*5 function is executed to select a pivot. This function selects the data at left, quarter, middle, 3rd quarter and right positions in the subarray. Then, sort those selected data and choose the middle position of them as pivot. Next, *LPar* and *RPar* functions are executed using the pivot position p. The tasking construct (*omp task*) is implemented to both functions and new positions of pivot (*new_midL* and *new_midR*) are returned. Note that, *new_midL* and *new_midR* are declared as shared variables which can be accessed after returning. Moreover, *omp taskwait* is executed to synchronize both left and right. After that, *MSwap* function is executed to swap the data which is greater than pivot from the left partition and less than or equal to pivot from the right. It returns a new pivot position (*new_midC*) of this level. Finally, *DPPartition* is run in parallel on the left and right subarrays using *omp task*. In addition, the subarray, which is smaller than c, sequential *STLSort*/*qsort* function is forked in parallel independently.

3.1 Dual Parallel Partitioning phase

There are two pointers in the *Dual Parallel Partitioning phase* which are used to partition the data in *LPar* for left partition and *RPar* for right partition. The pointers in each function traverse in the same direction. In *LPar* function, it traverses from left most (l) to the middle−1 ($p - 1$) of subarray. On the other hand, the pointer of *RPar* traverses from right most (r) to the middle+1 ($p + 1$).

There are *indexl* and *indexr* which split the data that are less than and greater than p. Moreover, i and j are used to divide partitioned and unpartitioned. In *LPar* function, *arr*[i] and *arr*[p] are compared every iteration. While *arr*[i] is less than or equal to *arr*[p], *arr*[i] is swapped with *arr*[*indexl*]. Then, *indexl* is increased by 1.

In the *RPar* function, *arr*[j] and *arr*[p] are compared every iteration. While *arr*[j] is greater than p, *arr*[j] is swapped with *arr*[*indexr*]. After that, *indexr* is decreased by 1. Finally, *indexl* and *indexr* are returned to *DPPartition* as *new_midL* and *new_midR*, respectively. This phase is demonstrated as shown in the 2nd to 4th line of Figure 1.

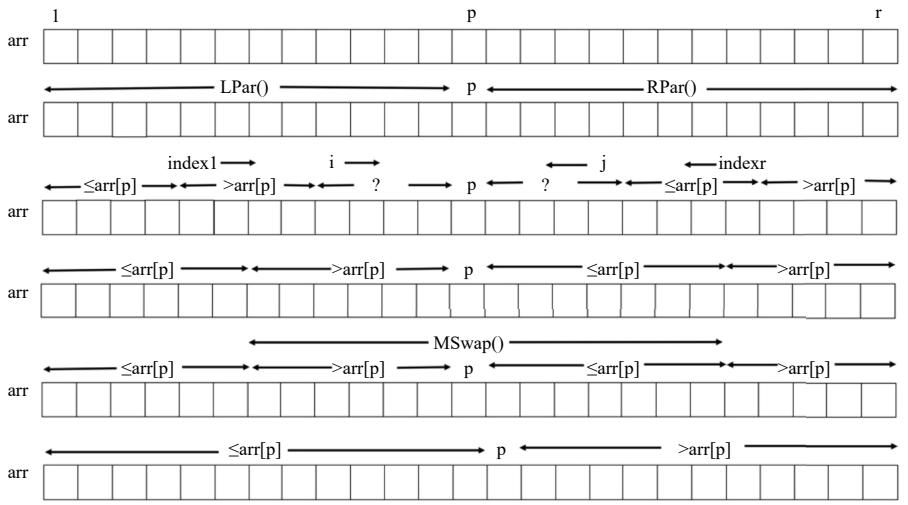

Figure 1.
The illustration of *DPPartition*

3.2 Multi-Swap Phase

The unpartitioned (sub)array is divided into two subarrays with *arr*[*p*] in the previous phase. It requires synchronization before merging by *Multi-Swap phase* using *omp taskwait*. There is data which is greater than pivot in the left partition and data which is less than pivot in the right partition. The data are only swapped without any comparisons until *i* or *j* is at *p* position.

It starts with *i* and *j* are initialized to the left position *l* and right position *r*, respectively. Next, *arr*[*i*] and *arr*[*j*] are swapped and then *i* and *j* are moved to the next position. This iteration continues until *i* is greater than *p* or *j* is less than *p*. Finally, if *i* is greater than *p*, *arr*[*j*] is swapped with *arr*[*p*] and return *j* as a new pivot position. On the other hand, *arr*[*i*] is swapped with *arr*[*p*] and returns *i* as a new pivot position. The *Multi-Swap phase* is demonstrated as shown in the 5th line of Figure 1.

3.3 Sorting phase

The data which are partitioned with *Dual Parallel Partitioning* and *Multi-Swap phases* successfully and smaller than sorting cutoff size are sorted by sorting function (*qsort* or *STLSort*) in parallel. The data are sorted using OpenMP parallel tasks by forking thread without blocking. The worker thread is joined with its master thread after the data are sorted automatically.

3.4 Lomuto's vs Hoare's algorithm in Dual Parallel Partitioning phase

In this algorithm, the array is divided into two halves. Then we use *Lomuto's partitioning algorithm* on the left half whose index is run from left to the middle (Algorithm 3) and the right half whose index is run from right to middle (Algorithm 4). Moreover, we replace *Hoare's partitioning algorithm* in *Dual Parallel Partitioning Phase* and compare this algorithm with our proposed algorithm.

4. Experiments, results and discussions

This section shows the setup of experiments, results and discussions. The metrics such as Run Time, Speedup, Speedup per core/thread and *Perf* profiling results are demonstrated and discussed.

4.1 Experiments setup

We compare *DPPSort* function with *qsort* ($DPPSort_{qsort}$) and *STLSort* ($DPPSort_{STL}$) as sorting cutoff Algorithms, *qsort* and *STLSort* functions. The random, few unique, nearly sorted and reversed 32-bit and 64-bit unsigned integers are sorted in this experiment. The 50, 100, 200 million data N are generated randomly in every experiment. The sorting cutoff c are $N/2$, $N/4$, $N/8$, $N/16$, $N/32$ and $N/64$. Each parameter is tested and averaged as Run time(sec). All sorting algorithms are run on Intel i7-6770 which consists of 4 cores (8 threads) with 32 GB main memory.

4.2 Results

There are 3 metrics which are used to measure the performance of $DPPSort_{qsort}$ and $DPPSort_{STL}$. (1) Run Time (2) Speedup and Speedup per core/thread (3) *Perf* profiling tool is used to profile cache misses and branch load misses.

4.2.1 Run time. Total run time of *DPPSort* ($T_{DPPSort}$) consists of *DPPartition* and its sequential sorting algorithm run time. Note that, *DPPartition* run time is partitioning algorithm run time ($T_{DPPartition}$) and its sequential sorting algorithm run time (T_{sort}) is run time of *qsort* or *STLSort* which is chosen as sorting algorithm. Total run time of *DPPSort* is shown in equation (1) .

$$T_{DPPSort} = T_{DPPartition} + T_{sort} \tag{1}$$

DPPSort is faster than *qsort* and *STLSort* algorithms. The fastest algorithm is $DPPSort_{STL}$. Its run time is only 3.97 and 2.87 seconds to sort 200 million Uint64 data using *qsort* and *STLSort*, respectively as a sorting cutoff algorithm. Run time of *qsort* and *STLSort* function are 26.06 and 16.06 seconds, respectively.

Sorting cutoff size is proportional to run time and effects with run time complexity. Run time of $DPPSort_{qsort}$ and $DPPSort_{STL}$ which run random data are the fastest at $c = N/32$ and $c = N/8$, respectively. Run time of $DPPSort_{qsort}$ and $DPPSort_{STL}$ are illustrated in Figure 2a and b. *DPPSort* run time slightly falls while the sorting cutoff size is smaller in any sorting cutoff algorithm. We can notice that the best sorting cutoff size is at $c = N/32$ on $DPPSort_{qsort}$. The *Dual Partitioning phase* should be run until its partitions are small enough. Then, sort the partitions using a sorting cutoff algorithm in parallel. On the other hand, the best sorting cutoff size of $DPPSort_{STL}$ is at $c = N/8$. Its run time depends on the sorting cutoff algorithm.

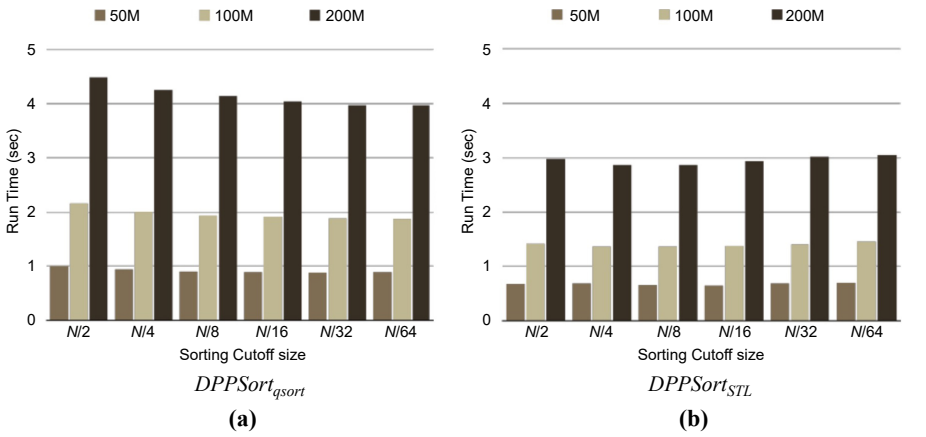

Figure 2.
Run time of $DPPSort_{qsort}$ in seconds vs Sorting cutoff size of various N ($M = 10^6$, Uint64 random)

STLSort is faster than *qsort* significantly and can sort the medium data size efficiently. Therefore, it is not needed to split the data into small sizes.

Table 1 shows average run time of each distribution at $c = N/8$. We can notice that reversed distribution run time of $DPPSort_{qsort}$, *STLSort* and *qsort* are the fastest compared with other distributions. It can be due to every algorithm using *Hoare's partitioning* as the partitioning algorithm. This algorithm swaps the most left and right data that is the greatest and lowest, respectively. Then, its indices run to the middle position and finally sort the data.

The $DPPSort_{STL}$ run time of reversed, nearly sorted and few unique distributions are almost the same. It can be due to $T_{DPPartition}$ being lower than the other distributions which affects $T_{DPPSort}$. The $DPPSort_{STL}$ run time of reversed, nearly sorted and few unique are almost the same. The run time of $DPPSort_{qsort}$ of reversed is slightly faster than the others. It can be due to the *DPPartition* algorithm which reduces $T_{DPPartition}$ that affects $T_{DPPSort}$.

4.2.2 Speedup. Speedup is the metric which can be measured by the performance of the *DPPSort* algorithm. It is the fraction of original run time versus the *DPPSort* algorithm run time as shown in equation (2).

$$Speedup = \frac{Run\ Time_{original}}{Run\ Time_{DPPSort}} \tag{2}$$

Our experiments show *DPPSort* with *qsort* and *STLSort* is a sorting cutoff algorithm. Average Speedup of $DPPSort_{qsort}$ and $DPPSort_{STL}$ (Uint64 random) are shown in Table 2.

Algorithms	Distributions	Run time (sec) $N = 50 \times 10^6$	$N = 100 \times 10^6$	$N = 200 \times 10^6$
$DPPSort_{STL}$	Random	0.66	1.37	2.87
	Reversed	*0.55*	*1.14*	2.50
	Nearly sorted	0.56	1.15	*2.44*
	Few unique	*0.55*	*1.14*	2.45
$DPPSort_{qsort}$	Random	0.91	1.93	4.14
	Reversed	*0.52*	*1.13*	*2.41*
	Nearly sorted	0.73	1.54	3.30
	Few unique	0.69	1.51	3.17
STLSort	Random	3.84	7.76	16.06
	Reversed	*0.56*	*1.15*	*2.40*
	Nearly sorted	3.09	6.49	13.50
	Few unique	1.54	3.13	6.45
qsort	Random	6.03	12.53	26.06
	Reversed	*1.72*	*3.57*	*7.46*
	Nearly sorted	3.51	7.28	15.10
	Few unique	3.97	8.15	16.76

Table 1. Average run time (Uint64 data) of each distribution at $c = N/8$ of $DPPSort_{qsort}$, $DPPSort_{STL}$, *STLSort* and *qsort*

Algorithms	N (10^6)	C $N/2$	$N/4$	$N/8$	$N/16$	$N/32$	$N/64$
$DPPSort_{qsort}$	50	6.02	6.41	6.64	6.76	*6.82*	6.78
	100	5.81	6.25	6.48	6.55	6.66	*6.69*
	200	5.80	6.12	6.29	6.44	*6.56*	6.55
$DPPSort_{STL}$	50	5.65	5.59	5.84	*5.88*	5.54	5.45
	100	5.49	*5.69*	*5.69*	5.61	5.50	5.33
	200	5.39	*5.60*	*5.60*	5.47	5.32	5.26

Table 2. Average Speedup of $DPPSort_{qsort}$ and $DPPSort_{STL}$ versus c (Uint64 Random)

Speedup of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$ significantly. It can be due to partitioning run time of both $DPPSort_{qsort}$ and $DPPSort_{STL}$ are similar. However, sorting run time of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$ significantly. In addition, Speedup of $DPPSort_{qsort}$ is greater than $DPPSort_{STL}$.

Figure 3a shows Speedup of $DPPSort_{qsort}$ of various sorting cutoff size. It can be noticed that Speedup increases until $c = N/32$. However, Speedup of $DPPSort_{STL}$ reaches the highest at $c = N/8$ as shown in Figure 3b. It can be due to the fraction between partitioning and sorting. The *qsort()* is quicksort algorithm which divides the data using *Hoare's algorithm* then sort them with insertion sort when the subarrays are smaller. The *STLSort* is *Introsort* algorithm which contains quicksort and heapsort in the partitioning step. It divides the data into subarray using quicksort and use heapsort to sort partially while partitioning. When the subarray is small enough, the insertion sort is called to sort that subarray. The *STLSort()* can sort the large data better than *qsort()* because of its partitioning algorithm.

The best Speedup of $DPPSort_{qsort}$ and $DPPSort_{STL}$ at Uint32 data size are at $c = N/64$ and $c = N/32$, respectively. Moreover, the best Speedup of $DPPSort_{qsort}$ at Uint64 data size is between $c = N/32$ and $c = N/64$. However, the best Speedup of $DPPSort_{STL}$ at Uint64 data size is lower than the other parameters. It is between $c = N/4$ and $c = N/8$. We can notice that the significant parameters are sorting cutoff algorithm, sorting cutoff size and data type. Sorting cutoff algorithm size is proportional to sorting cutoff algorithm. While *qsort* is a sorting cutoff algorithm, the sorting cutoff size is smaller than *STLSort*. The important parameter is data type. The best Speedup of Uint64 is larger compared with Uint32 data.

4.2.3 Speedup per core and thread. Speedup per core and Speedup per thread are the metrics used to measure the parallel algorithm. If these metrics are higher, the algorithm can use the processor core efficiently. It is the fraction of Speedup of any sorting algorithm versus CPU cores. Note that, Speedup per thread is the fraction of Speedup of any sorting algorithm versus Hardware threads.

Speedup per core and thread results of related algorithms are calculated and available on https://github.com/DPPSort/AnalyzeOptDPPSort/blob/main/Tables/t3.pdf. It can be noticed that Speedup per core of our algorithm is higher than the others. Speedup per core of $DPPSort_{qsort}$ and $DPPSort_{STL}$ is up to 1.71 and 1.47 which are greater than 1.00. This means it can use the processor core efficiently.

However, the 4-core with 8-thread i7-6770 is used to run our *DPPSort* in our experiment. The Speedup per thread is calculated because of the Hyperthreading Technology of Intel

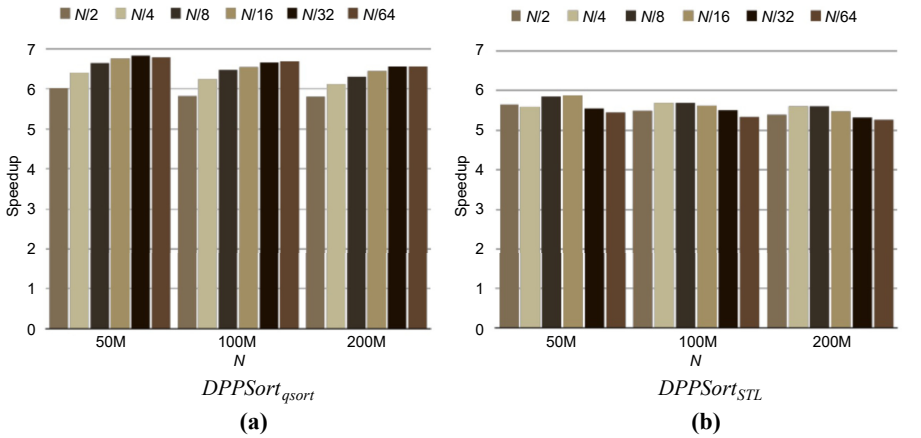

Figure 3.
Speedup of $DPPSort_{qsort}$ and $DPPSort_{STL}$ vs N ($M = 10^6$, Uint64 Random) of various sorting cutoff size

CPU. Speedup per thread of $DPPSort_{qsort}$ and $DPPSort_{STL}$ are up to 0.85 and 0.74, respectively. This means the hardware threads of the processor run this algorithm effectively compared with the others.

We can notice that the Speedup/thread of other algorithms such as *PPMQSort* [13] (0.77), *Introqsort* [11] (0.74), *DF IntroSort* [10] (0.51), *psort* [15, 16] (0.46), *MultiSort* [9] (0.43) and *HDPSort* [18] (0.31) are less than *DPPSort*. This result reflects the efficiency of the *DPPSort* algorithm that contains *DPPartition* and its sorting algorithm. The *DPPartition* affects Speedup/thread that means this algorithm can run on Simultaneous multithreading (SMT) better than the others.

4.2.4 Perf profiling results. *Perf* profiling tools can be used to profile the cache misses and branch load misses which affects run time of the algorithm. $DPPSort_{STL}$, $DPPSort_{qsort}$, *STLSort* and *qsort* are profiled with random, reversed, nearly sorted and few unique data distributions. *Perf* results are measured and available on https://github.com/DPPSort/AnalyzeOptDPPSort/blob/main/Tables/t4.pdf and https://github.com/DPPSort/AnalyzeOptDPPSort/blob/main/Tables/t5.pdf.

$DPPSort_{qsort}$, $DPPSort_{STL}$, *STLSort* and *qsort* are run with *Perf* Profiling tools at Uint32 and Uint64 with $N = 200 \times 10^6$ and $c = N/8$. Cache misses of Uint32 data are slightly less than Uint64 data. It can be due to the data type of the Uint32 which is smaller than the Uint64. On the other hand, data types do not affect branch load misses. Therefore, branch load misses of Uint32 and Uint64 are very similar.

It can be noticed that the run time of the reversed distribution is the lowest compared with the others. Its branch load misses metric is the lowest. Moreover, $DPPSort_{qsort}$ run time of reversed distribution is faster than $DPPSort_{STL}$. Branch load misses metric of $DPPSort_{qsort}$ is lower than $DPPSort_{STL}$. This can be due to the *Hoare's algorithm* in *qsort* which is the sorting cutoff algorithm.

We can notice that the random distribution of every algorithm is the slowest. Both cache misses and branch load misses metrics are the greatest.

There are 2 important metrics which affect the run time of the sorting algorithm. The first priority metric is branch load misses. While the branch load misses metric is greater, its run time is greater than the lower one. If branch load misses of two algorithms are about the same, the second priority metric is cache misses.

4.3 Our proposed vs Hoare's partitioning algorithm in Dual Parallel Partitioning phase results

In this paper, we replace and compare our proposed algorithm with *Hoare's partitioning algorithm*. The results of $DPPSort_{STL}$ which uses our proposed and *Hoare's algorithms* are available on https://github.com/DPPSort/AnalyzeOptDPPSort/blob/main/Tables/t6.pdf. In this experiment, the Uint32 200 million data are run with different sorting cutoff which $c = N/2$, $N/4$, $N/8$, $N/16$, $N/32$ and $N/64$.

It can be noticed that run time of $DPPSort_{STL}$ which uses *Hoare's partitioning algorithm* as partitioning algorithm, is faster than our proposed partitioning algorithm at $c = N/2$ and $N/4$. Moreover, its standard deviation is less than ours. Our proposed partitioning algorithm is faster than or equal to *Hoare's algorithm* at $c = N/8$, $N/16$, $N/32$ and $N/64$. In addition, the standard deviation of our proposed partitioning algorithm is less than *Hoare's algorithm* at $N/16$, N/32 and $N/64$. It means that while the sorting cutoff is smaller, our proposed partitioning algorithm is faster than *Hoare's algorithm* and it is stable more than *Hoare's*. It can be due to our proposed partitioning algorithm using *Lomuto's partitioning algorithm* inside. However, our proposed partitioning algorithm uses *Hoare's* style outside. Therefore, its locality is better than *Hoare's partitioning algorithm*.

5. Conclusion and future work

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

DPPSort is applied and runs on Intel core i7-6770 with Linux system. It is faster than other standard sorting algorithms like *qsort* and *STLSort*. Speedup on random distribution is up to 6.82× and 5.88×, respectively. Note that, Speedup per thread of 0.85× can be obtained. Its performance depends on the sorting cutoff algorithm, its size, data type and data distribution.

The first priority metric that affects run time is branch load misses. The second one is cache misses. It affects run time significantly while branch load misses of compared algorithms are the same.

DPPSort can be improved in the future works. We can apply this algorithm to the larger machines and heterogeneous systems. Moreover, we can implement this algorithm to the heterogeneous system to achieve Speedup of the algorithm.

References

1. Antony C, Hoare R. Algorithm 64: quicksort. Commun ACM. 1962; 4: 321. doi: 10.1145/366622.366644.
2. Sedgewick R. Implementing quicksort programs. Commun ACM. 1978; 21(10): 847-57. doi: 10.1145/359619.359631.
3. Hoare CAR. Algorithm 64: quicksort. Commun ACM. 1961; 4(7): 321.
4. OpenMP. OpenMP 4.5 specification. 2015. Available from: http://www.openmp.org/wp-content/uploads/openmp-4.5.pdf.
5. De Melo AC. The new linux'perf'tools. Slides from Linux Kongress. 2010; 18: 1-42.
6. Musser DR. Introspective sorting and selection algorithms. Softw Pract Exper. 1997; 27(8): 983-993. doi: 10.1002/(SICI)1097-024X(199708)27:83.0.CO;2-#.
7. Heidelberger P, Norton A, Robinson JT. Parallel quicksort using fetch-and-add. IEEE Trans Comput. 1990; 39(1): 133-8. doi: 10.1109/12.46289.
8. Tsigas P, Zhang Y. 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; 2003. 372-81.
9. Süß M, Leopold C. A user's experience with parallel sorting and OpenMP. In: Proceedings of the sixth European workshop on OpenMPEWOMP 2004; 2004. 23-38.
10. Traoŕe D, Roch J-L, Maillard N, Gautier T. Deque-free work-optimal parallel STL algorithms, Euro-Par 2008–. Parallel processing. Berlin, Heidelberg: Springer; 2008. 887-97.
11. Kim KJ, Cho SJ, Jeon J-W. Parallel quick sort algorithms analysis using OpenMP 3.0 in embedded system. In: 11th International Conference on Control, Automation and Systems; Gyeonggi-do, Korea: KINTEX; 2011. 757-61.
12. Saleem S, Lali MI, Nawaz MS, Nauman AB. Multi-core program optimization: parallel sorting algorithms in intel Cilk Plus. Int J Hybrid Inf Technol. 2014; 7(2): 151-64. doi: 10.14257/ijhit.2014.7.2.15.
13. Ranokphanuwat R, Kittitornkun S. Parallel partition and merge quicksort (PPMQSort) on multicore CPUs. J Supercomput. 2016; 72(3): 1063-91. doi: 10.1007/s11227-016-1641-y.
14. Rattanatranurak A, Kittitornkun S. A multistack parallel (MSP) partition algorithm applied to sorting. J Mob Multimedia. 2020; 16(3): 293-316.
15. Man D, Ito Y, Nakano K. An efficient parallel sorting compatible with the standard qsort. In: International Conference on Parallel and Distributed Computing, Applications and Technologies; Hiroshima, Japan; 2009. 512-7.

16. Man D, Ito Y, Nakano K. An efficient parallel sorting compatible with the standard qsort. Int J Found Comput Sci. 2011; 22(5): 1057-71. doi: 10.1142/S0129054111008568.

17. Mahafzah BA. Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. J Supercomput. 2013; 66(1): 339-63. doi: 10.1007/s11227-013-0910-2.

18. Taotiamton S, Kittitornkun S. Parallel hybrid dual pivot sorting algorithm. In: 14th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON); IEEE; 2017. 377-80.

19. Chen Y, Li K, Yang W, Xiao G, Xie X, Li T. Performance-aware model for sparse matrix-matrix multiplication on the Sunway TaihuLight supercomputer. IEEE Trans parallel Distrib Syst. 2018; 30(4): 923-38.

20. Chen Y, Xiao G, Özsu MT, Liu C, Zomaya AY, Li T. aeSpTV: an adaptive and efficient framework for sparse tensor-vector product kernel on a high-performance computing platform. IEEE Trans Parallel Distributed Syst. 2020; 31(10): 2329-45.

21. Cederman D, Tsigas P. GPU-quicksort: a practical quicksort algorithm for graphics processors. J Exp Algorithmics. 2010; 14: 1-4.

22. Kozakai S, Fujimoto N, Wada K. Efficient GPU-implementation for integer sorting based on histogram and prefix-sums. In: 50th International Conference on Parallel Processing; 2021. 1-11.

Further reading

23. He M, Wu X, Zheng SQ. An optimal and processor efficient parallel sorting algorithm on a linear array with a reconfigurable pipelined bus system. Comput Electr Eng. 2009; 35(6): 951-65. doi: 10.1016/j.compeleceng.2008.11.020.

Corresponding author

Sirilak Ketchaya can be contacted at: sirilak.ke@ssru.ac.th