[[1]](#footnote-1)

Parallelization of Tim Sort Algorithm Using MPI and CUDA

Siva Thanagaraja1, Keshav Shanbhag2, Ashwath Rao B3[0000-0001-8528-1646] , Shwetha Rai4[0000-0002-5714-2611],and N Gopalakrishna Kini5

*Department of Computer Science & Engineering*

*Manipal Institute Of Technology*

*Manipal Academy Of Higher Education*

*Manipal,Karnataka,India-576104*

*Abstract*— Tim Sort [1] is a sorting algorithm developed in 2002 by Tim Peters. It is one of the secure engineered algorithms, and its high level principle includes: The sequence S is divided into monotonic runs (i.e., non-increasing or non-decreasing subsequence of S), which should be sorted and should be combined pair wise according to some specific rules. To interpret and examine the merging strategy (meaning is that the order in which the merge and merge runs are performed) of Tim Sort, we have implemented it in MPI and CUDA environment. Finally, it can be seen difference in the execution time between serial Tim Sort and parallel Tim sort run in O (n log n) time [4].

*Index Terms*— Hybrid algorithm, Tim sort algorithm, Merge sort algorithm, Insertion sort algorithm.

# INTRODUCTION

S

ORTING algorithm Tim Sort [4] was designed in 2002 by Tim Peters. It is a hybrid parallel sorting algorithm which combines two different sorting algorithms that includes insertion sort and merge sort. This algorithm is an effective method for well-defined instructions which is concealed as a finite list for evaluating Tim sort function [1]. Starting from an initial state and its information the guidelines defines a method that when execution continues through a limited number of next states which are well characterized, at the end providing output and ending at a last completion state.

Tim sort works by identifying runs of least two elements. Element runs occurs either strictly descending (each element is lesser than its predecessor) or in non-descending (each element is greater than or equal to its predecessor) order. At its worst, it runs at comparable speed Merge Sort. In other words, it’s unexpectedly fast. In terms of space, Tim Sort is on the worse end of the spectrum; however the space consideration for most sorting algorithms is highly sparse. O(n) isn’t too difficult in most instances. Stability is the concept that when sorted, objects of equal value maintain their original order. If Tim sort follows unstable algorithm, it results in lose any reliability from your first sort when you run the second one. The following steps for Tim sort include:

(i) Existing structure of the list is taken and n-1 operations are performed on the structure list that is either sorted or is in strictly-descending (reverse) order.

(ii) Then the algorithm scans the structure list and finds “runs” of elements which is either in strictly descending or in ascending order.

(iii) If the element runs takes place in strictly descending order, reverse operation of Tim sort occurs.

(iv) If run is less then set “min run”, then the algorithm Tim sort performs Insertion Sort aggregate min run elements. Min run value is calculated based on the size of the array

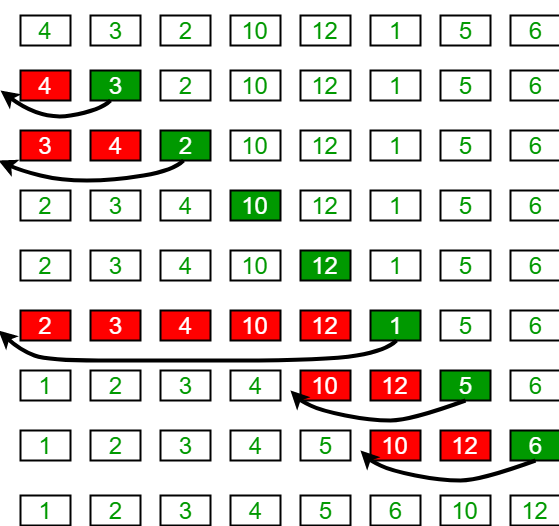
The algorithm merge runs when value of the array exceeds the min run values and also keeps merges balanced.

# Implementation

Tim Sort is complex, even by algorithmic standards. The implementation is best broken down into parts.

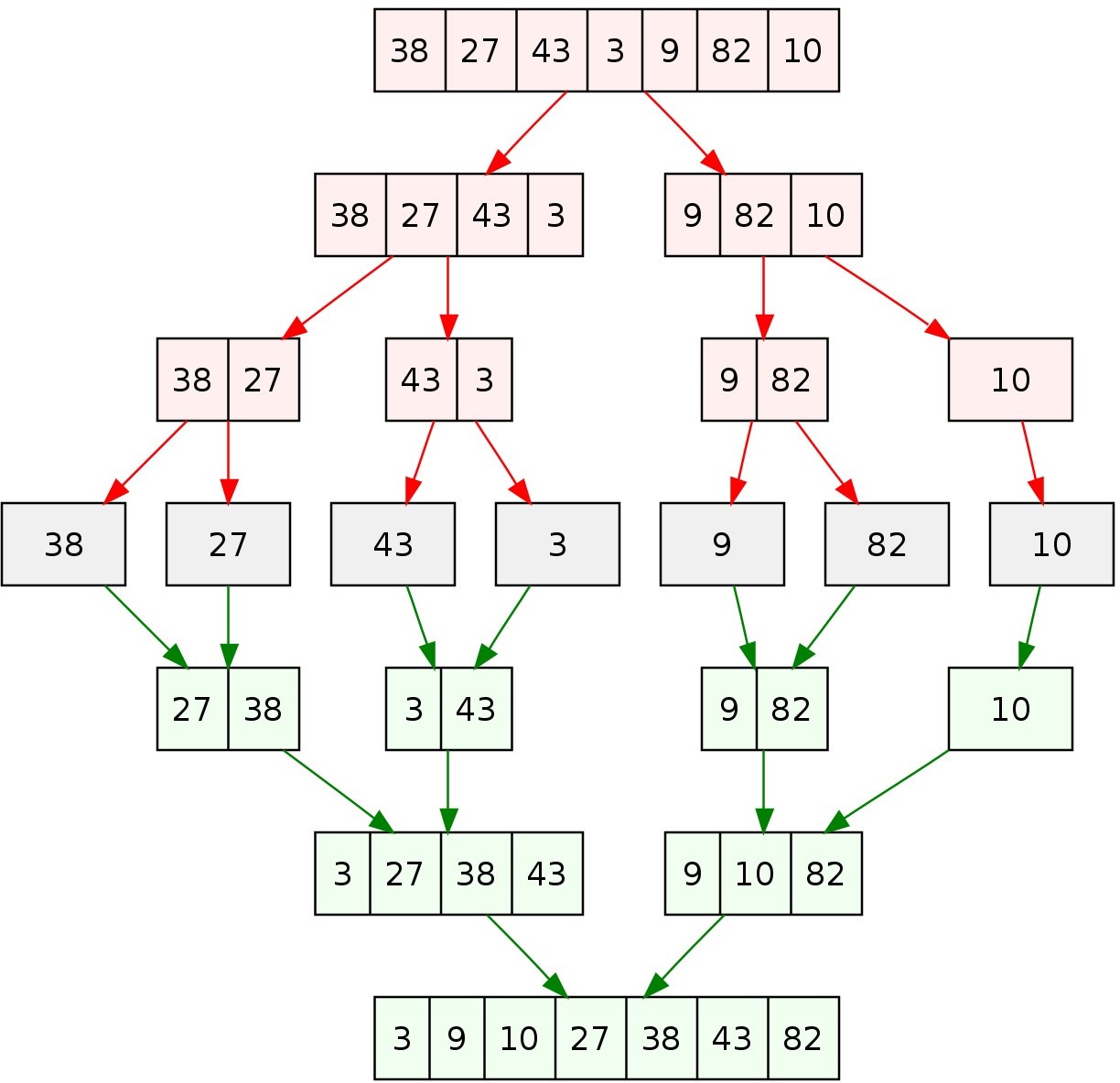
## Insertion Sort & Merge Sort

Insertion Sort is one of the basic fundamental sorting algorithms. It runs through the array, and every time it encounters an object that is out of order (strictly less/more than the object earlier than it), it moves it into the suitable position in the already sorted array as shown in Fig. 1. Insertion Sort is popular for working very rapidly on already sorted arrays, as well as smaller arrays. In fact, we can see from Fig 1, that Insertion Sort has an impressive best case run time of O (n). In Tim Sort the best case for Insertion Sort is an already existing sorted array.



**Fig 1.Insertion Sort Example (source [7])**

Merge Sort on the other hand operates by way of a basic principle: it is extraordinarily easy to merge already sorted arrays. So, it splits a starting array in half over and over until it is nothing however single elements. Then it slowly rebuilds the primary array via merging those elements back together in sorted order as shown in Fig.2. Because we began from building blocks of size one, it was very easy to build preliminary sorted arrays. Then, it’s easy to merge them and this requires O(nlogn) time.



## **Fig.2: Merge Sort Example (Source:[8])**

## Implementing Parallel Tim Sort

The way to comprehension Tim Sort's execution understands its utilization of runs. Tim Sort influences normally happening pre-sorted information to further its potential benefit. By pre-sorted it implies that consecutive components are on the whole expanding or diminishing. In insertion sort one element from the input elements is consumed in each iteration to find its correct position i.e., the position to which it belongs in a sorted array. It iterates the input elements by growing the sorted array at each iteration. It compares the current elements with largest value or smallest value in sorted array. If the current element is greater or lesser then it leaves the element place and moves to the next element else it finds its correct position in the sorted array and moves it to that position. In Merge Sort we divide an array of elements which is unsorted into sub array of equal halves until it can no more be divided. Then merge sort combines smaller sorted lists keeping the new list sorted. Note that insertion sort doesn’t indulge parallelism while merge sort can be implemented parallel.

First set a min run size. If input to Tim sort is lesser than min run then perform insertion sort. Otherwise perform merge sort in which every sub array obtained after divide operation in merge sort, once reaches min run size, it performs Insertion Sort as shown in Data Flow Diagram Fig 3.

Steps to implement parallel Tim sort:

a) Establish a minrun size that is a power of 2 (usually 32, never more than 64 or your Insertion Sort will lose efficiency)

b) Find a run in the first minrun of data.

c) If the run is not at least minrun in length, use Insertion Sort to grab subsequent or prior items and insert them into the run until it is the correct minimum size

.

d) Repeat until the entire array is divided into sorted subsections.

e) Use the latter half of Merge Sort to join the ordered arrays.

## For Parallelism in CUDA

In CUDA, we use threads to sort numbers as each thread controls one set of elements of size 64.First we divide the count of total input array elements by 64 and by this we will get count of threads .After getting number of threads, first divide entire array of size 64 for each threads to perform insertion sort where each thread perform insertion sort of their own array size in parallel. In second step, Groups of two threads will perform merging of their respective array elements obtained after insertion sort in parallel. So thereafter first thread of each group will be having array size of 128.Third step is again two threads group together to perform merging sort on their respective array of size 128 resulting in array size of 256 held by first thread of their group in parallel. Further same procedure of two threads grouping and merging technique is followed in parallel until we obtain a single array held by single thread, which is a final sorted array.

## **Fig.3: Data Flow Diagram of Tim Sort**

# Analysis

When the input size is Less than equal to 64, then the time used by MPI, CUDA and sequential program takes the same time. In MPI with more number of processors being created and in CUDA with threads the time taken by the computation DECREASES, but after a certain number of processors is already being created, if more number of processors are created then the computation time INCREASES.

The computation time is totally dependent on number of processes. Number of processes must be created according to the input Size for BETTER execution time.

PARELLEL ALGORITHM ANALYSIS:

1. For input being 10000 as size of the array, we evaluate the time taken based on the different number of processors as shown in Table 1.

**Table 1: For N=10000, number of processes v/s time.**

|  |  |
| --- | --- |
| **N=10000** | |
| **Processes** | **Time** |
| **2** | **0.028327** |
| **4** | **0.030933** |
| **8** | **0.021053** |
| **16** | **0.041526** |
| **32** | **0.024638** |
| **64** | **0.037315** |
| **128** | **0.035545** |

1. For input being 100000 as size of the array, we evaluate the time taken based on the different number of processors as shown in Table 2.

**Table 2: For N=100000, number of processes v/s time.**

|  |  |
| --- | --- |
| **N=100000** | |
| **Processes** | **Time** |
| **2** | **0.240311** |
| **4** | **0.244493** |
| **8** | **0.222464** |
| **16** | **0.229543** |
| **32** | **0.229677** |
| **64** | **0.234422** |
| **128** | **0.232665** |

3.For input being 1000000 as size of the array, we evaluate the time taken based on the different number of processors as shown in Table3.

**Table 3: For N=1000000, number of processes v/s time.**

|  |  |
| --- | --- |
| **N=1000000** | |
| **Processes** | **Time** |
| **2** | **2.119033** |
| **4** | **2.396746** |
| **8** | **2.313191** |
| **16** | **2.349319** |
| **32** | **2.231096** |
| **64** | **2.273086** |
| **128** | **2.297322** |

While in sequential logic, when N=500 the time consumed is 2.59000sec in parallel for 8 processes is 0.006483 sec. So speedup is 380.88.

# results

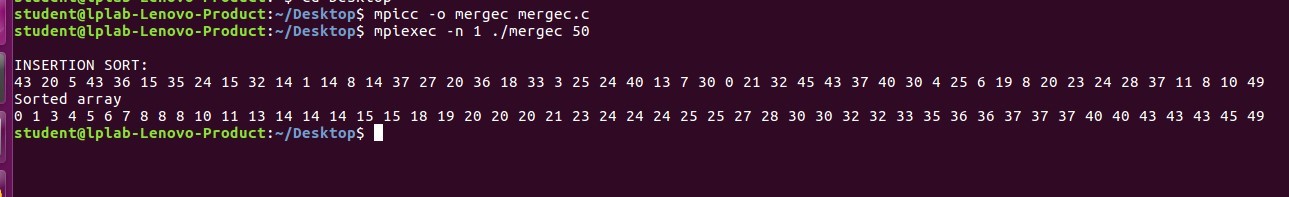
As input size is Less than equal to 64,Tim sort is performed and it undergoes Insertion Sort as shown in Fig.4.

## **Fig.4: Insertion Sort if size is less than 64**

If input size is greater than 64, Tim sort is performed and it undergoes Merge Sort as shown in Fig.5.

# **Fig.5: Merge Sort if size is greater than 64**

# Conclusion

Tim Sort is powerful. It is fast and stable, but perhaps most importantly it takes advantage of real world patterns and utilizes them to build a final product. Tim sort is a parallel hybrid sorting algorithm that takes in the features of merge sort and insertion sort. Merge sort is optimal on huge data set asymptotically, but on small data set overhead occurs. And for smaller data set insertion sort is best to be chosen. For better performance if divide and conquer algorithm is used then for smaller data set best optimal solution is obtained by using insertion sort. So a mixture of merge and insertion sort acts as a good hybrid sorting algorithm thereby allowing Tim sort to have far negligible than O(n log n) comparison, because it takes benefit that sub array is may already be sorted.

References

1. D. E. Knuth. The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching. Addison Wesley Longman Publish. Co.,

Redwood City, CA, USA, 1998

1. T. Peters. Timsort description, accessed june 2015.<http://svn.python.org/projects/python/trunk/Objects/listsort.txt>
2. http://en.wikipedia.org/wiki/Timsort [Accesssed on :23-Sept-2019]
3. <http://stromberg.dnsalias.org/strombrg/sort-comparison> [Accessed on:23-Sept-2019]
4. G. Jost, H. Jin, D. Mey, F. Hatay. Comparing the OpenMP, MPI, andhybrid programming paradigms on an SMP cluster. NAS Technical Report NAS-03-019, November 2003.
5. T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein,“Introduction to Algorithms”. MIT Press, Cambridge, MA, 2ndEdition, 2001.
6. https://media.geeksforgeeks.org/wpcontent/uploads/insertionsort.png[Acccessed On:22-october-2019]
7. https://en.wikipedia.org/wiki/Merge\_sort#/media/File:Merge\_sort\_alg orithm\_diagram.svg [Accessed On:03-October-2019]

1. [↑](#footnote-ref-1)