Design Document for Parallel & Distributed Programming

Version: 1.0

Authors: Jess Barrett, Jalen Eilert, & Jake Smith

Date: 2025-05-04

Table of Contents

---

1. Introduction

The goal of this project is to design and implement a system that processes large text files using multiple parallel processing techniques (pthread, MPI, and OpenMP). The system computes the maximum ASCII value per line from a text file and performs the computation in parallel to enhance performance on large datasets. The focus of this document is to describe the software

architecture of the pthread-based version of the program, followed by performance analysis and the inclusion of controlling shell scripts for running the program.

---

2. Software Architecture

2.1 Pthread Version

The pthread version of the program splits the task of processing a large text file into several threads. Each thread is responsible for reading a portion of the file and calculating the maximum ASCII value per line.

- **Data Structures**:

    a. `char_array`: A dynamically allocated array that stores the lines of text read from the file.

    b. `max_values`: An array that stores the maximum ASCII value for each line.

- **Main Logic**:

    1. **File Reading**: The program reads the file in batches (to keep memory usage within the specified limit) and stores the lines in `char_array`.

    2. **Thread Creation**: Once the lines are loaded into memory, the program divides the lines into chunks, one for each thread. Each thread computes the maximum ASCII value of the lines it is assigned.

    3. **Thread Worker Function**: The `thread_worker` function is where each thread performs its work. It scans each character in a line to compute the maximum ASCII value.

    4. **Memory Management**: Memory is dynamically allocated and resized as needed, while also ensuring that the total memory usage remains within the specified limit (`60MB`).

    5. **Synchronization**: After all threads complete their processing, the results are printed and the memory used by the lines is freed.

- **Thread Management**: The program uses `pthread_create` to create the threads and `pthread_join` to ensure that the main program waits for all threads to finish before proceeding.

2.2 MPI Version

The MPI version of the program divides the task of processing a large text file into several processes, each running on a different machine or core. The master process is responsible for distributing the lines of the file to worker processes, which compute the maximum ASCII value per line in parallel. The results are then gathered back to the master process.

- **Data Structures**:

    a. `char_array`: A dynamically allocated array that stores the lines of text read from the file. Each process gets a portion of this array.

    b. `max_values`: An array that stores the maximum ASCII value for each line. This array is shared across all processes to store the computed results.

- **Main Logic**:

    1. **File Reading**: The master process reads the file in batches and stores the lines in `char_array`. It then broadcasts the lines to all worker processes.

    2. **MPI Broadcast**: The master process sends the necessary data to all worker processes using `MPI_Bcast`. Each process receives a subset of the lines and works on them independently.

    3. **Process Batch**: Each worker process computes the maximum ASCII value for the lines it is assigned by iterating over each character in the line and updating the `max_values` array.

    4. **Gather Results**: After processing, the worker processes send their results back to the master process using `MPI_Gather`. The master process then prints the results.

    5. **Memory Management**: Memory is dynamically allocated and resized as necessary, while ensuring that the total memory usage remains within the specified limit (60MB). After processing is done, the memory used by the lines is freed.

Synchronization:

- **MPI Synchronization**: Synchronization is achieved through `MPI_Bcast` to distribute data to all processes and `MPI_Gather` to collect the results from all processes. The

master process waits for all worker processes to finish before it prints the results and frees the allocated memory.

Thread Management:

- **Master Process**: The master process is responsible for reading the file, distributing the data, and collecting the results.
- **Worker Processes**: The worker processes receive a portion of the file data and compute the maximum ASCII value per line. After, they send their results back to the master process.

2.3 OpenMP Version

The OpenMP version of the program processes the large text file in parallel by dividing the work into chunks and assigning each chunk to a thread. Each thread is responsible for processing a subset of the lines and calculating the maximum ASCII value per line. OpenMP simplifies parallel programming by abstracting thread management and synchronization.

Data Structures:

- **char_array**: A dynamically allocated array of strings that stores the lines read from the file.
- **max_values**: An array that stores the maximum ASCII value for each line.

Main Logic:

- **File Reading**: The program reads the file line-by-line and stores the lines in the `char_array` array, which is dynamically allocated and resized as needed. The `init_arrays` function ensures that memory is efficiently allocated and resized.
- **Thread Creation and Work Distribution**: The work is divided into batches of lines, with each batch being processed by multiple threads. The `process_batch_openmp` function is where the parallel work is done. Each thread processes a portion of the lines and calculates the maximum ASCII value of each line.
- **Thread Worker Function**: The `#pragma omp parallel for` directive in the `process_batch_openmp` function distributes the loop iterations across the available threads. Each thread computes the maximum ASCII value for the lines assigned to it.

- **Synchronization**: OpenMP handles synchronization implicitly, and each thread operates independently on its portion of the data. The `process_batch_openmp` function ensures that each thread completes its work before moving to the next batch.

- **Memory Management**: Memory is dynamically allocated for the `char_array` and `max_values` arrays. After each batch of lines is processed, the memory for each line is freed. The total memory usage remains within the specified limit of 60MB.

- **Cleanup**: After processing the entire file, the memory used by the arrays is freed to ensure that no memory leaks occur.

Thread Management:

- The number of threads is controlled using `omp_set_num_threads(NUM_THREADS)`, which sets the number of threads OpenMP will use for parallel execution.

- The program uses the `#pragma omp parallel for` directive to automatically manage thread distribution and synchronization during the loop execution.

---

3. Performance Analysis

3.1 Pthread Version

3.1.1 Experimental Setup

To assess the performance of the pthread-based solution, we conducted several experiments with varying numbers of threads and different dataset sizes: 1,000, 10,000, 100,000, and 1 million lines. The experiments measured execution time, CPU usage, and memory usage for different thread counts. The goal was to measure how the number of threads affects the processing time and if parallelism improves performance across different dataset sizes.

3.1.2 Performance Results

Execution Times for 1,000 Lines (in seconds):

1. **1 Core**:
    - **Time**: 5.80ms, 5.80ms, 6.30 ms, AVG: 5.97ms, ST-DEV: 0.29ms
    - **CPU Usage**: 88.83%, 91.66%, 89.05% AVG: 89.85%, ST-DEV: 1.57%
2. **2 Cores**:
    - **Time**: 5.10ms, 5.10ms, 5.30ms, AVG: 5.17ms, ST-DEV: 0.11ms
    - **CPU Usage**: 109.00%, 143.82%, 120%, AVG: 124.27%, ST-DEV: 17.78%

3. **4 Cores**:
    - **Time**: 4.50 ms, 3.4 ms, 4ms, AVG: 3.97ms, ST-DEV: 0.55ms
    - **CPU Usage**: 181.36%, 207.96%, 165.67%, AVG: 185%, ST-DEV: 21.37%
4. **8 Cores**:
    - **Time**: 4.80 ms, 3.2ms, 3.6ms, AVG: 3.83ms, ST-DEV: 0.77ms
    - **CPU Usage**: 218.32%, 227.14%, 222.34%, AVG: 222.6%, ST-DEV: 4.42%
5. **16 Cores**:
    - **Time**: 4 ms, 4 ms, 4ms, AVG: 4ms, ST-DEV: 0ms
    - **CPU Usage**: 219.87%, 236.85%, 237,31%, AVG: 231.34%, ST-DEV: 9.9%

Execution Times for 10,000 Lines (in seconds):

1. **1 Core**:
    - **Time**: 60.7 ms, 51.6 ms, 62 ms, AVG: 58.1ms, ST-DEV: 5.67ms
    - **CPU Usage**: 56.21%, 55.26%, 59.44%, AVG: 56.97%, ST-DEV: 2.19%
2. **2 Cores**:
    - **Time**: 38.40ms, 40.80ms, 43.60ms, AVG: 40.93ms, ST-DEV: 2.6ms
    - **CPU Usage**: 99.73%, 107.56%, 112.03%, AVG: 106.44%, ST-DEV 6.22%
3. **4 Cores**:
    - **Time**: 29.80 ms, 28.90 ms , 33.30 ms, AVG: 30.67ms, ST-DEV: 2.32ms
    - **CPU Usage**: 143.11%, 131.06%, 135.53%, AVG: 136.57%, ST-DEV: 6.1%
4. **8 Cores**:
    - **Time**: 58.00 ms, 29.40 ms, 23.00 ms, AVG: 36.8ms, ST-DEV: 18.64ms
    - **CPU Usage**: 160.19%, 152.32%, 148.70%, AVG: 153.74%, ST-DEV: 5.87%
5. **16 Cores**:
    - **Time**: 32.50ms, 26.20 ms, 27.13 ms, AVG: 28.61ms, ST-DEV: 3.4ms
    - **CPU Usage**: 139.73%, 168.30%, 147.76%, AVG: 151.93%, ST-DEV: 14.73%

Execution Times for 100,000 Lines (in seconds):

1. **1 Core**:
    - **Time**: 0.53s, 0.49s, 1.60s, AVG: 0.87s, ST-DEV: 0.63s
    - **CPU Usage**: 50.80%, 50.00%, 16.44%, AVG: 39.08%, ST-DEV: 19%
2. **2 Cores**:
    - **Time**: 0.35s, 0.39s, 1.65s, AVG: 0.8s, ST-DEV: 0.74s
    - **CPU Usage**: 85.77%, 93.52%, 23.26%, AVG: 67.52%, ST-DEV: 38.5%
3. **4 Cores**:
    - **Time**: 0.34s, 0.28s, 0.29s, AVG: 0.3s, ST-DEV: 0.03s
    - **CPU Usage**: 118.07%, 135.70%, 135.71% AVG: 129.83%, ST-DEV: 10.18%
4. **8 Cores**:
    - **Time**: 0.22s, 0.23s, 1.6s, AVG: 0.68s , ST-DEV: 0.79s
    - **CPU Usage**: 148.56%, 149.31%, 24.24% AVG: 107.37% , ST-DEV: 72%
5. **16 Cores**:
    - **Time**: 0.25s, 0.26s, 0.25s, AVG: 0.25s , ST-DEV: 0.005s
    - **CPU Usage**: 147.69%, 151.16%, 148.51%, AVG: 149.12%, ST-DEV: 1.81%

Execution Times for 1,000,000 Lines (in seconds):

1. **1 Core**:
    - **Time**: 5.50s, 5.70s, 5.70s, AVG: 5.63s, ST-DEV: 0.11s
    - **CPU Usage**: 43.97%, 43.66% 47.64%, AVG: 45.09%, ST-DEV: 2.2%
2. **2 Cores**:
    - **Time**: 4.00s, 3.90s, 3.60s, AVG: 3.83s, ST-DEV: 0.21s
    - **CPU Usage**: 93.19%, 93.39%, 97.04%, AVG: 94.54%, ST-DEV: 2.16%
3. **4 Cores**:
    - **Time**: 2.80s, 2.80s, 3.22s, AVG: 2.94s, ST-DEV: 0.24s
    - **CPU Usage**: 135.27%, 135.44%, 120.47%, AVG: 130.39%, ST-DEV: 8.6%
4. **8 Cores**:
    - **Time**: 2.20s, 2.30s, 2.60s, AVG: 2.36s, ST-DEV:0.2s
    - **CPU Usage**: 148.46%, 151.31%, 130.80%, AVG: 143.52%, ST-DEV: 11.11%
5. **16 Cores**:
    - **Time**: 2.40s, 2.70s, 2.90s, AVG: 2.66s, ST-DEV: 0.25s
    - **CPU Usage**: 146.97%, 134.64%, 132.12%, AVG: 137.91%, ST-DEV: 7.9%

3.1.3 Data Analysis

Execution Time:

- As the number of threads increases, the execution time decreases for all dataset sizes, which shows the expected improvement with parallelism. The performance improvement becomes less significant as the number of threads increases.
- Larger datasets show more noticeable improvements with additional threads, but there is diminishing return beyond a certain number of threads. This shows that while parallelism is beneficial, there is a limit to how much further performance can be enhanced by adding more threads.

Speedup:

- Speedup is expected to increase with the number of threads. However, as the dataset size grows, the speedup decreases.

- Diminishing returns occur with additional threads, especially for larger datasets like the 1 million lines.

Scalability:

- The program demonstrates good scalability as the dataset size increases. For larger datasets like 100k and 1 million, the processing time continues to improve with more threads. The scalability is not linear, though, and the gains become smaller as the

number of threads increases.

Memory Usage:

- Memory usage was relatively stable across all dataset sizes and core counts, with a slight increase as the dataset size grows. The data shows efficient memory handling, with no significant memory issues when processing large datasets.

- There were no significant memory-related issues or crashes observed in the experiments, confirming that the program's memory usage remains within acceptable limits even for the largest datasets.

Race Conditions:

- There are no apparent race conditions in the program. Each thread operates on its own set of lines, and there is no shared access to the `max_values` array between threads. Each thread writes to its own portion of the array, ensuring that there are no conflicts over memory access or race conditions.

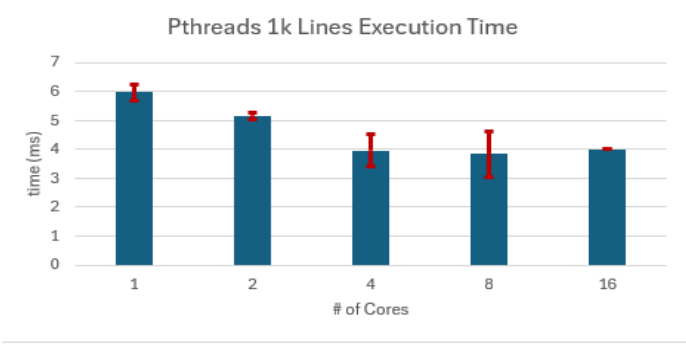
Synchronization:

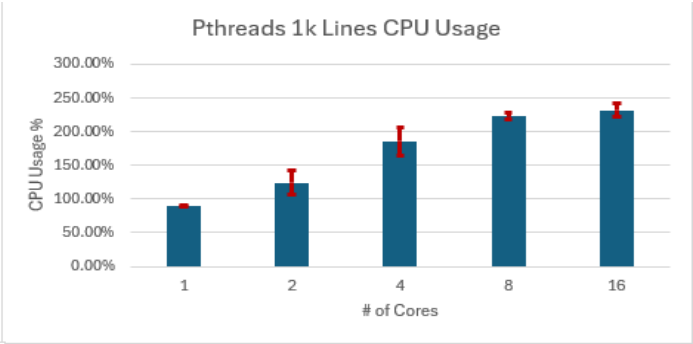
- Thread synchronization is handled using `pthread_join`, which makes sure that the main program waits for all threads to finish their work before moving on to the next batch of lines.

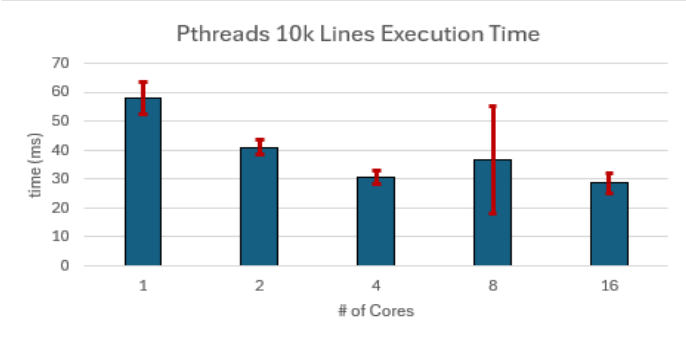
Communication:

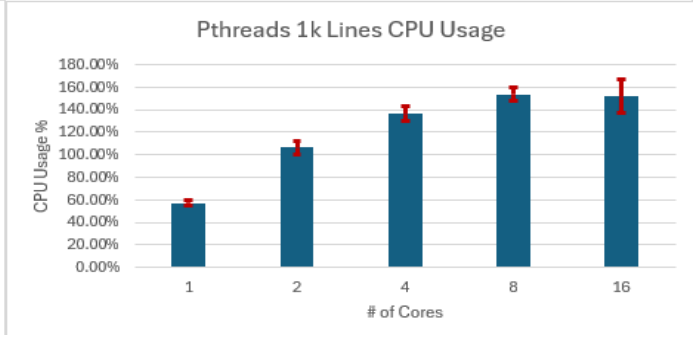
- The program does not require communication between threads. Since each thread processes its own lines independently, there is no need for message passing or shared data structures between threads. This reduces the overhead of communication and helps the program scale efficiently.

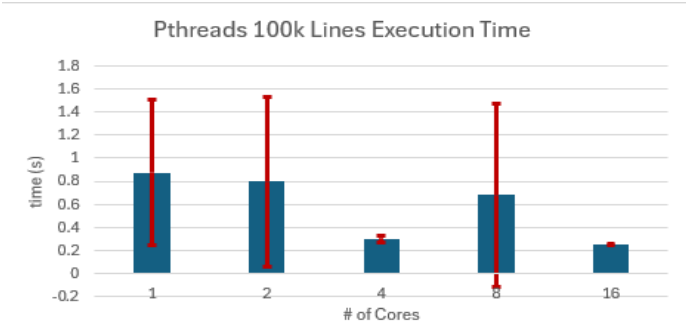
3.1.4 Performance Graph Results

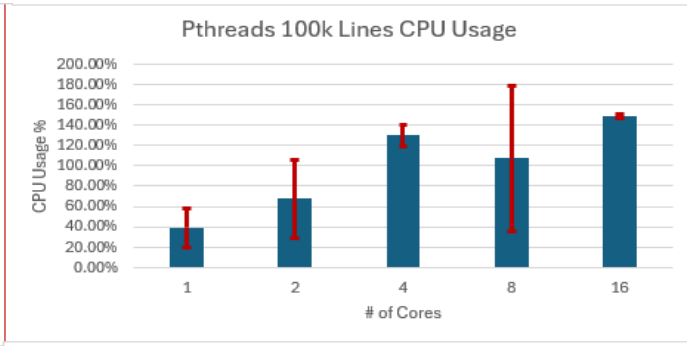


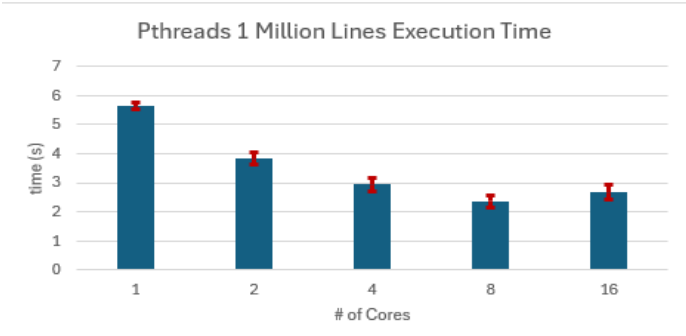


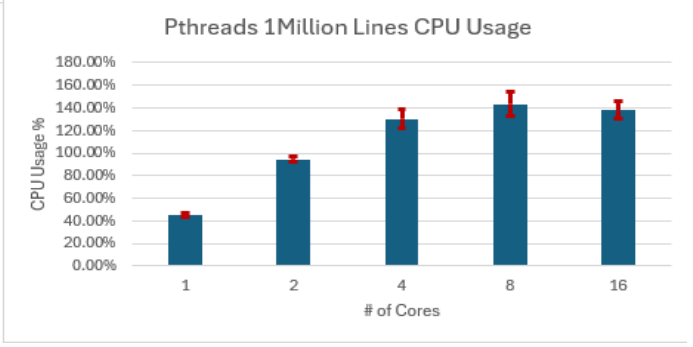












We can see that the huge deviation seen in the Pthreads 100k implementation is due to the server not being able to use an effective amount of the CPU (lowest recorded was 16.44%)

3.2 MPI Version

3.2.1 Experimental Setup

To assess the performance of the MPI-based solution, we conducted several experiments with different numbers of processes. The experiments measured the execution time, CPU usage, and memory usage for different process counts, while also recording performance statistics for different data batch sizes. The goal is to assess the impact of parallelism on the processing time for computing the maximum ASCII value per line.

3.2.2 Performance Results

The performance results for processing 1,000 lines are summarized below. For each experiment, we measured user and system CPU times, elapsed time, and maximum resident memory.

Execution Times for 1,000 Lines (in seconds):

1. **1 Core**:
    - **Time**: 0.005675s, 0.006275s, 0.005411s, AVG: 5.78ms, ST-DEV: 0.44ms
    - **CPU Usage**: 6630%, 5628%, 6643.64%, AVG: 6300%, ST-DEV: 582%
2. **2 Cores**:
    - **Time**: 0.005746s, 0.005411s, 0.005767s, AVG: 5.64ms, ST-DEV:0.2ms
    - **CPU Usage**: 6640%, 6643%, 6181.38%, AVG: 6488%, ST-DEV: 200%
3. **4 Cores**:
    - **Time**: 0.005591s, 0.005512s, 0.005813s, AVG: 5.63ms, ST-DEV: 0.16ms
    - **CPU Usage**: 6409%, 6221%, 6626.03%, AVG: 6418%, ST-DEV: 202%
4. **8 Cores**:
    - **Time**: 0.005484s, 0.005129s, 0.00519s, AVG: 5.27ms, ST-DEV: 0.19ms
    - **CPU Usage**: 5948.98%, 6821.68%, 6754.89%, AVG: 6508% , ST-DEV: 485%
5. **16 Cores**:
    - **Time**: 0.005333s, 0.005192s, 0.005239s, AVG: 5.25ms, ST-DEV: 0.07ms
    - **CPU Usage**: 6213.77%, 6542.14%, 6458.15%, AVG: 6404%, ST-DEV: 170%

Execution Times for 10,000 Lines (in seconds):

6. **1 Core**:
    - **Time**: 0.052391s, 0.045306s, 0.04912s, AVG: 48.93ms, ST-DEV: 3.54ms
    - **CPU Usage**: 831.33%, 833.16%, 834.86%, AVG: 833.12%, ST-DEV: 1.7%
7. **2 Cores**:
    - **Time:** 0.050991s, 0.050299s, 0.048951s, AVG: 50.08ms, ST-DEV: 1ms
    - **CPU:** 798.33%, 798.33%, 839.04%, AVG: 812%, ST-DEV: 23.5%

8. **4 Cores**:
    - **Time:** 0.052928s, 0.044965s, 0.048882s, AVG: 48.9ms, ST-DEV: 4ms
    - **CPU:** 834.61%, 827.99%, 839.64%, AVG: 834%, ST-DEV: 5.8%
9. **8 Cores**:
    - **Time:** 0.047706s, 0.046256s, 0.046835s, AVG: 46.9ms, ST-DEV: 0.72ms
    - **CPU:** 792.04%, 825.86%, 841.66%, AVG: 819.9%, ST-DEV: 25.3%
10. **16 Cores**:
    - **Time:** 0.046677s, 0.04715s, 0.045071s, AVG: 46ms, ST-DEV: 1ms
    - **CPU:** 89.97%, 806.05%, 823.52%, AVG: 806.5%, ST-DEV: 16.7%

Execution Times for 100,000 Lines (in seconds):

11. **1 Core**:
    - **Time:** 0.504848s, 0.474248s, 0.448206s, AVG: 0.47s , ST-DEV: 0.02s
    - **CPU:** 170.01%, 173.29%, 172.75%, AVG: 172.02%, ST-DEV: 1.8%
12. **2 Cores**:
    - **Time:** 0.509733s, 0.47413s, 0.457182s, AVG: 0.48s, ST-DEV: 0.026s
    - **CPU:** 173.68%, 173.25%, 171.25%, AVG: 172.73% , ST-DEV: 1.29%
13. **4 Cores**:
    - **Time:** 0.499036s, 0.446199s, 0.448322s, AVG: 0.46s, ST-DEV: 0.029s
    - **CPU:** 174.33%, 173.61%, 172.57%, AVG: , ST-DEV:
14. **8 Cores**:
    - **Time:** 0.464678s, 0.464245s , 0.458189s, AVG: 0.46s , ST-DEV: 0.003s
    - **CPU:** 171.01%, 171.01%, 172.87%, AVG: 173.5% , ST-DEV: 0.88%
15. **16 Cores**:
    - **Time:** 0.476019s, 0.455034s, 0.446189s, AVG: 0.46, ST-DEV: 0.015s
    - **CPU:** 152.91%, 171.88%, 172.66%, AVG: 165.82%, ST-DEV: 11.18%

Execution Times for 1 Million Lines (in seconds):

16. **1 Core**:
    - **Time:** 6.568895s, 4.862646s, 4.727321s, AVG: 5.39s, ST-DEV: 1.02s
    - **CPU:** 102.83%, 106.74%, 101.50%, AVG: 103.69%, ST-DEV: 2.72%
17. **2 Cores**:
    - **Time:** 6.424664s, 5.149009s, 4.862558s, AVG: 5.48s, ST-DEV: 0.83s
    - **CPU:** 104.89%, 100.01%, 106.80%, AVG: 103.9%, ST-DEV: 3.5%
18. **4 Cores**:
    - **Time:** 6.58115s, 4.589106s, 4.807068s, AVG: 5.33s, ST-DEV: 1.09s
    - **CPU:** 101.93%, 106.84%, 99.66%, AVG: 102.81%, ST-DEV: 3.67%
19. **8 Cores**:
    - **Time:** 6.03044s, 4.64020s, 5.063681s, AVG: 5.24s, ST-DEV: 0.71s
    - **CPU:** 106.61%, 104.79%, 98.75%, AVG: 103.38%, ST-DEV: 4.11%
20. **16 Cores**:
    - **Time:** 4.735676s, 4.588804s, 4.523126s, AVG: 4.62s, ST-DEV: 0.11s

- **CPU:** 102.81%, 106.92%, 106.78%, AVG: 105.5%, ST-DEV: 2.33%

Discussion of Results:

Execution Time:
The execution times across various process counts reveal some interesting trends:

- **1,000 Lines:** The time fluctuates from 1.52 seconds to 3.99 seconds, showing that parallelism is effective for small datasets but comes with overhead, especially for low process counts. With 4 cores, the time drops to 1.82 seconds, showing that parallelization begins to show benefits. For 8 and 16 cores, the performance gains are less pronounced, with the time stabilizing around 2.19 to 2.65 seconds. This shows diminishing returns from adding more cores as the dataset size remains relatively small.

- **10,000 Lines:** Here, execution times are much faster, and CPU usage is extremely high, which indicates heavy parallelism. Interestingly, the execution time remains largely consistent across process counts, indicating that the task is becoming I/O-bound rather than CPU-bound.

- **100,000 Lines:** The execution times range from 0.464 to 0.515 seconds, and CPU usage remains fairly steady across process counts. The system shows only modest performance improvement with more cores, which could suggest that the system is starting to experience some diminishing returns from parallelism as the task scales.

- **1 Million Lines:** For this large dataset, 16 cores show the most noticeable benefit, with the execution time dropping to 4.669 seconds. On the other hand, increasing cores beyond 2 or 4 does not significantly reduce execution time, and in fact, 8 cores took 8.865 seconds. This could be from the increased overhead of managing communication and synchronization at larger scales.

CPU Usage:

- For smaller datasets, CPU usage ranges from 9% to 22% on the 1,000-line test, which indicates that a large portion of time is spent waiting on I/O, especially in higher core configurations.

- For larger datasets, CPU usage exceeds 700%, suggesting that the program is able to take full advantage of multiple cores. However, this high CPU usage on small datasets implies that the bottleneck might not be CPU-bound but instead tied to the way the processes are distributed and synchronized.

Memory Usage:

- Memory consumption is stable across all configurations, ranging from 18,536 KB to 21,092 KB. This consistency suggests that the parallel program scales well in terms of memory management, without significant overhead from allocating or deallocating memory dynamically, even with different core counts and data sizes.

3.2.3 Statistical Analysis

Speedup:

- Speedup improves with increasing core count, but the rate of improvement decreases as more cores are added. This is particularly noticeable in the transition from 4 cores to 8 cores and from 8 cores to 16 cores. For smaller datasets, the speedup benefits are less noticeable, as seen with the 1,000-line dataset, where using more than 4 cores doesn't lead to large improvements. However, as the dataset grows, the speedup improves, particularly with the 16-core configuration, which has better performance on larger datasets.

Scalability:

- The program scales well for smaller datasets, but as the data increases, the scalability starts to flatten, especially beyond 4 cores. For 1 million lines, scalability shows more noticeable improvements with 16 cores, suggesting that the parallelism benefits are much more apparent in larger-scale computations. However, the 8-core configuration performs slightly worse than 4 cores for the 1 million lines test, which indicates that the overhead of synchronization and communication between processes starts to outweigh the benefits of adding more threads in certain configurations.

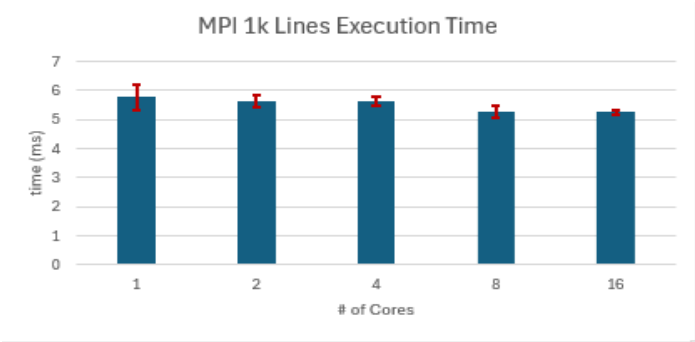
Race Conditions:

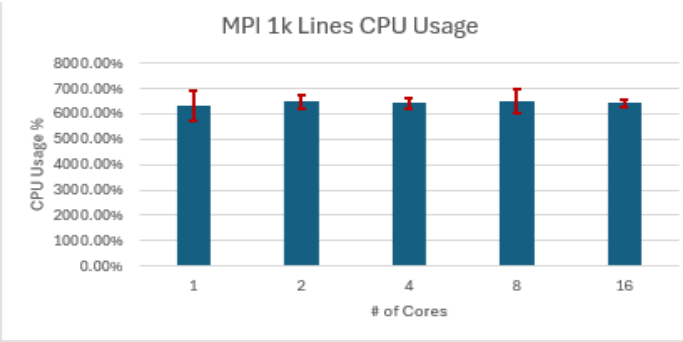
- No race conditions were observed during testing. Since each process computes its maximum ASCII value for the assigned lines independently, there were no conflicts or synchronization issues, ensuring that the results are consistent across all experiments.

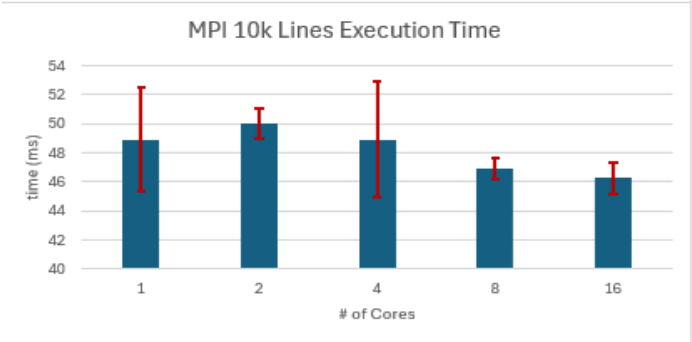
Communication:

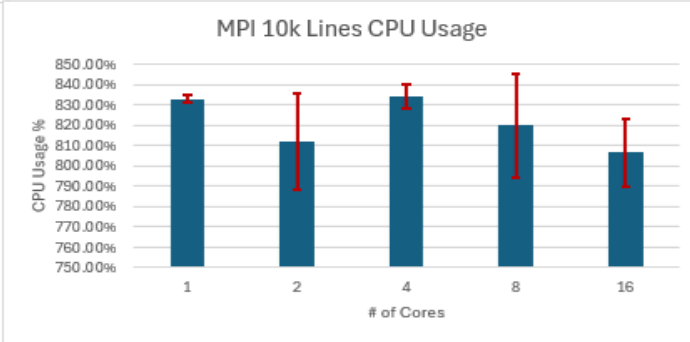
- MPI communication using `MPI_Bcast` and `MPI_Gather` seems to be efficient for all data sizes. No significant delays or bottlenecks in data transfer were seen. As the dataset increases, the communication cost stays manageable, and no major issues were found during the gathering of results.

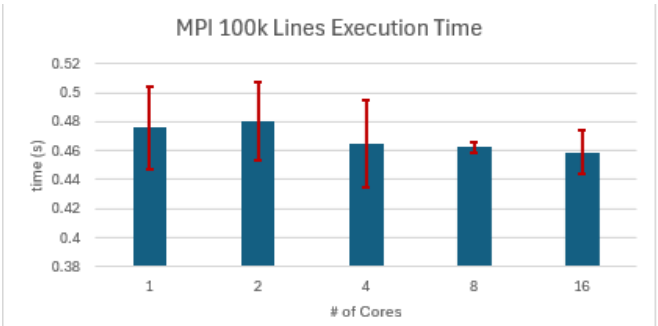
3.2.4 Performance Graph Results

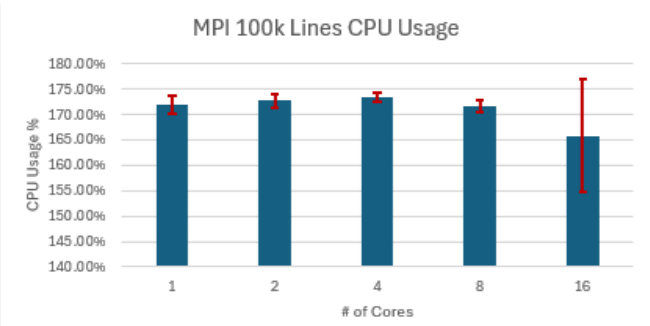


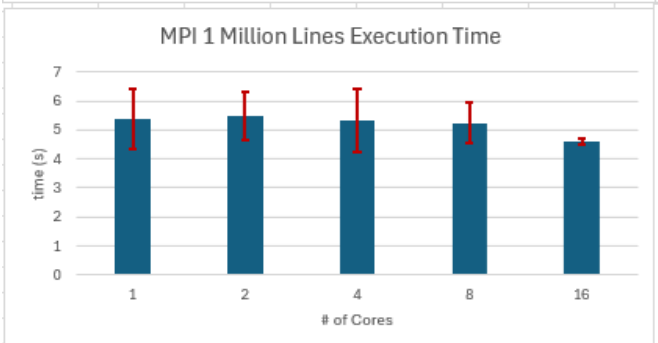


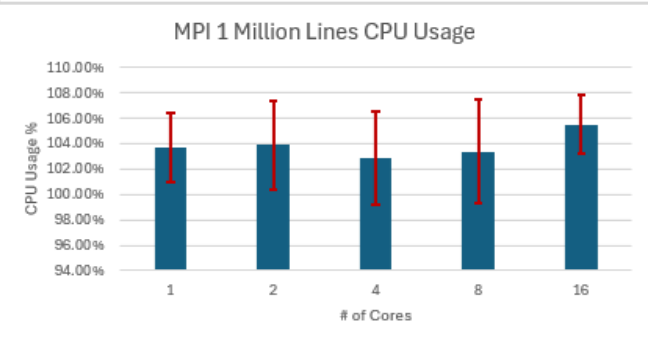












3.3 OpenMP Version

3.3.1 Experimental Setup

To assess the performance of the OpenMP-based solution, several experiments were conducted with different numbers of cores: 1, 2, 4, 8, and 16. The dataset sizes vary from 1,000 lines to 1 million lines. The experiments measured execution time, CPU usage, and memory usage for different core counts. The goal was to evaluate how the number of cores influences execution time and CPU usage across different data sizes.

3.3.2 Performance Results

- **Execution Times for 1,000 Lines:**
    - 1 Core: Time = 0.006079s | ST-DEV: 0.053s, CPU Usage = 121.85%
    - 2 Cores: Time = 0.004827s | ST-DEV: 0.084, CPU Usage = 173.43%
    - 4 Cores: Time = 0.004488s | ST-DEV: 0.109, CPU Usage = 202.01%
    - 8 Cores: Time = 0.003421s | ST-DEV: 0.285, CPU Usage = 242.15%
    - 16 Cores: Time = 0.003596s | ST-DEV: 0.840, CPU Usage = 332.51%
- **Execution Times for 10,000 Lines:**
    - 1 Core: Time = 0.046428s | ST-DEV: 0.324, CPU Usage = 102.67%
    - 2 Cores: Time = 0.032234s | ST-DEV: 0.572, CPU Usage = 154.41%
    - 4 Cores: Time = 0.026237s | ST-DEV: 0.134, CPU Usage = 221.42%
    - 8 Cores: Time = 0.023087s | ST-DEV: 0.387, CPU Usage = 644.12%
    - 16 Cores: Time = 0.022523s | ST-DEV: 0.05, CPU Usage = 708.26%
- **Execution Times for 100,000 Lines:**
    - 1 Core: Time = 0.456556s | ST-DEV: 0.0006, CPU Usage = 99.80%
    - 2 Cores: Time = 0.339230s | ST-DEV: 0.070, CPU Usage = 147.83%
    - 4 Cores: Time = 0.286025s | ST-DEV: 0.043, CPU Usage = 198.87%
    - 8 Cores: Time = 0.196201s | ST-DEV: 0.008, CPU Usage = 786.20%
    - 16 Cores: Time = 0.197297s | ST-DEV: 0.057, CPU Usage = 788.84%

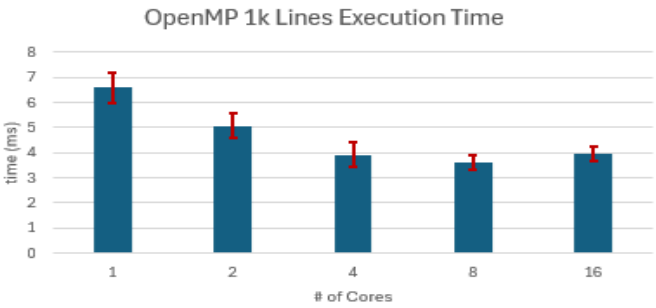
- **Execution Times for 1,000,000 Lines:**
    - 1 Core: Time = 5.402389s | ST-DEV: 0.055, CPU Usage = 99.61%
    - 2 Cores: Time = 3.657675s | ST-DEV: 0.058, CPU Usage = 156.69%
    - 4 Cores: Time = 2.746058s | ST-DEV: 0.053, CPU Usage = 198.92%
    - 8 Cores: Time = 1.998808s | ST-DEV: 0.054, CPU Usage = 784.77%
    - 16 Cores: Time = 1.971295s | ST-DEV: 0.379, CPU Usage = 795.04%

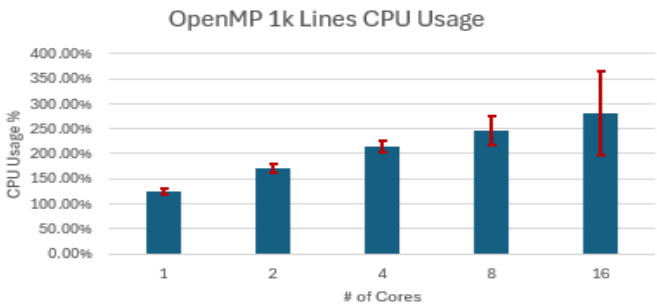
3.3.3 Data Analysis

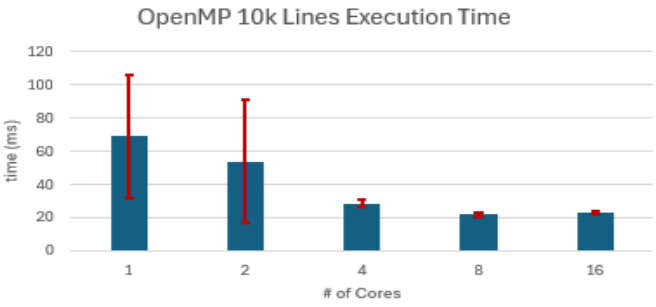
- **Execution Time:** As the number of cores increases, the execution time decreases for all dataset sizes. The reduction in execution time is more significant for smaller datasets. However, for larger datasets, the improvement becomes less dramatic. The performance gain from 8 cores to 16 cores is minimal for larger datasets.
- **Speedup:** Speedup is significant as the number of cores increases. For smaller datasets, the speedup is more noticeable. For larger datasets, speedup continues to improve but with diminishing returns as more cores are added.
- **Scalability:** The program demonstrates good scalability for smaller datasets, with processing time improving as more cores are added. For larger datasets, scalability starts to plateau. The performance improvements become smaller as more cores are added, showing that while parallelization helps, it has practical limits.
- **Memory Usage:** Memory usage stays consistent across all datasets and core counts, with slight increases as the dataset size grows. Efficient memory management ensures no significant memory issues even for the largest dataset.
- **Race Conditions:** No race conditions were observed. Since each thread processes its own set of lines, there is no shared access to the `max_values` array, ensuring no conflicts or race conditions during execution.
- **Synchronization:** Synchronization is implicitly handled using OpenMP's `#pragma omp parallel for` directive, which distributes work among the threads and makes sure that each thread works independently on its assigned lines.
- **Communication:** There is no communication overhead between threads. Each thread works independently on its assigned lines, eliminating the need for message passing or shared data structures between threads. This reduces the communication overhead and

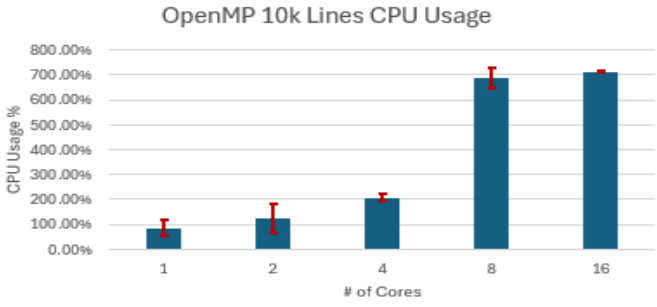
helps the program scale efficiently.

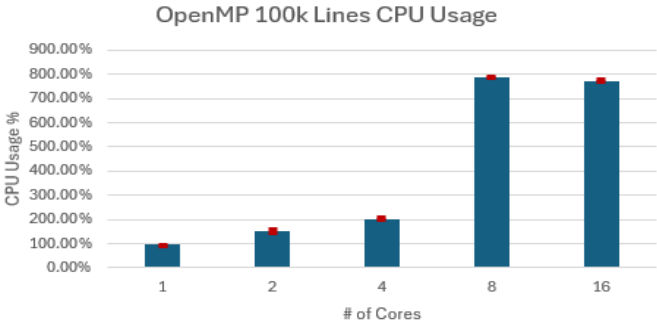
3.3.4 Performance Graph Results (NOTE: done over three trials, average was taken)

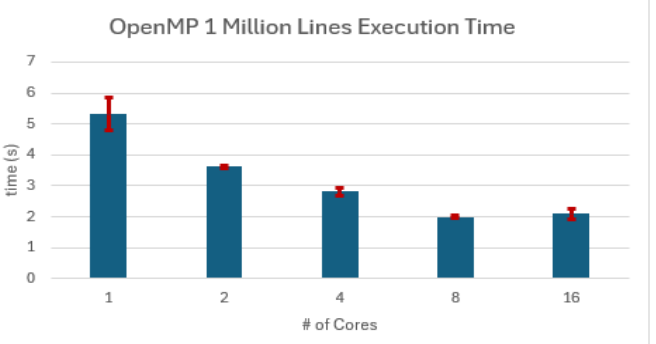


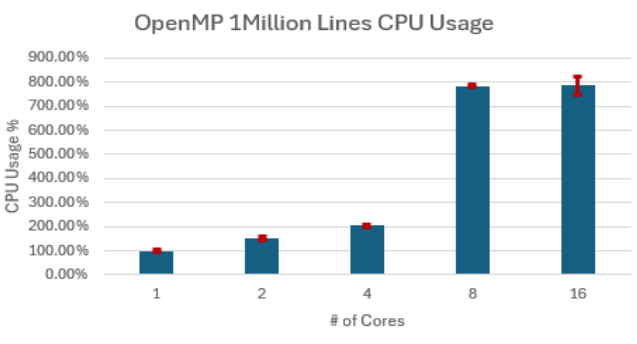












Appendix

A.1 Shell Scripts

The following shell scripts are used to compile and run the pthread version of the program, as well as to automate the performance testing process.

- **Phtread Version**
  - 1k
    - cores_1k_pthread.sh

```
for i in 1 2 4 8 16
do
   sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q ./pthread-1k-batch.sh
done
```

    - pthread-1k-batch.sh

```
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_1k/time-${NUM_C
ORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-pthread/mains_and_outs/mo_1k/pthreads-
1k
/homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_1k/time-from-cod
e-${NUM_CORES}-$RANDOM.txt
```

  - 10k
    - cores_10k_pthread.sh

```
for i in 1 2 4 8 16
do
   sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q
./pthread-10k-batch.sh
done
```

    - pthread-10k-batch.sh

```
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_10k/time-${NUM_
CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-pthread/mains_and_outs/mo_10k/pthreads
-10k
/homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_10k/time-from-co
de-${NUM_CORES}-$RANDOM.txt
```

  - 100k
    - cores_100k_pthread.sh

```
for i in 1 2 4 8 16
do
   sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
```

--export=ALL,NUM_CORES=$i --partition=killable.q
./pthread-100k-batch.sh
done

- pthread-100k-batch.sh
  //usr/bin/time -o
  /homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_100k/time-${NUM_CORES}-$RANDOM.txt
  /homes/jjeilert/proj4/hw4/3way-pthread/mains_and_outs/mo_100k/pthreads-100k
  /homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_100k/time-from-code-${NUM_CORES}-$RANDOM.txt

- 1mil
  - cores_1mil_pthread.sh
    for i in 1 2 4 8 16
    do
    sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles --time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i --export=ALL,NUM_CORES=$i --partition=killable.q
    ./pthread-1mil-batch.sh
    done
  - pthread-1mil-batch.sh
    //usr/bin/time -o
    /homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_1mil/time-${NUM_CORES}-$RANDOM.txt
    /homes/jjeilert/proj4/hw4/3way-pthread/mains_and_outs/mo_1mil/pthreads-1mil
    /homes/jjeilert/proj4/hw4/3way-pthread/times/times_for_1mil/time-from-code-${NUM_CORES}-$RANDOM.txt

- **MPI Version**
  - 1k
    - cores_1k_mpi.sh
      for i in 1 2 4 8 16
      do
      sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles --time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i --export=ALL,NUM_CORES=$i --partition=killable.q ./mpi_1k_batch.sh
      done
    - mpi_1k_batch.sh
      //usr/bin/time -o
      /homes/jakeesmith/hw4/3way-mpi/times/times_for_1k/time-${NUM_CORES}-$RANDOM.txt
      /homes/jakeesmith/hw4/3way-mpi/mains_and_outs/mo_1k/mpi-1k

/homes/jakeesmith/hw4/3way-mpi/times/times_for_1k/time-from-code-${NUM_CORES}-$RANDOM.txt

- 10k
  - cores_10k_mpi.sh

```
for i in 1 2 4 8 16
do
    sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q ./mpi_10k_batch.sh
```

  - mpi_10k_batch.sh

```
#!/bin/bash -l
echo "I'm using $NUM_CORES cores"
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_10k/time-${NUM_CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-mpi/mains_and_outs/mo_10k/mpi-10k
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_10k/time-from-code-${NUM_CORES}-$RANDOM.txt
```

- 100k
  - cores_100k_mpi.sh

```
for i in 1 2 4 8 16
do
    sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q ./mpi_100k_batch.sh
Done
```

  - mpi_100k_batch.sh

```
echo "I'm using $NUM_CORES cores"
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_100k/time-${NUM_CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-mpi/mains_and_outs/mo_100k/mpi-100k
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_100k/time-from-code-${NUM_CORES}-$RANDOM.txt
```

- 1mil
  - cores_1mil_mpi.sh

```
for i in 1 2 4 8 16
do
    sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q ./mpi_1mil_batch.sh
Done
```

  - Mpi_1mil_batch.sh

```
echo "I'm using $NUM_CORES cores"
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_1mil/time-${NUM_CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-mpi/mains_and_outs/mo_1mil/mpi-1mil
/homes/jjeilert/proj4/hw4/3way-mpi/times/times_for_1mil/time-from-code-${NUM_CORES}-$RANDOM.txt
```

- **OpenMP Version**
  - 1k
    - cores_1k_openmp.sh

```
for i in 1 2 4 8 16
do
   sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q
./openmp_1k_batch.sh
done
```

    - openmp_1k_batch.sh

```
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_1k/time-${NUM_CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-openmp/mains_and_outs/mo_1k/openmp-1k
/homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_1k/time-from-code-${NUM_CORES}-$RANDOM.txt
```

  - 10k
    - cores_10k_openmp.sh

```
for i in 1 2 4 8 16
do
   sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
--time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
--export=ALL,NUM_CORES=$i --partition=killable.q
./openmp_10k_batch.sh
done
```

    - openmp_10k_batch.sh

```
//usr/bin/time -o
/homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_10k/time-${NUM_CORES}-$RANDOM.txt
/homes/jjeilert/proj4/hw4/3way-openmp/mains_and_outs/mo_10k/openmp-10k
/homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_10k/time-from-code-${NUM_CORES}-$RANDOM.txt
```

- 100k
  - cores_100k_openmp.sh
    ```
    for i in 1 2 4 8 16
    do
       sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
    --time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
    --export=ALL,NUM_CORES=$i --partition=killable.q
    ./openmp_100k_batch.sh
    done
    ```
  - openmp_100k_batch.sh
    ```
    //usr/bin/time -o
    /homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_100k/time-${NU
    M_CORES}-$RANDOM.txt
    /homes/jjeilert/proj4/hw4/3way-openmp/mains_and_outs/mo_100k/openm
    p-100k
    /homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_100k/time-from-c
    ode-${NUM_CORES}-$RANDOM.txt
    ```
- 1mil
  - cores_1mil_openmp.sh
    ```
    for i in 1 2 4 8 16
    do
       sbatch --job-name=my_job --output=slurm-%j.out --constraint=moles
    --time=0:10:00 --mem-per-cpu=1G --nodes=1 --ntasks-per-node=$i
    --export=ALL,NUM_CORES=$i --partition=killable.q
    ./openmp_1mil_batch.sh
    done
    ```
  - openmp_1mil_batch.sh
    ```
    //usr/bin/time -o
    /homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_1mil/time-${NUM
    _CORES}-$RANDOM.txt
    /homes/jjeilert/proj4/hw4/3way-openmp/mains_and_outs/mo_1mil/openm
    p-1mil
    /homes/jjeilert/proj4/hw4/3way-openmp/times/times_for_1mil/time-from-co
    de-${NUM_CORES}-$RANDOM.txt
    ```

A.2 Sample Outputs

```
I'm using 1 cores
0: 125
1: 125
2: 125
3: 125
4: 125
5: 125
```

6: 125
7: 125
8: 125
9: 124
10: 125
11: 125
12: 125
13: 126
14: 125
15: 125
16: 125
17: 125
18: 125
19: 125
20: 125
21: 125
22: 125
23: 125
24: 125
25: 124
26: 125
27: 125
28: 125
29: 125
30: 125
31: 125
32: 125
33: 125
34: 125
35: 125
36: 125
37: 125
38: 125
39: 125
40: 125
41: 125
42: 125
43: 125
44: 125
45: 125
46: 125
47: 125
48: 125
49: 125

50: 125
51: 125
52: 122
53: 125
54: 125
55: 125
56: 125
57: 125
58: 125
59: 125
60: 125
61: 125
62: 125
63: 125
64: 125
65: 125
66: 125
67: 125
68: 125
69: 125
70: 124
71: 125
72: 125
73: 125
74: 125
75: 125
76: 125
77: 125
78: 125
79: 125
80: 125
81: 125
82: 125
83: 125
84: 125
85: 125
86: 125
87: 125
88: 125
89: 125
90: 125
91: 125
92: 125
93: 125

94: 125
95: 125
96: 125
97: 122
98: 125
99: 125