APD - QuickSort

QuickSort - Sequential Solution

1. Statement

In the quicksort algorithm, a special element called “pivot” is first selected and the array or list in question is partitioned into two subsets. The partitioned subsets may or may not be equal in size. The partitions are such that all the elements less than the pivot element are towards the left of the pivot and the elements greater than the pivot is at the right of the pivot. The Quicksort routine recursively sorts the two sub-lists. Quicksort works efficiently and also faster even for larger arrays or lists.

2. Classes

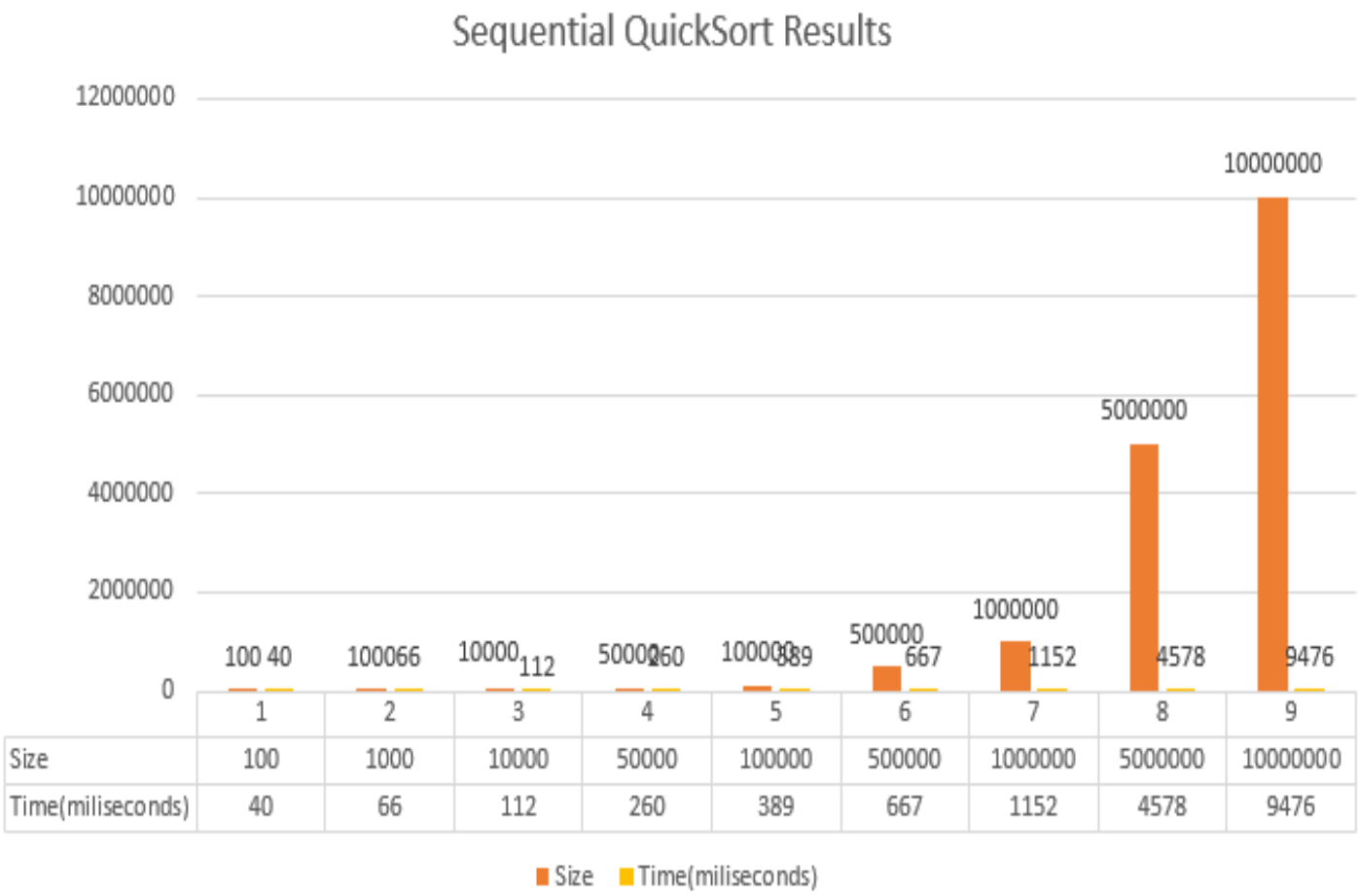
A total of three classes are used for the sequential solution of QuickSort:

- **QuickSort** - takes the array to be sorted as a parameter. This class has two methods:
  - **partition** - This function takes last element as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot;
  - **quicksort -** This is the main function that implements the quicksort algorithm. It finds the partition index and recursively calls quicksort for the two subparts created.

- **FileManager** - takes the size of the file as parameter (how many values are in that file):
  - **ReadFromFile** - returns an array containing the values read from the file that has the name “input(size).txt”;
  - **WriteToFile -** writes the values from the array given as parameter into the file with the name “output(size).txt”;

- **Main** - in the main program:
  - **FileManager** object is created in order to get a array with (size) elements;
  - **QuickSort** object is created in order to sort the array;
  - The FileManager object is used to put the values from the sorted array into a file;
  - The time is calculated from the start of the program to the end using **Instant** and **Duration** classes.

3. Table and Chart

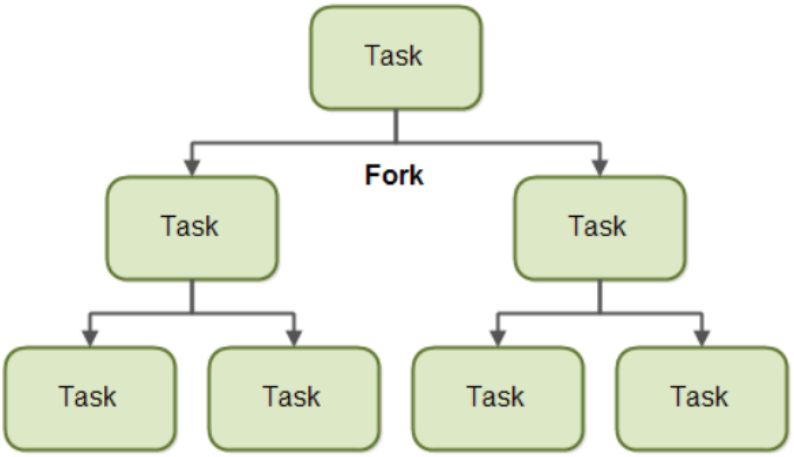


The running time increases with increasing input size.

QuickSort - Parallel Solution I

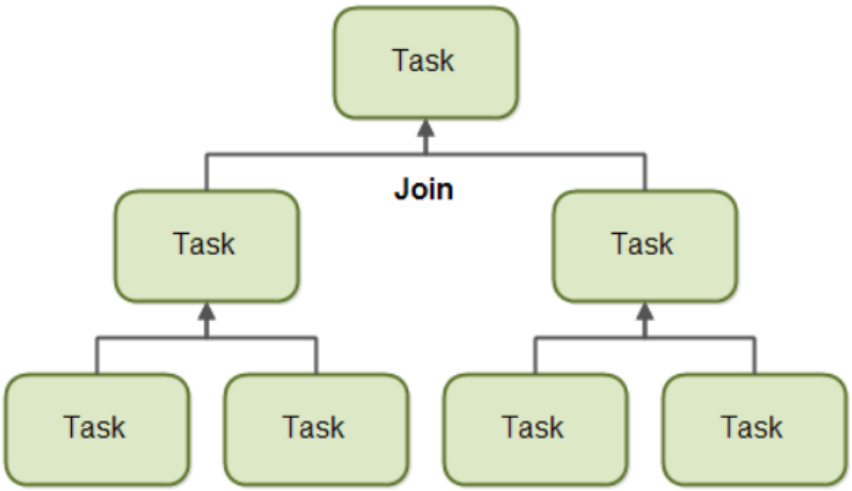
- **Java Threads - ForkJoinPool**
  Explanation: The fork and join principle consists of two steps which are performed recursively. These two steps are the fork step and the join step.

  Fork
  A task that uses the fork and join principle can *fork* (split) itself into smaller subtasks which can be executed concurrently. By splitting itself up into subtasks, each subtask can be executed in parallel by different CPUs, or different threads on the same CPU.



Join

When a task has split itself up into subtasks, the task waits until the subtasks have finished executing. Once the subtasks have finished executing, the task may join (merge) all the results into one result.



ForkJoinPool

The ForkJoinPool is a special thread pool which is designed to work well with fork-and-join task splitting.

1. Statement

- The numbers are read from the file using parallelStream and then put in an array;
- The main thread calls the quicksort method;
- The method partitions the array and checks for the number of current threads;
- New threads are called for the next step using the same parallel method;
- Use the single normal quicksort method;
- The sorted array is printed into a file;
- The duration of the program is displayed in the console.

2. Classes

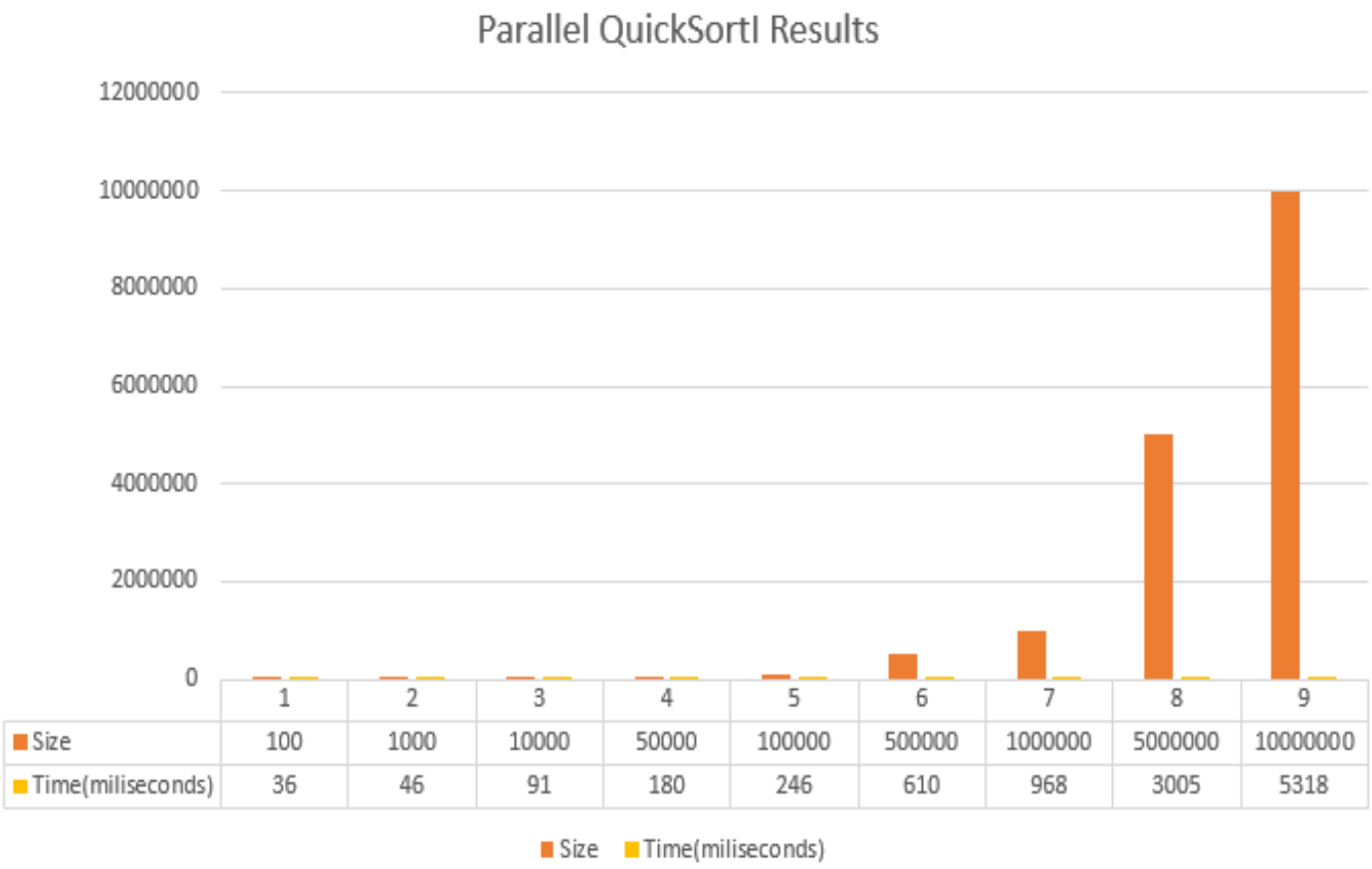
A total of three classes are used for the parallel solution of QuickSort:

- **ParallelQuickSort** - extends RecursiveTask. This class has two methods:
    - **partition** - This function takes a random number as pivot, places the pivot element at its correct position in sorted array, and places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot;
    - **compute -** This is the main function that implements the quicksort algorithm. It finds the partition index and calls new threads for sorting the subparts of the initial array divided by the partition index. Each subpart is executed concurrently. Once the subtasks have finished executing, the task may join (merge) all the results into one result.

- **FileManager** - takes the size of the file as parameter (how many values are in that file):
    - **ReadFromFile** - reads the numbers from the file that has the name “input(size).txt” using parallelStream and returns an array containing the values;
    - **WriteToFile -** writes the values from the array given as parameter into the file with the name “output(size).txt”;

- **Main -** in the main program:
    - **FileManager** object is created in order to get a array with (size) elements;
    - **ForkJoin ThreadPool** to keep thread creation as per resources
    - Start the first thread in fork join pool for range 0, n-1
    - The FileManager object is used to put the values from the sorted array into a file;
    - The time is calculated from the start of the program to the end using **Instant** and **Duration** classes.

3. Table and Chart



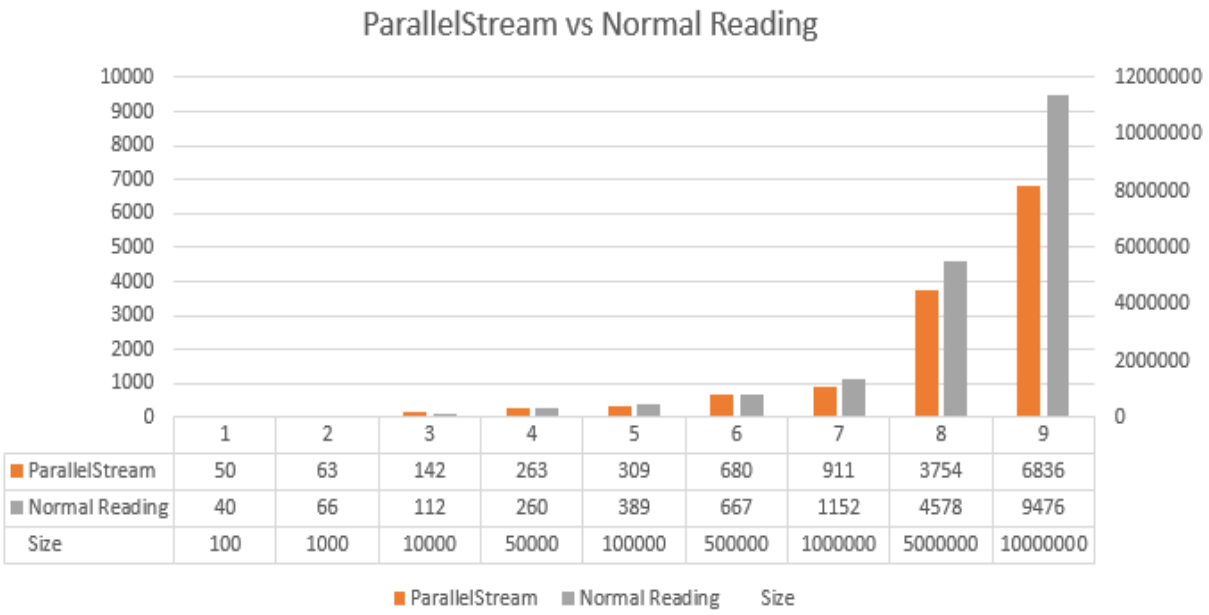
	1	2	3	4	5	6	7	8	9
Size	100	1000	10000	50000	100000	500000	1000000	5000000	10000000
Time(miliseconds)	36	46	91	180	246	610	968	3005	5318

Links:

- [Fork/Join Framework in Java]
- [Java Fork and Join using ForkJoinPool]
- [QuickSort using Multithreading]

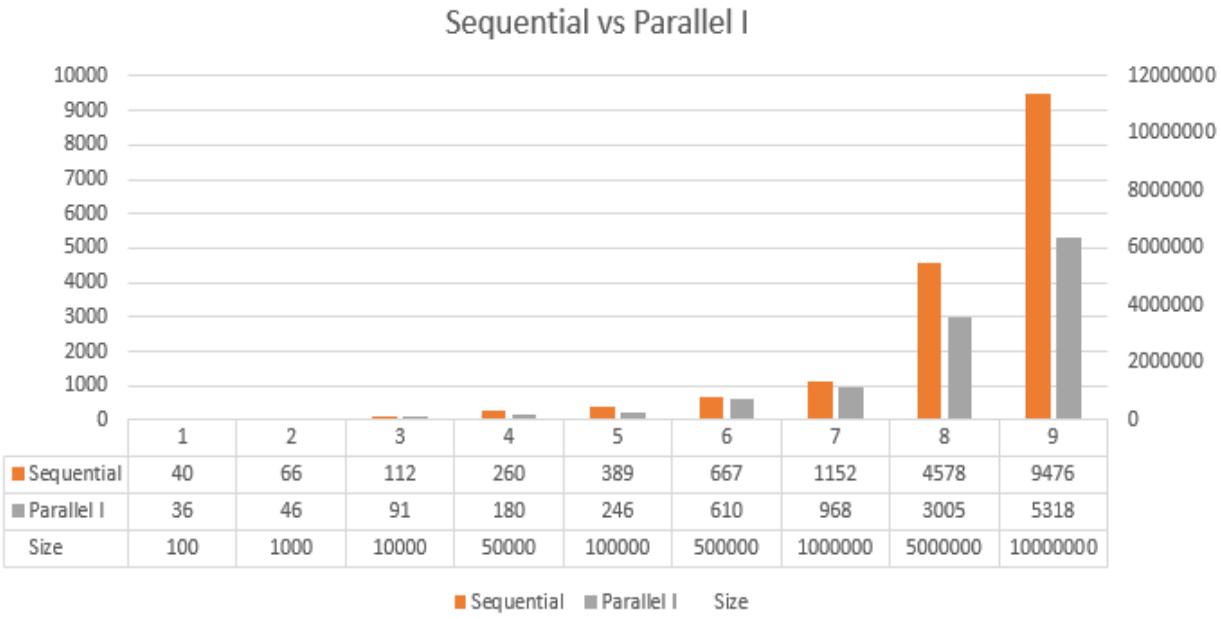
Chart and table for comparing sequential and parallel results

ParallelStream vs Normal Reading



	1	2	3	4	5	6	7	8	9
ParallelStream	50	63	142	263	309	680	911	3754	6836
Normal Reading	40	66	112	260	389	667	1152	4578	9476
Size	100	1000	10000	50000	100000	500000	1000000	5000000	10000000

Sequential vs Parallel

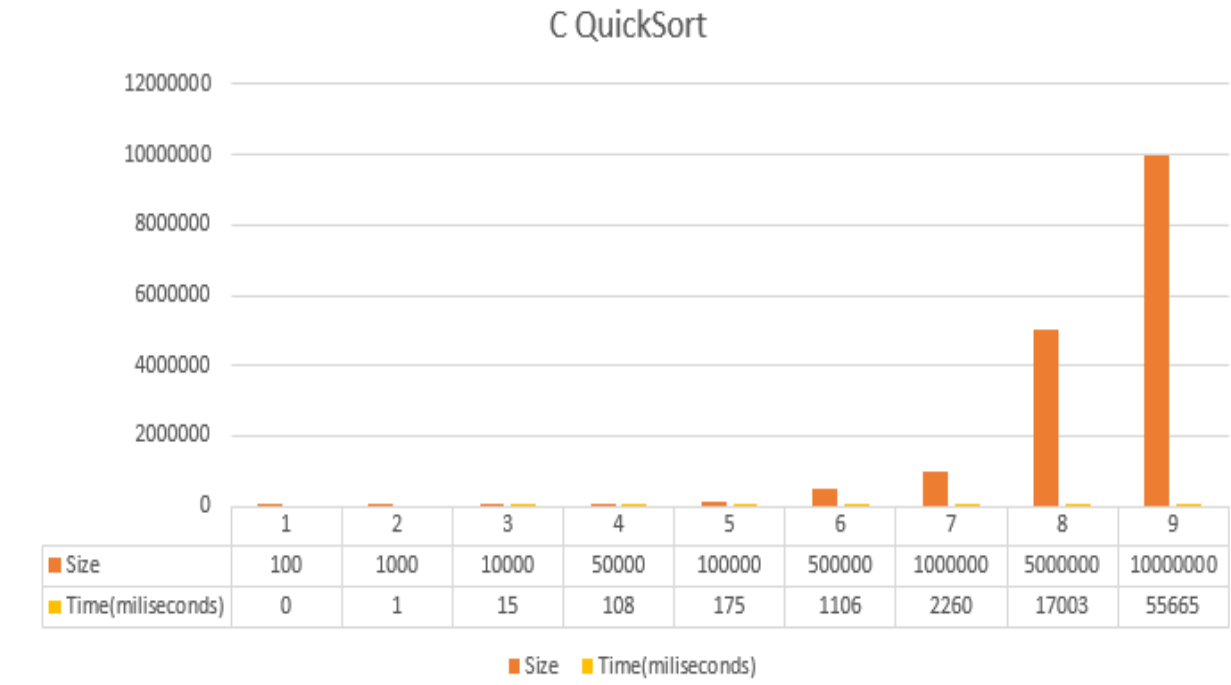


	1	2	3	4	5	6	7	8	9
Sequential	40	66	112	260	389	667	1152	4578	9476
Parallel I	36	46	91	180	246	610	968	3005	5318
Size	100	1000	10000	50000	100000	500000	1000000	5000000	10000000

Sequential QuickSort C

The implementation of C Sequential QuickSort is the same as the implementation in Java language.

The results are the following:



	1	2	3	4	5	6	7	8	9
Size	100	1000	10000	50000	100000	500000	1000000	5000000	10000000
Time(miliseconds)	0	1	15	108	175	1106	2260	17003	55665

C MPI QuickSort

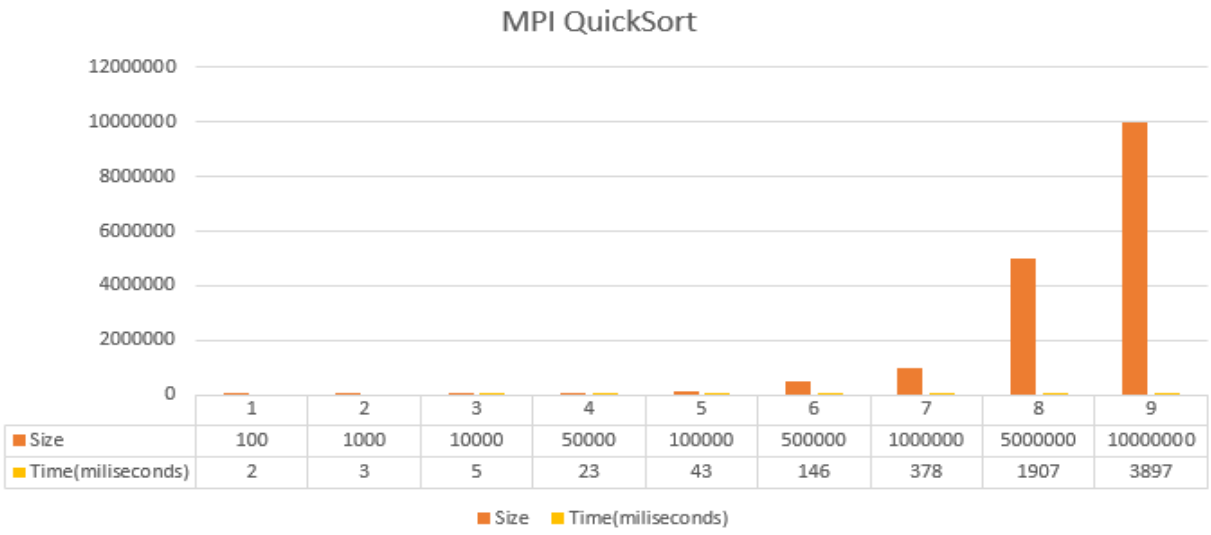
For the implementation of MPI QuickSort two methods are used:

- **quicksort -** the same implementation as for the sequential solution, the only difference being that the partition method is included in the quicksort method;
- **merge -** merge two sorted arrays into a single sorted array.

The main function contains the MPI utilities. The following MPI methods are used:

- **MPI_Bcast -** for broadcasting the size of the array;
- **MPI_Scatter -** for retrieving the chunk for each process;
- **MPI_Send -** for sending the chunk to a certain process;
- **MPI_Recv -** for receiving the chunks.

The results are the following:



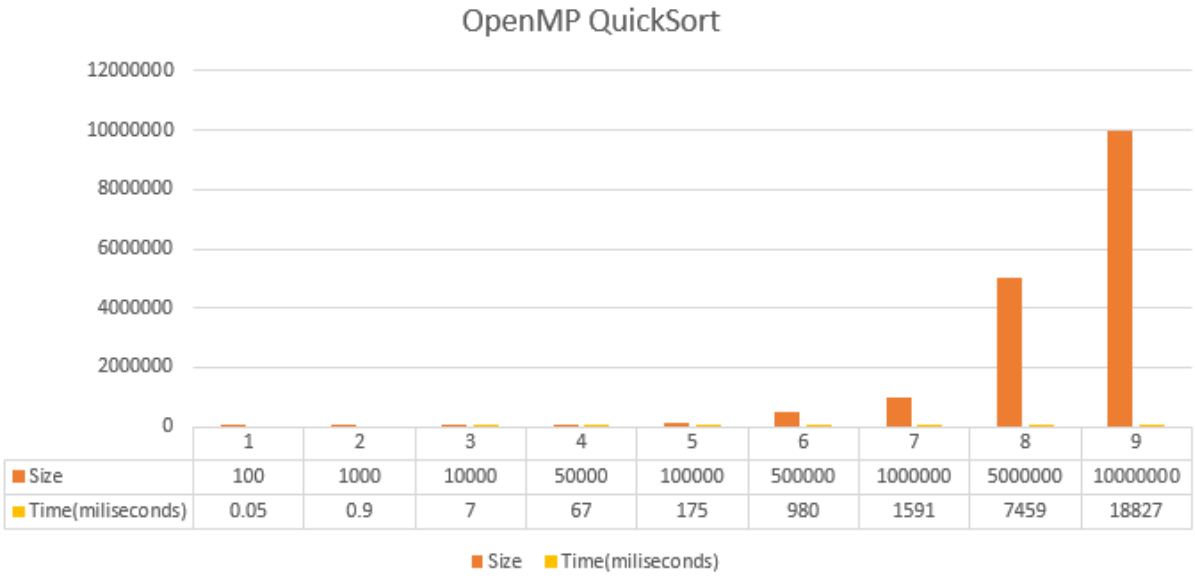
OpenMP QuickSort

For the OpenMP solution the same methods are used: **quicksort** and **partition**, the only difference being that the quicksort partitioning uses:

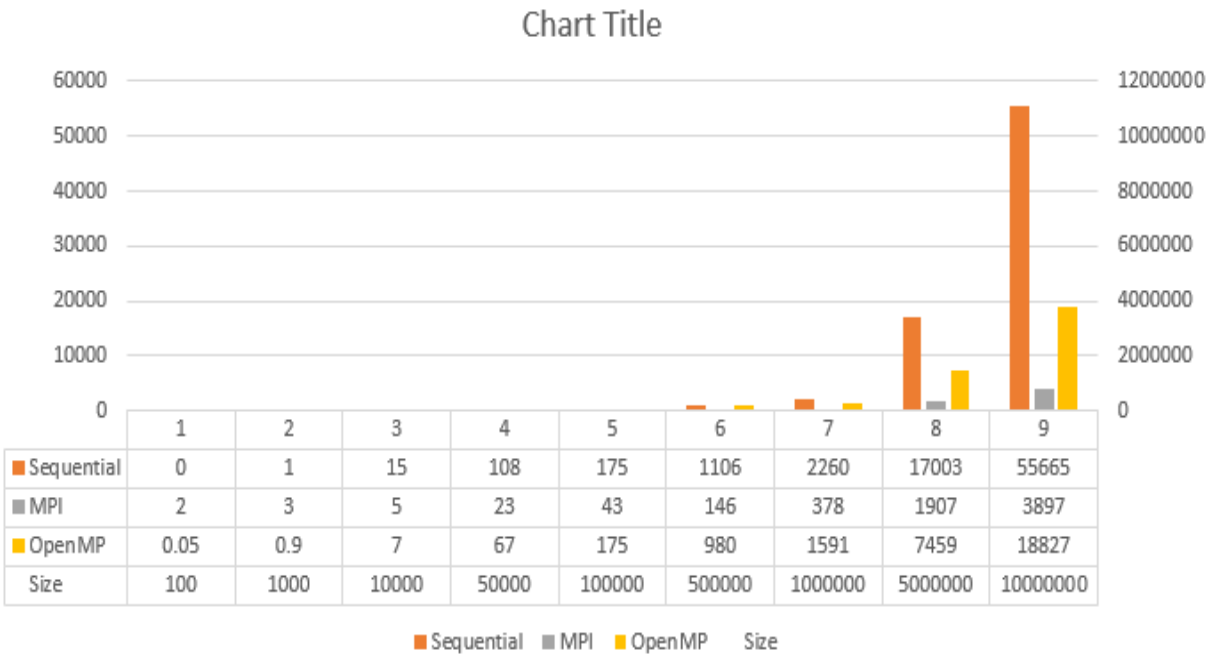
- **#pragma omp parallel sections**
- **#pragma omp section**.

The **omp parallel sections** directive effectively combines the **omp parallel** and **omp sections** directives. This directive lets you define a parallel region containing a single **sections** directive in one step.

The results are the following:



Comparison between Sequential Quicksort, MPI Quicksort and OpenMP QuickSort



	1	2	3	4	5	6	7	8	9
Sequential	0	1	15	108	175	1106	2260	17003	55665
MPI	2	3	5	23	43	146	378	1907	3897
OpenMP	0.05	0.9	7	67	175	980	1591	7459	18827
Size	100	1000	10000	50000	100000	500000	1000000	5000000	10000000

Platform Information

- Operation System: **Windows 10 Home**
- CPU: **i5-10210U**
  - Cores: **4**
  - Threads: **8**
  - Cache: **6 MB**
  - Base Frequency: **1.60 GHz**
  - Max Turbo Frequency: **4.20 GHz**
- GPU: **Intel(R) UHD Graphics**
- RAM: **8.0 GB at 2133 MHz running in dual channel mode**

6. [Github Repository]