J Supercomput (2016) 72:1063–1091
DOI 10.1007/s11227-016-1641-y

# Parallel Partition and Merge QuickSort (PPMQSort) on Multicore CPUs

**Ratthaslip Ranokphanuwat**[1] · **Surin Kittitornkun**[1]

Published online: 18 February 2016


**Abstract** An explosive amount of data has tremendous impacts on sorting, searching, indexing, and so on. Sorting is one of the basic Computer Science problems needed to be fast and efficient to serve Big Data. This paper presents an efficient and scalable algorithm called *Parallel Partition and Merge QuickSort (PPMQSort)* running on any shared memory/multicore/multi-socket systems. Together with OpenMP 3.0 library, the PPMQSort is developed to be compatible and benchmarked with the fastest C/C++ Stdlib $qsort()$. The PPMQSort recursively divides an unsorted input array into partially sorted partitions up to *Cutoff* length using nested multithreading. Finally, those independent partitions are $qsort()$ (conquered) such that no synchronizations are needed. The resulting Speedup of 12.29× on a dual-socket 8-core Xeon E5520 can be achieved for sorting random 200 M 32-bit integer data at 16 threads. With the same configuration, a 4-core AMD A6-3600 CPU (non-HyperThread) can reach up to 4.67×, a superlinear Speedup. It has been proved that the proposed PPMQSort can exploit all available cache levels and HyperThread CPU cores well thus utilizing up to 83 % and 96 % of CPU on E5520 and A6-3600, respectively.

**Keywords** QuickSort · Parallel · OpenMP · Multicore · Multithread · Superlinear

✉ Ratthaslip Ranokphanuwat
udom.ran@dpu.ac.th

Surin Kittitornkun
surin.ki@kmitl.ac.th

[1] Faculty of Engineering, King Mongkut's Institute of Technology Ladkrabang, No. 1, Soi Chalong Krung 1, Chalong Krung Rd., Ladkrabang, Bangkok 10520, Thailand

## 1 Introduction

Sorting has become highly important for Big Data analyses especially in social/web mining, large scale scientific, commercial application domains and so on. Among all the sorting algorithms, QuickSort [1,2] is the most well-known and standard sorting algorithm. To compare with any existing sorting algorithms, QuickSort is the fastest one in practice [3].

Numerous parallel architectures can be applied to perform sorting algorithms. Earlier studies have shown [4–6] that sorting can be done at the interconnection level of a particular network of processors named the MultiRing network. Recently, sorting networks have been implemented on FPGAs instead. References [7–9] used FPGAs as sorting kernels for database intensive operations. In addition to FPGAs, hundreds to thousands of processing elements/cores inside the GPUs can be applied as co-processors for sorting [10] based on SIMD parallelism including the Bitonic-Merge Sort on Intel Xeon Phi [11].

A few parallel algorithms have been proposed to enhance the existing QuickSort algorithm. Initially, Heidelberger [12] presented the parallel version on an *ideal* Parallel Random Access Machine. In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. In 2003, Tsigas and Zhang [13] proposed a fine-grain parallel QuickSort algorithm to fit data into L1 caches. A year later, [14] presented several alternative algorithms of parallel QuickSort based on pthreads and OpenMP 2.0. Man et al. [15,16] developed $psort()$ algorithm to be compatible with Stdlib $qsort()$. Their work can achieve Speedup by only 11 times faster with 24 cores. Meanwhile, Kim et al. [17] have shown that a dual-core OMAP-4430 can achieve only 1.47x Speedup from their Introspective QuickSort algorithm. Mahafzah [18] splitted the input array with multi-pivot/thread into partitions using extra space and then sorted them in parallel up to 8 threads. Very recently, Bingmann et al. [19] proposed multikey QuickSort algorithms for string sorting on NUMA (Non Uniform Memory Access) architectures. Their results show that the Speedup is bounded by memory bandwidth.

However, it is still challenging to enhance parallel QuickSort performance and efficiency at the same time. These challenges are due to sequential data partitioning, latency/bandwidth between memory hierarchy, and sequential and recursive nature of QuickSort. Furthermore, the bottlenecks of parallel QuickSort should be further investigated together with some performance characteristics such as CPU utilization, memory bandwidth and branch misprediction rate.

In this paper, we have proposed and developed a Parallel Partition and Merge QuickSort (PPMQSort) for various multicore CPUs. Our contributions are summarized as follows:

1. The PPMQSort algorithm is compatible and benchmarked with Stdlib $qsort()$ while achieving superlinear Speedups in some CPUs.
2. The efficiency called Speedup per Core of PPMQSort and any parallel algorithm on both HyperThread and non-HyperThread CPUs is proposed. Hence, the PPMQSort can achieve higher efficiency than previous algorithms.

3. The time complexity of PPMQSort has been analyzed and presented in big-O notations.
4. Based on the Linux Perf measurement tool, a system performance model of any shared memory/multiprocessor/multicore systems is proposed to estimate memory bandwidth.
5. The Speedups of PPMQSort with Worst-case input data although very rare but can be as high as those of Random cases.

The rest of the paper is organized as follows. Section 2 presents background and related work. Section 3 presents our algorithm and discusses the implementation details. Section 4 describes performance evaluation and discussions. Finally, Sect. 5 concludes and suggests future work.

## 2 Background and related work

We begin with a brief overview of QuickSort, Stdlib $qsort()$, a number of parallel QuickSort algorithms, and finally OpenMP library.

### 2.1 QuickSort algorithm [1, 2]

QuickSort is the most famous and widely used sorting algorithm. The divide and conquer concept recursively partitions and swaps an input array into two halves: less than or equal (LEQ) half and greater than (GT) half with respect to a selected pivot element at each recursion level. The time complexity on average is, therefore, $O(n \log n)$ although the poorly selected pivot can affect its complexity. Even worse, the worst-case input array can make the complexity become $O(n^2)$. In terms of space requirements, QuickSort is considered to be an in-place algorithm using minimal extra memory. During the recursion, extra space for calling stack is proportional to $O(\log n)$. To optimize its performance, selecting good pivot(s) from several candidates has been considered.

### 2.2 Stdlib $qsort()$

The Standard Library $qsort()$ is a very useful function for sorting an array of any data types with a user-defined comparison function. It is implemented in C/C++ and also provided as a built-in function for several C/C++ compilers. Its function prototype is declared in *Stdlib.h* as follows.

```
void qsort(void  *base, size_t num_elements,
   size_t element_size,
   int (*compare)(void const *, void  const *));
```

The argument *base* is a pointer to the unsorted array, *num_elements* indicates the number of elements, *element_size* is the size of each element, and *compare* is a pointer to the user-defined function that returns integer values according to the comparison result.

### 2.3 Parallel QuickSort algorithms

In 1990, Heidelberger et al. [12] presented a parallelization of the Quicksort on a theoretical/ideal Parallel Random Access Machine with average of $O(\log n)$ time complexity. In practice, the sequential QuickSort can be enhanced with several parallel techniques to run on any shared memory/multicore systems with multithreading operating system. Parallel versions of QuickSort normally start with partitioning data into several chunks to fit any cache level depending on the size. These chunks can be partially or fully sorted and then merged to form bigger chunks. These two steps may be recursive as indicated in the Recursion row of Table 1. Some algorithms may use extra space to hold the intermediate results as shown in Ex. Space row. Eventually, they shall be fully sorted again with either the Stdlib $qsort()$ or others. The comparison of previous parallel QuickSort algorithms is shown in Table 1 in chronological order from left to right.

Tsigas and Zhang [13] proposed a fine-grain (block-based) parallel Quicksort algorithm. Subsequently, [14] presented several alternative algorithms of parallel Quicksort based on pthreads and OpenMP 2.0. Rashid et al. [20] enhanced Tsigas and Zhang's [13] PQuicksort on x86 Multithreaded Architectures. Man et al. [15,16] developed $psort()$ algorithm to be compatible with Stdlib $qsort()$. The input array is divided into groups and $qsort()$ them. Later on, these partitions can be merged using extra space and finally $qsort()$ them again. Their work can achieve Speedup by 11 times faster with up to 24 cores. Kim et al. [17] have shown that an embedded dual-core OMAP-4430 can achieve 1.47x Speedup from their Introspective Quicksort algorithm. Mahafzah [18] splitted the input array with multi-pivot/thread into partitions using extra space and then sort them in parallel up to 8 threads. Recently, Saleem et al. [21] estimated Speedup for QuickSort and Merge sort algorithms using Intel Cilk Plus.

### 2.4 OpenMP library

OpenMP library [22] is the most well-known library that can be applied successfully to develop parallel programs running on multicore CPUs architecture. It provides an application program interface (API) for thread-based parallelism on shared memory multicore processors. The API consists of a set of compiler directives, library routines, and environmental variables that support FORTRAN and C/C++ on multiple architectures. OpenMP uses the fork-join model for multithreading execution model. The main advantage of using OpenMP is the ability of all CPU cores to share and access the same memory pool (data) with less communication overhead and network latency compared with other parallel computing paradigms such as cluster computing, grid computing, etc.

Since OpenMP version 3.0, Task construct has been introduced to handle irregular and dynamic parallelism in the form of recursive routines. Tasks are units of work which can be executed (forked) in parallel as threads. This paper specifically demonstrates how to exploit the Task construct in our parallel QuickSort algorithm.

**Table 1** Comparison of previous parallel QuickSort algorithms, *Par.* Parallel, *Seq.* Sequential, *Sync.* Synchronization, *NA* Not Available

| Year<br>References | 2003<br>[13] | 2004<br>[14] | 2011<br>[16] | 2011<br>[17] | 2013<br>[18] | 2014<br>[21] |
|---|---|---|---|---|---|---|
| Algo. name | PQuicksort | cv_1.0 | psort1 | Introspective | QuickSort | Quicksort |
| Partition | Par. in blocks of L1 size | Seq. | Par. $n/c$ and $qsort()$ | Seq. $n/c$ | Par. multiple pivots | Seq. $n$ |
| Merge | Seq. Swap | No | Seq. Merge and and $qsort()$ | No | No | No |
| Recursion | Yes | Yes | No | No | No | Yes |
| Time complexity | $O(\frac{n}{c} + \frac{n}{c} \log \frac{n}{c})$ | NA | $O(n + \frac{n}{c} \log \frac{n}{c})$ | $O(\frac{n}{2} + \frac{n}{2} \log \frac{n}{2})$ | $O(\frac{n}{h} \log \frac{n}{h})$ | NA |
| Extra space (size) | No | No | Yes($n$) | No | Yes($n$) | No |
| Using $qsort()$ | Similar | No | Yes | No | No | No |
| Other sort | Insertion | No | No | Insertion | No | No |
| Library | NA | pthreads | OpenMP 3.0 | OpenMP 3.0 | pthreads | Cilk Plus |
| Pros | Cache efficient, Fine-grained, | Load balance- With busy waiting | Qsort() lib. compatible, Good load balance | Limit deep partition, Cache friendly, Good load balance | Utilize SMT architecture, Good load balance | Easily |
| Cons | Bottleneck- In seq. merge, Special Sync.- Instruction | Sync. added, Less algorithm details | Difficult to implement, High overhead | No nested parallelism, Seq. partition | Sync. added, Extra space | Unpopular lib. |

## 3 PPMQSort algorithm

Previous parallel QuickSort algorithms focus on optimizing either partitioning phase or recursive QuickSort phase. Our PPMQSort pays attention on both phases. In this section, we propose the Parallel Partition and Merge QuickSort (PPMQSort) on any multicore CPUs. The concept of PPMQSort is to partition an unsorted array into partially sorted partitions in the Parallel Partition Step. Then, these partitions can be eventually sorted independently using OpenMP Task construct in the Parallel *qsort*() Step. Those important steps mentioned above can be illustrated in Fig. 1.


**Fig. 1** Illustration of Parallel Partition and Merge QuickSort (PPMQSort) consisting of Parallel Partition Step and Parallel *qsort*() Step

**Table 2** Notations

| | |
|---|---|
| $a$ | Input data array |
| $a_j$ | Data element at index $j$ |
| $a_{i_d}$ | Data element at the middle index $i_d$ |
| $a_p$ | Data element at pivot $p$ |
| $B$ | Branch Loads |
| $B_m$ | Branch Load Misses |
| $B_{m/s}$ | Branch Load Misses Per Second |
| $C$ | Cache References |
| $C_m$ | Cache Misses |
| $C_{m/s}$ | Cache Misses Per Second |
| $\|C_{\text{line}}\|$ | Cache Line Size |
| $c$ | Number of processor cores |
| $d$ | The middle index |
| $f()$ | A function |
| HT/NHT | HyperThread/Non-HyperThread |
| $i, j$ | Loop indices |
| $i_L, i_R$ | Left most and right most indices of the left-hand side subarray, respectively |
| $j_L, j_R$ | Left most and right most indices of the right-hand side subarray, respectively |
| $i_d, j_d$ | The middle indices of left subarray and right subarray, respectively |
| $k$ | Number of CPU sockets |
| K | $10^3$ or $2^{10}$ |
| $l$ | Recursion level $l$ |
| M | $10^6$ or $2^{20}$ |
| $M_{bw}$ | Total memory bandwidth |
| $n$ | Number of elements of array $a$ |
| $O()$ | BigO notation |
| $o$ | Optimization Level |
| $p$ | The pivot place |
| $R_{x,y}$ | Correlation Coefficient of $x$ and $y$ |
| $S$ | Speedup |
| $s$ | Second |
| $T_{\text{qsort}}$ | Run Time of sequential Stdlib $qsort()$ |
| $T_{\text{ppmqsort}}$ | Run Time of PPMQSort |
| $T_{\text{seq}}$ | Run Time of any sequential QuickSort |
| $T_{\text{par}}$ | Run Time of any parallel QuickSort |
| $U$ | %CPU Utilization |
| $u$ | Cutoff size |

The PPMQSort is actually developed in C language on top of an open-source/past version of Stdlib $qsort()$ utilizing stack rather than recursion. Due to limited space and ease of understanding, the algorithms are explained in recursion. Notations used in this paper are listed in Table 2.

We first present the *Partition Phase with 2 Threads and Merge Phase with 1 or 2 Threads*. Then we show how to apply OpenMP Task parallelism to call $qsort()$. At last, the time complexity of our algorithm is analyzed.

### 3.1 Parallel Partition Step

The partitioning operation has been a major bottleneck of QuickSort since it was invented. Previous work has tried to optimize it by both reducing the number of key comparisons and fast swapping code. The key idea of the Partition Phase with 2 Threads is to divide the input data array into two subarrays. Then, they can be partitioned in parallel with 2 threads into 4 sub-subarrays using the same pivot value. Next is the Merge Phase with 1 or 2 Threads swapping the second and third sub-subarrays. Both phases of Parallel Partition Step are explained in details as follows.

#### *3.1.1 Partition Phase with 2 Threads*

Initially, an unsorted data array, $a = a_0, a_1, \ldots, a_{n-1}$, is divided into two independent subarrays at the pivot $p$. Let $a_p$ denotes the pivot element selected by $MedianOfThree()$ function. Let $i_L$ and $i_R$ be left indices and $j_L$ and $j_R$ be right indices of $a$, respectively. The left subarray of $a$, $a_0, \ldots, a_{p-1}$ corresponds to $(i_L = 0, i_R = p - 1)$. Similarly, the right subarray, $a_{p+1}, \ldots, a_{n-1}$, corresponds to $(j_L = p + 1, j_R = n - 1)$. In this phase, both subarrays, $(a_{i_L}, a_{i_R})$ and $(a_{j_L}, a_{j_R})$, are compared and swapped with the same pivot $a_p$ simultaneously using 2 threads on line 13 and line 15 in $seq\_partition()$ of Algorithm 1. In addition, $seq\_partition()$ returns the partition index as $i_d$ and $j_d$ for the left and right partitions, respectively, as shown. As a result, $a_0, \ldots, a_{n-1}$ are splitted into 4 sub-subarrays; two sub-subarrays on the left, $a_0, \ldots, a_{i_d}$ and $a_{i_d+1}, \ldots, a_{p-1}$, and two sub-subarrays on the right, $a_{p+1}, \ldots, a_{j_d-1}$ and $a_{j_d}, \ldots, a_{n-1}$. Notice that $i_d$ and $j_d$ are the middle indices of the left and right subarrays, respectively.

From a programming perspective, we have applied OpenMP Parallel Tasks without barrier synchronization, leading to improved CPU utilization. To reduce the number of shared memory accesses, $d = p$ is copied to be a local private variable to improve cache locality. Both Phases are listed in Algorithm 1.

In summary, based on $i_d$, $p$, and $j_d$, two independent subarrays can be partitioned into 4 sub-subarrays in parallel with respect to the global $a_p$ pivot in this phase. These 4 sub-subarrays are ordered as follows: less than or equal (LEQ), greater than (GT), LEQ, and GT from left to right.

#### *3.1.2 Merge Phase with 1 or 2 Threads*

In this subsection, we will explain how the second (GT) sub-subarray, $a_{i_d+1}, \ldots, a_{p-1}$, and the third (LEQ) sub-subarray, $a_{p+1}, \ldots, a_{j_d-1}$, are swapped and merged together. The idea of this phase is to swap all data in both sub-subarrays to rearrange them in the correct order, LEQ and GT. Because this phase needs only to swap a bulk of data

**Algorithm 1** The Parallel Partition algorithm

```
1: function ParallelPartition(a,start,end)                    ▷ Parallel Partition Step
2:     i_L, j_R, p ← Partition(a, start, end)                 ▷ call Partition function
3:     p ← Merge(a, p, i_L, j_R)                              ▷ call Merge function
4:     return p
5: end function

6: function Partition(a, i_L, j_R)                            ▷ Partition Phase with 2 Threads
7:     p ← MedianOfThree(a, i_L, j_R)
8:     d ← p
9:     i_R ← p − 1
10:    j_L ← p + 1
11:    begin OpenMP parallel Tasks private(d)
12:    OpenMP Task
13:    i_d ← seq_partition(a, d, i_L, i_R)                    ▷ Partition the Left Subarray
14:    OpenMP Task
15:    j_d ← seq_partition(a, d, j_L, j_R)                    ▷ Partition the Right Subarray
16:    end parallel Tasks
17:    i_L ← i_d + 1
18:    j_R ← j_d − 1
19:    return (i_L, j_R, p)
20: end function

21: function Merge(a, p, i_L, j_R)                            ▷ Merge Phase with 1 or 2 Threads
22:                          ▷ Three cases for calculating location and moving the pivot p
23:    if len(i_L, p − 1) < len(p + 1, j_R) then              ▷ Left side is shorter.
24:        length ← len(i_L, p − 1)
25:        Swap(a_p, a_{j_R − length})
26:        p ← j_R − length
27:        temp ← p + 1
28:    else if len(i_L, p − 1) > len(p + 1, j_R) then         ▷ Right side is shorter.
29:        temp ← p + 1
30:        length ← len(p + 1, j_R)
31:        Swap(a_p, a_{i_L + length})
32:        p ← i_L + length
33:    else                                                   ▷ Left side equals right side.
34:        temp ← p + 1
35:        length ← len(p + 1, j_R)
36:    end if
37:    begin OpenMP parallel For with 1 or 2 Threads
38:    for i ← 0, length − 1 do                               ▷ Swapping with 1 Thread or 2 Threads
39:        Swap(a_{i_L + i}, a_{temp + i})
40:    end for
41:    end parallel For
42:    return p
43: end function
```

between them, no comparisons are necessary. Furthermore, swapping would work with data on the same sub-subarrays so that our method does not use an extra memory.

The Merge Phase with 1 or 2 Threads is shown as function $Merge()$ on line 21 of Algorithm 1 where $len()$ returns the number of elements between two arguments ($len(x, y) = y - x + 1; y \geq x$). Let $i_L = i_d + 1$ and $j_R = j_d - 1$ be the left most index and the right most index of the sub-subarray. So, the second (GT) sub-subarray consists of $a_{i_L}, \ldots, a_{p-1}$ and the third (LEQ) sub-subarray consists of $a_{p+1}, \ldots, a_{j_R}$.

Both arrays must be swapped to complete the Parallel Partition Step. The swapping will start from this pair $(a_{i_L+i}, a_{\text{temp}+i})$ and incrementally continue for $i = 0$ to length $-1$ on line 39 of Algorithm 1. After swapping is finished, $a_p$ must be adjusted to the correct position. Both phases of Parallel Partiton Step are recursive as shown in function $\_QSort()$ until each partition's size is no greater than Cutoff $u$ on line 1 of Algorithm 2. Although it is associated with OpenMP Single construct on line 15, parallelism can be achieved up to $2h$ threads in reality. That's because the Parallel Partition Step each forks 2 threads internally.

Three important steps need to be considered in this phase. Firstly, the total number of elements swapped between two sub-subarrays is calculated. This number can be determined from the shorter length of either $i_d$ and pivot place $p$ or $j_d$ and $p$. The variable length can be $\leq \frac{n}{4}$. Then, the direction of swapping sequence is determined. To be cache friendly, increasing order is chosen. The last step is to move the pivot to the appropriate position in the array after swapping process is finished to guarantee that the Parallel Partition Step is completed. In the next Parallel $qsort()$ Step, those partitions can be $qsort()$ in parallel up to $h$ threads.

### 3.2 Parallel $qsort()$ Step

The Parallel Partition Step can be cutoff by $u$ elements to avoid over partitioning so that $qsort()$ can efficiently sort in each core's private L2 cache or shared L3 cache depending on the hardware. The Parallel $qsort()$ Step is on the else part of function $\_Qsort()$ on line 9 of Algorithm 2. Therefore, Cutoff $u$ should be parameterized in the experiment to achieve the best Speedup. As a result, if the Stdlib $qsort()$ performance is improved, the performance of PPMQSort will be automatically enhanced. Next, the time complexity of PPMQSort will be analyzed in $O()$ notation.

**Algorithm 2** The PPMQsort Algorithm

```
1: function _QSort(a, i_L, j_R, u)
2:     if i_L + u < j_R then                          ▷ if j_R − i_L > Cutoff u
3:         p ← ParallelPartition(a, i_L, j_R)         ▷ Parallel Partition Step
4:         OpenMP Task
5:         _QSort(a, i_L, p − 1, u)                   ▷ Left Subarray
6:         OpenMP Task
7:         _QSort(a, p + 1, j_R, u)                   ▷ Right Subarray
8:     else                                           ▷ else less than or equal Cutoff u
9:         qsort(a, i_L, j_R)                         ▷ Parallel qsort() Step
10:    end if
11: end function

12: function PPMQSort(a,start,end,h,u)                ▷ PPMQsort() Function
13:     begin OpenMP parallel with h threads
14:     OpenMP Single
15:     _QSort(a, start, end, u)                      ▷ with Cutoff u
16:     end parallel
17: end function
```

### 3.3 Complexity analysis

The time complexity of PPMQSort is analyzed assuming that all $c$ cores are 100 % utilized by running $h \geq c$ threads. The analysis can be divided into two steps: Parallel Partition Step and Parallel $qsort()$ Step as follows.

**Lemma 1** *Let n be the size of data array a, where* $a = a_0, a_1, \ldots, a_{n-1}$. *Then, the time complexity of* Parallel Partition Step *with h Threads on c cores where* $h \geq c$ *is* $O(n + \frac{n}{c} \log \frac{n}{2uc})$.

*Proof* At the beginning (level 1), the number of comparisons in Partition Phase with 2 Threads is $2 \times \frac{n}{4}$. Due to $c \geq 2$ cores, the time complexity is $\frac{1}{c} \times 2 \times \frac{n}{4} = \frac{2}{2}(\frac{n}{4})$. The number of swappings in Merge Phase with 1 Thread is $\frac{n}{4}$. Due to its sequential operation, its time complexity is $\frac{n}{4}$. In the first recursion level, the time complexity is hence $\frac{2}{2}(\frac{n}{4})+\frac{n}{4}$. In the second level, there are two independent partitions with $c \geq 2$ processor cores. The time complexity of Partition Phase with 2 Threads is $\frac{1}{c} \times 4 \times \frac{n}{8} = \frac{4}{c}(\frac{n}{8})$. The number of swappings in Merge Phase with 1 Thread is $2 \times \frac{n}{8}$. Due to its parallel operation, its time complexity is now $\frac{1}{c} \times 2 \times \frac{n}{8} = \frac{2}{c}(\frac{n}{8})$. The total time complexity of the second level is $\frac{4}{c}(\frac{n}{8}) + \frac{2}{c}(\frac{n}{8})$. The partitioning process is recursive until the condition on line 2 of Algorithm 2 is FALSE. That means the partition size is not larger than Cutoff $u$ elements. Based on the divide and conquer concept, the number of this recursive partitioning is $\log_2 \frac{n}{u}$ levels on average with respect to Cutoff $u$.

Therefore, the total time complexity of the Parallel Partition Step is

$$
\begin{aligned}
&= \frac{2}{2}\left(\frac{n}{4}\right) + \frac{n}{4} + \frac{4}{c}\left(\frac{n}{8}\right) + \frac{2}{c}\left(\frac{n}{8}\right) + \frac{8}{c}\left(\frac{n}{16}\right) + \frac{4}{c}\left(\frac{n}{16}\right) \\
&\quad + \cdots + \frac{2^{\log_2 \frac{n}{u}}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right) + \frac{2^{\log_2 \frac{n}{u}-1}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right) \\
&= 3 \times \left[\frac{2^0}{2}\left(\frac{n}{2^2}\right) + \frac{2^1}{c}\left(\frac{n}{2^3}\right) + \frac{2^2}{c}\left(\frac{n}{2^4}\right) + \cdots + \frac{2^{\log_2 \frac{n}{u}-1}}{c}\left(\frac{n}{2^{\log_2 \frac{n}{u}+1}}\right)\right] \\
&= 3 \times \left[n \sum_{l=1}^{\log_2 c} \frac{1}{2^l}\left(\frac{2^{l-1}}{2^{l+1}}\right) + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} \left(\frac{2^{l-1}}{2^{l+1}}\right)\right] \\
&= \frac{3}{4} \times \left[n \sum_{l=1}^{\log_2 c} \frac{1}{2^l} + \frac{n}{c} \sum_{l=\log_2 c+1}^{\log_2 \frac{n}{u}} 1\right] \\
&= \frac{3}{4} \times \left[n\left(1 - \frac{1}{c}\right) + \frac{n}{c} \log_2 \frac{n/u}{c}\right]. \\
&= \frac{3}{4} \times \left[n - \frac{n}{c} + \frac{n}{c} \log_2 \frac{n/u}{c}\right].
\end{aligned}
$$

$$= \frac{3}{4} \times \left[ n + \frac{n}{c} \log_2 \frac{n/u}{2c} \right].$$
$$= \frac{3}{4} \times \left[ n + \frac{n}{c} \log_2 \frac{n}{2uc} \right].$$

As a result, the time complexity of Parallel Partition Step is $O(n + \frac{n}{c} \log \frac{n}{2uc})$. □

**Lemma 2** *Let c processor cores perform qsort() each partition of size u elements in parallel. Since there are at least* $\frac{n}{u}$ *partitions, the time complexity of Parallel qsort() Step is* $O(\frac{n}{c} \log u)$.

*Proof* From Parallel Partition Step, at least $\frac{n}{u}$ partitions can be obtained. Each partition of up to $u$ elements is sorted by $qsort()$ in parallel up to $h \geq c$ threads. The time complexity of Parallel $qsort()$ Step is, therefore, $\frac{1}{c} \times \frac{n}{u} \times u \log_2 u$
$= \frac{n}{c} \log_2 u$
$= O(\frac{n}{c} \log u)$. □

**Theorem 1** (PPMQSort's Theorem) *The total time complexity of sorting n elements with the proposed PPMQSort running in parallel on* $c \geq 2$ *processor cores with Cutoff u elements and* $h \geq c$ *threads is* $O(n + \frac{n}{c} \log \frac{n}{2c})$.

*Proof* The complexities of Parallel Partition Step (see Lemma 1) and of Parallel $qsort()$ Step (see Lemma 2) are $O(n + \frac{n}{c} \log \frac{n}{2uc})$ and $O(\frac{n}{c} \log u)$, respectively. The total time complexity is $O(n + \frac{n}{c} \log \frac{n}{2uc} + \frac{n}{c} \log u)$. Therefore, the time complexity of PPMQSort is $O(n + \frac{n}{c} \log \frac{n}{2c})$. □

The time complexity of PPMQSort is similar to that of psort1 algorithm [15] as listed in Time Complexity row of Table 1. PPMQSort requires no extra space for intermediate results. As the data size $n$ and number of cores $c$ grow, PPMQSort can eventually outperform other algorithms due to its simplicity, scalability, and efficiency. The next section will show how PPMQSort is evaluated.

## 4 Performance evaluation and discussions

This section presents how various performance metrics are measured. The experiment setups and results are discussed later on.

### 4.1 Performance measurement

To investigate how the multicore architectures impact the performance of the algorithm, various performance metrics are measured and analysed.

1. CPU Time (in Seconds)
   To fairly compare $T_{\text{qsort}}$ and $T_{\text{ppmqsort}}$ in any experimental configurations, the CPU time is measured without data file loading and other overheads and averaged by 5 times.

2. Speedup $S$(x)
   This metric indicates that how many times our PPMQSort can be executed faster than the sequential Stdlib $qsort()$. Based on the measured $T_{\text{qsort}}$ and $T_{\text{ppmqsort}}$, Speedup $S$ can be computed as

$$S = \frac{T_{\text{qsort}}}{T_{\text{ppmqsort}}} \tag{1}$$

   where $\times$ denotes times.

3. Efficiency: Speedup/Core
   We would like to propose a new metric to measure the efficiency of any parallel QuickSort called Speedup per Core, $S/c$. $S/c > 1.00$ corresponds to superlinear Speedups. It can be due to cache locality/friendliness of the algorithm [23,24]. Similarly, [18] proposed a similar metric, Speedup/Thread instead. Higher thread counts $h$ can lead to more opportunities to achieve more parallelism that will be limited by hardware.

4. %CPU Utilization $U$
   The metric can be obtained from the contents of $/proc/stat$ file which keeps track of statistics of all HyperThread-enabled/disabled CPU cores. This %CPU Utilization is based on *user-time* only.

5. Cache Refs/Cache Misses
   Perf [25] is a software tool that relies on a number of hardware/software counters to collect statistics of CPU resource usages with minimal overhead [26]. For this paper, Cache Ref, $C$, Cache Misses, $C_m$ and other performance events are collected and averaged by 5 times to achieve high accuracy. In addition, a new metric called cache miss per second, $C_{m/s}$, can be obtained as shown in Eq. (2).

$$C_{m/s} = \frac{C_m}{T_{\text{ppmqsort}}} \tag{2}$$

   It can be beneficial to measure the number of cache misses per time unit especially for highly multithreaded programs. Larger $C_{m/s}$ may result in higher demands for memory bandwidth which will be presented next.

6. Branch Loads/Branch Load Misses
   Other important metrics of Perf are Branch Loads, $B$, and Branch Load Misses, $B_m$. They can be used to address the algorithm whose performance is limited by branch prediction, i.e., parallel QuickSort. Perf makes use of the hardware counters to measure the branch prediction unit. Similarly, a new metric called branch load misses per second, $B_{m/s}$, can be obtained as shown in Eq. (3).

$$B_{m/s} = \frac{B_m}{T_{\text{ppmqsort}}} \tag{3}$$

   $B_{m/s}$ can be regarded as number of branch mispredictions per time unit. Larger $B_{m/s}$ and $C_{m/s}$ may result in lower utilization of the long execution pipelines and frequent memory stalls which may affect %CPU Utilization $U$ eventually.

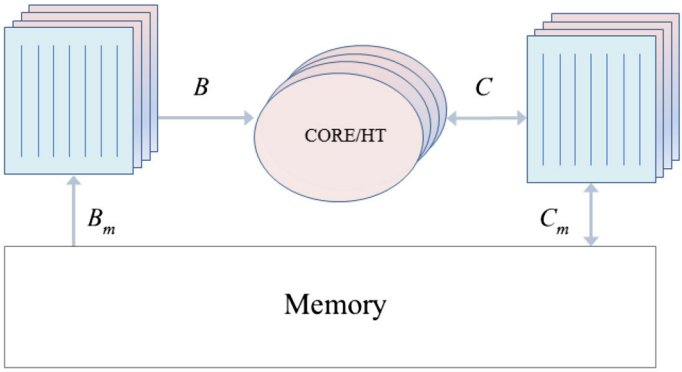


**Fig. 2** Our shared memory/multiprocessor/multicore system model measured by Perf (*blue boxes*on the *left*- and *right*-hand sides are instruction and data caches, respectively. *HT* HyperThread (color figure online)

7. Average Memory Bandwidth $M_{\text{bw}}$
   The complex interactions between multicore architecture and characteristics of a parallel algorithm directly and indirectly impact both branch mispredictions and cache misses. In multithreaded programs, off-chip memory bandwidth is one of the important metrics that can be the performance bottleneck due to memory contention, memory saturation and bad allocation among cores.
   Many researchers use hardware performance counters to track the amount of consumed memory bandwidth while the multithreaded program is running [27,28]. The measurement accuracy depends on measurement events, number of counters and the characteristics of memory system including DDR2/DDR3, channels (interleaving), bus clock frequency, etc. However, we cannot directly measure the amount of memory bandwidth consumption. This paper rather proposes a performance model to estimate and evaluate as shown in Fig. 2. Our model can utilize a number of available events measured by Perf resulted in Average Memory Bandwidth.

$$M_{\text{bw}} = f(B_{m/s}, C_{m/s}) \tag{4}$$

   Assume that $B_{m/s}$ has negligible effects due to small program size and its recursive nature. The majority of memory bandwidth should be proportional to $C_{m/s}$. Therefore, the Average Memory Bandwidth, $M_{\text{bw}}$, can be calculated in terms of cache line size $|C_{\text{line}}|$ multiplied by $C_{m/s}$ as shown in Eq. (5).

$$M_{bw} = |C_{\text{line}}| \times C_{m/s}. \tag{5}$$

### 4.2 Experiment setup

The results reported in this paper are based on five multicore CPUs: Intel E5405 Harpertown, Intel E5520 Nehalem-EP, Intel i3-2100 Sandybridge, Intel i7-2600 Sandybridge, and AMD A6-3650 APU. Table 3 provides a summary of these multicore systems.

**Table 3** Architectural details of multicore CPUs in our experiment

| System<br>Code name | Xeon E5405<br>Harpertown | Xeon E5520<br>Nehalem-EP | A6-3650<br>Llano | i3-2100<br>Sandybridge | i7-2600<br>Sandybridge |
|---|---|---|---|---|---|
| Clock (GHz) | 2.00 | 2.66 | 2.6 | 3.1 | 3.4 |
| $k \times c$ | $2 \times 4$ | $2 \times 4$ | $1 \times 4$ | $1 \times 2$ | $1 \times 4$ |
| HyperThread | No | Yes | No | Yes | Yes |
| L1/L2 (KB)/core | 32/3072 | 32/256 | 64/1024 | 32/256 | 32/256 |
| L3 (KB)/socket | – | 8192 | – | 3072 | 8192 |
| RAM (GB) | 4 | 12 | 16 | 8 | 16 |
| RAM | DDR2-667 | DDR3-800/1066 | DDR3-1866 | DDR3-1066/1333 | DDR3-1066/1333 |
| Others | | Smart Cache 8 MB | PCI express 2.0 | Smart cache 3 MB | Smart cache 8 MB |
| | | QPI2 5.86 GT/s | 16-way L2 | DMI 5GT/s | DMI 5GT/s |

**Table 4** Parameter set of the experiments

| Parameters | Values |
|---|---|
| Data types | Uint32, Uint64, Double |
| Size $n$ (M) | 10, 20, 50, 100, 200 |
| Cases | Random, Worst |
| Cutoff $u$ (K) | 50, 100, 200, 500 |
| Sockets $k$ | 1, 2 |
| Cores $c$ | 1, 2, 4, 8 |
| Threads $h$ | 1, 2, 4, 8, 16, 32 |
| HyperThread | Enable, Disable |
| Optimization | o2, o3 |

In every system listed in Table 3, the operating system is 64-bit Ubuntu 14.04 kernel 3.13 LTS. The PPMQSort is compiled with GCC 4.8 and linked with OpenMP 3.0 library under *-fopenmp* option. The measurement tool, Perf version 4.2, is called using *perf stat -r 5 -e* to profile PPMQSort algorithm for 5 times.

Data sets are unsigned 32-bit integer (*Uint32*), unsigned 64-bit (*Uint64*) and 64-bit double precision floating point (*Double*). These are generated using the GCC *random()* function with two distributions: Random and Worst-case and in different number of elements, $n$ = 10M, 20M, 50M, 100M, 200M. The first distribution contains random elements with small number of duplicates. The second distribution is generated such that the sequence seems to be sorted in a descending manner. However, for each distribution, the input sequence once generated is stored as a file. Therefore, both *PPMQSort* and sequential *qsort()* algorithms sort the same input sequences. All parameters are listed in Table 4.

### 4.3 Results and discussions

This subsection elaborates various aspects of the PPMQSort algorithm such as best Speedups, trade-offs between Speedup, Cutoff, and Thread, etc. Finally, the last two subsections are based on statistical analysis of Perf results.

#### *4.3.1 The best Speedups*

Table 5 tabulates the best Speedup, $T_{\text{qsort}}$, and $T_{\text{ppmqsort}}$ of all systems based on various data types, cases, and optimizations. The $T_{\text{qsort}}$ is obtained with the same experiment configuration as $T_{\text{ppmqsort}}$. It can be noticed that the best Speedups of Uint32 are higher than those of Uint64 and Double. Remark that i3-2100, i7-2600 and E5520 systems are HyperThread enabled. Therefore, their Speedups are higher than the number of physical cores. For a non-HyperThread 8-core Intel Xeon E5405 system, the best Speedup is as high as 7.75×. Due to limited space, best Speedups of Xeon E5404 are omitted. An exceptional case is the 4-core AMD A6-3600 whose Speedups are superlinear at 4.91× and 4.96× in Random and Worst cases, respectively. It can be observed that %CPU Utilizations approach 100 % in every Random-case configuration while those of Worst-case are significantly lower.

**Table 5** Best Speedup $S$ and other metrics on different data types and corresponding parameters

| CPU | Opt. | Uint32 | | | | Uint64 | | | | Double | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Random | | Worst | | Random | | Worst | | Random | | Worst | |
| | | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 |
| i3-2100 | $n$(M) | 200 | 200 | 200 | 200 | 100 | 100 | 200 | 200 | 200 | 200 | 200 | 200 |
| | $u$(K) | 50 | 100 | 50 | 50 | 200 | 500 | 50 | 50 | 200 | 500 | 50 | 50 |
| | $S$(x) | 3.19 | 3.03 | **3.79** | **3.67** | 2.88 | 2.79 | **3.44** | **3.35** | 2.82 | 2.72 | **3.69** | **3.63** |
| | $T_{\text{qsort}}$(s) | 39.80 | 39.88 | 14.88 | 14.89 | 20.39 | 20.37 | 16.83 | 17.00 | 45.18 | 45.21 | 17.72 | 17.87 |
| | $T_{\text{ppmqsort}}$(s) | 12.49 | 13.14 | 3.92 | 4.04 | 7.07 | 7.31 | 4.89 | 5.06 | 16.00 | 16.64 | 4.79 | 4.91 |
| | $h$ (threads) | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 16 | 8 | 8 |
| | $U$(%) | 98 | 98 | 86 | 85 | 96 | 96 | 82 | 80 | 95 | 94 | 84 | 83 |
| | $M_{\text{bw}}$ (MB/s) | 150 | 170 | *319* | *300* | 361 | 395 | **700** | *669* | 331 | 372 | *619* | *608* |
| | $B_{m/s}$ | *1.3e+8* | *1.4e+8* | 4.9e+6 | 4.7e+6 | *1.5e+8* | *1.6e+8* | 3.2e+6 | 3.1e+6 | *1.4e+8* | *1.4e+8* | 3.2e+6 | 3.1e+6 |
| A6-3600 | $n$(M) | 200 | 200 | 200 | 100 | 50 | 20 | 200 | 200 | 20 | 200 | 200 | 50 |
| | $u$(K) | 200 | 200 | 200 | 200 | 200 | 100 | 100 | 100 | 100 | 500 | 100 | 100 |
| | $S$(x) | 4.67 | **4.91** | **4.96** | 4.74 | 3.64 | 3.54 | **4.62** | **4.37** | 3.72 | 3.56 | **4.60** | **4.43** |
| | $T_{\text{qsort}}$(s) | 59.18 | 65.03 | 23.43 | 11.39 | 12.53 | 4.76 | 26.45 | 25.32 | 5.47 | 60.98 | 26.66 | 6.30 |
| | $T_{\text{ppmqsort}}$(s) | 12.67 | 13.25 | 4.71 | 2.40 | 3.45 | 1.35 | 5.71 | 5.78 | 1.47 | 17.15 | 5.79 | 1.42 |
| | $h$ (threads) | 4 | 16 | 4 | 4 | 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| | $U$(%) | 96 | 96 | 83 | 80 | 94 | 91 | 76 | 76 | 91 | 92 | 77 | 75 |
| | $M_{\text{bw}}$ (MB/s) | 1.85 | 2.25 | *4.39* | *5.98* | 9.11 | 6.53 | 4.57 | 3.81 | 6.85 | 2.30 | 5.97 | **20.33** |
| | $B_{m/s}$ | *4.3e+6* | *6.8e+6* | 2.4e+4 | 3.2e+4 | *1.3e+7* | *1.7e+7* | 7.9e+4 | 5.9e+4 | *2.1e+7* | *1.4e+7* | 6.4e+4 | 1.6e+5 |

**Table 5** continued

| CPU | Opt. | Uint32 | | | | Uint64 | | | | Double | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | Random | | Worst | | Random | | Worst | | Random | | Worst | |
| | | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 | o2 | o3 |
| i7-2600 | $n$(M) | 200 | 200 | 200 | 200 | 100 | 100 | 200 | 200 | 200 | 200 | 200 | 200 |
| | $u$(K) | 100 | 200 | 500 | 500 | 200 | 200 | 500 | 500 | 200 | 500 | 500 | 500 |
| | $S$(x) | 5.65 | 5.35 | **5.71** | **5.46** | **5.01** | **4.85** | 4.66 | 4.48 | 4.79 | 4.61 | **5.12** | **4.98** |
| | $T_{\text{qsort}}$(s) | 32.53 | 32.53 | 12.36 | 12.36 | 16.76 | 16.78 | 14.03 | 14.08 | 37.05 | 37.15 | 14.79 | 14.85 |
| | $T_{\text{ppmqsort}}$(s) | 5.76 | 6.08 | 2.16 | 2.26 | 3.34 | 3.46 | 3.00 | 3.14 | 7.74 | 8.06 | 2.88 | 2.98 |
| | $h$ (threads) | 16 | 16 | 8 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 |
| | $U$(%) | 92 | 92 | 78 | 75 | 90 | 90 | 72 | 70 | 87 | 86 | 75 | 73 |
| | $M_{\text{bw}}$ (MB/s) | 145 | 148 | *536* | *519* | 266 | 256 | **1042** | *993* | 249 | 279 | *971* | *958* |
| | $B_{m/s}$ | *1.2e+7* | *1.5e+7* | 4.9e+4 | 1.6e+5 | *2.2e+7* | *2.1e+7* | 9.6e+4 | 7.7e+4 | *2.5e+7* | *4.4e+7* | 9.5e+4 | 6.4e+4 |
| E5520 | $n$(M) | 200 | 200 | 200 | 100 | 200 | 200 | 50 | 100 | 200 | 200 | 100 | 100 |
| | $u$(K) | 100 | 100 | 500 | 200 | 100 | 200 | 200 | 500 | 200 | 500 | 500 | 500 |
| | $S$(x) | **12.29** | **11.20** | 9.44 | 8.60 | **11.34** | **10.96** | 8.16 | 7.26 | **9.43** | **9.06** | 8.40 | 7.90 |
| | $T_{\text{qsort}}$(s) | 72.35 | 70.00 | 21.05 | 10.57 | 80.41 | 80.53 | 6.33 | 12.17 | 69.40 | 69.37 | 12.65 | 12.70 |
| | $T_{\text{ppmqsort}}$(s) | 5.89 | 6.25 | 2.23 | 1.23 | 7.09 | 7.35 | 0.78 | 1.67 | 7.36 | 7.66 | 1.51 | 1.61 |
| | $h$ (threads) | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 16 | 32 | 32 | 16 |
| | $U$(%) | 83 | 82 | 67 | 59 | 80 | 80 | 55 | 54 | 73 | 73 | 59 | 56 |
| | $M_{\text{bw}}$ (MB/s) | 190 | 172 | *751* | *595* | 306 | 278 | *1537* | *1390* | 303 | 337 | **1558** | *1278* |
| | $B_{m/s}$ | *7.8e+8* | *4.0e+8* | 1.9e+8 | 1.9e+8 | *6.6e+8* | *6.1e+8* | 2.9e+8 | 3.8e+8 | 7.1e+8 | *1.2e+9* | 7.2e+8 | 4.1e+8 |

Bold values indicate the maximum results of each CPU
Italics values indicate the maximum results of each Data type

PPMQSort can achieve high Speedup regardless of the data types and randomness even in the Worst case. It can be obviously noticed that Worst-case $T_{\text{qsort}}$ and $T_{\text{ppmqsort}}$ are always faster than those of Random-case with the same configuration. Furthermore, their Speedups are almost always higher than those of Random-case except in dual-socket systems, E5520 and E5405. PPMQSort can exploit the Branch Prediction Unit and caches well, although $seq\_partition()$ must execute a large number of comparisons and swappings on lines 13 and 15 in Algorithm 1. That means the Branch Prediction unit can learn/yield higher prediction rate than the Random-case due to remarkably low Branch Misprediction Rate $B_{m/s}$ except those of E5520 cases.

However, the highest memory bandwidth $M_{\text{bw}}$ of Worst-case is always greater than Random-case because of its two to three times higher $C_{m/s}$. The highest $M_{\text{bw}}$ of each system is highlighted in bold face. This also concurs with Eq. (5) that memory bandwidth of PPMQSort depends heavily on $C_{m/s}$. Despite 2–3 orders of magnitude lower $B_{m/s}$, %CPU Utilization $U$'s of Worst-case are generally lower than those of Random-case in every configuration. It can be due to often memory stalls. On the other hand, high $B_{m/s}$ can be the performance bottlenecks in all Random-case as shown in Italic. Much lower $M_{\text{bw}}$ can be observed.

In both Random and Worst cases, Cutoff $u$ should fit the last level cache of each system. It can be noticed that the suitable Cutoff $u$ for Uint32 ranges between 50 and 200 K elements. For Uint64 and Double cases, Cutoff $u$ ranges between 200K and 500K elements or even bigger instead. The best Cutoff $u$ of i3-2100 (Uint32) is 50 K by majority vote. It seems like 50 K of Uint32 can fit the private L2 cache (256 KB) in each core. The rest can almost fit Cutoff in their last level caches except in some cases of $u = 500$ K of Uint64 and Double.

#### *4.3.2 Speedup S vs. Cutoff u and Thread h*

For a given system and experiment configuration, Speedup $S$ of PPMQSort is a function of Cutoff $u$ and Thread $h$. As already listed in Table 5, the best $S$ of i7-2600 system is 5.65× at $n = 200$ M of Uint32, $u = 100$ K, and $h = 16$ threads. Figure 3 shows a 3-D surface plot of PPMQsort with this configuration. Speedups can be visualized as surface height on the $Z$ axis with colors according to the Color bar on the right-hand side. This plot presents the scalability and trade-offs between Speedups, Cutoffs, and Threads. While increasing thread count $h$, the Speedup $S$ scales up for all Cutoffs. Therefore, high thread counts enable the PPMQSort to utilize the CPU cores more until $S$ saturates. As discussed earlier in Sect. 4.3.1 Best Speedups, while varying Cutoff $u$, Speedup changes slightly as darker and lighter colors at the same thread count. This behavior in this 3-D surface plot agrees with the derived time complexity in Theorem 1, where $u$ has been canceled out.

#### *4.3.3 HyperThread vs. non-HyperThread CPUs*

This subsection will contrast and compare Speedups of PPMQSort on Intel HyperThread and non-HyperThread CPUs with the same experiment configuration. Figure 4 illustrates Speedups (Line) and %CPU Utilization (Bar) of Intel HyperThread and non-HyperThread of PPMQSort (Uint32, Random-case, o2). The cyan bars and lines

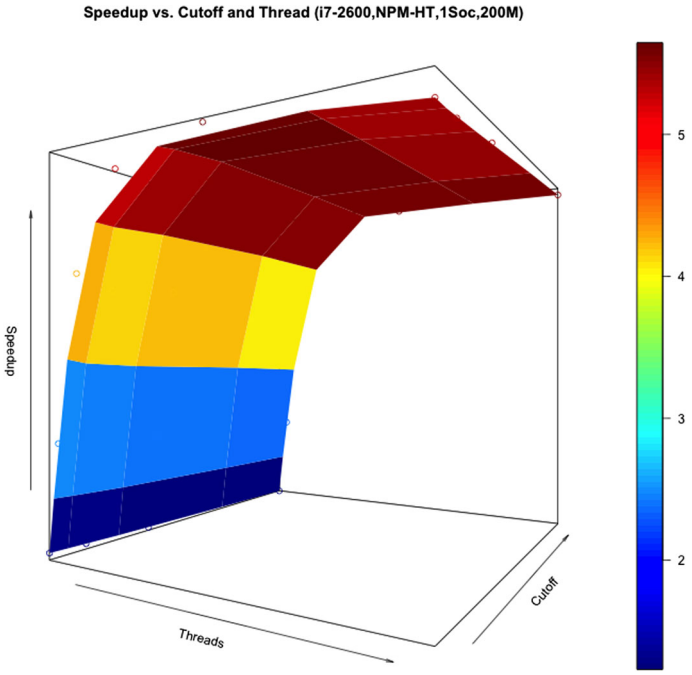


**Fig. 3** Three-D Surface Plot of Speedup, $S$ vs. Cutoff, $u$ and Thread, $h$ of PPMQSort on i7-2600 (Uint32, Random, o2, $n = 200$ M)

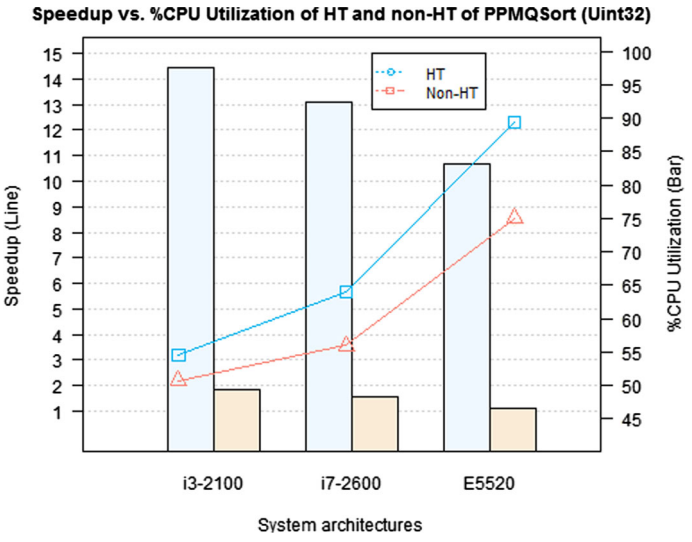


**Fig. 4** Best Speedup, S (*Line, Left*) vs. %CPU Utilization, U (*Bar, Right*) of PPMQSort on Intel Hyper-Thread (HT) and non-HyperThread (non-HT) Platforms (Uint32, Random, o2)

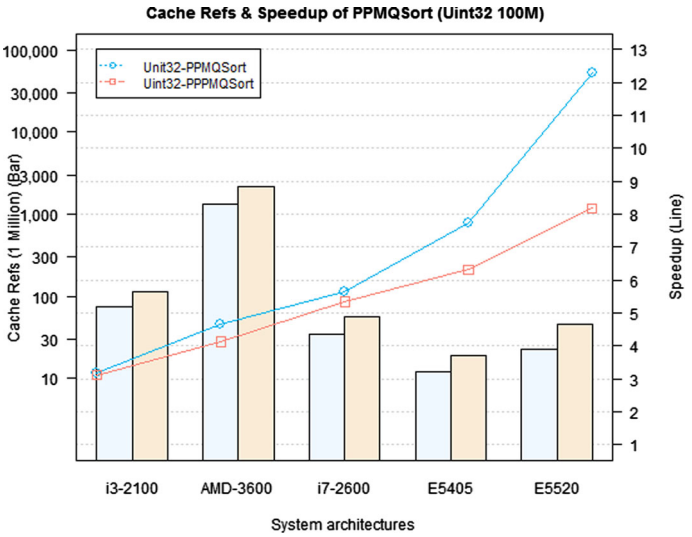


**Fig. 5** Best Speedup (*Line, Right*) vs. Cache Refs (*Bar, Left*) of PPMQSort (*Cyan*) and PPPMQSort (*Brown*) on all Platforms (Uint32, Random, o2) (color figure online)

are of HT enabled while the brown ones are HT disabled. The Speedup differences between HT-enabled and HT-disabled systems are significant due to lower average %CPU Utilization $U$, despite the fact that other statistics are similar. It can be roughly estimated that HT can boost up the performance by more than 50 % which is comparable to [29].

### *4.3.4 PPMQSort vs. PPPMQSort*

PPPMQSort is a minor variation of PPMQSort where its Merge Phase is parallelized with 2 threads on line 37 of Algorithm 1. To compare PPMQSort (Cyan) with PPPMQSort (Brown), their Speedups (Line) and Cache Refs (Bar) are plotted on all platforms (Uint32, Random, o2) with the same parameter set. Note that Cache Refs on the left Y axis are in logarithm and scaled by 1 million. It can be observed in Fig. 5 that PPMQSort can achieve better Speedups on the same experiment configurations due to significantly lower $C$.

Cache Refs are particularly high on AMD A6-3600 compared to other Intel systems. It might be due to fewer general-purpose Integer/Floaing-Point registers thus resulting in more register spills. However, AMD A6-3600 demands $M_{\text{bw}}$ up to 20.3 MB/s as listed in Table 5 due to both large private L1 data cache (64 KB/core) and L2 cache (1 MB/core). In addition, its Branch Load Misses/sec $B_{m/s}$'s are considerably lower than those of Intel systems. Therefore, its PPMQSort Speedups can be superlinear in some configurations. The rest is comparable on all Intel systems.

### *4.3.5 Efficiency: Speedup/Core*

Figure 6a, b depicts the scatter plot of $S/c$ vs. $c$ of non-HT and HT, respectively. It can be observed in Fig. 6a that PPMQSort can achieve $S/c \simeq 1.00$ or above (inside the

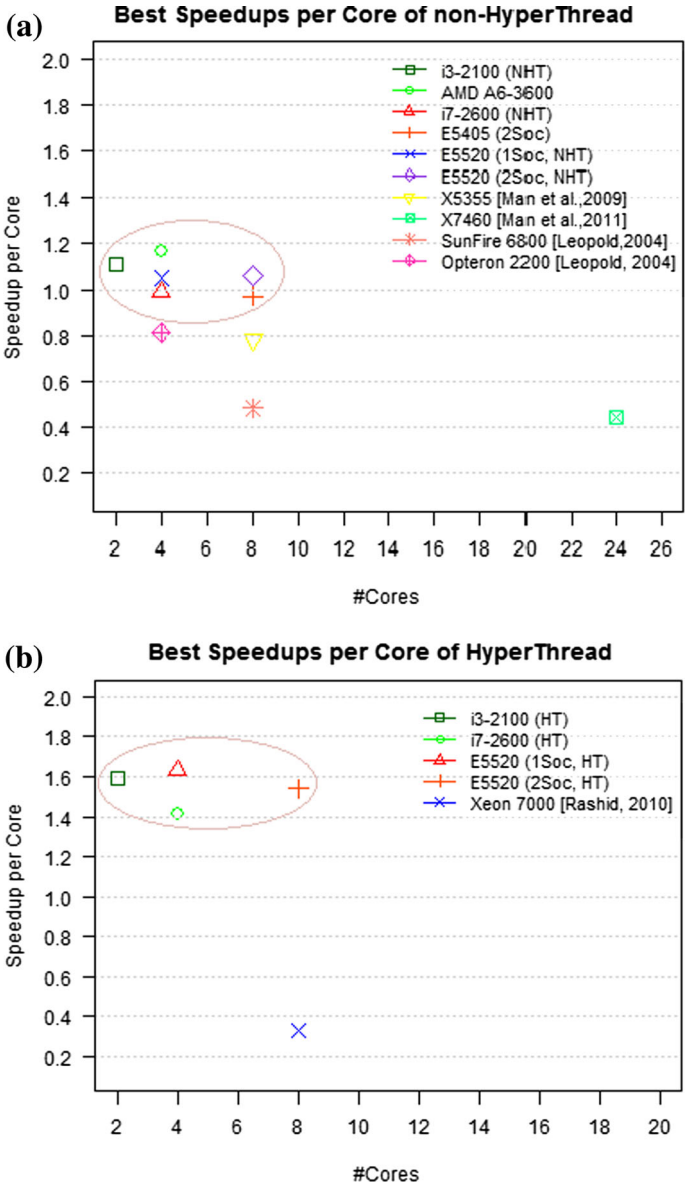


**Fig. 6** Speedups per Core S/c of PPMQSort (inside the oval) vs. Others (Random, Uint32) (**a**) non-HyperThread (NHT) (**b**) HyperThread (HT)

oval) while others can only reach up to 0.8. The HyperThread-disabled i3-2100 and non-HT A6-3600 can achieve $S/c$ at 1.11 and 1.17 resulting in superlinear Speedups because of high %CPU Utilization at 98 and 96, respectively. Similarly, Fig. 6b shows that PPMQSort can achieve $S/c \simeq 1.40$ or above while others can only reach up to 0.33. Some HT systems like i3-2100 and E5520 can achieve $S/c$ at 1.59 and 1.63,

respectively, because of better %CPU utilization with 3-MB and 8-MB Smart Caches, respectively. It can be concluded that PPMQSort can exploit the CPU cores much better than other algorithms on both non-HT and HT architectures. Moreover, PPMQSort can be scablable on any non-HT/HT/multicore/multi-socket systems with $S/c \simeq 1.00$ and $S/c \simeq 1.50$ or better.

#### *4.3.6 Comparison with previous implementations*

Table 6 compares our PPMQSort with previous parallel QuickSort implementations to show that we can achieve the best performance at data size around 100M 32-bit Integers with respect to $T_{\text{par}}$ and Efficiency, $S/c$. [18] reported only the Speedup based on Pthreads Library resulting in higher $S/c$ that may not compare against Stdlib $qsort()$. In addition, he also did not report the run time. With respect to 11.58× Speedup, our PPMQSort on an 8-core HyperThread E5520 can clearly outperform pqsort1 of [15] on an 8-core Xeon X5355 using the same $qsort()$ benchmark. Although [13] can achieve 25.03x Speedup on a 32-core UltraSPARC, their efficiency is not quite good and the benchmark may not be Stdlib $qsort()$.

On the Uint64 data, Man et al. [16] reported their highest Speedup of 10.47× for 100M random on 24 cores and $T_{\text{par}} = 2.712$ s. With only 8 cores, PPMQSort can achieve $S = 10.24\times$ at 3.54 s with the same configuration. This can confirm that PPMQSort is more efficient than others.

#### *4.3.7 Statistical analysis*

Figure 7 shows matrix scatter plots between $T_{\text{ppmqsort}}$ vs. Cache Refs $C$, Cache Misses $C_m$, Branch Loads $B$, and Branch Load Misses $B_m$ of PPMQSort on E5520, Uint32, o2, all Cutoffs, data sizes and threads. The upper half, the diagonal, and the lower half of the matrix plot illustrate the scatter plots, the density, and the correlation value between/of them, respectively. Each dot in the scatter plot represents an experiment configuration.

The top-row figures show the regression analysis between parameters that Time or $T_{\text{ppmqsort}}$ is proportional to Cache Refs $C$, Cache Misses $C_m$, Branch Loads $B$, and Branch Load Misses $B_m$, respectively. The green line is a linear regression generated by $lm()$ function in R Project (http://cran.r-project.org). The solid red line is a local regression smoothing (LOESS) mean fit line according to $loess()$ function. The red dotted lines above and below are positive and negative residual squares above and below the LOESS mean fit line, respectively.

$C_m$ is highly correlated with $C$ as indicated by correlation value $R_{C,C_{m/s}} = 0.91$. As expected, they are highly correlated. The higher $C$, the more $C_m$, and the longer the $T_{\text{ppmqsort}}$. Similarly, the higher $B$, the more $B_m$, and the longer $T_{\text{ppmqsort}}$. In addition, they are highly correlated with one another. Other systems in our experiment show similar behaviors.

**Table 6** Comparison of PPMQSort with other parallel QuickSort implementations (100 M, Uint32, Random), *NA* Not Available

| References<br>Year | PPMQSort | [18]<br>2013 | [20]<br>2010 | [15]<br>2009 | [14]<br>2004 | [14]<br>2004 | [13]<br>2003 |
|---|---|---|---|---|---|---|---|
| $n$(M) | 100 | 80 | 64 | 100 | 60 | 100 | 100 |
| $S$(x) | 11.58 | 3.8 | 2.65 | 6.22 | 3.24 | 3.875 | 25.03 |
| $T_{\text{seq}}$(s) | 34.78 | NA | 15.9 | 28.81 | 24.3 | 37.2 | 139.22 |
| $T_{\text{par}}$(s) | 3.00 | NA | 6 | 4.63 | 7.5 | 9.6 | 5.56 |
| $S/c$ | 1.45 | 1.9 | 0.33 | 0.26 | 0.81 | 0.48 | 0.78 |
| Using qsort() | Yes | No | No | Yes | No | No | No |
| Architecture | ×86 | ×86 | ×86 | ×86 | ×86 | UltraSPARC III | UltraSPARC |
| GHz | 2.66 | 2.66 | NA | 2.66 | 2.2 | 0.9 | 0.25 |
| $k \times c$ | 2 × 4 | 1 × 2 | 4 | 2 × 4 | 1 × 4 | 1 ×8 | 32 × 1 |
| HT | Yes | Yes | No | No | Yes | No | No |
| Cache | L3 8 MB | L2 3 MB | L2 4 × 2 MB | L2 2 × 2 MB | L2 2 × 1 MB | NA | L2 4 MB |
| Compiler | GCC -o2 | G++ | Intel C++ 9.1 | GCC -o2 | Intel C++ 8.1 -o3 | Guide | NA |
| Library | OpenMP 3.0 | pthreads | OpenMP 2.5 | OpenMP 2.0 | OpenMP 2.0 | NA | NA |
| Remarks | Xeon E5520 | Intel | Xeon 7000 | Xeon X5355 | Opteron 2200 | Sun Fire 6800 | Sun Enterprise |
| | MAC Pro 2010 | Dual-Core | PowerEdge 6800 | | sort_omp_2.0 | sort_pthreads_cv_1.0 | 10,000 |

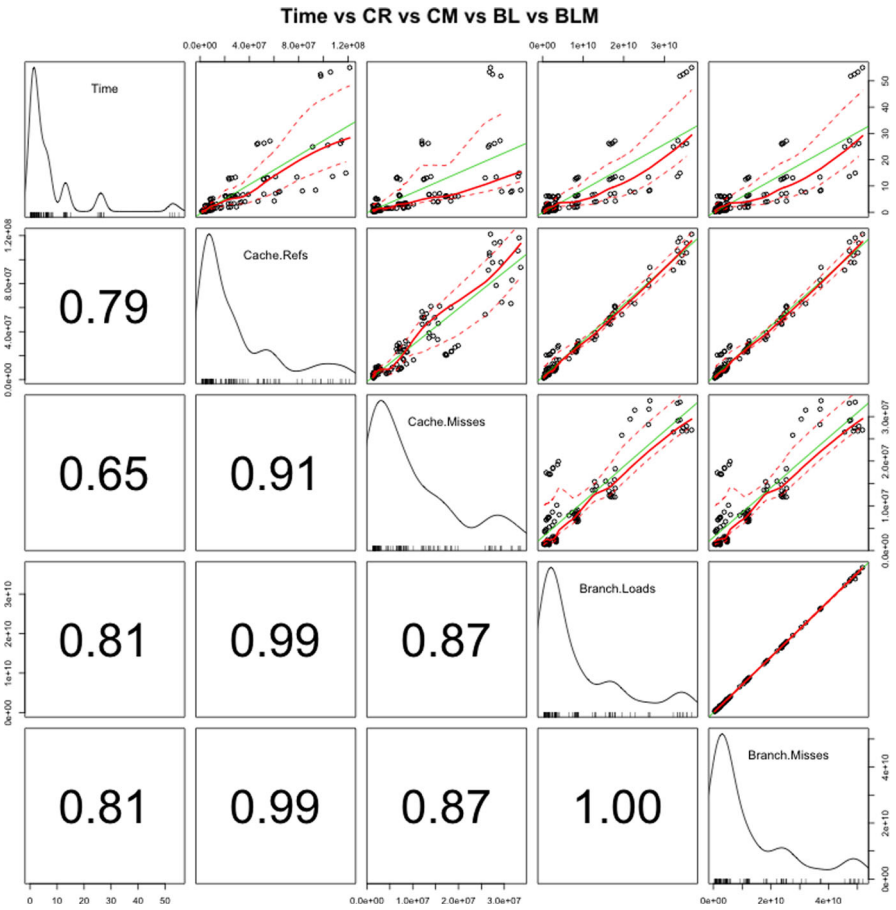


**Fig. 7** Matrix Scatter Plots between Time $T_{\text{ppmqsort}}$ vs. Cache Refs $C$, Cache Misses $C_m$, Branch Loads $B$, and Branch Load Misses $B_m$ of PPMQSort on E5520, Uint32, Random, o2, all cutoffs, data sizes, and threads

### *4.3.8 Speedup vs. %CPU Utilization vs. Memory Bandwidth*

Speedup $S$ vs %CPU Utilization $U$ vs. Cache Misses per Second $C_{m/s}$ and Branch Load Misses per Second $B_{m/s}$ of PPMQSort on i7-2600 can be depicted in Fig. 8. The configuration of this figure is random $n$ = 200 M Uint32, o2 and $h$ = 4–32 threads. Both $C_m/s$ and $B_m/s$ can be obtained by Eqs. (2) and (3), respectively.

As plotted, Speedup $S$ is directly proportional to %CPU Utilization $U$ because the correlation coefficient $R_{S,U}$ is 1.00. That means the higher %CPU Utilization, the better Speedup because all the forked threads can effectively execute with fewer memory stalls and pipeline stalls/flushes.

In general, cache misses can be due to cold misses, capacity misses, conflict misses, and coherence misses. Lower $C_{m/s}$ can be due to better cache locality resulted from suitable Cutoff $u$ and Thread $h$ of PPMQSort as shown in Fig. 8. On the other hand, lower $B_m/s$ represents infrequent branch mispredictions thus more efficient pipelin-

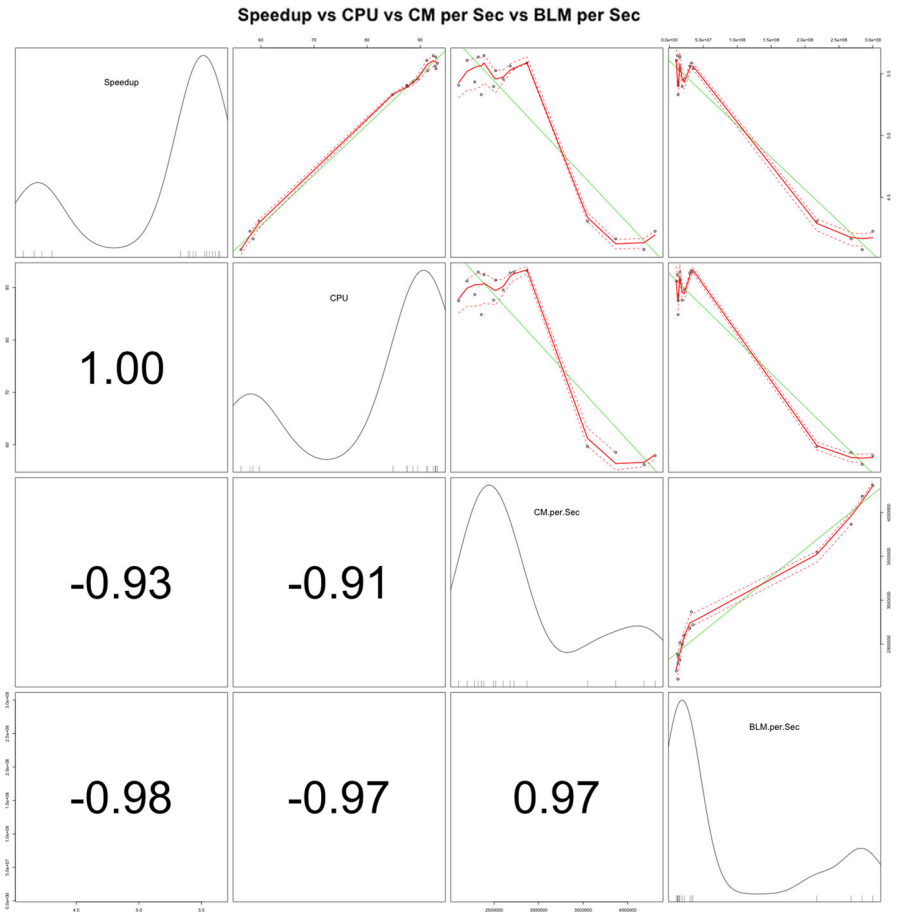


**Fig. 8** Speedup $S$ vs %CPU utilization $U$ vs. $C_{m/s}$ and $B_{m/s}$ of PPMQSort on i7-2600 (Uint32, 200M, random, o2, 4–32 threads)

ing. Both frequent cache misses and branch mispredictions per unit time can lead to memory stalls and pipeline stalls and thus lower $U$. It can be reflected on both $R_{U,C_{m/s}}$ and $R_{U,B_{m/s}}$ approaching $-1.00$. That means $U$ is negatively proportional to $C_{m/s}$ and $B_{m/s}$.

This figure confirms with the basic concept that memory is the bottleneck of the parallel algorithms [30] especially in the Worst case bounded by $C_m/s$. However, Random-case Speedups are limited by $B_m/s$ rather than $C_m/s$. As shown in Table 5, Random-case $B_m/s$'s are two to three orders of magnitude higher than those of Worst case with the same data size $n$ in one-socket systems. For dual-socket systems, the gap is not that wide. This results in almost three times longer Random-case $T_{\text{qsort}}$ and $T_{\text{ppmqsort}}$ than those of Worst case in the same table. As pointed out by Eyerman et al. [31], the misprediction penalty of superscalar CPUs with Reorder Buffer and deep pipeline equals to the number of clock cycles to refill the front-end pipeline.

Other systems show similar behaviors as the i7-2600 system. We can conclude that the branch prediction unit is as performance critical as the memory hierarchy for parallel sorting algorithms due to the randomness of input data in modern multicore CPUs.

## 5 Conclusion

The proposed PPMQSort algorithm is different from others as the partitioning process has been simply parallelized since the beginning. The basic concept of the PPMQSort is to divide the input data array by half in parallel/recursively until the obtained partitions are up to Cutoff size $u$. These partitions can be locally cached and $qsort()$ them simultaneously by $h \geq c$ threads. Hence, the performance bottleneck can be eliminated.

PPMQSort is compatible with the Stdlib $qsort()$ since we use it as a benchmark. Various OpenMP 3.0 parallel constructs are employed and coded in C language. Performance of PPMQSort was evaluated on one AMD and four Intel CPUs running 64-bit Ubuntu Linux 14.4 LTS. In general, PPMQSort can achieve the best Speedup up to and beyond the number of CPU cores. In spite of the Worst cases's fast $T_{\text{qsort}}$, their Speedups are almost always greater than those of Random. For HyperThread CPUs, PPMQSort can get up to 50 % Speedup increase over HT-disabled ones. In terms of efficiency, the PPMQSort can get Speedup/Core from 0.97 to 1.17 and from 1.41 to 1.63 on NHT and HT CPUs, respectively, and more superior than previous parallel QucikSort algorithms.

Statistical analysis of PPMQSort shows that $T_{\text{ppmqsort}}$ is proportional to Cache Misses and Branch Load Misses. On the other hand, its Speedup $S$ is proportional to %CPU Utilization $U$ and limited by $B_{m/s}$ and $C_{m/s}$. The proposed system performance model can estimate memory bandwidth required by the PPMQSort. In addition, Branch Prediction Units are as performance critical as the memory hierarchy for PPMQSort algorithm due to randomness of input data.

For future work, PPMQSort should be optimized further to support thread affinity/cache locality and minimize cache coherence misses even more. The performance model and average memory bandwidth shall be analyzed and fine-tuned to support a variety of algorithms/programs. To serve big data, task scheduling and load balancing strategy are investigated by mixed CPU, memory, and I/O-intensive [32].

In addition, on-chip and off-chip graphics processing unit (GPUs) should be investigated whether PPMQSort can be applied to exploit a massive number of GPU cores as it has been done on multicore CPUs.

**Acknowledgments** The authors wish to thank Mr. Apisit Rattanatranurak and Mr. Surapong Towtiamton for experiments and discussions on some of the algorithms in this paper. The authors wish to thank the reviewers for their insightful comments which greatly improved the paper.

## References

1. Hoare CAR (1962) Quicksort ACM 4:321
2. Sedgewick R (1978) Implementing quicksort program. Commun ACM 21(10):847–857

3. Mishra AD (2009) Selection of best sorting algorithm for a particular problem. Master's thesis, Thapar University, Computer Science and Engineering Department
4. Bhandarkar SM, Arabnia HR (1995) The hough transform on a reconfigurable multi-ring network. J Parallel Distrib Comput 24(1):107–114
5. Arabnia HR, Bhandarkar SM (1996) Parallel stereocorrelation on a reconfigurable multi-ring network. J Supercomput 10(3):243–269
6. Bhandarkar SM, Arabnia HR (1997) Parallel computer vision on a reconfigurable multiprocessor network. IEEE Trans Parallel Distrib Syst 8(3):292–309
7. Koch D, Torresen J (2011) Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In: Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '11. ACM, New York, pp 45–54
8. Mueller R, Teubner J, Alonso G (2012) Sorting networks on fpgas. VLDB J 21(1):1–23
9. Casper J, Olukotun K (2014) Hardware acceleration of database operations. In: Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays, FPGA '14. ACM, New York, pp 151–160
10. Capannini G, Silvestri F, Baraglia R (2012) Sorting on gpus for large scale datasets: a thorough comparison. Inf Process Manag 48(5):903–917
11. Xiaochen T, Rocki K, Suda R (2013) Register level sort algorithm on multi-core simd processors. In: Proceedings of the 3rd Workshop on Irregular Applications: Architectures and Algorithms, p 9. ACM
12. Heidelberger P, Norton A, Robinson JT (1990) Parallel quicksort using fetch-and-add. IEEE Trans Comput 39(1):847–857
13. Tsigas P, Zhang Y (2003) 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, pp 372–381
14. Sub M, Leopold C (2004) A user's experience with parallel sorting and openmp. In: Proc. of the 6th European Workshop on OpenMP (EWOMP 2004). Stockholm
15. Man D, Ito Y, Nakano K (2009) An efficient parallel sorting compatible with the standard qsort. In: International Conference on Parallel and Distributed Computing, Applications and Technologies. Hiroshima, pp 512–517
16. Man D, Ito Y, Nakano K (2011) An efficient parallel sorting compatible with the standard qsort. Int J Found Comput Sci 22(5):1057–1071
17. Kim KJ, Cho SJ, Jeon JW (2011) Parallel quick sort algorithms analysis using openmp 3.0 in embedded system. In: 11th International Conference on Control, Automation and Systems. KINTEX, Gyeonggi-do, pp 757–761
18. Mahafzah BA (2013) Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture. J Supercomput 66:339–363
19. Bingmann T (2015) Andreas Eberle, and Peter Sanders. Engineering parallel string sorting. Algorithmica, pp 1–52
20. Rashid L, Hassanein WM, Hammad MA (2010) Analyzing and enhancing the parallel sort operation on multithreaded architectures. J Supercomput 53:293–312
21. Saleem S, Lali MIU, Nawaz MS, Nauman AB (2014) Multi-core program optimization: parallel sorting algorithms in intel cilk plus. Int J Hybrid Inf Technol 7(2):151–164
22. Architecture Review Board (2014) The openmp api specification for parallel programming. http://www.openmp.org
23. Gustafson JL (1990) Fixed time, tiered memory, and superlinear Speedup. In: Proceedings of the Fifth Distributed Memory Computing Conference (DMCC5)
24. Helmbold DP, Mcdowell CE (1990) Modeling Speedup (n) greater than n. IEEE Trans Parallel Distrib Syst 1(2):250–256
25. Weaver VM (2013) Linux perf event features and overhead. In: Second International Workshop on Performance Analysis of Workload Optimized Systems (FastPath 2013). Austin
26. Zhang Y, Li ZP, Cao HF (2015) System-enforced deterministic streaming for efficient pipeline parallelism. J Comput Sci Technol 30(1):57–73
27. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing. 2nd ed. Pearson Education Limited
28. Akhter S, Roberts J (2006) Multi-core programming increasing performance through software multithreading. Intel Press, Hillsboro

29. Barker KJ, Davis K, Hoisie A, Kerbyson DJ, Lang Mike, Pakin Scott, Sancho Jose Carlos (2008) A performance evaluation of the nehalem quad-core processor for scientific computing. Parallel Process Lett 18(4):453–469
30. Wulf WA, McKee SA (1995) Hitting the memory wall: implications of the obvious. SIGARCH Comput Archit News 23(1):20–24
31. Eyerman S, Smith JE, Eeckhout L (2006) Characterizing the branch misprediction penalty. In: IEEE International Symposium on Performance Analysis of Systems Software (ISPASS 2006). Austin, pp 48–58
32. Qureshi K, Majeed B, Kazmi JH, Madani SA (2012) Task partitioning, scheduling and load balancing strategy for mixed nature of tasks. J Supercomput 59(3):1348–1359