Parallel Sorting

Problem Statement:

Task is to implement a parallel sorting algorithm such that each partition of the array is sorted in parallel. You should consider two different schemes for deciding whether to sort in parallel.

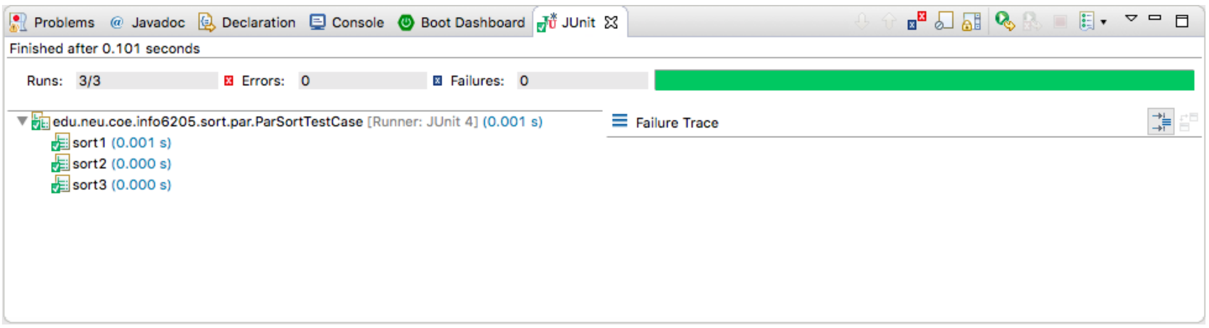
A cutoff (defaults to, say, 1000) which you will update according to the first argument in the command line when running. It's your job to experiment and come up with a good value for this cutoff. If there are fewer elements to sort than the cutoff, then you should use the system sort instead.

Recursion depth or number of available threads. Using this determination, you might decide on an ideal number (t) of separate threads (stick to powers of 2) and arrange for that number of partitions to be parallelized (by preventing recursion after depth of lg t is reached).

An appropriate combination of these.

There is a Main class and the ParSort class in the sort.par package of the INFO6205 repository. The Main class can be used as is but the ParSort class needs to be implemented where you see "TODO..."

Snapshot of Unit test case:



```
Finished after 0.101 seconds
Runs: 3/3    Errors: 0    Failures: 0
edu.neu.coe.info6205.sort.par.ParSortTestCase [Runner: JUnit 4] (0.001 s)
    sort1 (0.001 s)
    sort2 (0.000 s)
    sort3 (0.000 s)
```

Considered conditions:

- The array is shuffled for optimum average value
- The cutoff value is calculated by trial method-running each method in increments
- The sort method is run 100 times for each pair of array size and cut off
- The number of threads created are calculated

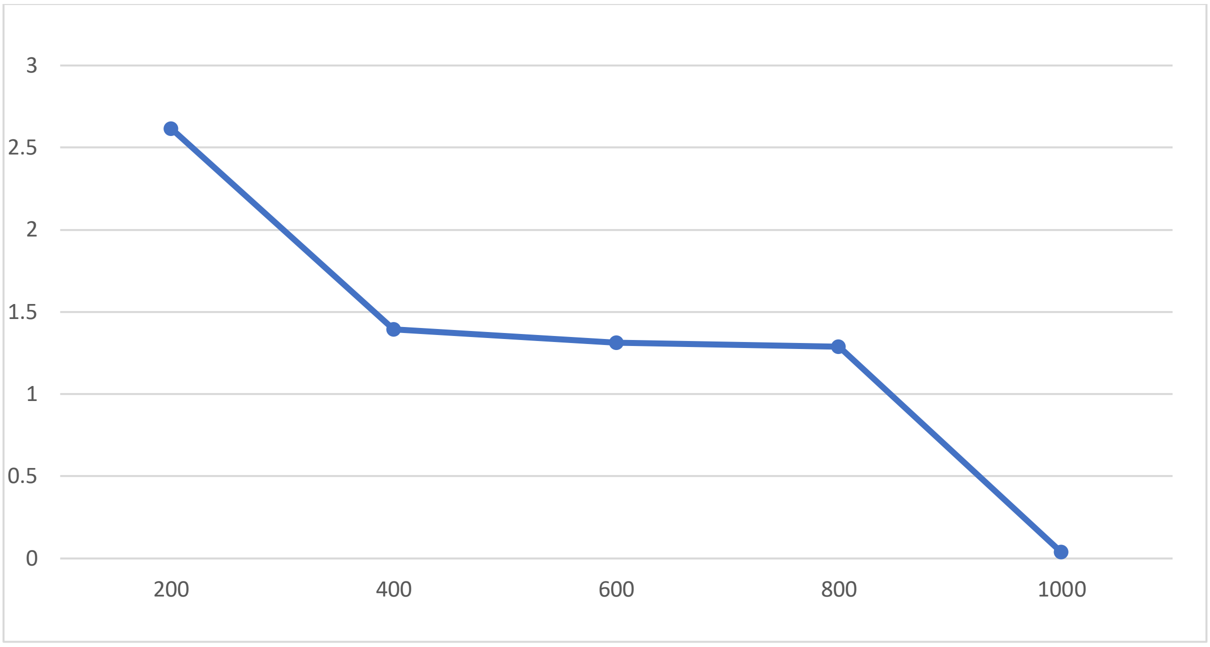
Observations:

Bold ones are optimum values

For array size=1000 & cut-off = 200 to 1000

Cut-off	Array Size	Time	Threads	Cut-off/Array-size
200	1000	2.615388	14	0.2
400	1000	1.393961	6	0.4
600	1000	1.312244	2	0.6
800	1000	1.287645	2	0.8
1000	**1000**	**0.038011**	**0**	1

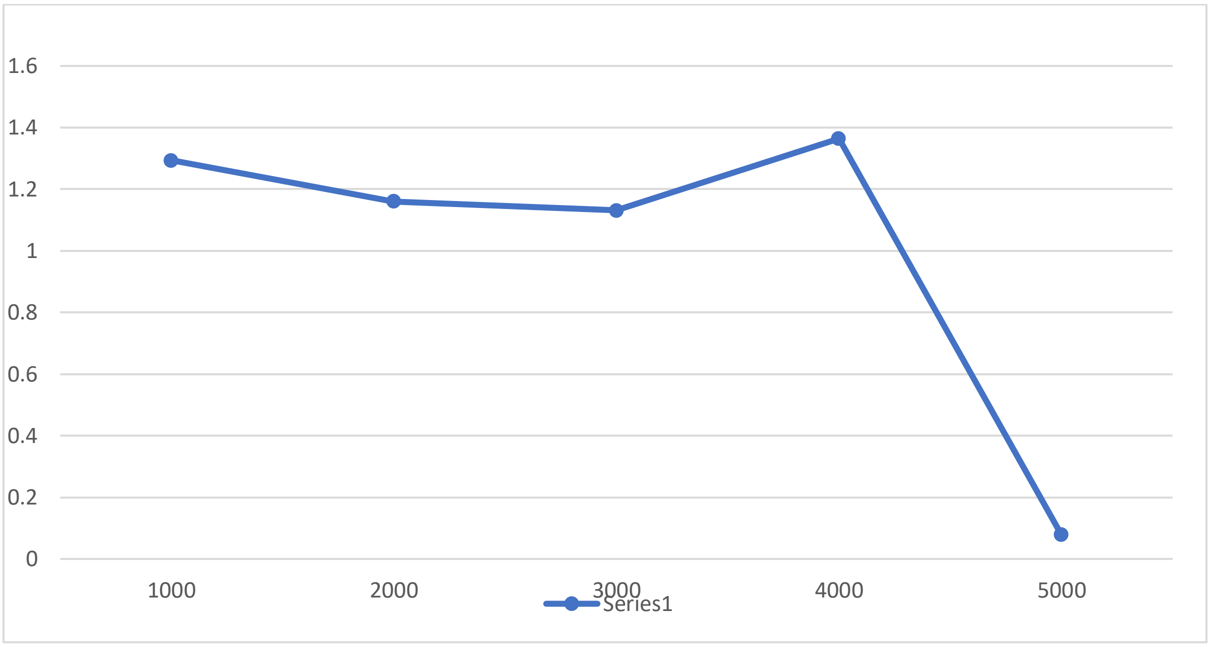
cut-off vs time



For array size=5000 & cut-off = 1000 to 5000

Cut-off	Array Size	Time	Threads	Cut-off/Array-size
1000	5000	1.293592	14	0.2
2000	5000	1.159844	6	0.4
3000	5000	1.13064	2	0.6
4000	5000	1.364129	2	0.8
5000	**5000**	**0.078912**	**0**	1

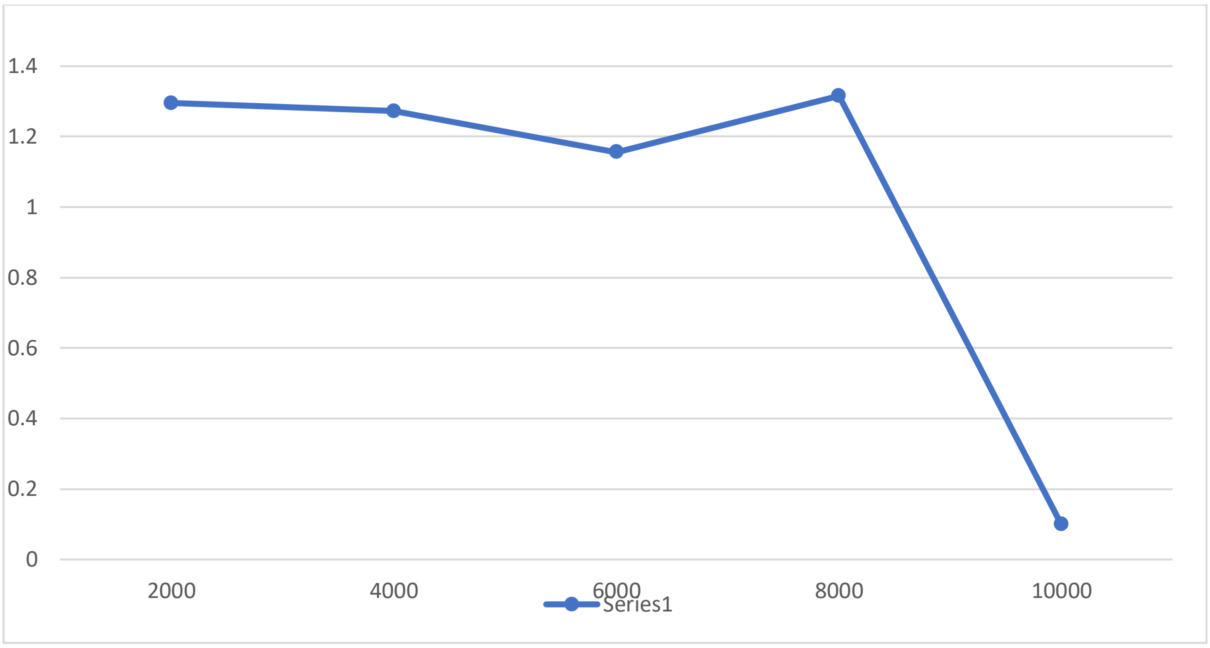
cut-off vs time



For array size=10000 & cut-off = 2000 to 10000

Cut-off	Array Size	Time	Threads	Cut-off/Array-size
2000	10000	1.295511	14	0.2
4000	10000	1.272464	6	0.4
6000	10000	1.156375	2	0.6
8000	10000	1.315596	2	0.8
10000	**10000**	**0.099387**	**0**	1

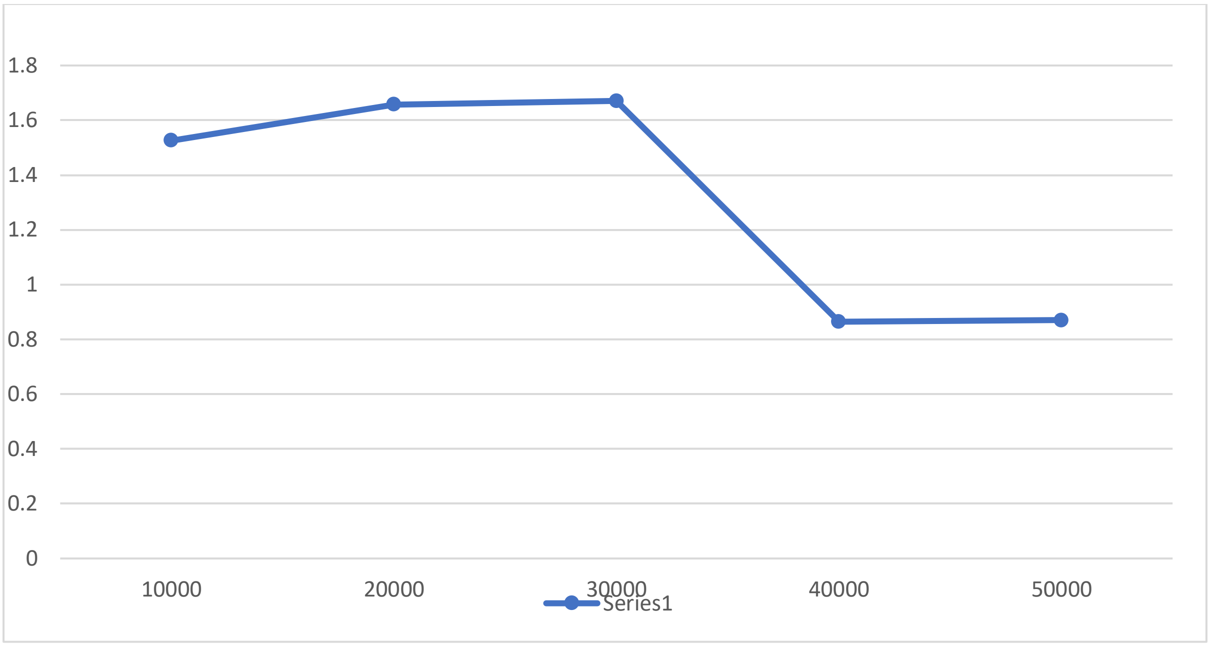
cut-off vs time



For array size=50000 & cut-off = 10000 to 50000

Cut-off	Array Size	Time	Threads	Cut-off/Array-size
10000	50000	1.526529	14	0.2
20000	50000	1.657699	6	0.4
30000	50000	1.670792	2	0.6
40000	**50000**	**0.86438**	**2**	**0.8**
50000	50000	0.87056	0	1

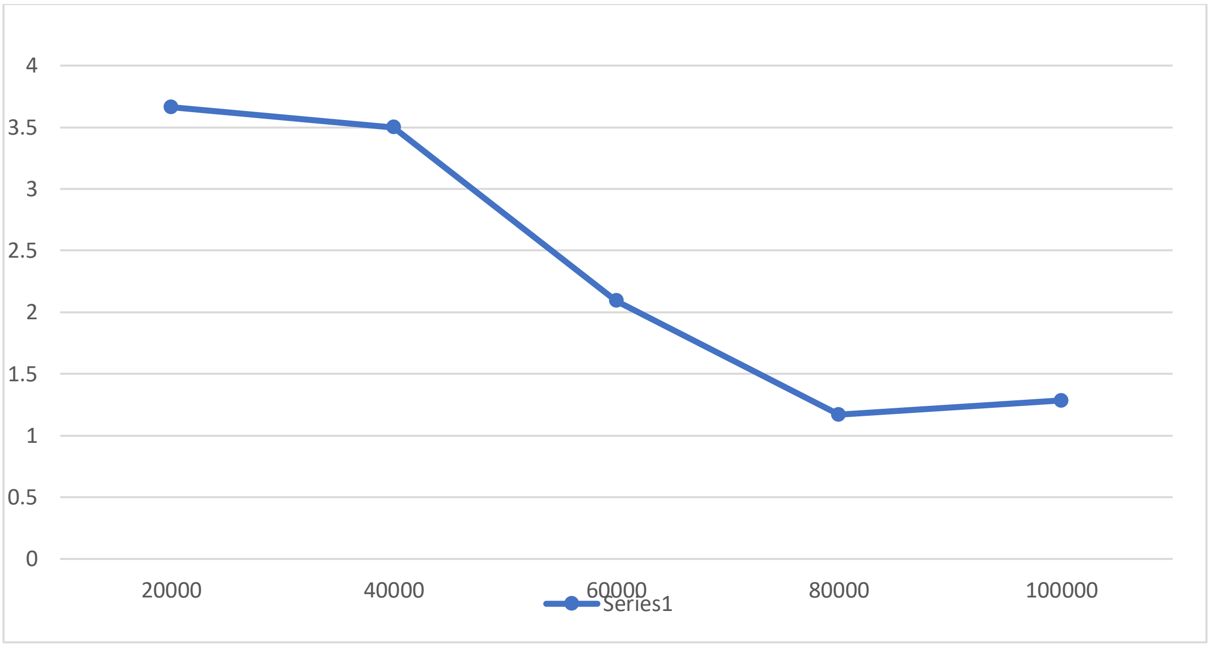
cut-off vs time



For array size=100000 & cut-off = 20000 to 100000

Cut-off	Array Size	Time	Threads	Cut-off/Array-size
20000	100000	1.792245	14	0.2
40000	100000	2.011754	6	0.4
60000	100000	2.09259	6	0.6
80000	**100000**	**1.170072**	**2**	**0.8**
100000	100000	1.284239	0	1

cut-off vs time



Conclusion:

1. For the small size arrays, the cut off is same as array size and thus only system sort is invoked. Thus, parallel threads/depth is 0.

2. For small arrays, System Sort works at optimum level because the thread creation time being skipped.

3. If array size is large then threads help in optimization.

4. Recursion depth or available threads is calculated based on the threads created during parallel sort and the ideal number for this parameter is calculated just before system sort is called.

5. The optimum value is highlighted (in bold) in the table for each array size.

6. From above graphs and table, we can easily deduce that 0.8 is the optimum value of threads/depth for cutoff/array size for larger arrays