Lab 2 - Parallel Sorting Algorithms
Parallel and Distributed Computing
DD2443 - Pardis24

Names:
Casper Kristiansson
Nicole Wijkman

Group 14

September 22, 2024

1 Sequential Sort

1.1 Program Explanation

Source files:

- ‘SequentialSort.java‘

1.1.1 Explanation

For the sequential sorting algorithm, we chose to implement quicksort to sort integer arrays. The `sort()` method begins by calling the `quickSort()` function, which then recursively partitions the array and sorts each partition. The partitioning function selects the last element as the pivot and rearranges the array such that all elements smaller than the pivot are on the left, and larger elements are on the right. This process repeats for each subarray until the entire array is sorted. Since this is a sequential sort, only a single thread is used for execution.

1.1.2 Discussion

The sequential quicksort implementation performs as expected with consistent results. Since it operates with a single thread, there is no overhead related to thread management or synchronization, which can benefit smaller arrays. However, the performance is limited by the lack of parallelism, and it scales poorly with larger data sets or higher computational demands. In our test with an array size of 1000, the average execution time was approximately 51,316 nanoseconds, demonstrating that sequential sorting is efficient for small arrays but becomes a bottleneck for larger ones.

2 Amdahl's Law and Speedup

2.1 Formulation of Amdahl's Law

Amdahl's Law estimates the potential speedup of a parallel algorithm by dividing the task into a parallelizable portion, p, and a sequential portion, $1 - p$. In quicksort, parallelization helps up to a point, but as thread count increases, the benefit diminishes due to overhead and small sub-arrays that gain little from parallelization.

To account for this, we modified Amdahl's Law with a quadratic decay for thread effectiveness:

$$S = \frac{1}{(1-p) + \frac{p}{T^2}}$$

Where:

- p is the parallelizable portion.
- T is the number of threads.
- T^2 reflects the faster diminishing returns as more threads are added, capturing quicksort's behavior.

This formulation better models quicksort's performance, where beyond 4 to 8 threads, adding more threads yields limited gains due to overhead.

2.2 Results and Plot

The plot below shows the speedup using the modified Amdahl's Law for 2, 4, 8, and 16 threads, with different values of p (0.2, 0.4, 0.6, 0.8). The quadratic term T^2 causes the speedup to level off more quickly, reflecting how overhead reduces the benefit of adding more threads in parallelized quicksort.

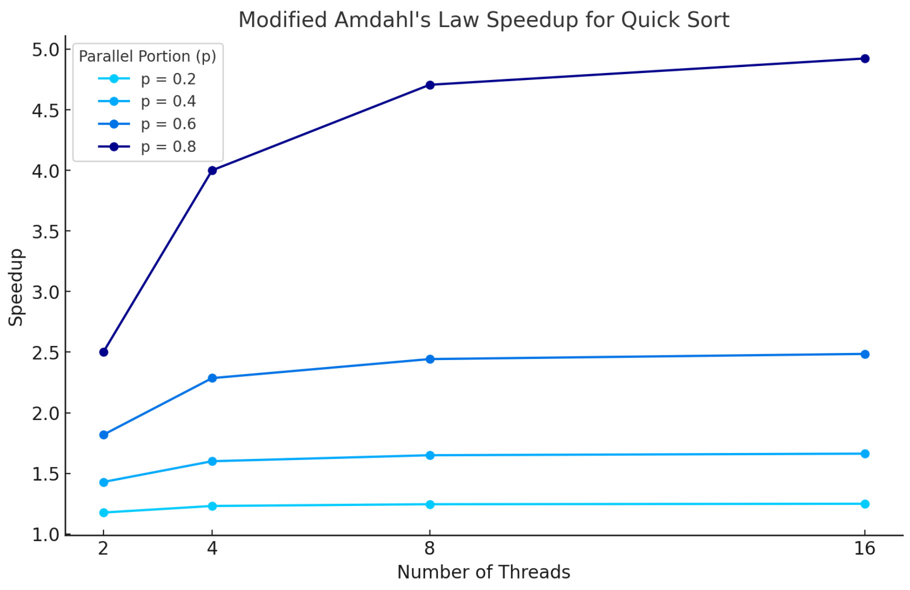

Figure 1: Speedup of Quick Sort using Modified Amdahl's Law for different values of p and thread counts.

3 ExecutorService Parallel Sort

3.1 Program Explanation

Source files:

- ‘ExecutorServiceSort.java‘

3.1.1 Explanation

In this task, we implemented a parallel quicksort using Java’s `ExecutorService` with a fixed thread pool. The algorithm recursively divides the array into subarrays, and tasks are submitted to the thread pool to sort each subarray concurrently. For smaller subarrays (size below 1000), sequential quicksort is used to minimize overhead. The `Future` objects synchronize the tasks, ensuring that both halves are fully sorted before proceeding. After sorting, the executor service is properly shut down to terminate all threads gracefully.

3.1.2 Discussion

The `ExecutorService` efficiently parallelizes quicksort by dividing the workload among multiple threads. This approach improves performance for larger arrays by reducing overall sorting time. However, managing threads and synchronizing tasks introduces some overhead, especially for smaller arrays where the cost of parallelism outweighs its benefits. Despite this, for larger arrays, the ExecutorService approach effectively distributes the sorting process and results in significant speedup.

4 ForkJoinPool and RecursiveAction

4.1 Program Explanation

Source files:

- ‘ForkJoinSort.java‘

4.1.1 Explanation

In this task, we implemented a parallel quicksort using Java’s `ForkJoinPool` and `RecursiveAction`. The sorting process is split into smaller tasks using the `Worker` class, which extends `RecursiveAction`. If the subarray size exceeds a threshold (1000), the array is partitioned and two new tasks are created to sort the left and right subarrays concurrently. The `invokeAll()` method is used to execute the tasks in parallel. If the subarray is small enough, sequential quicksort is used to avoid the overhead of task creation.

4.1.2 Discussion

The `ForkJoinPool` efficiently parallelizes the sorting process by leveraging task decomposition, making it well-suited for larger arrays. The use of `RecursiveAction` allows for a clear division of labor between tasks, ensuring concurrent sorting of subarrays. However, for smaller arrays, the overhead of task creation and management can outweigh the benefits of parallelism. Overall, the ForkJoin-Pool approach improves performance by utilizing multiple threads, but the effectiveness depends on the array size and the chosen threshold.

5 ParallelStream and Lambda Functions

5.1 Program Explanation

Source files:

- ‘ParallelStreamSort.java‘

5.1.1 Explanation

In this task, we implemented a parallel quicksort using Java’s `ParallelStream` and Lambda functions. The sorting algorithm partitions the array and then uses `Arrays.stream()` with `parallel()` to execute the sorting of subarrays concurrently. If a subarray is smaller than a threshold (1000), sequential quicksort is used. To control the number of threads, we wrap the parallel stream in a `ForkJoinPool`, which limits the number of threads to the specified amount. The pool ensures proper management of parallel tasks and is shut down after the sort completes.

5.1.2 Discussion

The use of `ParallelStream` and Lambda functions allows for a clean and functional approach to parallel sorting. It efficiently parallelizes the sorting process, especially for larger arrays. However, there is some overhead due to stream management, and for smaller arrays, the performance benefits diminish compared to a fully sequential approach. Overall, this method is effective in utilizing multiple cores and threads for large datasets, but the choice of threshold and thread management is crucial for optimal performance.

6 Performance Measurements with PDC

6.1 Instrumentation

To measure performance, we instrumented each sorting algorithm to record execution time using `System.nanoTime()`. The tests were conducted with 1,000,000 elements, using thread counts of 2, 4, 8, 16, 32, 48, 64, and 96. For parallel algorithms, we ensured the thread pool and parallel streams were configured to match the specified thread count. Each algorithm was warmed up with 10 rounds before conducting 100 measurement rounds. The sequential implementation served as the baseline, while the JavaSort library was used as a reference for comparison.

6.2 Execution Time and Speedup Plot

The following graph presents the execution time for each sorting algorithm across varying thread counts. The speedup is normalized to the sequential sort.

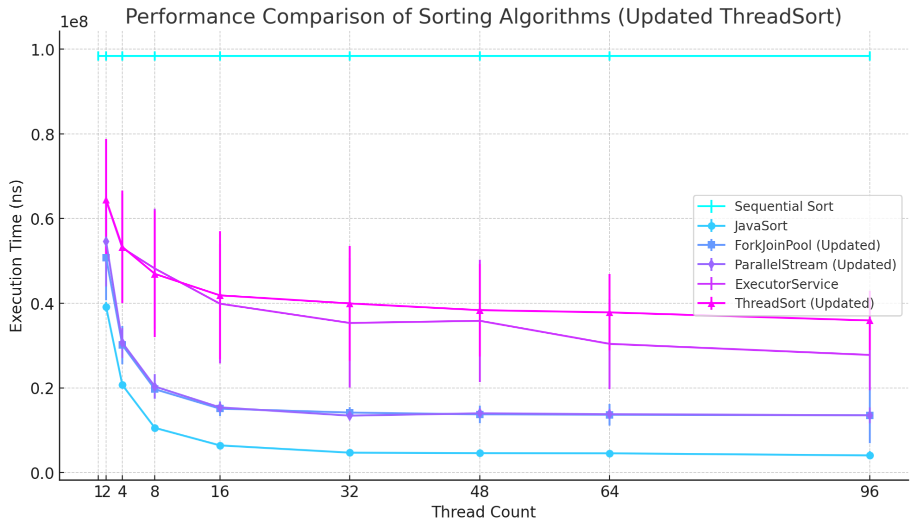

Figure 2: Performance Comparison of Sorting Algorithms with Different Thread Counts.

6.2.1 Discussion

As we expected, parallelized algorithms showed significant speedups compared to the sequential version, especially with larger thread counts. JavaSort exhibited the best overall performance, benefiting from optimized internal algorithms. ForkJoinPool and ExecutorService also performed well, but speedups plateaued after 32 threads due to thread management overhead. ParallelStream, though easy to implement, had higher overhead and showed diminishing returns earlier than ForkJoinPool or ExecutorService. The sequential algorithm remained consistent but was significantly outperformed by all parallel versions at higher thread counts.

7 ThreadSort using start() and join()

7.1 Program Explanation

Source files:

- ‘ThreadSort.java‘

7.1.1 Explanation

We did this extra task after realizing we had used PDC wrong during the presentation. For the task, we implemented a parallel quicksort using Java’s native `Thread` class with the `start()` and `join()` methods. The array is divided into chunks, each of which is assigned to a separate thread for sorting. The number of threads used is specified by the `threads` parameter. Each thread sorts its designated chunk using a sequential quicksort method.

7.1.2 Discussion

The use of Java’s `Thread` class allows direct control over thread management but also introduces challenges in managing thread creation and synchronization. This implementation effectively parallelizes sorting for medium to large arrays, where each thread can handle a significant chunk of data. Additionally, for smaller arrays or when the chunk size is below the threshold, the overhead from thread management can lead to longer execution times compared to other parallel methods like `ForkJoinPool`. Despite these limitations, the implementation demonstrates the basics of parallelism using Java threads.