Parallel Merge Sort Implementation (with Parallel Tasking Construct)

Erik Saule

Goal: Implement a parallel merge sort algorithm in C++ that efficiently sorts large arrays by leveraging multithreading capabilities. The implementation must use the **provided parallel tasking abstraction** (`omp_tasking.hpp`) to create and manage parallel tasks.

Before starting, make sure you understand how the sequential merge sort algorithm works and how recursive divide-and-conquer can be expressed using parallel tasks.

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 utilizes **tasks** for parallel processing. You must use the tasking construct provided in `omp_tasking.hpp` (specifically `doinparallel`, `taskstart`, and `taskwait`) rather than using raw OpenMP pragmas.

2 TODO: Parallel Merge Sort Implementation

The parallel implementation should follow the divide-and-conquer approach of merge sort while leveraging parallel tasks for parallelism. Your implementation should:

- Start with a standard sequential merge sort implementation.
- Add parallelism using the provided tasking abstraction:
  - Use `doinparallel()` to set up the parallel environment and number of threads.
  - Use `taskstart()` to spawn recursive tasks for sorting subarrays in parallel.
  - Use `taskwait()` to synchronize and ensure all child tasks complete before merging results.
- Implement recursion such that both left and right halves can be sorted concurrently as independent tasks.

2.1 Array Size Threshold

A critical aspect of your implementation is determining when to use parallel processing versus sequential sorting. This threshold prevents task creation overhead from dominating runtime.

- For large array segments (e.g., subarrays larger than 1000 elements), create separate tasks for sorting the left and right halves concurrently.
- For smaller subarrays, perform sequential merge sort.
- Implement a threshold mechanism that switches between parallel and sequential sorting based on array size.

2.2 Task Management

Proper task management is essential for efficient and correct parallel sorting. Keep in mind the following:

- Always use the provided tasking functions.
- Each recursive call that spawns tasks must also include a `taskwait()` before merging results to ensure that all child tasks have completed.
- Avoid creating tasks when the array segment is too small.
- Ensure that merging only happens after both subarrays are sorted.
- Avoid race conditions by carefully managing access to shared data

3 Benchmarking and Testing

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

- Generate test arrays of various sizes (small, medium, large).
- Measure and record execution times for both the sequential and parallel implementations.
- Compare results across different array sizes and task thresholds to analyze performance.
- Test with different thread counts using the command-line parameter to `doinparallel()` (e.g., 1, 2, 4, 8 threads).
- Analyze the speedup achieved through task-based parallelization.
- Remember that you need to compile with `-fopenmp`.

4 Submission

TODO: Submit an archive containing:

- Your C++ source code.
- A Makefile for compiling the code.
- A README file explaining how to compile and run the program.
- Performance results comparing sequential and parallel runs with different array sizes and thread counts.