**Instructions for Compiling and Running the Code  
  
  
  
Compilation**

The provided C++ code can be compiled using a standard C++ compiler like g++.

Open a terminal and navigate to the directory containing the code file (sorting.cpp).

Use the following command to compile the code:

“g++ -std=c++20 -pthread sorting.cpp -o sorting”

This command will compile the code and generate an executable named sorting.

**Execution**

To run the compiled executable, use the following command:

“./sorting”

Ensure you have the required input file named input.csv in the same directory as the executable.  
  
  
  
**1. Architecture and Algorithms Used  
   
a) Parallel Merge Sort Algorithm :**

The core algorithm implemented in this code is Parallel Merge Sort, a variant of the traditional merge sort. It follows these steps:Divide and Conquer: The input array is recursively divided into smaller subarrays until they reach a base case (typically single elements).

**Parallel Sorting:** The sorting of subarrays is done concurrently using multiple threads, improving the sorting efficiency on multi-core systems.

**Merging:** The sorted subarrays are merged to produce the final sorted array. This merge step is also performed using multiple threads**.**

**b) Thread-based Parallelism**

The code utilizes std::thread library to create and manage multiple threads. Key points include:Determining Thread Count: The number of threads is determined based on the minimum of the maximum allowed CPU cores (N) and the available CPU cores on the system (std::thread::hardware\_concurrency()). This ensures optimal utilization of system resources.

**Parallel Sorting:** The array is divided into subarrays, and sorting is parallelized by spawning threads to sort each subarray concurrently.

Parallel Merging: During the merging step, multiple threads merge the sorted subarrays in parallel, enhancing the merging performance.

**c) Input and Output Handling**

**Input Reading:** The code reads input numbers and the maximum allowed CPU cores (N) from the configuration file (input\_test1.csv).

Output Writing: After performing parallel merge sort, the sorted numbers are written to an output file (output.csv).

**d) Merge Sort Optimization**

**In-place Merge Operation:** An optimization is employed where the merge operation is performed in-place, using the original input array and two additional vectors (leftArr and rightArr). This reduces memory overhead by minimizing the creation of temporary arrays during merging.

**e) Algorithm Execution**

**Main Function:** The main() function orchestrates the entire process, reading input, determining the number of threads, performing parallel merge sort, and writing the sorted numbers to the output file.

**Recursive Parallelization:** The parallelMergeSort function recursively parallelizes the merge sort algorithm, spawning threads for sorting and merging subarrays until the base case is reached.

Overall, this architecture combines parallelism with the merge sort algorithm to efficiently sort large arrays in a multi-core environment, optimizing both time and memory complexity. The use of threads and the in-place merge operation contribute to the algorithm's efficiency and effectiveness.  
  
**F) Threshold Adjustment Criteria for Optimized Sorting :**Small Dataset:If there are less than 500 numbers, the threshold is set higher (4 times the base value) to avoid parallel overhead.

Large Dataset with Few CPU cores:For big data with 10,000 or more numbers and only 1 or 2 CPU cores, the threshold is increased a bit (1.5 times the base value) to reduce parallel tasks.

Large Dataset with Many CPU cores:With more than 10,000 numbers and more than 2 CPU cores, the threshold stays at the base value to use the standard parallel approach.

Almost Sorted Small Dataset:For up to 1,000 nearly sorted numbers, the threshold is set higher (4 times the base value) to sort more sequentially.

Default Case:If none of the above apply, the threshold defaults to 4 times the base value as a catch-all**.**

2.   
  
C. Zelenka’s work mainly revolves around standard parallel merge sort [1]. In contrast, the given implementation incorporates dynamic data nature recognition, setting thresholds not just based on dataset size but on characteristics like 'almost sorted' status. This results in a more adaptive and possibly more efficient sorting process.

References :   
  
1. C. Zelenka, "Parallel Merge Sort," San Jose State University, Computer Science Department, San Jose, CA 95192, Tech. Rep., 2023. <https://www.sjsu.edu/people/robert.chun/courses/cs159/s3/T.pdf>

3. **Test Cases ->**   
  
**a)**  
# 4 (cpu cores)

295  
75  
…  
-61  
282 (100 numbers)  
  
  
  
**b)**   
  
# 2 (cpu cores)

499

699

…  
372

347 (10000 numbers)

**c)**   
  
# 4 (cpu cores)

374

825

…

105

652

**d)**# 2 (cpu cores)  
  
0

2

2

…  
998

998

999

999

**4.  
  
a) output ->**Small Dataset: So merge sort is used resulting, hence less memory is utilizedNumber of CPU cores: 4

Threshold value 100

Is Almost Sorted: 0 (False)

Elapsed time: 8.0375e-05 seconds

Memory used during sorting: 0 KB

**b) output ->**Large Dataset with Few CPU cores: So parallel merge sort is used. Hence more memory is utilizedNumber of CPU cores: 2

Threshold value 7500

total numbers 10000

Is Almost Sorted: 0 (False)

Elapsed time: 0.0090365 seconds

Memory used during sorting: 393216 KB

**c) output ->**Large Dataset with Many CPU cores: So parallel merge sort is used. Hence more memory is utilizedNumber of CPU cores: 4

Threshold value 10000

total numbers 10000

Is Almost Sorted: 0 (False)

Elapsed time: 0.0145321 seconds

Memory used during sorting: 81920 KB

d) output ->  
  
Almost Sorted Small Dataset : So merge sort is used. Hence less memory is utilized  
  
Number of CPU cores: 2

Threshold value 2000

Is Almost Sorted: 1 (True)

Elapsed time: 0.00168554 seconds

Memory used during sorting: 0 KB

Used std::thread::hardware\_concurrency() function from the C++ Standard Library to measure the number of CPU cores utilized

**int numThreads = std::min(max\_cores, static\_cast<int>(std::thread::hardware\_concurrency()));**

**5.   
  
  
2st test case with 2 cpu cores :**# 2 (cpu cores)

499

699

…  
372

347 (10000 numbers)  
  
Number of CPU cores: 2

Threshold value 7500

total numbers 10000

Is Almost Sorted: 0 (False)

Elapsed time: 0.00839196 seconds

Memory used during sorting: 294912 KB  
  
2nd test case with 4 cpu cores :   
  
Number of CPU cores: 4

Threshold value 10000

total numbers 10000

Is Almost Sorted: 0

Elapsed time: 0.0167277 seconds

Memory used during sorting: 114688 KB

2nd test case with 6 cpu cores :   
  
  
Number of CPU cores: 6

Threshold value 6664

total numbers 10000

Is Almost Sorted: 0

Elapsed time: 0.00766625 seconds

Memory used during sorting: 180224 KB  
  
2nd test case with 8 cpu cores :   
  
Number of CPU cores: 8

Threshold value 5000

total numbers 10000

Is Almost Sorted: 0

Elapsed time: 0.00927642 seconds

Memory used during sorting: 376832 KB

* With more cores, like 6, the program sorted faster.
* Oddly, the 4-core test took longer than 2-cores.
* The 8-core test used the most memory.
* So, more cores can mean faster sorting, but sometimes it uses more memory.