Jordanian Journal of Computers and Information Technology (JJCIT), Vol. 10, No. 03, September 2024.

PARALLEL BUCKET-SORT ALGORITHM ON OPTICAL CHAINED-CUBIC TREE INTERCONNECTION NETWORK

Basel A. Mahafzah

(Received: 15-Mar.-2024, Revised: 12-May-2024 and 19-Jun.-2024, Accepted: 23-Jun.-2024)

ABSTRACT

The performance of sorting algorithms has a great impact on many computationally intensive applications. Researchers worked on parallelizing many sorting algorithms on various interconnection networks to improve their sequential counterpart performance. One of these interconnection networks is the optical chained-cubic tree (OCCT). In this paper, a parallel bucket sort (PBS) algorithm is presented and applied to the OCCT interconnection network. This PBS algorithm is evaluated analytically and by simulation in terms of various performance metrics including parallel runtime, computation time, communication time, concatenation time, speedup and efficiency, for a different number of processors, dataset sizes and data distributions including random and descending distributions. Simulation results show that the highest obtained speedup is approximately 912x on OCCT using 1020 processors, which means that the parallel runtime of the PBS on 1020 processors is 912 times faster than the sequential runtime of BS on a single processor.

KEYWORDS

Bucket sort, Parallel sorting algorithm, Interconnection network, Opto-electronic architecture.

1. INTRODUCTION

Many researchers concentrate their efforts on minimizing the run time needed to perform sorting algorithms efficiently on various architectures [1]-[11]. Also, several comparative sorting algorithms have been presented and analyzed in detail to show their advantages and disadvantages [12]-[17]. In general, sorting algorithms are among the most studied algorithms and are important in the computer science field, since sorting is one of the most essential operations used in many problems and applications, such as integer problems, databases, search engines, text data, image processing and information retrieval [18]-[24].

One of the well-known sorting algorithms is the bucket sort (BS) [14][19][25], which is a good choice for sorting elements with values uniformly distributed over an interval. In the BS algorithm, the interval is divided into consecutive non-overlapping sub-intervals called buckets to sort the input, where each element is placed in an appropriate bucket based on the element's value and each bucket is sorted using any sorting algorithm, such as quicksort, merge sort, count sort, insertion sort, …etc. Then, buckets are concatenated to form the sorted list [19], [25]-[26].

Practically, sorting a large number of elements using a sequential bucket-sort algorithm requires a high runtime. So, one way to improve the runtime of the bucket-sort algorithm is to run it on parallel or distributed architectures [27]-[30]. Examples of these architectures are optical chained-cubic tree (OCCT) [31] and optical transpose interconnection system (OTIS) and its variants, such as OTIS-Mesh, OTIS-Hypercube and OTIS Hyper Hexa-Cell (OHHC) [32]-[34].

The OCCT interconnection network is based on the chained-cubic tree (CCT) which is constructed from a tree and hypercubes in addition to electronic and optical links [31][35]. The electronic links connect processors within tree levels and hypercubes, whereas optical links are added on a certain level of the tree to reduce the distance between processors. In general, optical links can carry data with less power consumption and a high data rate compared to electronic links [36]-[37]. OCCT shows efficient topological properties including low diameter, high maximum node degree and high bisection width [31]-[32]. Also, the CCT was evaluated by implementing a parallel bitonic sort algorithm on this interconnection network, where it showed a great performance [3]. The efficient properties of

B. A. Mahafzah is with Department of Computer Science, King Abdullah II School of Information Technology, The University of Jordan and with Department of Computer Science, King Hussein School of Computing Sciences, Princess Sumaya University for Technology, Amman, Jordan. Emails: b.mahafzah@ju.edu.jo and b.mahafzah@psut.edu.jo

OCCT and the previous work on CCT motivate us to implement a parallel bucket-sort (PBS) algorithm on OCCT taking advantage of the OCCT-structure properties to get an efficient parallel sorting algorithm.

The main contribution of this paper is implementing an efficient PBS algorithm on the OCCT interconnection network and evaluating the PBS algorithm analytically and by simulation in terms of parallel runtime, computation time, communication time, concatenation time, speedup and efficiency, for different numbers of processors and dataset sizes and two types of data distributions; namely, random and descending distributions.

2. RELATED WORK

Several research works have been conducted on the parallel bucket-sort algorithm using various architectures and platforms. For example, in [27], the author showed how to convert a sequential bucket-sort algorithm into a parallel algorithm, which has been implemented and executed using OpenMP API. Experimental results showed that this parallel version is a scalable algorithm, where its performance can be improved as the number of cores is increased. Also, in [28], the authors implemented a parallel bucket-sort algorithm for a many-core architecture of graphics processing units (GPUs) based on convex optimization. Moreover, in [29], the author used threads and GPU programming to optimize the bucket-sort algorithm. Experimental results showed that for a small number of elements, it is better to carry out the sorting in a single thread. Also, using the bucket-sort algorithm, the bottleneck of GPU and CPU is shown in this research work clearly.

Additionally, several research works have been conducted on opto-electronic architectures. In [32], the authors presented a detailed review of nine optoelectronic architectures in terms of their topological structure and topological properties including the OCCT. These opto-electronic architectures are interconnection networks that use electronic and optical links to connect processors. All these architectures except the OCCT are based on OTIS. These architectures are evaluated in terms of various topological properties; namely, size, diameter, cost, bisection width, maximum node degree and minimum node degree and Hamiltonian path and cycle. Among these architectures, the OCCT showed great performance in terms of diameter, maximum node degree and bisection width [31]-[32]. However, up to this time and up to our knowledge, none of the parallel bucket-sorting algorithms has been applied to opto-electronic architectures, which motivates us to implement an efficient parallel bucket-sort algorithm on the OCCT opto-electronic architecture and evaluate it analytically and by simulation in terms of various performance metrics.

3. OCCT INTERCONNECTION NETWORK

The structure of the OCCT interconnection network [31] is based on CCT [35], where the CCT interconnection network is based on a binary tree and hypercubes. The height h of OCCT is floor(log G) and each hypercube in OCCT is a group G of 2^d processors of dimension d, in addition to a specific level lv that is chosen according to the height of the tree where the optical links are added in a cascading manner between distant hypercubes at that level. Thus, OCCT is referred to as OCCT (h, d, lv). An OCCT can be a full or complete binary tree network based on the status of its last level. Figure 1 shows a full OCCT(3, 2, 2) [31], where 3 is the height of the tree, 2 is the dimension of each hypercube group and 2 is the lv level number wherein at that level, the optical links are added (thick black lines). Figure 2 shows the lv level where lv = 2 in details of the OCCT(3, 2, 2) [31]. Also, as shown in Figure 2, the label of each processor is unique and contains a pair of numbers (G_i, p_j). For example, processor (3, 2) means processor number 2 in group number 3. However, more details regarding the labeling of groups and processors in OCCT can be found in [31].

The lv value depends on two factors; the type of binary tree whether it is full or complete. If the tree is a full binary tree, then the level lv = ceiling(h/2) and if the tree is a complete binary tree, then the level lv depends on the tree height type; whether odd or even and the number of groups in the last level. Thus, there are three cases; the first case is if the tree height h is even, then lv= h/2. In the second case, if the tree height h is odd and the number of groups in the last level is less than $(2^{(h-1)/2})\times3+1$, then lv = (h-1)/2. In the third case, if the tree height h is odd and the number of groups in the last level is greater than or equal to $(2^{(h-1)/2})\times3+1$, then lv = (h+1)/2 [31]. However, more details regarding implementing

the structures of OCCT and CCT can be found in [31][35].

The size is the number of processors in the OCCT interconnection network. The size of the OCCT(*h*, *d*, *lv*) is $G \times 2^d$, where G is the number of hypercube groups in the tree and 2^d is the number of processors in each hypercube of dimension *d* [31]-[32].

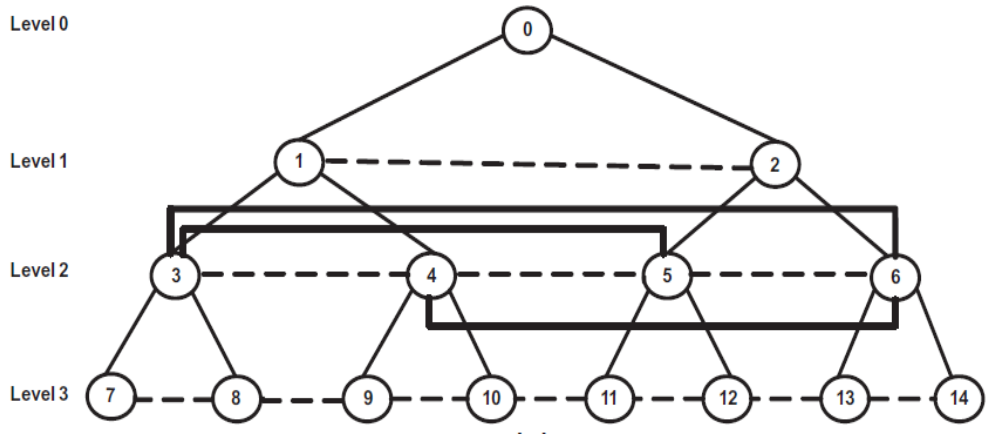

Figure 1. An OCCT(3, 2, 2).

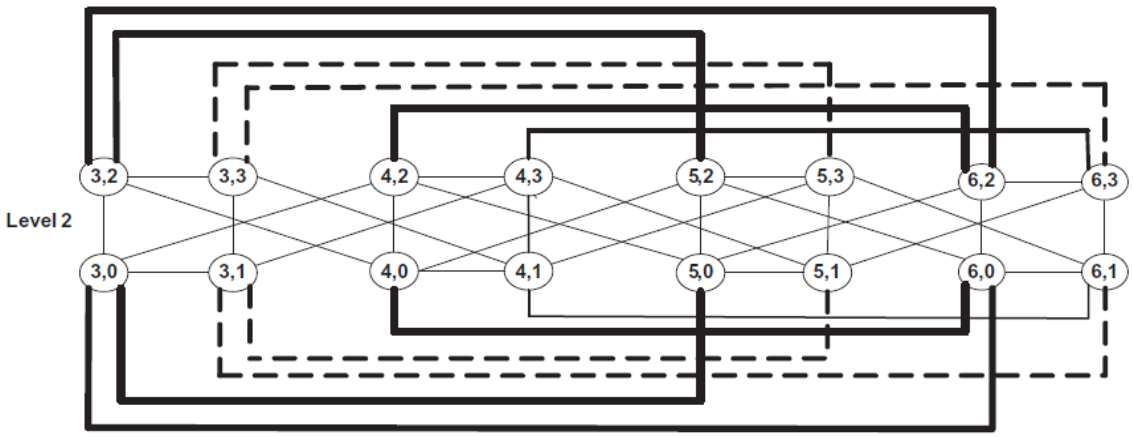

Figure 2. Level two of OCCT(3, 2, 2).

4. SEQUENTIAL BUCKET SORT ALGORITHM

The sequential BS algorithm is a well-known sorting algorithm. It sorts *n* elements the values of which are uniformly distributed over an interval [1, *n*], where this interval is divided into *b* equal-sized sub-intervals called buckets. That is, the BS algorithm uses buckets of the same size and each element is placed in the appropriate bucket according to its value. As a result, each bucket will have almost the same number of elements which is approximately *n*/*b*. Then, the BS algorithm uses an efficient and easy-to-implement sorting algorithm, such as quicksort [14][38], to sort the elements in each bucket. Finally, these sorted buckets are concatenated in the appropriate order to form the final sorted list. The run time of this sequential BS algorithm is $\Theta(n \log (n/b))$, where this low time complexity is due to the assumption that the *n* elements to be sorted are uniformly distributed over the interval [1, *n*].

The sequential BS algorithm's steps are shown in Algorithm 1, where the input parameters are defined in Table 1 and the size of a bucket (*s*) and its sub-interval are computed using Equations (1)-(3).

$$s = (max - min + 1) / b \tag{1}$$

$$Bucket_{start}(B) = min + B \times s, \qquad \text{where } B = 0, 1, 2, \ldots, b-1 \tag{2}$$

$$Bucket_{end}(B) = Bucket_{start}(B) + s - 1 \tag{3}$$

Algorithm 1	Sequential BS algorithm's steps.
Input	Initially, a single processor has the input of n elements, which are distributed uniformly over the interval [1, n].
Output	Sorted n elements in ascending order.
1	Sizes and sub-intervals of the buckets are computed by a single processor using Equations (1)-(3).
2	Using the created buckets in Step 1, each element is placed in its appropriate bucket according to its value and the bucket's sub-interval. That is the input array is scanned from left to right and each element is moved to its appropriate bucket.
3	After all elements are placed in their buckets, a call to a quicksort algorithm is executed to sort each bucket.
4	After each bucket is sorted, buckets are concatenated in the required order to produce the final sorted list.

Table 1. Input parameters of sequential BS algorithm.

Parameters	**Description**
n	Number of elements to be sorted (input size)
b	Number of buckets
B	Bucket number
max	Maximum element value in the input (last index of input array equals n)
min	Minimum element value in the input (first index of input array equals 1)
s	Size of the bucket, which is the number of elements in the bucket, where all buckets have almost the same size
$Bucket_{start}(B)$	First element in the sub-interval of bucket number B
$Bucket_{end}(B)$	Last element in the sub-interval of bucket number B

5. PARALLEL BUCKET SORT ALGORITHM ON OCCT

In this section, we introduce the PBS algorithm on the OCCT interconnection network, as shown in Algorithm 2. In the proposed PBS algorithm, we assume a list of n elements uniformly distributed over the interval [1, n] to be sorted using a number of buckets (b), where these b buckets are almost of the same size. Each bucket is assigned to a single processor in OCCT; that is, $p = b$, where p is the number of processors. Also, we assume that the bucket size is s, where $s = n/b$. Additionally, we assume initially that each processor has a complete copy of the input n elements.

The PBS algorithm consists of two phases: the computation phase and the communication and concatenation phase, where, in Algorithm 2, steps 1–4 present the computation phase and steps 5 and 6 present the communication and concatenation phase.

Algorithm 2 works as follows: In step 1, in parallel, each processor in OCCT calculates the size of the bucket and the sub-intervals of the buckets using Equations (1)-(3).

In step 2, each processor is assigned a bucket according to a global group sequential ordering, where for example every four buckets are assigned to one group (i.e., one hypercube of four processors) by sequential order. For example, the first four buckets numbered 0 to 3 are assigned to group 0, while the second four buckets which are numbered 4 to 7 are assigned to group 1, …and so on.

In step 3, in parallel, each processor scans the entire n input elements and determines the elements that belong to its bucket according to both its bucket sub-interval and the elements' values which must be within the bucket's sub-interval. As a result, each bucket will have almost the same number of elements, which is approximately n/b.

In step 4, in parallel, each processor applies the sequential quicksort algorithm to sort the elements of its bucket which are approximately n/b elements. The quicksort algorithm is an in-place sorting

algorithm that does not need additional memory space to sort the n/b elements and it is easy to implement using the divide-and-conquer approach. Also, it is efficient in terms of time complexity, which is $O(n/b \log_2 n/b)$, since it sorts n/b elements in each processor in parallel and independently.

In step 5, the processors communicate with each other to combine their sorted buckets at a single processor located at the left-most hypercube group of the *lv* level in OCCT, where the optical links exist to take advantage of these links. Thus, the communication pattern of the proposed PBS algorithm takes place in three major stages as follows:

1. Upper and lower tree communication stage: In this stage, processors at the *lv* level gather results from processors at upper and lower levels in the tree at the same time in parallel.
2. Hypercube-communication stage: In this stage, each group of processors from a hypercube at the *lv* level gathers their results at processor number 0 of that hypercube.
3. Optical-communication stage: In this stage, processor number 0 of the left-most hypercube at the *lv* level gathers results from other processors number 0 of other hypercubes of the same *lv* level.

Finally, in step 6, at this left-most group, the single processor that received all sorted buckets concatenates them as one list of elements according to the buckets' number from lowest to highest which presents the buckets' sub-intervals from lowest to highest to have the n elements sorted in ascending order.

Algorithm 2	PBS algorithm's steps on OCCT.
Input	Initially, each processor p_i has a complete copy of the input n elements which are uniformly distributed over the interval [1, n].
Output	Sorted n elements in ascending order.
1	In parallel, the size of the bucket and the sub-intervals of buckets are computed by each processor in OCCT using Equations (1)-(3).
2	Each processor p_i is assigned a bucket b_i according to a global ordering, where for example every four buckets are assigned to one group (i.e., one hypercube of four processors) by sequential order.
3	In parallel, each processor p_i scans the input n elements and determines the elements that belong to its bucket b_i according to its sub-interval and the elements' values.
4	In parallel, each processor p_i applies the sequential quicksort algorithm to sort the elements of its bucket b_i.
5	Processors communicate with each other to combine and concatenate their results at a single processor located at the left-most group of the *lv* level in OCCT.
6	At this single processor, the buckets are concatenated according to their number and sub-intervals from lowest to highest to have the n elements sorted.

The proposed PBS algorithm is modified slightly and customized to be applied to OCCT architecture efficiently. The modification is made in the computation phase by having the buckets almost equal in size and $p=b$ to distribute buckets on processors evenly (i.e., load-balanced) to have all processors finish approximately at the same time, which leads to better performance. The customization is made in the communication and concatenation phase, where the buckets are distributed and gathered in less communication time using the electronic and optical links; that is reaching all processors using the shortest path (the diameter of OCCT).

An example of the PBS algorithm's steps is shown in Figure 3. In this example, we do not show the communication phase in detail, for simplicity. Also, in this example, we assume that $n = 16$ and $b = p = 4$. Therefore, the size of each bucket is 4. Initially, each processor has a copy of the input list which contains uniformly distributed 16 elements over the interval [1, 16], as shown in Figure 3(a). Then the bucket's size and sub-interval of each bucket are computed using Eqs. (1–3) and each processor determines its elements according to its bucket's sub-interval, as shown in Figure 3(b). Then, each processor sorts its bucket, as shown in Figure 3(c) and finally, the processors communicate to gather

the buckets at processor(0), where it concatenates the buckets sequentially according to their number and sub-intervals from lowest to highest to have a sorted list in ascending order, as shown in Figure 3(d).

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(0)	5	3	1	2	6	8	10	11	15	16	14	13	12	7	9	4
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(1)	5	3	1	2	6	8	10	11	15	16	14	13	12	7	9	4
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(2)	5	3	1	2	6	8	10	11	15	16	14	13	12	7	9	4
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(3)	5	3	1	2	6	8	10	11	15	16	14	13	12	7	9	4

(a) Initially, each processor has a copy of the input list which contains uniformly distributed 16 elements over the interval [1, 16].

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(0) Subinterval [1, 4]	5	**3**	**1**	**2**	6	8	10	11	15	16	14	13	12	7	9	**4**
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(1) Subinterval [5, 8]	**5**	3	1	2	**6**	**8**	10	11	15	16	14	13	12	**7**	9	4
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(2) Subinterval [9, 12]	5	3	1	2	6	8	**10**	**11**	15	16	14	13	**12**	7	**9**	4
Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Processor(3) Subinterval [13, 16]	5	3	1	2	6	8	10	11	**15**	**16**	**14**	**13**	12	7	9	4

(b) Bucket sizes and the sub-intervals of buckets are computed and each bucket is assigned to a processor.

Index	1	2	3	4		Index	1	2	3	4
Processor(0) Unsorted Bucket(0)	3	1	2	4	➔	**Processor(0) Sorted Bucket(0)**	1	2	3	4
Index	1	2	3	4		Index	1	2	3	4
Processor(1) Unsorted Bucket(1)	5	6	8	7	➔	**Processor(1) Sorted Bucket(1)**	5	6	7	8
Index	1	2	3	4		Index	1	2	3	4
Processor(2) Unsorted Bucket(2)	10	11	12	9	➔	**Processor(2) Sorted Bucket(2)**	9	10	11	12
Index	1	2	3	4		Index	1	2	3	4
Processor(3) Unsorted Bucket(3)	15	16	14	13	➔	**Processor(3) Sorted Bucket(3)**	13	14	15	16

(c) Each processor sorts its unsorted bucket.

Buckets subintervals	**1**	2	3	**4**	**5**	6	7	**8**	**9**	10	11	**12**	**13**	14	15	**16**
Concatenated sorted buckets at Processor(0)	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	Bucket(0)				**Bucket(1)**				**Bucket(2)**				**Bucket(3)**			

(d) Processor(0) concatenates all received buckets to form a sorted list.

Figure 3. Example of the PBS algorithm's steps.

6. ANALYTICAL EVALUATION

In this section, the proposed PBS algorithm is evaluated analytically in terms of several performance metrics, including parallel runtime complexity, speedup and efficiency. The parallel runtime is the time that passes from the moment that a parallel computation starts to the moment at which the last processor finishes execution, where it includes the time that a parallel program spends in computation and communication [13]. Thus, the analytical evaluation of the PBS algorithm is presented for its computation phase first in sub-section 6.1 followed by its communication and concatenation phase in sub-section 6.2 and lastly, in sub-section 6.3, the mentioned performance metrics are presented.

6.1 Computation-phase Analysis

In this sub-section, the computation analysis of the proposed PBS algorithm (Algorithm 2) is presented. In the computation phase, according to Algorithm 2, four steps are shown as follows, where each step is followed by its expected sequential and parallel runtime complexity:

1. *Finding bucket size and sub-intervals of all buckets*: Sequential and parallel runtime complexity for finding bucket size is constant O(1) and for finding the sub-intervals of all buckets is O(b), since in BS and PBS algorithms, finding the size and the sub-intervals of buckets can be carried out using Equations (1)-(3).
2. *Assigning buckets*: Sequential and parallel runtime complexity is O(1), since, in the BS algorithm, all buckets are assigned to a single processor and in the PBS algorithm, each processor is assigned one bucket according to a global sequential ordering; specifically, buckets are assigned according to group number and processor number.
3. *Putting each element in its proper bucket*: Sequential and parallel runtime complexity is O(n), since, in BS and PBS algorithms, each processor passes over the n elements to find its bucket elements.
4. *Sorting buckets*: Sequential runtime is O($b \times s \log s$) as best and average cases, since we have b buckets to sort each of size s, assuming that each bucket has the same number of elements and using quicksort to sort each bucket sequentially. The quicksort best and average-case time complexity is O($n \log n$) to sort n elements, where these cases occur when the elements are random [14][38]. Since $n = b \times s$, then the sequential runtime is O($n \log s$). In the worst case, the time complexity of quicksort is O(n^2) to sort n elements, where this case occurs when the elements are already ascendingly or descendingly sorted [14]. Thus, the bucket sort worst-case time complexity would be O($b \times s^2$) = O($n \times s$). Whereas, the parallel runtime complexity is O($s \log s$), since in parallel each processor uses a quicksort algorithm to sort its s elements in the best and average cases and in the worst case, it would be O(s^2). Note that $s << n$, since $s = n / b$.

The sequential and parallel computation runtimes of the PBS algorithm differ only in the fourth step, which is the step of sorting buckets. Thus, based on the four computational steps, the sequential runtime (T_{seq}) of the BS algorithm as best and average cases and worst case is shown in Equations (4)-(5), respectively. The parallel computation runtime (T_{comp}) of the PBS algorithm as best and average cases and worst case is shown in Equations (6)-(7), respectively.

$$T_{seq} = O(1) + O(b) + O(1) + O(n) + O(n \log s) \approx O(n \log s) \quad (\textit{best \& average cases}) \quad (4)$$

$$T_{seq} = O(1) + O(b) + O(1) + O(n) + O(n \times s) \approx O(n \times s) \quad (\textit{worst case}) \quad (5)$$

$$T_{comp} = O(1) + O(b) + O(1) + O(n) + O(s \log s) \approx O(s \log s) \quad (\textit{best \& average cases}) \quad (6)$$

$$T_{comp} = O(1) + O(b) + O(1) + O(n) + O(s^2) \approx O(s^2) \quad (\textit{worst case}) \quad (7)$$

6.2 Communication and Concatenation Phase Analysis

In this sub-section, the communication and concatenation phase of the proposed PBS algorithm (Algorithm 2) is presented. The communication pattern of the proposed PBS algorithm has the following three stages: The upper and lower tree communication stage, the hypercube-communication stage and the optical-communication stage.

In general, the communication time equals the number of required steps multiplied by the message size, where a message may contain one sorted bucket or two concatenated sorted buckets or more, multiplied by the time to transmit a word of data on an electronic (T_{we}) or an optical link (T_{wo}).

For the first stage, the upper tree communication time is presented in Equation (8). The number of required steps for the upper tree communication equals the value of *lv*, because we have to pass *lv* levels from the root of the tree (level 0) to the optical links level (*lv*). In each step, we pass a bucket of size $s = (n / b)$. Also, in this upper communication, buckets are transmitted using only electronic links. So, the total communication time required for the upper tree is T_{up}, as shown in Equation (8).

$$T_{up} = lv \times s \times T_{we} \tag{8}$$

The tree height from the last level of the tree to the *lv* level is $(h - lv)$, which is equal to *lv* or $(lv–1)$ depending on the place of the optical links, as discussed in Section 3. Thus, the number of communication steps is the maximum between *lv* and $(lv - 1)$, which is *lv*. In the first step, the size of the transferred message is *s*, which is the size of a single bucket. In the second step, the size becomes $(2 \times s)$, while in the third step, the size is $(4 \times s)$, …and so on. At maximum, the size of the transferred message is $(2^{lv-1} \times s)$; that is at every step, the size of the transferred message doubles, which means that the number of transferred buckets doubles. Also, only the electronic links are used to transfer the buckets to the *lv* level. So, the total communication time required for the lower tree is T_{low}, as shown in Equation (9), where the size of the transferred buckets is $(\sum_{i=1}^{lv} 2^{i-1} \times s)$. Equation 9 can be simplified, as shown in Equation (10). Note that, in our simulation runs, the values of *lv* vary according to the size of OCCT, specifically from 2 to 4, which is a small value.

$$T_{low} = lv \times (\sum_{i=1}^{lv} 2^{i-1} \times s) \times T_{we} \tag{9}$$

$$T_{low} = lv \times ((2^{lv} - 1) \times s) \times T_{we} \tag{10}$$

Since the upper and the lower tree communication are carried out in parallel, then the communication time of this stage is T_{up-low}, which is the maximum time between the upper and the lower tree communication times, as shown in Equation (11). Thus, since the lower tree communication time is larger than the upper tree communication time because a larger message size is transferred, the communication time of this stage is dominated by the lower-communication time.

$$T_{up-low} = \max((lv \times s \times T_{we}), (lv \times ((2^{lv} - 1) \times s) \times T_{we})) \tag{11}$$

The number of communication steps in the hypercube is *d*, which is the dimension of the hypercube, in our case $d = 2$. Specifically, in the first step, the size of the message equals the results (concatenated sorted buckets) gathered from stage 1 in addition to one bucket that each processor in the hypercube originally has. However, the hypercube-communication time in this stage depends on the number of received buckets from the previous stage. So, the total hypercube-communication time (T_Q) is shown in Equation (12), where the size of the received buckets from the upper tree levels is $(lv \times s)$ and the size of the received buckets from the lower tree levels including the bucket that each processor in the hypercube originally had is $\left(2 \times \left(2^{lv} - 1\right) \times s + s\right)$ where it can be simplified as $(\left(2^{lv+1} - 1\right) \times s)$. Also, the communication links used in the hypercubes are electronic links.

$$T_Q = d \times ((lv \times s) + (\left(2^{lv+1} - 1\right) \times s)) \times T_{we} \tag{12}$$

In the last stage of communication, the optical communication stage, processor number 0 at the left-most hypercube of level *lv* gathers all results (concatenated sorted buckets) from its counterpart processors numbered 0 of other hypercubes in the same level *lv* using the optical links. This required at most two optical steps, since there are no adjacent nodes and groups connected using optical links in the *lv* level [31]. The size of the transferred concatenated sorted buckets is the size of the received buckets from stage 2 (hypercube-communication stage) to processor 0, in addition to its bucket. Equation 13 presents the total optical communication time of this stage, which is T_{op}.

$$T_{op} = 2 \times (2^d \times (((lv - 1) \times s) + (\left(2^{lv+1} - 1\right) \times s))) \times T_{wo} \tag{13}$$

The total communication time (T_{comm}) of the PBS algorithm on OCCT is the summation of the upper and the lower-communication time, hypercube communication time and optical communication time, as shown in Equation (14), which are presented in Equations (11)-(13), respectively.

$$T_{comm} = T_{up-low} + T_Q + T_{op} \tag{14}$$

During each stage of communication, sorted buckets are concatenated once they are received by a processor according to the group and processor numbers. So, the parallel runtime complexity of the

concatenation of the buckets is T_{pc}, as shown in Equation (15), which is the number of communication steps in each stage, where *lv* is the number of communication steps in the upper and the lower communication stage, *d* is the number of communication steps in the hypercube-communication stage and 2 is the number of communication steps in the optical-communication stage. Also, *lv* and *d* are small values, where in our case $d = 2$ and *lv* varies between 2 and 4 according to the OCCT size. However, the sequential runtime complexity of the concatenation of buckets is O(*b*), where *b* is the number of buckets.

$$T_{pc} = O(lv + d + 2) \tag{15}$$

6.3 Performance Metrics

In this sub-section, the performance metrics of the proposed PBS algorithm are presented, including the parallel runtime complexity, speedup and efficiency. The parallel runtime complexity (T_p) of the PBS algorithm on OCCT is the summation of the computation, communication and concatenation times, as shown in Equation (16), which are presented in Equations (6) (14) (15), respectively, as shown in Equation (17). However, the parallel runtime complexity of the PBS algorithm, which is presented in Equation (17) as the best and average cases, is dominated by the computation time for large *n*, which is the common case. The speedup (S_p) is defined as the sequential runtime divided by the parallel runtime of solving the same problem, as shown in Equation (18). Thus, the speedup of the PBS algorithm on OCCT is shown in Equation (19), where the sequential runtime complexity (T_{seq}) of the BS algorithm is shown as the best and average cases. Accordingly, Equation (19) shows the speedup as the best and average cases. The efficiency (E_f) is defined as the speedup divided by the number of used processors, as shown in Equation (20). Thus, the efficiency of the PBS algorithm on OCCT is shown in Equation (21) as the best and average cases.

$$T_p = T_{comp} + T_{comm} + T_{pc} \tag{16}$$

$$T_p = O(s \log s) + (T_{up-low} + T_Q + T_{op}) + O(lv + d + 2) \tag{17}$$

$$S_p = \frac{T_{seq}}{T_p} \tag{18}$$

$$S_p = \frac{O(n \log s)}{O(s \log s) + (T_{up-low} + T_Q + T_{op}) + O(lv + d + 2)} \tag{19}$$

$$E_f = \frac{S_p}{p} \tag{20}$$

$$E_f = \frac{O(n \log s)}{p \times (O(s \log s) + (T_{up-low} + T_Q + T_{op}) + O(lv + d + 2))} \tag{21}$$

7. SIMULATION ENVIRONMENT AND RESULTS

In this section, the simulation environment and results are presented and discussed. The simulation results are evaluated in terms of two performance metrics; namely, speedup and efficiency.

7.1 Simulation Environment

The OCCT interconnection network does not exist as a real-machine or real-computing environment. Therefore, the OCCT interconnection network and the algorithms are implemented using Java Virtual Threads, simulated as a distributed memory model. However, the simulation runs under a shared memory multi-core computer machine.

In this sub-section, the simulation environment is presented, including software and hardware specifications and input-data distributions. The simulation implementation is programmed using Java Virtual Threads, which offer lightweight and efficient concurrency management within the Java Virtual Machine, on a multi-core computer machine with the specifications provided in Table 2. The parameter settings of the PBS algorithm's conducted simulation runs are shown in Table 3. Also, Table 4 presents the required parameter settings to implement the OCCT interconnection network which are the type of OCCT whether it is full or complete, the height *h* of the tree, the number of processors (*p*), the number of groups (*G*) in OCCT, the number of groups at the last level of OCCT (G_L) and the values of the level *lv*, where these values are calculated according to the equations presented in Section 3.

Table 2. Hardware specifications of the computer machine used for simulation runs.

Processor	Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz 1.50 GHz (4 Cores and 8 Threads)
RAM	16 GB
Operating system	64-bit Windows 10 Pro

Table 3. Parameter settings of the PBS algorithm's conducted simulation runs.

Type of sorting	Ascending
Type of input elements	4-byte integer number
Input size (MB)	0.4, 1.2, 2, 2.8, 4, 12, 20, 40, 60, 100, 150, 200, 400
Type of OCCT	Full OCCT(*h*, 2, *lv*), two-dimensional hypercube Complete OCCT(*h*, 2, *lv*), two-dimensional hypercube
Number of processors	Full OCCT: 60, 124, 252, 508, 1020 Complete OCCT: 92, 188, 380, 764
Input-data distribution	R: Random D: Descending and continuous (reverse ordered)

Table 4. Parameter settings of the OCCT interconnection network.

OCCT Type	Height (*h*)	Number of Processors (*p*)	Number of Groups (*G*)	Number of Groups at Last Level (G_L)	Level (*lv*)
Full	3	60	15	8	2
Complete	4	92	23	8	2
Full	4	124	31	16	2
Complete	5	188	47	16	3
Full	5	252	63	32	3
Complete	6	380	95	32	3
Full	6	508	127	64	3
Complete	7	764	191	64	4
Full	7	1020	255	128	4

Table 5. Communication-time parameters.

Parameters	Values
W_{size}	$4\ Bytes \times Byte\ Size = 4 \times 8 = 32\ bits$
L_{size}	$4\ Bytes \times Byte\ Size = 4 \times 8 = 32\ bits$
El_{speed}	$250\ Mbps$
Op_{speed}	$2.5\ Gbps$
T_{we}	$\frac{W_{size}}{El_{speed}} = \frac{32}{250 \times 1024 \times 1024} = 122.1\ nsec$
T_{wo}	$\frac{W_{size}}{Op_{speed}} = \frac{32}{2.5 \times 1024 \times 1024 \times 1024} = 12.2\ nsec$

Moreover, to compute and analyze the communication time, the values of word size (W_{size}) which is equal to the element size (L_{size}), electronic link speed (El_{speed}) [36]-[37], optical link speed (Op_{speed}) [36], time to transmit word of data on the electronic link (T_{we}) and time to transmit word of data on the optical link (T_{wo}), are shown in Table 5.

7.2 Simulation Results

In this sub-section, the simulation results are presented and evaluated in terms of speedup and efficiency using random and descending input-data distributions. Figures 4 and 5 show the speedup of

PBS using random and descending distributions of different sizes on a different number of processors on OCCT, respectively. In these figures, the speedup was highest when the input is 40 MB and lowest when the input is 0.4 MB. However, two main cases can be observed from Figures 4 and 5 as follows:

- For a certain size of the input data distribution, the speedup increases as the number of processors increases. This is because, as we increase the number of processors, the computation time on each processor decreases since the data on each processor is decreased in size.
- For a certain number of processors, the speedup increases as we increase the size of the input-data distribution. This is because the gap between parallel runtime and sequential runtime increases as the size of data is increased.

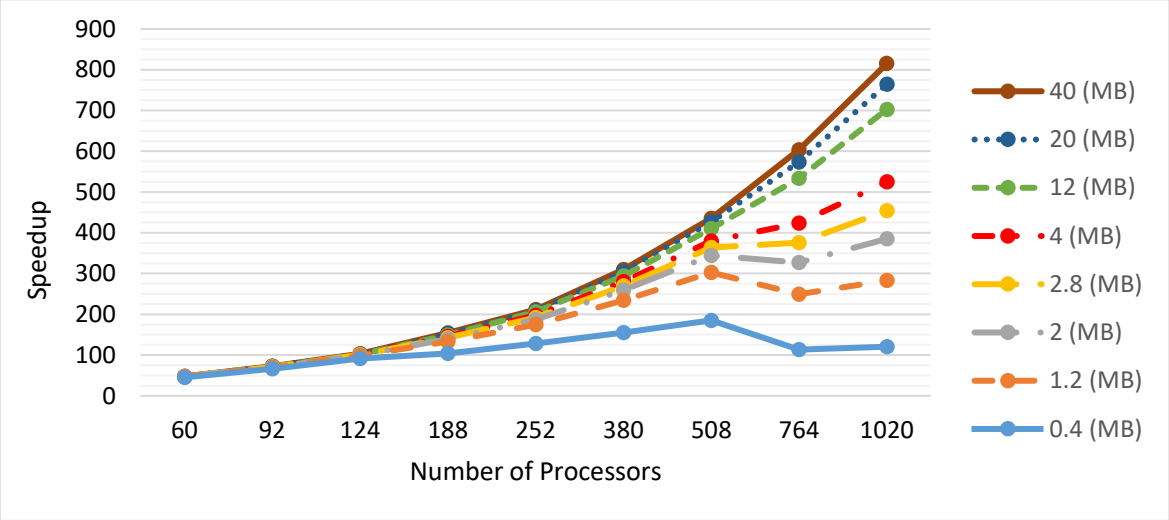

Figure 4. PBS speedup for various random data-distribution sizes on OCCT.

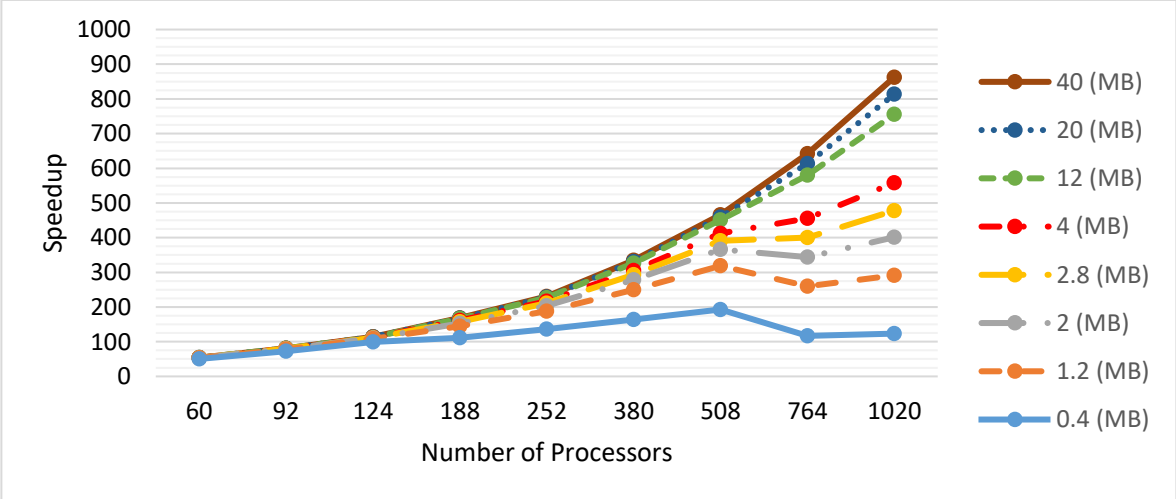

Figure 5. PBS speedup for various descending data-distribution sizes on OCCT.

Figures 6 and 7 show the efficiency of the PBS algorithm running on a different numbers of processors using random and descending distributions over OCCT, respectively. The highest efficiency is achieved, which is approximately 92%, when we used descending data distribution of size 40 MB on 124 processors, as shown in Figure 7. However, two main cases can be observed from Figures 6 and 7 as follows:

- For a certain small size of the input-data distribution, the efficiency decreases as the number of processors increases.
- For a certain number of processors, the efficiency decreases as we decrease the size of the input data distribution.

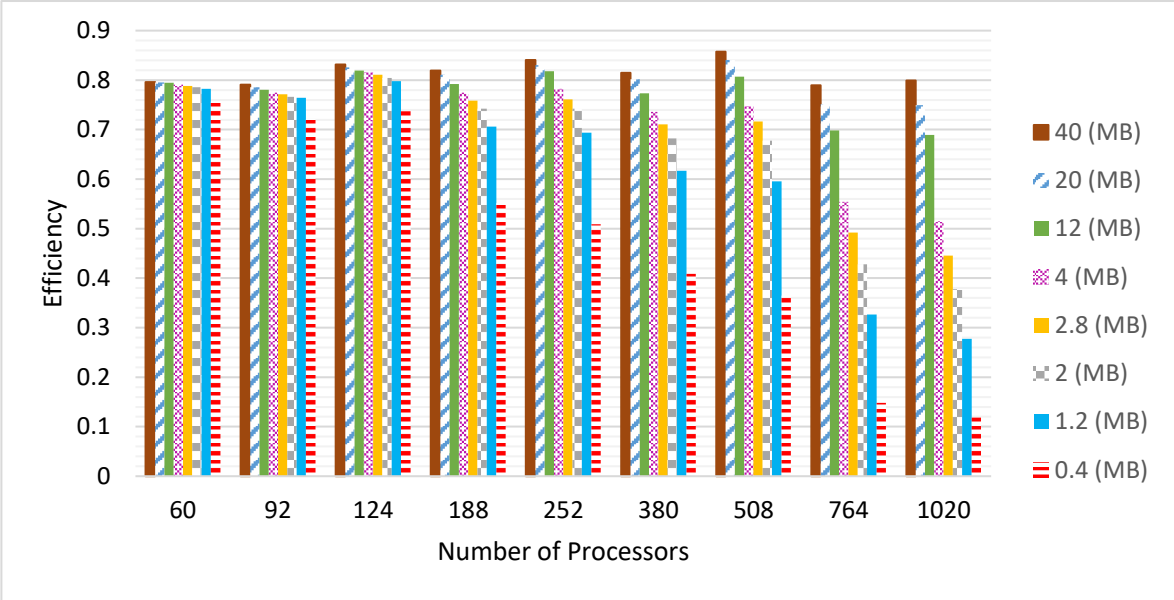

Figure 6. PBS efficiency for various random data-distribution sizes on OCCT.

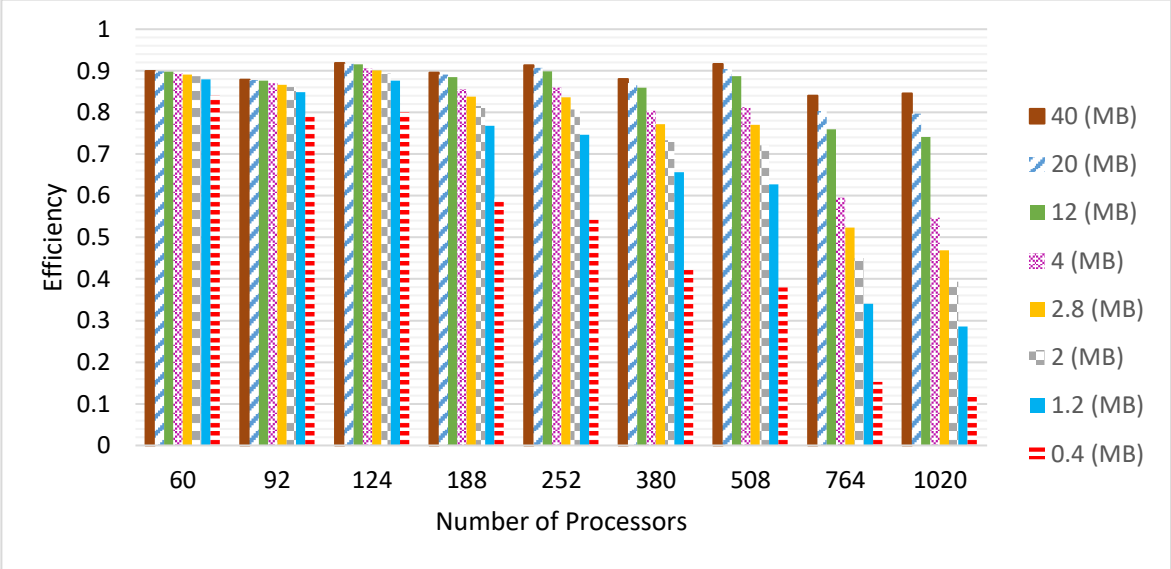

Figure 7. PBS efficiency for various descending data-distribution sizes on OCCT.

In general, data distribution affects the performance of the PBS algorithm. Specifically, sorting random data distribution on OCCT using the sequential quicksort on each processor, as mentioned in Algorithm 2, will lead to the best and average case scenarios, whereas sorting descending data distribution will lead to the worst-case scenario.

Figure 8 shows the scalability of the proposed PBS algorithm in terms of different large data sizes ranging from 60 MB to 400 MB presented as random data distributions on 1020 OCCT processors. Specifically, as shown in Figure 8, as the data gets larger, the speedup gets higher; that is for 60, 100, 150, 200 and 400 MB, the speedup is approximately 832, 846, 857, 861 and 871, respectively. Additionally, the proposed PBS algorithm is compared with the parallel quicksort (PQS) algorithm in terms of speedup, as shown in Figure 8. In this comparison, the PBS algorithm is applied on the OCCT interconnection network using 1020 processors, whereas the PQS algorithm is applied on the OHHC interconnection network using 1152 processors. The difference in the number of processors is due to the structures of the OCCT and OHHC interconnection networks, as shown in [31][34]. As shown in Figure 8, the PBS algorithm outperforms the PQS algorithm for all ranges of data sizes. However, for 400 MB, the PBS algorithm outperforms the PQS algorithm slightly; specifically the PBS algorithm achieved a speedup of 871, whereas the PQS algorithm achieved a speedup of 867. This is due to the number of processors in OHHC which has more processors than OCCT by 132.

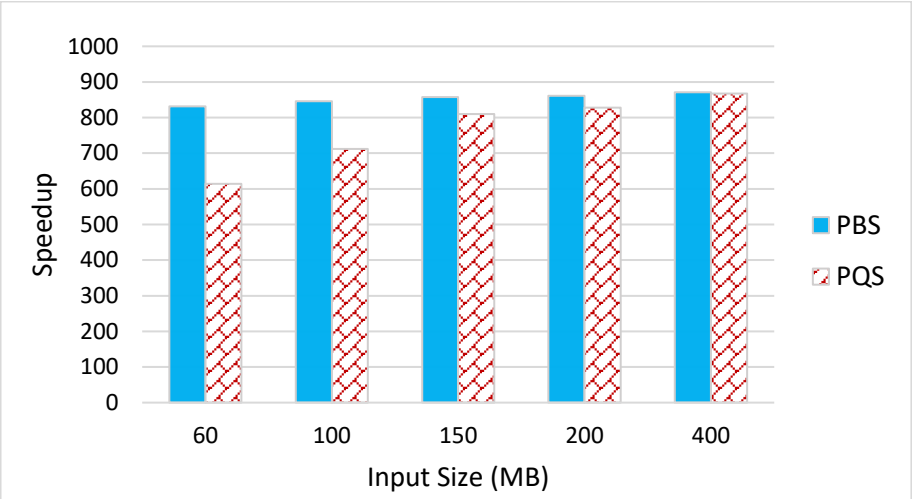

Figure 8. Speedup comparison between the PBS algorithm on 1020 OCCT processors and the PQS algorithm on 1152 OHHC processors for various random data-distribution sizes.

Additionally, as shown in Table 6, the PBS on OCCT is compared with the parallel bucket-sort algorithm using various techniques and architectures, where random data distribution of size 40 MB is used. In this table, the results show that PBS on OCCT outperforms these techniques and architectures since the parallel bucket-sort algorithms are implemented on shared-memory architectures with limited resources. Specifically, as the number of threads increases, the performance of these algorithms decreases; that is, creating more threads than cores degrades the performance of these algorithms, as mentioned in [27][29].

Table 6. Speedup comparisons between the PBS algorithm on OCCT and other parallel bucket-sort algorithms using different techniques and architectures.

Technique/Architecture	Speedup	Threads/Nodes	Number of Buckets
Multi-threaded [29]	1.88	8	100
OpenMP [29]	3.13	8	100
GPGPU using CUDA [29]	1.25	8	100
OpenMP API [27]	2.2	8	100
PBS on OCCT	**6.54**	8	100

Table 7 shows analytical *versus* simulation speedup for the PBS algorithm using 400 MB descending data distribution on OCCT for different numbers of processors. It can be seen from the table that the difference between the analytical and simulation speedup is small, ranging from 7% up to 12%, which validates the correctness of the obtained simulation results.

Table 7. Analytical versus simulation speedup results of PBS algorithm over various numbers of processors on OCCT using descending data distribution of 400 MB input size.

	Number of Processors								
	60	**92**	**124**	**188**	**252**	**380**	**508**	**764**	**1020**
Analytical	59.995	91.989	123.981	187.902	251.824	379.594	507.2651066	760.377	1013.471
Simulation	53.996	80.951	114.063	169.112	231.678	337.839	471.7565491	669.132	912.124
Difference	**10%**	**12%**	**8%**	**10%**	**8%**	**11%**	**7%**	**12%**	**10%**

8. CONCLUSIONS

In this paper, an efficient PBS is implemented on OCCT using up to 1020 processors, up to 400 MB of input-data size and two data distributions; namely, random and descending. The performance of the PBS algorithm on OCCT is evaluated analytically in terms of parallel runtime, which includes computation, communication and concatenation, in addition to speedup and efficiency. Also, the PBS algorithm is evaluated by simulation in terms of speedup and efficiency. Moreover, a comparison is

presented in terms of speedup between the PBS algorithm on 1020 processors of the OCCT and the PQS algorithm on 1152 processors of the OHHC for random data distribution ranges from 60 MB to 400 MB.

As simulation results, the PBS algorithm on OCCT outperforms the PQS algorithm on OHHC in terms of speedup for various random data sizes ranging from 60 MB to 400 MB. A maximum speedup of approximately 912x is obtained on OCCT using 1020 processors and descending input-data distribution of size 400 MB. Also, a maximum efficiency of approximately 92% is obtained on OCCT using 124 processors and descending input-data distribution of size 40 MB, which means that the utilization of the OCCT processors reaches 92%.

In general, the PBS algorithm has some limitations, where its performance can be affected by the type of data distribution. For example, when the data distribution is random, then the best and average cases are obtained, since we used the sequential quicksort to sort the data locally at each processor. Whereas the worst-case is obtained when we used the descending data distribution for the same mentioned reason.

Moreover, in general, bucket-sort performance is sensitive to the distribution of the input values; so, if you have tightly clustered values, it is not recommended. Also, the performance of bucket sort depends on the number of buckets chosen, which might require some extra performance tuning compared to other algorithms. However, these limitations need to be considered when the bucket-sort algorithm is applied to various architectures.

As potential future research directions to this work, the PBS algorithm can be applied to other opto-electronic interconnection networks, such as the OTIS and its variants, to show the performance of such opto-electronic interconnection networks [31] [33]. Moreover, the PBS algorithm can be applied to other well-known architectures, such as multi-threaded architectures, shared-memory multi-core architectures and mesh-connected multi-processors to evaluate their performance [1], [39]-[40].

ACKNOWLEDGMENT

The author would like to express his deep gratitude to the anonymous referees for their valuable comments and helpful suggestions, which enhanced this research manuscript. This research work was conducted during the sabbatical leave from the University of Jordan for the academic year 2022/2023, where this research work was accomplished at the Department of Computer Science, King Hussein School of Computing Sciences, Princess Sumaya University for Technology, Amman, Jordan.

REFERENCES

[1] L. Rashid, W. M. Hassanein and M. A. Hammad, "Analyzing and Enhancing the Parallel Sort Operation on Multithreaded Architectures," Journal of Supercomputing, vol. 53, pp. 293–312, 2010.

[2] N. R. Nitin, "Analysis of Multi-sort Algorithm on Multi-mesh of Trees (MMT) Architecture," Journal of Supercomputing, vol. 57, pp. 276–313, 2011.

[3] S. Al-Haj Baddar and B. Mahafzah, "Bitonic Sort on a Chained-cubic Tree Interconnection Network," Journal of Parallel and Distributed Computing, vol. 74, pp. 1744–1761, 2014.

[4] F. Dehne and H. Zaboli, "Parallel Sorting for GPUs," In: Adamatzky, A., editor. Emergent Computation. Emergence, Complexity and Computation (ECC), vol. 24, DOI: 10.1007/978-3-319-46376-6_12, Springer, Cham., 2017.

[5] A. Al-Adwan, R. Zaghloul, B. Mahafzah and A. Sharieh, "Parallel Quicksort Algorithm on OTIS Hyper Hexa-Cell Optoelectronic Architecture," Journal of Parallel and Distributed Computing, vol. 141, pp. 61–73, DOI: 10.1016/j.jpdc.2020.03.015, 2020.

[6] M. K. I. Rahmani, "Smart Bubble Sort: A Novel and Dynamic Variant of Bubble Sort Algorithm," Computers, Materials & Continua, vol. 71, no. 3, pp. 4895–4913, 2022.

[7] P. Preethi, K. G. Mohan, S. Kumar and K. K. Mahapatra, "Sorter Design with Structured Low Power Techniques," SN COMPUT. SCI., vol. 4, no. 129, DOI: 10.1007/s42979-022-01546-7, 2023.

[8] Y. Han and X. He, "More Efficient Parallel Integer Sorting," International Journal of Foundations of Computer Science, vol. 33, no. 5, pp. 411–427, DOI: 10.1142/S0129054122500071, 2022.

[9] B. Bramas, "A Fast Vectorized Sorting Implementation Based on the ARM Scalable Vector Extension (SVE)," PeerJ Computer Science, vol. 7, p. e769, DOI: 10.7717/peerj-cs.769, 2021.

[10] O. Obeya, E. Kahssay, E. M. Fan and J. Shun, "Theoretically Efficient and Practical Parallel In-place Radix Sorting," Proc. of the 31st ACM Symposium on Parallelism in Algorithms and Architectures,

DOI: 10.1145/3323165.3323198, 2019.

[11] T. Tokuue and T. Ishiyama, "Performance Evaluation of Parallel Sortings on the Supercomputer Fugaku," Journal of Information Processing, vol. 31, pp. 452–458, 2023.

[12] S. K. Gill, V. P. Singh, P. Sharma and D. Kumar, "A Comparative Study of Various Sorting Algorithms," Int. Journal of Advanced Studies of Scientific Research, vol. 4, pp. 367–372, 2019.

[13] A. Grama, A. Gupta, G. Karypis and V. Kumar, Introduction to Parallel Computing, 2ed Edition, Reading: Addison-Wesley, 2003.

[14] T. Cormen, C. Leiserson, R. Rivest and C. Stein, Introduction to Algorithms, 4th Edition, Massachusetts: The MIT Press, 2022.

[15] H. Wang et al., "PMS-Sorting: A New Sorting Algorithm Based on Similarity," Computers, Materials & Continua, vol. 59, no. 1, pp. 229–237, DOI: 10.32604/cmc.2019.04628, 2019.

[16] M. Nowicki, "Comparison of Sort Algorithms in Hadoop and PCJ," Journal of Big Data, vol. 7, no. 101, DOI: 10.1186/s40537-020-00376-9, 2020.

[17] M. Garland, "Chapter 13 – Sorting," in the Book: Programming Massively Parallel Processors: A Hands-on Approach, 4th Edition, Morgan Kaufmann Publisher, pp. 293–310, DOI: 10.1016/B978-0-323-91231-0.00019-7, 2023.

[18] W. X. Zhang and Z. Wen, "Efficient Parallel Algorithms for Some Integer Problems," Proc. of the 19th Annual Conference on Computer Science (CSC '91), pp. 11–20, San Antonio, USA, DOI: 10.1145/327164.327169, 1991.

[19] T. Rożen, K. Boryczko and W. Alda, "GPU Bucket Sort Algorithm with Applications to Nearest-Neighbour Search," Journal of WSCG, vol. 16, pp. 161–167, 2008.

[20] M. Amirul et al., "Sorting Very Large Text Data in Multi GPUs," Proc. of the 2012 IEEE Int. Conf. on Control System, Computing and Engineering, pp. 160–165, Penang, Malaysia, DOI: 10.1109/ICCSCE.2012.6487134, 2012.

[21] M. Asiatici, D. Maiorano and P. Ienne, "How Many CPU Cores Is an FPGA Worth? Lessons Learned from Accelerating String Sorting on a CPU-FPGA System," Journal of Signal Processing Systems, vol. 93, pp. 1405–1417, DOI: 10.1007/s11265-021-01686-8, 2021.

[22] H. Chen, S. Madaminov, M. Ferdman and P. Milder, "Sorting Large Datasets with FPGA-accelerated Sample Sort," Proc. of 27th IEEE Int. Symposium on Field-Programmable Custom Computing Machines (FCCM 2019), Art. no. 8735541, p. 326, DOI: 10.1109/FCCM.2019.00067, 2019.

[23] M. Kaur and V. Kumar, "Parallel Non-dominated Sorting Genetic Algorithm-II-based Image Encryption Technique," The Imaging Science Journal, vol. 66, no. 8, pp. 453–462, 2018.

[24] J. Xie, Z. Li, H. Wu, L. Li, B. Pan, P. Guo and G. Sun, "Application of Quicksort Algorithm in Information Retrieval," Journal on Big Data, vol. 3, no. 4, pp. 135–145, 2021.

[25] N. Faujdar and S. Saraswat, "The Detailed Experimental Analysis of Bucket Sort," Proc. of the 7th Int. Conf. on Cloud Computing, Data Science & Engineering (Confluence), pp. 1–6, Noida, India, DOI: 10.1109/confluence.2017.7943114, 2017.

[26] M. Khurana, N. Faujdar and S. Saraswat, "Hybrid Bucket Sort Switching Internal Sorting Based on the Data Inside the Bucket," Proc. of the 6th Int. Conf. on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), pp. 476–482, DOI: 10.1109/icrito.2017.8342474, Noida, India, 2017.

[27] H. Hong, "Parallel Bucket Sorting Algorithm Hiep Hong," [Online], Available: https://api.semanticscholar.org/CorpusID:33533373, 2014.

[28] G. Beliakov, G. Li and S. Liu, "Parallel Bucket Sorting on Graphics Processing Units Based on Convex Optimization," Optimization, vol. 64, pp. 1033–1055, 2015.

[29] H. I. S. Wijayabandara, Performance Analysis of Parallel Bucket Sort, Thesis for Master's Degree in Computer Science, University of Colombo, School of Computing, 2018.

[30] K. Chen, H. Chen and C. Wang, "Bucket MapReduce: Relieving the Disk I/O Intensity of Data-intensive Applications in MapReduce Frameworks," Proc. of the 29th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing (PDP), DOI: 10.1109/pdp52278.2021.00013, 2021.

[31] B. Mahafzah, M. Alshraideh, T. Abu-Kabeer, E. Ahmad and N. Hamad, "The Optical Chained-cubic Tree Interconnection Network: Topological Structure and Properties," Computers & Electrical Engineering, vol. 38, pp. 330–345, DOI: 10.1016/j.compeleceng.2011.11.023, 2012.

[32] B. Mahafzah, A. Al-Adwan and R. Zaghloul, "Topological Properties Assessment of Opto-electronic Architectures," Telecomm. Systems, vol. 80, pp. 599–627, DOI: 10.1007/s11235-022-00910-5, 2022.

[33] G. C. Marsden, P. J. Marchand, P. Harvey and S. C. Esener, "Optical Transpose Interconnection System Architectures," Optics Letters, vol. 18, pp. 1083–1085, 1993.

[34] B. A. Mahafzah, A. Sleit, N. A. Hamad, E. F. Ahmad and T. M. Abu-Kabeer, "The OTIS Hyper Hexa-Cell Optoelectronic Architecture," Computing, vol. 94, pp. 411–432, 2012.

[35] M. Abdullah, E. Abuelrub and B. Mahafzah, "The Chained-cubic Tree Interconnection Network," Int. Arab Journal of Information Technology, vol. 8, pp. 334–343, 2011.

[36] O. Kibar, P. J. Marchand and S. C. Esener, "High Speed CMOS Switch Designs for Free-space Opto-

electronic MINs," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 6, pp. 372–386, DOI: 10.1109/92.711309, 1998.

[37] I. Kaminow, T. Li and A. Willner, "Optical Fiber Telecommunications VB: Systems and Networks," 5th Edition, Academic Press, 2008.

[38] D. K. J. Lin and J. Chen, "Adaptive Order-of-Addition Experiments *via* the Quick-sort Algorithm," Technometrics, vol. 65, no. 3, pp. 396–405, DOI: 10.1080/00401706.2023.2174601, 2023.

[39] M. S. Mohammed and G. A. Abandah, "Characterization of Shared-memory Multi-core Applications," Jordanian Journal of Computers and Information Technology (JJCIT), vol. 2, no. 1, pp. 37–54, DOI: 10.5455/jjcit.71-1448574289, 2016.

[40] J. Al-Azzeh, "Distributed Mutual Inter-unit Test Method for D-Dimensional Mesh-connected Multiprocessors with Round-Robin Collision Resolution," Jordanian Journal of Computers and Information Technology (JJCIT), vol. 5, no. 1, pp. 1–16, DOI: 10.5455/jjcit.71-1539688899, 2019.

ملخص البحث:

إنّ لأداء خوارزميات التّصنيف أثراً كبيراً على العديد من التّطبيقات التي تنطوي على حساباتٍ مكثّفة. وقد عمل الباحثون على موازاة الكثير من خوارزميات التّصنيف في شبكات اتّصال متنوّعة بغية تحسين أدائها المقابل التّتابعي. ومن بين تلك الشّبكات ما يُعرف بالشّجرة المكّعبة المسَلْسَة ضوئياً.

في هذه الورقة، نقدّم خوارزمية تصنيف دَلْوِيّة متوازية، ونقوم بتطبيقها على شبكة اتّصال مكّعبة مُسَلْسَة ضوئياً. كذلك نعمل على تقييم الأداء التّصنيفي لتلك الخوارزمية عن طريق كلٍّ من التّحليل والمحاكاة من حيث مجموعة متنوّعة من مقاييس الأداء بما فيها زمن التّشغيل المتوازي، وزمن الحساب، وزمن الاتّصال، وزمن التّجميع، والسّرعة، والفعالية؛ وذلك لأعداد مختلفة من المعالِجات، ولأحجام مختلفة من مجموعات البيانات، ولأنواع مختلفة من توزيع البيانات (التّوزيع العشوائي، والتّوزيع التّنازلي).

وبيّنت نتائج المحاكاة أنّ أقصى سرعة تمّ الحصول عليها كانت لشبكة اتّصال فيها (1020) معالِجاً، حيث بلغ زمن التّشغيل المتوازي باستخدام (1020) مُعالِجاً (912) ضِعْفاً مقارنة بزمن التّشغيل التّتابعي للتّصنيف الدَّلْوِي باستخدام مُعالِجٍ واحد.