J Supercomput (2013) 66:339–363
DOI 10.1007/s11227-013-0910-2

Performance assessment of multithreaded quicksort algorithm on simultaneous multithreaded architecture

Basel A. Mahafzah

Published online: 19 March 2013

Abstract Sorting huge amounts of datasets have become essential in many computer applications, such as search engines, database and web-based applications, in order to improve searching performance. Moreover, due to the witnessed prevalence of the commercial Simultaneous Multithreaded architecture (SMT), parallel programming using multithreading becomes a dire need for efficiently using all available hardware resources for one application. In this paper, one of the efficient and quick algorithms, the Quicksort, is applied as a parallel multithreaded algorithm on SMT architecture, where virtual parallelization has been achieved using the POSIX threads (Pthreads) library. The proposed algorithm is evaluated and compared with its sequential counterpart. The obtained analytical and experimental results reveal that multithreading is a viable technique for implementing the parallel Quicksort algorithm efficiently on SMT architecture, where it has been shown both analytically and experimentally that the parallel multithreaded Quicksort algorithm outperforms the sequential Quicksort algorithm in terms of various performance metrics including; time complexity and speedup.

Keywords Quicksort · Sorting · Performance evaluation · Multithreaded programming · Simultaneous multithreaded architecture

1 Introduction

Sorting algorithms have been introduced in the 1960s, where computer manufacturers in 1960s estimated that more than 25 % of the running time on their computers was spent on sorting [1, 2]. Since then, huge amounts of datasets have been accumulated in many computer applications, such as search engines, database, and web-based ap-

B.A. Mahafzah (✉)
King Abdullah II School for Information Technology, The University of Jordan, Amman 11942, Jordan
e-mail: b.mahafzah@ju.edu.jo

plications. These datasets need to be sorted for improving the performance of such applications. However, sorting huge datasets require a long time and large memory space. That is, if a dataset is not sorted, then when relevant data or action is required by an application, a searching process is performed in order to look for these data sequentially, from the start to the end of the dataset, which consumes a lot of processing time and large memory space; thus, degrading the performance of such applications. The rapid improvements in data collections and storage technology have enabled organizations to accumulate huge datasets. However, sorting these data collections has been proved as an extremely challenging problem. Therefore, the need for efficient sorting algorithms is a key requirement for improving the performance of these applications.

One common and fast sequential sorting algorithm is the Quicksort [3–5]. Performing the sequential Quicksort algorithm on a huge dataset is computationally an expensive process, which may degrade the performance of this algorithm, because it requires solving many partitions sequentially. However, one way to improve the performance of the sequential Quicksort algorithm for sorting a huge dataset is to parallelize it and implement it on a parallel system. Conversely, the parallel Quicksort algorithm is more expensive than the sequential algorithm since it requires special designed hardware, and it may degrade the performance of such algorithms for sorting small datasets due to the fact that the parallel algorithm requires expensive communications between processors in addition to the computation process. Dual-core microprocessors, which became prevalent with Simultaneous Multithreaded architecture (SMT) support, are invested as a solution, which encouraged the use of multithreaded parallel programming in order to efficiently use all available hardware resources for one application. Using multithreading, sequential computers are provided with virtual parallelization, thus providing faster execution and easy communication [6–10]. These attractive features are provided through creating a dynamic number of concurrent threads at the application's run time.

In this paper, a multithreading technique is applied, using the POSIX threads (Pthreads) library [7, 8], on the sequential Quicksort algorithm in order to design and implement a new parallel multithreaded Quicksort algorithm. More specifically, a parallel Kth-Partition algorithm has been designed and implemented for dividing the original input dataset into small partitions in order to distribute these partitions among different threads. Then each thread executes, concurrently with other threads, the sequential Quicksort algorithm on its partition. The new parallel multithreaded Quicksort algorithm is proposed for sorting large datasets, where the input data sequences were generated using the following five different distributions: random, zero, Gaussian, local, and remote distributions [11] for different sizes (from 10 MB to 80 MB). In order to evaluate the performance of the proposed algorithm, both algorithms, parallel multithreaded and sequential Quicksort, have been compared and evaluated analytically and experimentally in terms of various performance metrics including; time complexity, speedup, and efficiency.

In summary, the following contributions are drawn out of this paper:

- A new parallel multithreaded Quicksort algorithm is designed and implemented.
- This new algorithm is evaluated analytically in terms of time complexity, speedup, and efficiency.

- This new algorithm is evaluated experimentally under SMT architecture in terms of speedup and efficiency.
- A comparison is made between our new parallel multithreaded Quicksort algorithm and the sequential Quicksort algorithm.

Thus, the importance of these contributions is to show how multithreading technique takes advantage of all resources of SMT architecture for one application namely parallel multithread Quicksort, whereas sequential Quicksort does not. Therefore, the performance of our parallel multithread Quicksort algorithm is improved in terms of speedup and efficiency in comparison with the sequential Quicksort algorithm.

This paper is organized as follows. Section 2 reviews some of the previous research work performed on the Quicksort algorithm. The sequential Quicksort algorithm is presented in Sect. 3, while Sect. 4 presents the proposed parallel multithreaded Quicksort algorithm. This is followed, in Sect. 5, by an analytical assessment of the sequential and the proposed parallel multithreaded Quicksort algorithms. Afterward, the experimental environment of this work and the performance assessment of the experimental results are presented and discussed in Sects. 6 and 7, respectively. Finally, the conclusions drawn out of this paper are presented in Sect. 8.

2 Related work

Sorting algorithms are applied in various areas of applications, such as search engines, database applications, and web-based applications [5]. Hence, the demands on such algorithms are increasing rapidly. However, the huge increase of dataset sizes forms a big challenge for these algorithms because sorting a huge dataset using the sequential approach, such as insertion sort and merge sort [5, 12], requires a long time and a large memory space.

The Quicksort algorithm is one of the most common and fastest sequential sorting algorithms [3–5]. The Quicksort algorithm is a divide-and-conquer based algorithm [5, 13, 14], which performs the sorting process recursively. In general, the divide-and-conquer technique involves three steps at each level of the recursion. In the first step, the original problem is divided into several smaller subproblems, which are similar to the original problem, but they are smaller in size. During the second step, these subproblems are conquered in order to be solved recursively. However, if the subproblems' sizes are small enough, then, these subproblems can be solved in a straightforward manner. At the third step, the solutions of the subproblems are combined in order to get the final solution for the original problem.

Generally, designing a parallel language helps in improving end-user productivity [15]. Thus, one way to improve the performance of sorting a huge dataset is to use sorting networks and parallel sorting algorithms, such as Bitonic Sort, merging network, parallel Quicksort, parallel merging, etc. [5, 8, 16–18]. Such algorithms have been applied on different architectures and interconnection networks, using different parallel programming languages. For example, the Parallel Quicksort algorithm has been applied on Hypercube interconnection network [19], PC cluster [18], SUN

Enterprise 10000 [20], shared memory multiprocessor with an efficient implementation of the fetch-and-add operation [21], and on shared-address-space and message-passing systems [8]. However, parallel sorting algorithms require special hardware, which is highly costly.

Many computational and some sorting problems have been solved by using both single-threaded (i.e., sequential approach), and multithreaded techniques, using Pthreads and Java Threads, on a single processor, multiprocessor systems, and multicore and multithreaded architectures [6, 22–29]. The experimental results reveal the outperformance of multithreading over the single-threaded techniques [23, 25]. Several benefits are obtained by using the multithreading techniques, among of which are the improved application responsiveness, better program structure, efficient use of multiple processors and multithreaded architectures, and using fewer system resources [7, 23]. Moreover, using multithreading, all threads can be created from the same process, and they can share its resources, such as memory address and operating system. Therefore, the communication between concurrent threads can be executed within the process, without the operating system's interference, thus reducing the latency [7, 23].

Moreover, Rashid, Hassanein, and Hammad [29] implemented the parallel Quicksort in [20] and analyzed the memory behavior in terms of miss rates on two machines, where machine one supporting SMT and machine two supporting SMT, CMP (Chip Multiprocessors), and SMP (Symmetric Multiprocessors). However, in our work, we presented a new parallel multithreaded Quicksort algorithm using Pthreads running under SMT architecture, where the goal of our work is to show how multithreading technique takes advantages of all resources of SMT architecture, which yields to a better performance in comparison with sequential Quicksort algorithm.

3 Sequential Quicksort algorithm

The Quicksort algorithm is a divide-and-conquer based algorithm [5, 13, 14]. The sequential Quicksort algorithm [5] is presented in Fig. 1. More specifically, in Fig. 1, the Median_Partition procedure (Fig. 2) is called in order to return the pivot index. Accordingly, the input sequence is partitioned into two subsequences, where each subsequence is sorted recursively.

Figure 2 presents the median of the three rule algorithm [5]. Selecting the pivot using the median of three rule guarantees the time complexity of the sequential Quicksort to be at most $\Theta(n \log_2 n)$, especially when the elements in the input sequence are not all equal or completely sorted. Basically, the Median_Partition procedure chooses the median element from the three selected elements (first, middle, and last elements). After that, the selected element (median element) is swapped with the last element in the sequence, and finally, the Partition procedure (Fig. 3) is called.

Figure 3 shows the partition algorithm [5]. In general, the partition algorithm compares elements with the pivot, which is located at the end of the sequence, and moves the elements that are less than or equal to the pivot to the left sides of the sequence (array), which leaves the rest of the elements that are greater than the pivot on the right side of the sequence, then it moves the pivot from its location (last element) to its right location, then it returns the index of the pivot.

Fig. 1 Sequential Quicksort algorithm

```
1. **Sequential_Quicksort(Seq, first, last)**
2. if $first < last$
3. Then
4.     $p \leftarrow$ Median_Partition($Seq, first, last$)
5.     Sequential_Quicksort($Seq, first, p - 1$)
6.     Sequential_Quicksort($Seq, p + 1, last$)
```

```
1. Median_Partition(Seq, first, last)
2.     Select first element $Seq[first]$, middle element $Seq[\text{flour}((first+last)/2))]$, and
       last element $Seq[last]$ from input sequence $Seq$.
3.     Choose the median element $Seq[i]$ from the selected three elements.
4.     Exchange $Seq[last] \leftrightarrow Seq[i]$
5.     return Partition($Seq, first, last$)
```

Fig. 2 Median of three rule algorithm

Fig. 3 Partition algorithm

```
1. Partition(Seq, first, last)
2.     $z \leftarrow Seq[last] // z$ is the pivot
3.     $i \leftarrow first - 1$
4.     for $j \leftarrow first$ to $last - 1$
5.         do if $Seq[j] \leq z$
6.             then $i \leftarrow i + 1$
7.             exchange $Seq[i] \leftrightarrow Seq[j]$
8.     exchange $Seq[i + 1] \leftrightarrow Seq[last]$
9.     return $i + 1$
```

4 Parallel multithreaded Quicksort algorithm

In this section, the proposed parallel multithreaded Quicksort algorithm using POSIX threads (Pthreads) library is presented. Also, a demo example is presented in order to illustrate the design and the process of the proposed algorithm.

In general, the proposed parallel multithreaded Quicksort algorithm divides the original unsorted sequence (array) into t partitions, where each partition's size is equal to n/t elements, where n is the number of elements in the original unsorted sequence and t is the number of the assumed software-defined threads. Each partition is presented as a thread, which is created after each partition is produced. Then each thread sorts its partition using sequential Quicksort, concurrently, with other threads achieving virtual parallelization.

The proposed parallel multithreaded Quicksort algorithm is presented in Fig. 4. As the figure shows, the parallel multithreaded Quicksort procedure can be clarified as follows: the set of inputs for the algorithm are given in line 1. This set of inputs consists of an n-element input sequence *Seq*, the first element's index in the sequence (*first*), and the last element's index (*last*). The size of the sequence is then checked in line 2 in order to guarantee that the sequence *Seq* contains more than one element. If the condition is true, the algorithm is performed in order to sort the sequence

```
1.  Parallel_Quicksort(Seq, first, last)
2.  if first < last
3.  Then
4.      $size \leftarrow last + 1$ // assume first index = 0
5.      $t \leftarrow 2^{i-1}$;    // $i >= 1$
6.      $k \leftarrow 2^{i-1} - 1$
7.      $Kth_1 \leftarrow \lceil size/t \rceil - 1$; $Kth_2 \leftarrow Kth_1 + \lceil size/t \rceil \ldots Kth_k \leftarrow Kth_{k-1} + \lceil size/t \rceil$;
        // initial splitters
8.      $part_1 \leftarrow 0$; $part_2 \leftarrow 0 \ldots part_k \leftarrow 0$; // final splitters
9.      Parallel_Kth_Partition(Seq, last, $part_1, part_2 \ldots part_k, Kth_1, Kth_2 \ldots Kth_k$)
10.     $Thread_1 \leftarrow$ Sequential_Quicksort(Seq, 0, $part_1$)
11.     $Thread_2 \leftarrow$ Sequential_Quicksort(Seq, $part_1 + 1, part_2$)
12.       .
13.       .
14.       .
15.     $Thread_t \leftarrow$ Sequential_Quicksort(Seq, $part_k + 1$, last)
```

Fig. 4 Parallel multithreaded Quicksort algorithm

Seq of elements. In line 4, the size of the sequence *Seq* is computed. Next, the total number of threads is computed, in line 5, and used to distribute and sort elements concurrently (virtual parallelization), where the number of threads is denoted by t which is 2^{i-1}, where i is an integer that is greater than or equal to one. After that, the total number of splitters, denoted by k, is computed, which is equal to $2^{i-1} - 1$ (line 6). However, three types of splitters are used: initial, intermediate, and final. Line 7 shows the calculations of the initial splitters (first initial splitter Kth_1, second initial splitter Kth_2, and k^{th} initial splitter Kth_k) for the parallel Kth-Partition algorithm, where these initial splitters are used to partition the sequence into equal sizes in order to be assigned to t threads, in the parallel Kth-Partition algorithm. However, Kth_1 is equal to $\lceil size/t \rceil - 1$, Kth_2 equals $Kth_1 + \lceil size/t \rceil$, and Kth_k is equal to $Kth_{k-1} + \lceil size/t \rceil$. Afterward, in line 8, the final splitters $(part_1, part_2, \ldots, part_k)$ are initialized to zero, where $part_1$ is the first splitter of the sequence *Seq*, $part_2$ is the second splitter of the sequence *Seq*, and $part_k$ is the k^{th} splitter of the sequence *Seq*. These final splitters are calculated in the parallel Kth-Partition algorithm, where this algorithm is called in Line 9 in order to partition the sequence *Seq* into $k + 1$ partitions, that is, t partitions, which are distributed into t threads. Finally, in lines 10–15, a parallel multithreaded sorting is performed using t threads, where each thread sorts its partition using sequential Quicksort, simultaneously with other threads, achieving virtual parallelization. However, the size of each thread depends on the final splitters' values $(part_1, part_2, \ldots, part_k)$.

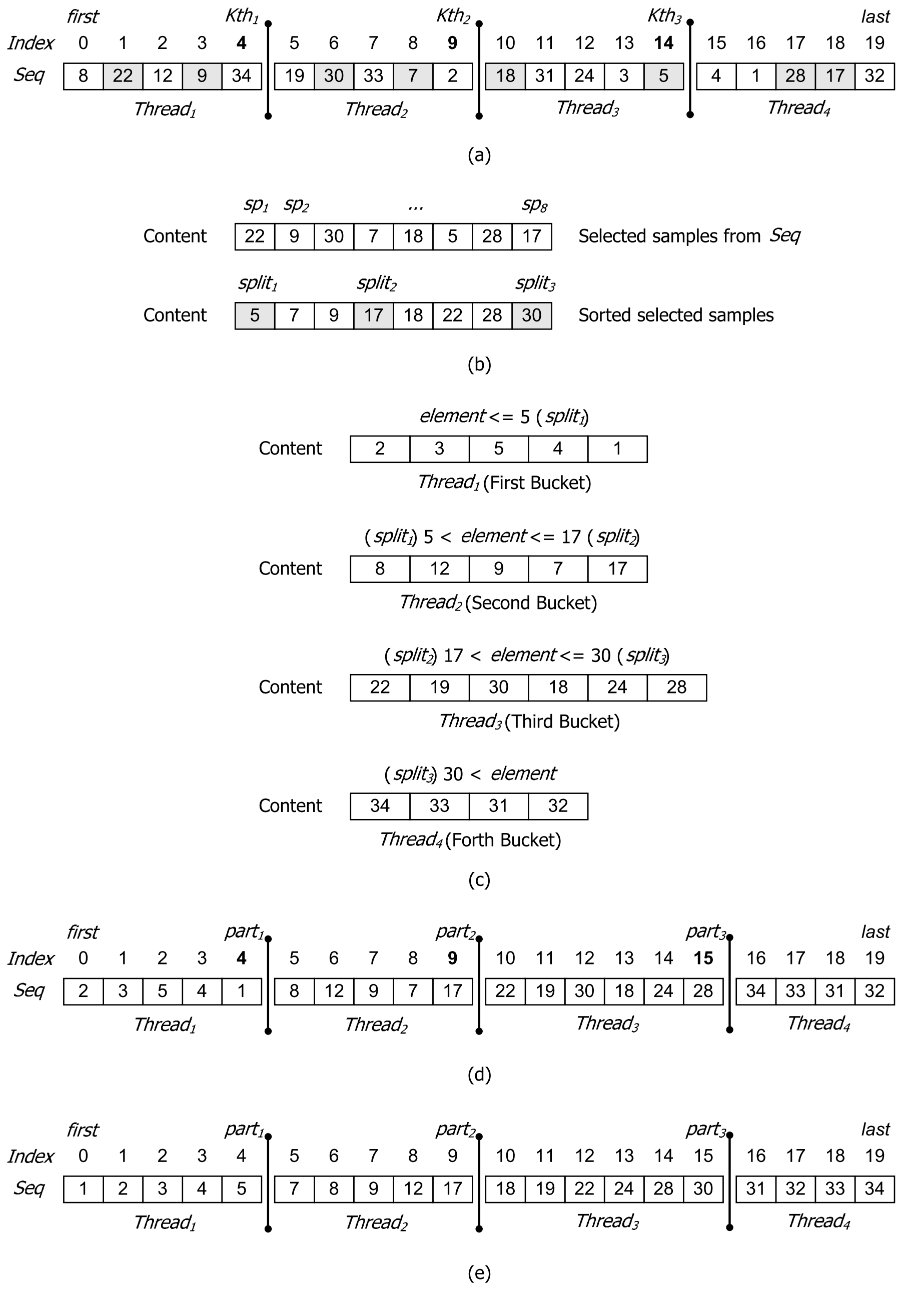
The parallel Kth-Partition algorithm is presented in Fig. 5. This algorithm can be better clarified as follows: Line 1 presents the input set of the algorithm, which is formed by the input n-element sequence *Seq*, *last*, $part_1, part_2, \ldots, part_k$, and $Kth_1, Kth_2, \ldots, Kth_k$; where *last* is the last position index in the sequence *Seq*, while $part_1, part_2, \ldots, part_k$ refer to the final k splitters, which are initially zero, as shown in Fig. 4, and $Kth_1, Kth_2, \ldots, Kth_k$ refer to the k initial splitters, which are received from the Parallel_Quicksort algorithm (Fig. 4), in order to initially partition the input

```
1.  **Parallel_Kth_Partition**($Seq, last, part_1, part_2, \ldots, part_k, Kth_1, Kth_2 \ldots Kth_k$)
2.    Select $s$ random elements from $Seq$, called samples ($sp_1, sp_2, \ldots, sp_s$).
3.    Sort these samples using sequential Quicksort.
4.    Select $k$ evenly-spaced elements from the sorted samples, called intermediate
      splitters ($split_1 < split_2 < \cdots < split_k$).
      // These intermediate splitters determine the ranges of $t$ virtual buckets.
5.    $Thread_1 \leftarrow$ distribute elements from the index 0 to the index $Kth_1$
6.    $Thread_2 \leftarrow$ distribute elements from the index $Kth_1 + 1$ to the index $Kth_2$
7.      .
8.      .
9.      .
10.   $Thread_t \leftarrow$ distribute elements from the index $Kth_k + 1$ to the index $last$
11.   $count_1 \leftarrow 0; count_2 \leftarrow 0 \ldots count_k \leftarrow 0;$
12.   All threads concurrently distribute elements into virtual buckets as follows:
13.   **If** $element <= split_1$
14.     **then** $Thread_1 \leftarrow$ assign element into first virtual bucket
15.       $count_1 = count_1 + 1;$
16.     **else** if $split_1 < element <= split_2$
17.     **then** $Thread_2 \leftarrow$ assign $element$ into second virtual bucket
18.       $count_2 = count_2 + 1;$
19.         .
20.         .
21.         .
22.     **else** if $split_{k-1} < element <= split_k$
23.     **then** $Thread_{t-1} \leftarrow$ assign element into $t - 1$ virtual bucket
24.       $count_k = count_k + 1;$
25.     **else** $Thread_t \leftarrow$ assign element into $t$ virtual bucket
26.   **return** $part_1 = count_1 - 1; part_2 = part_1 + count_2 \ldots part_k = part_{k-1} + count_k;$
27.     **return** $Thread_{t+1} \leftarrow$ copy elements from virtual buckets into the original
                          sequence $Seq$ in parallel.
```

Fig. 5 Parallel Kth-Partition algorithm

sequence *Seq* into t equal-sized partitions, which are distributed among t threads, where each partition belongs to one thread. Afterward, in line 2, s random elements, called samples $(sp_1, sp_2, \ldots, sp_s)$, are selected from the original input sequence *Seq*, where the number of these selected sample elements is very small. However, these selected sample elements are sorted using sequential Quicksort, as appears in line 3. In line 4, k evenly-spaced elements are selected from the sorted samples, called intermediate splitters $(split_1 < split_2 < \cdots < split_k)$, where these selected elements determine the ranges of virtual buckets. Moreover, these intermediate splitters are selected evenly-spaced in order to try to have no empty threads or small threads in size; that is to get almost equal threads in size, when virtual buckets allocated to threads. Next, in lines 5–10, the input sequence *Seq* is initially partitioned into t partitions, which are equal in size in order to have load-balanced threads. However, each partition is presented as a thread, which is created after its corresponding partition is produced. As shown in lines 5–10, the first initial splitter is denoted by Kth_1, which is an index that refers to the position ($\lceil size/t \rceil - 1$) in the input sequence *Seq*, the second initial splitter is denoted by Kth_2, which is an index of the position ($Kth_1 + \lceil size/t \rceil$) in the input sequence *Seq*, and the *Kth* splitter is denoted by Kth_k, which is an index of position ($Kth_{k-1} + \lceil size/t \rceil$) in the input sequence *Seq*. Each thread starts with n/t elements, where n is the number of elements in the input unsorted sequence *Seq* and t is the number of partitions (threads). In line 11, the counters ($count_1, count_2, \ldots, count_k$) are initialized to zero in order to count the number of elements in each thread (virtual bucket). In lines 12–25, all t threads simultaneously distribute elements into virtual buckets depending on the intermediate splitters ($split_1 < split_2 < \cdots < split_k$), which are elements selected from the sorted samples, where all the elements that are less than or equal to the first intermediate splitter $split_1$ are assigned to the first virtual bucket, and $count_1$ is used to count the number of elements in this first virtual bucket. As well, all the elements that are greater than the first intermediate splitter $split_1$ and less than or equal to the second intermediate splitter $split_2$ are assigned to the second virtual bucket, and $count_2$ is used to count the number of elements in this second virtual bucket, and so on until the elements that are greater than the intermediate splitter $split_k$ are assigned to the t^{th} virtual bucket. In line 26, the k final splitters ($part_1, part_2, \ldots, part_k$) are calculated and returned to the Parallel_Quicksort algorithm (Fig. 4). In addition, as shown in line 27, the elements are copied in parallel, from the virtual buckets to the original sequence *Seq* using a new thread and returned to the Parallel_Quicksort algorithm (Fig. 4).

Figure 6 shows a demo example of the proposed parallel multithreaded Quicksort algorithm. The proposed parallel multithreaded Quicksort algorithm calculates the size of the array which is equal to 20, computes the number of threads which is equal to 4, and finds the number of splitters which is equal to 3. Then the initial splitters are calculated, where Kth_1 is $(20/4) - 1 = 4$, while Kth_2 is equal to $4 + (20/4) = 9$, and Kth_3 is equal to $9 + (20/4) = 14$, as shown in Fig. 6(a). After that, the parallel Kth-Partition algorithm is performed. In this example, eight elements are selected randomly from the input sequence *Seq*. Then these selected elements are sorted using sequential Quicksort. Moreover, three intermediate splitters ($split_1, split_2$, and $split_3$) are selected (5, 17, and 30) from the sorted selected elements, as shown in Fig. 6(b). Note that, $split_1, split_2$, and $split_3$ are the first, middle, and last elements in the sample-sorted array respectively, where these splitters are evenly-spaced selected in order to obtain threads with no empty elements and no small threads in size. Also, this algorithm distributes the elements from the input sequence *Seq* to four threads ($Thread_1, Thread_2, Thread_3$, and $Thread_4$) based on the initial three splitters (Kth_1, Kth_2, and Kth_3). That is, $Thread_1$ contains elements from index 0 to index 4 (Kth_1), $Thread_2$ contains elements from index 5 ($Kth_1 + 1$) to index 9 (Kth_2), $Thread_3$ contains elements from index 10 ($Kth_2 + 1$) to index 14 (Kth_3), and $Thread_4$ contains elements from index 15 ($Kth_3 + 1$) to the last index 19, as shown in Fig. 6(a). Then these threads will, concurrently, distribute their elements into four virtual buckets based on the intermediate three splitters ($split_1, split_2$, and $split_3$), as shown in Fig. 6(c). That is, first bucket contains elements less than or equal to 5 ($split_1$), second bucket contains elements greater than 5 ($split_1$) and less than or equal to 17 ($split_2$), third bucket contains elements greater than 17 ($split_2$) and less than or equal to 30 ($split_3$), and forth bucket contains elements greater than 30 ($split_3$). Each



first
Kth1
Kth2
Kth3
last
Index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Seq 8 22 12 9 34 19 30 33 7 2 18 31 24 3 5 4 1 28 17 32
Thread1
Thread2
Thread3
Thread4
(a)
sp1 sp2 ... sp8
Content 22 9 30 7 18 5 28 17
Selected samples from Seq
split1 split2 split3
Content 5 7 9 17 18 22 28 30
Sorted selected samples
(b)
element <= 5 (split1)
Content 2 3 5 4 1
Thread1 (First Bucket)
(split1) 5 < element <= 17 (split2)
Content 8 12 9 7 17
Thread2 (Second Bucket)
(split2) 17 < element <= 30 (split3)
Content 22 19 30 18 24 28
Thread3 (Third Bucket)
(split3) 30 < element
Content 34 33 31 32
Thread4 (Forth Bucket)
(c)
first
part1
part2
part3
last
Index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Seq 2 3 5 4 1 8 12 9 7 17 22 19 30 18 24 28 34 33 31 32
Thread1
Thread2
Thread3
Thread4
(d)
first
part1
part2
part3
last
Index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
Seq 1 2 3 4 5 7 8 9 12 17 18 19 22 24 28 30 31 32 33 34
Thread1
Thread2
Thread3
Thread4
(e)

Fig. 6 A demo example of the parallel multithreaded Quicksort algorithm

of these four virtual buckets is presented as a new thread (Fig. 6(c)). Then the final three splitters ($part_1, part_2$, and $part_3$) are calculated in the parallel Kth-Partition algorithm (Fig. 5), and returned to the main procedure Parallel_Quicksort (Fig. 4). So, as a result, $part_1, part_2$, and $part_3$ are equal to 4, 9, and 15, respectively, as shown in Fig. 6(d). Also, the elements are, in parallel, copied from the four virtual buckets into the sequence *Seq*, which is presented as a single thread, and returned to the main procedure Parallel_Quicksort (Fig. 4) too. According to these final three splitters, the returned sequence *Seq* is partitioned into four partitions, where each partition is assigned to a new thread, as shown in Fig. 6(d). Finally, these four new threads perform the sequential Quicksort, in parallel, each to its partition. Figure 6(e) shows the final result, where the *Seq* is sorted.

5 Analytical assessment

This section presents the metrics that are used to perform an analytical evaluation of the proposed parallel multithreaded Quicksort algorithm; these metrics include: time complexity, speedup, and efficiency.

5.1 Time complexity

In general, the time complexity metric is the number of steps required by an algorithm to solve a problem [5, 8]. More specifically, the sequential Quicksort algorithm's time complexity is the number of steps from the start of the algorithm until the input n elements sequence is sorted, which is $\Theta(n \log_2 n)$ as an average case, where it is more common to happen in practice, and $\Theta(n^2)$ as a worst case, where this case happens when only the input sequence is already completely sorted, which is a rare case [5, 8]. This metric is expressed in the parallel multithreaded Quicksort algorithm as the number of steps required by the proposed parallel multithreaded Quicksort algorithm until the algorithm is completed; that is, the input sequence is sorted. The average case time complexity of the proposed parallel multithreaded Quicksort algorithm is given in Theorem 1. However, the time complexity here does not include the time (i.e., number of steps) that is required for creating and terminating concurrent threads in addition to the time (i.e., number of steps) of synchronizing shared data.

Theorem 1 *The average case time complexity of the parallel multithreaded Quicksort algorithm is* $\Theta(n/t \log_2 n/t)$.

Proof The proposed parallel multithreaded Quicksort algorithm contains two procedures namely: Parallel_Quicksort and Parallel_Kth_Partition, as shown in Sect. 4 (Figs. 1 and 2). The procedure Parallel_Quicksort requires $(a + n/t \log_2 n/t)$ as an average time complexity, excluding the Parallel_Kth_Partition algorithm, where a is a constant that presents basic operations and $(n/t \log_2 n/t)$ presents the average case time complexity for each thread performing the sequential Quicksort on an input sequence of size n/t, where n is the size of the original input sequence and t is the number of software-defined threads, since the average case time complexity of the sequential Quicksort is $\Theta(n \log_2 n)$ to sort n elements [5, 8]. All threads, thus concurrently, run the sequential Quicksort on an input of size n/t, which yields $\Theta(n/t \log_2 n/t)$

as an average case time complexity to sort an input sequence of n elements. Furthermore, the procedure Parallel_Kth_Partition requires $(b + n/t + s \log_2 s)$ as an average case time complexity, where b is a constant representing basic operations, (n/t) represents the process of partitioning the n elements input sequence into t partitions, where each partition is almost equal to n/t, and $(s \log_2 s)$ refer to the process of sorting s random selected elements from the n elements input sequence, using sequential Quicksort. Adding the average case time complexity of both procedures gives average case time complexity of the proposed parallel multithreaded Quicksort, which is $(a + n/t \log_2 n/t + b + n/t + s \log_2 s) \approx \Theta(n/t \log_2 n/t)$ by ignoring constants and lower terms, taking into account that $s < n/t$. □

5.2 Speedup

The analysis of the speedup metric provides an assessment of the performance improvement of the parallel multithreaded Quicksort algorithm over the sequential Quicksort algorithm. Therefore, the speedup S_p of the proposed parallel multithreaded Quicksort algorithm is equal to the ratio of the sequential Quicksort algorithm time complexity T_s to the parallel multithreaded Quicksort time complexity T_p for sorting the same data elements, as shown in Eq. (1) [8, 30]. Thus, the average case speedup of the proposed parallel multithreaded Quicksort algorithm is given in Theorem 2.

$$S_p = T_s/T_p \tag{1}$$

Theorem 2 *The average case speedup of the parallel multithreaded Quicksort algorithm is $\Theta((t \log_2 n)/(\log_2 n - \log_2 t))$.*

Proof Since the average case time complexity of the sequential Quicksort is $\Theta(n \log_2 n)$ [5, 8] and the average case time complexity of the parallel multithreaded Quicksort is $\Theta(n/t \log_2 n/t)$ as shown in Theorem 1, using Eq. (1) the average case speedup of the proposed parallel multithreaded Quicksort algorithm is $\Theta((n \log_2 n)/(n/t \log_2 n/t)) = \Theta((t \log_2 n)/(\log_2 n - \log_2 t))$. □

5.3 Efficiency

The efficiency of the proposed parallel multithreaded Quicksort algorithm is a measurement of how effectively Simultaneous Multi-Threading (SMT) hardware threads are utilized. The efficiency E_f is, therefore, expressed as the ratio of the speedup S_p to the number of SMT hardware threads, which is denoted by h, as shown in Eq. (2) [8, 30]. Theorem 3 shows the average case efficiency of the proposed parallel multithreaded Quicksort algorithm.

$$E_f = S_p/h \tag{2}$$

Theorem 3 *The average case efficiency of the parallel multithreaded Quicksort algorithm is $\Theta((ht \log_2 n)/(\log_2 n - \log_2 t))$.*

Table 1 Analytical comparison

Metric	Sequential Quicksort	Parallel Multithreaded Quicksort
Time Complexity	$\Theta(n \log_2 n)$	$\Theta(n/t \log_2 n/t)$
Speedup	–	$\Theta((t \log_2 n)/(\log_2 n - \log_2 t))$
Efficiency	–	$\Theta((ht \log_2 n)/(\log_2 n - \log_2 t))$

Proof Since the average case speedup of the proposed parallel multithreaded Quicksort algorithm is $\Theta((t \log_2 n)/(\log_2 n - \log_2 t))$ as shown in Theorem 2, using Eq. (2) the average case efficiency of the proposed parallel multithreaded Quicksort algorithm is $\Theta(((t \log_2 n)/(\log_2 n - \log_2 t))/h) = \Theta((ht \log_2 n)/(\log_2 n - \log_2 t))$. □

5.4 Summary of analytical results

The proposed parallel multithreaded Quicksort algorithm achieves virtual parallelization using Pthreads library, which improves the performance of the proposed parallel algorithm over the sequential Quicksort. More specifically, as the number of concurrent software-defined threads in the proposed parallel multithreaded Quicksort algorithm increases, the speedup is increased, while the required execution time is reduced since the analytical time complexity here does not include the time that is required for creating and terminating concurrent threads in addition to the time of synchronizing shared data. Thus, the previous analysis in Sects. 5.1–5.3 proves that, compared to sequential Quicksort algorithm, the proposed parallel multithreaded Quicksort algorithm excels in most performance metrics, as shown in Table 1, which summarizes the assessment metrics for both the sequential Quicksort and parallel multithreaded Quicksort.

6 Experimental environment

The experimental runs were performed on a Dual-Core Intel Processor (CPU 2.26 GHz) with 14 pipeline stages and Hyper-Threading Technology (HT) based on Simultaneous Multithreaded architecture (SMT), where each core has dual SMT hardware threads. So, the experimental runs were performed on four SMT hardware threads in total. Multithreaded architecture enhances instruction throughput by issuing multiple instructions from multiple threads within one clock cycle [9, 10]. Moreover, the experimental system is supported with 2 GB RAM, and 3 MB L2 Cache.

Both the parallel multithreaded and sequential Quicksort algorithms are implemented using an object-oriented programming language (C++) [31], where the C++ programs are compiled using G++ compiler, which is a GNU compiler collection version 3.3.5, running under SUSE Linux operating system version 10. Moreover, this work uses the POSIX threads (Pthreads) library [7, 8] for writing the proposed parallel multithreaded Quicksort algorithm and for supporting the multithreaded functions, under SUSE Linux operating system. The Pthreads standard library was specified by IEEE, and has become a popular standard for writing parallel programs due to some features, such as inexpensive thread creation and termination, a large number of threads can be created for parallel programs, and finally, a program, which is written using Pthreads, can be run on both sequential and multiprocessor computers [7, 8]. However, to write a program using Pthreads, there are some issues that should be taken into consideration, such as Pthreads scheduling and synchronization to ensure the consistency of the shared data [7, 8].

The proposed parallel multithreaded Quicksort algorithm is implemented using the Pthreads library in order to achieve virtual parallelization, where the number of software-defined threads varies from two to eight, sharing the four SMT hardware threads of the Dual-Core Intel Processor. Using the Pthreads library, the number of instructions issued by independent software-defined threads, which can be executed simultaneously in a pipelined processor, is limited by the number of that processor's pipeline stages. If the number of concurrent instructions issued by the software-defined threads is not greater than the number of processor's pipelined stages, then all the concurrent instructions can be simultaneously executed in distinct stages of the pipelined processor. Otherwise, the number of instructions that can be executed concurrently is equal to processor's pipelined stages. Thus, all the concurrent instructions are scheduled to use the pipeline stages. That is, at a certain point of execution time, all the processor's pipeline stages are full of instructions to be executed in the processor, while the rest of instructions will be waiting to use the pipelined processor. The multithreaded architecture optimizes the throughput of multiprogramming workloads rather than single-thread performance [9], that is, the multithreaded architecture enhances instruction throughput by allowing several independent threads to issue instructions to multiple functional units in a single cycle [10].

The experimental algorithms use array data structure as an input sequence for saving data elements, which will be sorted. These data elements are integers, which are generated using five distributions: random, zero, Gaussian, local, and remote [11]. The first distribution is called random distribution, where elements are generated randomly using the C++ library's random function. Therefore, the resulting input sequence contains random elements without a high number of duplicates. The second distribution is called zero distribution, which is similar to the random distribution, except that every tenth element was replaced with a zero value. Therefore, the resulting input sequence contains a high number of duplicates of one element, in particular, the zero value. The third distribution is called Gaussian distribution, where the mean is set to $range/2$, and sigma is set to $range/16$, where $range$ is equal to 2^{32} for 32-bit unsigned integers. The Gaussian distribution is meant to create values within a tight range and with a higher number of duplicates compared to zero and random distributions. The fourth distribution is called local distribution, where the input sequence is generated such that the smaller elements are located in the front side of the sequence and the larger elements are located in the back side of the sequence, which means that the sequence seems to be nearly sorted in an ascending manner. The fifth distribution is called remote distribution, where the input sequence is generated such that the smaller elements are located in the back side of the sequence and the larger ones are located in the front side of the sequence, which means that the sequence seems to be nearly sorted in a descending manner.

However, for each distribution, the input sequence elements are generated eight times. In each time, the generated elements are saved in different input sequence sizes (10 MB, 20 MB, 30 MB, 40 MB, 50 MB, 60 MB, 70 MB, and 80 MB). So, both the proposed parallel multithreaded and sequential Quicksort algorithms are executed for sorting these input sequences.

Conducting a comparison between the proposed parallel multithreaded Quicksort algorithm and other parallel sorting algorithms and sorting networks mentioned in the related work section (Sect. 2) is not applicable since these algorithms are implemented and run on different parallel machines and interconnection networks, not on a single Dual-Core machine with SMT architecture support, which is the most currently available machine nowadays. Therefore, the next section presents an experimental comparison between the proposed parallel multithread and sequential Quicksort algorithms.

7 Performance assessment of experimental results

In this section, the experimental runs of the proposed parallel multithreaded Quicksort algorithm and the sequential Quicksort algorithm are presented. More specifically, the experimental runs of the proposed parallel multithreaded Quicksort algorithm using different number of concurrent threads (2, 4, and 8 software-defined threads) and the sequential Quicksort algorithm are compared and discussed both; for different problem sizes (from 10 MB to 80 MB) and different input sequences of data distributions (random, zero, Gaussian, remote, and local distributions) in terms of the following performance metrics: speedup and efficiency.

7.1 Speedup

Speedup provides an evaluation of the performance improvement of the parallel multithreaded Quicksort algorithm over the sequential Quicksort algorithm. So, in this section, the relative speedup S_p of the proposed parallel multithreaded Quicksort algorithm is presented. More specifically, the relative speedup of the proposed parallel multithreaded Quicksort algorithm is equal to the ratio of the sequential Quicksort algorithm execution time T_s to the parallel multithreaded Quicksort execution time T_p for sorting the same data elements, as shown in Eq. (1) (Sect. 5.2) [8, 30, 32]. The execution time, in general, is the amount of time required by an algorithm to solve a problem. The sequential Quicksort algorithm's execution time is the time consumed from the start of the algorithm until the input sequence is sorted [8]. This metric is expressed in the parallel multithreaded Quicksort algorithm as the amount of time required by the parallel multithreaded Quicksort algorithm until the algorithm is completed; that is the input sequence is sorted. Thus, the execution time here includes the time that is required for creating and terminating concurrent software-defined threads in addition to the time of serializing the algorithm and synchronizing shared data.

Figures 7–11 show the speedup of the parallel multithreaded Quicksort algorithm according to Eq. (1) (Sect. 5.2) using two, four, and eight concurrent software-defined threads on random, zero, Gaussian, local, and remote distributions, where the problem size (input sequence size) ranges from 10 MB to 80 MB. It is clear from these figures that the parallel multithreaded Quicksort algorithm achieves better speedup than the sequential Quicksort algorithm for all problem sizes and distributions.

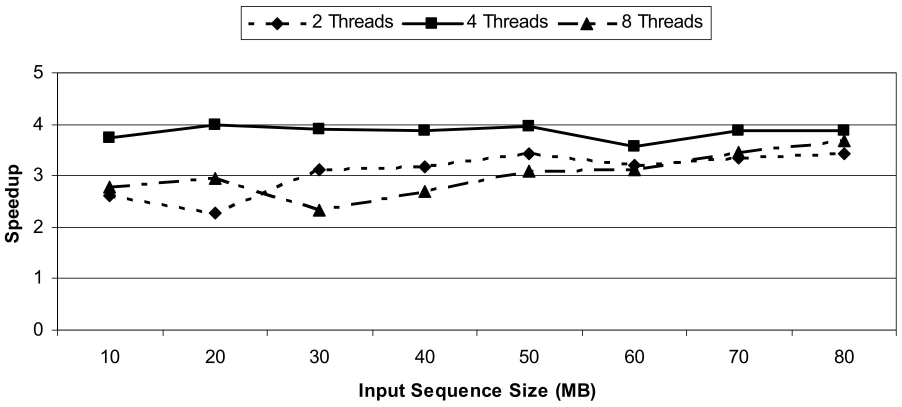


Fig. 7 Speedup of parallel multithreaded Quicksort algorithm for random distribution

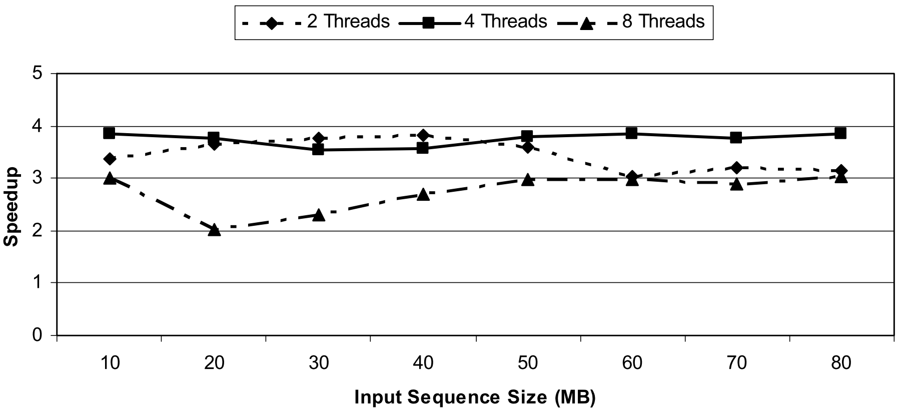


Fig. 8 Speedup of parallel multithreaded Quicksort algorithm for zero distribution

As Fig. 7 shows, the proposed parallel multithreaded Quicksort algorithm achieves higher speedup (between 3.58 and 3.99) using four concurrent threads than using two or eight concurrent threads for all problem sizes (from 10 MB to 80 MB), where the input sequence is presented as random distribution. This is because the proposed parallel algorithm has lower execution time using four concurrent threads than two or eight concurrent threads. Also, the proposed parallel algorithm achieves higher speedup using two concurrent threads than eight concurrent threads for problem sizes of 30 MB to 60 MB since eight concurrent threads require more execution time. This refers to the fact that more overhead is added for threads' management and threads' sizes are largely varied, that is, unbalanced threads' sizes, where some threads are very small and others are very large. However, the proposed parallel algorithm using eight concurrent threads achieves slightly higher speedup than using two concurrent threads for large problem sizes (70 MB–80 MB). In this case, using eight concurrent threads for large problem sizes yields enough large threads to achieve higher speedup.

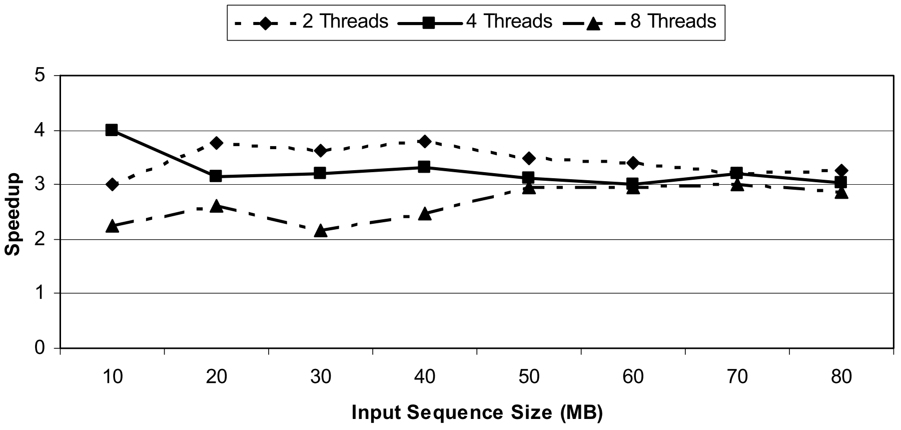


Fig. 9 Speedup of parallel multithreaded Quicksort algorithm for Gaussian distribution

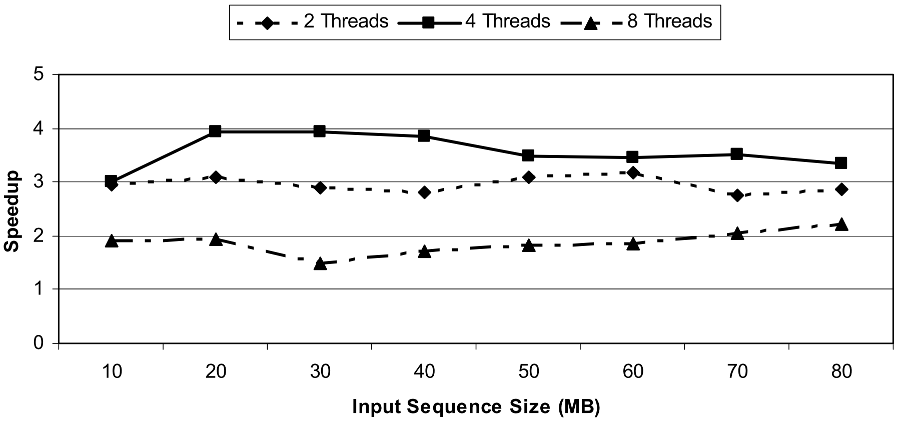


Fig. 10 Speedup of parallel multithreaded Quicksort algorithm for local distribution

Moreover, for small problem size (20 MB), using two concurrent threads gives worse speedup than using eight concurrent threads since one thread is small in size and the other is large, leading to only 2.27 times better speedup using two concurrent threads (parallel multithreaded Quicksort) than the sequential Quicksort. Finally, in case of eight-thread scenario, Fig. 7 shows that as the problem size increases from 30 MB to 80 MB, the speedup becomes higher (increases from 2.34 up to 3.7), since all eight threads work concurrently on almost same problem size, that is, the work load is almost evenly distributed among threads.

In Fig. 8, the proposed parallel multithreaded Quicksort algorithm achieves higher speedup using four concurrent threads than using two or eight concurrent threads for problem of sizes between 50 MB and 80 MB, where the input sequence is presented as zero distribution. This is because the proposed parallel algorithm has lower execution time using four concurrent threads than two or eight concurrent threads, and

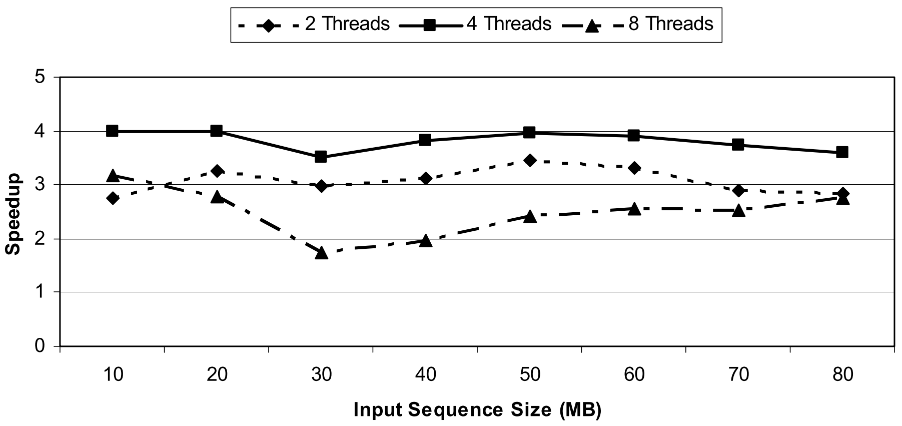


Fig. 11 Speedup of parallel multithreaded Quicksort algorithm for remote distribution

the zero distribution contains a high number of duplicates of one key, which is zero. Therefore, the thread that contains this duplicate key will be very large in size, where the speedup drops for large input sequence, when using two concurrent threads. In summary, the four-thread scenario speedup is the best for size ranges between 50 MB and 80 MB, since four threads working concurrently where in the case of sequential Quicksort only one thread do the sorting for large problem size and in case of two-thread scenario still the problem size is too large for two threads. However, in case of eight-thread scenario, the problem size becomes small and more communication is required between threads, which lead to less speedup. Moreover, in Fig. 8, when the problem size increases from 20 MB to 50 MB, higher speedup is achieved (increases from 2 to 2.97) using eight concurrent threads, the reason behind this increase is the problem size, as the problem size increases the speedup increases too, then stays almost stable as the problem size increases from 50 MB to 80 MB, since each thread has almost same size of work load.

Figure 9 shows that the proposed parallel multithreaded Quicksort algorithm achieves higher speedup using two concurrent threads than using four or eight concurrent threads for problem sizes in the range from 20 MB to 80 MB, where the input sequence is presented as Gaussian distribution. This is because the proposed parallel algorithm has lower execution time using two concurrent threads than four or eight concurrent threads. The reason behind this is that the Gaussian distribution presents input sequence within a tight range and with a higher number of duplicates as compared to the zero and random distributions. Thus, there would be some small threads and others that are larger in size. Therefore, as the number of concurrent threads increases, the chance to have smaller threads might also increase, thus adding extra overhead for threads' management.

Figures 10 and 11 show that the proposed parallel multithreaded Quicksort algorithm achieves higher speedup using four concurrent threads than using two or eight concurrent threads for all problem sizes (from 10 MB to 80 MB), where the input sequence is presented as local and remote distributions because the proposed parallel algorithm has lower execution time using four concurrent threads than two or eight

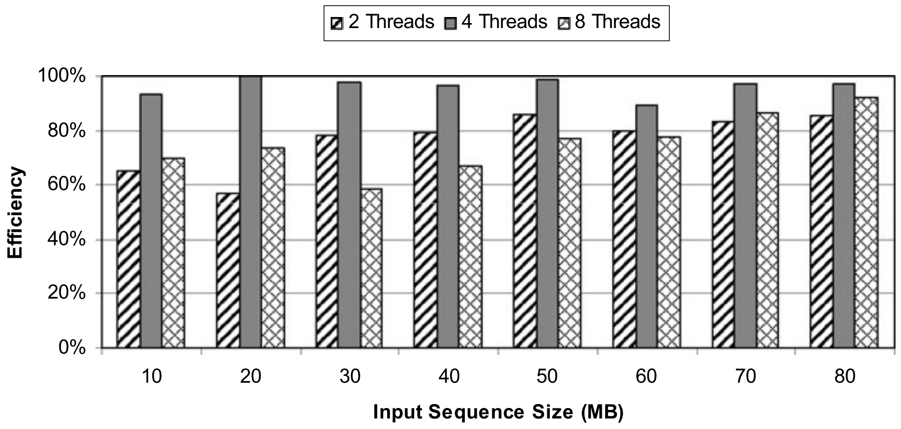


Fig. 12 Efficiency of parallel multithreaded Quicksort algorithm for random distribution

concurrent threads. The reason behind this is that Local distribution presents input sequence such that smaller keys are at the front of the sequence and larger ones are at the back of the sequence, whereas remote distribution presents input sequence as opposite of local distribution, which leads for both distributions to almost equal sized threads since both distributions present the input sequence with a wide range and very few duplicates. However, for both distributions using eight concurrent threads in the proposed parallel multithreaded Quicksort algorithm requires slightly less execution time than the sequential Quicksort algorithm in some problem sizes. For example, when the problem size is 30 MB, the speedup of the proposed algorithm using eight concurrent threads is 1.48 and 1.73 for local and remote distributions respectively, because the eight concurrent threads add more overhead for creating and terminating them, in addition to threads management.

In summary, we conclude from Figs. 7–11 that the speedup of the proposed parallel multithreaded Quicksort algorithm is affected by the distribution type of the input data sequence (random, zero, Gaussian, local, and remote distributions), input data sequence size, in addition to the number of threads are used. As a result, we conclude that the four-thread scenario presents best speedup for almost all problem sizes and for all distribution types except for Gaussian distribution, where the two-thread scenario presents better speedup.

7.2 Efficiency

This section presents the efficiency of the proposed parallel multithreaded Quicksort algorithm. The efficiency evaluation metric is defined as the fraction of time in which a Simultaneous Multithreaded (SMT) hardware thread is engaged beneficially. Thus, the efficiency is expressed as the ratio of the speedup to the number of SMT hardware threads, as shown in Eq. (2) (Sect. 5.3) [8, 30].

Figures 12–16 show the efficiency of the parallel multithreaded Quicksort algorithm according to Eq. (2) (Sect. 5.3) using two, four, and eight concurrent software-defined threads on random, zero, Gaussian, local, and remote distributions, where the problem size (input sequence size) ranges from 10 MB to 80 MB.

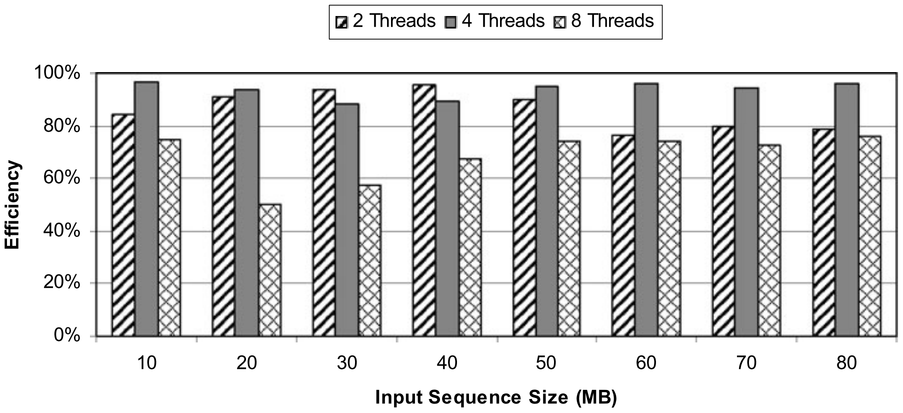


Fig. 13 Efficiency of parallel multithreaded Quicksort algorithm for zero distribution

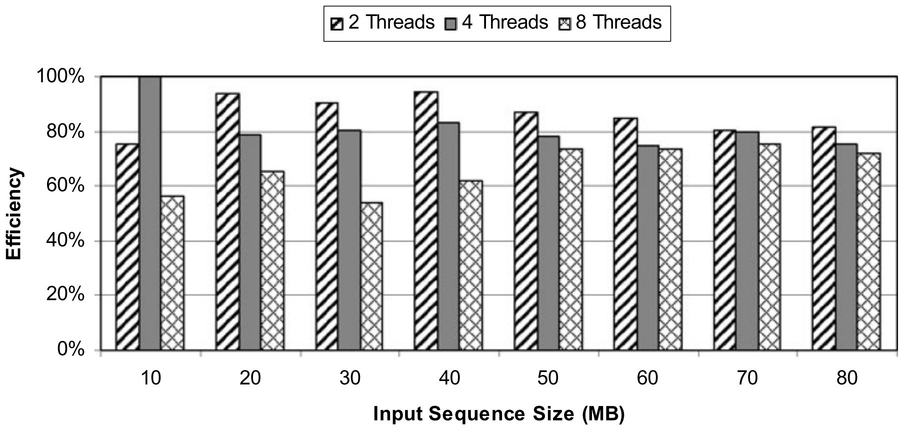


Fig. 14 Efficiency of parallel multithreaded Quicksort algorithm for Gaussian distribution

In Figs. 12, 15, and 16, the parallel multithreaded Quicksort algorithm for random, local, and remote distributions, respectively, using four concurrent software-defined threads, achieves better efficiency than using two and eight concurrent software-defined threads for all problem sizes. The reason behind this is that the speedup for these distributions using four concurrent threads is better than using two and eight concurrent threads, as shown in Figs. 7, 10, and 11. In particular, the efficiency of the parallel multithreaded Quicksort algorithm using four concurrent software-defined threads for Random distribution gets near to 97 % for large input sizes (70 MB and 80 MB), as shown in Fig. 12.

In Fig. 12, the efficiency of the parallel multithreaded Quicksort algorithm for random distribution using eight software-defined threads achieves better efficiency than using two software-defined threads for large input sizes (70 MB and 80 MB) due to the fact that using eight concurrent threads for large problem sizes is sufficient to achieve higher speedup and, therefore, better efficiency.

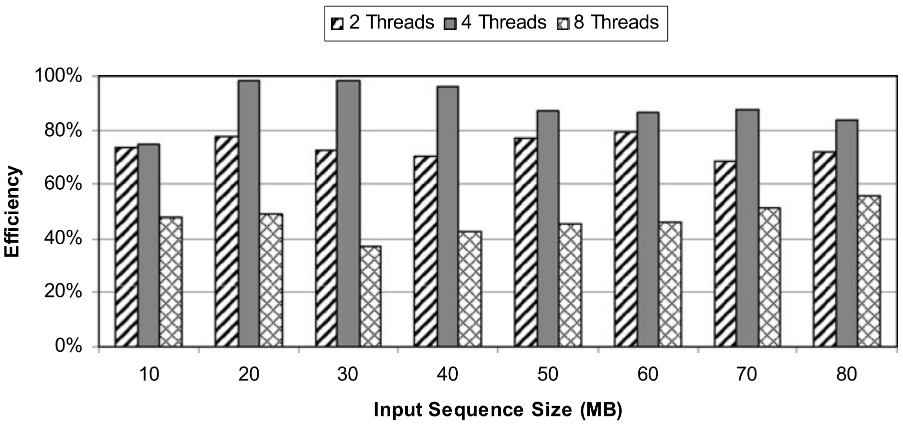


Fig. 15 Efficiency of parallel multithreaded Quicksort algorithm for local distribution

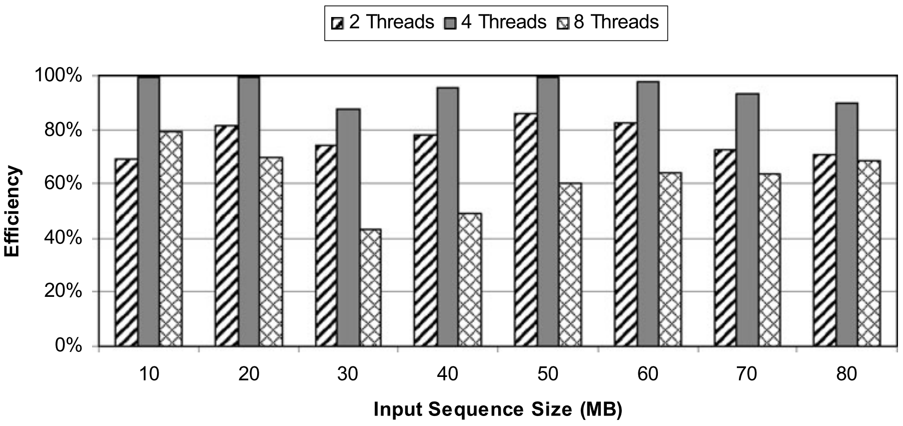


Fig. 16 Efficiency of parallel multithreaded Quicksort algorithm for remote distribution

Moreover, in Figs. 13, 14, 15, 16, as the number of concurrent software-defined threads increases to more than four, the efficiency drops due to the competition among concurrent software-defined threads for using system recourses; in other words, using two or four concurrent threads gives higher efficiency than using eight concurrent threads for most problem sizes of zero, Gaussian, local, and remote distributions.

7.3 Summary of experimental results

This section summarizes the results of the experimental runs, by comparing the performance of the two algorithms: the proposed parallel multithreaded Quicksort algorithm (using two, four, and eight concurrent software-defined threads) and the sequential Quicksort algorithm. The performance of the two algorithms is compared using the following five distributions: random, zero, Gaussian, local, and remote distribu-

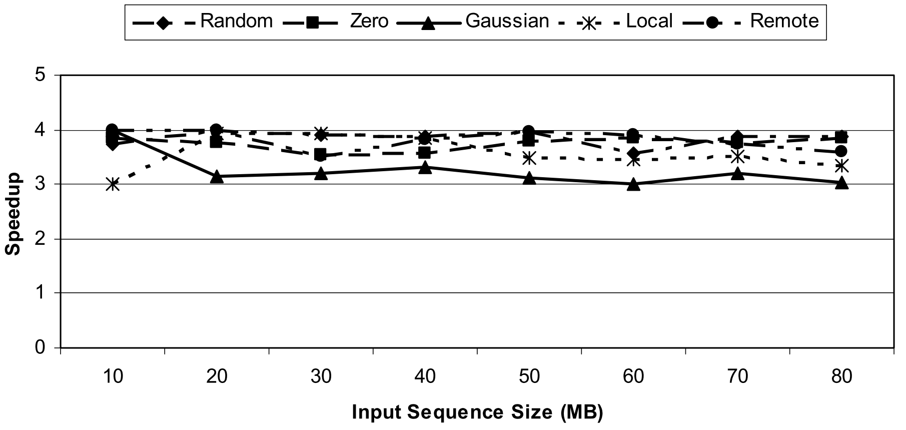


Fig. 17 Speedup of parallel multithreaded Quicksort algorithm using four threads

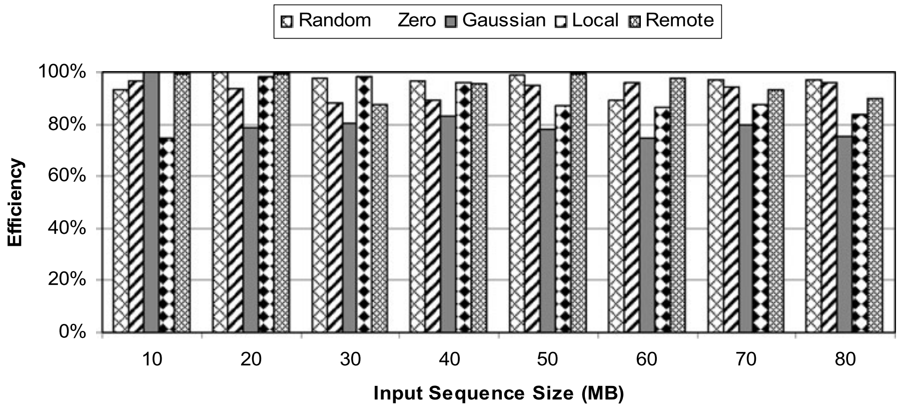


Fig. 18 Efficiency of parallel multithreaded Quicksort algorithm using four threads

tions, where each distribution presents the input sequence for different sizes ranges from 10 MB to 80 MB.

Figure 17 shows that the performance in terms of speedup achieved by the parallel multithreaded Quicksort using four concurrent software-defined threads is better than the sequential Quicksort algorithm at least by 2.99 times and at most by 3.99 times for random, zero, Gaussian, local, and remote distributions and for input sequence sizes ranging from 10 MB to 80 MB. The reason behind this increased speedup is that the proposed parallel multithreaded Quicksort algorithm requires less execution time than the sequential Quicksort algorithm because the proposed parallel algorithm takes advantage of the multithreaded architecture, which enhances instruction throughput by issuing multiple instructions from multiple threads within one clock cycle, whereas the sequential algorithm does not take advantage of this feature.

Figure 18 shows the performance in terms of efficiency of the parallel multithreaded Quicksort using four concurrent software-defined threads, which ranges be-

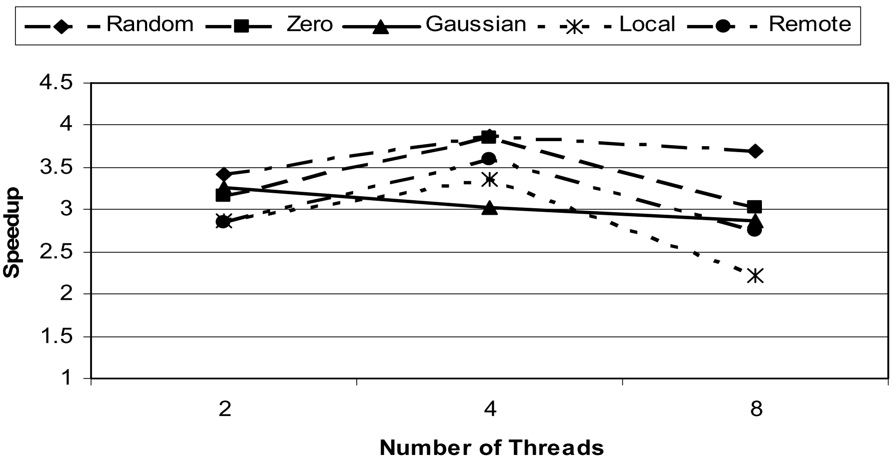


Fig. 19 Speedup of parallel multithreaded Quicksort algorithm for 80 MB input size

tween 74.8 % and 99.7 % for random, zero, Gaussian, local, and remote distributions and for input sequence sizes ranges from 10 MB to 80 MB. The reason behind this percentage of efficiency (74.8 % to 99.7 %) is due to the fact that more communication and threads management is needed between these four concurrent threads, which lead to reduced CPU utilization.

Figure 19 reveals the performance in terms of speedup for the proposed parallel multithreaded Quicksort algorithm using two, four, and eight concurrent software-defined threads for 80 MB input sequence size and for random, zero, Gaussian, local, and remote distributions. As shown in this figure, the parallel algorithm using four concurrent threads achieves higher speedup than using two or eight concurrent threads for large problem size (80 MB) and using random, zero, local, and remote distributions. However, for Gaussian distribution, as the number of concurrent threads increases, the speedup drops down since the Gaussian distribution presents the input sequence within a tight range and with a higher number of duplicates compared to other distributions, leading to a wide variation in threads' sizes as the number of concurrent threads increases, which explains the decreased speedup.

Figure 20 shows the efficiency of the proposed parallel multithreaded Quicksort algorithm using two, four, and eight concurrent software-defined threads for input sequence size of 80 MB of random, zero, Gaussian, local, and remote distributions. In this figure, the efficiency of the parallel multithreaded Quicksort algorithm using four threads is better than using two or eight threads for most distributions since in case of using eight threads more communication and threads management is required, which lowers the CPU utilization, and in case of using two threads, the possibility of getting unbalanced threads sizes is higher than using four threads, thus, reducing the efficiency.

As a result, the best number of concurrent threads, which is required by the proposed parallel multithreaded Quicksort algorithm, is four. This number is small enough to increase the performance in terms of speedup with relatively good CPU utilization, and it is large enough to achieve virtual parallelization. On the other hand, the

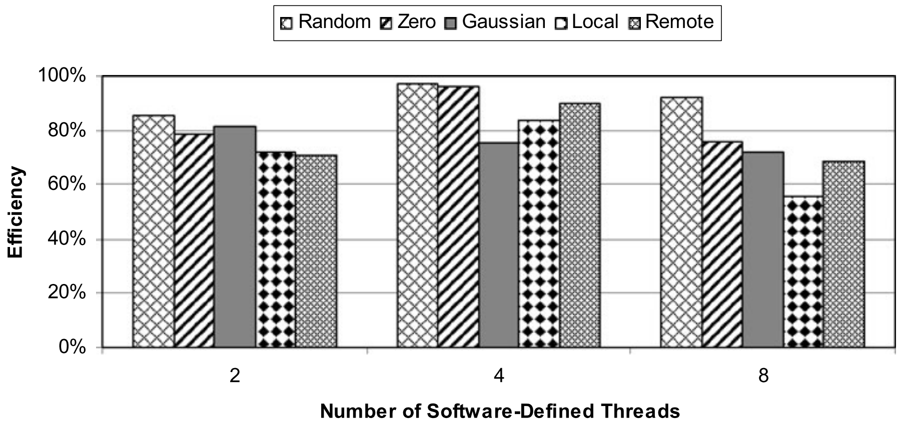


Fig. 20 Efficiency of parallel multithreaded Quicksort algorithm for 80 MB input size

analytical results in Sect. 5 show that as the number of concurrent threads increases in the proposed parallel multithreaded Quicksort algorithm, the analytical execution time decreases and the speedup increases, whereas in the experimental results this is not always valid, since the execution time here includes the time that is required for creating and terminating concurrent software-defined threads in addition to the time of serializing the algorithm and synchronizing shared data, which slows down the proposed algorithm as the number of concurrent threads increases to more than four. Therefore, comparing the analytical with the experimental results will not be applicable, and will result in a wide variation, especially as the number of concurrent threads increases.

8 Conclusions

This paper proposed a parallel multithreaded Quicksort algorithm, in which virtual parallelization is achieved using POSIX threads (Pthreads) library. The proposed algorithm employs an array data structure for storing the input data sequence. Five distributions (random, zero, Gaussian, local, and remote distributions) are used for generating the input data sequences of various sizes (from 10 MB to 80 MB). These distributions are used by the proposed parallel multithreaded Quicksort and the sequential Quicksort algorithms throughout the experimental runs.

Both the parallel multithreaded Quicksort and the sequential Quicksort algorithms have been evaluated analytically and experimentally in terms of the following performance metrics: time complexity, speedup, and efficiency. It has been revealed by the obtained results that the proposed parallel multithreaded Quicksort algorithm outperforms the sequential Quicksort algorithm both; analytically and experimentally. For example, the time complexity of the parallel multithreaded Quicksort is $\Theta(n/t \log_2 n/t)$, whereas the time complexity for the sequential Quicksort is $\Theta(n \log_2 n)$, where n is the size of the original input sequence and t is the number of software-defined threads. Moreover, the experimental results show that sorting various dataset sizes (from 10 MB to 80 MB) of all five distributions using the proposed parallel multithreaded Quicksort algorithm with four concurrent software-defined threads achieves from 2.99 to 3.99 times better speedup than the sequential Quicksort algorithm. This is clarified by the fact that the proposed parallel multithreaded Quicksort algorithm invests the advantageous features of all hardware resources of the Dual-Core microprocessors with SMT architecture support.

Acknowledgements The author would like to express his deep gratitude to the anonymous referees for their valuable comments and helpful suggestions, which improved the paper.

References

1. Knuth DE (1998) The art of computer programming. Sorting and searching, vol 3, 2nd edn. Addison-Wesley, Reading
2. Knuth DE (1970) Von Neumann's first computer program. Comput Surv 2:247–260
3. Hoare CAR (1962) Quicksort Comput J 5(1):10–15
4. JaJa J (2000) A perspective on Quicksort. Comput Sci Eng 2(1):43–49
5. Cormen T, Leiserson C, Rivest R, Stein C (2001) Introduction to algorithms, 2nd edn. The MIT Press, Cambridge
6. Jelenković L, Omrčen-Čeko G (1997) Experiments with multithreading in parallel computing. In: Proceedings of the 19th international conference on information technology interfaces (ITI'97), computer and intelligent systems, Croatia
7. Sun Microsystems (2008) Multithreaded programming guide. http://docs.oracle.com/cd/E19253-01/816-5137/816-5137.pdf. Accessed 06 August 2012
8. Grama A, Gupta A, Karypis G, Kumar V (2003) Introduction to parallel computing, 2nd edn. Addison-Wesley, Reading (an imprint of Pearson Education Limited)
9. Ungerer T, Robic B, Silc J (2002) Multithreaded processors. Comput J 45:320–348
10. Zang C, Imai S, Frank S, Kimura S (2008) Issue mechanism for embedded simultaneous multithreading processor. IEICE Trans Fundam Electron Commun Comput Sci E91-A(4):1092–1100
11. Jain R (1991) The art of computer systems performance analysis: techniques for experimental design, measurement, simulation, and modeling. Wiley, New York
12. Kumari A, Chakraborty S (2007) Software complexity: a statistical case study through insertion sort. Appl Math Comput 190:40–50
13. Walsh TR (1984) How evenly should one divide to conquer quickly? Inf Process Lett 19(4):203–208
14. Al-Darwish N (2005) Formulation and analysis of in-place MSD radix sort algorithms. J Inf Sci 31(6):467–481
15. Chamberlain B, Callahan D, Zima H (2007) Parallel programmability and the Chapel language. Int J High Perform Comput Appl 21(3):291–312
16. Bitton D, DeWitt D, Hsiao D, Menom J (1984) A taxonomy of parallel sorting. Comput Surv 16(3):287–318
17. Sanders P, Hansch T (1997) On the efficient implementation of massively parallel Quicksort. In: 4th international symposium on solving irregularly structured problems in parallel. Lecture notes in computer science, vol 1253. Springer, Berlin, pp 13–24
18. Brest J, Vreže A, Žumer V (2000) A sorting algorithm on PC cluster. In: Proceedings of the 2000 ACM symposium on applied computing, Como, Italy, vol 2, pp 710–715
19. Lan Y, Mohamed MA (1992) Parallel Quicksort in hypercubes. In: Proceedings of the 1992 ACM/SIGAPP symposium on applied computing: technological challenges of the 1990's, Kansas City, Missouri, pp 740–746
20. Tsigas P, Zhang Y (2003) A simple, fast parallel implementation of Quicksort and its performance evaluation on SUN enterprise 10000. In: Eleventh Euromicro conference on parallel, distributed and network-based processing, Genova, Italy
21. Heidelberger P, Norton A, Robinson JT (1990) Parallel Quicksort using fetch-and-add. IEEE Trans Comput 39(1):133–138
22. Tikhonova A, Tanase G, Tkachyshyn O, Amato N, Rauchwerge L (2005) Parallel algorithms in STAPL: sorting and the selection problem. Technical Report TR05-005 in Parasol Lab, Department of Computer Science, Texas A&M University

23. Nagaraja S, Pan Y, Badii M (2001) A parallel merging algorithm and its implementation with Java threads. In: Proceeding of MASPLAS'01 the mid-Atlantic student workshop on programming languages and systems. IBM Watson Research Centre, Armonk
24. Johnson E (1998) Support for parallel generic programming. Dissertation, Indiana University
25. Berlin J, Tanase G, Bianco M, Rauchwerger L, Amato NM (2007) Sample sort using the standard template adaptive parallel library. Technical Report TR07-002, Parasol Lab, Dept of Computer Science, Texas A&M University, College Station, USA
26. Garcia P, Korth HF (2005) Multithreaded architectures and the sort benchmark. In: Proceedings of the first international workshop on data management on new hardware (DaMoN '05), Baltimore, Maryland
27. Wang D, Zhang X, Men T, Wang M, Qin H (2012) An implementation of sorting algorithm based on Java multi-thread technology. In: Proceedings of 2012 international conference on computer science and electronics engineering (ICCSEE 2012), vol 1, pp 629–632
28. Lin H, Li C, Wang Q, Zhao Y, Pan N, Zhuang X, Shao L (2010) Automated tuning in parallel sorting on multi-core architectures. In: Lecture notes in computer science, LNCS, vol 6271. Springer, Berlin, pp 14–25
29. Rashid L, Hassanein WM, Hammad MA (2010) Analyzing and enhancing the parallel sort operation on multithreaded architectures. J Supercomput 53(2):293–312
30. Wilkenson B (1996) Computer architecture design and performance, 2nd edn. Prentice Hall, London
31. Deitel PJ, Deitel HM (2007) C++ how to program, 6th edn. Prentice Hall, New York
32. Hennesy J, Patterson D (2006) Computer architecture: a quantitative approach, 4th edn. Morgan Kaufmann, San Mateo