PCDP Mid Term Report

Dabeer Ahmed

September 2024

1 Introduction

If I had to sum up this project and course, it would be: "PCDP is Data Structures and Algorithms—but on steroids (the healthy kind)." And I'm all about that health-conscious programming life.

This assignment took the familiar world of sorting and supercharged it with concurrency, parallelism, and the raw power of modern multi-core processing. What seemed like a simple task quickly became a complex dance of threads, tasks, locks, and queues, pushing me to rethink how code should run in a parallel universe.

In this report, I'll take you through the journey—from the meticulous design of my thread pool to the performance tweaks that turned a straightforward sorting algorithm into a high-performing, multi-threaded machine. I'll share the moments of triumph, the concurrency phenomena observed, and, yes, the moments where things went completely off the rails (because debugging concurrency is a beast of its own).

2 Understanding Quick Sort

To get a clear grasp of the project's goal, I took a step back and revisited the basics, starting with a warm-up exercise: implementing Quicksort in Scala. The warm-up genuinely did its job—it warmed me up to the world of sorting once again. Instead of just copy-pasting code from a pseudocode example, I wanted to dive deep and truly understand every line and the underlying mechanics of Quicksort.

For this, I turned to Mr. Abdul Bari, a legendary YouTuber known for his clear and intuitive explanations of algorithms. It had been about 1.5 years since I last touched Data Structures and Algorithms, so a refresher was in order. I revisited the core intuition behind Quicksort: *"divide the array using a pivot and then recursively sort the sub-arrays."* The idea of partitioning and pivoting came back quickly, but I noticed something peculiar in the implementation—a reference to infinity being added at the end, which puzzled me at first.

To bridge this gap, I referred to Geeks for Geeks for a more implementation-focused perspective on Quicksort. With these resources, I grasped both the

theory and practice, ultimately implementing Quicksort in Scala by adapting these ideas into code.

3 Ensuring Quicksort Works: A Late-Night Grind

Before wrapping up for the night, I needed to make sure my Quicksort implementation was actually working and passing the tests. To streamline this, I implemented 'ArrayUtil', which quickly became the unsung hero of the sorting tests. All the helper functions were straightforward to implement and kept the code clean and manageable.

The real challenge came with 'CheckSameElements'. As a seasoned CS major and avid LeetCode grinder, I knew there was only one rule: "When in doubt, use a hash map." Hash maps are like the Swiss army knife of coding—they solve problems fast, and they do it in O(n) time. Instead of sifting through nested loops, I let the hash map work its magic, efficiently checking if two arrays contained the same elements.

With all the pieces in place, my Quicksort not only passed the tests but did so with the quiet confidence of code that knows it won't be breaking anytime soon.

4 Parallel QuickSort Implementation

Implementing 'ParallelQuickSort' was an interesting challenge, but the provided description made the task feel straightforward. The description clearly outlined that Quicksort's "divide-and-conquer" nature makes it inherently parallelizable. Each recursive sub-call, performed after partitioning, operates on disjoint subarrays, which can be sorted independently. This independence is the key to parallelizing Quicksort effectively, allowing us to spawn threads for each subcall and make full use of parallel processors.

Parallel QuickSort Design

Given this naturally parallel structure, implementing 'ParallelQuickSort' involved creating separate threads for the left and right sub-arrays after each partitioning step. By spawning new threads for each recursive call, the work is divided, enabling the sorting of each part concurrently.

Spoiler Alert: I ran the benchmark on `ParallelQuickSort`, and let's just say- I saw hell in Computer Science. More on that later.

Do We Need Synchronization?

A critical aspect of this parallel approach is understanding whether synchronization is necessary between these threads. Fortunately, because each thread works on a separate, non-overlapping portion of the array, no explicit synchronization is required during the sorting process itself. The threads do not access shared data that would necessitate locks or other synchronization mechanisms, as each thread only modifies its assigned sub-array.

However, synchronization is still implicitly handled through the use of ‘join()‘ calls, which ensure that the main thread waits for all spawned threads to complete before proceeding.

5 Performance Comparison: Parallel vs. Simple QuickSort

To evaluate the effectiveness of ‘ParallelQuickSort‘, I ran tests to compare its performance against ‘SimpleQuickSort‘. Interestingly, the results showed that ‘ParallelQuickSort‘ was often slower than its simple counterpart, especially on smaller arrays. This might seem counterintuitive at first, given the allure of parallel processing, but it actually makes sense when you consider the nature of the overhead involved in parallelization.

Understanding the Observed Phenomena

The key reason for ‘ParallelQuickSort‘ being slower than ‘SimpleQuickSort‘ lies in the overhead associated with creating and managing threads. While ‘ParallelQuickSort‘ theoretically divides the sorting task into smaller, concurrent units, the act of spawning threads, context switching, and managing synchronization between threads introduces significant overhead. The use of the ‘join()‘ method is essential for ensuring that the main thread waits for all spawned threads to complete their tasks, but it also introduces additional synchronization overhead that can impact the overall execution time. This overhead can easily outweigh the benefits of parallelism, particularly when the size of the array is small, or the depth of recursion is shallow.

6 ThreadPool: The Core of the Assignment

The thread pool was the heart of this assignment. While the main components were laid out, it took considerable effort to understand how the different methods—`run`, `async`, `startAndWait`, and `shutdown`—work together. It wasn't just about implementing them; it was about figuring out the logical flow, the connections between each method, and determining what required synchronization and what didn't. Like any computer scientist, I broke down the problem into manageable sub-problems. Let's dive into these sub-problems, starting with the cornerstone of the `ThreadPool` class: the Task Queue.

6.1 Task Queue

The task queue in the `ThreadPool` is implemented as a simple sequential (non-concurrent) queue. At first glance, this design might seem counterintuitive in a concurrent environment, but it is, in fact, a deliberate and effective choice.

Why Use a Sequential Queue? Using a sequential queue has no limitations because enqueueing and dequeueing are made thread-safe through the use of locks, allowing concurrent access while ensuring safe manipulation of the

queue. In other words, the queue is only being enqueued or dequeued under the protection of the same lock, ensuring that access and retrieval are mutually exclusive.

6.2 workInProgress

The `workInProgress` array is non-atomic and sequential because each thread in the thread pool accesses and modifies its specific element within a synchronized block. This approach leverages the synchronized access to ensure that modifications to the array are thread-safe, without requiring each element to be atomic.

Since each worker thread has a unique ID corresponding to its specific position in the array, there is no contention between threads over the same index. Each thread modifies only its own flag, and this update is done within the protection of a lock. Thus, there is no risk of concurrent modifications to the same element.

6.3 Locks

I decided to use Intrinsic Monitor for locks; however, that decision soon changed. Stay tuned!

6.4 Worker Threads

In the thread pool, each worker continues executing tasks until the pool is shut down. The process is simple: the worker thread acquires a lock, dequeues and executes a task, and then releases the lock. However, there's more going on behind the scenes—we're also doing some strategic bookkeeping to keep tabs on whether the pool has reached a "quiescent moment."

A quiescent moment is that elusive point where all tasks have been completed, and no threads are actively working—essentially, the thread pool has gone quiet. When this moment is detected, we signal the waiting threads, particularly waking up the one stuck in the `startAndWait` function.

What's really happening here is that the bookkeeping keeps track of the pool's activity, allowing the waiting thread in `startAndWait` to recognize when a quiescent moment has been reached. When this moment is detected—meaning all tasks are complete and no threads are working—the bookkeeping signals the waiting thread, allowing it to exit its loop and terminate the process. This synchronization ensures that `startAndWait` only finishes when the entire pool has truly come to a halt, making the shutdown orderly and precise.

6.5 Interrupts

But what happens if a thread is indefinitely waiting? When a thread is indefinitely waiting, an `InterruptedException` is thrown if interrupted. Catching this exception allows the thread to handle the interruption gracefully, perform

necessary cleanup, and shut down properly, rather than crashing. But we wont trigger the shutdown flag here because we want the rest of the tasks to be completed too. This approach ensures that other threads continue their work smoothly, maintaining the flow and stability of the thread pool while managing resources effectively.

In the **interrupt example,** if you don't call `t.interrupt()`, thread `t` will remain stuck in the `while (true)` loop, continuously waiting on `monitor.wait()`. Without the interrupt, the thread will never wake up or terminate, resulting in it staying blocked indefinitely, wasting resources and preventing a clean shutdown of the thread.

6.6 Async

If the pool is shut down, the `async()` method will throw a `ThreadPoolException`, preventing any new tasks from being enqueued. This ensures that no further tasks are added once the pool is marked as inactive.

After enqueuing a new task, the method calls `condition.signalAll()`, which wakes up any waiting worker threads, informing them that a new task is available for execution. This ensures that the workers promptly start processing the newly enqueued tasks.

6.7 startAndWait

Like mentioned earlier, this method enqueues an initial task and blocks the calling thread until all tasks in the pool are completed and the pool reaches a quiescent state. The method unblocks when the pool is in a quiescent state, meaning the task queue is empty and no worker threads are actively processing tasks.

ThreadPool Design Overview

Initially, I implemented the thread pool using intrinsic monitors (`ThreadPool.this.synchronized`). However, this approach was slow and limited, prompting me to switch to using a `ReentrantLock` with a `Condition` variable. This change provided more control over thread coordination and significantly improved performance.

1. Initialization of Worker Threads

The thread pool initializes with a specified number of worker threads (`n`). Each worker thread is created and stored in an array (`workers`). The threads are immediately started upon creation, allowing them to be ready for task execution as soon as tasks are available.

2. Worker Threads and the `run` Method

Lock Usage: A `ReentrantLock` is used to guard access to shared resources like the task queue and the work state array (`workInProgress`). The lock ensures that only one thread can modify or access these shared resources at any given time, preventing race conditions. Inside the `run` method, the lock is acquired before checking the queue and dequeuing a task. This is crucial because accessing the task queue without a lock could lead to inconsistent states or concurrent modification errors.

When Not Using the Lock: The actual execution of the task (`task()`) is performed outside the locked section. This design choice is intentional to avoid holding the lock while the task runs, which could block other threads unnecessarily and reduce concurrency. By executing the task outside of the locked section, the pool maximizes throughput and allows other threads to access the queue concurrently.

3. The `async` Method

Lock Usage: The lock is used when adding new tasks to the task queue to prevent concurrent modifications and ensure that only one thread can add a task at a time. This keeps the queue's state consistent and avoids potential race conditions. After enqueuing a task, `condition.signalAll()` is called to wake up any waiting workers, ensuring that tasks are picked up promptly.

4. The `shutdown` Method

Lock Usage: The lock is used when setting the shutdown flag and signaling all worker threads. This ensures that the shutdown process is coordinated, and no new tasks are accepted once the pool is marked inactive. Worker threads are interrupted to ensure that any waiting threads can exit gracefully, allowing the pool to shut down without leaving threads hanging.

5. The `startAndWait` Method

Lock Usage: The lock is used when enqueuing the initial task and signaling worker threads to ensure that task addition and signaling are performed atomically. The calling thread then waits until the pool reaches a quiescent state, where no tasks remain and no workers are active. This waiting process is managed with the lock to safely check the pool's state and avoid race conditions.

Why Not Hold the Lock Continuously? After enqueuing the task and signaling, the lock is released to allow workers to operate freely. Holding the lock continuously would unnecessarily block worker threads, reducing the efficiency and responsiveness of the pool.

7 Pooled QuickSort

Implementing the thread pool was straightforward once I grasped the logical connections between the key methods: `startAndWait`, `async`, and `shutdown`. `startAndWait` initializes the first task, `async` handles the recursive task calls, and `shutdown` is triggered when the quiescent moment is reached. However, when I ran the benchmarks, I was surprised to find that the Pooled QuickSort was slower than the Simple QuickSort, even when I varied the input sizes.

After a solid 6 hours of contemplating and sleeping on the problem, the realization came, perhaps in a dream, when I understood that QuickSort's efficiency comes from recursively reducing the size of the sub-arrays being sorted. Pooled QuickSort introduces overhead with task management, lock contention, and thread coordination. As the array size gets smaller in recursive steps, this overhead outweighs the benefits, making it slower than Simple QuickSort. However, Pooled QuickSort performs better when the recursive steps involve sorting larger sub-arrays, where the parallelism can be fully leveraged.

Haha Gotcha!

8 Hybrid Sort

This "gotcha" moment was crucial in shaping the approach for implementing Hybrid Sort by effectively combining the strengths of both simple sorting and parallelism. The idea was straightforward yet powerful: use Pooled QuickSort for larger sub-arrays in the recursive steps to exploit parallelism and switch to Simple QuickSort for smaller sub-arrays to avoid the overhead of task management and lock contention. However, the real challenge lay in defining what constitutes "large" and "small." Determining this threshold was key to maximizing the efficiency of Hybrid Sort and balancing the trade-offs between concurrency and overhead.

8.1 Benchmarking

I then embarked on a mini experiment of trial and error, tweaking the number of threads and the threshold for switching between Pooled and Simple QuickSort. The results? The Hybrid Sort's speed compared to Scala's built-in sort ranged from 1 to 3 times faster. With 8 threads and a threshold of 4000 for the sub-array size, I hit the sweet 3x mark. That felt like a win—not just for me but also for my M2 MacBook Air (no flex, just facts). In fancier terms, I fine-tuned the parameters of the Hybrid Sort, and it paid off. Thanks, optimization! In all honesty, the feeling would have been the same if I fine tuned an LLM to produce accurate results.

Oh, and when I said I saw hell while benchmarking Parallel Sort, it turned

out every recursive call was spawning a new thread. So, on an array of 10,000,000, the math on the thread count was... not pretty. My laptop tapped out on threads—epiphany achieved! (better sooner than later).