Lab 2 Report

Gabriel Zagury de Magalhães

System

Ubuntu linux arm64, running on an **Apple M3 Pro** processor, 18 GB of unified ram, **6 “performance” cores** that run on **4.06 GHz** and **6 “efficiency” cores** that run on **2.8 GHz.**

Bubble Sort

✅ Bubble Sort sequential
✅ Bubble Sort parallel

For the sequential version, I separated the algorithm into the bubble and sequential_bubble_sort functions. Bubble is a pass through a subarray, and the sequential version is a loop that performs bubble passes until the array is sorted. This separation of functions will be beneficial for the parallel version.

⏰ **Parallel schedule:** Custom static schedule with barriers

For the parallel version, we divide the array into multiple parts, and then sort the edges of those arrays.

First, I attempted a dynamic for loop because it seemed more logical, considering that a region might finish executing before another.

Next, I tried a static for comparison and discovered that it’s actually faster. This is likely because the speedup of the precise dynamic for loops doesn’t compensate for the additional cost of dynamically scheduling them instead of simply dividing.

Then, I realized that both versions had something that could be improved. They had two parallel sections, which resulted in the unnecessary creation of threads twice. To address this, I implemented a more manual version where the threads wait in a barrier instead of being created twice. This modification further improved the algorithm’s speed.

Merge Sort

✅ Merge sort sequential
✅ Merge sort parallel
✅ **Improvement:** only parallelize after a big enough array size

For the sequential one, it's the standard algorithm in C.

For the parallel version, I employed **parallel tasks** that recursively create threads until the array size is sufficiently small (default: 1024). Once the array is small enough, I call the sequential merge sort on it.

To wait for the task to complete, we use **taskwait.** Additionally**,** I added **firstprivate(mid) t**o ensure that each task receives a copy of mid instead of the global variable.

Parallelizing only after a big array size, like 1024, has some benefits. It saves time and resources by not creating and managing threads for small arrays. But when the array size is smaller, the time saved in sorting might not be worth it.

Odd Even Sort

Why is parallelization more adapted to the odd-even sort algorithm?

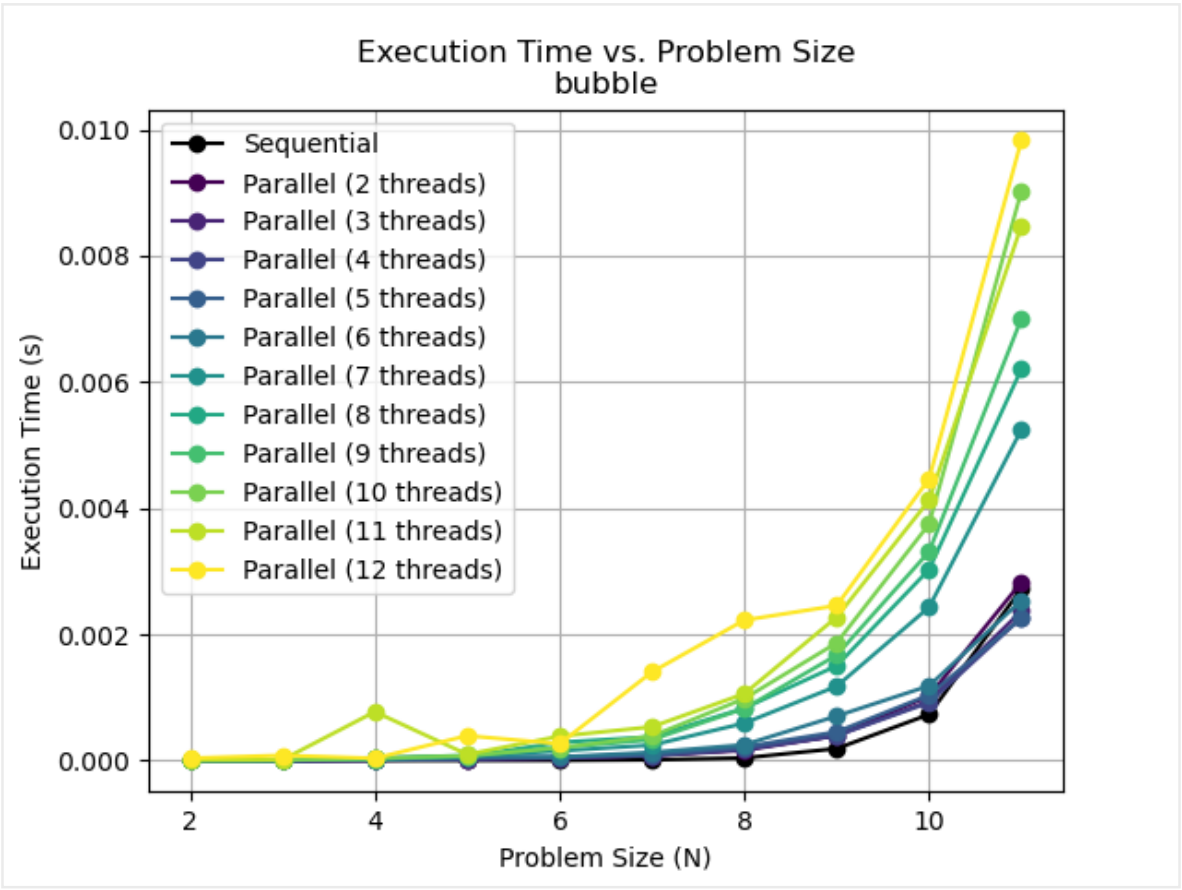
In the odd-even sort algorithm, elements are compared in two distinct phases: odd and even phases. Within each phase, comparisons between pairs of elements can be performed independently of one another.

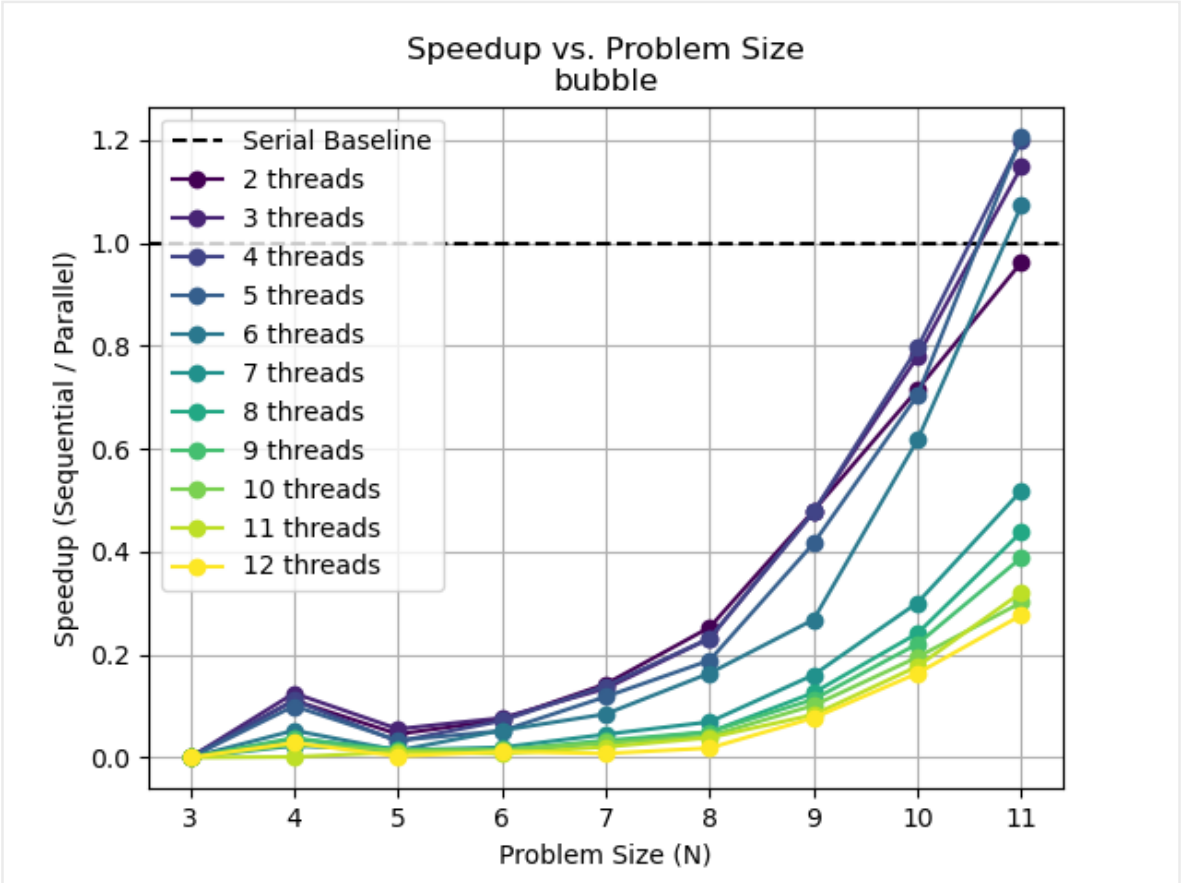
Since each comparison within a phase is isolated, multiple processors or threads can simultaneously handle different pairs, leading to a significant enhancement in the algorithm's efficiency.

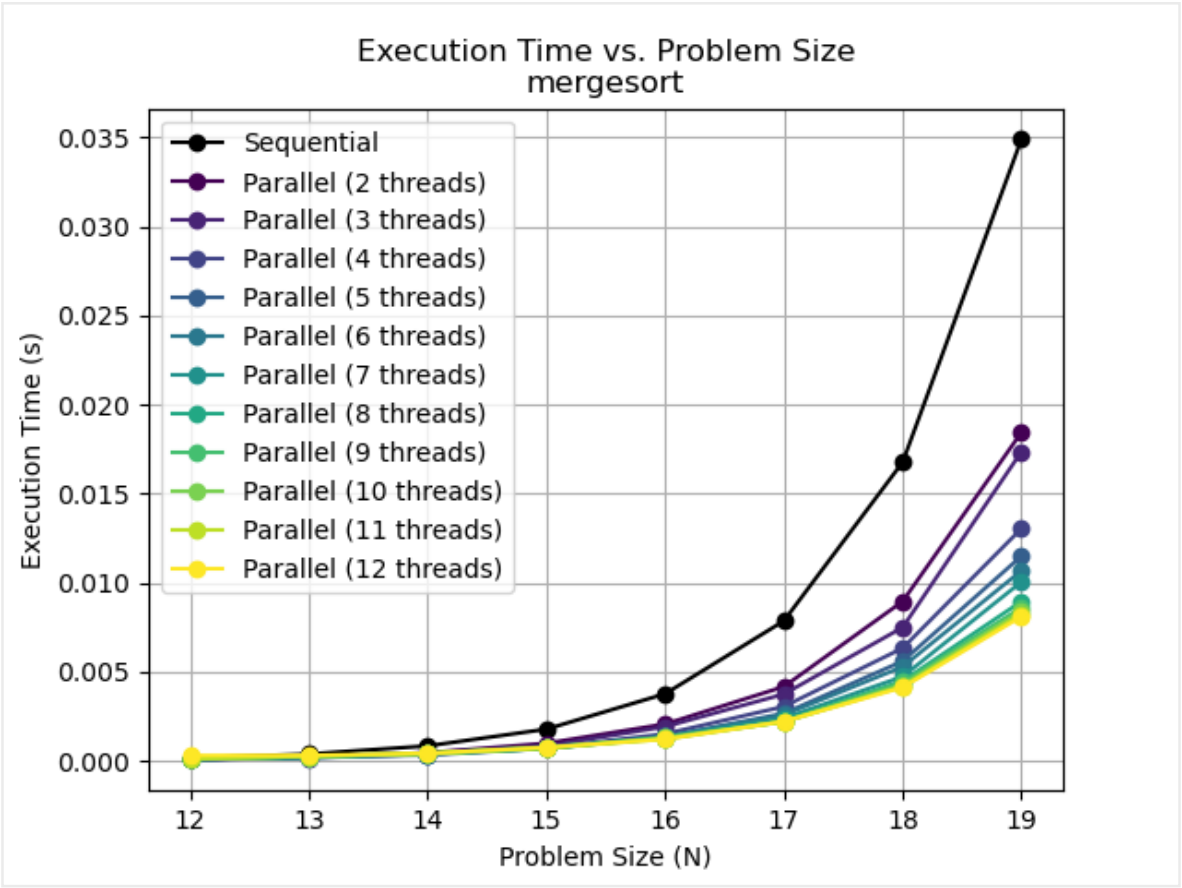
✅ Odd even sort sequential
✅ Odd even sort parallel

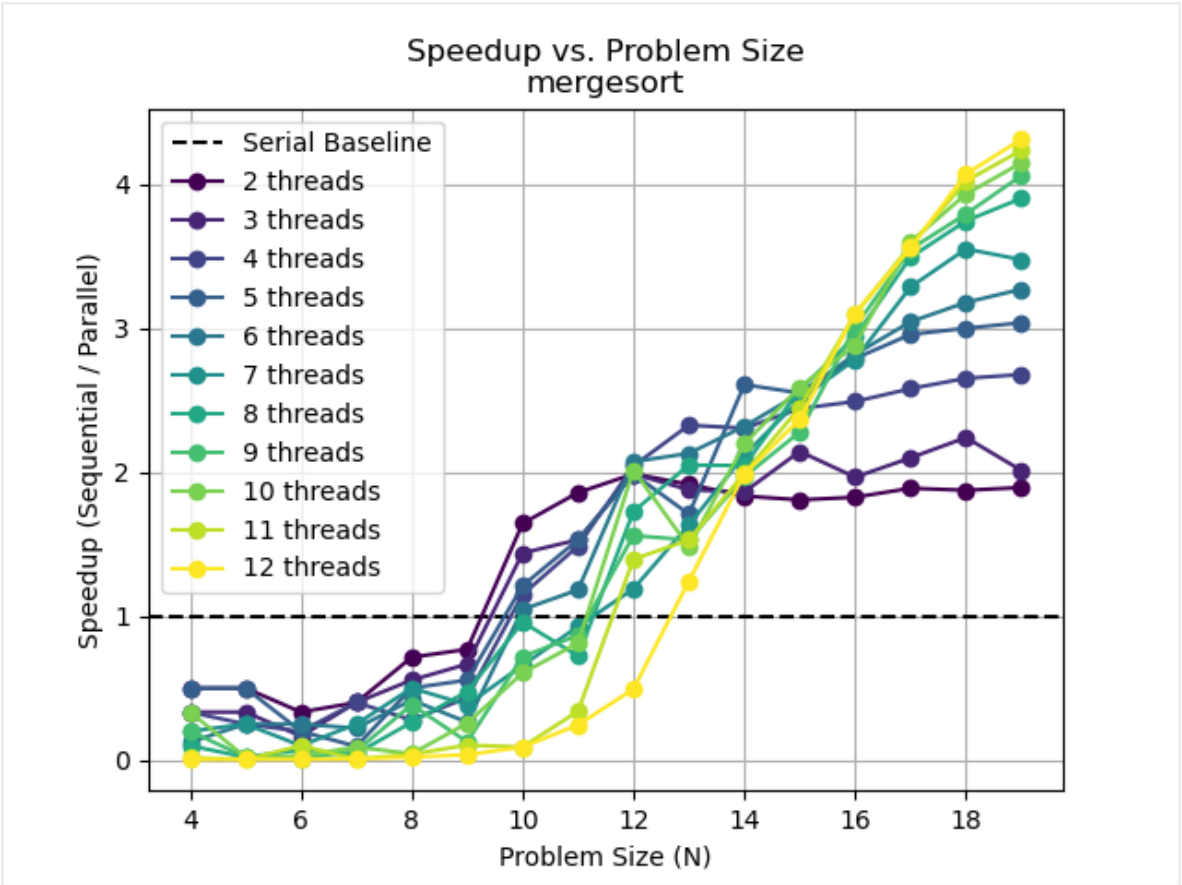
For this algorithm, both codes are completely independent. I chose a static schedule because it was faster, using pragma parallel for schedule(static)

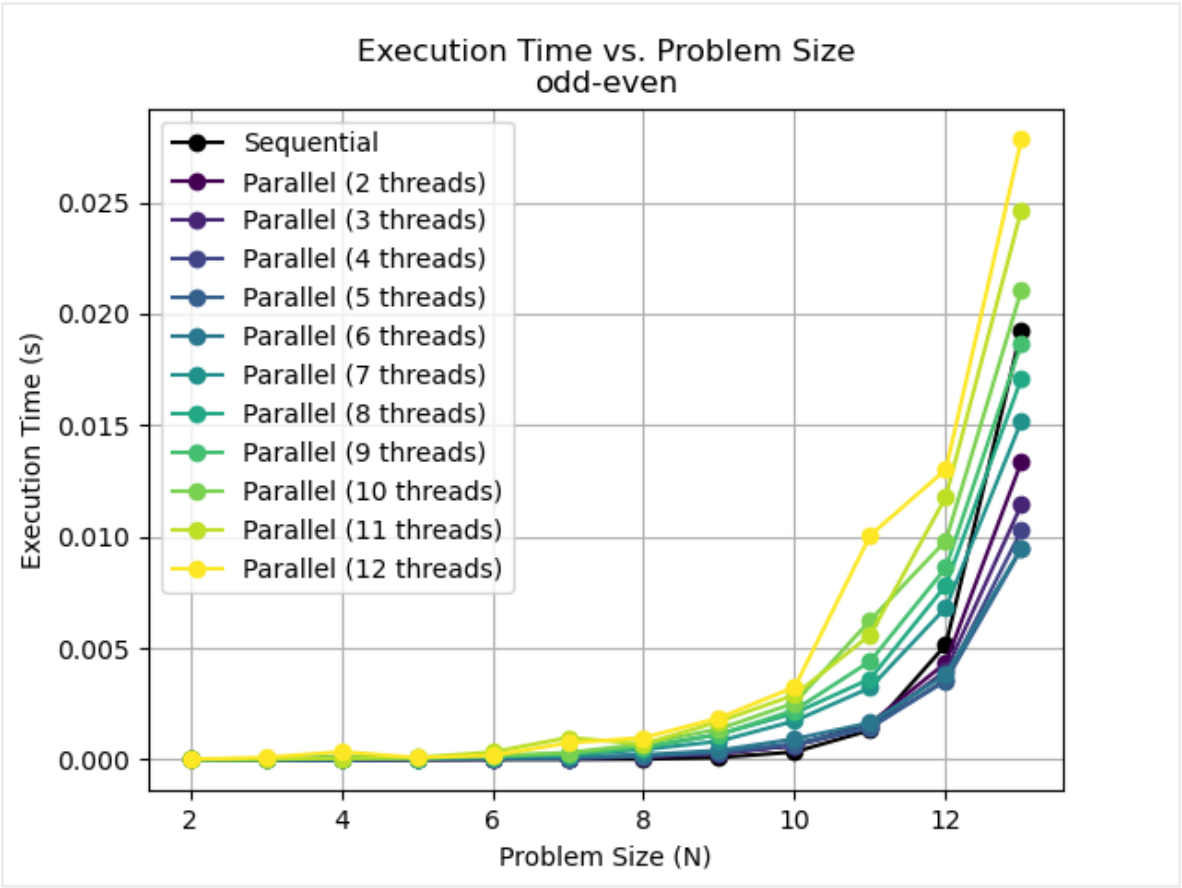
Plotting

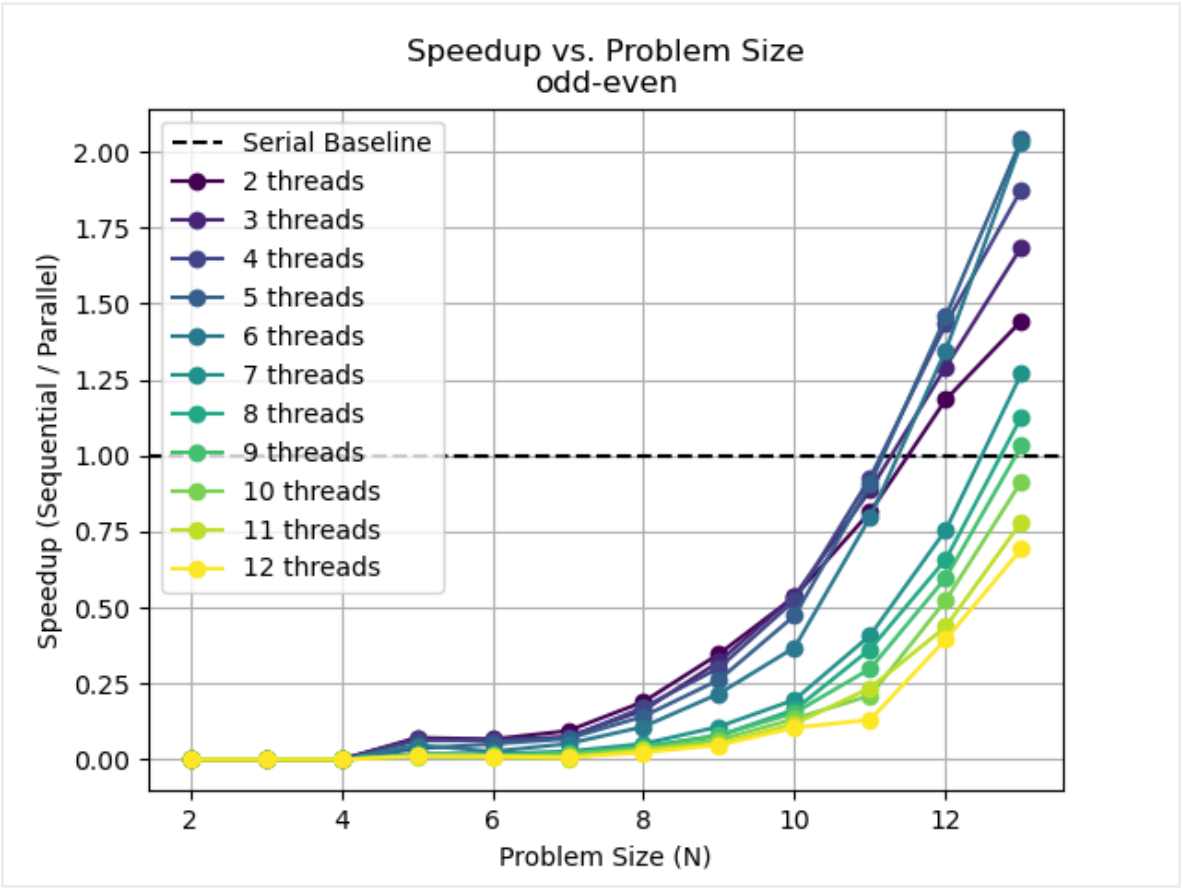

Execution Time vs. Problem Size
bubble
Execution Time (s)
Problem Size (N)
Sequential
Parallel (2 threads)
Parallel (3 threads)
Parallel (4 threads)
Parallel (5 threads)
Parallel (6 threads)
Parallel (7 threads)
Parallel (8 threads)
Parallel (9 threads)
Parallel (10 threads)
Parallel (11 threads)
Parallel (12 threads)
0.010
0.008
0.006
0.004
0.002
0.000
2
4
6
8
10

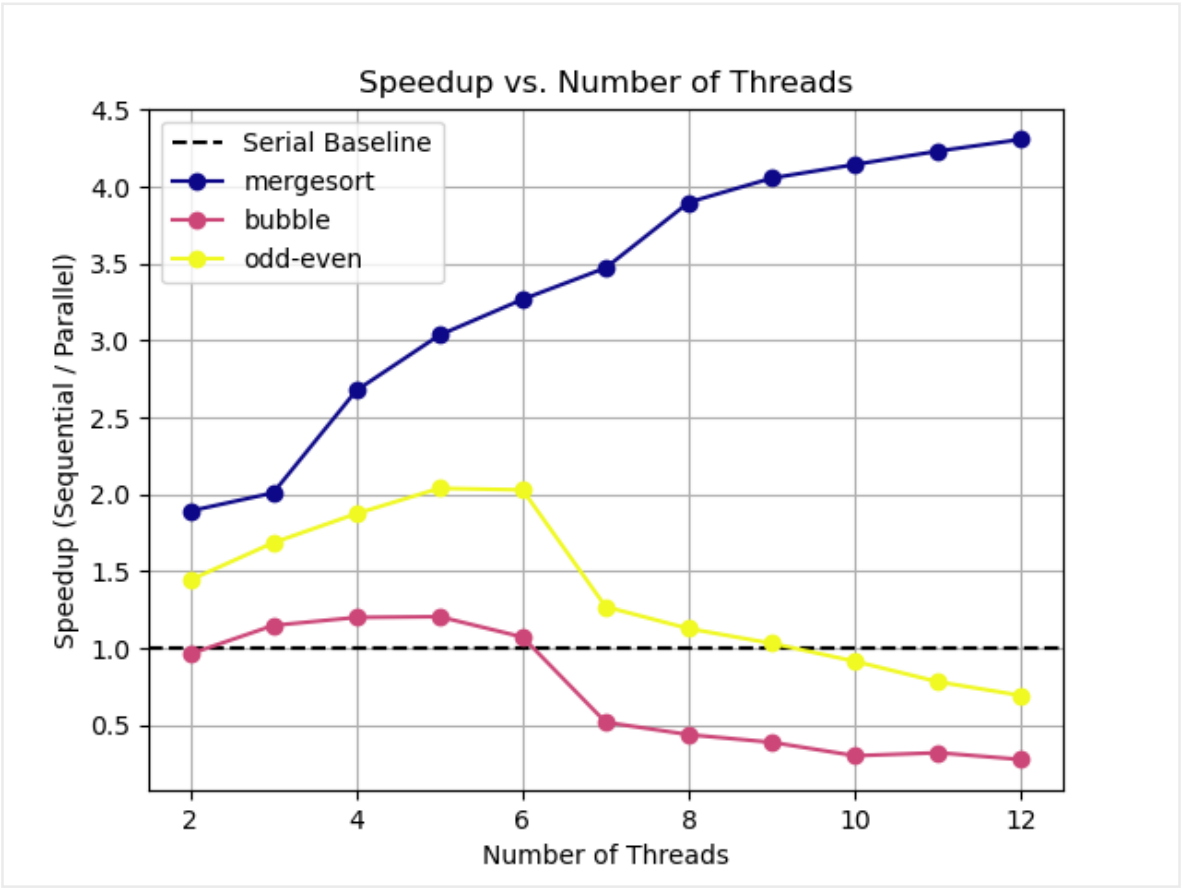

Speedup vs. Problem Size
bubble
Speedup (Sequential / Parallel)
Problem Size (N)
Serial Baseline
2 threads
3 threads
4 threads
5 threads
6 threads
7 threads
8 threads
9 threads
10 threads
11 threads
12 threads
1.2
1.0
0.8
0.6
0.4
0.2
0.0
3
4
5
6
7
8
9
10
11


Execution Time vs. Problem Size
mergesort
Execution Time (s)
Problem Size (N)
Sequential
Parallel (2 threads)
Parallel (3 threads)
Parallel (4 threads)
Parallel (5 threads)
Parallel (6 threads)
Parallel (7 threads)
Parallel (8 threads)
Parallel (9 threads)
Parallel (10 threads)
Parallel (11 threads)
Parallel (12 threads)
0.035
0.030
0.025
0.020
0.015
0.010
0.005
0.000
12
13
14
15
16
17
18
19


Speedup vs. Problem Size
mergesort
Speedup (Sequential / Parallel)
Problem Size (N)
Serial Baseline
2 threads
3 threads
4 threads
5 threads
6 threads
7 threads
8 threads
9 threads
10 threads
11 threads
12 threads
4
3
2
1
0
4
6
8
10
12
14
16
18


Execution Time vs. Problem Size
odd-even
Sequential
Parallel (2 threads)
Parallel (3 threads)
Parallel (4 threads)
Parallel (5 threads)
Parallel (6 threads)
Parallel (7 threads)
Parallel (8 threads)
Parallel (9 threads)
Parallel (10 threads)
Parallel (11 threads)
Parallel (12 threads)
Execution Time (s)
0.025
0.020
0.015
0.010
0.005
0.000
2
4
6
8
10
12
Problem Size (N)


Speedup vs. Problem Size
odd-even
Serial Baseline
2 threads
3 threads
4 threads
5 threads
6 threads
7 threads
8 threads
9 threads
10 threads
11 threads
12 threads
Speedup (Sequential / Parallel)
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
2
4
6
8
10
12
Problem Size (N)



Execution Time and Speedup Analysis on Linux (Apple M3 Pro):

The provided graphs illustrate execution times and speedups for three sorting algorithms—Bubble Sort, Merge Sort, and Odd-Even Sort—tested in sequential and parallel implementations (with thread counts from 2 to 12). The testing environment is Linux running on an Apple M3 Pro chip, with 6 performance cores operating at 4.06 GHz and additional efficiency cores at 2.8 GHz.

Bubble Sort and Odd-Even Sort:

- **Execution Time Analysis:**
    - Both Bubble Sort and Odd-Even Sort experience rapid increases in execution time with growing problem sizes.
    - Parallel implementations show clear performance gains when using up to 6-7 threads, closely matching the processor's number of available performance cores.
    - Beyond 7 threads, execution time begins to increase due to parallelization overhead and the inclusion of efficiency cores, which have lower performance compared to the performance cores.

- **Speedup Analysis:**
    - Speedups for both algorithms are initially above the baseline (speedup > 1), particularly when utilizing up to 6-7 threads.
    - The maximum benefit from parallelization occurs around the number of performance cores, emphasizing their efficiency and processing capability.
    - Past this threshold, the speedup declines sharply due to increased overhead, thread synchronization issues, and the lower performance of efficiency cores.
    - There is a clear gap in the graph in the middle, due to the use of the slower efficiency cores when more than 6 threads are executed

Merge Sort:

- **Execution Time Analysis:**
    - Merge Sort demonstrates significant reductions in execution time when parallelized.
    - The algorithm scales well with higher thread counts.
    - The diminishing returns at higher thread counts reflect a normal pattern of parallelization overhead.
    - The benefits of parallelization remain consistent even beyond 6-7 threads.

- **Speedup Analysis:**
    - Merge Sort shows excellent scalability with increasing threads.
    - It surpasses the sequential baseline by a substantial speedup of approximately 4.5x at 12 threads for larger problem sizes.
    - Merge Sort’s effective divide-and-conquer parallelization capability is demonstrated by its scalability.
    - The presence of additional efficiency cores does not significantly harm Merge Sort’s scalability.

Speedup vs. Number of Threads (Comparative Summary):

- Merge Sort outperforms both Bubble and Odd-Even sorts, achieving sustained high speedups across all tested thread counts.
- Odd-Even Sort achieves moderate speedup benefits, with the best results seen within the number of performance cores (up to 6 threads).
- Bubble Sort gains minimal parallel speedup up to approximately 6 threads, quickly diminishing afterward, reflecting its inherent sequential limitations and higher overhead.
- The Odd-Even sort is a slightly better fit for parallelization compared to the bubble sort because it allows you to work with more independent arrays.

Bonus note:
cpu_stats.c was missing a **return res;** at the end of the **struct cpu_stats_report cpu_stats_end(struct cpu_stats *stats)** non linux function.