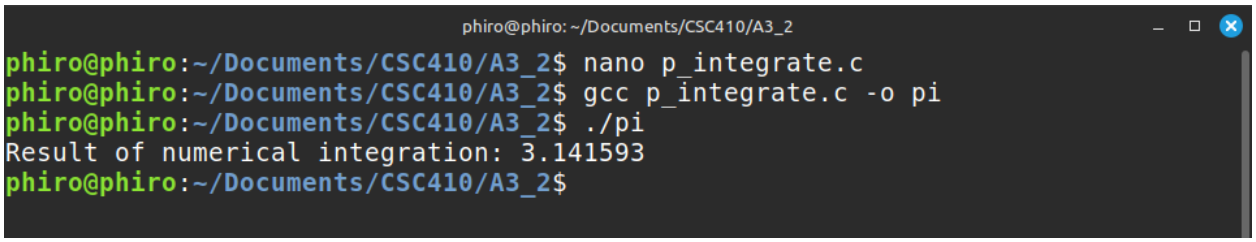
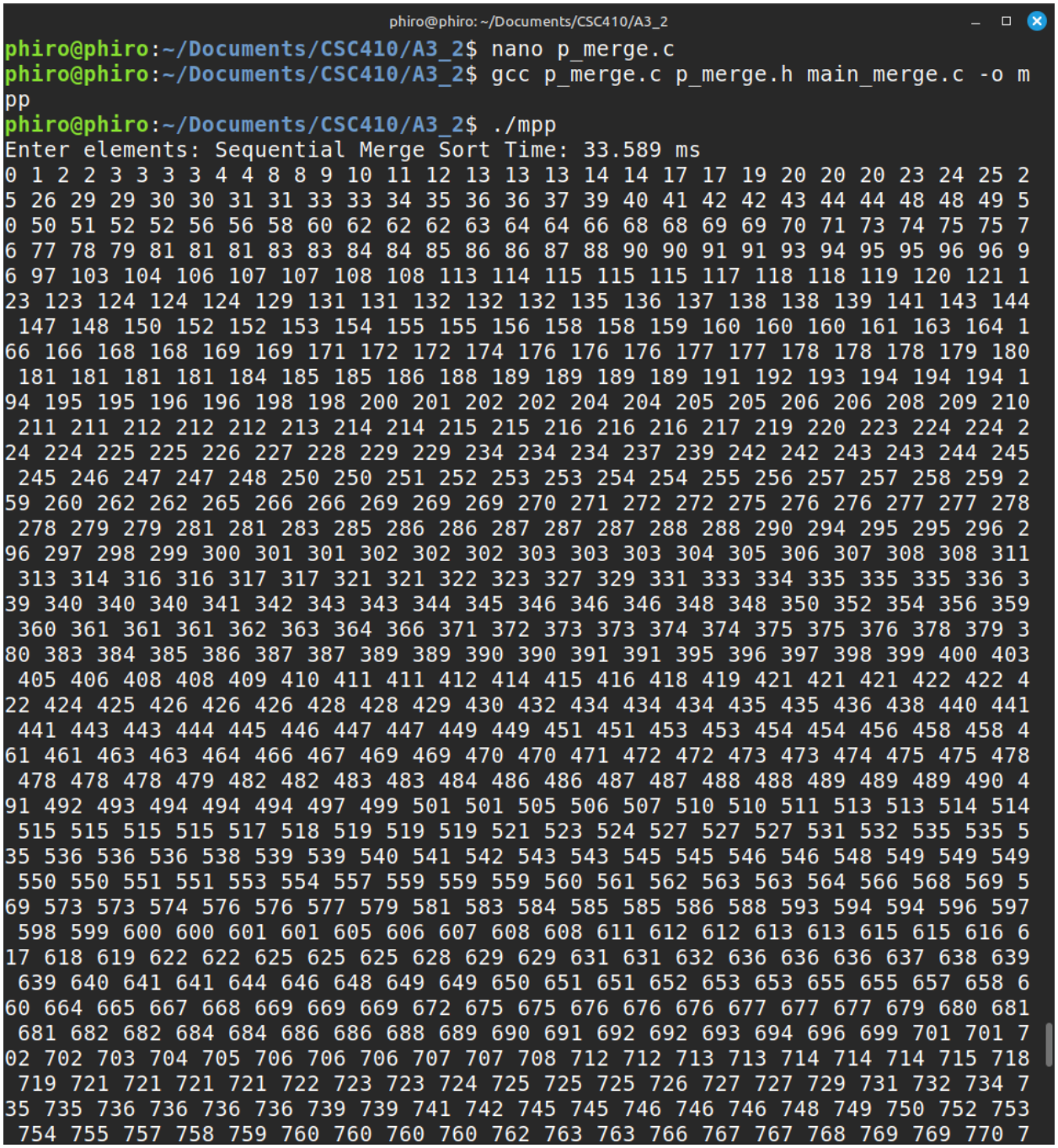
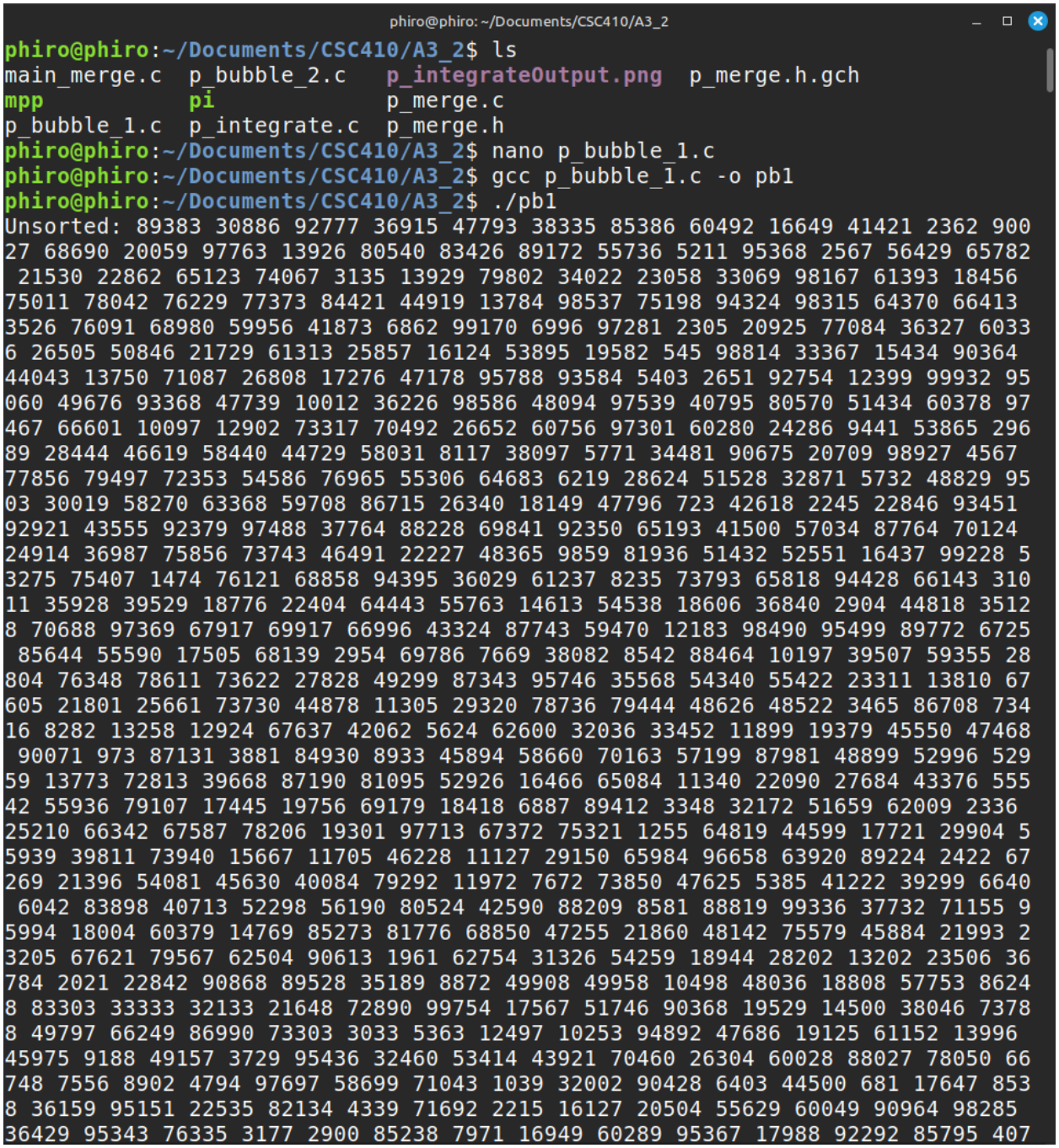
Screenshots:







Task 3:

Parallel Bubble Sort 1 (Segment-Based):
This program divides the array into chunks, and each thread performs bubble passes locally within its segment, synchronizing with barriers after each pass. It is simple to implement and modifies the classic bubble sort with minimal changes. However, heavy synchronization overhead limits speedups, especially due to dependencies at chunk boundaries. This approach

suffers from limited parallelism because the outer loop remains sequential in effect. Overall, its scalability is poor, and performance gains over sequential are minimal for large arrays.

Parallel Bubble Sort 2 (Odd-Even / Brick Sort):
This another version of bubble sort program uses the odd-even transposition method, alternating between even-indexed and odd-indexed adjacent swaps in phases that can safely run in parallel. It achieves better concurrency by avoiding overlapping data access within each phase and synchronizing only at phase boundaries. The algorithm is naturally suited to parallelization with predictable synchronization points and fewer dependencies. Despite this, the total number of phases remains O(n)O(n), so the quadratic time complexity persists. It outperforms the first approach but still far lags behind more advanced parallel sorts for large datasets.

Both approaches parallelize bubble sort but remain limited by bubble sort's inherent quadratic time and synchronization overhead, making them mainly educational tools rather than practical solutions for large or performance-critical datasets.

Task 4:

Program	Sequential	1 Thread	2 Threads	3 Threads	4 Threads
Num_Integrate	14.027	14.027	7.374	7.420	7.876
Merge_approach_1	0.074	0.074	0.063	0.056	0.062
Merge_approach_2	X	0.071	0.061	0.061	0.064
Merge_approach_3	X	0.362	0.342	0.377	0.390
Bubble_1	48.605	48.415	37.445	32.574	31.933
Bubble_2	X	17.831	12.240	16.870	14.036

Task 5:

I noticed that more threads increase the programs' performance but there are not a huge difference in the performance between the programs.

When parallelizing programs using pthreads, it is important to carefully divide the work so that each thread has its own portion of data to process, which reduces contention and race conditions. Synchronization mechanisms like mutexes and barriers must be used to ensure that shared data is not accessed simultaneously, preventing errors and guaranteeing correctness. Balancing the number of threads with the hardware capabilities is key, as too many threads can

cause overhead that outweighs the benefits of parallelism. Another consideration is minimizing the overhead of thread creation and joining, so tasks should be large enough to justify the parallelization cost. Overall, pthreads provide great control and flexibility but require careful programming to avoid common multithreading pitfalls like deadlocks and excessive synchronization.

Pthreads offer direct access to low-level thread management, allowing efficient use of system resources and potentially significant speedups on multicore processors. They are portable across POSIX systems and integrate well with C and C++ programs. The main challenges include complexity in writing and debugging concurrent code, as timing-dependent bugs can be very subtle. Additionally, programmers must explicitly manage all synchronization and data sharing, which increases the chance of errors. Despite this, mastering pthreads is valuable for understanding the foundations of parallel programming and producing high-performance programs when done carefully.