Explanation of Parallel Sorting Practical

1 Overview of the Practical

This practical focuses on implementing and comparing **Bubble Sort** and **Merge Sort** algorithms in both **sequential** (single-threaded) and **parallel** (multi-threaded) versions using **OpenMP**, a parallel programming tool for C/C++. The objectives are:

- Write programs for sequential and parallel Bubble Sort and Merge Sort.
- Measure performance on a dataset of 10,000 random numbers.
- Compare execution times to determine efficiency.
- Analyze the algorithms' time complexity and parallelization strategies.

2 Code Explanation

The code implements four sorting algorithms and measures their performance. Below is a breakdown of each component.

2.1 Sequential Bubble Sort

Purpose: Sorts an array by repeatedly comparing and swapping adjacent elements.

How it Works:

- Loops through the array multiple times.
- Compares adjacent elements and swaps them if they are in the wrong order.
- After each pass, the largest unsorted element moves to its correct position.
- Runs on a single thread.

Code:

```
void bubbleSortSequential(vector<int> &arr) {
    int n = arr.size();
    for(int i = 0; i < n-1; ++i)
        for(int j = 0; j < n-i-1; ++j)
            if(arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
}
```

2.2 Parallel Bubble Sort

Purpose: Uses OpenMP to parallelize comparisons for faster sorting.

How it Works:

- Splits comparison tasks across multiple threads using OpenMP.
- Alternates between even and odd indices to avoid thread conflicts.
- Each thread handles a subset of comparisons.

Code:

```
void bubbleSortParallel(vector<int> &arr) {
    int n = arr.size();
    for(int i = 0; i < n; ++i) {
        #pragma omp parallel for
        for(int j = i % 2; j < n-1; j += 2) {
            if(arr[j] > arr[j+1])
                swap(arr[j], arr[j+1]);
        }
    }
}
```

2.3 Sequential Merge Sort

Purpose: Divides the array into smaller subarrays, sorts, and merges them.

How it Works:

- Recursively divides the array into two halves until each subarray has one element.
- Merges subarrays by comparing and combining elements in sorted order.
- Runs on a single thread.

Code:

```
void mergeSortSequential(vector<int>& arr, int left, int right) {
    if(left < right) {
        int mid = (left + right) / 2;
        mergeSortSequential(arr, left, mid);
        mergeSortSequential(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

2.4 Parallel Merge Sort

Purpose: Parallelizes the recursive division of the array.

How it Works:

- Uses OpenMP to run the sorting of left and right subarrays in parallel threads.
- Merges the sorted subarrays sequentially.

Code:

```
void mergeSortParallel(vector<int>& arr, int left, int right) {
    if(left < right) {
        int mid = (left + right) / 2;
        #pragma omp parallel sections
        {
            #pragma omp section
            mergeSortParallel(arr, left, mid);
            #pragma omp section
            mergeSortParallel(arr, mid + 1, right);
        }
        merge(arr, left, mid, right);
    }
}
```

2.5 Main Function

Purpose: Sets up the experiment and measures performance.

How it Works:

- Creates four identical arrays of 10,000 random numbers.
- Measures execution time for each algorithm using `omp_get_wtime()`.
- Prints times and compares sequential vs. parallel performance.
- Identifies the fastest algorithm.

3 Output Explanation

The output shows the execution times and performance comparison:

```
Sequential Bubble Sort Time: 0.795000 seconds
Parallel Bubble Sort Time:   1.043000 seconds
Sequential Merge Sort Time: 0.007000 seconds
Parallel Merge Sort Time:   0.045000 seconds

Efficiency Summary:
Sequential Bubble Sort is faster.
Sequential Merge Sort is faster.

Best Performer Overall: Sequential Merge Sort
```

Key Points:

- **Sequential Bubble Sort** (0.795s) was faster than Parallel Bubble Sort (1.043s) due to thread creation overhead.
- **Sequential Merge Sort** (0.007s) was faster than Parallel Merge Sort (0.045s) for the same reason.
- **Sequential Merge Sort** was the fastest overall because Merge Sort has a better time complexity (O(n log n)) than Bubble Sort ($O(n^2)$).

4 Answers to Review Questions

1. **Time Complexity of Parallel Bubble Sort**:
   - Worst-case complexity remains $O(n^2)$, but parallelism can reduce run-time to $O(n^2/p)$ for p threads, assuming no overhead. In practice, thread synchronization reduces efficiency.
2. **Difference Between Parallel Bubble Sort and Merge Sort**:
   - Bubble Sort parallelizes comparisons, but swaps depend on prior swaps, limiting parallelism.
   - Merge Sort parallelizes independent recursive divisions, making it more suitable for parallelism.
3. **Scaling Parallel Merge Sort**:
   - Scales well for large datasets as recursive tasks are independent. The sequential merge step can be a bottleneck for very large arrays.
4. **Parallelizing Bubble Sort**:

- Divide comparisons across threads, alternating between even and odd indices to avoid conflicts.

5 How to Explain to the Examiner

To present this practical confidently to the examiner, follow these steps:

1. **Introduce the Practical**:
   - “This practical involves implementing Bubble Sort and Merge Sort in sequential and parallel versions using OpenMP. We test them on 10,000 random numbers, measure their execution times, and compare performance to find the most efficient algorithm.”
2. **Explain the Code**:
   - **Sequential Bubble Sort**: “It sorts by comparing and swapping adjacent elements in multiple passes, running on one thread.”
   - **Parallel Bubble Sort**: “It uses OpenMP to split comparisons across threads, alternating even and odd indices to avoid conflicts.”
   - **Sequential Merge Sort**: “It divides the array into smaller parts, sorts them recursively, and merges them, all on one thread.”
   - **Parallel Merge Sort**: “It uses OpenMP to sort the left and right halves in parallel threads, then merges them sequentially.”
   - **Main Function**: “We create four identical arrays, run each algorithm, measure time with OpenMP’s timer, and compare results.”
3. **Explain the Output**:
   - “Sequential Bubble Sort took 0.795 seconds, Parallel Bubble Sort took 1.043 seconds, so sequential was faster. Sequential Merge Sort took 0.007 seconds, Parallel Merge Sort took 0.045 seconds, so sequential was faster. Sequential Merge Sort was the fastest overall.”
   - “Parallel versions were slower due to the overhead of creating and managing threads, which outweighed the benefits for 10,000 elements.”
4. **Address Review Questions** (if asked):
   - **Time Complexity**: “Parallel Bubble Sort is $O(n^2)$ but can be $O(n^2/p)$ with p threads, though overhead reduces gains.”
   - **Differences**: “Bubble Sort’s swaps depend on each other, limiting parallelism, while Merge Sort’s recursive tasks are independent.”
   - **Scaling**: “Parallel Merge Sort scales well for large datasets, but the se-

quential merge step can limit performance."

   - **Parallelizing Bubble Sort**: "We parallelize comparisons by splitting them across threads and alternating indices."

5. **Explain Why Sequential Was Faster**:

   - "The dataset of 10,000 elements was too small for parallelism to be effective. Thread overhead was higher than the time saved, and Merge Sort's O(n log n) complexity made its sequential version the fastest."

6 Tips for Explanation

- Explain each algorithm briefly: Bubble Sort swaps adjacent elements; Merge Sort divides and merges.
- Highlight OpenMP directives: `#pragma omp parallel for` splits loops, `#pragma omp parallel sections` runs different tasks in parallel.
- Emphasize why sequential was faster: Thread overhead outweighed parallelism benefits for 10,000 elements.
- Be ready to discuss time complexity and parallelization strategies.