Parallel Merge Sort (C++ with OpenMP)

```
#include <iostream>
#include <vector>
#include <omp.h>

// Function to merge two sorted halves
void merge(std::vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1, n2 = right - mid;
    std::vector<int> leftArr(n1), rightArr(n2);

    // Copy elements into temporary left and right arrays
    for (int i = 0; i < n1; i++) leftArr[i] = arr[left + i];
    for (int i = 0; i < n2; i++) rightArr[i] = arr[mid + 1 + i];

    // Merge process: Compare elements and place in sorted order
    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        arr[k++] = (leftArr[i] < rightArr[j]) ? leftArr[i++] : rightArr[j++];
    }

    // Copy remaining elements from left and right arrays (if any)
    while (i < n1) arr[k++] = leftArr[i++];
    while (j < n2) arr[k++] = rightArr[j++];
}

// Recursive parallel merge sort function
void mergeSort(std::vector<int>& arr, int left, int right) {
    if (left >= right) return; // Base case: array of size 1 is already sorted

    int mid = left + (right - left) / 2;

    // Parallel processing using OpenMP sections
    #pragma omp parallel sections
    {
        #pragma omp section
        mergeSort(arr, left, mid);  // Sort left half

        #pragma omp section
        mergeSort(arr, mid + 1, right);  // Sort right half
```

```
    }

    // Merge the sorted halves
    merge(arr, left, mid, right);
}

// Main function to execute the sorting algorithm
int main() {
    std::vector<int> data = {38, 27, 43, 3, 9, 82, 10};
    mergeSort(data, 0, data.size() - 1);

    // Print sorted output
    std::cout << "Sorted Data: ";
    for (int num : data) std::cout << num << " ";
    return 0;
}
```

Explanation:

- The **mergeSort function** recursively divides the array into two halves.
- **OpenMP sections** allow the left and right halves to be sorted simultaneously.
- The **merge function** merges the sorted halves in a standard way.

Parallelization Key Point:

- `#pragma omp parallel sections` ensures the sorting of left and right halves happens **at the same time** using multiple threads.

Parallel Quick Sort (C++ with OpenMP)

```
#include <iostream>

#include <vector>

#include <omp.h>


// Partition function for quicksort
```

```
int partition(std::vector<int>& arr, int low, int high) {
    int pivot = arr[high];  // Choosing pivot element
    int i = low - 1;  // Index for placing elements smaller than pivot

    // Swap elements based on pivot comparison
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) std::swap(arr[++i], arr[j]);
    }
    std::swap(arr[++i], arr[high]);  // Place pivot in correct position
    return i;  // Return pivot index
}

// Recursive parallel quicksort function
void quickSort(std::vector<int>& arr, int low, int high) {
    if (low < high) {
        int pivot = partition(arr, low, high);  // Partition array

        // Parallel execution of recursive calls
        #pragma omp parallel sections
        {
            #pragma omp section
            quickSort(arr, low, pivot - 1);  // Sort left partition
```

```
            #pragma omp section
            quickSort(arr, pivot + 1, high);  // Sort right partition
        }
    }
}


// Main function to execute the sorting algorithm
int main() {
    std::vector<int> data = {38, 27, 43, 3, 9, 82, 10};
    quickSort(data, 0, data.size() - 1);


    // Print sorted output
    std::cout << "Sorted Data: ";
    for (int num : data) std::cout << num << " ";
    return 0;
}
```

Parallel Quick Sort (C++ with OpenMP)

Explanation:

- The **partition function** selects a pivot and rearranges elements into left and right parts.
- **OpenMP sections** allow recursive sorting of left and right partitions **in parallel**.

Parallelization Key Point:

- `#pragma omp parallel sections` speeds up sorting by **processing both partitions simultaneously** instead of sequential recursion.

Parallel Radix Sort (C++ with OpenMP)

```
#include <iostream>

#include <vector>

#include <omp.h>


// Function for counting sort used in radix sort

void countingSort(std::vector<int>& arr, int exp) {

    std::vector<int> output(arr.size());

    int count[10] = {0};  // Count array for digit occurrences


    // Count occurrences of each digit in input array

    #pragma omp parallel for

    for (int i = 0; i < arr.size(); i++) count[(arr[i] / exp) % 10]++;


    // Convert counts into cumulative counts

    for (int i = 1; i < 10; i++) count[i] += count[i - 1];


    // Build sorted output array based on digit position

    #pragma omp parallel for

    for (int i = arr.size() - 1; i >= 0; i--) {

        output[--count[(arr[i] / exp) % 10]] = arr[i];
```

```
    }

    // Copy sorted output array back to original array
    #pragma omp parallel for
    for (int i = 0; i < arr.size(); i++) arr[i] = output[i];
}

// Parallel radix sort function
void radixSort(std::vector<int>& arr) {
    int maxVal = *std::max_element(arr.begin(), arr.end());

    // Process each digit place using counting sort
    for (int exp = 1; maxVal / exp > 0; exp *= 10) {
        countingSort(arr, exp);
    }
}

// Main function to execute the sorting algorithm
int main() {
    std::vector<int> data = {170, 45, 75, 90, 802, 24, 2, 66};
    radixSort(data);

    // Print sorted output
```

```
    std::cout << "Sorted Data: ";
    for (int num : data) std::cout << num << " ";
    return 0;
}
```

Parallel Bucket Sort (C++ with OpenMP)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <omp.h>

// Function to execute parallel bucket sort
void bucketSort(std::vector<int>& arr, int bucketSize) {
    int minVal = *std::min_element(arr.begin(), arr.end());
    int maxVal = *std::max_element(arr.begin(), arr.end());

    int bucketCount = (maxVal - minVal) / bucketSize + 1;
    std::vector<std::vector<int>> buckets(bucketCount);

    // Assign elements to appropriate buckets
    #pragma omp parallel for
    for (int i = 0; i < arr.size(); i++) {
        int bucketIndex = (arr[i] - minVal) / bucketSize;
```

```
        #pragma omp critical
        buckets[bucketIndex].push_back(arr[i]);
    }

    // Sort each bucket independently in parallel
    #pragma omp parallel for
    for (int i = 0; i < bucketCount; i++) {
        std::sort(buckets[i].begin(), buckets[i].end());
    }

    // Merge sorted buckets back into the array
    int index = 0;
    for (auto& bucket : buckets) {
        for (int num : bucket) {
            arr[index++] = num;
        }
    }
}

// Main function to execute the sorting algorithm
int main() {
    std::vector<int> data = {42, 32, 33, 52, 37, 47, 51};
    bucketSort(data, 5);
```

```
    // Print sorted output
    std::cout << "Sorted Data: ";
    for (int num : data) std::cout << num << " ";
    return 0;
}
```

Parallel Bucket Sort (C++ with OpenMP)

Explanation:

- The array is divided into multiple **buckets**, each bucket containing numbers within a fixed range.
- Each bucket is **sorted in parallel** using OpenMP directives.

Parallelization Key Point:

- `#pragma omp parallel for` distributes **bucket assignment** across multiple threads.
- `#pragma omp critical` ensures **safe updates** when multiple threads add data to buckets.

Summary of Parallelization Benefits

- **Reduced Execution Time:** OpenMP allows multiple sections of the code to execute **simultaneously**, improving sorting speed.
- **Efficient CPU Utilization:** Threads run independently, making use of **multi-core processors**.
- **Scalability:** These algorithms can be extended to large datasets and work **more efficiently** than sequential implementations.