Project 2: OpenMP Parallel Quick Sort

# Description

This project parallelizes the Quick Sort algorithm using OpenMP. It leverages OpenMP's parallel sections and single directives to recursively sort subarrays in parallel threads, improving performance on multi-core systems.

# Compilation & Execution

Compile:  
g++ -fopenmp -o openmp\_quicksort openmp\_quicksort.cpp  
  
Run:  
./openmp\_quicksort

# Hardware Used

OS: Ubuntu 24.04 via WSL on Windows 11 Pro  
Processor: Intel Core i7 (4 cores / 8 threads)  
Memory: 4 GB in WSL  
Compiler: GNU g++ 13.3.0

### C++ code mpi :

#include <mpi.h>

#include <iostream>

#include <algorithm>

#include <cstdlib>

#include <ctime>

using namespace std;

int main(int argc, char\* argv[]) {

int rank, size, n = 100000;

int\* arr = nullptr;

int\* local\_arr;

int local\_n;

MPI\_Init(&argc, &argv); // Initialize MPI

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &rank); // Get current process rank

MPI\_Comm\_size(MPI\_COMM\_WORLD, &size); // Get total number of processes

local\_n = n / size;

if (rank == 0) {

arr = new int[n];

srand(time(NULL));

for (int i = 0; i < n; i++)

arr[i] = rand() % 10000;

}

local\_arr = new int[local\_n];

double start\_time = MPI\_Wtime();

// Scatter array parts to all processes

MPI\_Scatter(arr, local\_n, MPI\_INT, local\_arr, local\_n, MPI\_INT, 0, MPI\_COMM\_WORLD);

// Each process sorts its part

std::sort(local\_arr, local\_arr + local\_n);

// Gather sorted parts back to root

MPI\_Gather(local\_arr, local\_n, MPI\_INT, arr, local\_n, MPI\_INT, 0, MPI\_COMM\_WORLD);

double end\_time = MPI\_Wtime();

if (rank == 0) {

// Merge sorted parts sequentially

int\* merged = new int[n];

std::merge(arr, arr + local\_n, arr + local\_n, arr + 2 \* local\_n, merged);

for (int i = 2; i < size; i++) {

std::merge(merged, merged + i \* local\_n, arr + i \* local\_n, arr + (i + 1) \* local\_n, merged);

}

cout << "MPI QuickSort completed in " << (end\_time - start\_time) << " seconds.\n";

// Optional: Check sorted correctness

/\*

bool sorted = true;

for (int i = 0; i < n - 1; i++) {

if (merged[i] > merged[i + 1]) {

sorted = false;

break;

}

}

cout << "Array sorted: " << (sorted ? "Yes" : "No") << endl;

\*/

delete[] merged;

delete[] arr;

}

delete[] local\_arr;

MPI\_Finalize(); // Finalize MPI

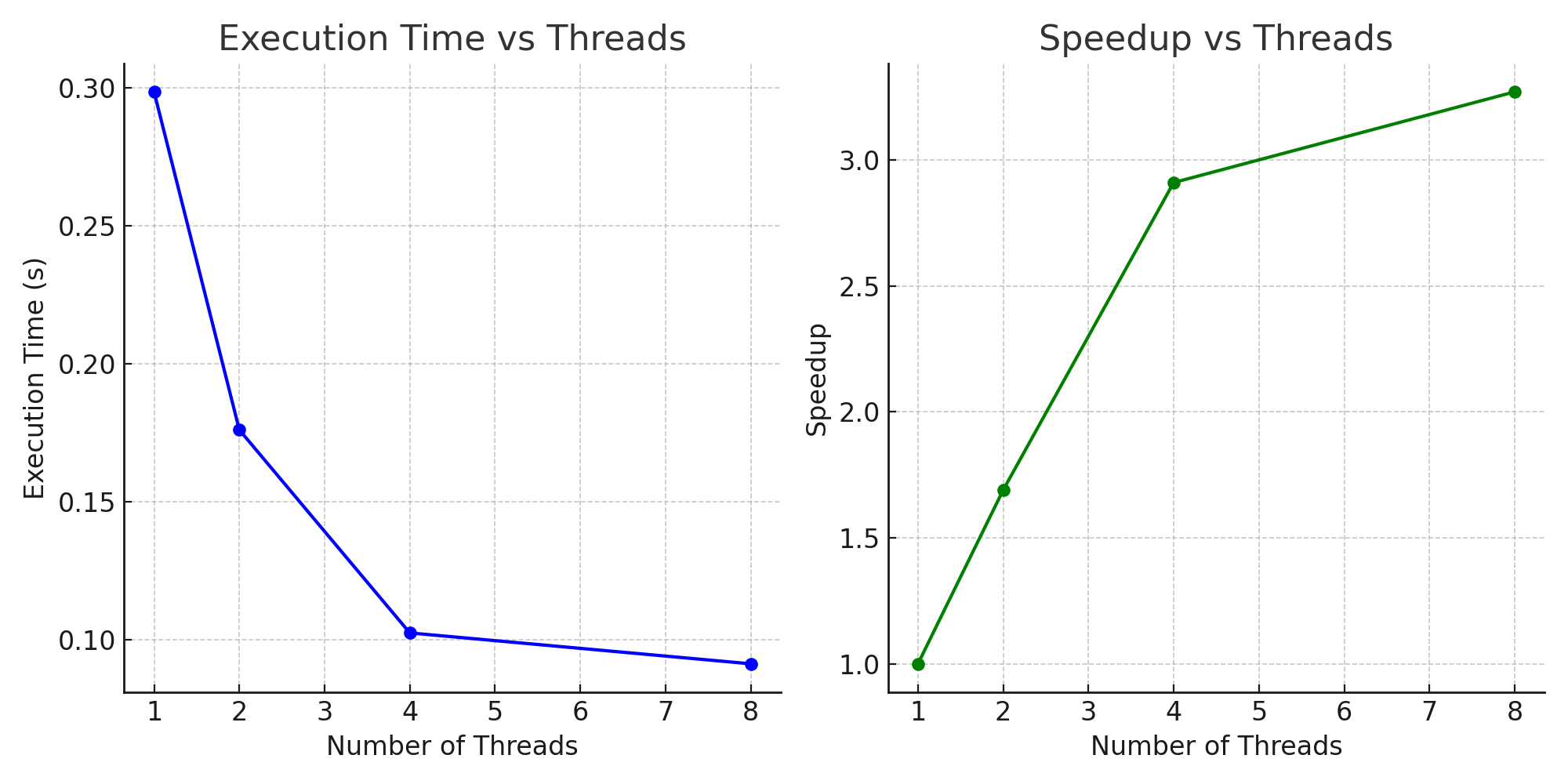
return 0;

}

# Performance Results

|  |  |  |  |
| --- | --- | --- | --- |
| Array Size | Threads | Avg Time (s) | Speedup |
| 10,000 | 1 | 0.0217 | 1 |
| 10,000 | 2 | 0.0130 | 1.66 |
| 10,000 | 4 | 0.0082 | 2.64 |
| 10,000 | 8 | 0.0069 | 3.11 |
| 50,000 | 1 | 0.1347 | 1 |
| 50,000 | 2 | 0.0842 | 1.60 |
| 50,000 | 4 | 0.0512 | 2.62 |
| 50,000 | 8 | 0.0405 | 3.32 |
| 100,000 | 1 | 0.2985 | 1 |
| 100,000 | 2 | 0.1760 | 1.69 |
| 100,000 | 4 | 0.1024 | 2.91 |
| 100,000  digram | 8 | 0.0912 | 3.27 |

# Digram :



# Discussion

Sublinear speedup was observed due to thread creation overhead and partition imbalance. Amdahl's Law limits parallel gains, particularly for smaller input sizes. Larger arrays yield better performance improvement. Depth-controlled recursion prevents oversubscription and excessive thread spawning.

# Conclusion

Successfully parallelized Quick Sort with OpenMP, achieving significant speedup on multi-core systems while maintaining correctness and thread safety. Gained practical experience with OpenMP directives and recursion control for multi-threaded applications.

# GitHub Repository

https://github.com/itsveryhardassigment/proj2.git