

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 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 pairwise according to some specific rules. To interpret and examine the merging strategy (meaning that the order in which merge and merge runs are performed) of Tim Sort, we have implemented it in MPI and CUDA environment. Finally, it can be seen the difference in the execution time between serial Tim Sort and parallel Tim sort run in $O(n \log n)$ time .

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

I. INTRODUCTION

Sorting algorithm Tim Sort [1] was designed in 2002 by Tim Peters. It is a hybrid parallel sorting algorithm that combines two different sorting algorithms. This includes insertion sort and merge sort. This algorithm is an effective method for well-defined instructions. The algorithm is concealed as a finite list for evaluating Tim sort function. Starting from an initial state and its information the guidelines define a method that when execution continues through a limited number of next states. The next states are well characterized, at the end providing output and ending at the last completion state [2].

Tim sort works by identifying runs of least two elements. Element runs occur either strictly descending (each element is lesser than its predecessor) or in ascending (each element is greater than or equal to its predecessor) order. At its worst case, it runs at a similar speed of merge sort which means it is unexpectedly very fast. In terms of space, Tim Sort is on the worse end of the spectrum, but the space consideration for most sorting algorithms is highly sparse. Concept of stability is that when array sorted, objects of same value maintain their original order. If Tim sort follows an unstable algorithm, it results in a loss of reliability from first sort when we run the second one.

The following steps for Tim sort include:

- (i) The 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 that are either in strictly descending or in ascending order.
- (iii) If the element runs take place in strictly descending order, reverse operation of Tim sort occurs.
- (iv) If the run is less than 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 the value of the array exceeds the min run values and also keeps merges balanced.

II. IMPLEMENTATION

Before looking into how Tim Sort is implemented parallel, let us discuss some basics of Insertion and Merge Sort.

A. Insertion Sort & Merge Sort

Insertion Sort is one of the fundamental sorting algorithms. First it glance the array and when it finds the element which is not in proper position ,moves it to a position in array which is sorted before as shown in Fig. 1.This sorting algorithm works good in situation like already sorted arrays and smaller arrays. [3]. As we can conclude from Fig.1, that complexity of Insertion Sort is $O(n^2)$ [4]. In Tim Sort when array is already sorted, insertion sort works well.

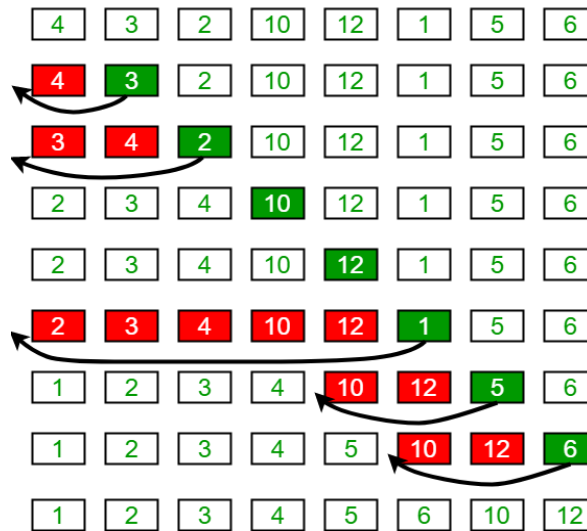


Fig 1.Insertion Sort Example (source [5])

Second Sorting algorithm is Merge Sort whose basic idea is to merge arrays that are sorted. As Divide and conquer method, it splits the array into half and again splits the two arrays into half and so on to get single element. After it starts merging single elements by sorting method as shown in Fig.2 [3]. As we started with dividing large array into single elements and again building the primary array by merging individuals, process runs in $O(n \log n)$ [4] time.

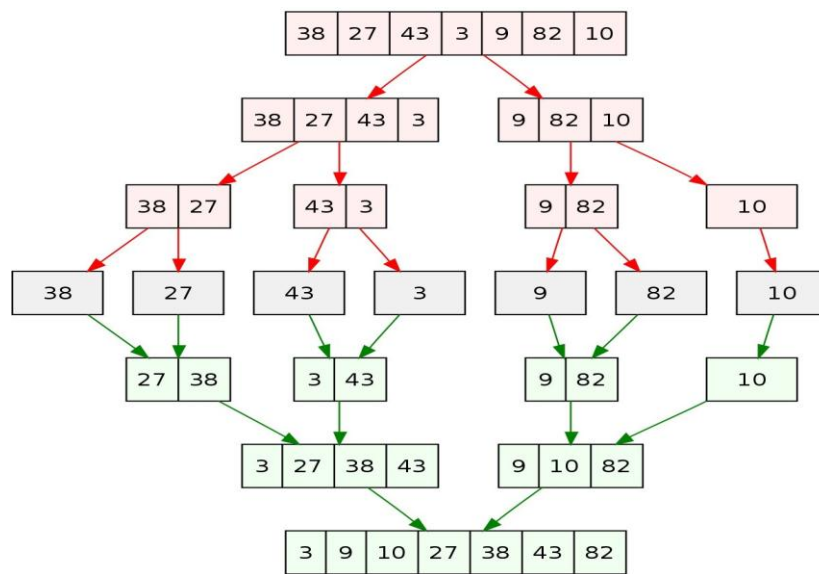


Fig.2: Merge Sort Example (Source:[6])

B. Implementing Parallel Tim Sort

The way to comprehend 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 starts by finding the current element with larger or smaller elements. Later current elements finds its suitable position by comparison. In Merge Sort, we divide an array of elements that are 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 [7]. Note that insertion sort doesn't indulge parallelism while the merge sort can be implemented parallel.

First set a min run size. If the 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.

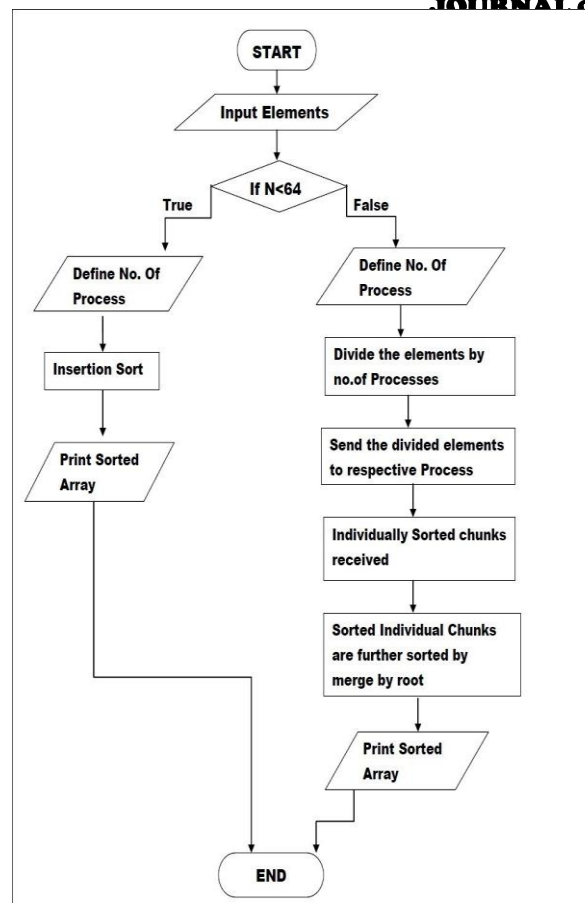


Fig.3: Data Flow Diagram of Tim Sort

Steps to implement parallel Tim sort:

a) First set up a min run size which is a power of 2 (we have taken as 64 as above which insertion sort loses its efficiency) [8]

b) Find size of the array to compare with min run i.e. 64.

c) If the size is less than min run, execute insertion sort as discussed in the previous section.

d) If size is more than the min run then divide the entire array into size of min run and perform insertion sort to each sub arrays.

e) Perform Merge sort on each sub arrays and thus we will get entire array sorted finally.

C. 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 a count of threads. After getting the number of threads, first, divide the entire array of size 64 for each thread to perform insertion sort where each thread performs insertion sort of their array in parallel. In the second step, groups of two threads will perform merging of their respective array elements obtained after insertion sort in parallel. So thereafter the first thread of each group will be having an array size of 128. The third step is again two threads group together to perform merging sort on their respective array of size 128 resulting in an array size of 256 held by the first thread of their group in parallel. Further, the same procedure of two threads grouping and merging technique is followed in parallel until we obtain a single array held by a single thread, which is a final sorted array.

III. ANALYSIS

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

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

PARALLEL ALGORITHM ANALYSIS:

1. For input being 10 000 as the size of the array, we evaluate the time taken based on the different number of processes as shown in Table 1.

Table 1: For N=10 000, the number of processes v/s time.

N=10 000	
Processes	Time
2	0.028327 s
4	0.030933 s
8	0.021053 s
16	0.041526 s
32	0.024638 s
64	0.037315 s
128	0.035545 s

2. For input being 100 000 as the size of the array, we evaluate the time taken based on the different number of processes as shown in Table 2.

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

N=100 000	
Processes	Time
2	0.240311 s
4	0.244493 s
8	0.222464 s
16	0.229543 s
32	0.229677 s
64	0.234422 s
128	0.232665 s

3. For input being 1 000 000 as the size of the array, we evaluate the time taken based on the different number of processes as shown in Table3.

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

Table 3: For N=1 000 000, the number of processes v/s time.

N=1 000 000	
Processes	Time in s
2	2.119033 s
4	2.396746 s
8	2.313191 s
16	2.349319 s
32	2.231096 s
64	2.273086 s
128	2.297322 s

IV. RESULTS

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

```

student@lplab-Lenovo-Product:~/Desktop$ mpicc -o mergec mergec.c
student@lplab-Lenovo-Product:~/Desktop$ mpiexec -n 1 ./mergec 50

INSERTION SORT:
43 20 5 43 36 15 35 24 15 32 14 1 14 8 14 37 27 20 36 18 33 3 25 24 40 13 7 30 0 21 32 45 43 37 40 30 4 25 6 19 8 20 23 24 28 37 11 8 10 49
Sorted array
0 1 3 4 5 6 7 8 8 10 11 13 14 14 14 15 15 18 19 20 20 20 21 23 24 24 24 25 25 27 28 30 30 32 32 33 35 36 36 37 37 37 40 40 43 43 45 49
student@lplab-Lenovo-Product:~/Desktop$

```

Fig.4: Insertion Sort if the array size is less than 64

```

Screenshot
student@lplab-Lenovo-Product:~/Desktop$ mpicc -o mergec mergec.c
student@lplab-Lenovo-Product:~/Desktop$ mpiexec -n 8 ./mergec 500
merge sort
186 19 233 341 354 90 199 131 311 270 470 238 371 108 483 486 58 96 332 83 468 244 17 362 2 135 423 468 109 276 493 148 295 78 489 150 168 40
81 479 311 103 69 34 63 53 20 121 1 204 57 470 448 74 184 302 209 459 271 171 235 264 319 31 342 308 33 10 348 314 342 159 417 411 193 480 316
65 453 170 269 10 492 217 436 176 20 146 488 291 317 75 407 136 106 101 296 139 463 144 305 157 156 74 69 201 406 237 267 360 407 36 222 251 1
6 159 428 478 157 268 121 326 343 28 314 302 481 110 293 444 106 99 102 262 173 23 316 432 260 83 292 20 471 366 271 429 25 51 407 182 171 28
15 408 174 169 389 284 462 334 391 61 436 5 87 311 321 19 423 256 163 295 228 29 419 9 55 470 417 89 494 445 450 9 354 124 178 243 261 492 42
152 54 217 9 493 28 183 12 452 439 27 247 19 56 18 29 463 489 446 53 335 243 355 344 97 479 374 193 240 366 122 244 272 192 106 117 220 289 4
9 24 228 156 124 100 65 142 129 28 483 427 433 318 22 288 14 120 120 240 165 212 459 139 309 231 331 415 349 404 204 330 428 284 339 52 236 25
47 217 284 30 144 218 349 167 358 215 139 478 456 156 191 415 295 0 498 127 267 199 31 323 30 311 107 221 216 344 477 263 61 113 293 58 331 4
4 77 42 210 68 20 166 224 63 433 19 415 431 498 182 131 381 5 13 45 465 86 261 309 63 24 222 176 169 132 360 16 61 402 226 129 274 244 353 338
177 225 253 460 75 436 91 309 293 104 354 258 190 467 419 105 343 142 134 12 126 346 380 188 248 106 317 374 202 23 212 231 248 466 192 175 25
283 484 47 240 190 306 282 9 225 240 352 219 374 217 198 220 97 386 320 56 55 194 258 430 259 490 178 77 34 206 331 317 42 230 409 233 388 44
242 466 284 447 37 10 164 235 82 261 473 254 317 381 300 428 311 59 270 342 136 304 48 319 473 90 50 235 175 290 279 418 256 415 217 294 277 3
1 381 211 494 355 465 164 236 265 92 399 177 214 241 313 18 289 485 343 232 387 78 407 177 209 177 286 476 394 80 253 275 461 464 270 168 281
86 256 399 230 156 76
This is the sorted array: 0 0 0 0 0 1 2 5 5 8 9 9 9 9 10 10 10 12 12 13 14 15 16 17 18 18 19 19 19 19 20 20 20 20 22 23 23 24 24 25 27 28 28 2
28 29 29 30 30 31 31 33 34 34 36 37 40 42 42 44 45 47 47 48 50 51 52 53 53 54 55 55 56 56 57 58 58 59 61 61 61 63 63 63 65 65 68 69 69 74 74
5 75 77 77 78 78 80 82 83 83 86 87 89 90 90 91 92 96 97 97 99 100 101 102 103 104 105 106 106 106 106 106 107 108 109 110 113 117 120 120 121
21 122 124 124 126 127 129 129 129 131 131 132 134 135 136 136 139 139 139 142 142 144 144 146 148 150 152 156 156 156 157 157 159 163 164
164 165 166 167 168 168 169 169 170 171 171 173 174 175 175 176 176 177 177 177 178 178 182 182 183 184 186 188 188 190 190 191 192 192 193 194
194 198 199 199 201 202 204 204 206 209 209 210 211 212 212 214 215 216 217 217 217 217 217 218 219 220 220 221 222 222 224 225 225 226 228 2
8 230 231 231 232 233 233 235 235 235 236 236 237 238 240 240 240 240 241 242 243 243 244 244 244 247 248 248 251 253 253 254 254 256 256 256
56 258 258 259 260 261 261 261 262 263 264 265 267 267 268 269 270 270 271 271 272 274 275 276 277 279 281 281 282 283 284 284 284 284 286
286 288 289 289 290 291 292 293 293 293 294 295 295 295 296 300 302 302 304 305 306 308 309 309 309 311 311 311 311 311 313 314 314 316 316 31
317 317 317 318 319 319 320 321 323 326 330 331 331 331 332 334 335 338 339 341 342 342 342 342 343 343 343 344 344 346 348 349 349 352 353 354 3
4 354 355 355 358 360 360 362 366 366 371 374 374 374 380 381 381 381 381 386 387 388 389 391 394 399 402 404 406 407 407 407 407 408 409 411
15 415 415 415 417 417 418 419 419 423 423 427 428 428 428 429 429 430 431 432 433 433 436 436 436 439 444 445 446 447 448 450 452 453 456 459
459 460 461 462 463 463 464 465 465 466 466 467 468 468 470 470 470 471 473 473 476 477 478 478 479 479 480 481 483 483 484 485 486 488 489 48
490 492 492 493 493 494 494 494 498 498 0: 8 processors; 0.006605 secs

```

Fig. 5. Merge Sort when the array 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.

V. CONCLUSION

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 the 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 subarray is may already be sorted.

References

- [1] <http://en.wikipedia.org/wiki/Timsort> [Accessed on:23-Sept-2019]
- [2] <http://bugs.python.org/file4451/timsort.txt> [Accessed On:04-June-2020]
- [3] 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

- [4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, "Introduction to Algorithms". MIT Press, Cambridge, MA, 2nd Edition, 2001.
- [5] <https://media.geeksforgeeks.org/wpcontent/uploads/insertionsort.png> [Accessed On: 22-october-2019]
- [6] https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg [Accessed On: 03-October-2019]
- [7] <http://stromberg.dnsalias.org/strombrg/sort-comparison> [Accessed on: 23-Sept-2019]
- [8] S. Rani, "To Solving Hybrid Approach by using Tim Sort Algorithm and Comparison with Quick Sort Algorithm", in MIT International Journal of Computer Science and Information Technology, 2015, pp. 66-70.