THE ARAB AMERICAN UNIVERSITY

FACULTY OF ENGINEERING

PARALLEL AND DISTRIBUTED PROCESSING

Parallel And Distributed Processing Project 1:

Student: Hashem Erainat

Instructor: Hussein Younes

1. Introduction

In this report, the Histogram Calculation algorithm is examined as a simple statistical tool used in the presentation of the distribution of data based on frequency. The primary purpose of this work is to compare its sequential execution with that of a parallel run and the worth of using parallel processing on this specific algorithmic task.

The Histogram Calculation algorithm is chosen due to its inherent high parallelizability. Each data element can be processed and counted independently without affecting others. This allows for easy division of the input data among multiple threads, with each thread computing a local histogram. These local results are then combined in a final, efficient merge step. This design effectively leverages multi-core processors for significant speedup, especially with large datasets.

2. Sequential Implementation

This section covers the sequential implementation of the histogram algorithm and its time measurement.

The algorithm primarily uses two std::vector<int> objects: data_array to store input numerical data (read from an external file), and histogram_Algorithm_data to hold frequency counts (bins) for values within the defined range. The choice of std::vector is advantageous for its dynamic sizing capability, contiguous memory allocation (which enhances cache efficiency), and O(1) random access by index, all crucial for efficiently handling input data and updating histogram bins.

The central calculation resides in the sequentialHistogram function. It processes each element in the data_array one by one. For every value, it directly increments the corresponding bin in histogram_Algorithm_data by using the value (adjusted by MIN_RANGE_VAL) as an index. This makes the function's time complexity O(N), where N is the number of elements in the input data_array, as it performs a constant number of operations for each element.

3. Parallelization Strategy

3.1 Work Division Among Threads

The parallel strategy employs data decomposition, dividing the input data_array into contiguous segments. Each thread is assigned a unique segment (start_index to end_index). This is achieved by dividing the total array size among NUM_THREADS, with any remaining elements (ARRAY_SIZE % NUM_THREADS) distributed one by one to the initial threads to ensure all data is processed. Crucially, each thread computes its own local histogram, avoiding contention during the counting phase. This solution was chosen because it has better performance since it prevents the high overhead and contention that mutexes cause, so local counting will be much faster than mutex-protected update. After all threads complete, these local histograms are combined in a final sequential merge step into a global result, ensuring efficient parallel counting followed by a minimal synchronization phase.

3.2 Pthread Functions and Structures Used

The parallel implementation leverages the POSIX Threads (Pthreads) library to manage concurrent execution. Key Pthread functions and structures are utilized to facilitate the work division and synchronization necessary for the parallel strategy:

- struct ThreadData: This custom structure is essential for passing specific work parameters to each new thread. It holds the thread_id, start_index, and end_index for its assigned data segment, allowing each thread to know its processing range.
- pthread_create() and pthread_t: The pthread_create() function is used to launch each worker thread, utilizing a pthread_t variable to track its unique identifier. This function takes the thread's entry point (parallelHistogram) and a pointer to its corresponding ThreadData

structure, effectively initiating parallel processing on designated data chunks.

- pthread_join(): After creating all worker threads, the main thread uses pthread_join() to wait for each worker thread to complete its local histogram calculation. This ensures all parallel computations are finished before the sequential merge step begins.

4. Experiments

4.1 Hardware Specifications

The performance tests were conducted on a system with the following hardware and software configuration:

Processor (CPU): Intel Core i7-12700H

Number of Cores/Threads: 14 Physical Cores / 20 Logical Threads

Operating System (OS): Ubuntu 22.04 LTS (WSL)

4.2 Input Sizes and Thread Counts Tested

Performance measurements were performed across various input dataset sizes to evaluate the scalability of both sequential and parallel implementations. For the parallel version, different thread counts were utilized to assess the impact of parallelism on execution time.

The following configurations were systematically tested:

Input Data Sizes:

Small: data_100k.txt (N=100,000 elements)

Medium: data_1M.txt (N=1,000,000 elements)

Large: data_50M.txt (N=50,000,000 elements)

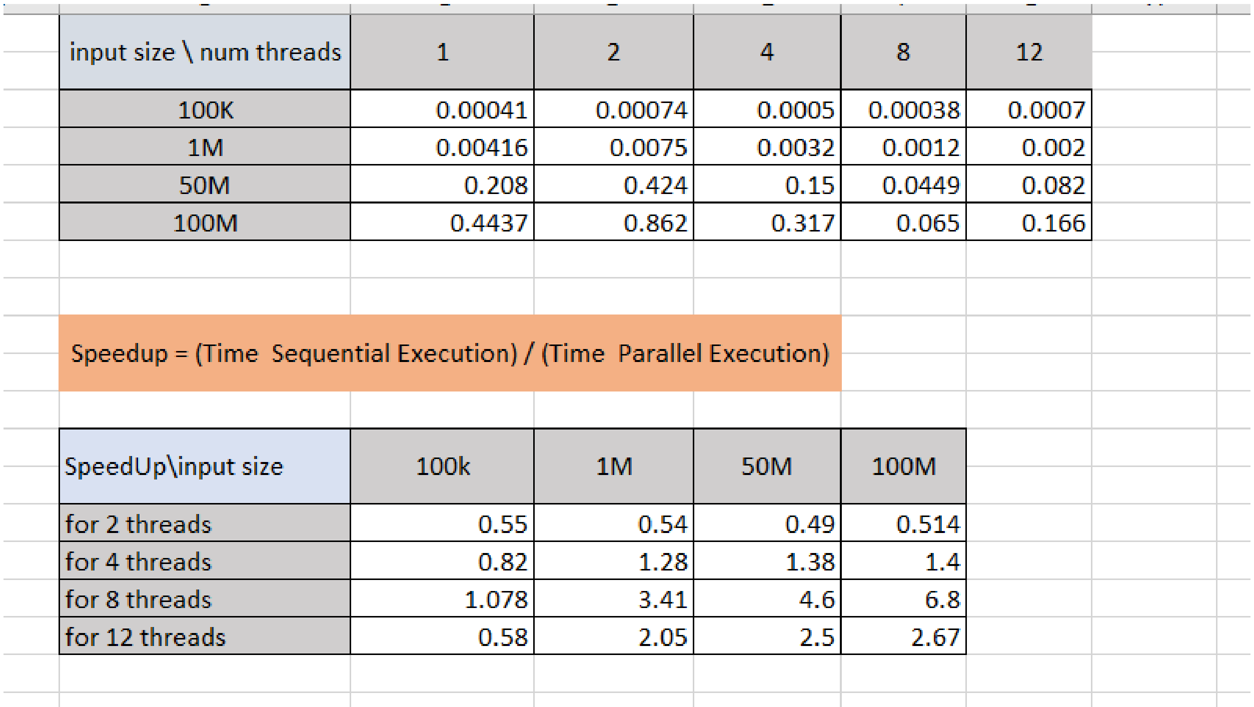
Very Large: data_100M.txt (N=100,000,000 elements)

Thread Counts (for Parallel Implementation):

2,4,8,10 threads

Each combination of input size and thread count was executed 5 times, and the average execution time was recorded to mitigate the effect of system noise and transient processes.

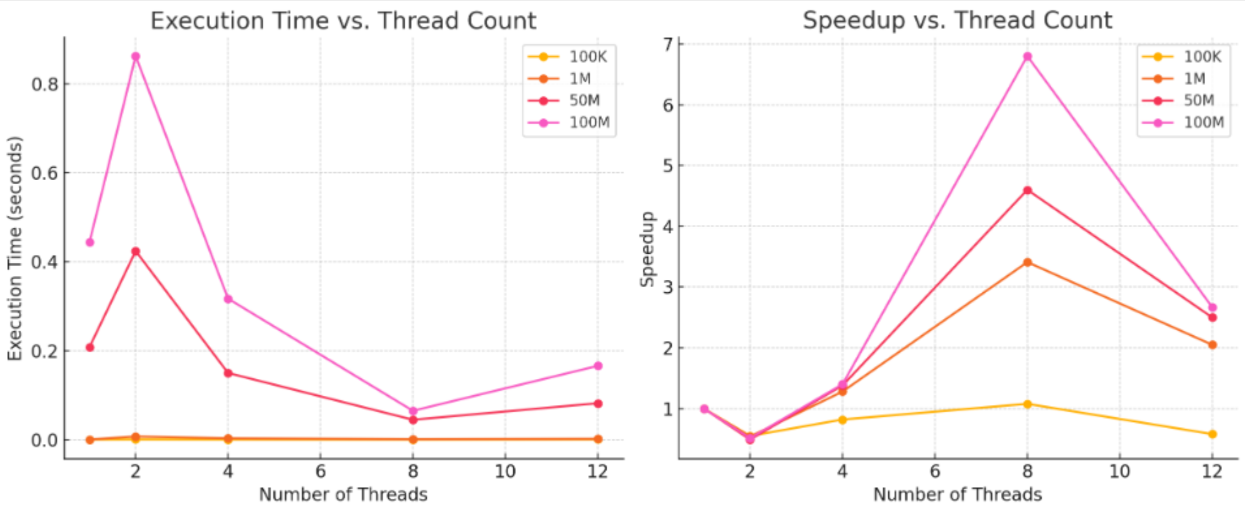
5. Results



input size \ num threads	1	2	4	8	12
100K	0.00041	0.00074	0.0005	0.00038	0.0007
1M	0.00416	0.0075	0.0032	0.0012	0.002
50M	0.208	0.424	0.15	0.0449	0.082
100M	0.4437	0.862	0.317	0.065	0.166

Speedup = (Time Sequential Execution) / (Time Parallel Execution)

SpeedUp\input size	100k	1M	50M	100M
for 2 threads	0.55	0.54	0.49	0.514
for 4 threads	0.82	1.28	1.38	1.4
for 8 threads	1.078	3.41	4.6	6.8
for 12 threads	0.58	2.05	2.5	2.67



6.Discussion

Why speedup is sublinear

Ideal Speedup in parallel computing refers to the hypothetical maximum possible increase in performance, wherein the time taken to execute decreases in proportion to the number of processors (e.g., 4 threads in an ideal scenario deliver 4x speedup). Real parallel programs never quite fulfill this ideal due to intrinsic constraints and overheads.

The key reasons for deviations from ideal speedup are:

- Parallelization Overhead:
  This is the non-computation overhead of parallelizing a program that is not incurred by its sequential counterpart.
  Examples:
  Thread Creation & Management: Time spent creating, managing, and deallocating threads. For small data sets (like your 10K or 100k sizes), this overhead can overshadow the benefits of parallel execution and cause the 2-thread version to be slower than sequential (Speedup < 1).

- Work Distribution: The cost of delivering input data to threads.

Synchronization (Implicit in Merge): While your current code uses local histograms and a sequential merge, in advanced parallel systems, explicit synchronization (e.g., mutexes) adds overhead. Even your sequential merge step contributes to the non-parallelizable component of the total execution time.

- Load Imbalance: This occurs if the workload is not distributed unevenly between the threads, or threads require varying amounts of time to finish their assigned jobs. While our data distribution utilizes a constant chunk size, minor load imbalances can result from variations in data characteristics within chunks or operating system scheduling overhead that can cause some threads to block while others complete their tasks, thus lowering overall efficiency.

Comparison with Amdahl's Law Projections

Amdahl's Law declares the speedup of a parallel program to be directly limited by the proportion of the program that may not be paralleled. Regardless of how much of the effort is done in parallel, eventually the sequential piece such as combining results will turn into a bottleneck as the thread count increases.

Our results reflect this behavior exactly:

In very small input sizes such as 100K and 1M, several-threading management overheads dominate parallelism gains. One can observe from the speedup values, which are less than or around 1.0, and even decrease a little at larger numbers of threads (e.g., speedup goes down from 1.078 using 8 threads to 0.58 using 12 threads for 100K). This conforms to Amdahl's prophecy: if the sequential portion is large enough in comparison to the overall workload, adding threads does not give a significant or even any benefit.

For the larger inputs (50M and 100M), the parallel portion takes over, and we have a much higher speedup. The speedup increases almost linearly to 8 threads approxtemlly 6.8x for 100M, but starts to plateau or even decline after that. This is due to the increasing impact of the sequential merge phase, which does not scale with the number of threads and closely tracks Amdahl's Law.

The Execution Time vs. Number of Threads plot shows diminishing returns past a point after a certain number of threads, specially for big inputs. The Speedup plot also tops at 8 threads and thereafter decreases slightly or levels off, which is exactly what Amdahl's Law predicts when the fraction that can be parallelized (P) is less than 100%.

7.Conclusion

❖ Lessons learned:

- Importance of Data Size in Parallelization:
  We have found that parallelization is not the best solution under all circumstances. For very tiny input sizes (e.g., 100K and 1M), parallelization overhead causes the overheads of thread management and creation (Parallelization Overhead) to overwhelm any likely advantage from using parallel execution and leads to less efficient performance than the sequential. In contrast, parallelization performed extremely well on larger data sizes (50M and 100M), achieving significant performance gain.

- Practical Application of Amdahl's Law:
  We observed empirically how the sequential part of the program bounds the overall speedup, regardless of the amount of threads. That is precisely what

Amdahl's Law describes, where speedup advantage dips or plateaus after a certain quantity of threads.

- We learned that optimizing even a small sequential part can have a substantial impact on the overall performance of a parallel program.

- Importance of Efficient Parallel Construction:
  The selected approach of "local computation followed by sequential merge" was effective in reducing contention during the counting phase. By allowing each thread to calculate its own local histogram, the requirement for explicit synchronization primitives (such as mutexes) during the most computationally intensive phase was eliminated, thus enhancing performance.

❖ challenges Faced:

- Overhead and Load Balancing :
  At first, we struggled a bit with the overhead introduced by using threads. On small input sizes, the extra cost actually made performance worse.
- Bottlenecks in the Merge Step :
  During testing, it became clear that the final step — merging all the partial results was limiting how much speedup we could get, even when using many threads.
- Scalability Limitations :
  We also noticed that adding more threads beyond a certain point didn't help. In fact, sometimes performance got worse due to too much overhead or idle threads. This highlighted the limits imposed by the parts of the program that can’t be parallelized.

8.AI Assistance

- Learning to read a file: Specifically, to be aware of how to read data from a file so that the same dataset was used for each test. (It's stated that even using random data for each case, there likely wouldn't be much variation because the algorithm processes all elements.)