Optimizing MultiStack Parallel (MSP) Sorting Algorithm

Apisit Rattanatranurak and Surin Kittitornkun*

Dept. of Computer Engineering, School of Engineering, King Mongkut's Institute of Technology Ladkrabang, Bangkok, Thailand
E-mail: apisit.ra@ssru.ac.th; surin.ki@kmitl.ac.th
**Corresponding author*

Received 05 February 2021; Accepted 15 March 2021;
Publication 18 June 2021

Abstract

Mobile smartphones/laptops are becoming much more powerful in terms of core count and memory capacity. Demanding games and parallel applications/algorithms can hopefully take advantages of the hardware. Our parallel MSPSort algorithm is one of those examples. However, MSPSort can be optimized and fine tuned even further to achieve its highest capabilities. To evaluate the effectiveness of MSPSort, two Linux systems are quad core ARM Cortex-A72 and 24-core AMD ThreadRipper R9-2920. It has been demonstrated that MSPSort is comparable to the well-known parallel standard template library sorting functions, i.e. Balanced QuickSort and Multiway MergeSort in various aspects such as run time and memory requirements.

Keywords: Partition, sort, multithread, OpenMP, optimize, synchronization.

1 Introduction

Smartphones' CPUs are becoming multicore towards manycore to support a variety of multimedia applications. These mobile manycore CPUs are soon pervasive in both notebook and tablet computers. Therefore, basic computing algorithms shall be adapted to exploit that. Parallel sorting and

Journal of Mobile Multimedia, Vol. 17_4, 723–748.
doi: 10.13052/jmm1550-4646.17412

data partitioning algorithms [4, 5, 9, 10] are based on the well known single-pivot Hoare's algorithm [1]. It is also known as divide and conquer (D&Q) behavior of QuickSort.

Those algorithms can be broken into two phases: first level partition and partition & sort phases. The first phase is to compare and swap (C&S) between elements in parallel followed by a cleanup algorithm and sequential partition. The second phase recursively partitions two long subarrays and finally sorts the short ones in single thread.

The first level partition is the bottleneck of D&Q Hoare's algorithm. This paper intends to tackle this problem with simple multithreading techniques while minimizing the unnecessary memory accesses. Based on our MultiStack Parallel Sorting (MSPSort) [7], we optimize and fine tune it further. Unlike other F&A parallel block-based QuickSort algorithms, MSPSort enforces data parallelism with left and right stacks of data blocks private to each thread. In addition, MSPSort is based on barrier synchronization to handle the classic skew pivot problem. However, the cleanup algorithm can be improved even more.

Our key contributions are as follows. First, the MSPPartition can be adaptive with respect to pivot value. It can adapt to the skew pivot such that fewer unnecessary recursive calls can be reduced while cleaning up the first level partition phase. Second, dynamic thread allocation can increase CPU threads to handle much longer subarrays due to imbalanced pivot. This can be beneficial to huge workloads on manycore CPUs. Third, the number of partitioning threads or CPU cores can be adjusted to support a wide variety of data types. As a result, fewer memory accesses are required instantaneously thus lower pressure to the memory hierarchy and power consumption. Workloads can be shared among partitioning threads where more free cores are available to sort subarrays in parallel. Our optimization methods are considered global because they can be effectively applied to any manycore CPU system.

We shall present the paper in the following order. Section 2 reviews related background and previous work of parallel D&Q sorting algorithms. The optimized and fine tuned MSPPartition and MSPSort are elaborated in Section 3. Consecutively, experiment results are compared on AMD R9-2920 and Raspberry Pi4 systems. The last section is Conclusions and Future Work.

2 Background & Related Work

This section consists of the following subsections, *Standard Template Library Sequential/Parallel Sorting Functions* and *Anatomy of QuickSort Algorithms*.

2.1 Standard Template Library Sequential/Parallel Sorting Functions

The Standard Template Library $(STL)Sort$ is a sequential sorting function for any data type. It is available in almost C++ compilers and prototyped as follow.

```
std::sort(RandomAccessIterator first,
          RandomAccessIterator last);
```

On the other end of spectrum, Balanced QuickSort or BQSort as mentioned earlier is a classic parallel/multithreaded quicksort algorithm [9]. Each thread assign blocks of data to compare&swap (C&S). A double ended queue of each thread holds the block boundaries and allows other threads to access from the second end. MultiWay Merge Sort or MWSort equally divides data into several subarrays and $std :: sort$ them [9]. Each subarray can be independently sorted in parallel and stored in a temporary array. It requires an extra space of memory with complexity of N where N is the data size. The Parallel MultiWay merging algorithm can yield the final sorted array. As a result, its run time is more stable compared with QuickSort algorithm.

2.2 Anatomy of Parallel QuickSort Algorithms

A number of parallel QuickSort algorithms have been proposed by [2, 4–6, 8–10] and ordered in time series. The recent one is called MSPSort by us [7]. All of them can be decomposed into two phases: first level partition and partition & sort phases. The **First Level Partition Phase** is the most challenging task for parallelization. It consists of parallel compare and swap (C&S) between elements followed by a cleanup algorithm and sequential partition. Due to its comparison-based nature and global memory accesses, it is limited by latency of address translation look aside buffer (TLB) and page table all the way to cache misses. It can be regarded as *memory bound* problem [8]. In addition, CPU hardware constraints for some basic data types such as floating-point numbers still exist. The second phase called **Partition&Sort** is to recursively partition two long subarrays while traversing the resulting tree of subarrays in multithread and to finally sort a shorter one by a single thread. Only major algorithms are summarized and compared with ours as follows.

In 1990, Heidelberger et al. [5] first proposed a block-based parallel partition algorithm on shared memory multiprocessor system. Reserving blocks of data to C&S must be synchronized using F&A instruction. Although it is initially in-place, the cleanup phase needs two extra buffers to keep track

of elements' indices to be swapped. The next phase can be accomplished using two global queues. The first queue keeps the longer subarrays to be partitioned in parallel with proportional number of processors. The second queue keeps shorter ones to be partitioned sequentially and consequently sorted. Several scheduling algorithms can be applied. The entire algorithm was simulated rather than implemented.

PQuickSort [10] was first implemented a block based parallel partition algorithm on a shared memory multiprocessor system in 2003. Each processor reserves a pair of blocks from both left and right sides to C&S data elements. The process continues to reserve one more fresh block from either on left or right side. All processors stop until no fresh block is available. Some unfinished blocks must be swapped to separate between the left and right finished blocks. The sequential partition eventually finishes the first-level partition. The next step assigns a number of processors proportional to subarrays' sizes. A special stack is instantiated to store subarrays and shared among processors to partition and finally sorted them. Its run time complexity is $O(\frac{N}{c}\log\frac{N}{c}+\frac{N}{c})$ where N and c are array size and number of CPU cores, respectively.

The BQSort [9] as mentioned earlier is also block based and recursive C&S. Fetch and add (F&A) synchronization is needed every time a data block is reserved. All threads stop when their C&S indices get closer within a block distance. Some data swaps must be carried out to prepare for the next recursive function call. Eventually, the sequential partition finishes the first level partition phase. Next, each thread's local deque (double ended queue) can keep track of subarrays to be partitioned with shorter partition first (SPF) traversal. As a result, an idle thread can steal workload from the other end of a non-empty deque. Its run time complexity is $O(\frac{N}{c}\log N + c\log c)$.

A year later, Frias and Petit [4] used the same F&A synchronization scheme. However, they propose a new cleanup algorithm to avoid extra C&S. They apply a complete binary tree structure to keep track of the unfinished blocks so that they can be shared to balance workload on all CPU cores. Left data elements shall be swapped only with the right ones according to the leaves and finish up with sequential partition.

On the GPU (Graphic Processing Unit) computing side, these references [2, 6, 8] have all implemented parallel QuickSort algorithms based on NVIDIA GPUs using the CUDA Application Program Interface. One of the first papers implementing QuickSort on GPU was published by Sengupta, et al. in 2007 [8]. They are not in-place like CPU-based algorithms, Most of them consist of two phases. Phase I, the input array is divided into segments

to fit the GPU local memory. Each section is assigned to a block of threads to compare against the pivot value to create a bit vector or C&S within a segment. The resulting two subsegments will be copied with synchronizations to form left and right subarrays in a buffer/shared memory. Phase II, either iterative or recursive partition continues and utilize different structures such as stack to keep track of the partitioned subarrays. Short subarrays are then sorted given a threshold.

Our MSPPartition/Sort [7] is a simple yet efficient block based C&S algorithm. Each thread has its own private left and right stacks of data blocks to enforce data parallelism without any F&A instruction. When either left or right stack of each thread runs out, the thread stops temporarily to synchronize with other threads. As a cleanup algorithm, recursive MSPPartition or a sequential one can be invoked while sacrificing C&S some elements again thanks to current smart branch prediction. In the next phase, combinations of breadth first and depth first traversals can partition to exploit cache locality and consequently $std :: sort$ them. No work stealing is necessary.

3 Optimized MSPSort Algorithm

Various factors of our MSPPartition can affect its performance such as block size, how to handle the leftover data, how to balance the workload between left and right partitions, etc.

MSPPartition and MSPSort can be optimized and fine-tuned in the following aspects: block size, pivot value skew handling, number of threads in the first level, middle levels and final level of recursion. We can review and improve the MSPartition in pseudocode as shown in Algorithm 1.

3.1 Block Size B

This *MultiStack Parallel Partition* function [7] The Recursive MultiStack Parallel Partition Phase can be decomposed of 2 steps: Parallel Stacked Blocks Partition Step and Unfinished Blocks Partition Step. The *Parallel Stacked Blocks Partition Step* divides an unsorted (sub)array $A = A[0], A[1], \ldots, A[N-1]$, into left and right sides according to the pivot index p. Each side is again divided into blocks of $b = B/sizeof(Type)$ elements from both ends (Algorithm 1, line 4) where B is a block size in KB (Kilo Bytes). Data blocks from left and right sides are assigned to τ threads in round robin to enforce data parallelism and avoid overlapping operations. Within its own assigned data blocks, each thread can C&S data just like Hoare's algorithm. Therefore, there is no data race among threads.

Algorithm 1: Optimized Multi-Stack Partition Algorithm

```
1  Function MSPPartition(A, i_L, j_R, p, τ)
2      N_L ← (p − i_L)/b                          // Number of left blocks
3      N_R ← (j_R − p)/b                          // Number of right blocks
4      for i ← 0 to N_L − 1, i++ do
5          L_s[i mod τ].push(i_L + i, i_L + i + b)   // Push left blk indices
6      end
7      for i ← 0 to N_R − 1, i++ do
8          R_s[i mod τ].push(j_R − i − b, j_R − i)   // Push right blk indices
9      end
10     OpenMP parallel for private(i, j, l_b, r_b)
11     for t ← 0 to τ − 1 do
12         while !L_s[t].empty() && !R_s[t].empty() do
13             (i, l_b) ← L_s[t].pop()              // Pop left block indices
14             (r_b, j) ← R_s[t].pop()              // Pop right block indices
15             do
16                 while A[i] ≤ A[p] && i ≤ l_b do
17                     i++                          // Increase i index
18                 end
19                 while A[j] > A[p] && j ≥ r_b do
20                     j- -                         // Decrease j index
21                 end
22                 if i ≤ l_b && j ≥ r_b then
23                     SWAP(A[i++], A[j- -])        // Swap and continue
24                 end
25             while i ≤ l_b && j ≥ r_b
26             if i > l_b then
27                 R_s[t].push(r_b, j)              // Left side runs out
28             end
29             else if j < r_b then
30                 L_s[t].push(i, l_b)              // Right side runs out
31             end
32         end
33     end
34     l_min ← min(L_s[t], ∀t)                      // Get the left most index
35     r_max ← max(R_s[t], ∀t)                      // Get the right most index
36     μ ← GetMu (i_L, j_R, r_max, l_min, u_sort, τ)   // Get mu threads
37     π ← GetIndex (i_L, j_R, r_max, l_min, p)     // Find the new pivot index
38     SWAP(A[p], A[π])                             // Swap to new pivot index
39     if μ > 1 then
40         return MSPPartition(A, l_min, r_max, π, μ)   // Recursion again
41     end
42     else
43         return LomutoPartition(A, l_min, r_max, π)   // Serial Partition
44     end
45 EndFunction
```

3.2 Handling Pivot Value Skew in MSPPartition

To ideally balance workload and achieve maximum parallelism, each thread is initially assigned with about the same number of blocks. Each thread has its own private variables (local to each thread) i and j that are left and right indices within the current left and right blocks, respectively. In addition, two private variables, l_b and r_b are the boundaries of the current left and right blocks, respectively. The data from both sides are C&S within each iteration until either left or right index reaches its boundary (Alg.1, lines 15 to 24). Next, the current index pair either (i,l_b) or (r_b,j) of the latest unfinished block is pushed back to its corresponding stack (Alg.1, lines 27 and 30). In the next iteration, the boundaries of a new block and the unfinished one will be popped off to continue. This loop of each thread stops when either its left or right stack is empty (Alg.1, line 12).

After that, $l_{min} = min(L_s[t], \forall t)$ and $r_{max} = max(R_s[t], \forall t)$ of all τ threads, must be determined. The number of threads μ=**GetMu()** (Algorithm 1, line 36) corresponds to how many folds ($r_{max} - l_{min}$) is longer than u_{sort} and the current number of threads τ in Equation (1),

$$\mu = \begin{cases} \tau & , \mu' \geq \tau \\ \mu' & , \mu' < \tau \end{cases} \tag{1}$$

where

$$\mu' = \lfloor \frac{r_{max} - l_{min} + 1}{u_{sort}} \rfloor \tag{2}$$

Next, the *Unfinished Blocks Partition Step* must take care of the pivot value skew issue. This step is the most critical part due to the random distribution of data and pivot value selection of $MO5()$. Earlier [7], the number of blocks on both sides must equal such that $\pi = (r_{max} + l_{min})/2$. However, pivot value skew can lead to the time consuming recursive calls to eventually partition the unfinished part. In this optimized MSPSort, the target index π can be adaptive to the pivot value skew such that the number of recursions can be minimized. Therefore, it should be proportional to the finished (swapped) data elements as shown in Equation (3).

$$\pi = i_L + \frac{(l_{min} - i_L) \times (r_{max} - l_{min})}{(l_{min} - i_L + j_R - r_{max})} \tag{3}$$

where ($l_{min} - i_L$) and ($j_R - r_{max}$) represent the lengths of left and right already swapped parts, respectively and ($r_{max} - l_{min}$) is the length of clean-up part.

In summary, the long unfinished part due to extreme pivot value skew can specify the number of μ threads of **MSPPartition()** and estimate the next pivot index π from Equation (3) (Algorithm 1, line 40) in the *Unfinished Blocks Partition Step*. Otherwise, the Lomuto's Partition [3] sequentially returns the pivot index (Algorithm 1, line 43).

3.3 Partition Thread Ratio θ

Based on our observations in the previous work [7], faster partition time can be also achieved at $\tau < \tau_{max}$ as well. As a result, the initial number of threads to partition can be parameterized to $\theta \times \tau_{max}$ where θ is the Partition Thread Ratio. At line 2 of Algorithm 2, if $\theta < 1.00$, it shall relieve some pressure of the last level caches as well as the data TLBs where some partitioning threads must access data (memory) at various locations simultaneously.

Algorithm 2: Modified MSPSort Algorithm

```
1  Function Main()
2  |  MSPSort(A, 0, N − 1, θ × τ_max)    // MSPSort with θ × τ_max threads
3  End
4  Function MSPSort(A, i_L, j_R, τ)
5  |  p ← MO5(A, i_L, j_R)                          // p = Median of Five
6  |  if j_R − i_L > u_sort then
7  |  |  p ←MSPPartition(A, i_L, j_R, p, τ)  // Partition with τ threads
8  |  |  if τ/α > τ_min then
9  |  |  |  τ ← τ/α                          // Reduce τ threads by α
10 |  |  end
11 |  |  else
12 |  |  |  τ ← τ_min                       // Saturate at τ_min threads
13 |  |  end
14 |  |  if j_R − i_L > u_trav then
15 |  |  |  BF-MSPSort(A, i_L, j_R, p, τ)     // Breadth First τ threads
16 |  |  end
17 |  |  else
18 |  |  |  DF-MSPSort(A, i_L, j_R, p, τ)       // Depth First τ threads
19 |  |  end
20 |  end
21 |  else
22 |  |  OpenMP Task untied
23 |  |  std :: sort(A + i_L, A + j_R)             // Fork an untied thread
24 |  end
25 End
```

3.4 Dynamic Thread Allocation

The Dynamic Thread Allocation consists of two independent parameters: Thread Divider α and Thread Equalizer β. The Thread Divider $1 < \alpha < 3$ is between each recursive level. On the other hand, β is the Thread Equalizer of the next recursion level to compensate with extremely skew pivot from the previous recursion. That means longer subarray on either left or right side may require more threads than its default number. If Thread Equalizer β=ON, each side may get two different number of threads, τ_L and τ_R. Otherwise, β=OFF, threads on both sides are equal, τ_L=τ_R=τ in Equations (4) and (5).

$$\beta_L = \begin{cases} 1.5 & , l/r \geq 3 \\ 1 + \frac{l-r}{l+r} & , 1 \leq l/r < 3 \\ 1 & , otherwise \end{cases} \tag{4}$$

and

$$\beta_R = \begin{cases} 1.5 & , r/l \geq 3 \\ 1 + \frac{r-l}{l+r} & , 1 \leq r/l < 3 \\ 1 & , otherwise \end{cases} \tag{5}$$

3.5 Depth First Traversal Algorithms

In the **MSPSort()** function, Median of Five function $MO5()$ (Algorithm 2, line 5) selects a pivot index p and swaps it to the middle of subarray A. The **MSPPartition()** partitions the subarray A with respect to the pivot index p and finally returns it back (Algorithm 2, line 7). **MSPSort()** continues traversing the tree of subarrays with Breadth-First (Algorithm 2, line 15) by default. The details of **BF-MSPSort()** can be explored in Algorithm 3. It can be noticed that **MSPSort()** is invoked as two recursive untied OpenMP tasks (Algorithm 3, lines 3 and 7) corresponding to the left and right subarrays, respectively.

With the default BF traversal, any resulting shorter than u_{sort} subarray is sorted as an independent thread (Task) (Algorithm 2, line 23) using $std :: sort$ where $u_{sort} = U_{sort}/sizeof(Type)$, U_{sort} is Sorting Cutoff parameter, and $Type$ corresponds to the data type (in bytes) to be sorted.

To achieve as high parallelism as possible, BF traversal is of the choice. Therefore, its unpredictable execution order can be due to various factors such as the subarray sizes, data TLB misses/page faults and branch/memory stalls. However, cache locality can be exploited at almost the deepest level of

Algorithm 3: Breadth-First and Depth-First MSPSort

```
1  Function BF-MSPSort(A, i_L, j_R, p, τ)
2      τ_L ← β_L × τ                                   // τ_L threads
3      OpenMP Task untied
4      MSPSort(A, i_L, p − 1, τ_L)            // left subarray τ_L threads
5      τ_R ← β_R × τ                                   // τ_R threads
6      OpenMP Task untied
7      MSPSort(A, p + 1, j_R, τ_R)           // right subarray τ_R threads
8  End
9  Function DF-MSPSort(A, i_L, j_R, p, τ)
10     P_s.push(i_L, j_R)                  // Push the partition's boundary
11     while !P_s.empty() do
12         i_L, j_R ← P_s.pop()             // Pop the partition's boundary
13         if j_R − i_L > u_sort then
14             p ← MO5(A, , i_L, j_R)                  // p=Median of Five
15             p ←MSPPartition(A, i_L, j_R, p, τ)       // with τ threads
                                  // Decide to push which partition first
16             P_s.push(i_L, p − 1)          // Push left side first (RAL)
17             P_s.push(p + 1, j_R)         // Push right side later (RAL)
18         end
19         else
20             OpenMP Task untied
21             std :: sort(A + i_L, A + j_R)        // Fork an untied thread
22         end
23     end
24 End
```

recursion. We have been implementing DF (Depth First) traversal algorithm in **DF-MSPSort()** function. Once the subarray ($j_R - i_L$) is shorter than u_{trav} elements (Algorithm 2, line 18), MSPSort switches to DF traversal where $u_{trav} = U_{trav}/sizeof(Type)$ elements and U_{trav} is Traversal Cutoff in bytes. The pseudocode in Algorithm 3 illustrates RAL (Right Always) algorithm as an example.

There are two different approaches of BF-DF traversal, direction versus subarray size oriented. Both RAL and LAL (Left Always) are direction oriented. On the contrary, Longer Partition First (LPF) and Shorter Partition First (SPF) are size oriented. Several hybrid algorithms can be devised.

3.6 Minimum Threads τ_{min}

As the number of subarrays to be processed grows exponentially, deeper recursive level requires fewer threads. As a result, the number of software

threads τ is reduced to τ/α (Algorithm 2, line 12) and kept at $\tau = \tau_{min}$ threads where τ_{min} is the minimum number of threads. In order to share CPU cores with other threads while maximizing parallelism, τ_{min} should be fine tuned as well.

4 Experiments, Results & Discussions

The experiments on two different Linux systems. Experiment parameters are listed and rationalized. Consecutively, the obtained results are elaborated and discussed.

4.1 Experiment Setup

The fine-tuned MSPSort algorithm can be experimented on two totally different systems as listed in Table 1. They both run the same and G++ version. The number of cores c is reported by Linux System Monitor. Moreover, these systems widely differ in terms of memory size, technology and configuration. Nonetheless, to compare between non-L3 cache CPU and L3-cache CPU cores. R9-2920 represents high-end mobile systems with high CPU core count, large RAM capacity and shared L3 caches. On the other hand,

Table 1 Specifications of multicore CPUs, KB: Kilo Bytes, MB: Mega Bytes

CPU	Cortex A72	ThreadRipper
Name	Pi4	R9-2920
Clock (GHz)	1.5	3.50
c (cores)	4	24
RAM	8GB	64GB
Configuration	1×8GB	8×8GB
Technology	DDR4	DDR4
Frequency (MHz)	2133	2400
L1 I-Cache	4×48 KB 3 Way	12×64 KB 4 Way
L1 D-Cache	4×32 KB 2 Way	12×32 KB 8 Way
L2 Cache	1024 KB 16 Way	12×512 KB 8 Way
L3 Cache	–	4×8 MB 16 Way
64-bit OS	Raspberry Pi	Ubuntu
Version	Beta	18.04 LTS
Kernel	5.4.42-v8+	5.3.0-62-generic
G++	8.3.0	7.5.0
OpenMP	4.5	4.5

Table 2 Experiment parameters of $MSPSort$, BF: Bread-First, DF: Depth-First, M=10^6

Parameters	Values
Algorithms	$MSPSort$, $BQSort$, $MWSort$
Data Types	Uint32, Double
Random Dist	Uniform
GCC Optimization	O2
Data size N (Pi4)	20M, 50M, 100M, 200M
Data size N (R9)	200M, 500M, 1000M, 2000M
Traversal	RAL, LAL, LPF, SPF
Block size B	32, 64, 128, 256 KB
Sorting Cutoff U_{sort}	2, 4, 8, 16 MB
Traversal Cutoff U_{trav}	8, 16, 32, 64 MB
Multiplier m	1, 2
Minimum Thread τ_{min}	2, 3
Partition Thread Ratio θ	0.67, 0.75, 0.8, 0.9
Thread Divider α	4/3, 3/2, 2, 5/2

Raspberry Pi4 represents ordinary mobile systems with low core count, small RAM capacity and a shared L2 cache instead.

As listed in Table 2, the parameters on both Pi4 and R9-2920 systems are the same except N. The data types to be evaluated include Unsigned 32-bit integer (Uint32) and 64-bit double precision floating point numbers (Double). Only uniform distribution is evaluated. Each algorithm is compiled with -O2 flag to save experiment time. There are two sets of data size N where N ranges from 20M to 200M elements due to system RAM limit for Pi4 and 200M to 2000M for R9-2920.

As mentioned earlier, the block size B, Sorting Cutoff U_{sort} and Traversal Cutoff U_{trav} are **not** parameterized with L3 Cache size 8MB. The block size B is in order of KBs. Sorting Cutoff U_{sort} and Traversal Cutoff U_{trav} are in order of MBs as listed in Table 2. MSPSort can fork as many $\tau_{max} = m \times c$ threads where the Multiplier m= 1, 2. The minimum threads τ_{min} can be independent of hardware configuration, τ_{min} = 2, 3.

4.2 Key Performance Indicators (KPIs)

In this paper, some experiment results shall be normalized and compared based on these KPIs. They all represent time domain aspects of each sorting algorithm. Although lower $\overline{T}$ and σ_T are better in terms of run time and

stability, other statistics play important roles as well. We shall discuss the experiment results with respect to the following aspects.

4.2.1 Average run time ($\bar{T}$) and run time per zM ($\bar{T}_z$)

An omp_get_wtime() can be invoked twice just before and after running the algorithm to get accurate time stamps of each algorithm. The obtained time difference is the Run Time T. Note that data array randomization and initialization as well as verification after are excluded.

The Average Run Time ($\bar{T}$) can be normalized at z Million elements and called Run Time per zM ($\bar{T}_z$). Based on this, It is easy to compare at any data size for certain experiments. Furthermore, this normalized $\bar{T}_z$ can be system independent and compared among several systems running the same algorithm.

4.2.2 Run time standard deviation σ_T and σ_z

Run Time Standard of Deviation (σ_T) represents the stability of each algorithm due to the randomness of each workload. In addition, the normalized standard deviation (σ_z) per z Million elements can justify some parameters specially Block size B.

4.2.3 Run time stability

In addition to arithmetic mean $\bar{T}$ and standard deviation σ_T, the first, second and third quartiles are T_{Q1}, T_{Q2} and T_{Q3}, respectively. The InterQuartile Range can be determined as T_{IQR}=T_{Q3}-T_{Q1} for stability analyses. These statistics can specify how the Run Time T distributes over 1000 trials.

4.2.4 Run time difference percentage: $\Delta\bar{T}/\bar{T}_{msp}$

To emphasize differences between MSPSort $\bar{T}_{msp}$ and other algorithms $\bar{T}_{alg}$, this KPI is formulated as follows:

$$\Delta\bar{T}/\bar{T}_{msp}(\%) = \frac{(\bar{T}_{alg} - \bar{T}_{msp}) \times 100\%}{\bar{T}_{msp}}. \tag{6}$$

It is similar to Speedup equation. The positive values show how much MSP-Sort is faster in percentage. On the contrary, the negative values represent the opposite direction.

4.2.5 User-mode CPU utilization: V_{usr}

The amount of time that the CPU cores spending of a program can be measured in a virtual file /proc/stat. The CPU time consists of four major

components: user, system, nice and idle modes. We are interested in user mode CPU utilization because the higher user-mode utilization V, the more CPU is occupied by an algorithm. Other tasks shall be slowed down.

4.2.6 RAM utilization: V_{ram}

RAM can be limited due to cost constraints especially mobile platforms. The MaxResident is the maximum amount memory residing in RAM measured by /usr/bin/time Linux command in KiloBytes. Its man page is available online. Hence, the RAM Utilization percentage can be computed as follow.

$$V_{ram}(\%) = \frac{1024 \times MaxResident \times 100\%}{N \times sizeof(type)} \tag{7}$$

The more memory (RAM) consumption compared to the workload size, the higher V_{ram} percentage. It is of concern to current mobile platforms especially smartphones/tablets due to RAM limitation.

4.3 Results & Discussions

In order to optimize and fine tune parameters of MSPSort effectively, we start with the bottom most parameters all the way up to the top ones.

4.3.1 BF-MSPSort vs block size B

This subsection explains how we fine tune the block size B for Uint32 and Double. Earlier [7] the block size $B = 0.01 \times 8$MB which is about 80 KB. We start tuning B to be multiples of 32 KB (L1 data cache) on both systems. It is expected that B of larger data type as Double to be twice as big as that of Uint32 as show in Tables 3 and 4.

The selected values of B are 64 KB for Uint32 and 128 KB for Double on both systems. The focus is on N=200M and 2000M for Pi4 and R92920, respectively. It can be noticed that these values yield the optimal KPIs in terms of $\bar{T}_{10}$ and σ_{10} as well as. The same reason for R9-2920 is $\bar{T}_{100}$ and σ_{100} instead.

4.3.2 BF-MSPSort vs U_{sort}, m, τ_{min}

Given a fixed B from the previous experiments, the next parameters, U_{sort}, m and τ_{min} shall be investigated in BF-MSPSort. At the same time, we can learn how they interact to each other given the same data size N. The top (m and τ_{min}) tuples are listed in Table 5. It can be noticed that the whole run time is a trade off between partition time and sorting time.

Table 3 $\bar{T}_{10}$ and σ_{10} of BF-MSPSort, Uint32 and Double at various block sizes B on Pi4 system after 100 trials

Type	KPI (Sec.)	20M	50M	100M	200M
Uint32					
B=32KB	$\bar{T}_{10}$	**0.6396**	**0.6615**	**0.7056**	**0.7332**
	σ_{10}	0.0079	0.0058	0.0139	0.0068
B=64KB	$\bar{T}_{10}$	**0.6311**	**0.6627**	**0.6948**	**0.7359**
	σ_{10}	0.0112	0.0054	0.0071	0.0138
B=128KB	$\bar{T}_{10}$	**0.7270**	**0.7616**	**0.7870**	**0.8334**
	σ_{10}	0.0206	0.0139	0.0090	0.0180
Double					
B=64KB	$\bar{T}_{10}$	**1.0483**	**1.0978**	**1.1359**	**1.2004**
	σ_{10}	0.0250	0.0309	0.0193	0.0217
B=128KB	$\bar{T}_{10}$	**1.0714**	**1.0887**	**1.1576**	**1.1986**
	σ_{10}	0.0203	0.0207	0.0383	0.0245
B=256KB	$\bar{T}_{10}$	**1.0253**	**1.1047**	**1.1560**	**1.2007**
	σ_{10}	0.0115	0.0298	0.0379	0.0249

Table 4 $\bar{T}_{100}$ and σ_{100} of BF-MSPSort, Uint32 and Double at various block sizes B on R9-2920 system after 100 trials

Type	KPI (Sec.)	20M	50M	100M	200M
Uint32					
B=32KB	$\bar{T}_{100}$	**0.5762**	**0.6008**	**0.6244**	**0.6609**
	σ_{100}	0.05762	0.0172	0.0206	0.0285
B=64KB	$\bar{T}_{100}$	**0.5774**	**0.6018**	**0.6250**	**0.6533**
	σ_{100}	0.0120	0.0147	0.0210	0.0141
B=128KB	$\bar{T}_{100}$	**0.5753**	**0.5963**	**0.6247**	**0.6535**
	σ_{100}	0.0081	0.0141	0.0164	0.0156
Double					
B=64KB	$\bar{T}_{100}$	**0.8775**	**0.9503**	**0.9937**	**1.0484**
	σ_{100}	0.0208	0.249	0.0194	0.0317
B=128KB	$\bar{T}_{100}$	**0.8785**	**0.9473**	**0.9908**	**1.0325**
	σ_{100}	0.0154	0.224	0.0179	0.0382
B=256KB	$\bar{T}_{100}$	**0.8776**	**0.9481**	**0.9827**	**1.0397**
	σ_{100}	0.0133	0.205	0.0265	0.0411

4.3.3 BF-MSPSort vs BF-DF algorithms

Given the top parameter tuple from BF-MSPSort listed on Table 5, we shall study how each traversal algorithm differs from one another. Earlier, U_{trav} must be configured to multiples of 8MB-L3 cache. In this optimized version, U_{trav} could be adjusted to be any number or as a function of problem size especially for huge ones. It turns out that MSPSort with RAL BF-DF traversal yields the best $\bar{T}$ and σ_T for both systems and data types. Therefore, RAL is

Table 5 Top (B, U_{sort}, m, τ_{min}) tuples with BF traversal for all N's, after 20 Trials

System	Pi4	R9-2920
Uint32	(64kB, 4MB, 1, 2)	(64kB, 4MB, 2, 2)
	(64kB, 4MB, 1, 3)	(64kB, 4MB, 1, 3)
	(64kB, 4MB, 2,2)	(64kB, 4MB, 1, 2)
Double	(128kB, 8MB, 1, 2)	(128kB, 8MB, 1, 2)
	(128kB, 8MB, 2, 2)	(128kB, 8MB, 1, 3)
	(128kB, 4MB, 1, 2)	(128kB, 8MB, 2, 2)

Table 6 Selected top (B, U_{sort}, U_{trav}, m, τ_{min}) tuples for RAL-MSPSort for all N's, after 100 Trials

System	Uint32	Double
Pi4		
20M-200M	(64KB, 4MB, 4MB, 1, 2)	(128KB, 8MB, 8MB, 1, 2)
R9-2920		
200M	(64KB, 2MB, 8MB, 2, 2)	(128KB, 4MB, 8MB, 1, 2)
500M	(64KB, 4MB, 16MB, 2, 2)	(128KB, 8MB, 16MB, 1, 2)
1000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 32MB, 1, 2)
2000M	(64KB, 8MB, 32MB, 2, 2)	(128KB, 8MB, 64MB, 1, 2)

selected and thus called it RAL-MSPSort. Its final parameters are listed in Table 6.

RAL-MSPSort can yield the best Run Time statistics among all traversal algorithms, even though the BF execution order is similar to SPF algorithm. For both BF and SPF algorithms, shorter subarrays get partitioned done faster the resulting two subarrays can be enqueued sooner. This may not be good for last level cache on average.

With a single shared L2 cache, Pi4 system can be regarded as one-fourth of R9-2920 system. RAL-MSPSort still can yield better Run Time than BF traversal. especially when $U_{trav} = U_{sort}$ because MSPPartition() continues one more time resulting in two subarrays shorter than u_{sort}.

The cutoff may be different U_{trav} due to the data types and a wide variety of cache configurations especially mobile System on Chips. Some of them do not have L3 caches only L2 caches are available instead. In addition, some CPU cores are not identical such as ARM big.Little technology. Cache coherency may be an issue at the block as well as subarray boundaries. These boundaries can fall into the same cache block being accessed by two threads simultaneously.

Table 7 $\bar{T}$ and σ_T of MSPSort, RAL for Uint32 and Double at various sizes N and Partition Thread Ratio θ on Pi4 system after 100 trials

Alg.	KPI (Sec.)	20M	50M	100M	200M
Uint32					
$\theta = 1.00$	$\bar{T}$	**1.2640**	**3.2878**	**6.8597**	**14.3510**
	σ_T	0.0269	0.0357	0.0805	0.1368
$\theta = 0.75$	$\bar{T}$	**1.2945**	**3.3392**	**6.8744**	**14.6111**
	σ_T	0.0544	0.0578	0.0699	0.2899
Double					
$\theta = 1.00$	$\bar{T}$	**2.1488**	**5.4478**	**11.4136**	**24.2979**
	σ_T	0.0491	0.1079	0.2354	0.5938
$\theta = 0.75$	$\bar{T}$	**2.1637**	**5.5548**	**11.5595**	**24.6450**
	σ_T	0.0251	0.0666	0.1432	0.2352

4.3.4 RAL-MSPSort vs partition thread ratio θ

In this experiment, we shall explore how the number of threads running MSPPartition() affect the total Run Time T. The maximum threads stays the same at τ_{max} only the initial partition threads is reduced to $\theta \times \tau_{max}$ where $\theta \leq 1.00$. Based on parameters in Table 6, $\bar{T}$ and σ_T of RAL-MSPSort at different θ are listed in Tables 7 and 8 for Pi4 and R9-2920 systems, respectively.

We can observe that Pi4 loses some performance on both data types as we apply $\theta < 1.00$. For R9-2920, θ of less than 1.00 mitigates that pressure thus yields faster Run Time. On the other hand, parallelism of Pi4 is limited by CPU cores rather than memory bottleneck.

On the contrary, R9-2920 gains significant performance only for Double at θ=0.8 not Uint32. Two major reasons are behind this phenomenon. Firstly, this is because Double data type is twice as big as Uint32 per data element. That means either L3 cache or RAM is twice as busy with Double data requests given the same amount of time as Uint32 data. Secondly, it may be due to floating-point hardware resource constraints as well. Note that R9-2920 is multithreaded CPU and actually a 12-core 24-thread CPU.

4.3.5 RAL-MSPSort vs thread divider α

The earlier MSPSort [7], the Thread Divider α is always 2. The question may arise why τ in the next recursion is always divided by 2. Can other values be better than this? Thread Dividers α=5/3, 3/2 and 4/3 that are less than 2 can result in more threads in the next recursion level. On the other hand,

Table 8 $\bar{T}$ and σ_T of MSPSort, RAL for Uint32 and Double at various sizes N and Partition Thread Ratio θ on R9-2920 system after 100 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
Uint32					
$\theta = 1.00$	$\bar{T}$	**1.1598**	**2.9678**	**6.2311**	**12.5458**
	σ_T	0.0146	0.0245	0.1192	0.2564
$\theta = 0.90$	$\bar{T}$	**1.1459**	**2.9757**	**6.1624**	**12.7117**
	σ_T	0.0245	0.0847	0.2307	0.3633
$\theta = 0.80$	$\bar{T}$	**1.1396**	**2.9329**	**6.2100**	**12.9028**
	σ_T	0.0167	0.0388	0.1980	0.5081
$\theta = 0.75$	$\bar{T}$	**1.1429**	**2.9468**	**6.2087**	**12.8381**
	σ_T	0.0140	0.0524	0.2250	0.4650
$\theta = 0.67$	$\bar{T}$	**1.1437**	**2.9518**	**6.2619**	**12.9881**
	σ_T	0.0223	0.0566	0.3486	0.6049
Double					
$\theta = 1.00$	$\bar{T}$	**1.7408**	**4.6700**	**9.6917**	**20.1682**
	σ_T	0.0231	0.1736	0.2836	0.8512
$\theta = 0.90$	$\bar{T}$	**1.7363**	**4.6454**	**9.5878**	**20.3231**
	σ_T	0.0189	0.1460	0.2413	0.5916
$\theta = 0.80$	$\bar{T}$	**1.7118**	**4.6077**	**9.5261**	**20.0296**
	σ_T	0.0795	0.2034	0.4592	1.0444
$\theta = 0.75$	$\bar{T}$	**1.7230**	**4.6286**	**9.5329**	**20.1099**
	σ_T	0.0318	0.1658	0.4191	0.7131
$\theta = 0.67$	$\bar{T}$	**1.7197**	**4.6421**	**9.6873**	**20.4310**
	σ_T	0.0384	0.1327	0.3766	0.9811

higher α=5/2 can result in fewer threads. Based on the best parameters from the previous experiment, the effects of α on Pi4 and R9-2920 systems are provided in Tables 9 and 10, respectively.

It can be observed that α=2 yields the best results for Pi4 system and both types. On R9-2920 at Double N=1000M and 2000M, the saddle points at α=5/3 can be observed instead of α=2. Hugh double workload may get benefit from Dynamic Thread Allocation. The effects of Thread Equalizer β shall be presented in the last subsection. However, for simplicity α=2 for both data types and all systems.

4.3.6 RAL-MSPSort: final results vs. thread equalizer β

Once the parameters are finalized (β=OFF), the RAL-MSPSort is evaluated again at 1000 trials to get more stable results and benchmarked with BQSort and MWSort. Run time statistics of all sorting algorithm of Pi4 and R9-2920 are enumerated in Tables 11 and 12, respectively. On Pi4 system and $N \leq$100M, all $\bar{T}$ of both data types of all algorithms are similar. At workload

Table 9 $\bar{T}$ and σ_T of MSPSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 1.00$) at various sizes N and α, β on Pi4 system after 100 trials

Alg.	KPI (Sec.)	20M	50M	100M	200M
Uint32					
$\alpha = 4/3$	$\bar{T}$	**1.2931**	**3.3368**	**6.9511**	**14.5192**
β =OFF	σ_T	0.0265	0.0568	0.0909	0.2386
$\alpha = 3/2$	$\bar{T}$	**1.2931**	**3.3321**	**6.8597**	**14.4847**
β =OFF	σ_T	0.0297	0.0638	0.0976	0.2391
$\alpha = 2$	$\bar{T}$	**1.2640**	**3.2878**	**6.8597**	**14.3510**
β =OFF	σ_T	0.0269	0.0357	0.0805	0.1368
$\alpha = 5/2$	$\bar{T}$	**1.3303**	**3.4218**	**7.0489**	**14.7332**
β =OFF	σ_T	0.0723	0.0686	0.1065	0.1512
Double					
$\alpha = 5/3$	$\bar{T}$	**2.1535**	**5.5741**	**11.5349**	**24.4927**
β =OFF	σ_T	0.0464	0.1681	0.1554	0.6259
$\alpha = 2$	$\bar{T}$	**2.1488**	**5.4478**	**11.4136**	**24.2979**
β =OFF	σ_T	0.0491	0.1079	0.2354	0.5938
$\alpha = 5/2$	$\bar{T}$	**2.2146**	**5.6488**	**11.6705**	**24.6764**
β =OFF	σ_T	0.0479	0.1959	0.3204	0.5461

Table 10 $\bar{T}$ and σ_T of MSPSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 0.80$) at various sizes N and α, β on R9-2920 system after 100 trials

Alg.	KPI (Sec.)	200M	500M	1000M	2000M
Uint32					
$\alpha = 5/3$	$\bar{T}$	**1.1603**	**2.9786**	**6.2968**	**12.7600**
β =OFF	σ_T	0.0167	0.0388	0.1980	0.5081
$\alpha = 2$	$\bar{T}$	**1.1598**	**2.9678**	**6.2311**	**12.5458**
β =OFF	σ_T	0.0146	0.0245	0.1192	0.2564
$\alpha = 5/2$	$\bar{T}$	**1.1740**	**3.0219**	**6.4761**	**13.0133**
β =OFF	σ_T	0.0245	0.0869	0.3986	0.7027
Double					
$\alpha = 3/2$	$\bar{T}$	**1.7356**	**4.6483**	**9.6046**	**20.1066**
β =OFF	σ_T	0.0557	0.1152	0.2038	0.5079
$\alpha = 5/3$	$\bar{T}$	**1.7255**	**4.6356**	**9.4302**	**19.8885**
β =OFF	σ_T	0.0725	0.1871	0.5151	0.9203
$\alpha = 2$	$\bar{T}$	**1.7118**	**4.6077**	**9.5261**	**20.0296**
β =OFF	σ_T	0.0795	0.2034	0.4592	1.0444
$\alpha = 5/2$	$\bar{T}$	**1.7889**	**4.8657**	**10.2009**	**21.4310**
β =OFF	σ_T	0.0724	0.3059	0.6525	1.7900

Table 11 KPIs (Seconds) of of RAL-MSPSort vs BQSort vs MWSort for Uint32 ($\theta = 1.00$) and Double ($\theta = 1.00$) at various sizes N on Pi4 system after 1000 trials, β=OFF

Alg.	KPI (Sec.)	20M	50M	100M	200M	200M
Uint32						β=ON
MSPSort	T_{Q3}	1.2949	3.3129	6.9138	14.4948	14.6631
	$\bar{T}$	**1.2749**	**3.2883**	**6.8673**	**14.4120**	**14.5390**
	T_{Q2}	1.2732	3.2866	6.8607	14.3877	14.4783
	T_{Q1}	1.2539	3.2622	6.8136	14.3117	14.3709
	T_{IQR}	0.0410	0.0507	0.1002	0.1831	0.2922
	σ_T	0.0306	0.0374	0.0875	0.1626	0.2313
BQSort	T_{Q3}	1.2814	3.4008	7.1837	14.9355	
	$\bar{T}$	**1.2682**	**3.3726**	**7.1146**	**14.8235**	
	T_{Q2}	1.2640	3.3663	7.0907	14.8122	
	T_{Q1}	1.2482	3.3389	7.0259	14.6850	
	T_{IQR}	0.0332	0.0619	0.1578	0.2505	
	σ_T	0.0327	0.0480	0.1268	0.1986	
MWSort	T_{Q3}	1.2517	3.2947	6.9192	14.6352	
	$\bar{T}$	**1.2401**	**3.2739**	**6.8760**	**14.5602**	
	T_{Q2}	1.2354	3.2678	6.8708	14.5445	
	T_{Q1}	1.2243	3.2469	6.8253	14.4635	
	T_{IQR}	0.0274	0.0478	0.0939	0.1717	
	σ_T	0.0243	0.0400	0.0693	0.1453	
Double						β=ON
MSPSort	T_{Q3}	2.1275	5.5136	11.6239	24.6005	24.6358
	$\bar{T}$	**2.0899**	**5.4371**	**11.4603**	**24.2545**	**24.3857**
	T_{Q2}	2.0806	5.4507	11.4952	24.2736	24.3834
	T_{Q1}	2.0452	5.3733	11.2931	23.8399	24.1348
	T_{IQR}	0.0823	0.1403	0.3308	0.7606	0.5010
	σ_T	0.0610	0.1154	0.2571	0.5938	0.4926
BQSort	T_{Q3}	2.0836	5.5412	11.7007	24.8755	
	$\bar{T}$	**2.0442**	**5.4775**	**11.5933**	**24.6479**	
	T_{Q2}	2.0424	5.4627	11.5666	24.6200	
	T_{Q1}	1.9991	5.4014	11.4318	24.3610	
	T_{IQR}	0.0845	0.1398	0.2689	0.5145	
	σ_T	0.0540	0.1134	0.2541	0.4409	
MWSort	T_{Q3}	1.9794	5.3865	11.6665	24.9317	
	$\bar{T}$	**1.9667**	**5.3554**	**11.5684**	**24.8053**	
	T_{Q2}	1.9639	5.3343	11.5659	24.6787	
	T_{Q1}	1.9504	5.2840	11.4709	24.4965	
	T_{IQR}	0.0290	0.1025	0.1956	0.4352	
	σ_T	0.0249	0.0826	0.1848	0.5598	

Table 12 KPIs (Seconds) of of RAL-MSPSort vs BQSort vs MWSort for Uint32 and Double at various sizes N on R9-2920 system after 1000 trials, β=OFF

Alg.	KPI (Sec.)	200M	500M	1000M	2000M	2000M
Uint32						β=ON
MSPSort	T_{Q3}	1.1615	3.0147	6.2208	12.8731	12.7956
	$\bar{T}$	**1.1532**	**2.9871**	**6.1665**	**12.8096**	**12.6881**
	T_{Q2}	1.1511	2.9720	6.1285	12.7004	12.6646
	T_{Q1}	1.1416	2.9378	6.0565	12.5842	12.5497
	T_{IQR}	0.0199	0.0765	0.1743	0.2889	0.2459
	σ_T	0.0194	0.0245	0.1192	0.2564	0.1976
BQSort	T_{Q3}	1.2849	3.3515	6.9910	14.8314	
	$\bar{T}$	**1.2641**	**3.3027**	**6.8910**	**14.4880**	
	T_{Q2}	1.2608	3.2872	6.8587	14.3754	
	T_{Q1}	1.2366	3.2211	6.7273	13.9707	
	T_{IQR}	0.0483	0.1304	0.2637	0.8607	
	σ_T	0.0398	0.1266	0.2883	0.7737	
MWSort	T_{Q3}	1.2690	3.1965	6.9421	14.6856	
	$\bar{T}$	**1.2605**	**3.1748**	**6.7743**	**14.4544**	
	T_{Q2}	1.2482	3.1424	6.8500	14.5239	
	T_{Q1}	1.2368	3.1253	6.4225	14.3433	
	T_{IQR}	0.0322	0.0712	0.5196	0.3423	
	σ_T	0.0351	0.0748	0.2940	0.3672	
Double						β=ON
MSPSort	T_{Q3}	1.7501	4.6861	9.7268	20.5472	20.2142
	$\bar{T}$	**1.7335**	**4.6198**	**9.6218**	**20.3233**	**20.0028**
	T_{Q2}	1.7302	4.6215	9.5645	20.2944	19.9662
	T_{Q1}	1.7137	4.5675	9.4322	20.0442	19.7528
	T_{IQR}	0.0364	0.1186	0.2946	0.5030	0.4614
	σ_T	0.0328	0.1771	0.3427	0.6693	0.4716
BQSort	T_{Q3}	1.8262	4.8709	10.2529	21.6878	
	$\bar{T}$	**1.7977**	**4.7990**	**10.0812**	**21.3060**	
	T_{Q2}	1.7821	4.7602	10.0035	21.0510	
	T_{Q1}	1.7562	4.6768	9.8058	20.6554	
	T_{IQR}	0.0700	0.1941	0.4471	1.0324	
	σ_T	0.0593	0.1729	0.4180	1.0467	
MWSort	T_{Q3}	1.5818	4.0025	8.9359	18.2426	
	$\bar{T}$	**1.5700**	**4.0276**	**8.7317**	**18.0441**	
	T_{Q2}	1.5659	3.9595	8.8775	18.0956	
	T_{Q1}	1.5542	3.9384	8.7910	17.9187	
	T_{IQR}	0.0276	0.0641	0.1449	0.3239	
	σ_T	0.0244	0.1598	0.3418	0.4057	

N=200M, RAL-MSPSort's T_{Q3} to T_{Q1} are the best in all categories. On R9-2920, RAL-MSPSort performs extremely fast on Uint32 data. However, MWSort is still the fastest of Double data type.

For run time stability analyses of both systems, σ_T and T_{IQR} can be of interests. The σ_T and T_{IQR} of RAL-MSPSort are mostly lower than those of BQSort and MWSort for both data types. It can be concluded that RAL-MSPSort is consistently stable on both low-end and high-end mobile systems.

The Thread Equalizer β is only switched ON for N=200M and N=2000M on Pi4 and R9-2920, respectively. The experiment results are augmented to the final ones at the right most column. Left and right Thread Equalizers β_L and β_R correspond to Equations (4) and (5), respectively. Due to the off balanced pivot selection, the much longer subarray is to be partitioned further with more threads. It can be noticed at N=200M that Pi4 system loses both $\bar{T}$ and σ_T. This can be due to small number of c=4 cores. However, R9-2920 system has shown that β=ON can handle both huge Uint32 and Double workloads, i.e. N=2000M well. Their Run Time statistics are much better than that of β=OFF.

4.4 Other Percentage KPIs

Other KPIs in percentage are summarized of Pi4 and R9-2920 systems in Tables 13 and 14, respectively. The Percentages of Run Time Difference of MSPSort become positive values at larger workloads on both algorithms and both systems. The overhead of MSPPartition is well justified with Uint32 data type at larger workloads. The only exception is at MWSort on R9-2920 system that can localize data to the CPU cores.

From average user mode CPU utilization V_{usr} point of view, RAL-MSPSort incurs lower values than BQSort but a bit higher than MWSort. That means operations of stacks to interleave C&S data within blocks can be effective yet similar to the original Hoare's. Unlike BQSort that can steal workloads from other deques, RAL-MSPSort relies on thread scheduling of OpenMP to balance the CPU Utilization. Therefore, Some CPU cores can be available to other urgent tasks instead of synchronizing.

The extra RAM Utilization percentage over 100% is considered overhead. The bigger workload, the smaller overhead is for all algorithms. This overhead stems from the data structures to handle parallel scheduling operations. Given the same problem size N, Double data type incurs less amount of overhead percentage. RAL-MSPSort and BQSort need similar amount of RAM for both Uint32 and Double. However, MWSort requires double the

Table 13 KPIs (%) of of RAL-MSPSort vs BQSort vs MWSort for Uint32 and Double at various sizes N on Pi4 system after 1000 trials, β=OFF

Alg.	KPI (%)	20M	50M	100M	200M
Uint32					
MSPSort	$\bar{V}_{usr}$	92.19	95.89	97.17	97.77
	$\bar{V}_{ram}$	103.79	101.67	100.85	100.54
BQSort	$\Delta\bar{T}/\bar{T}_{msp}$	-0.53	2.56	3.60	2.86
	$\bar{V}_{usr}$	98.28	98.64	98.81	98.91
	$\bar{V}_{ram}$	103.99	101.60	100.80	100.40
MWSort	$\Delta\bar{T}/\bar{T}_{msp}$	-2.73	-0.44	0.13	1.03
	$\bar{V}_{usr}$	93.33	93.89	94.26	94.52
	$\bar{V}_{ram}$	203.83	201.54	200.78	200.40
Double					
MSPSort	$\bar{V}_{usr}$	93.38	96.28	97.22	97.67
	$\bar{V}_{ram}$	100.83	100.79	100.43	100.26
BQSort	$\Delta\bar{T}/\bar{T}_{msp}$	-2.19	0.74	1.16	1.62
	$\bar{V}_{usr}$	97.93	98.25	98.41	98.53
	$\bar{V}_{ram}$	101.97	100.79	100.40	100.20
MWSort	$\Delta\bar{T}/\bar{T}_{msp}$	-5.90	-1.50	0.94	2.27
	$\bar{V}_{usr}$	90.95	91.78	92.32	92.75
	$\bar{V}_{ram}$	201.92	200.78	200.39	200.19

Table 14 KPIs (%) of of RAL-MSPSort vs BQSort vs MWSort for Uint32 and Double at various sizes N on R9-2920 system after 1000 trials, β=OFF

Alg.	KPI (%)	200M	500M	1000M	2000M
Uint32					
MSPSort	$\bar{V}_{usr}$	90.73	93.34	94.02	94.04
	$\bar{V}_{ram}$	100.56	100.38	100.26	100.21
BQSort	$\Delta\bar{T}/\bar{T}_{msp}$	9.62	10.57	11.75	13.10
	$\bar{V}_{usr}$	94.30	94.80	94.38	91.03
	$\bar{V}_{ram}$	100.57	100.23	100.12	100.06
MWSort	$\Delta\bar{T}/\bar{T}_{msp}$	9.30	6.28	9.86	12.84
	$\bar{V}_{usr}$	89.39	91.31	88.72	85.93
	$\bar{V}_{ram}$	200.55	200.22	200.11	200.05
Double					
MSPSort	$\bar{V}_{usr}$	87.32	87.31	86.92	87.95
	$\bar{V}_{ram}$	100.31	100.25	100.20	100.18
BQSort	$\Delta\bar{T}/\bar{T}_{msp}$	3.70	3.88	4.77	4.84
	$\bar{V}_{usr}$	92.06	89.51	86.67	87.42
	$\bar{V}_{ram}$	100.29	100.12	100.06	100.03
MWSort	$\Delta\bar{T}/\bar{T}_{msp}$	-9.43	-12.82	-9.25	-11.21
	$\bar{V}_{usr}$	86.31	86.81	82.99	82.75
	$\bar{V}_{ram}$	200.24	200.11	200.05	200.03

RAM spaces as the /usr/bin/time command reports. It can be due to fact that the MultiWay Parallel Merge function [9] needs an extra buffer with space complexity $O(N)$ as shown $\bar{V}_{ram}$ >200% in Tables 13 and 14. It is crucial on mobile platforms that are RAM limited.

5 Conclusions & Future work

Our fine-tuned MSPSort has been proved that its performance is comparable to the Standard Template Library parallel sorting algorithms, i.e. BQSort and MWSort. The comparison is carried out on two mobile-equivalent Linux systems, i.e. quad core ARM Cortex A72 on Raspberry Pi4 and 24-core AMD ThreadRipper representing low-end and high-end systems. The key performance indicators include run time statistics, user-mode CPU utilization and memory (RAM) utilization.

For further improvements, the current stack implementation shall be investigated and replaced with other structures such as deque. Our MSPPartition shall be applied to solve other fundamental problems such as parallel median, minimum and maximum value selections. In addition, a combination of Shortest Partition First and RAL traversal algorithms shall be explored to gain even more cache locality.

References

[1] Charles Antony and Richard Hoare. Quicksort. *ACM*, 4:321, 1962.

[2] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *J. Exp. Algorithmics*, 14:4:1.4–4:1.24, January 2010.

[3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[4] Leonor Frias and Jordi Petit. *Parallel Partition Revisited*, volume 5038 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.

[5] Philip Heidelberger, Alan Norton, and John T. Robinson. Parallel quicksort using fetch-and-add. *IEEE Transactions on Computers*, 39(1):847–857, January 1990.

[6] Emanuele Manca, Andrea Manconi, Alessandro Orro, Giuliano Armano, and Luciano Milanesi. Cuda-quicksort: an improved gpu-based implementation of quicksort. *Concurrency and computation: practice and experience*, 28(1):21–43, 2016.

[7] Apisit Rattanatranurak and Surin Kittitornkun. A multistack parallel (msp) partition algorithm applied to sorting. *Journal of Mobile Multimedia*, pages 293–316, 2020.

[8] Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan Primitives for GPU Computing. In Mark Segal and Timo Aila, editors, *SIGGRAPH/Eurographics Workshop on Graphics Hardware*. The Eurographics Association, 2007.

[9] Johannes Singler, Peter Sanders, and Felix Putze. Mcstl : The multi-core standard template library. *Euro-Par 2007 Parallel Processing. Springer Berlin Heidelberg*, pages 682–694, 2007.

[10] Philippas Tsigas and Yi 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)*, pages 372–381, Genoa, Italy, February 5th-7th 2003.

Biographies

Apisit Rattanatranurak received his M.Eng. and B.Eng. degrees in Computer Engineering from King Mongkut's Insitute of Technology Ladkrabang (KMITL), Bangkok, Thailand. Now, he is pursuing a doctoral degree at the School of Engineering, KMITL. His research interest is in the area of parallel programming, computing on multi-core CPU and GPU on Linux/Unix system.

Surin Kittitornkun received his Ph.D. and M.S. degrees in Computer Engineering from University of Wisconsin-Madison, USA. Currently, he is an Assistant Professor at School of Engineering, King Mongkut's Insitute of Technology Ladkrabang (KMITL), Bangkok, Thailand. His research interests include parallel algorithms, mobile/high performance computing and computer architecture.