Parallel Merge Sort Implementation

Erik Saule

Goal: Implement a parallel merge sort algorithm in C++ that efficiently sorts large arrays by leveraging multithreading capabilities. The implementation should dynamically determine when to use parallel processing versus sequential sorting based on array size.

1 Programming

For reference, here is an overview of merge sort from Geeks for Geeks:

Merge Sort - GeeksforGeeks

Here's a step-by-step explanation of how merge sort works:

- Divide the unsorted array into two halves.
- Recursively sort each half by applying merge sort.
- Merge the two sorted halves to produce one sorted array.

Your task is to implement a parallel version of this algorithm that can effectively utilize multiple threads to improve performance for large arrays.

2 TODO: Parallel Merge Sort Implementation

The parallel implementation should follow the divide-and-conquer approach of merge sort while utilizing threads for parallel processing. Your implementation should:

- Start with a standard merge sort implementation.
- Add parallel processing capabilities using the `std::thread` library.
- Implement a recursive approach that spawns threads for large array segments.
- Determine intelligently when to use parallel processing vs. sequential sorting.

2.1 Array Size Threshold

A critical aspect of your implementation is determining when to use parallel processing:

- For large array segments, create separate threads to sort the left and right halves concurrently.
- For smaller array segments, use sequential sorting to avoid the overhead of thread creation.
- Implement a threshold mechanism that switches between parallel and sequential sorting based on array size.
- The threshold should be chosen to optimize performance based on the tradeoff between parallelization benefits and thread creation overhead.

2.2 Thread Management

Proper thread management is essential for efficient parallel sorting:

- Create threads only when the array size justifies the overhead.
- Use `join()` to wait for threads to complete their sorting tasks.
- Ensure proper synchronization when merging sorted sub-arrays.
- Avoid race conditions by carefully managing access to shared data.

2.3 Example Strategy

A typical implementation strategy might include:

- Define a threshold (e.g., several hundred or thousand elements).
- If the array segment is larger than the threshold, sort the two halves in parallel using separate threads.
- If the array segment is smaller than the threshold, use sequential merge sort.
- Join the threads after parallel sorting and merge the sorted halves.

3 Merge Operation

The merge operation combines two sorted sub-arrays into a single sorted array:

- Implement a standard merge function that takes two sorted arrays and merges them.
- Ensure the merge operation is thread-safe if multiple threads might access it.

4 Benchmarking and Testing

TODO: Evaluate the performance of your parallel merge sort implementation.

- Generate test arrays of various sizes (small, medium, large).
- Measure the execution time for both sequential and parallel implementations.
- Compare performance across different array sizes to identify the optimal threshold.
- Analyze the speedup achieved through parallelization.

5 Submission

TODO: Submit an archive containing:

- Your C++ source code.
- A Makefile for compiling the code.
- A README explaining how to run the program.
- Performance measurement results comparing sequential and parallel execution.