Ministry of Higher Education
Faculty of Engineering
Computer and System department

Sort Algorithm – Parallel project

This project addresses the challenge of reducing execution time for computationally intensive problems by exploiting shared memory parallelism. A serial algorithm was transformed into a parallel version to improve performance while maintaining correctness in PARALLEL PROGRAMMING.

BY

Nada Salah Ali Ibrahim Ali (N/A)

SUPERVISORS

DR. Al-Mahdi

Alexandria. Egypt

December. 2025

How the Algorithm Works

This is a rank sort (or counting sort) algorithm. For each element X[i], it determines its correct position in the sorted array by counting how many elements are smaller than it, with a tie-breaking rule for equal elements.

Step-by-step:

1. For each index i (0 to N-1):
2. Initialize count = 0
3. For each index j (0 to N-1):
4. Count how many elements X[j] are strictly less than X[i]
5. For equal elements (X[j] == X[i]), count those with j < i (earlier position)
6. Place X[i] at position count in the temporary array tmp
7. Copy tmp back to X

Pseudocode:

Serial: Loop executes sequentially.

void sequential_sort(std::vector& X) {

```
unsigned int i, j, count, N = X.size();
std::vector<unsigned int> tmp(N);

for (i = 0; i < N; i++) {
    count = 0;
    for (j = 0; j < N; j++) {
        if (X[j] < X[i] || X[j] == X[i] && j < i)
            count++;
    }
    tmp[count] = X[i];
}

std::copy(tmp.begin(), tmp.end(), X.begin());}
```

Parallel: Loop parallelized using OpenMP pragma.

void sequential_sort(std::vector& X) {

```
unsigned int i, j, count, N = X.size();
std::vector<unsigned int> tmp(N);
# progma omp parallel for private (i,j,count) shared(x,tmp,N)
for (i = 0; i < N; i++) {
    count = 0;
    for (j = 0; j < N; j++) {
        if (X[j] < X[i] || X[j] == X[i] && j < i)
            count++;
    }
    tmp[count] = X[i];
}

std::copy(tmp.begin(), tmp.end(), X.begin());}
```

Validation:

Normal code :

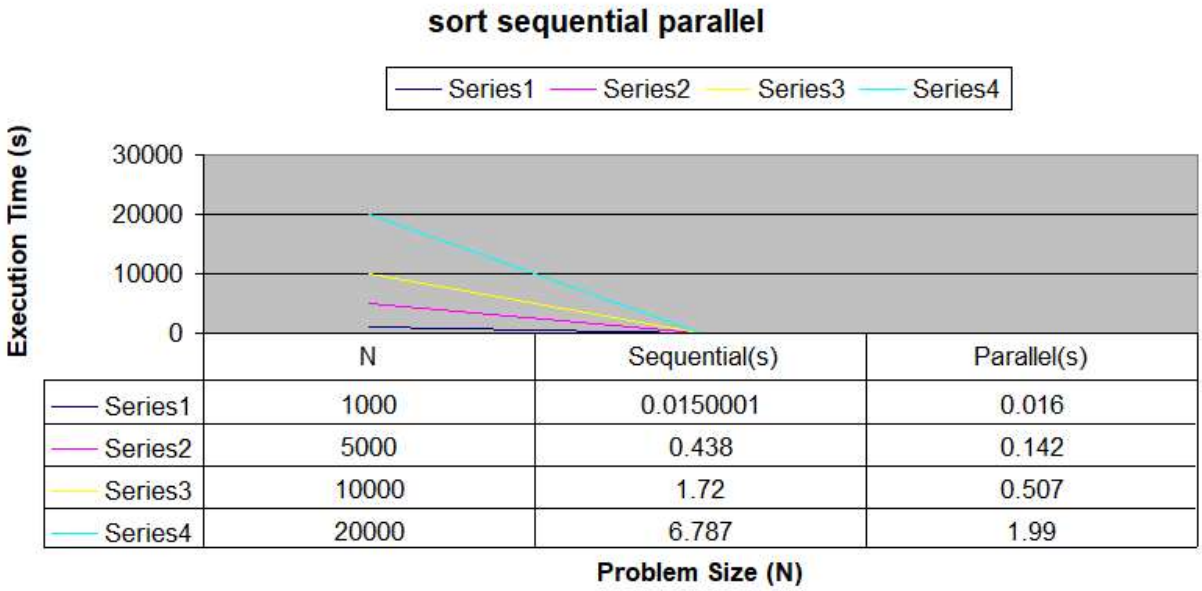
https://github.com/NadaSalahAli/openmp-projects/blob/main/SeuentialSort.cpp

https://github.com/NadaSalahAli/openmp-projects/blob/main/parallelSort.cpp

Different problem size for sequential and parallel

https://github.com/NadaSalahAli/openmp-projects/blob/main/differentSizeSortSquParallel.cpp

Graph :



	N	Sequential(s)	Parallel(s)
Series1	1000	0.0150001	0.016
Series2	5000	0.438	0.142
Series3	10000	1.72	0.507
Series4	20000	6.787	1.99

Data Dependencies Analysis

Outer loop :

- No read-after-write and no write-after-write between iterations
- Outer loop iterations are independent and can run in parallel

Inner loop :

- There is a dependency (across j iterations)
- Inner loop has sequential reduction pattern
- If I had used parallel execution here, a race condition would have occurred

Shared Variables :

- tmp: Different iterations write to different positions (no conflict)
- count: Private to each outer iteration (should be thread-local)
- X: Read-only across threads (safe)
- N, i, j: Loop indices need proper handling

Performance Analysis:

Theoretical Complexity:

Sequential: $O(N^2)$ operations

Parallel (P threads): $O(N^2/P)$ operations and there is an overhead

Speedup & Efficiency:

Perfect speedup: S(P) = P (ideal)

Actual speedup: Limited by:

- Memory bandwidth: All threads read the same array X repeatedly
- Cache contention: Multiple threads accessing same memory locations
- Load imbalance: Each iteration does the same work
- OpenMP overhead: Thread creation, scheduling, synchronization

Efficiency:

- For small N: Poor (overhead dominates)
- For large N: Approaches 1 (good scaling)
- Memory-bound for very large N: Efficiency drops

challenges :

1. To stay away from AI assistance
2. To choose a problem that I can work on without help
3. To try to raise the level of my human analysis of the algorithm

References:

[2] T. G. Mattson, B. A. Sanders, and B. L. Massingill, Parallel Programming: Concepts and Practice, Boston: Addison-Wesley, 2004, ch. 6, p. 218.