Parallelization of Cocktail Sort with MPI and CUDA

Karthik C R [0000-0002-8196-4092], Ashwin G Shanbhag [0000-0002-8694-5130], Ashwath Rao B [0000-0001-8528-1646], Prakash K. Aithal [0000-0002-4304-9512], Gopalakrishana N Kini [0000-0002-9293-7460]

Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal

Academy of Higher Education, Manipal, Karnataka- 576104

Karthikcr604@gmail.com

ashwingshan98@gmail.com

Abstract. Sorting is an essential operation that has a wide range of applications in the day-to-day life of programming. It is quicker and easier to recognize items in a sorted list compared to that in unsorted. The parallelization of an algorithm is appropriate when the problem is highly intense computationally or when it is crucial to get the results rapidly. In the past few years, a plethora of works has been proposed to contract the time complexity and space complexity. Parallelizing the sorting algorithms gives additional strength to the algorithm by enhancing the performance. In this paper, we derive an algorithm to parallelize one of the in-place sorting algorithms called cocktail sort. We have employed MPI (Message passing interface) and Nvidia CUDA (Compute Unified Device Architecture) parallel computing platforms for parallelizing the cocktail sort. Comparison of time taken by the algorithm with MPI and CUDA platform are exhibited and a graph depicting same is plotted. Calculation of the speedup of the cocktail sort has been showcased. Also, proved that the time complexity of the parallelized cocktail sort has been reduced to O(n) from $O(n^2)$ when compared with the sequential algorithm. The results unveil that the parallelizing enhances the performance and efficiency of the cocktail sorting algorithm.

Keywords: CUDA, Cocktail sort, Parallel Computation, MPI, Threads, Time complexity.

1 Introduction

Sorting is a way of arranging the data in a specific format. The sorting algorithm determines how data can be sorted. There are plenty of applications where these sorting algorithms are utilized. Major applications are in searching data in the list and matching entries in the list. Sorting algorithms can be categorized as in-place sorting and not in-place sorting. In-place [1] sorting algorithms are algorithms that do not require any extra space and sorting is intended to happen within the array or list itself. One of the commonly used in-place algorithms is Bubble sort. Not in-place algorithms refer to those algorithms that require equal or extra space based on the elements to be sorted. A general example of a not in-place sorting algorithm is merge sort. Sorting algorithms has a major role in the day-to-day life of programming because of their extensive range of applications.

MPI is an interface between processes. Processes coordinate and communicate via calls to message passing library routines. MPI addresses primarily the message-passing parallel programming model. The message-passing program can exchange local data by using communication operations. MPI consists of a collection of processes. Processes in MPI are heavy-weighted and single-threaded with separate address spaces. On many parallel systems, an MPI program can be started from the command line.

Nvidia Introduced CUDA, a parallel computing platform. CUDA [2] utilizes the power of GPU (graphical processing unit) for computing parallel parts and CPU (central processing unit) for the sequential part of the program. At first, GPU was only used for gaming purposes. Now due to its high computation capability, the use of GPU gradually extended to the field of science maths, etc. when computation to be done is very high and operations are loosely coupled, using GPU increases the performance of the system.

In this paper, we parallelize the cocktail sort [3] algorithm using MPI and CUDA platforms. Before discussing the parallelization of cocktail sort let's discuss cocktail sort in brief. Cocktail sort, also known as cocktail shaker sort is an in-place sorting

algorithm. Like bubble sort [4], it is also an even and comparison-based sorting technique. Cocktail sort sorts the list in two phases per iteration. Firstly, it performs rightward pass and then leftward pass. After the rightward pass, the largest element is moved to the rightmost corner of the list and in the leftward pass, the smallest element is moved to the left-most corner. The outcome of the first iteration is that the smallest element in the list will be placed on the leftmost corner in the list and the largest element is placed at the rightmost corner. These iterations are continued until the array or list is sorted.

Working of the cocktail sort is illustrated using the below example. Consider an array of integers as shown in Fig 1. Now in cocktail sort, the initial procedure is to check the first two elements of the array if the first element is greater than the second it swaps the element. At the end of the first forward pass, the largest element will be at the rightmost corner as shown in Fig 1 (left). Then it performs a backward pass where at the end the smallest element is settled at the leftmost corner of the array as shown in Fig 1 (right). Thus, the first iteration completes. If the array is not sorted then it goes to the second iteration and follows the process same as that of iteration 1 as shown in Fig 2 (left). But in iteration 2 second forward pass we can see that at step 3 itself the array is sorted but the algorithm does not know that the array is sorted so it performs the second backward pass as shown in Fig 2 (right) when no element is swapped in a pass it concludes that the array is sorted.

Fig 1: First iteration. First rightward pass to take the largest element to the rightmost corner (left). First leftward pass to bring the smallest element to the left most corner of the array (right).

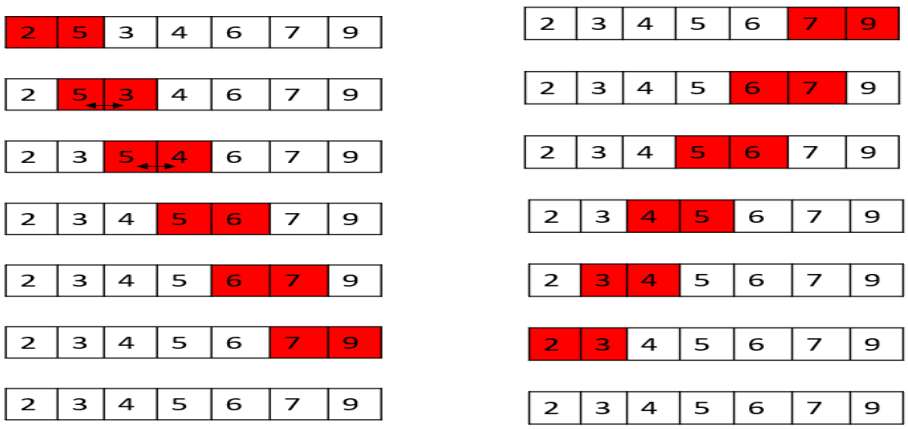

Fig 2: second Iteration. second rightward pass (left). The array is sorted in the second forward pass itself but to confirm second leftward pass (right) is performed when no element is swapped in a pass it concludes that the algorithm is sorted.

From the above example, the time complexity for the best case is O(n), when the array is already sorted. But, in the case of average and worst-case when the array is not

sorted the time complexity of the algorithm is $O(n^2)$. The rest of the paper is organized as follows: Section 2 gives a brief knowledge on previously carried work in the field of sorting as well as parallelizing the algorithm. Section 3 broadly discusses the implementation of the cocktail sort both sequentially as well as parallelly. Section 4 gives a detailed analysis of the results to support the proposed algorithm. Section 5 concludes the paper and section 6 has the references used in this paper.

2 LITERATURE SURVEY

In this section, we discuss various previously carried work in the field of sorting as well as parallelizing the algorithms using a parallel computing platform. Sonal et.al [5] in their paper compared different sorting algorithms like bubble, quick, heap, merge, insertion sort and derived a conclusion that speed is not the only factor that determines the good sorting algorithm but also it depends on length, the complexity of code, stability, datatype handling capacity, and performance also play a part in determining the good sorting algorithm. Rohit et.al [6] in their work compared the time complexity of different sorting algorithms like bucket, radix, and counting sorting by taken 1, 2, …, 10 as input and showed that non-comparison-based algorithms have O(n) complexity which is better than comparison-based algorithm with O (n log n).

Nidhi et.al [7] in their work compared three sorting algorithms Quick, heap and insertion sort, and showed that when compared to the heap and quick sort insertion sort is slower when the input of 10,000 and 30,000 random values were given. However, the time complexity of all three algorithms is $O(n^2)$. Gaurav et.al [8] in their paper compared the bubble, quick, merge, heap, and insertion sort based on speed by ranging the number of input values from 100000 to 1500000. They also ranked these algorithms based on speed as follows merge, quick, heap, insertion, and bubble sort. They also showed that when the large data is used merge, quick and heap sort performance is better than the bubble and insertion sort.

Recently, the parallelization of Tim's sort, Super sort, cut sort, and Matrix sort [15,16,17,18] have been carried out using MPI and CUDA. Each of these sorting algorithms has been redesigned to run in a parallel environment consisting of CPU and GPU. The results are encouraging and prove that many sorting algorithms can be parallelized with a significant reduction in execution time.

Parallelizing the sorting algorithm increases the performance of the algorithm. But, parallelizing a sorting algorithm is a challenging task. Ashwin Kumar et.al [9] in their hybrid approach showcased that how a sequential code can be parallelized with function level and block level parallelization. Tjaden et.al [10] in their work proposed approaches for detection and execution of parallel instruction. David W [11] in his study showcased the drawbacks of instruction, block, and functional level parallelization. In the case of instruction-level parallelization, there are the chances of high communication overheads. In function level parallelization there are chances that performance is not satisfactory I,e. if there are two functions that if one function has more loops than the other the speedup of the system may be reduced. Block-level parallelization has better performance when compared with the other two but if there are more than one function calls in a block and they take part in determining the total complexity then running such blocks in parallel may not give good results.

In the case of selection sort [12], there is a need to find out min and max in the unsorted every time due to a decrease in the size of the unsorted list with each iteration. Hence the recursive sorting algorithms are best suited for parallelization. Tsigas et.al [13] proposed an algorithm for parallelizing the quick sort by introducing the multiple threads during the time of partitioning the array into two. Verman et.al [14] introduced the parallel merge sort algorithm where they divided the unsorted array based on the number of threads and each partition was sorted by one thread. After sorting using each thread sorted partitions are merged. Using these concepts, we introduce a novel algorithm for parallelizing the cocktail sort using MPI and CUDA platforms which are

discussed in detail in the next section.

3 METHODOLOGY

In this section, we discuss how the cocktail sort algorithm can be implemented in sequential as well as parallel using MPI and CUDA platforms. In the sequential implementation of cocktail sort, we just take N number of elements as input and simply perform the operations as discussed in section 1. The general flow diagram for sequential cocktail sort is shown in Fig 3.

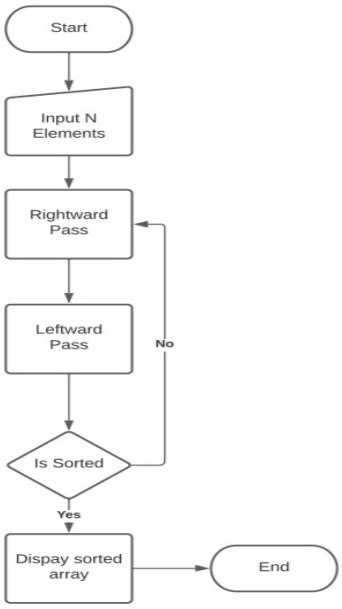

Fig 3. Flow diagram for sequential Cocktail Sort

Steps involved in implementing sequential cocktail sort:

Step 1: Take N elements as an input that is to be sorted

Step 2: Check whether the list is sorted or not, if not go to step 3.

Step 3: Perform right forward pass on the list

Step 4: Perform left forward pass on the list

Step 5: Perform steps 2 and 3 until the list is not sorted.

Step 6: If no elements in step 4 are swapped then the array is sorted. Hence, exit.

For implementing cocktail sort in parallel the general idea is distributing the number of elements (N) to the different processes so that the array is sorted simultaneously in different processes(P). Now the major task is to assign the processes with the number of elements (N). To do this we divide the given array based on the number of processes (N/P) used and then we send these chunks to the respective processes. Then we perform the sorting on these data chunks. After sorting is done, we gather all the sorted chunks for the root process. Later these sorted chunks are merged in the root process and the final result is printed. Fig 4 represents the flow diagram for parallelization of cocktail sort using MPI and Fig 5 represents the implementation using the CUDA platform. Now, the implementation of cocktail sort parallelly is performed in the following steps.

Steps involved in parallelizing the cocktail sort using MPI:

Step 1: Take N elements as input that is to be sorted.

Step 2: Define the P number of processes.

Step 3: Create chunks of array based on the number of processes P. Chunk size is N/P.

Step 4: Scatter these chunks of array elements to all 'N' processes.

Step 5: Perform sorting operations on these chunks of data then gather all the data from the processes.

Step 6: Gathered sorted chunks from all the processes are merged in root using the merge function.

Step 7: Print the sorted elements in the root.

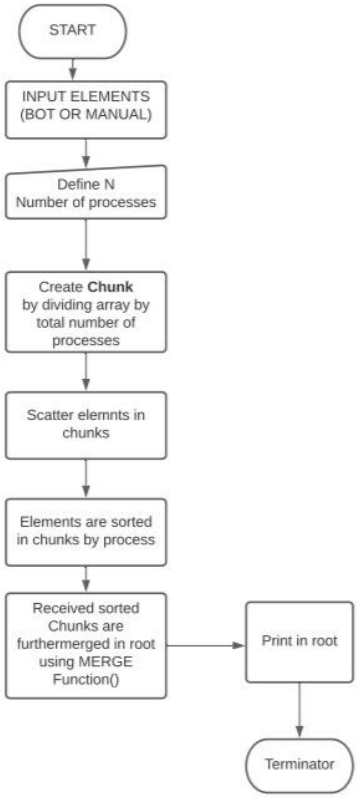

Fig 4. Flow diagram for parallelizing cocktail sort using MPI

From the above steps, we could observe that for the odd number of steps we need (N/2)-1 number of processes and for even-numbered steps require (N/2). Hence it needs O(n) processes. To perform sequential cocktail sort for N elements time complexity of $O(n^2)$. The time complexity of parallel implementation is O(N/P) where N is the number of elements and P is the number of processes. So, the time complexity of the parallel implementation in MPI can be written as $O(n^2)/O(n)$ resulting in O(n) complexity. Hence the time complexity of the parallel cocktail sort in MPI is O(n).

Steps involved in implementing cocktail sort in CUDA:

Step 1: Take input N elements into an array to be sorted.

Step 2: Allocate the variable of size array in the CUDA device's memory.

Step 3: Copy all elements from the host's memory to GPU's memory.

Step 4: Calling parallel cocktail sort kernel function with desired blocks and threads.

Step 5: Computation of sorting performed on each thread simultaneously.

Step 6: Copying back sorted array back from the device to the host's memory.

Step 7: Display sorted array with the time taken. Also, free GPU's memory.

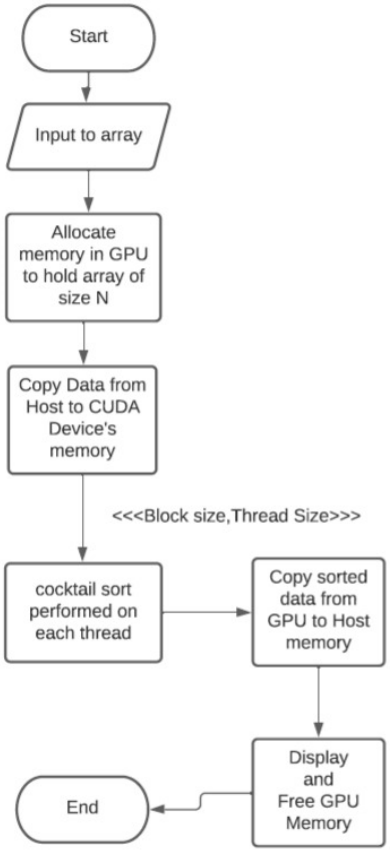

Fig 5. Flow diagram for parallelizing cocktail sort using CUDA

For testing the algorithm, using random values of a different range may lead to different results each time tested. This may mislead the performance of the algorithm. To avoid shortcomings and to achieve the best possible results by minimizing the errors we have made use of a standard dataset [T10I4D100K (.gz)] [19]. Hence using the standard value for sorting will provide an ideal condition for conducting the experiments.

4 Result and Analysis

In this section, we analyze the time consumed by the parallel algorithm when the different number of processes are used to sort the array of different lengths. In MPI as the number of the process increases the time consumed for computation decreases but after a certain saturation point when the number of the process is increased computation time increases becoming directly proportional to the number of processes. This is because in MPI heavyweight threads are used, due to which each thread will be having its memory hence it requires a lot of time for communication among each thread. However, the computation time is purely dependent on the number of processes used. Hence to get better performance choosing the processes according to the number of elements is a good practice. To test the algorithm, we use a standard dataset [T10I4D100K (.gz)] [19]. Table 1 shows the time taken by the sequential algorithm to sort the N number of elements. Where we can observe that as the number of elements increases the computation time also increases.

Table 1. computation time taken for different number of elements for sequential algorithm

Number of elements (N)	Time (ms)
1000	**0.005**
5000	**0.097**
10000	**0.308**
30000	**2.069**
100000	**22.544**

Table 2 represents the time taken by the algorithm in MPI to sort the data by ranging the number of processes when the number of elements is 30000. In MPI the number of processes has to be specified explicitly while at the time of execution. From Table 2 it can be observed that the number of processes is ranging in between 1 to 64. But after 64 the time gradually starts increasing which means that for 30000 elements the number of processes required is around 64 to achieve the best possible parallelism. Hence choosing the proper number of processes based on the given input elements plays a major role in MPI execution. Similarly, Table 3 represents the time consumed by the algorithm in MPI when the number of elements is 100000 and the number of processes is ranging from between 1 to 128. The graph for processes vs time is plotted when the number of elements is 30000 and 100000 which is shown in Fig 6 and Fig 7 respectively.

Table 2. computation time taken by the algorithm when the number of elements is 30000

Number of Processes (P)	Time (ms)
1	**2.141**
2	**0.772**
4	**0.223**
8	**0.080**
16	**0.067**
32	**0.031**
64	**0.026**

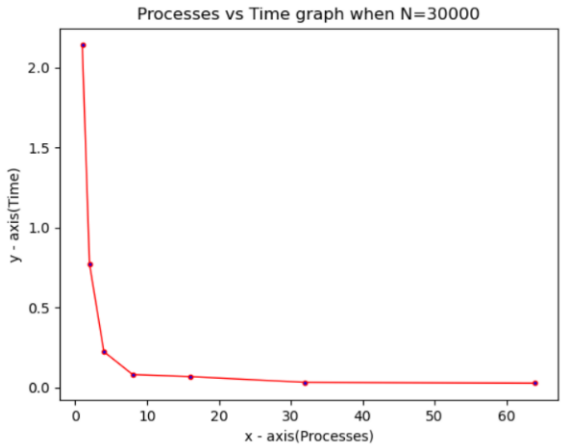

Fig 6. graph of processes vs time when the number of elements is 30000

Table 3. computation time taken by the algorithm when the number of elements is 100000

Number of Processes (P)	Time (ms)
1	**21.547**
2	**6.168**
4	**2.533**
8	**0.967**
16	**0.452**
32	**0.238**
64	**0.154**
128	**0.111**

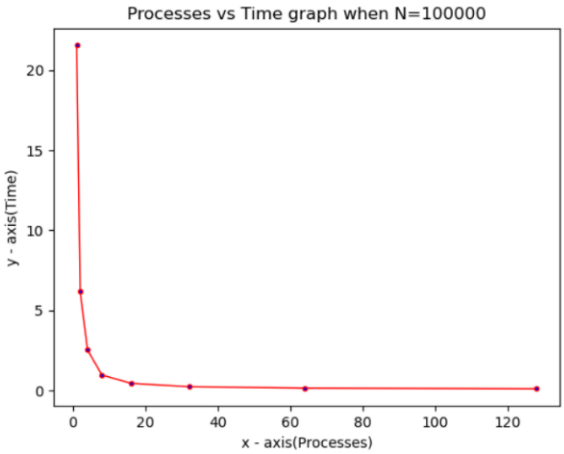

Fig 7. graph of processes vs time when the number of elements is 100000

$$Speedup = \frac{Time\ taken\ by\ sequential\ algorithm}{Time\ taken\ by\ parallel\ algorithm} \quad (1)$$

Considering data from Table 1 and Table 3, the speedup can be computed using equation (1). The speedup is defined as the ratio of time taken by a sequential algorithm to the time taken by a parallel algorithm. To compute consolidated speedup, time taken by various processes in Table 2 and Table3 has been averaged respectively. The speedup of MPI against sequential is shown in Table 4.

Table 4. Speedup of MPI against Sequential.

Number of elements (N)	Speedup
30000	**4.336**
100000	**5.606**

In CUDA, the user need not specify the number of processes as each thread takes an element and sorting operation is performed. Hence based on the number of elements CUDA generates the same number of threads for sorting the data. Table 5 represented the computation time taken by the algorithm when the CUDA platform is used. To test

the computation time the number of elements is ranged between 1000 to 100000 and the respective time is noted down. The graph for number elements vs time is plotted in comparison with the sequential execution time as depicted in fig 8. Using equation (1) The speedup of the algorithm is calculated which is noted in Table 6.

Table 5. computation time taken for different number of elements when CUDA platform is used

Number of elements (N)	Time (ms)
1000	**0.002765**
5000	**0.015432**
10000	**0.052670**
30000	**0.381782**
100000	**3.444165**

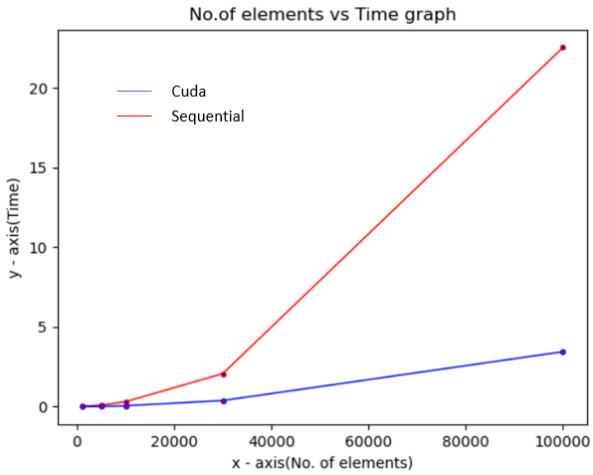

Fig 8. Graph of the number of elements vs time for both sequential and parallel algorithm using CUDA

Table 6. Speedup of CUDA against Sequential.

Number of elements (N)	Speedup
1000	**1.8083**
5000	**6.2856**
10000	**5.8477**
30000	**5.4193**
100000	**6.5456**

5 Conclusion and Future Scope

In this paper, we discuss how cocktail shaker sort can be implemented in a parallel manner using computing platforms like MPI and CUDA. We have also analyzed the computation time taken by the sequential as well as a parallel algorithm using a standard dataset so that experimentation could be performed in ideal conditions. Tabulation of empirical results and representation of it in a graphical manner has been accomplished. We also showed that parallelizing a cocktail sort reduces its time complexity from O (n^2) to O (n) where 'n' is the number of elements. Thus, the results of the experiment depict

that parallelizing the cocktail sort gives additional strength to the algorithm by increasing the performance of the algorithm. In the future, this parallelized algorithm can be packaged into a library and used in multiple platforms where sorting is essential. This parallel implementation can be containerized and deployed over the cloud as free software so that if anyone intends to perform sorting in bulk, can make use of this in the best possible way as the time taken is decreased exponentially.

References

[1] Prajapati, P., Bhatt, N. and Bhatt, N., 2017. Performance Comparison of Different Sorting Algorithms. *vol. VI, no. Vi*, pp.39-41.

[2] Stojanovic, N. and Stojanovic, D., 2014. High-performance processing and analysis of geospatial data using CUDA on GPU. *Advances in Electrical and Computer Engineering*, *14*(4), pp.109-115.

[3] Elkahlout, A.H. and Maghari, A.Y., 2017. A comparative study of sorting algorithms comb, cocktail and counting sorting. *International Research Journal of Engineering and Technology (IRJET), e-ISSN 2395-0056*, *4*(1).

[4] Astrachan, O., 2003. Bubble sort: an archaeological algorithmic analysis. *ACM Sigcse Bulletin*, *35*(1), pp.1-5.

[5] Beniwal, S. and Grover, D., 2013. Comparison of various sorting algorithms: A review. *International Journal of Emerging Research in Management &Technology*, *2*.

[6] Joshi, R., Panwar, G. and Pathak, P., 2013. Analysis of non-comparison-based sorting algorithms: A review. *International Journal of Emerging Research in Management & Technology*.

[7] Chhajed, N., Uddin, I. and Bhatia, S.S., 2013. A comparison-based analysis of four different types of sorting algorithms in data structures with their performances. *International Journal of Advance Research in Computer Science and Software Engineering*, *3*(2), pp.373-381.

[8] Kumar, G. and Chugh, H., 2013. Empirical Study of Complexity Graphs for Sorting Algorithms. *International Journal of Computer, Communication and Information Technology (IJCCIT)*, *1*(1).

[9] Kumar, K.A., Pappu, A.K., Kumar, K.S. and Sanyal, S., 2006, September. Hybrid approach for parallelization of sequential code with function level and block level parallelization. In *International Symposium on Parallel Computing in Electrical Engineering (PARELEC'06)* (pp. 161-166). IEEE.

[10] Tjaden, G.S. and Flynn, M.J., 1970. Detection and parallel execution of independent instructions. *IEEE Computer Architecture Letters*, *19*(10), pp.889-895.

[11] David, W., 1991, April. Wall. Limits of instruction-level parallelism. In *Proceedings of the fourth international conference on Architectural support for programming languages and operating systems* (Vol. 19, pp. 176-188).

[12] Jadoon, Sultanullah, et al. "Design and analysis of optimized selection sort algorithm." International Journal of Electric & Computer Sciences (IJECS-IJENS) 11.01 (2011): 16-22

[13] Tsigas, P. and Zhang, Y., 2003, February. 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, 2003. Proceedings.* (pp. 372-381). IEEE.

[14] Varman, P.J., Scheufler, S.D., Iyer, B.R. and Ricard, G.R., 1991. Merging multiple lists on hierarchical-memory multiprocessors. *Journal of Parallel and Distributed Computing*, *12*(2), pp.171-177.

[15] Siva Thanagaraja, Keshav Shanbhag, Ashwath Rao B, Shwetha Rai, N Gopalakrishna Kini, 2020 “Parallelization of Tim sort algorithm using MPI and CUDA", Journal of Critical Reviews 7.9, 2910-2915.

[16] Pereira, Sushmita Delcy, Rao B. Ashwath, Shwetha Rai, and N. Gopalakrishna Kini 2020. "Super Sort Algorithm Using MPI and CUDA." In Intelligent Data Engineering and Analytics, pp. 165-170. Springer, Singapore.

[17] Yadav, Harshit, Shraddha Naik, B. Ashwath Rao, Shwetha Rai, and Gopalakrishna Kini. 2020 "Comparison of CutShort: A Hybrid Sorting Technique Using MPI and CUDA." In Evolution in Computational Intelligence, pp. 421-428. Springer, Singapore.

[18] Priyanka Ojha, Pratibha Singh, N. Gopalakrishna Kini, Ashwath Rao B, and Shwetha Rai, 2020 “Parallel Matrix sort using MPI and CUDA,” RTU TEQIP-III sponsored 2nd International Conference on Communication and Intelligent systems (ICCIS 2020), 26-27.

[19] http://fimi.cs.helsinki.fi, “Frequent Itemset mining Implementation’s repository” last accessed on 11-02-2021