Project Write Up

Name: Zohaib Rahim

Topic: Parallelized Sorting Algorithm Implementation

Introduction:

The main goal of this project is for me to create and use a sorting algorithm that works in parallel, using the merge sort method, in the realm of concurrent and distributed programming. The project is all about diving deep into the challenges of concurrency and getting hands-on experience in both shared memory and message-passing systems. I got the initial idea for my sorting algorithm from the merge sort technique, which I learned about in CPSC 200: Algorithm Analysis and Development. I've set out five specific objectives and goals for this project. It all kicks off with designing and putting together a parallelized sorting algorithm. Then, I'll dive into the realm of concurrency in sorting, gracefully handling errors and termination. Lastly, I aim to optimize performance.

Background:

The project's objective remains centred on crafting a parallelized sorting algorithm within the domain of concurrent and distributed programming. This area of study holds significant importance, especially in addressing the challenges presented by contemporary computing environments. These environments often demand the efficient processing of vast datasets while harnessing the computational prowess offered by multi-core processors and distributed systems.

Relevant Concepts, Algorithms, and Technologies:

In the realm of concurrent and distributed programming, the utilization of parallel sorting algorithms, exemplified by merge sort, emerges as a prominent approach for efficiently managing large datasets. Merge sort's inherent divide-and-conquer methodology renders it particularly conducive to parallelization. It recursively partitions the dataset, independently sorts each segment, and consolidates the results. This parallel nature makes it well-suited for leveraging multiple threads or processes, thereby enhancing sorting efficiency. The implementation of parallel merge sort in the provided code leverages Java's Fork/Join framework, specifically the **ForkJoinPool**, for managing parallel execution. The **ForkJoinPool** dynamically distributes tasks across available threads in a work-stealing manner, optimizing resource utilization. This allows for efficient parallel sorting of large datasets by dividing the workload among multiple processing units. Concurrent applications rely on synchronization mechanisms, such as the ones provided by the Fork/Join framework, to ensure thread safety and prevent data corruption during concurrent access to shared resources. These mechanisms handle task coordination and synchronization internally, simplifying the development of parallel sorting algorithms without the explicit use of low-level synchronization primitives.

Project Structure:

The codebase comprises four Java files within the Project_CPSC222 package: "ImplementationAndTesting.java", "ParallelMergeSort.java", “SequentialMergeSort.java” and “ParallelStepByStepRepresentation.java”.

1. **ImplementationAndTesting Class:** The class functions as the entry point for the application, housing the main method. It orchestrates multiple test cases to showcase the functionality of the ParallelMergeSort class in sorting arrays. Each test case initializes an array of integers, instantiates an object of ParallelMergeSort, sorts the array using the implemented algorithm, and then prints the sorted result.
2. **ParallelMergeSort Class:** Within the class lies the implementation of the parallel merge sort algorithm. It encompasses methods for sorting arrays (**sort**), executing merge sort (**parallelMergeSort**), executing merge sort for small subarrays (**sequentialMergeSort**), merging sorted subarrays (**merge**), and ensuring the graceful shutdown of the Fork/Join pool (**forkJoinPool**). Additionally, the class includes a utility method for printing the elements of an array (**printArray**).
3. **SequentialMergeSort Class**: The class implements the original merge sort algorithm which I took from CPSC 200 course. It offers methods for sorting arrays both sequentially (sequentialMergeSort) and with custom comparators (sequentialMergeSort with comparator). Additionally, it provides an overloaded method for performing sequential merge sort with custom comparators and temporary arrays (sequentialMergeSort with temporary array and comparator). The sequential merge sort algorithm recursively divides the array into subarrays, sorts them individually, and then merges the sorted subarrays. This class is utilized for comparison purposes and as a fallback option when parallelization is not feasible or efficient.
4. **ParallelStepByStepRepresentation Class:** The class provides a step-by-step representation of the parallel merge sort algorithm's execution. It illustrates the process of parallel merge sort by printing the original array, the merging of subarrays, and the sorted array along with the time taken for each step. This class serves as a valuable educational tool for understanding the parallelization of merge sort.

Main Functionality and Features:

The primary functionality implemented in the codebase revolves around a parallel merge sort algorithm designed to efficiently sort arrays of integers. This algorithm harnesses the power of concurrency by utilizing multiple threads to divide the input array concurrently and recursively into smaller subarrays. Each subarray is then independently sorted, leveraging the available processing power for enhanced performance. Subsequently, the sorted subarrays are merged to generate the final sorted output.

Concurrency Approach:

Concurrency is utilized in the project to leverage the available processing power of modern multi-core processors and achieve parallel execution of sorting tasks. By concurrently sorting partitions of the input array using multiple threads, the algorithm can distribute the sorting workload across multiple CPU cores, leading to improved performance and faster sorting times. This approach is particularly beneficial for sorting large datasets, where parallelization can significantly reduce the overall sorting time.

Concurrency Patterns or Techniques Employed:

1. Parallelization with Threads: The project harnesses Java's concurrency utilities, specifically the Fork/Join framework, to parallelize the sorting process. Multiple threads are created within a ForkJoinPool, with each thread responsible for sorting a partition of the input array independently. This pattern allows the algorithm to exploit parallelism and make efficient use of available CPU cores, thereby speeding up the sorting process.
2. Graceful Shutdown of Executor Service: To ensure proper cleanup and resource management, the project gracefully shuts down the ForkJoinPool after sorting is completed. This pattern involves calling the **shutdown** method on the ForkJoinPool to initiate the shutdown process and await the termination of all threads in the pool. By shutting down the ForkJoinPool in this manner, the algorithm releases all acquired resources, terminates threads, and prevents resource leaks or thread starvation.

Concurrency Tools:

The project utilizes Java's Fork/Join framework for parallelizing the sorting process. This framework simplifies concurrent programming by abstracting away the complexities of thread creation and management. By employing a ForkJoinPool, the Fork/Join framework manages a pool of worker threads, distributing sorting tasks across them efficiently. The framework handles thread lifecycle management, including thread creation, execution, and termination, ensuring optimal resource utilization and thread safety during parallel sorting tasks.

Code Walkthrough:

1. **ImplementationAndTesting Class:**
    - **Purpose:** This class serves as the entry point of the application and contains test cases to validate the functionality of the ParallelMergeSort class.
    - **Functionality:** Test cases involve initializing arrays with different sizes and contents, sorting them using ParallelMergeSort, and printing the sorted results.
2. **ParallelMergeSort Class:**
    - Sequential Merge Sort Method (sequentialMergeSort):

1. **Functionality:** Performs sequential merge sort on a specified range of the array.
2. **Details:** Recursively divides the array into halves until each subarray contains only one element (base case). Then, it merges the sorted subarrays using the merge method.

- Merge Method (merge):
  1. **Functionality:** Combines two sorted subarrays into a single sorted array.
  2. **Details:** Iterates through both subarrays, comparing elements and copying them to the result array in sorted order. Ensures stability in sorting by maintaining the order of equal elements from the original array. Finally, copies the merged elements back to the original array.
- Print Array Method (printArray):
  1. **Functionality:** Prints the elements of the array.
  2. **Details:** Iterates through the array and prints each element followed by a space, ensuring readability.
- Parallel Merge Sort Task (parallelMergeSort):
  1. **Functionality:** Represents a ForkJoinTask for parallel merge sort.
  2. **Details:** Divides the array into smaller tasks recursively until the threshold for sequential merge sort is reached. At that point, it performs sequential merge sort on the subarray. For larger subarrays, it splits them into halves and creates tasks to sort each half independently. After sorting, it merges the sorted halves using the merge method.
- Compute Method (compute):
  1. **Functionality:** Executes the parallel merge sort algorithm.
  2. **Details:** Checks if the size of the subarray is below the sequential threshold. If so, performs sequential merge sort. Otherwise, splits the subarray into halves, creates tasks for each half, and invokes them in parallel using the ForkJoinPool. After the tasks are completed, merges the sorted halves.

3. **SequentialMergeSort Class:**
   - **Purpose:** This class implements the sequential merge sort algorithm, primarily used for comparison and testing purposes.
   - **Functionality:** Provides methods for sequential merge sort, including sorting a partition of the array and merging sorted subarrays.

4. **ParallelStepByStepRepresentation Class:**
    - **Purpose:** This class demonstrates the step-by-step execution of the parallel merge sort algorithm, providing insights into the sorting process and thread coordination.
    - **Functionality:** Executes the parallel merge sort algorithm on a sample array and prints step-by-step details of the sorting process, including thread IDs and merged subarrays.

Performance:

1. Scalability:

    The codebase demonstrates good scalability, especially in scenarios with large datasets and multi-core CPUs. As the size of the input array increases, the algorithm can efficiently distribute the sorting workload among multiple threads, leveraging parallelism to scale with the available hardware resources.

2. Efficiency in Concurrent Scenarios:

    The codebase is efficient in concurrent scenarios, where multiple threads concurrently sort different partitions of the input array. By minimizing contention and ensuring proper synchronization, the algorithm achieves high efficiency in utilizing available CPU cores and minimizing idle time.

3. Trade-offs:

    The parallel overhead, stemming from task management and synchronization, must be balanced against the benefits of parallelism, requiring thoughtful selection of the sequential threshold. Load balancing challenges may arise due to recursive array splitting, and the extra memory usage for the merging phase could be a concern in memory-constrained environments.

4. Worst and Average Running Time:

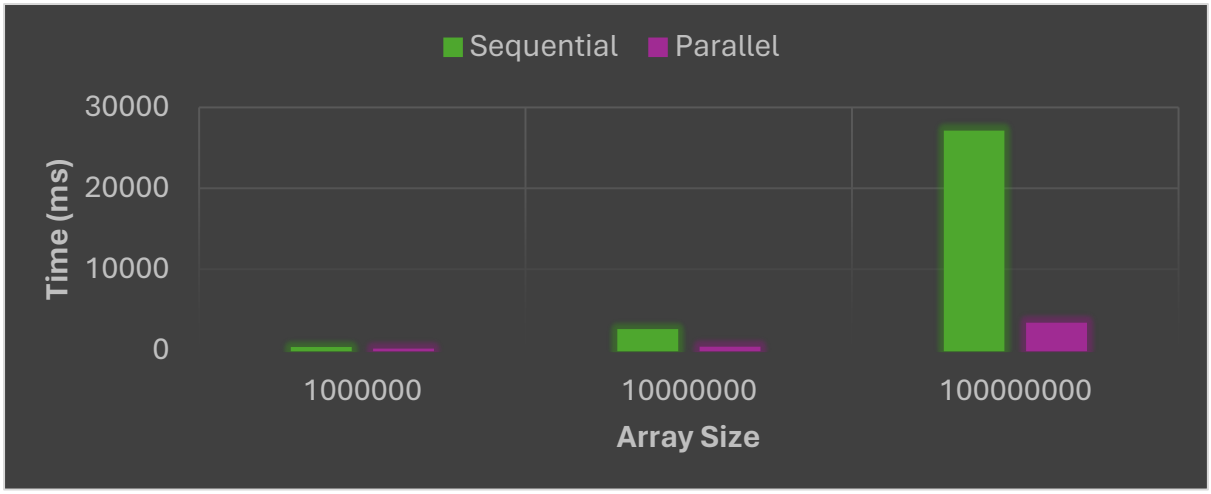
    Theoretically, the worst-case time complexity of parallel merge sort is **Θ(n log n)**, where n is the number of elements in the array. This is because each level of the recursion involves splitting the array into halves, which takes **Θ(log n)** time, and each merge operation takes **Θ (n)** time. The average-case time complexity of parallel merge sort is also **Θ(n log n)**. While parallelism can speed up the sorting process, the overall time complexity remains the same.

5. Extra Space Complexity:

    The extra space complexity of the parallel merge sort remains $O(n)$ because each thread still requires additional space to store its portion of the array and the merged result. Even though the merging process may involve

combining smaller arrays in parallel, the overall space complexity does not change because the total size of the arrays being merged remains ***n***.

Time-Comparison Graph



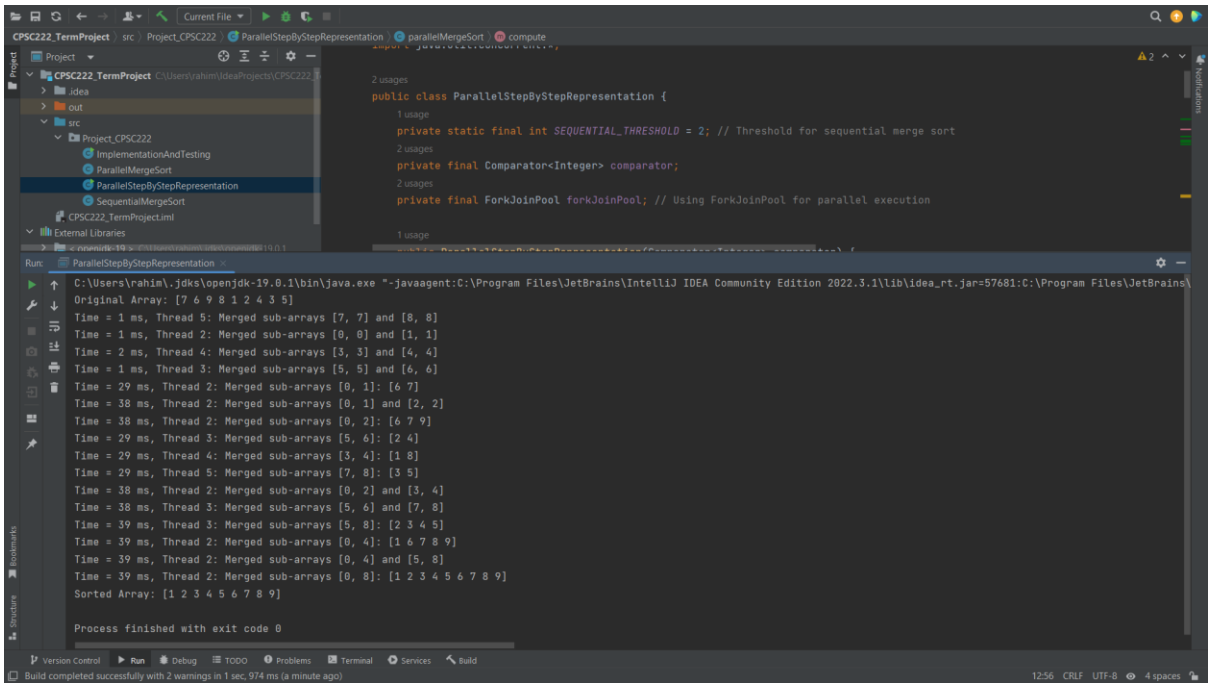
Testing and Validation:

1. Basic Test Cases:
    - Small Input Arrays: The array conducted using small input arrays to validate the basic functionality and correctness of the algorithm. These test cases encompass arrays with a few elements to ensure the sorting algorithm works effectively.
    - Positive and Negative Integers: The array contains both positive and negative integers to verify that the sorting algorithm functions correctly regardless of the sign of the elements.

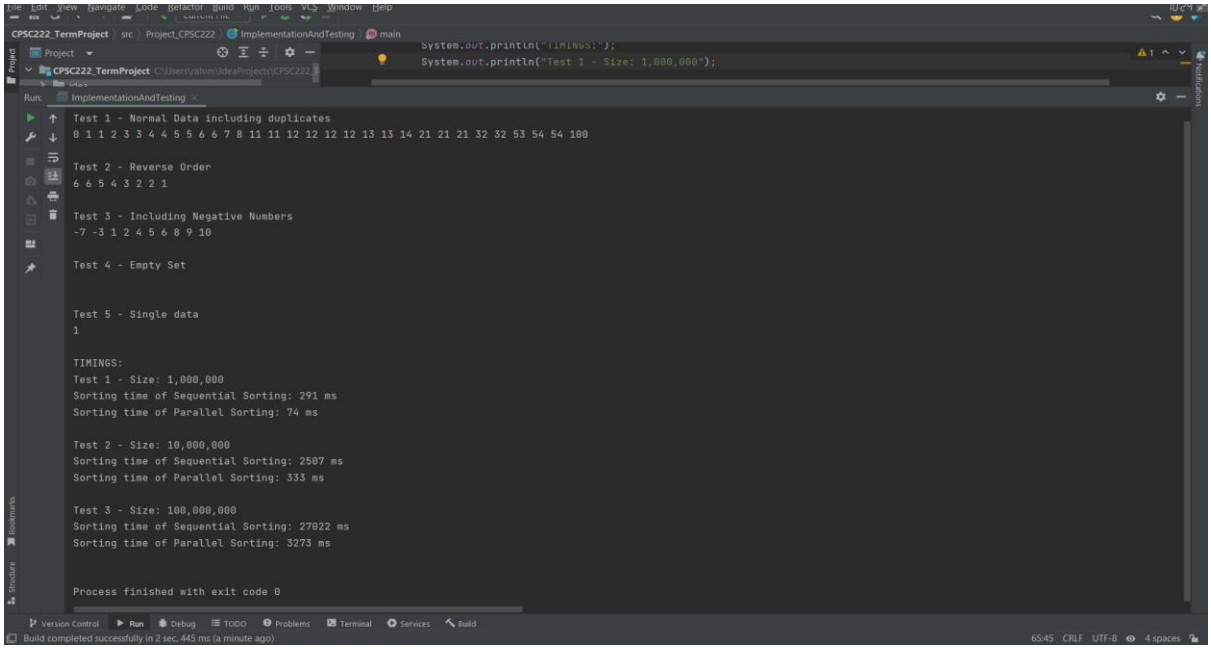
2. Edge Cases:
    - Empty Arrays: It includes an empty array to ensure that the algorithm gracefully handles this edge case without errors or unexpected behaviour.
    - Single Element Arrays: It consists of a single element used to validate the algorithm's behaviour when sorting minimal input sizes.
    - Arrays with Duplicate Elements: It involves arrays with duplicate elements to verify that the algorithm accurately handles repetitions and does not overlook or misplace duplicate values.
3. Performance Testing:

- It involves evaluating the algorithm's efficiency and scalability with large datasets. Test cases include arrays with a significant number of elements to measure the sorting algorithm's performance in terms of execution time and resource utilization.

Step-by-Step Representation



Original Code:



Coding Process and Difficulties:

I initially attempted to parallelize the merge sort algorithm, which originally operated sequentially on a single thread. My approach involved exploring the algorithm and adapting it to leverage parallelism. Initially, I tried modifying the existing code within

the main method to achieve parallelism, but encountered challenges and pitfalls, including deadlocks and hanging threads at Unsafe.park(). Despite experimenting with various synchronization methods such as semaphores and latches, I eventually settled on using a ReentrantLock. However, even after several iterations, I realized that the sorting was still occurring on a single thread, failing to achieve the desired parallelism. Subsequently, I introduced the ExecutorService to manage multiple threads, enabling simultaneous sorting of data segments. Despite encountering issues with incorrect sorting results initially, further adjustments to the logic and code structure eventually led to successful data sorting. After searching the internet a little more and looking at the code, I figured out the algorithm isn't utilizing the parallelism fully due to the synchronization as it made the threads go one by one to the critical section which didn't let it do the task in parallel. Thus, after removing the lock, I used the Fork/Join framework instead of ExecutorService as it ensures thread safety as well.

Real World Application:

The parallelized sorting algorithm developed can significantly benefit a wide range of applications by improving data processing performance, scalability, and efficiency in concurrent and distributed computing environments. It has numerous real-world applications across various domains:

1. Big Data Processing: In environments dealing with massive datasets, such as data warehouses, analytics platforms, and distributed file systems like Hadoop or Spark, parallel sorting algorithms are crucial for efficient processing and analyzing data. Sorting is often a preliminary step in many data processing tasks, including data cleaning, aggregation, and analysis.
2. Database Systems: Database management systems (DBMS) frequently utilize sorting algorithms for query processing, indexing, and result retrieval. Parallel sorting can enhance the performance of operations like sorting query results, building indexes, and executing joint operations by leveraging multiple CPU cores to process data concurrently.
3. Scientific Computing: Parallel sorting algorithms play a vital role in scientific simulations, numerical computations, and data analysis in fields like physics, biology, and engineering. Sorting is often a fundamental operation in algorithms for solving computational problems, such as searching, graph algorithms, and numerical simulations.
4. Financial Trading: High-frequency trading systems and financial analytics platforms rely on efficient sorting algorithms to process large volumes of financial data in real-time. Parallel sorting can improve the performance of tasks like market data analysis, order matching, and risk assessment by enabling faster data processing and decision-making.
5. Multimedia Processing: Applications in multimedia processing, such as image and video processing, often require sorting algorithms for tasks like image filtering, feature extraction, and content-based retrieval. Parallel sorting can accelerate these tasks, allowing for real-time processing of multimedia data

streams and improving the user experience in applications like video streaming and content recommendation systems.

6. E-commerce and Recommendation Systems: Sorting algorithms are essential for ranking and recommending products, services, and content in e-commerce platforms, recommendation engines, and personalized marketing systems. Parallel sorting can enhance the efficiency of product recommendation algorithms by enabling faster sorting of user preferences, purchase history, and product attributes.

Conclusion:

The project successfully demonstrates the design and implementation of a parallel merge sort algorithm within the domain of concurrent and distributed programming. By leveraging concurrency principles and synchronization mechanisms, the algorithm achieves efficient and reliable sorting of arrays, making effective use of available processing power and resources. The codebase exhibits good scalability, performance, and robustness, validated through comprehensive testing and validation efforts. Overall, the project contributes to a deeper understanding of concurrency challenges and provides practical experience in developing parallel algorithms for real-world applications.

References:

CPSC 200: Algorithm Analysis and Development course slides.

CPSC 222: Concurrent and Distributed Programming course slides.

W3schools.com

Stackoverflow.com

Google.com (To search real-life applications)

Youtube.com

Word count (2551 words)