

Problem Statement (Simple & Clear)

Write a program in C++ to implement both sequential and parallel versions of Bubble Sort and Merge Sort using OpenMP. Measure and compare the performance (execution time) of sequential and parallel implementations.

Objectives

1. Implement:
 - **Sequential Bubble Sort**
 - **Parallel Bubble Sort** using OpenMP
 - **Sequential Merge Sort**
 - **Parallel Merge Sort** using OpenMP
 2. Use `omp_get_wtime()` to measure execution time of all four.
 3. Input: An array of integers (you can use randomly generated or user input).
 4. Output:
 - Sorted array (optional, for small sizes)
 - Execution times for comparison
-

Approach

◆ **Bubble Sort (Parallel)**

- Bubble Sort has limited parallelism.
- Use **odd-even transposition** technique for parallelism:
 - One iteration for odd indices
 - Another for even indices
- Use `#pragma omp parallel` for those swaps.

◆ **Merge Sort (Parallel)**

- More naturally parallel.
- Divide array recursively.
- Use `#pragma omp task` for each recursive call.
- Combine using merge.

Objectives

1. Use OpenMP to **parallelize** the sorting algorithms:
 - Bubble Sort using `#pragma omp parallel for`
 - Merge Sort using `#pragma omp parallel sections`
 2. Take user input for the array.
 3. Sort the array using both algorithms.
 4. Measure execution time using `clock_t` and print the sorted arrays and durations.
-

Approach Summary

◆ Bubble Sort (Parallel)

- Normally compares adjacent elements and swaps if out of order.
- In this code:
 - A while loop continues until no swaps are made.
 - A **parallel for loop** checks and swaps in one pass.

⚠ Limitation:

- `swapped` is a shared variable, and no synchronization is used.
 - It may cause incorrect behavior or race conditions in real parallel execution.
-

◆ Merge Sort (Parallel)

- Recursive divide-and-conquer algorithm.
- In this code:
 - Uses `#pragma omp parallel sections` to **recursively sort the two halves in parallel**.
 - Merges the sorted halves afterward.

✅ This is a much **better use of OpenMP** than bubble sort.

Step-by-Step Code Explanation

◆ `bubbleSort(vector<int>& arr)`

cpp

```
while (swapped) {
    swapped = false;
    #pragma omp parallel for
    for (int i = 0; i < n - 1; i++) {
        if (arr[i] > arr[i + 1]) {
            swap(arr[i], arr[i + 1]);
            swapped = true;
        }
    }
}
```

- This tries to sort in parallel by allowing all adjacent pairs to be checked at the same time.
 - But swapped is shared and not thread-safe — technically a **race condition**.
 - Works for small data, but not safe for large or real-time parallel computing.
-

◆ mergeSort(...)

cpp

```
#pragma omp parallel sections
{
    #pragma omp section
    mergeSort(arr, l, m);
    #pragma omp section
    mergeSort(arr, m + 1, r);
}
```

- Splits work into **two threads** to sort each half recursively.
- Each section is handled **in parallel** by different threads.
- Then merge() merges the sorted subarrays.

✓ **Correct and effective parallel merge sort** implementation.

◆ **main()**

cpp

```
clock_t bubbleStart = clock();
```

```
bubbleSort(arr);
```

```
clock_t bubbleEnd = clock();
```

- Records time before and after bubble sort.

Then does the same for merge sort:

cpp

```
clock_t mergeStart = clock();
```

```
mergeSort(arr, 0, n - 1);
```

```
clock_t mergeEnd = clock();
```

✓ Execution time is calculated and printed for comparison.

✓ **Sample Input & Output**

Input:

yaml

CopyEdit

Enter the number of elements: 6

Enter the elements: 5 3 1 6 4 2

Output:

sql

]

Sorted array using Bubble Sort: 1 2 3 4 5 6





Sorted array using Merge Sort: 1 2 3 4 5 6

Bubble sort time in seconds: 0.0002

Merge sort time in seconds: 0.00005

(Note: Time will vary based on machine and input)

Improvements You Can Mention

Problem	Suggestion
 Bubble sort has race condition on swapped	Use OpenMP reduction or atomic to make it safe
 No sequential version included for fair comparison	Add pure sequential versions and compare
 Merge sort is fine	Could also use <code>#pragma omp task</code> for deeper parallel recursion
 Time is shown	For large arrays, use <code>omp_get_wtime()</code> instead of <code>clock()</code> for better accuracy

What to Say in Viva

"This program demonstrates parallel sorting using OpenMP. I parallelized Bubble Sort using `#pragma omp parallel for` and Merge Sort using `#pragma omp parallel sections`. Merge Sort benefits more from parallelism because it's recursive and naturally divides tasks. I also measured the time taken by both algorithms and compared their performance."

What is `#pragma`?

`#pragma` is a **preprocessor directive** in C and C++.

- It gives **special instructions** to the **compiler**.
 - The compiler uses these instructions to **optimize** or **alter behavior**.
 - It does **not** affect the actual logic of the program — it's like a *hint* to the compiler.
-

What is `#pragma omp`?

When you're using **OpenMP** (Open Multi-Processing), you use `#pragma omp` to tell the compiler:

"Hey! I want to run the following part of the code in **parallel** using multiple threads."

It activates **multi-threading** for loops, sections, or tasks.

✓ **Why #pragma omp is Used (in your code)**

Example	What it does
#pragma omp parallel for	Runs a for loop in parallel using multiple threads
#pragma omp parallel sections	Runs multiple code sections simultaneously
#pragma omp section	Defines a block to run in a parallel section