Introduction :

Algorthim choice:merge sort

Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the divide-and-conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.

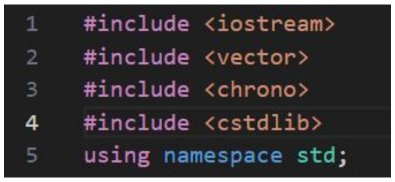
How does it work

1-Divide: Divide the list or array recursively into two halves (left and right)until it can no more be divided. (parallelizable)

2-Conquer: Each subarray is sorted individually (no dependencies) using the merge sort algorithm.

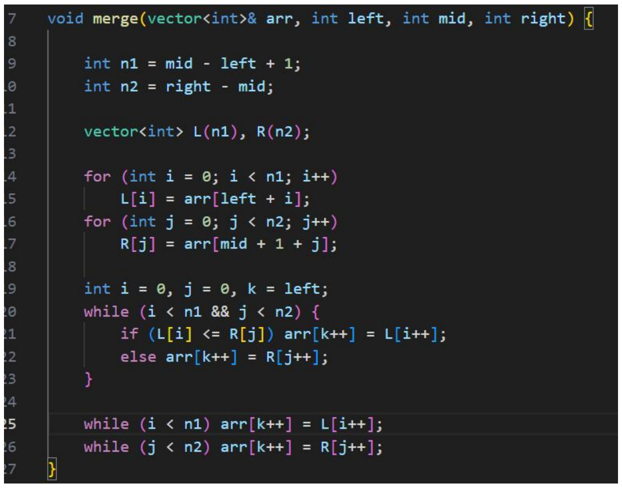
3-Merge: The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged. (parallelizable but complex) so we are not doing it

Sequential Implementation



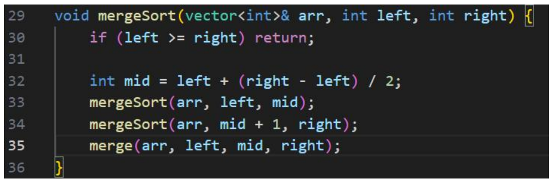
```cpp
#include <iostream>
#include <vector>
#include <chrono>
#include <cstdlib>
using namespace std;
```

iostream is for input/output.
vector is used for dynamic array storage.
chrono is used for high-resolution timing.
cstdlib is used for random number generation (rand()).



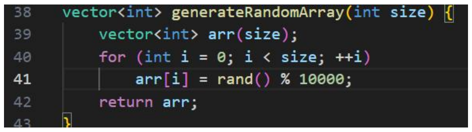
```cpp
void merge(vector<int>& arr, int left, int mid, int right) {

    int n1 = mid - left + 1;
    int n2 = right - mid;

    vector<int> L(n1), R(n2);

    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0, k = left;
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) arr[k++] = L[i++];
        else arr[k++] = R[j++];
    }

    while (i < n1) arr[k++] = L[i++];
    while (j < n2) arr[k++] = R[j++];
}
```

merge() Function:
Splits the array into two temporary arrays L and R.
Merges them in sorted order back into arr by comparing elements sequentially.
Copies any remaining elements back into the original array.
one side finishes early.



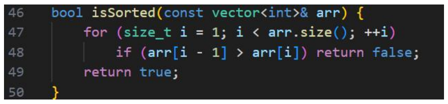
```cpp
void mergeSort(vector<int>& arr, int left, int right) {
    if (left >= right) return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

Recursively splits the array into halves.
Sorts each half using mergeSort.
Merges the sorted halves using merge().



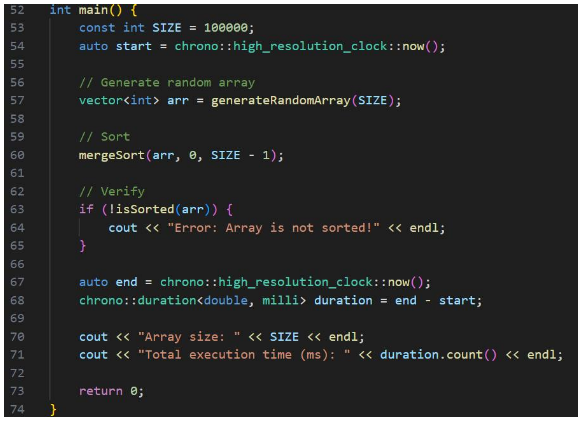
```cpp
vector<int> generateRandomArray(int size) {
    vector<int> arr(size);
    for (int i = 0; i < size; ++i)
        arr[i] = rand() % 10000;
    return arr;
}
```

Generates an array of a given size filled with random integers.



```cpp
bool isSorted(const vector<int>& arr) {
    for (size_t i = 1; i < arr.size(); ++i)
        if (arr[i - 1] > arr[i]) return false;
    return true;
}
```

Validates the array after sorting.
Returns true if the array is sorted in non-decreasing order.

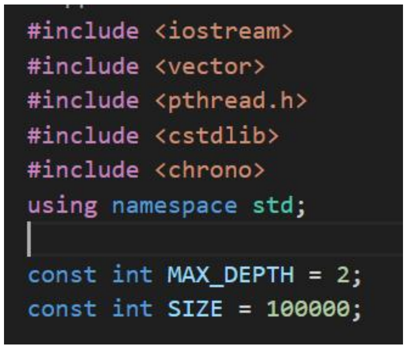


```cpp
int main() {
    const int SIZE = 100000;
    auto start = chrono::high_resolution_clock::now();

    // Generate random array
    vector<int> arr = generateRandomArray(SIZE);

    // Sort
    mergeSort(arr, 0, SIZE - 1);

    // Verify
    if (!isSorted(arr)) {
        cout << "Error: Array is not sorted!" << endl;
    }

    auto end = chrono::high_resolution_clock::now();
    chrono::duration<double, milli> duration = end - start;

    cout << "Array size: " << SIZE << endl;
    cout << "Total execution time (ms): " << duration.count() << endl;

    return 0;
}
```

Defines a test size and starts the timer.
Generates a random array, sorts it, and checks validity.
Stops the timer and prints the total time in milliseconds.

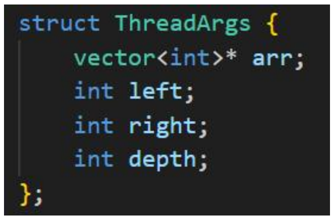
Parallelization Strategy

The parallel version uses Pthreads to execute the left and right halves of the mergeSort() function in parallel. A custom struct is used to pass arguments to the thread function. To control thread overhead, a maximum recursion depth is defined. Beyond that depth, the algorithm falls back to sequential sorting. Merge operations are only performed after both threads have completed, ensuring thread safety without the need for locks.



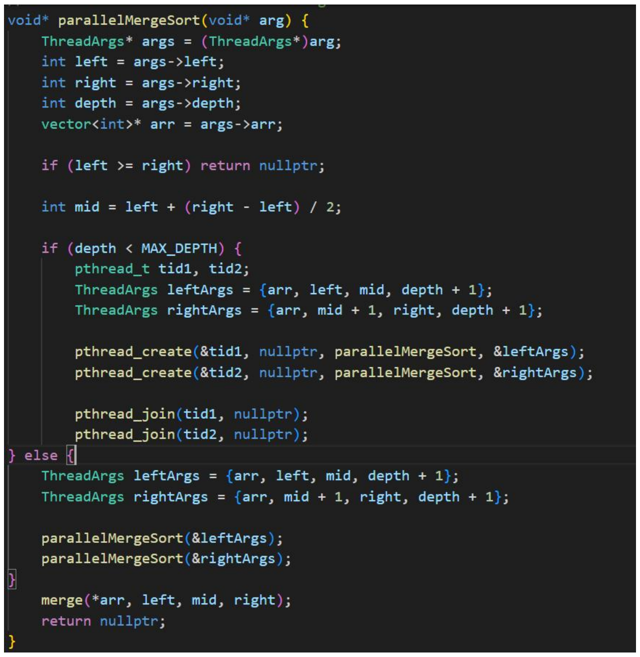
```cpp
#include <iostream>
#include <vector>
#include <pthread.h>
#include <cstdlib>
#include <chrono>
using namespace std;

const int MAX_DEPTH = 2;
const int SIZE = 100000;
```

We include necessary libraries: pthread.h for threading.
MAX_DEPTH limits how many recursive levels are allowed to spawn new threads. Considering my CPU 4 threads working at the same time is the maximum.
SIZE is the number of elements in the array.



```cpp
struct ThreadArgs {
    vector<int>* arr;
    int left;
    int right;
    int depth;
};
```

Because pthread_create() only accepts a void* argument, we put all needed values into a struct.
This struct is passed to each thread and contains:
The array to sort
The current subarray bounds
The current recursion depth (for limiting thread creation)



```
void* parallelMergeSort(void* arg) {
    ThreadArgs* args = (ThreadArgs*)arg;
    int left = args->left;
    int right = args->right;
    int depth = args->depth;
    vector<int>* arr = args->arr;

    if (left >= right) return nullptr;

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

    if (depth < MAX_DEPTH) {
        pthread_t tid1, tid2;
        ThreadArgs leftArgs = {arr, left, mid, depth + 1};
        ThreadArgs rightArgs = {arr, mid + 1, right, depth + 1};

        pthread_create(&tid1, nullptr, parallelMergeSort, &leftArgs);
        pthread_create(&tid2, nullptr, parallelMergeSort, &rightArgs);

        pthread_join(tid1, nullptr);
        pthread_join(tid2, nullptr);
} else {
    ThreadArgs leftArgs = {arr, left, mid, depth + 1};
    ThreadArgs rightArgs = {arr, mid + 1, right, depth + 1};

    parallelMergeSort(&leftArgs);
    parallelMergeSort(&rightArgs);
}
    merge(*arr, left, mid, right);
    return nullptr;
}
```

If recursion depth is below MAX_DEPTH, create two threads to sort the left and right halves in parallel.
If the depth limit is reached, fall back to sequential recursion.
After both halves are sorted (via thread or recursion), merge() is called to combine them.
This structure balances parallel efficiency with thread overhead.

Experiments

All experiments were done on a laptop with the following specifications:
Processor: Intel(R) Core(TM) i3-8130U CPU @ 2.20GHz
Cores: 2 Physical Cores, 4 Logical Threads (Hyper-Threading)
RAM: 8.00 GB
Operating System: Windows 10 (64-bit)
Development Environment: Visual Studio Code running on WSL (Ubuntu)
This hardware made it important to limit the number of threads created during the experiment to 4 to avoid performance loss from thread contention or context switching.

Input Sizes Tested:

- 1,000
- 10,000
- 100,000
- 1,000,000

Each one (input size + thread count) was executed five times. The average execution time was recorded.

RESULTS

Input Size	Sequential (1 Thread)	2 Threads	4 Threads	Speedup for 2 threads	Speedup for 4 threads
1000	0.55176	0.7178362	1.638756	0.77	0.34
10000	8.196748	4.55063	3.712154	1.81	2.31
100000	104.03082	55.52588	42.30308	1.87	2.48
1000000	1003.5902	605.2946	503.093	1.66	1.99

notes
Thread Overhead:
Small inputs (1k elements) ran slower with threads due to creation costs.
Best Scaling:
100k elements achieved 2.46× speedup (close to 2-core theoretical max).
Limitations:
Sublinear speedup from sequential merges and memory
bottlenecks.

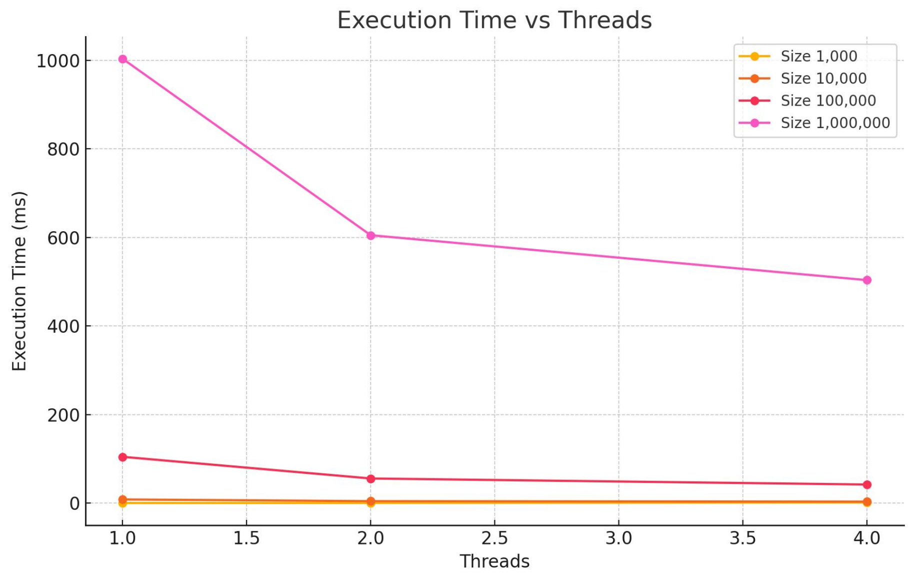


Figure 1: Execution Time vs Threads

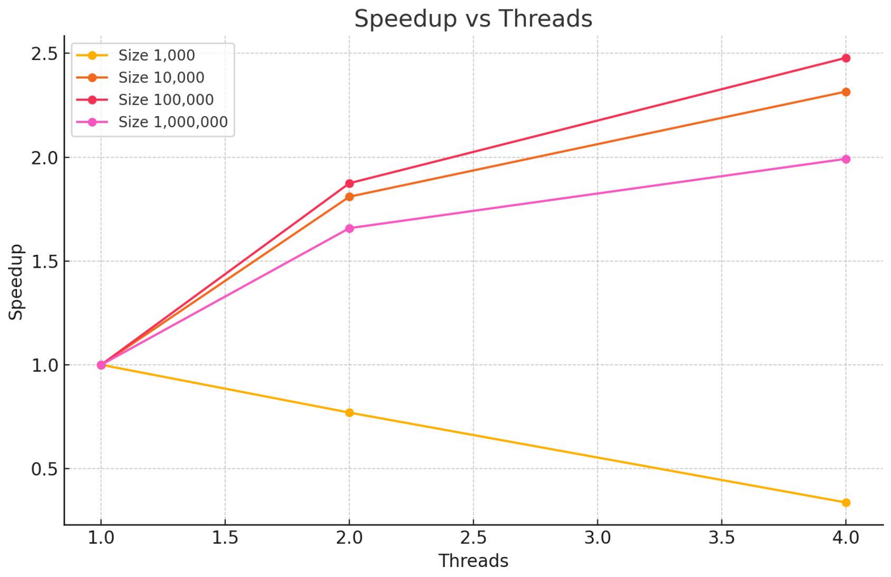


Figure 2: Speedup vs Threads

Discussion

For small input sizes (e.g., 1,000), the multi-threaded version performed worse(was slower) than the sequential one due to overhead from thread creation and management. This showes a fundamental concept in parallel computing — small tasks can suffer from parallel overhead. At larger sizes (100,000 and 1,000,000), the benefits of parallelism were clear. The 4-thread almost halved the execution time compared to the sequential run. This aligns with Amdahl's Law, where speedup is limited by the proportion of the code that must be executed serially. Merge operations and recursive control logic are not parallelized, which limits the theoretical maximum speedup.

Conclusion

We implemented both sequential and parallel versions of Merge Sort using Pthreads. Performance confirmed that parallelism significantly improves execution time for large input sizes.also showed the importance of workload size, thread control, and system hardware when designing parallel algorithms.

Citation

"Some parts of the parallel code design, and documentation language were supported by OpenAI's ChatGPT. Specifically, ChatGPT was used to troubleshoot thread synchronization issues, make the graphs (I didn't know how to) , and help with the snippets. All final code and analysis were fully understood, tested, and interpreted by me."