CMPE 180C Operating Systems
Final Project Presentation

Group Members: Debasish Panigrahi [015702414]
Gaurav Sarkar [016197857]
Romit Ganjoo [016054038]

Problem Statement

Write a multithreaded sorting program that works as follows: A list of integers is divided into two smaller lists of equal size. Two separate threads (which we will term sorting threads) sort each sublist using a sorting algorithm of your choice. The two sublists are then merged by a third thread—a merging thread—which merges the two sublists into a single sorted list.

Implement the preceding project (Multithreaded Sorting Application) using Java's fork-join parallelism API. This project will be developed in two different versions. Each version will implement a different divide-and-conquer sorting Algorithms:

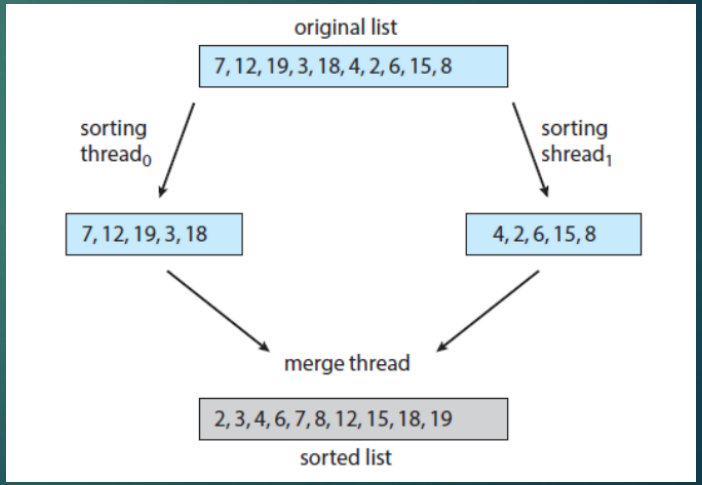
- Quick sort
- Merge sort

Introduction

- Sorting algorithms are used to put items in order.
- Quick Sort and Merge Sort algorithms use divide and conquer technique to sort elements.
- Parallelism is achieved by using multiple threads to sort the divided lists and the list is sorted faster.
- This project, involves the usage of fork-join Parallelism API to implement multi-threading to sort the lists.
- Also, Java's Comparable interface is incorporated that compares two objects and returns value accordingly.

Divide and Conquer Technique

- Used in Merge Sort and Quicksort
- The problem is divided into smaller subproblems that are similar to the original problem.
- The smaller problems are solved using recursion and then the smaller parts are combined to get the final solution



Multi-Threading

- Multiple threads are created within a process.
- All threads are executed independently but concurrently.
- The resources of the process are shared between the threads.
- If hardware allows, all threads can run fully parallelly if they are distributed to their own CPU cores.
- Example: Web Browser
  - Multiple threads doing different tasks
  - Loading content, playing video, etc.

Comparable Interface

- Java Comparable interface is used to order the objects of the user-defined class
- Only one method named compareTo(Object)
- public int compareTo(Object obj): Used to compare the current object with the specified object. It returns:
  - positive integer, if the current object is greater than the specified object.
  - negative integer, if the current object is less than the specified object.
  - zero, if the current object is equal to the specified object.

```
if (data.get(i).compareTo(pivotValue) < 0)
{
    swap(i, storeIndex);
    storeIndex++;
}
```

- We can sort the elements of String objects, Wrapper class objects and User-defined objects.

Fork Join Pool Parallelism API

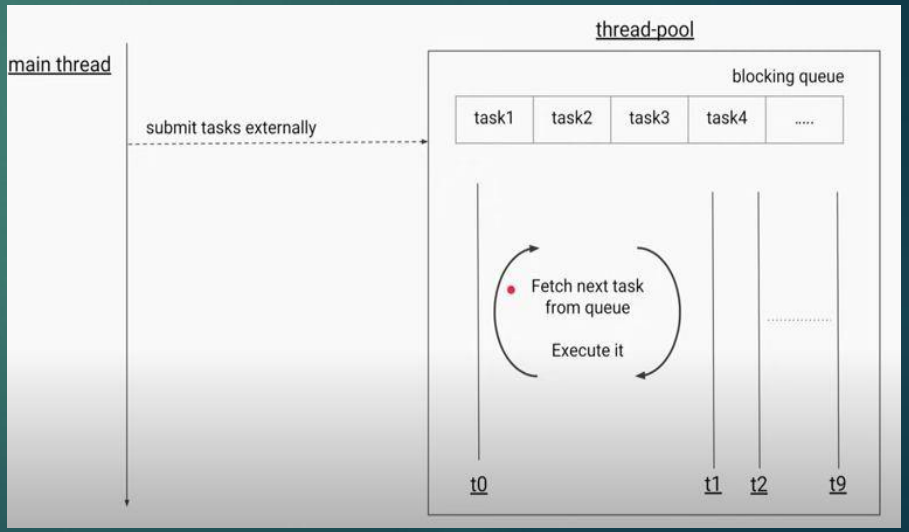
- The fork/join framework has two main classes:
  - ForkJoinPool
  - ForkJoinTask
- ForkJoinPool- implementation of the interface ExecutorService
- Executors provide an easier way to manage concurrent tasks than plain old threads.
- We can create our own ForkJoinPool instance using the following constructor:
  - ForkJoinPool()

Fork Join Pool Parallelism API

```
ForkJoinPool forkJoinPool = new ForkJoinPool();

forkJoinPool.invoke(new ListSort<Integer>(list, 0, list.size() - 1));
```

```
ForkJoinPool pool = new ForkJoinPool();
pool.invoke(new ListSortq<Integer>(list));
```



Fork Join Pool Parallelism API

- ForkJoinPool class invokes a task of type ForkJoinTask, which can be implemented by extending one of its two subclasses:
  - RecursiveAction: represents tasks that do not yield a return value, like a Runnable.
  - RecursiveTask: represents tasks that yield return values, like a Callable
- These classes contain the compute() method, which will be responsible for solving the problem directly or by executing the task in parallel.

Fork Join Pool Parallelism API (Contd.)

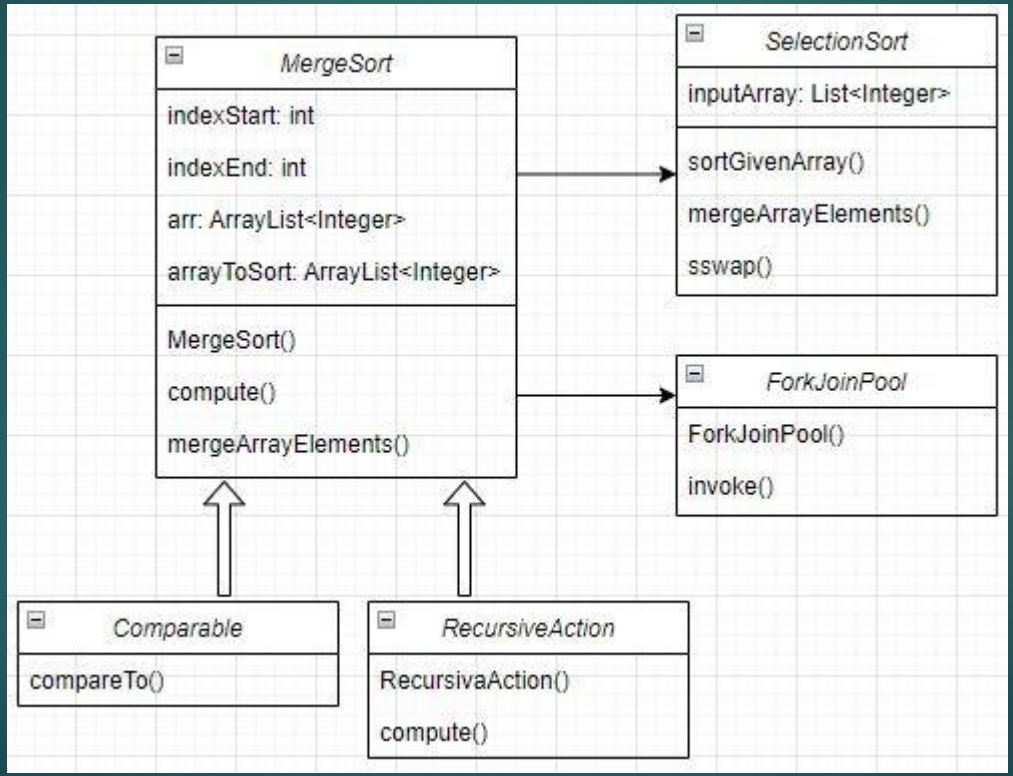
Quick Sort:

```
protected void compute() {
        if (left < right){
            int pivotIndex = left + ((right - left)/2);
            pivotIndex = partition(pivotIndex);
            invokeAll(new ListSortq(data, left, pivotIndex-1), new ListSortq(data, pivotIndex+1, right));
        }
}
```

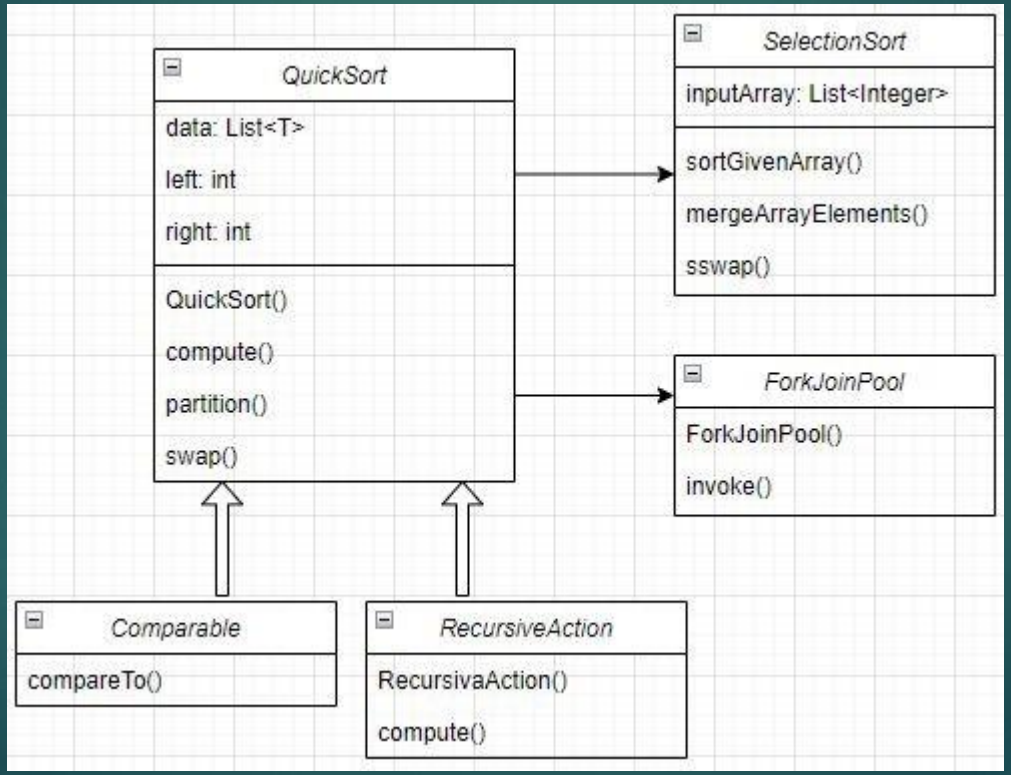
Merge Sort:

```
protected void compute() {
    if (indexStart < indexEnd && (indexEnd - indexStart) >= 1){
      int middleElement = (indexEnd + indexStart) / 2;
      ListSort<Integer> left = new ListSort<Integer>(arrayToSort, indexStart, middleElement);
      ListSort<Integer> right = new ListSort<Integer>(arrayToSort, middleElement + 1, indexEnd);
      invokeAll(left, right);
      mergeArrayElements(indexStart, middleElement, indexEnd);
    }
}
```

Software Architecture : Merge Sort



Software Architecture : Quick Sort



Merge Sort Implementation

```
class ListSort<N extends Comparable> extends RecursiveAction {

    int indexStart, indexEnd, arr[];
    private final ArrayList<Integer> arrayToSort;

    public ListSort(ArrayList<Integer> arrayToSort) { this.arrayToSort = arrayToSort;}
    public ListSort(ArrayList<Integer> arr, int low, int high) {
        this.arrayToSort = arr; this.indexStart = low;  this.indexEnd = high;
    }
    public ArrayList<Integer> getArrayAfterSorting() { return arrayToSort; }

    @Override
    protected void compute() {
      if (indexStart < indexEnd && (indexEnd - indexStart) >= 1){
          int middleElement = (indexEnd + indexStart) / 2;
          ListSort<Integer> left = new ListSort<Integer>(arrayToSort, indexStart, middleElement);
          ListSort<Integer> right = new ListSort<Integer>(arrayToSort, middleElement + 1, indexEnd);
          invokeAll(left, right);
          mergeArrayElements(indexStart, middleElement, indexEnd);
      }
    }
```

Merge Sort Implementation:
Merge Function

```
    // MERGESORT Code
    public void mergeArrayElements(int indexStart, int indexMiddle, int indexEnd){
        ArrayList<Integer> tempArray = new ArrayList<>();
        int getLeftIndex = indexStart, getRightIndex = indexMiddle + 1;
        while (getLeftIndex <= indexMiddle && getRightIndex <= indexEnd){
            if (arrayToSort.get(getLeftIndex).compareTo((arrayToSort.get(getRightIndex))) <= 0){
                tempArray.add(arrayToSort.get(getLeftIndex));  getLeftIndex++;
            }
            else { tempArray.add(arrayToSort.get(getRightIndex));  getRightIndex++; }
        }
        while (getLeftIndex <= indexMiddle){
            tempArray.add(arrayToSort.get(getLeftIndex)); getLeftIndex++;
        }
        while (getRightIndex <= indexEnd){
            tempArray.add(arrayToSort.get(getRightIndex)); getRightIndex++;
        }
        for (int i = 0; i < tempArray.size(); indexStart++)
            arrayToSort.set(indexStart, tempArray.get(i++));
    }
}
```

Quick Sort Implementation

```
class ListSortq<T extends Comparable> extends RecursiveAction {
    private List<T> data; private int left, right;

    public ListSortq(List<T> data){
        this.data = data; this.left = 0; this.right = data.size() - 1;
    }

    public ListSortq(List<T> data, int left, int right){
        this.data = data; this.left = left; this.right = right;
    }

    @Override
    protected void compute(){
        if (left < right){
            int pivotIndex = left + ((right - left)/2);
            pivotIndex = partition(pivotIndex);
            invokeAll(new ListSortq(data, left, pivotIndex-1), new ListSortq(data, pivotIndex+1, right));
        }
    }
```

Quick Sort Implementation:
Partition Function

```
    private int partition(int pivotIndex){
        T pivotValue = data.get(pivotIndex);
        swap(pivotIndex, right);
        int storeIndex = left;
        for (int i=left; i<right; i++){
            if (data.get(i).compareTo(pivotValue) < 0){
                swap(i, storeIndex); storeIndex++;
            }
        }
        swap(storeIndex, right);
        return storeIndex;
    }
    private void swap(int i, int j){
        if (i != j){
            T iValue = data.get(i);
            data.set(i, data.get(j));
            data.set(j, iValue);
        }
    }
}
```

Testing using Junit

For MergeSort Code

```
@Test
void test1() {
    System.out.println("\nTestCase-1: ");
    int size = 200;
    ArrayList<Integer> list = new ArrayList<>();//(Arrays.asList(1,3,5,7,2,6,3));

    System.out.print("\nUnsorted list: ");
    for (int i = 0; i < size; i++) {
        int value = (int) (Math.random() * size);
        list.add(value);
        System.out.print(" " +list.get(i));
    }
    System.out.println(" ");
    ArrayList<Integer> slist=list;
    if (size <= 100) {
        SelectionSort o = new SelectionSort(list);
        o.sortGivenArray();
        System.out.println("\nSorted list: ");
        for (int i = 0; i < size; i++) {
            System.out.print(" "+list.get(i));
            assertEquals(slist.get(i),list.get(i));
        }
    }
    else {
        ListSort<Integer> ob = new ListSort<Integer>(list);
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        forkJoinPool.invoke(new ListSort<Integer>(list, 0, list.size() - 1));
        ArrayList<Integer> arraySorted= ob.getArrayAfterSorting();
        Collections.sort(slist);
        System.out.print("\nSorted list: ");
        for(int i=0;i<arraySorted.size();i++) {
            System.out.print(arraySorted.get(i)+" ");
            assertEquals(slist.get(i),arraySorted.get(i));
        }
    }
}
```

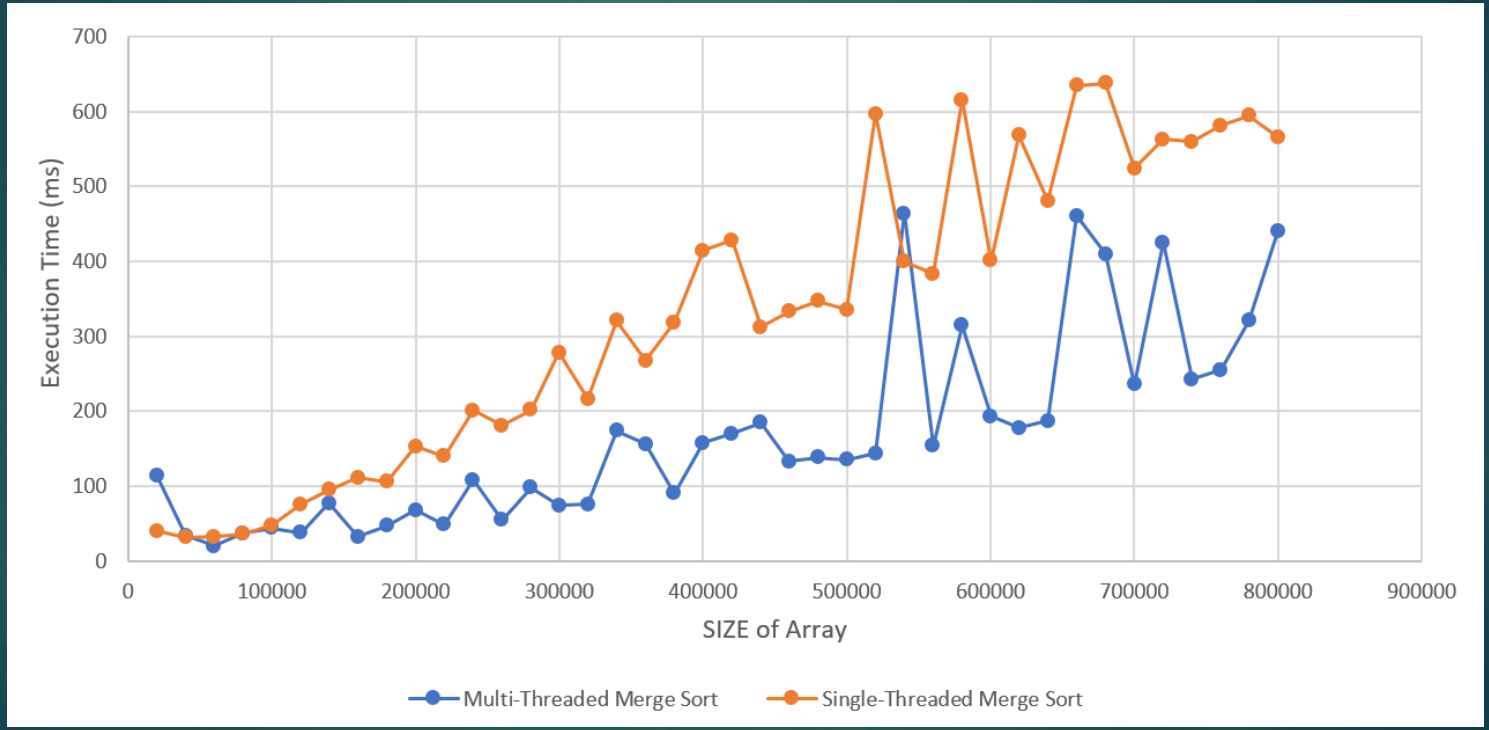
For QuickSort Code

```
@Test
void test() {
    final int SIZE = 101;
    System.out.println("\n\nTest case-1:");
    List<Integer> myList = new ArrayList<Integer>(SIZE);
    System.out.print("\nUnsorted list: ");
    for (int i=0; i<SIZE; i++){
        int value = (int) (Math.random() * 100);
        myList.add(value);
        System.out.print(" "+myList.get(i));
    }
    List<Integer> sortlist=myList;

    if(SIZE<=100) {
        SelectionSort o = new SelectionSort(myList);
        o.sortGivenArray();
        System.out.println("\nSorted List:");
        for (int i=0; i<SIZE; i++){
            System.out.print(" " + myList.get(i));
        }
    }
    else {
        Collections.sort(sortlist);
        long start = System.nanoTime();
        ListSortq<Integer> quickSort = new ListSortq<Integer>(myList);
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(quickSort);
        long end = System.nanoTime();
        System.out.print("\nSorted List:");
        for (int i=0; i<SIZE; i++){
            System.out.print(" "+myList.get(i));
            assertEquals(sortlist.get(i), myList.get(i));
        }
System.out.println(String.format("\nTime taken: %f msec", (end - start) / 1000000.0));
    }
}
```

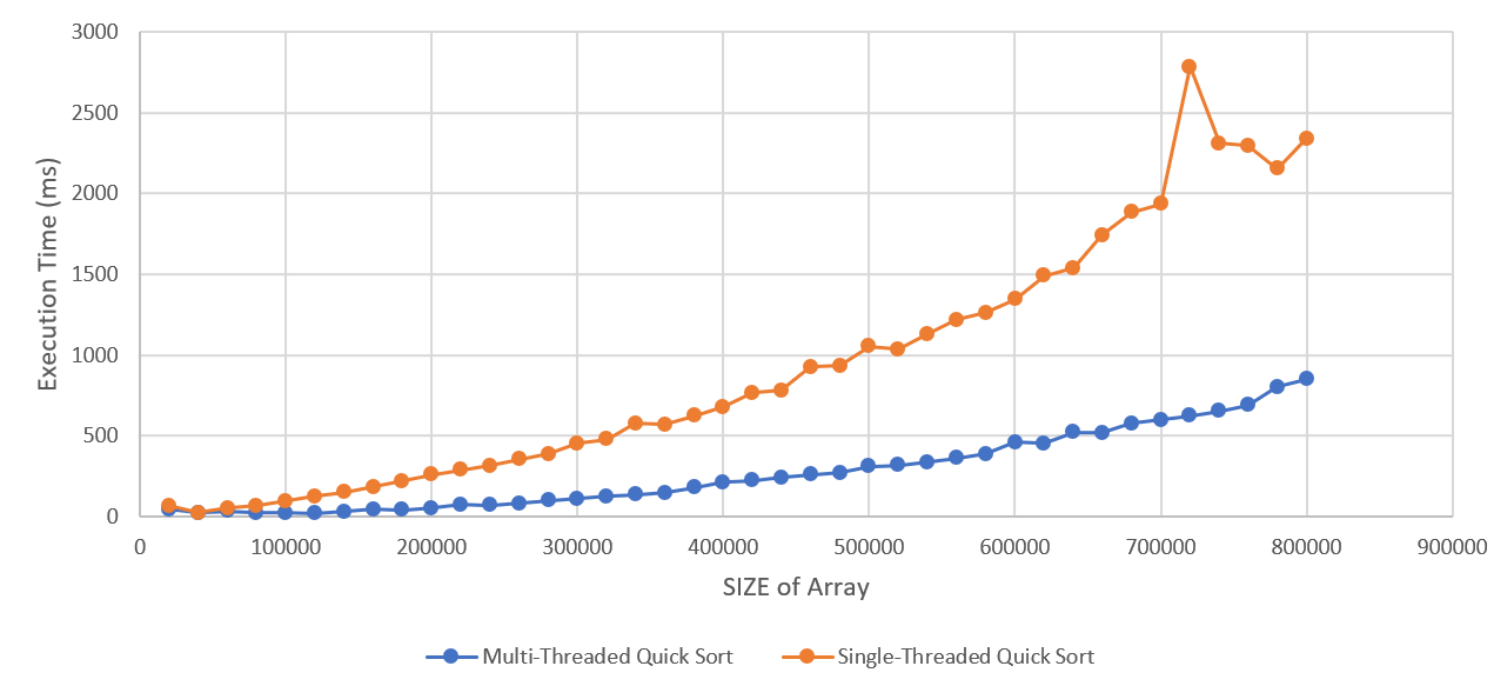
Research work Observations

Execution Time of Multi-Threaded Merge Sorting and Single-Threaded Merge Sorting



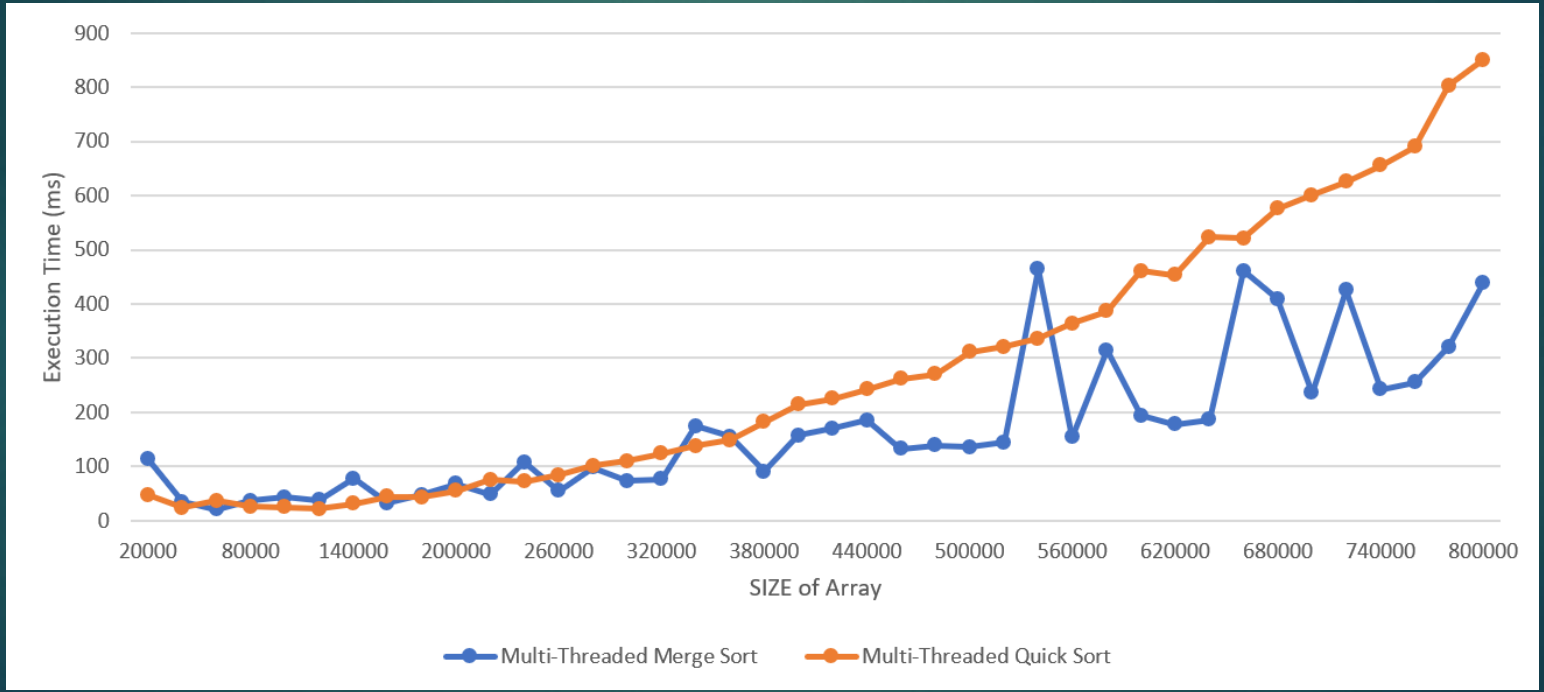
Research work Observations

Execution Time of Multi-Threaded Quick Sorting and Single-Threaded Quick Sorting



Research work Observations

Execution Time of Multi-Threaded Merge Sorting and Multi-Threaded Quick Sorting



Observations

Multi-Threaded Merge Sorting vs Multi-Threaded Quick Sorting vs Single-Threaded Merge Sorting vs Single-Threaded Quick Sorting

SIZE of Array	Multi-Threaded Merge Sort (ms)	Multi-Threaded Quick Sort (ms)
20000	114.8298	47.3909
40000	34.5953	24.6066
220000	48.9163	75.7825
240000	108.4639	72.1381
260000	55.0154	84.9622
280000	98.6903	101.44
360000	155.9307	148.5941
380000	90.6427	182.4472
580000	315.03	386.6689
600000	193.7055	460.4006
660000	461.2971	520.8313
760000	255.2862	690.1221
780000	320.8397	803.5147
800000	439.736	850.1991

SIZE of Array	Multi-Threaded Merge Sort (ms)	Single-Threaded Merge Sort (ms)
20000	114.8298	40.5487
40000	34.5953	32.2591
140000	78.0043	95.3417
160000	32.6125	111.4422
180000	47.4848	105.9981
200000	68.1043	153.088
340000	174.1319	320.7811
380000	90.6427	318.3772
500000	135.4561	335.8717
620000	177.569	568.2505
660000	461.2971	634.7613
740000	242.4484	559.3338
800000	439.736	565.9278

SIZE of Array	Multi-Threaded Quick Sort	Single-Threaded Quick Sort
20000	47.3909	70.478
100000	25.4362	97.62
120000	22.2705	127.206
200000	55.0626	261.164
220000	75.7825	289.655
300000	110.3722	454.665
400000	214.1448	681.1
500000	311.572	1054.492
600000	460.4006	1345.69
620000	453.6739	1490.719
680000	576.2991	1883.996
700000	601.3056	1934.513
800000	850.1991	2339.664

Conclusion

In this Project, we learned the practical implementation of ForkJoinPool & RecursiveAction classes, and also how and where to implement it in real-time issues.

We discovered and compared the execution time taken by MergeSort Algorithm and QuickSort Algorithm to produce results for the problem statement.

The execution time was discovered by performing the tests till 800000 as the size of the list of the algorithms in single-threaded environment as well as multi-threaded environment.

Finally, we found the multi-threaded merge-sort algorithm to be most efficient in execution time comparision.

References

Operating System Concepts (Tenth Edition),

By Abraham Silberschatz, Peter Baer Galvin & Greg Gagne

Thank You