ChatGPT

Bubble Sort (Sequential and Parallel)

Bubble Sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order [1]. In other words, on each pass through the array, the largest remaining element "bubbles up" to its correct position at the end. We keep doing passes until no swaps are needed (the array is sorted). Sequential bubble sort code typically uses two nested loops: the outer loop makes repeated passes, and the inner loop scans through adjacent pairs, swapping if needed. For example, a simple version would be: for each index i from 0 to n–2, compare `arr[j]` and `arr[j+1]` for all j < n–i–1, and swap if out of order; stop early if no swaps occur in a pass (meaning the list is sorted).

- **Sequential Bubble Sort Code:** We initialize a flag (e.g. `swapped=false`) before each pass. In each inner loop, if we swap any pair we set `swapped=true`. After the inner loop, if `swapped` is still false, we can break early since the array is sorted. This avoids unnecessary work. Pseudo-steps:
- `for (i = 0; i < n-1; ++i)` (each pass)
- Set `swapped = false`.
- `for (j = 0; j < n-i-1; ++j)` (scan adjacent pairs)
- If `arr[j] > arr[j+1]`, swap them and set `swapped = true`.
- If no swap happened in this pass (`swapped==false`), break (array is sorted).

This runs in O(n^2) time in the worst case, because of the nested loops [1]. It is easy to implement but inefficient on large arrays.

Parallel Bubble Sort (Odd-Even transposition): A parallel variant known as *odd-even bubble sort* can help divide the work among threads. It alternates between two phases [2]:

- **Even phase:** compare and swap elements at even indices with their right neighbor (pairs (0,1), (2,3), etc.).
- **Odd phase:** compare and swap at odd indices with their right neighbor (pairs (1,2), (3,4), etc.).

Each phase can be done in parallel because each pair of comparisons in that phase are independent (they involve disjoint elements). In OpenMP, we do this by enclosing the work in a parallel region and using `#pragma omp for` to split the loop iterations among threads. A common structure is:

```
bool sorted = false;
while (!sorted) {
    sorted = true;
    #pragma omp parallel private(tmp)
    {
        // Even phase (i=0,2,4,...)
        #pragma omp for reduction(&&:sorted)
        for (int i = 0; i < n-1; i += 2) {
            if (arr[i] > arr[i+1]) {
                swap(arr[i], arr[i+1]);
                sorted = false;
            }
        }
        // Odd phase (i=1,3,5,...)
        #pragma omp for reduction(&&:sorted)
        for (int i = 1; i < n-1; i += 2) {
            if (arr[i] > arr[i+1]) {
                swap(arr[i], arr[i+1]);
                sorted = false;
            }
        }
    }
}
```

Here's how it works:

- We wrap the two phases in `#pragma omp parallel private(tmp)` (each thread has its own `tmp` variable for swapping).
- In the **even phase**, we use `#pragma omp for reduction(&&:sorted)` to divide the loop among threads. Each thread handles a subset of even-indexed swaps in parallel. We use a *reduction* on `sorted` with the logical AND operator: if any swap happens in any thread, that thread sets its private `sorted=false`, and the reduction ensures if *any* thread set `false`, then the combined `sorted` becomes false [3].
- The **odd phase** is similar with another `#pragma omp for reduction(&&:sorted)` loop over odd indices [4].
- After both phases, if `sorted` remains true, the array is fully sorted; otherwise we repeat.

This even-odd pattern is standard for parallelizing bubble sort [2]. The `#pragma omp parallel for` directives split each loop among threads. The **private(tmp)** clause ensures each thread uses its own temporary swap variable, avoiding conflicts. The **reduction(&&:sorted)** clause combines the `sorted` flags from all threads: it computes the logical AND of all thread-local values, so if *any* thread found an out-of-order pair (set `sorted=false`), the overall `sorted` becomes false. This guarantees the while-loop continues until *no* swaps occur in a full iteration.

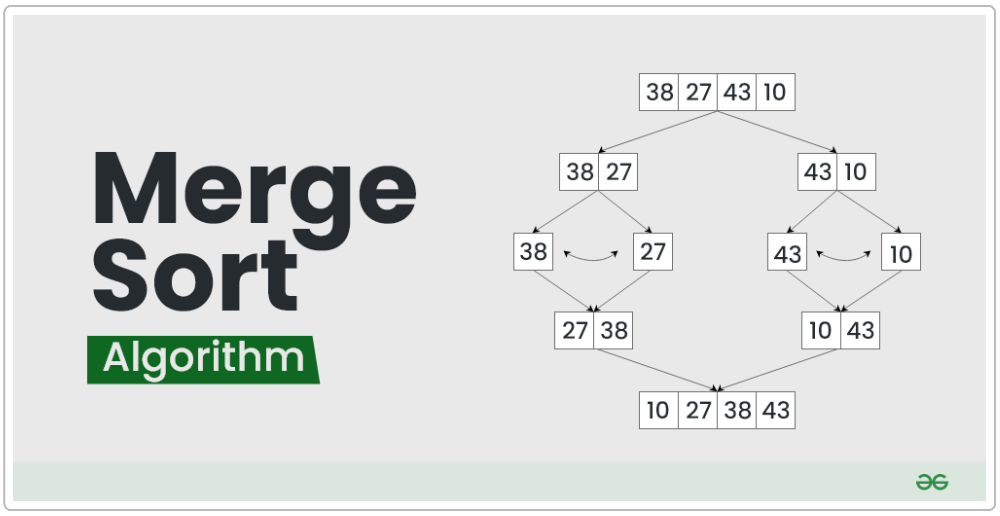
In summary, the parallel bubble sort code does two sub-steps per pass (even and odd) with parallel loops. Each thread independently compares/swaps distinct pairs, and then they synchronize (implicit barrier at end of each `#pragma omp for`) before the next phase. The use of `reduction` ensures threads cooperate to decide when sorting is done. This lets multiple cores work on different parts of the array at once, though bubble sort still has inherent data dependencies and is less scalable than divide-and-conquer sorts.

Merge Sort (Sequential and Parallel)

Merge Sort is a divide-and-conquer sorting algorithm with average time O(n log n). It works by recursively splitting the array into two halves, sorting each half, and then **merging** the two sorted halves into a single sorted array [5]. In concept, merge sort does three steps:

1. **Divide:** Split the array into two halves (left and right).
2. **Conquer:** Recursively sort each half.
3. **Merge:** Merge the two sorted halves into one sorted result.

This process continues until we reach subarrays of size 1 (which are trivially sorted), then merges progressively combine them. GfG explains: *"It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together"* [5].



Example of merge sort: the array [38, 27, 43, 10] is split into [38,27] and [43,10], each is further split into single elements, then merged back step by step into [10, 27, 38, 43].

- **Sequential Merge Sort Code:** A typical recursive implementation in C++ might look like:

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid+1, right);
        merge(arr, left, mid, right);
    }
}
```

Here `merge` is a helper that merges two sorted subarrays `arr[left..mid]` and `arr[mid+1..right]` into a sorted range. The `merge` function usually copies the two halves into temporary arrays, then iterates through them, picking the smaller head element each time, finally copying any leftovers. The code from our sources shows exactly that: create two temp arrays `L` and `R`, fill them from `arr`, then merge by comparing their elements [6] [7]. Step-by-step, each recursive call sorts half the array, and then one merge call combines them, achieving overall O(n log n) complexity [5].

- **Parallel Merge Sort (OpenMP sections):** We can also parallelize merge sort because the two recursive sorts of the halves are independent tasks. A common OpenMP approach is to use `#pragma omp parallel sections`, which allows each section to run in a separate thread. For example:

```
void mergeSortParallel(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);
    }
}
```

  Here's what happens:

- The `#pragma omp parallel sections` directive creates a team of (up to) two threads.
- Each `#pragma omp section` specifies a block to run in parallel. One section calls `mergeSortParallel` on the left half, the other on the right half. These two recursive calls can proceed at the same time on different threads [8].
- After the sections block ends, there is an implicit barrier: the code waits until both halves are sorted.
- Then the code calls `merge(arr, left, mid, right)` to combine the results (still in serial).

This fits the divide-and-conquer model: two independent tasks for the halves are run in parallel (as noted in a blog tutorial [9]). For large arrays, each half sort itself may spawn further parallel sections, though often an `if (right-left < threshold)` may be used to stop parallel recursion on small subarrays. In our code, merging is done after both sections complete, ensuring correctness (since merging requires both halves sorted). The OpenMP *sections* construct simplifies spawning exactly two tasks for the halves. As one source notes: "we use 'section' to define the two individual sections we want running in parallel" [9].

Overall, parallel merge sort will create multiple threads (one per section) at each recursive level. In practice, the program typically uses `#pragma omp parallel` in `main()` (or limits parallel recursion depth) to avoid creating too many threads. But the key idea is: *two recursive calls run concurrently*, then merge results. This leads to improved performance on multi-core CPUs, since the independent sorting work is shared.

Random Array Initialization

Before sorting, the program fills an array with random integers. Typically this is done by seeding C's random number generator and then assigning each element. For example, one might call `srand(time(NULL))` at the start of `main()` to seed the generator with the current time [10]. Then a loop like `for(i=0; i<n; i++) arr[i] = rand() % 100;` can fill the array with random values (here `% 100` bounds the range, but that is optional). The important part is *seeding* and *generating*. As the lab notes say: *"Initialize the random number generator by calling* `srand(time(NULL))`*. Then call* `rand()` *to get a new random number"* [10]. In our code, one array (say `original[]`) is filled this way. We may use a seed passed from command line or `time(NULL)` for unpredictability.

Copying Arrays (Helper Function)

To fairly compare algorithms, each sort must operate on the same initial data. Thus, a helper function is often used to copy the original random array into separate arrays for each sort. For instance, a function `copyArray(dest, src, n)` might loop `for(int i=0; i<n; i++) dest[i] = src[i];`. This ensures that the parallel sort and sequential sort each see the same unsorted data, rather than one sort modifying the array for the next. In the code, before calling each sorting function, the program copies the original into a working array. This is a simple element-by-element copy; no special OpenMP use is needed here.

Timing with `omp_get_wtime()`

The code measures performance using the OpenMP wall-clock timer `omp_get_wtime()`. This function returns the elapsed time in seconds since some fixed point (it's a "per-thread wall clock time" [11]). In practice, you call it before and after a sort, and subtract to get the duration. For example:

```c
double start = omp_get_wtime();
// ... call sorting function ...
double end = omp_get_wtime();
printf("Time = %f seconds\n", end - start);
```

As the OpenMP spec explains, this yields the elapsed wall-clock time [11]. Our code does exactly that: we record `start = omp_get_wtime()`, run the sort, then `end = omp_get_wtime()`, and compute `(end - start)` as the runtime. This is shown in examples from the OpenMP documentation [12]. The measured time (often in milliseconds or seconds) is then reported. Using `omp_get_wtime()` allows accurate timing even when code is running on multiple threads.

`main()` Function and Performance Reporting

In `main()`, the program typically performs these steps (in order):

1. **Read input:** Often the array size (and maybe a random seed) are given via command-line arguments. For example, `int N = atoi(argv[1]); int seed = atoi(argv[2]);`.

2. **Initialize arrays:** Allocate an array `original[N]`, seed the RNG (e.g. `srand(seed)` or `srand(time(NULL))`), and fill `original` with random numbers as described above.
3. **Prepare for each sort:** Create copies of the array for each variant. For example, `int arr[N]; copyArray(arr, original, N);`.
4. **Sequential Bubble Sort:** Copy original into `arr`, record start time with `omp_get_wtime()`, call `bubbleSortSequential(arr, N);`, record end time, and compute elapsed. Print this time, e.g.

```
printf("Sequential Bubble Sort Time: %f seconds\n", end-start);
```

5. **Parallel Bubble Sort:** Copy original into `arr` again, time `bubbleSortParallel(arr, N)`, and print that time.
6. **Sequential Merge Sort:** Copy original, then time `mergeSortSequential(arr, 0, N-1)`, print time.
7. **Parallel Merge Sort:** Copy original, then time the parallel merge sort. (Often one does `#pragma omp parallel` and `#pragma omp single` around the initial call to set up threading context, then call `mergeSortParallel(arr,0,N-1)`. Alternatively, one might call it directly if it internally creates parallel sections.) Then record and print its time.

After each sort, the program might also verify that `arr` is correctly sorted (for debugging). Finally, it reports all timings. A sample output might look like:

```
Sequential Bubble Sort Time: 1.234 sec
Parallel Bubble Sort Time:   0.789 sec
Sequential Merge Sort Time: 0.010 sec
Parallel Merge Sort Time:   0.007 sec
```

or similar. This way, the user can compare how much speedup (if any) is gained by parallelization. Typically, merge sort shows a clear improvement with multiple threads, while bubble sort's parallel version may only help for large N (due to its O(n^2) nature and synchronization overhead) [13] [14].

Throughout `main()`, the code uses the helper functions (copy, sorts) and `omp_get_wtime()` as described. By resetting the array before each test, it ensures fairness. All reported times come from differences of `omp_get_wtime()`, which match examples given in the OpenMP spec [12]. This completes the end-to-end explanation: concept, code behavior (sequential vs parallel), random data generation, timing, and result reporting.

References: Bubble sort and merge sort concepts and code patterns are described in algorithm tutorials [1] [5]. The use of OpenMP pragmas for odd-even bubble sort is illustrated in parallel sorting guides [3] [4] and slides [2]. The `#pragma omp parallel sections` approach for merge sort is shown in examples [8]. Seeding `srand()` for random numbers and using `omp_get_wtime()` for timing are standard as per documentation [10] [11] [12]. These together explain every part of the C++ code in an accessible way.

---

1 **Bubble Sort Algorithm | GeeksforGeeks**
https://www.geeksforgeeks.org/bubble-sort-algorithm/

2 **Parallel sorting Algorithms | PPT**
https://www.slideshare.net/slideshow/parallel-sorting-algorithms/63566725

3 4 13 **Parallelizing Sorting Algorithms using OpenMP - DEV Community**
https://dev.to/sahrohit/parallelizing-sorting-algorithms-using-openmp-1hec

5 **Merge Sort – Data Structure and Algorithms Tutorials | GeeksforGeeks**
https://www.geeksforgeeks.org/merge-sort/

6 7 **HPC CODES | PDF | Software Engineering | Algorithms And Data Structures**
https://es.scribd.com/document/853036767/HPC-CODES

8 9 **Parallel Merge Sort – Parallel Computing**
https://parcomp.wordpress.com/2017/02/26/parallel-merge-sort/

10 **Microsoft Word - Parallel Sort.docx**
https://www.csc.tntech.edu/pdcincs/resources/modules/plugged/parallel_sort/Parallel%20Sort.pdf

11 12 **omp_get_wtime**
https://www.openmp.org/spec-html/5.0/openmpsu160.html

14 **Parallelizing Sorting Algorithms using OpenMP | by Rohit Sah | Medium**
https://medium.com/@sahrohit9586/parallelizing-sorting-algorithms-using-openmp-136aa5a1e3ba