CMPE 180C Operating Systems

Final Project Presentation

Harshika Shrivastava
[016019120]

Mahavir Chandaliya
[016003325]

Rutik Sangle
[016007589]

Problem Statement

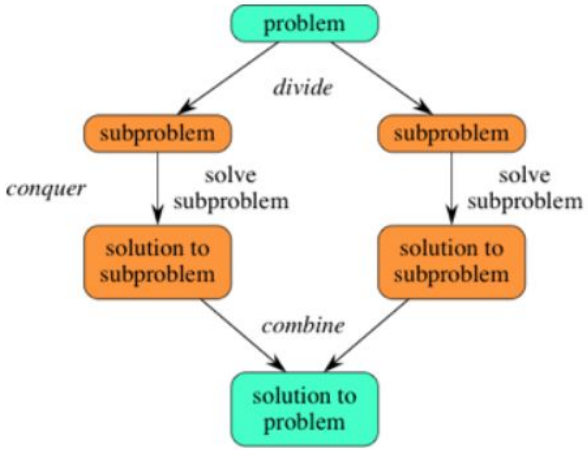
Implementation of Sorting Algorithms such as MergeSort and QuickSort using multithreading approach by using fork-join parallelism API of java and comparable interface.

Introduction

- Sorting algorithms are used widely in computer applications to put items in an order
- Quick Sort and Merge Sort are the algorithms that use the divide and conquer technique to sort elements
- Multiple threads can be used to sort these divided lists such that parallelism is achieved and the list is sorted faster
- In this project, the fork-join Parallelism API of java is used to implement multi-threading to sort the lists
- Along with this, Java's Comparable interface is used which compares two objects and returns value accordingly

Divide and Conquer Technique

- This technique is used in Merge Sort and Quicksort
- In Divide and Conquer Technique, the problem is divided into smaller subproblems that are similar to original problem
- These smaller problems are solved recursively and then merged 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 parallel 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
- It contains only one method named compareTo(Object)
- public int compareTo(Object obj): It is 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.

```
(arrayToSort.get(getLeftIndex).compareTo((arrayToSort.get(getRightIndex))) <= 0)
```

- 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 and ForkJoinTask
- ForkJoinPool is an implementation of the interface ExecutorService
- Executors provide an easier way to manage concurrent tasks than plain old threads.
- You can also create your own ForkJoinPool instance using the following constructor:
    - ForkJoinPool()

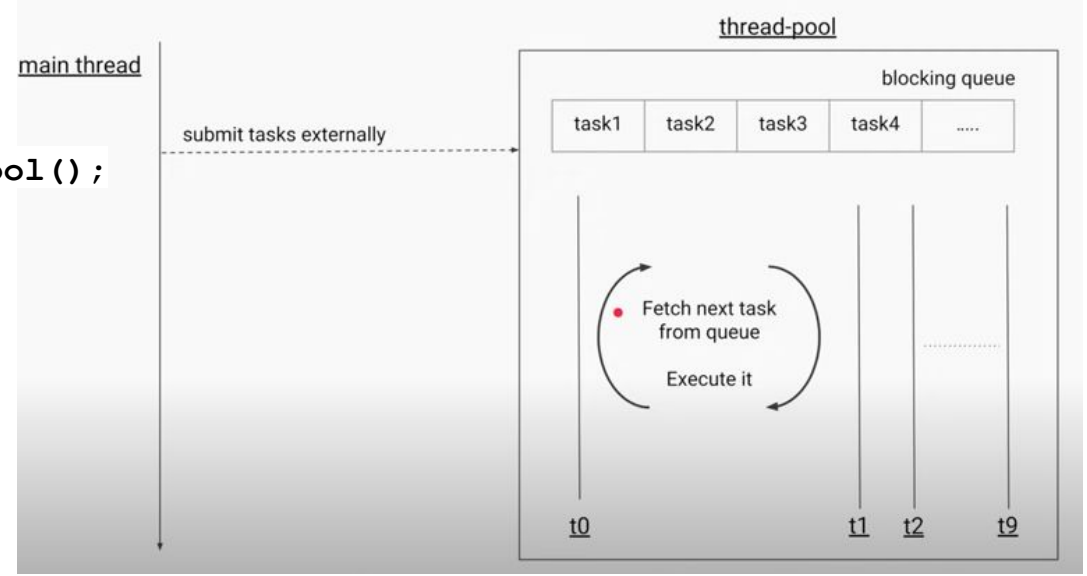
Fork Join Pool Parallelism API (contd.)

```
ForkJoinPool forkJoinPool = new ForkJoinPool();

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

```
ForkJoinPool pool = new ForkJoinPool();

pool.invoke(quickSort);
```

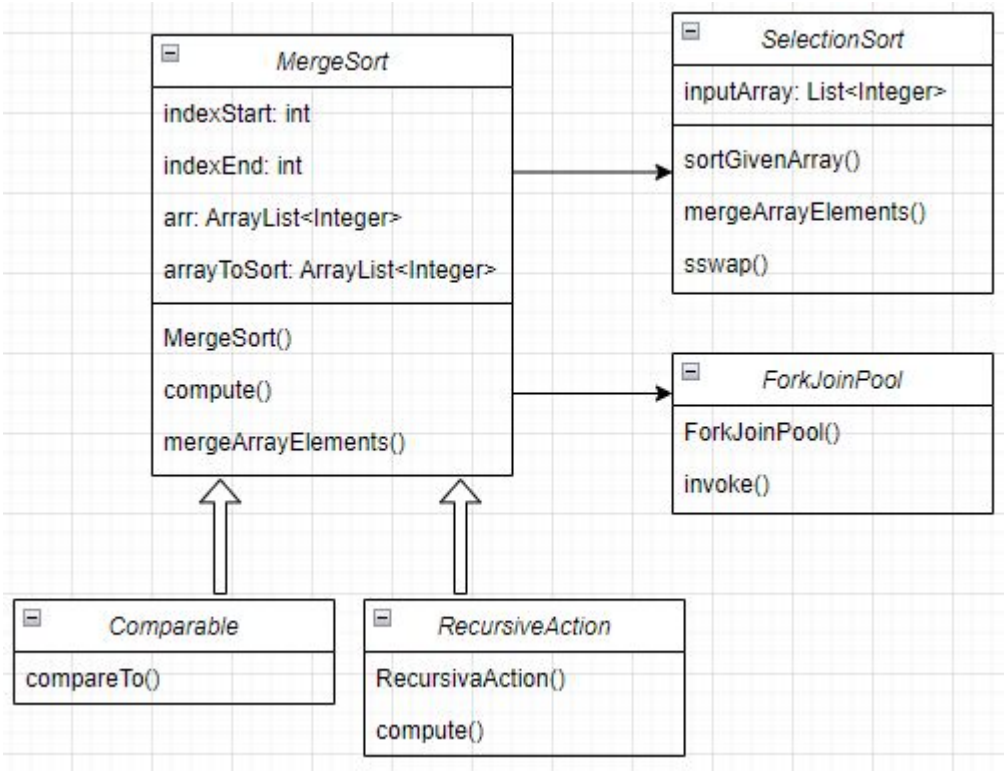


Fork Join Pool Parallelism API (contd.)

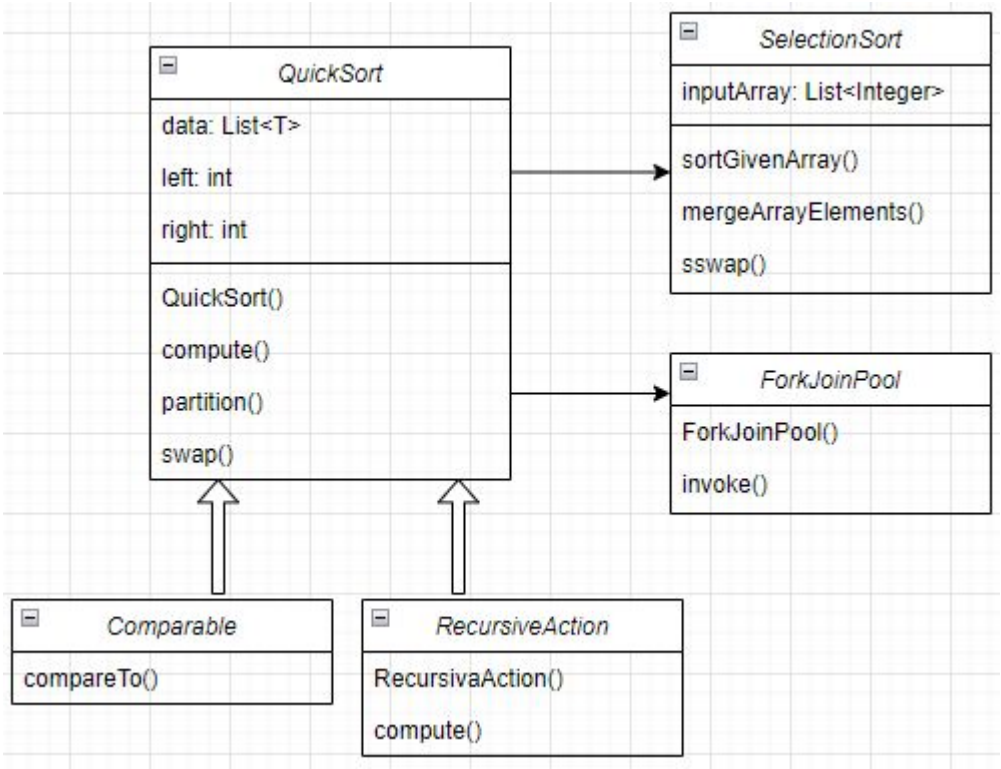
- ForkJoinPool class invokes a task of type ForkJoinTask, which can be implemented by extending one of its two subclasses:
  - RecursiveAction, which represents tasks that do not yield a return value, like a Runnable.
  - RecursiveTask, which represents tasks that yield return values, like a Callable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

```
@Override
protected void compute()
……
invokeAll(new QuickSort(data, left, pivotIndex-1),new QuickSort(data, pivotIndex+1,right));
……
MergeSort<Integer> left = new MergeSort<Integer>(arrayToSort, indexStart, middleElement);
MergeSort<Integer> right = new MergeSort<Integer>(arrayToSort, middleElement + 1, indexEnd);
invokeAll(left, right);
```

Software Architecture : Merge Sort



Software Architecture : Quick Sort



Merge Sort Implementation

```
protected void compute() {
    if (indexStart < indexEnd && (indexEnd - indexStart) >= 1) {

        int middleElement = (indexEnd + indexStart) / 2;
        MergeSortClass<Integer> left = new MergeSortClass<Integer>(arrayToSort, indexStart, middleElement)
        MergeSortClass<Integer> right = new MergeSortClass<Integer>(arrayToSort, middleElement + 1, indexE
        invokeAll(left, right);
        mergeArrayElements(indexStart, middleElement, indexEnd);
    }
```

Merge Sort Implementation- Merge Function

```
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

```
@Override
protected void compute() {
    if (left < right){
        int pivotIndex = left + ((right - left)/2);

        pivotIndex = partition(pivotIndex);

        invokeAll(new QuickSort(data, left, pivotIndex-1),
                  new QuickSort(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

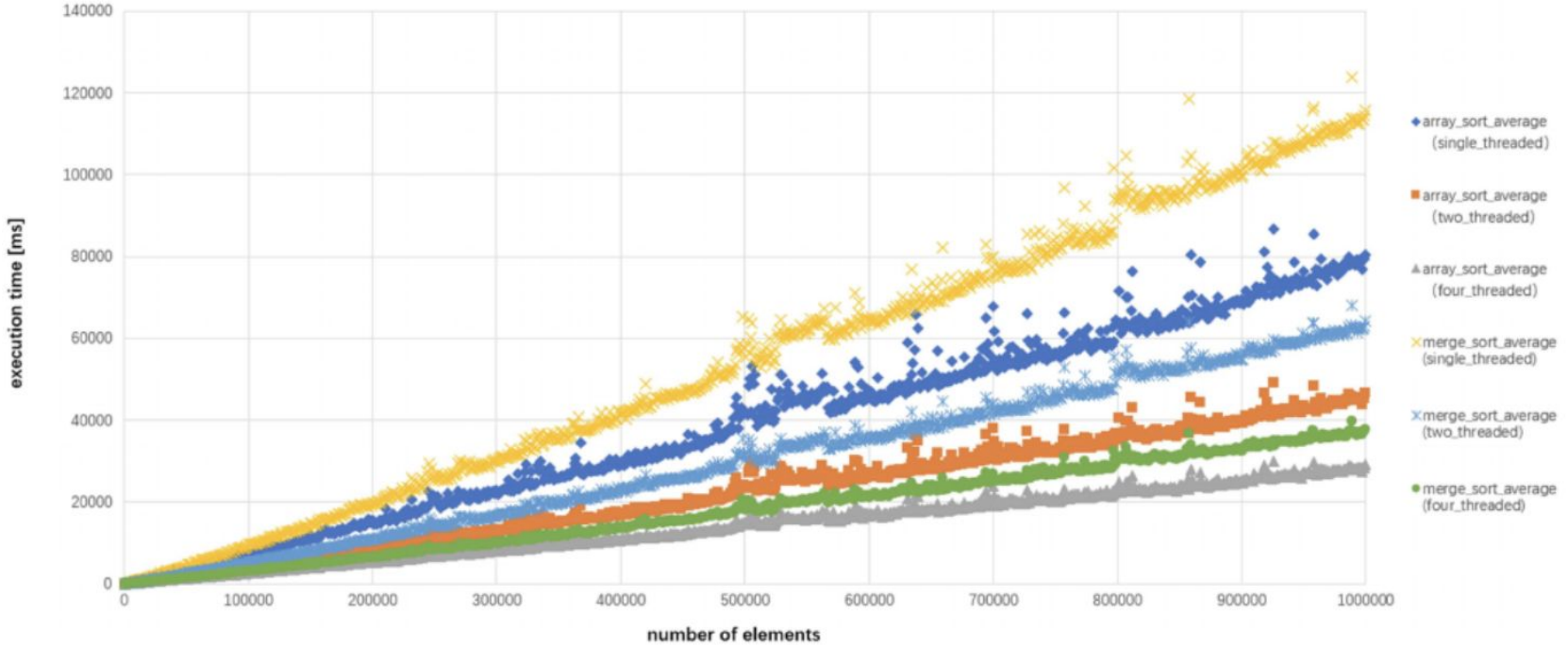
```
void test() {
    System.out.println("\nTestCase");
        int SIZE = 100000;
        ArrayList<Integer> myList = new ArrayList<>();
        System.out.print("\nBefore sorting ");
        for (int i = 0; i < SIZE; i++) {
            int value = (int) (Math.random() * SIZE);
            myList.add(value);
            System.out.print(" " +myList.get(i));
        }
        System.out.println(" ");
        ArrayList<Integer> slist=myList;
        MergeSortClass<Integer> ob = new MergeSortClass<Integer>(myList);
        ForkJoinPool forkJoinPool = new ForkJoinPool();
        forkJoinPool.invoke(new MergeSortClass<Integer>(myList, 0, myList.size() - 1));
        ArrayList<Integer> arraySorted= ob.getArrayAfterSorting();
        Collections.sort(slist);
        System.out.print("\nSorted array ");
        for(int i=0;i<arraySorted.size();i++) {

             System.out.print(arraySorted.get(i)+" ");
            assertEquals(slist.get(i),arraySorted.get(i));
        }
```

Research work:

number of elements

Fig. 8 *Execution time of array sort and merge sort (single-threaded)*



Observations:

Time taken by Merge Sort algorithm

No. of elements to be compared	Single-Threaded Merge Sort	Multi-Threaded Merge Sort
200	4.47 ms	6.33 ms
400	6.6 ms	8.2 ms
600	8.7 ms	8.3 ms
5000	29.43 ms	24.33 ms
100000	418.7 ms	250.3 ms
200000	1566 ms	340.08 ms

Observations:

Time taken by Quick Sort algorithm

No. of elements to be compared	Single-Threaded Sort	Multi-Threaded Merge Sort
10000	3.34 ms	21.88 ms
50000	20.43 ms	60.29 ms
100000	27.72 ms	100.09 ms
500000	426.72 ms	239.73 ms
900000	2544.36 ms	340.90 ms

Thank You!