YSC4231: Parallel, Concurrent, and Distributed Programming
Midterm Project Report
"The Lighting Quicksort"

by Raihanah Nabilah Fatinah
A0207754B

Table of Contents

Section 1: Simple Quicksort

Implementation

In the Simple Quicksort, I implemented the "regular" quick sort the way I learned it in Introductory Data Structure Algorithm class. The algorithm will pick an element as a pivot. The chosen pivot will be placed in the "correct" place in the array and partition the other elements into two subarrays. The subarray of the left of the pivot contains elements smaller than the elements of the pivot, and the subarray to the right of the pivot contains elements larger than the elements of the pivot. Then, each of these subarrays are sorted recursively by calling quick sort function again. It will then do the same method, that is picking an element as a pivot and partition the elements until all the elements in the array are sorted. This quick sort algorithm has the worst-case time complexity of $O(n^2)$ and the best-case time complexity of $O(nlogn)$.

General challenges and discoveries

My main challenges when I ran this quick sort were:

1. In working with the indices. I had to be careful when defining my indices because if I wrote the wrong range of indices or selected the wrong indices, it might give errors such as infinite loop, index out of range, or taking the wrong elements to be worked on during pivot selection and partition.
2. In defining another function (i.e. *swap*) that is used in the quick sort, as it might have high time complexity. I encountered this when my simple quick sort took 32 minutes to run under 100,000 elements. After doing some debugging, I realized that my implementation of *swap* is way too slow and inefficient. What I did was doing a for-loop on all elements in the array to check if both values that we wanted to *swap* are in the array. This issue is now resolved in which I no longer used for-loop to check, and the quick sort runs much faster!

Since we have learned the beauty of concurrency, we want to see how we can implement concurrency in quick sort and make it runs faster.

Section 2. Parallel Quicksort

Implementation

In this implementation, we parallelised the recursive sub-calls when it's working on the disjoint sub-arrays. To do so, every time we finished partitioning the arrays, when it's calling the recursive sub-calls, it will spawn new threads to work on the sub-arrays. This way, the recursive sub-calls can run concurrently. Every time we spawned new threads to work on the sub-arrays, we should call the method *start()* to start both threads to execute the recursive call on the lower and the upper sub-arrays respectively. Then, we should call the method *join()* to wait for the threads to die when they are done executing. This way, we synchronise the new threads with the parent thread that spawned them.

Comparing Parallel Quicksort with Simple Quicksort

When I tried to run both Simple Quicksort and Parallel Quicksort for different array sizes, I checked their runtime, and it gives me the following result:

[Array size 5]
SimpleSort: 35 ms
ParallelSort: 12 ms

[Array size 100]
SimpleSort: 15 ms
ParallelSort: 41 ms

[Array size 5000]
SimpleSort: 60 ms
ParallelSort: 2,919 ms

From the above result, we see that parallel sort runs faster on simple sort when the array size is small, as shown in the example of array size 5. However, when array size is medium to large, the simple sort runs faster than parallel sort, as shown in the array size of 100 and 5000. This is because when the array size is small, the recursive sub-calls are not being called too many times. Hence, we don't spawn that many new threads. Due to this, doing quick sort in parallel will make it run faster and more efficient. However, when the array size is large, then it will spawn arbitrarily many new threads. Spawning new threads at every recursive sub-call is already expensive. Doing this many times will make the implementation very expensive and inefficient. Moreover, in real life, it is not possible to spawn arbitrarily large number of threads at the same time, as we can only have limited number of threads. To aid this, we can create an implementation such that we only have limited number of threads at every moment, but it will still run the recursive calls concurrently each time after an array of elements is being partitioned. This is where Thread Pool comes in.

Section 3. Thread Pool

Thread Pool consists of array of fixed number of workers threads in which each of these threads will concurrently work on arbitrary number of "tasks". The "tasks" here are the recursive sub-calls of quick sort. The main functionality of Thread Pool works in this way:

- The threads will be waiting for the user to give them tasks
- When there's a task, the threads will be made aware of them, so that they can start working on the tasks and executing them

But before I started discussing the implementation of the "workers" threads and the methods of Thread Pool, let me go through the components of Thread Pool first.

Thread Pool class

Thread Pool class takes the number of "worker" threads as an argument. Then, their main components are:

1. *taskQueue*. As the name suggests, *taskQueue* is a queue of all the pending tasks that the threads in the pool must work on after the tasks were enqueued by the user. In our code, the queue is a simple sequential (non-concurrent) queue because we want the threads to exclusively enqueue and dequeue a task in. This will avoid multiple threads to enqueue or dequeue the same tasks.
2. *workers.* This is the array of fixed number of threads that will execute the tasks in parallel. The index of the worker thread in the array corresponds to their thread IDs. Unlike parallel sort, we did not spawn new threads whenever there is a new concurrent task to be executed. Instead, we have fixed number of threads that will take the task from the *taskQueue* and execute them.
3. *workersHelper.* I created an array variable called *workersHelper* in which it will help create the *workers* array where the index number of the array corresponds to the thread ID of the worker thread. I decided to do for-loop here and combine each of the array of new worker thread using the ++ operator.
4. *workInProgress.* This is the array of Boolean to record which threads are currently executing a task and which ones are not. This array is a non-atomic array because we want multiple threads to be able to access the array at any time. Recording the thread's activity status to the array does not have to be mutually exclusive. This is because each thread exactly has one flag in the array, and they will only be updating theirs. Since these workers threads are executing their tasks concurrently, they can access and update this array at their own timing when they started executing or finished executing a task.
5. *isShutDown*. This is a Boolean variable that will indicate that the thread-pool is no longer usable.

The Thread Pool has an inner class *Worker*, which implements the worker thread functionality. Moreover, the Thread Pool class also provides three methods, which are *shutdown(), async(task: Unit => Unit)* which takes task as its argument, and *startAndWait(task: Unit => Unit)* which also takes task as its argument. My design decisions of each of these methods are affected by the Worker thread, and my Worker thread implementation is updated every time I implemented each of my Thread Pool methods. Hence in the following sections, I will talk about my design process, discovery, and challenges that I encountered when implementing them.

The Worker thread (first draft)

After reading Professor Ilya's assignment guide and to do list, in my first draft of Worker thread, I want my threads to do the following:

(1) When the *taskQueue* does not have a task, the thread will wait
(2) When there is a task, the thread will go out of waiting
(3) The thread will dequeue the task
(4) The thread will update *workInProgress* array that they are working on the task, assigning the Boolean in their position index to be true

(5) The thread will execute the task
(6) The thread will update *workInProgress* array that they are no longer working on the task, assigning the Boolean in their position index to be false
(7) Repeat step 1 to step 6

After writing the above pseudocode, I identified that the thread must always run in the while-loop to repeat step 1 to 6, meaning that they must happen inside a *while(true)* loop. Then, for step 1 to step 3, the access to read the *taskQueue* state and enqueuing the task require mutual exclusion. This is because I want to avoid multiple threads taking the same task from the Queue, resulting the tasks being executed twice or more. To do the mutual exclusion, I am using locks. For the threads to do step 1 to step 3, they should acquire the locks (provided by java the class *ReentrantLock*) and then release them when they are done in step 3. Then, since the threads require waiting on certain conditions, I decided to use monitors. By using the monitor, when a certain condition is fulfilled (the taskQueue is empty), the thread will call the method *await()* on the condition object of the corresponding lock. The thread will then "sleep" in the "waiting room" and then it will "wake up" again when another thread call a method *signalAll()* on the same condition object. Note that when the thread is "sleeping", it will only temporarily release the lock without leaving the critical section. Then, when the threads "wake up", the threads in the "waiting room" will acquire the lock again and check if it fulfills the condition. If it doesn't, the thread will go out of the "waiting room", but if it does, the thread will be put back to "sleep". This *signalAll()* method will be called when the *async* method or the *startAndWait* method is called, meaning that there is a task being added to the queue, which I will discuss in detail later. My current worker thread looks like the following:

```
while (true){
      lock.lock()
      try {
            while (taskQueue.isEmpty) {
                  isEmpty.await()
            }
      task = taskQueue.dequeue()
      } finally {
            lock.unlock()
      }

      workInProgress(id) = true
      task()
      workInProgress(id) = false
}
```

Notice that step 4 to step 6 did not have any synchronization and did not have mutually exclusive access to it unlike step 1 to step 3. This is because I want the threads to execute the tasks concurrently. If executing and keeping the record of the thread's status require mutual exclusion, then the threads will execute the task sequentially instead of concurrently. In addition, don't forget that the threads can be interrupted while they are waiting. Hence, after the *while(true)* loop, the worker thread should catch *InterruptedException*, in case the threads are interrupted. Why do we need to interrupt the

threads? This will become clearer when I talked about the *shutdown()* method in later sections.

When I finished writing this *Worker* thread for the first time, I wasn't really sure what's the purpose of updating the *workInProgress*. However, the purpose of this becomes clear to me when I started implementing *startAndWait*. In fact, this was my biggest source of bug. But for now, let's discuss the *async* method first.

The async method

In the *async* method, I want the thread to:

(1) Enqueue a task to the queue
(2) Signal all or make all the worker threads aware that there is now a task in the queue

Much like when dequeuing a task from the queue, I also want enqueueing the task to the queue to be mutually exclusive, which means we will be using the locks. Moreover, when we enqueue a task, we call a method *signalAll()* on the same condition object when we send the threads to "sleep" when there's no task in the queue. In addition, I also want my async method to handle "shut down" gracefully. In other words, when a thread pool is shut down, calling the method async should return an exception. Hence, I will only run step 1 to step 2 if the thread pool is not shut down. My *async* method then looks like the following:

```
if (!isShutdown){
     lock.lock()
     try {
          taskQueue.enqueue(task)
          isEmpty.signalAll()
     } finally {
          lock.unlock()
     }
}
```

The *async* method main functionality is essentially to only schedule a task for the threads in the thread pool. It doesn't block the main thread from moving into the next line of execution, and it does not guarantee that all the tasks in the queue are executed before *async* returns. Therefore, in running asyncExample file, if I did not allow the thread to wait for some time before shutting down, we can see that the main thread moves on to the next line of execution (printing "Finished before executed" and "About to shut down the pool") before the worker threads finished executing all their tasks. This means that our thread pool can only pick up and complete 1 execution before the main thread prints the line "Finished before executed". Due to this, if we want our workers thread to finish their execution, we put the main thread to sleep for 1000 ms until all 10 tasks added into the queue by the main thread are picked up by the thread pool and executed. This will be resolved in *startAndWait* function.

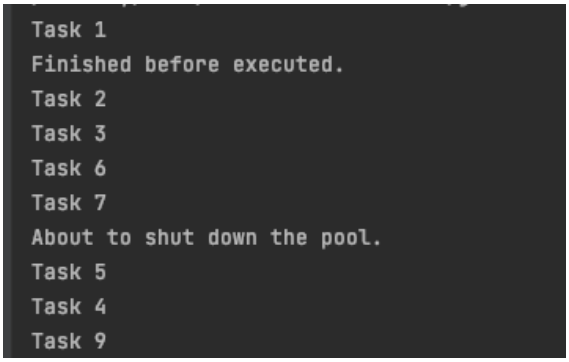

Figure 1. Running async() on asyncExample file without Thread.sleep(1000)

The startAndWait method

Much like *async* method, I also want the main thread to add tasks to the queue and signal the worker threads that there's tasks in the queue. However, unlike *async* method, I want the main thread to be blocked until the thread pool picked up and executed all the tasks added to the queue by the main thread. In pseudocode, this is what I want my *startAndWait* method to do:

(1) Enqueue a task to the queue
(2) Signal all or make all the worker threads aware that there is now a task in the queue
(3) When there is a thread that's still executing tasks or when the queue still has some tasks, then the main thread should be blocked

Like *async* method, enqueuing in *startAndWait* method is also mutually exclusive. Then, per usual, we call the method *signalAll()* on condition object *isEmpty* to wake up the "sleeping" threads that were put to "sleep" when waiting for a new task in the queue. Then, now I have a new condition object that I must create to do step 3. While there is a thread that's still executing some tasks or the queue still has some tasks to do, I can send the main thread to sleep until another thread calls the method signalAll() on the same condition object. I called this new condition object *onProgress*. In addition, like in async, I also want my *startAndWait* method to handle "shut down" gracefully. I will only run step 1 to step 3 if the thread pool is not shut down. My *startAndWait* method then looks like the following:

```
if (!isShutdown){
     lock.lock()
     try {
          taskQueue.enqueue(task)
          isEmpty.signalAll()
          while (!taskQueue.isEmpty ||
               !workInProgress.forall(x => x == false)) {
               onProgress.await()
          }
     } finally {
          lock.unlock()
     }
}
```

Notice that I am using a different condition object for when the main thread added a new task and when the main thread is being blocked. As Professor Ilya mentioned in the lecture notes, this multiple condition variables makes it easier to reason around giving permissions for different threads. That is, each conditional variable focuses on different purposes. The *onProgress* condition variable focuses on blocking the main thread until all tasks executed by the workers threads are finished. On the other hand, the *isEmpty* condition variable focuses on blocking the workers thread until there is a task in the queue ready to be executed.

Though it might look like I implemented this method smoothly, I initially encountered a lot of bugs. When I first implemented step 3, I only put "*!taskQueue.isEmpty*" as a condition to block the main thread. Meaning that I only check whether the queue has tasks or not. If it has tasks, then it will send the main thread to sleep. However, I realized that it could be the case that all the threads have finished dequeuing the task, but they have not finished the execution. This means that my condition is flawed and I am missing a condition.

Recall that earlier, I wasn't sure what's the purpose of keeping record of the threads' activeness status. It was at this moment of implementing *startAndWait* that I realize what's *workOnProgress*' true purpose is. In my condition to block the main thread, aside from checking the status of the queue, I also must check if there is at least one thread that is still executing the task. If there is, then the main thread will still be blocked until all the threads have done executing the task. Hence, I added a new condition to block the main thread, that is by checking if at least one of the Boolean values in *workOnProgress* array is true.

Even with this knowledge, I still had a bug. I initially thought that the condition is that the queue should not be empty and that at least one the threads should be active when the main thread is being blocked. However, I realized that it should not be an "and" condition but rather an "or" condition. If we put an "and" condition, then when all threads just finished executing the tasks while there are still some tasks in the queue, the main thread will not be blocked. Hence, the condition should be "or" instead of "and".

The Worker thread updated (final draft)

Now that I know I have a new monitor and a new condition variable *onProgress*, I can go on updating my *Worker* thread. Let's look at my old pseudocode and make changes by adding a new step in step 1, written in blue and italic.

(1) If the taskQueue is empty and if all the threads have finished executing the tasks, then signal the main thread to no longer be blocked
(2) When the *taskQueue* does not have a task, the thread will wait
(3) When there is a task, the thread will go out of waiting
(4) The thread will dequeue the task
(5) The thread will update *workInProgress* array that they are working on the task, assigning the Boolean in their position index to be true
(6) The thread will execute the task
(7) The thread will update *workInProgress* array that they are no longer working on the task, assigning the Boolean in their position index to be false
(8) Repeat step 1 to step 6

I then adjusted my *Worker* thread which look like the following:

```
while (true){
      lock.lock()
```

```
        try {
            if (taskQueue.isEmpty &&
                workInProgress.forall(x => x == false)) {
                onProgress.signalAll()
            }
            while (taskQueue.isEmpty) {
                isEmpty.await()
            }
            task = taskQueue.dequeue()
        } finally {
            lock.unlock()
        }

        workInProgress(id) = true
        task()
        workInProgress(id) = false
    }
```

The reason why I put the new step at the top is because I wanted to check this condition after a thread finishes a task and before it is about to dequeue a new task. Another option is to put them at the bottom (after step 7), but in my opinion, this will be rather inefficient as I have to implement another mutual exclusion with the locks at the bottom. Initially, I had concern that when the worker threads are first started, then it will fulfill the condition (that is *taskQueue* being empty and that all threads are inactive), which means it will *signalAll()* the main thread on the *onProgress* condition object. However, when the worker threads are first started and ran, the *startAndWait()* method hasn't been called and that the tasks have not been enqueued. Hence, the threads will be sent to "sleep" rightaway on the *isEmpty* condition object. This means putting the new condition at the start of the workers thread will not do any harm to the implementation.

Another important thing to note is that we should start all of our workers thread in the *workers* array by calling the *start()* function. This is done inside our Thread Pool class. This way, when we create a new instance of Thread Pool, the workers threads are also started.

The shutdown method

The *shutdown* method is needed to interrupt or kill the worker threads that are stuck in *await()*. The Thread class in Java has an *interrupt()* method, so to kill the threads, I can simply call this method on all of our worker threads. Moreover, I also have to set the *isShutDown* boolean to true, so that the thread pool can no longer be used and will throw an exception if it's used again. My implementation looks like the following:

```
        for (i <- 0 until n){
            workers(i).interrupt()
        }
        isShutDown = true
```

The *shutdown()* method is very crucial, which is very evident when experimenting with the *shutdown()* method call in the AsyncExample and StartAndWaitExample files. In both

AsyncExample and StartAndWaitExample, when I tried to remove the *pool.shutdown()* line, the threads are continuously "running", and they are not being killed. This is because when there is no tasks being added to the queue, the workers threads are stuck in *await().* Since my workers threads worked in a *while(true)* loop, this will always continue. Even though the threads have finished adding some tasks and executing all of them, eventually when there is no more tasks to be executed, the workers threads will always be stuck in *await()* until the main thread adds new tasks again to the queue. This is why when the main thread is done with adding tasks and the workers threads have finished executing them, it is crucial to kill the threads, so that they get out of *await().*

Section 4. Pooled Quicksort

Implementation

Now that we have thread pool, we can implement our quick sort using thread pool. This version of quick sort will allocate each recursive sub-calls (or the "tasks") to the threads in the thread pool instead of spawning and joining new threads for every recursive sub-calls. The implementation follows the same logic as the quick sort.

The main *Sort* function or the "main thread" will first create new instance of Thread Pool that takes the fixed number of arrays that we want to have in the Quicksort as an input. Then, it will call *startAndWait* method and run the function *(_ => quickSort(arr, lo, hi))*, which essentially executes *quickSort*. The main thread ensures that the execution of *quickSort* is done until all the recursive sub-calls of *quickSort* is executed. In other words, the main thread waits until the whole sorting process is done. Then, in the recursive sub-calls, it will call async method and run the functions *(_ => quickSort(arr, lo, mid))* and *(_ => quickSort(arr, mid+1, hi))* respectively. We call async method in the recursive sub-calls of *quickSort* because we want the execution of the recursive sub-calls to happen concurrently for faster executions without waiting for other recursive calls to finish. Remember that a thread's job when *async* is called is only to execute *quickSort* and do partition. That's it. After that, calling another recursive *quickSort* will be a new "task" for the other threads in the pool to do.

When there are limited number of threads in the pool, there are indeed going to be more "tasks" or more recursive sub-calls than there are threads. However, this is the main purpose of thread pool. In executing the recursive sub-calls or the "tasks", when all the threads are busy (are active in executing the task), the execution of the recursive sub-calls will be delayed until the resource (or the threads) is available again to execute a new task. This allows for concurrent execution of tasks but with limited number of threads instead of spawning new threads at every recursive sub-calls.

Discoveries and challenges

I noticed that the way we generate arrays matter, as it affected the time complexity of our Quicksort implementation. When the arrays have little diversity in their elements (i.e. we generated a random array of size 100,0000 with numbers ranging between 1 and 5), the Quicksort implementation becomes significantly slower, as shown in the result below.

[Array size 100000 – numbers ranging between 1 and 5] PooledSort time: 6,180 ms
[Array size 100000] PooledSort time: 441 ms

This is because when there is little diversity in the elements, it depreciates the efficiency of our Quicksort implementation, as our time complexity will be $O(n^2)$. Hence, to maximize time complexity and get the best result, in my array generator, I did not limit the range of numbers of the elements in the array.

Benchmark comparison

I then tried to run a benchmark comparison for ScalaSort, SimpleSort, and PooledSort. For the PooledSort, I run the benchmark with Thread Pool having five worker threads. I got the following result:

```
Running benchmarks for array size 100000
ScalaSort : 63 ms
SimpleSort: 81 ms
PooledSort: 265 ms

Running benchmarks for array size 1000000
ScalaSort : 456 ms
SimpleSort: 569 ms
PooledSort: 1,142 ms

Running benchmarks for array size 10000000
ScalaSort : 4,553 ms
SimpleSort: 7,064 ms
PooledSort: 14,626 ms

Running benchmarks for array size 20000000
ScalaSort : 10,208 ms
SimpleSort: 12,824 ms
PooledSort: 30,401 ms
```

Figure 2. Running benchmarks for different array sizes for Scala Quicksort, Simple Quicksort, and Pooled Quicksort

From the above result, we see that our Pooled Quicksort runs the slowest. It takes twice to thrice as long to run the Pooled Quicksort in compared to running the Simple Quicksort. I have to say I am quite taken aback by the result. I thought that since the recursive sub-calls are ran concurrently, the execution of quicksort will become faster. However, reflecting on the Parallel Quicksort where it is faster on small array size and slow on medium to large array size, this result is not surprising. One possible explanation to Pooled Quicksort running slower than Simple Quicksort is perhaps when the array size is small, the process of doing concurrent executions in different threads becomes expensive and it depreciates efficiency. This is because when the array size is small, so does the number of concurrent tasks that the thread pool has to do. Then, there are not many tasks that the thread pool can work on, which means the workers threads in the thread pool are not really optimized. To me, this is evident in the benchmark result. In the array size of 100,000, the time it takes to run the Pooled Quicksort is about thrice as large as that of Simple Quicksort. However, as the array

size grows larger, in the array size of 10,000,000, the time it takes to run the Pooled Quicksort is only about twice as large as that of Simple Quicksort. As the array size grows larger, Pooled Quicksort becomes more efficient. Hence, to produce the best and the fastest implementation of Quicksort, I want to combine Pooled Quicksort with Simple Quicksort, making the best out of concurrent and sequential worlds in Hybrid Quicksort.

Section 5. Hybrid Quicksort

Implementation

In my Hybrid Quicksort, I combined the Pooled Quicksort and the Simple Quicksort together. In my implementation, I wrote it such that when the array size is larger than a certain number (let's call it *minSize*), then it will use the Pooled Quicksort method when calling the recursive sub-calls so that the tasks will be run concurrently. That is by calling the *async* method and run the function *(_ => quickSort(arr, lo, mid))* and *(_ => quickSort(arr, mid+1, hi)).* However, when the array size is less than *minSize*, then it will use the Simple Quicksort method when calling the recursive sub-calls. That is the usual *quickSort(arr, lo, mid)* and *quickSort(arr, mid+1, hi).*

We know that Pooled Quicksort works better for arrays that are larger. This means *minSize* value should be an array size that is around medium size. I would usually experiment with different values and try to run each of them, but let's try to see what value could make the most sense by comparing the benchmarks for array size 5, 100, 1,000, 10,000, 100,000, and 1,000,000 in Simple Quicksort and Pooled Quicksort.

```
Running benchmarks for array size 5
SimpleSort: 0 ms
PooledSort: 2 ms

Running benchmarks for array size 100
SimpleSort: 0 ms
PooledSort: 3 ms

Running benchmarks for array size 1000
SimpleSort: 1 ms
PooledSort: 5 ms

Running benchmarks for array size 10000
SimpleSort: 14 ms
PooledSort: 38 ms

Running benchmarks for array size 100000
SimpleSort: 87 ms
PooledSort: 177 ms

Running benchmarks for array size 1000000
SimpleSort: 560 ms
PooledSort: 860 ms
```

Figure 3. Running benchmarks for array sizes 5, 100, 1,000, 10,000, 100,000, and 1,000,000 for Simple Quicksort and Pooled Quicksort

In the above result, we see that the benchmark gap between Simple Quicksort and Pooled Quicksort is largest when the array size is around 1,000. At this size, Simple Quicksort runs about 5 times faster than Pooled Quicksort. This shows when the array size is around 1,000, the Pooled Quicksort time execution is at its worst. However, when the array size is increased to 10,000, 100,0000, and 1,000,000, the benchmark gap between Simple Quicksort and Pooled Quicksort decreases, as the Simple Quicksort is only around 2 to 3 times faster than Pooled Quicksort. Since the benchmark gap becomes smaller when the array size is between 1,000 and 10,000, let's zoom in in the array size between 1,000 to 10,000 and try to compare the benchmark between Simple Quicksort and Pooled Quicksort again to approximate the array size of the worst possible time execution for Pooled Quicksort.

Figure 4. Running benchmarks for array sizes 1,000, 2,000, 3,000, 4,000, 5,000, 6,000, 7,000, 8,000, 9,000 and 10,000 for Simple Quicksort and Pooled Quicksort

From the above result, we see that the benchmark difference between Simple Quicksort and Pooled Quicksort is largest when the array size 4,000. At this size, Simple Quicksort runs about 5 times faster than Pooled Quicksort. This shows when the array size is around 4,000, the Pooled Quicksort works at its worst. We can then define our *minSize* to be 4,000. Therefore, when array size is smaller than 4,000, our Hybrid Quicksort should use Simple Quicksort method when calling the recursive sub-calls. However, when array size is larger than 4,000, our Hybrid Quicksort should use Pooled Quicksort method when calling the recursive sub-calls. We will then run the benchmark for all Scala Quicksort, Simple Quicksort, Pooled Quicksort, and Hybrid Quicksort. Note that for both Pooled Quicksort and Hybrid Quicksort, I ran the benchmark with Thread Pool having 5 worker threads.

Benchmark comparison

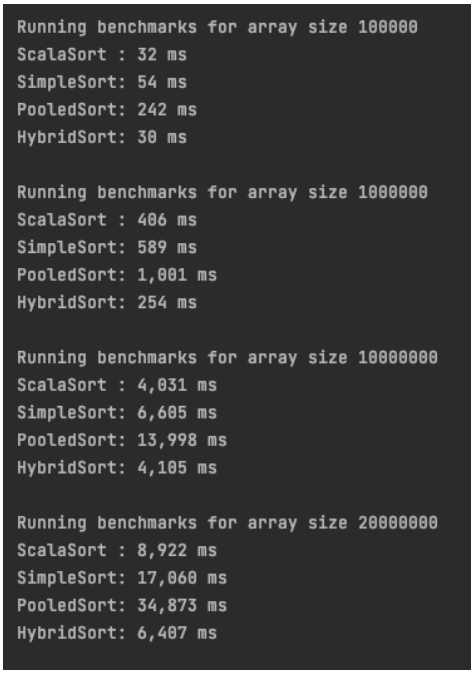

Figure 5. Running benchmarks for different array sizes for Scala Quicksort, Simple Quicksort, Pooled Quicksort, and Hybrid Quicksort

From the result above, we see that Hybrid Quicksort runs faster than both Simple Quicksort and Pooled Quicksort. In three of the cases where array size is 100,000, 1,000,000, and 20,000,000, our Hybrid Quicksort even runs faster than the Scala Quicksort. This shows that when we combined the concurrent and sequential worlds together, we can get the best result.

Acknowledgement

I would like to thank Professor Ilya Sergey for teaching PCDP so patiently and for helping me identify my problems and bugs during office hours. I also would like to thank Mark, the peer tutor, for always patiently answering my questions during peer-tutoring sessions.

References

These are the main learning materials that greatly helped me with working on this midterm project.

Sergey, I. (n.d.). 7. Week 06: Monitors and Blocking Synchronisation. *Parallel, Concurrent and Distributed Programming (Autumn 2021).* Retrieved October 7, 2021, from https://ilyasergey.net/YSC4231/week-05-monitors.html.

Sergey, I. (n.d.). 4.3. Quicksort and its Variations. *Introductory Data Structures and Algorithms.* Retrieved October 7, 2021, from https://ilyasergey.net/YSC2229/week-04-quicksort.html.