Assignment 1: Parallel Merge Sort with OpenMP

Due Date: 11:59PM, Aug 26, 2024

Problem Description

You are required to implement both sequential and parallel versions of the Merge Sort algorithm, which is a classic sorting algorithm that employs a divide-and-conquer strategy to sort elements. Your parallel implementation should use OpenMP task constructs to leverage multiple cores. Additionally, you will measure and plot the execution time for each combination of data size and core count, calculate the speedup and efficiency of your implementation, and analyze cache performance using profiling tools. Finally, on the sorted array, you will perform a parallelised version of binary search.

Input

We will sort an array of type "Record", which is defined as follows:

```
struct Record {
    int key;
    char char_data[1024];
    vector<int> int_data;
};
```

Here, the size of int_data will be constant for all "Records" in a test case.

Tasks

1. **Merge Sort:** Implement the standard version of Merge Sort. Sort on "key". Sort in ascending order. The order of those records with the same key does not matter. Modify the array passed in the arguments.

```
void merge_sort(vector<Record> &arr);
```

2. **Parallel Merge Sort:** Design and implement a parallel version of Merge Sort using OpenMP task constructs.

```
void parallel_merge_sort(vector<Record> &arr);
```

3. **Binary Search:** Implement the standard version of a binary search.

```
vector<Record> binary_search(vector<Record> &arr, int key);
```

4. **Parallel Binary Search:** Implement a parallel binary search algorithm that divides the search range among n threads, where n is the number of threads available. Each thread should handle a portion of the array and perform binary search within that portion. How you divide the array within the threads (statically / dynamically) is your design decision and you have to explain this in your report.

```
vector<Record> parallel_binary_search(vector<Record> &arr, int key);
```

Test Cases

Your code will be tested on 3 files:

1. **small.txt** - contains 10^6 records
2. **medium.txt** - contains 10^7 records
3. **large.txt** - contains 10^8 records

Report

Submit a brief report discussing the following:

1. **Your implementation** - Your design decisions
2. **Challenges faced**
3. **Performance analysis**
   (a) Measure the execution time of both the sequential and parallel merge sort implementations. Test with 2, 4, 8, 16, 32, 40 threads. Plot the execution time for each combination of data size (small, medium, large) and thread count. Calculate the speedup and efficiency of the parallel implementation.
   (b) Analyze the time taken and the total number of comparisons per search required by both implementations of your binary search. Test with 2, 4, 8, 16, 32, 40 threads. Calculate the speedup and efficiency of the parallel implementation.
4. **Cache Performance Analysis:** Use tools like `perf` or other cache profiling tools to measure cache hits, misses, and other relevant metrics. Compare the cache performance of the sequential and parallel implementations.

Grading Criteria

0 Marks will be awarded in case you use any built-in sorting functions.

1. **Correctness** (marks awarded only if sequential version is correct)
   (a) Parallel Merge Sort - 20
   (b) Parallel Binary Search - 20
2. **Speed**
   (a) Parallel Merge Sort - 20
   (b) Parallel Binary Search - 20
3. **Report** - 20

Submission Instructions

You are provided with a header file **functions.h** that contains the declarations of four functions. Implement these functions in a file named **functions.cpp**. Create a report in PDF format named **report.pdf**. Place these files in a folder named with your entry number (e.g., 2020CS50886). DO NOT change or submit **functions.h**. Compress the folder named <**ENTRY_NUMBER**> into a ZIP file. Ensure that the ZIP file is named <**ENTRY_NUMBER**>**.zip**

```
2020CS50475/
|-- functions.cpp
|-- report.pdf
```