# CS410 Assignment 1

**Mandar Deshpande**
**210010014**

## How does my program work?

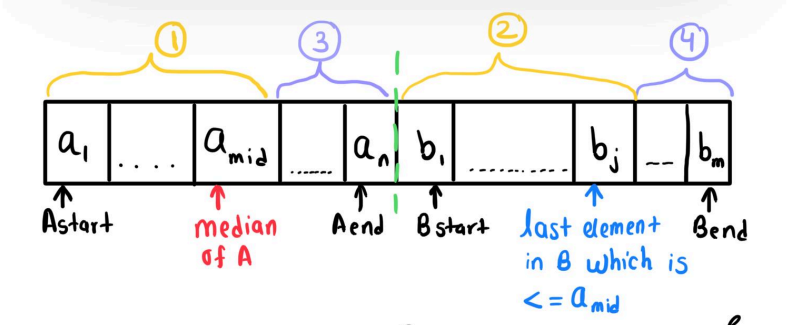
I wrote the program after taking an idea from the 8th page of this pdf Cilk, Matrix Multiplication, and Sorting Lecture Summary 1 Parallel Processing With Cilk.

**parallelMergeSort():**

I used the same template that was provided. As the sorting of the left half and right half of the vector are independent of each other I wrote **cilk_spawn** before the first recursive call. After the two recursive calls, a **cilk_sync** is required, because before merging, both the halves must be sorted. To avoid large granularity, I have added an if condition which checks the size of the vector and whether the cutoff is reached or not. When two halves are sorted, the program calls the *parallelMerge* function.

**parallelMerge():**

Let's call the first sorted half as *A* and second sorted half as *B.* In this function, *[Astart…Aend*] represents the first and last index of *A*, while *[Bstart…Bend*] represents the first and last index of *B* respectively. I am calculating the median of ***A,*** call it *amid.* Using binary search from C++ STL library, I am finding the last element **less than or equal** to *amid* inside *B* (let's call that index *ind*). If no such index is found or when both *A* and *B* are sufficiently smaller, I am calling the *SerialMerge* function that was provided. Otherwise, all elements <=*amid* inside *A* and elements with index <=*ind* inside *B* can be **merged independently** of all elements >amid from *A* and elements with index>ind from *B*. So, I wrote **cilk_spawn** before the first recursive call. This is possible because *A* and *B* are already sorted. The diagram below explains it well:




→ We can merge ① & ② independently of ③ & ④ (∵ no element in ③ & ④ is less than any element in ① & ②).

→ Before calling parallelMerge function on ③ & ④, we need to increment the value of tmpIdx by length of (①+②)

## How does the program exploit parallelism?

Parallel merge sort follows divide and conquer strategy in both the functions. I am using this strategy to make sure that tasks become small enough to solve straightforwardly. Once the array is divided into smaller pieces, each piece can be sorted in parallel. Different threads **can** work on different parts of the array simultaneously. As explained in the previous section, two sorted halves can be merged in parallel which also exploits parallelism. Additionally, I added parallelism when we are calling *Init* function and inside *validate* function.

## Where did I choose not to exploit parallelism?

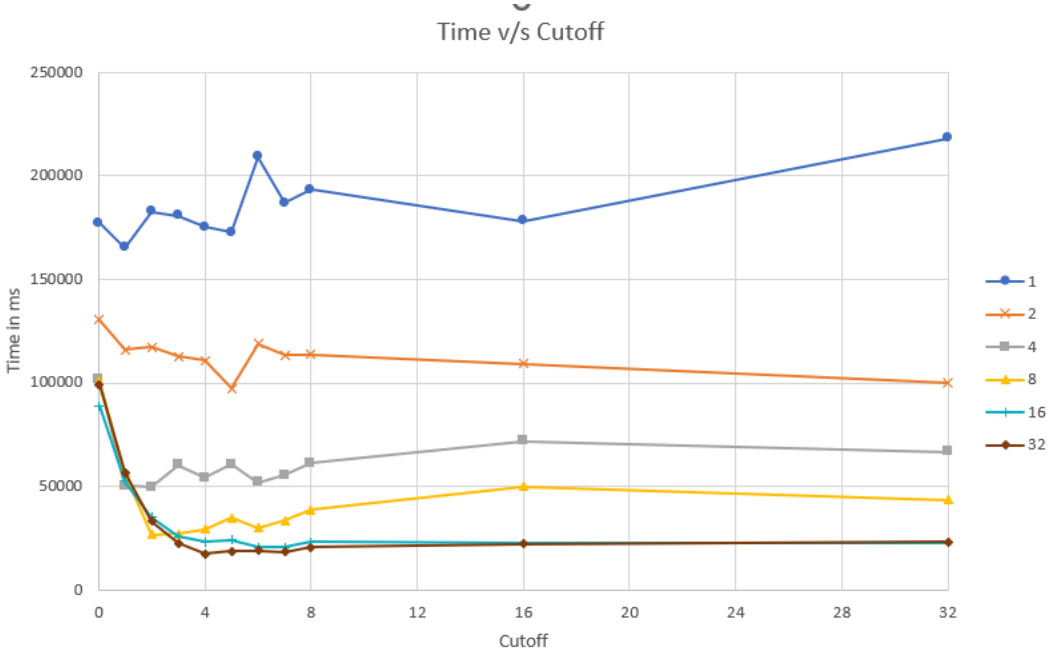
Inside the Init function, there is a for loop with a maximum of 100000 iterations doing minimal work. I didn't parallelize this loop as cost due to spawns will kill the parallelism for such a small task.

## How is the parallel work synchronized?

When merging the sorted subarrays, synchronization is needed to ensure that the elements are merged correctly. Typically, this involves comparing elements from the two subarrays being merged and determining the correct order based on the values of these elements. Also at the end of parallelMerge function, there is an implicit cilk_sync. Also, inside the *Validate* function, I used reducer to ensure that there are no data races.

## Experiment Results

1. The graph below is the time taken to sort a vector of **2 billion** elements. The numbers on the right of the diagram denote the number of workers.



| CUTOFF | CILK_NWORKERS | | | | | |
|---|---|---|---|---|---|---|
| | 1 | 2 | 4 | 8 | 16 | 32 |
| 0 | 177146 | 130415 | 101507 | 100738 | 88902 | 99363 |
| 1 | 165179 | 115881 | 50139 | 56559 | 52593 | 56847 |
| 2 | 182782 | 117378 | 49673 | 26848 | 35217 | 33248 |
| 3 | 180849 | 112721 | 60191 | 27001 | 25857 | 22434 |
| 4 | 175235 | 110985 | 54093 | 29502 | 23079 | 17324 |
| 5 | 172738 | 97259 | 60596 | 34897 | 24209 | 18701 |
| 6 | 209218 | 118799 | 51950 | 29905 | 20529 | 19145 |
| 7 | 186909 | 113372 | 55368 | 33633 | 20706 | 18260 |
| 8 | 193420 | 113682 | 61221 | 38587 | 23267 | 20795 |
| 16 | 178256 | 109192 | 71779 | 50097 | 22474 | 22205 |
| 32 | 218279 | 99954 | 66822 | 43587 | 22991 | 23137 |

Observation: when the cutoff value is more, the process with 16 CILK_NWORKERS performs almost the same as the process with 32 CILK_NWORKWERS.

2. My program sorts an array of 2 billion integers in less than a minute with 32 CILK_NWORKERS.

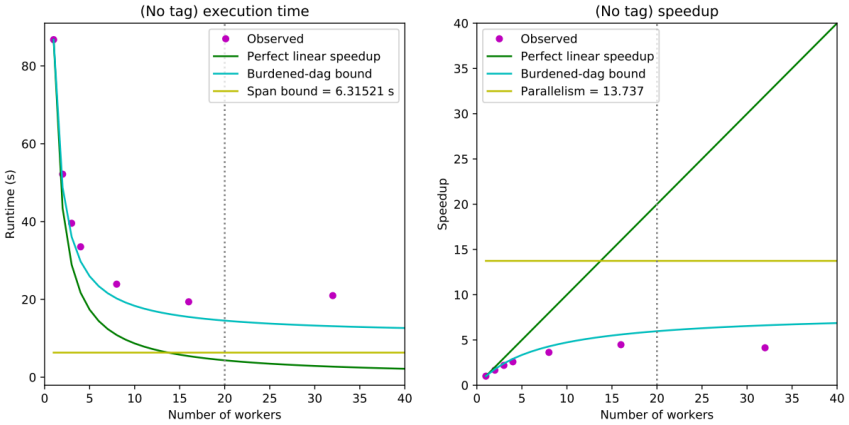
```
c210010014@iitadmin:~/cs410assignment1-Mandar1511$ make CILK_NWORKERS=32 runSortCilk SIZE=2000000000 CUTOFF=5
CILK_NWORKERS=32 ./sortCilk 2000000000 5

vector size=2000000000

millisec=18999
Mistakes=0
```

3. The image below shows that my program has no data races.

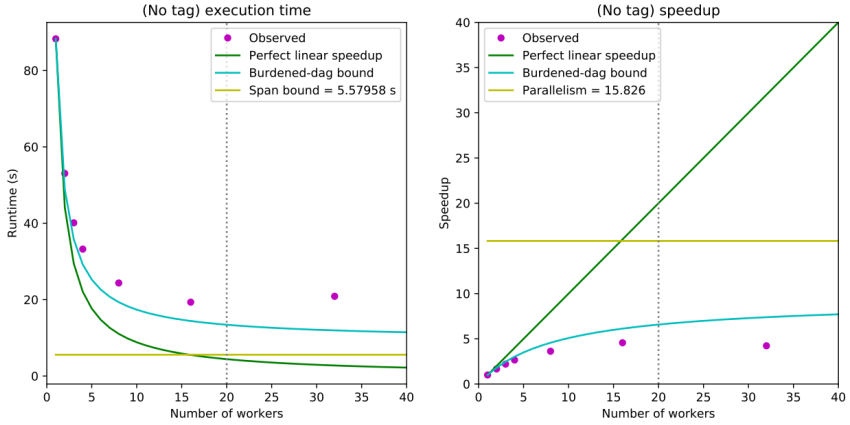
```
c210010014@iitadmin:~/cs410assignment1-Mandar1511$ make runSortSan SIZE=100000000 CUTOFF=6
CILK_NWORKERS=1 ./sortSan 100000000 6
Running Cilksan race detector.

vector size=100000000

millisec=138225
Mistakes=0

Cilksan detected 0 distinct races.
Cilksan suppressed 0 duplicate race reports.
```

4. **CilkScale** observations (array size: **1 billion**), time is total time taken to execute the program (initialization + sort + validation).

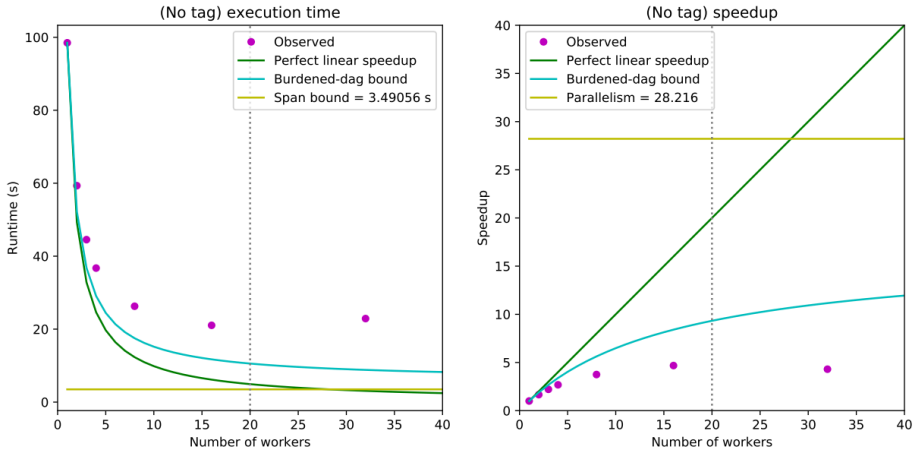
## CUTOFF=4



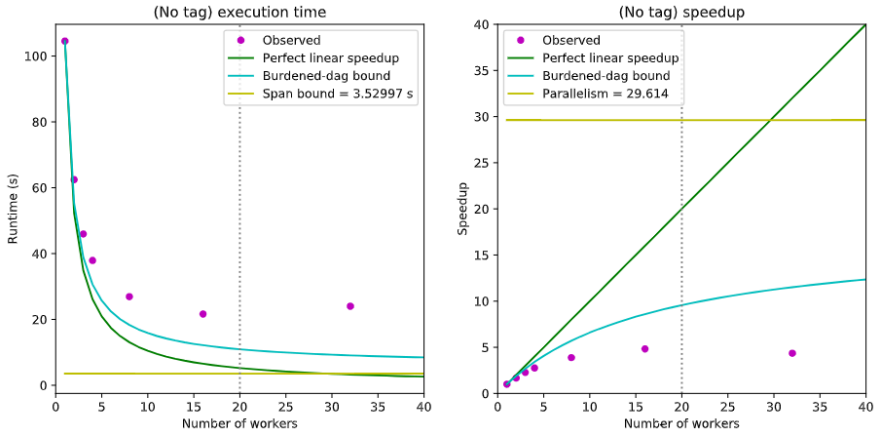

## CUTOFF=5




## CUTOFF=10



**CUTOFF=16**




5. **Parallel efficiency** Observations:

```
c210010014@iitadmin:~/cs410assignment1-Mandar1511$ make runSortSerial SIZE=2000000000 CUTOFF=2
./sortSerial 2000000000 2

vector size=2000000000

millisec=192942
Mistakes=0
```

**Serial** execution time : **192942** milliseconds