**PDC PROJECT WRITE UP**

**K21 3156, K21 3446, K21 4612**

**Sorting Algorithms:**

1. **MERGE SORT**

1. **Introduction:**

The provided code implements parallel merge sort using OpenMP, a widely used API for parallel programming. Merge sort is a divide-and-conquer algorithm that recursively divides the array into halves, sorts them, and then merges them back together. Parallelizing this algorithm can improve its performance, especially on multicore systems.

2. **Data Generation**:

The dataset is generated using the `rand()` function to fill an array of integers with random values between 0 and 99. The size of the dataset is specified as a command-line argument when running the program.

3. **OpenMP Pragmas:**

The OpenMP pragmas in the code are used to parallelize the merge sort algorithm. Let's break down the relevant parts:

- #pragma omp parallel sections: This pragma creates a parallel region with two sections. Each section represents a task that can be executed concurrently. In this case, the merge sort algorithm is divided into two sections - one for the left half of the array and one for the right half.

- #pragma omp section: This pragma specifies that the following code block is a section of a parallel region. In this code, it is used to parallelize the recursive calls to `MSP` for the left and right halves of the array.

- omp\_set\_num\_threads(num\_threads): This function call sets the number of threads to be used for parallel execution. The number of threads is provided as a command-line argument.

**4. Performance Measurements:**

The program measures the execution time of the parallel merge sort using the `omp\_get\_wtime()` function. The difference between the start and end times gives the total execution time. The results indicate the impact of varying the number of threads on performance.

- With 1 thread and a dataset size of 20,000,000, the execution time is 30.26 seconds.

- With 200 threads and the same dataset size, the execution time decreases to 21.22 seconds.

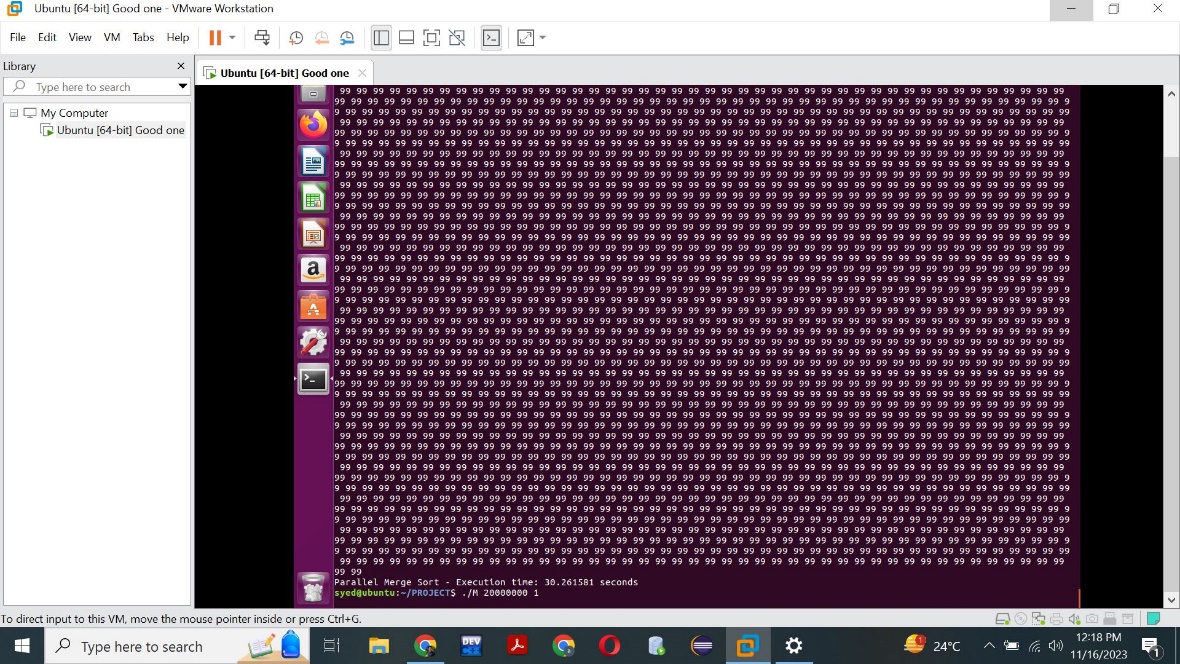
**5. Performance Analysis:**

The performance improvement observed with an increased number of threads is due to the parallelization of the merge sort algorithm. Each section of the array can be sorted independently by different threads, reducing the overall execution time.

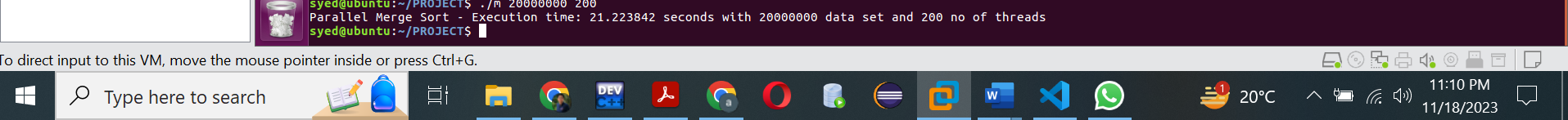
**6. Conclusion:**

This parallel merge sort implementation demonstrates the benefits of utilizing multiple threads to enhance the performance of sorting large datasets. It leverages OpenMP pragmas to distribute the workload among threads, resulting in a more efficient execution compared to a single-threaded approach.

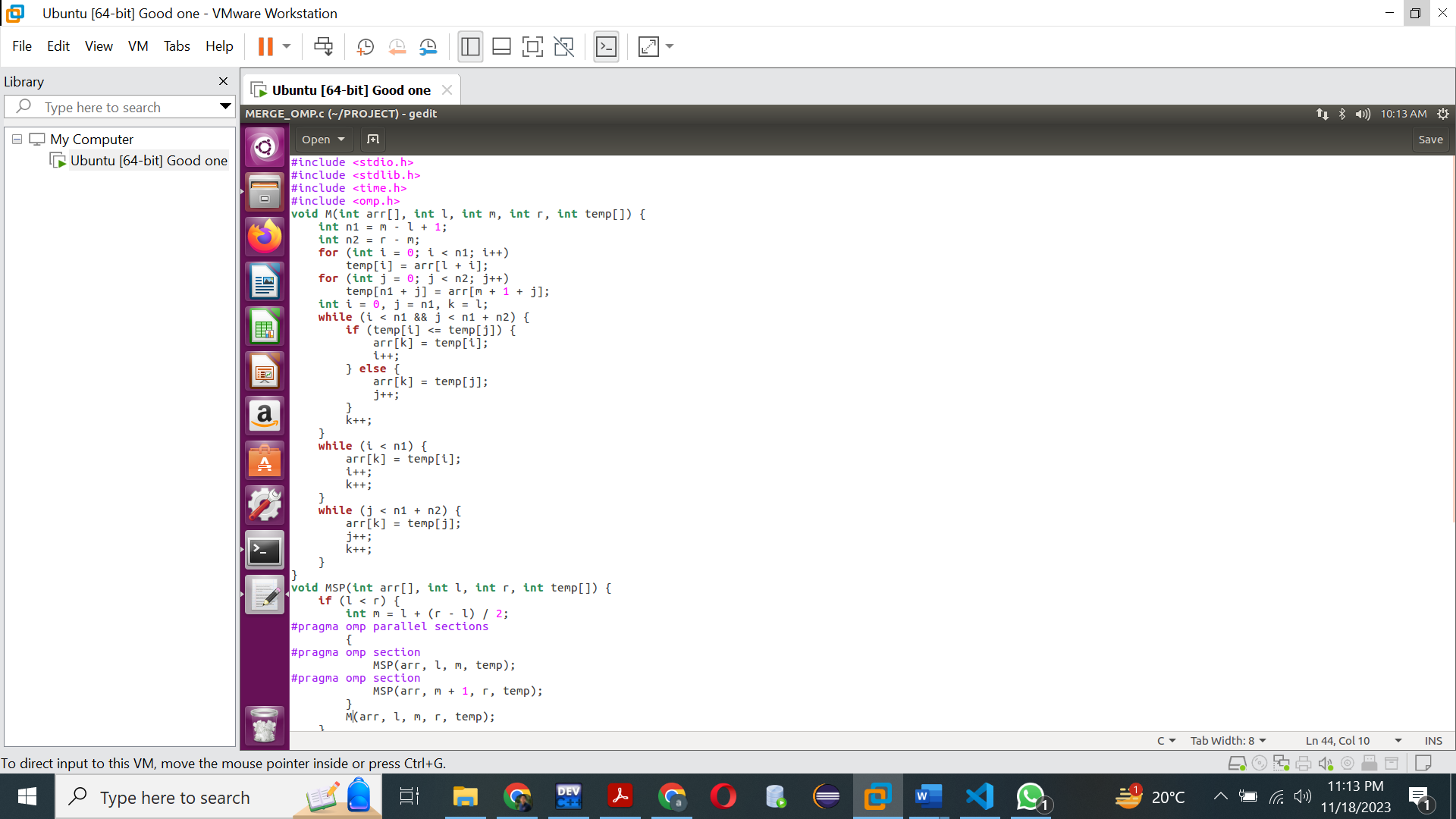
20000000 1 thread



20000000 200 threads



Code:



1. **QUICK SORT:**

1. **Introduction:**

The provided code implements parallel quicksort using OpenMP. Quicksort is another popular sorting algorithm known for its efficiency. In this implementation, the code divides the array into partitions using a pivot element and recursively sorts these partitions.

2. **OpenMP Pragmas:**

The OpenMP pragmas in the code are used to parallelize the quicksort algorithm. Let's analyze the relevant parts:

- `#pragma omp parallel`: This pragma creates a parallel region where multiple threads can execute. Each thread will contribute to the parallel execution of the following code block.

- `#pragma omp single`: This pragma specifies that the enclosed code block should be executed by a single thread. In this code, it ensures that the initial call to `QSP` is only executed once.

-`#pragma omp task`: This pragma identifies a task that can be executed independently. In this case, it is used to parallelize the recursive calls to `QSP` for the left and right partitions of the array.

3. **Performance Measurements:**

The program measures the execution time of the parallel quicksort using the `omp\_get\_wtime()` function. The difference between the start and end times gives the total execution time. The results indicate the impact of varying the number of threads on performance.

- With 1 thread and a dataset size of 1,000,000, the execution time is 2416 seconds.

- With 100 threads and the same dataset size, the execution time decreases to 2346 seconds.

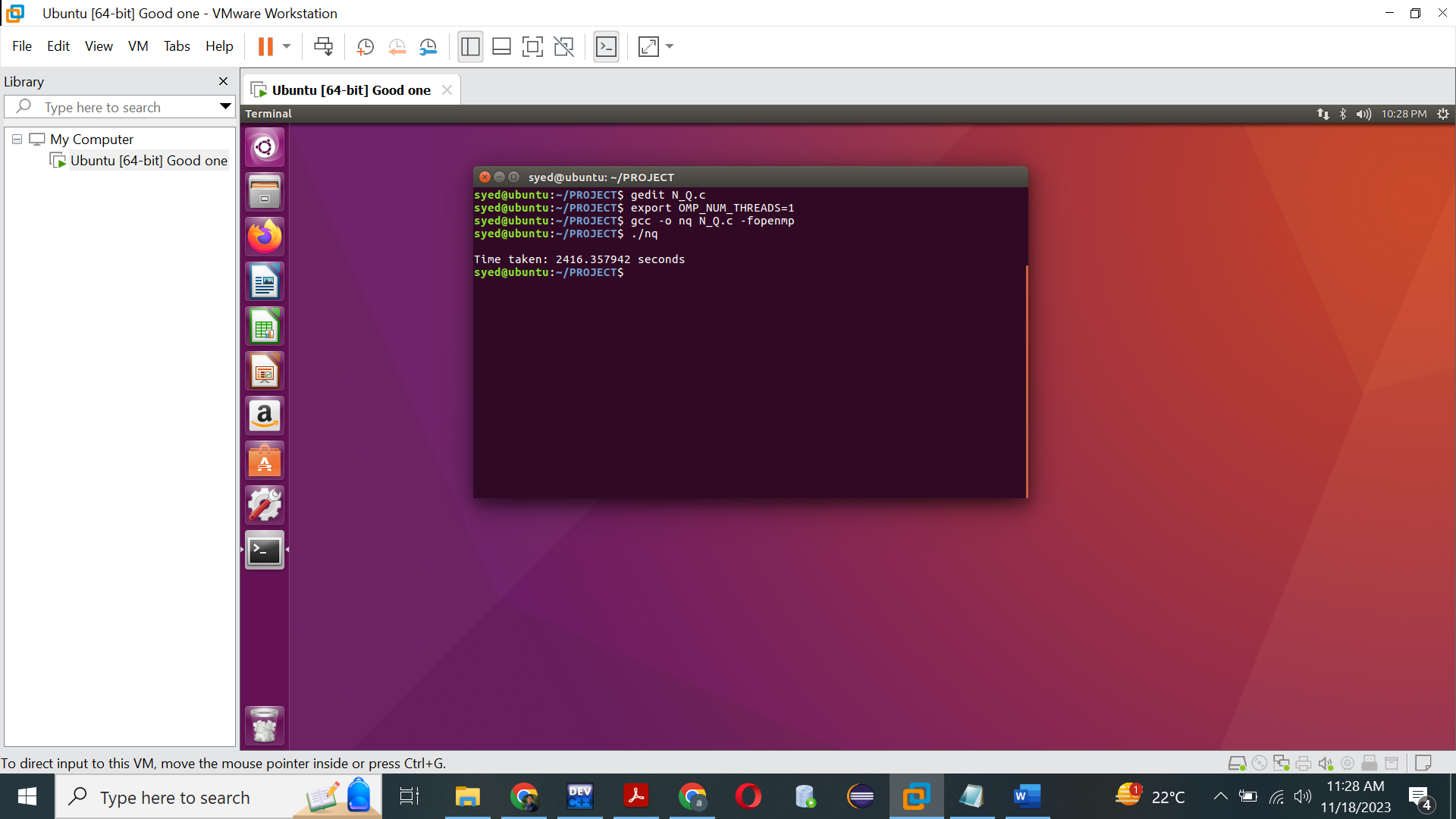
4. **Performance Analysis:**

The performance improvement observed with an increased number of threads is attributed to the parallelization of the quicksort algorithm. Each thread works on a different partition of the array, enhancing the overall sorting process.

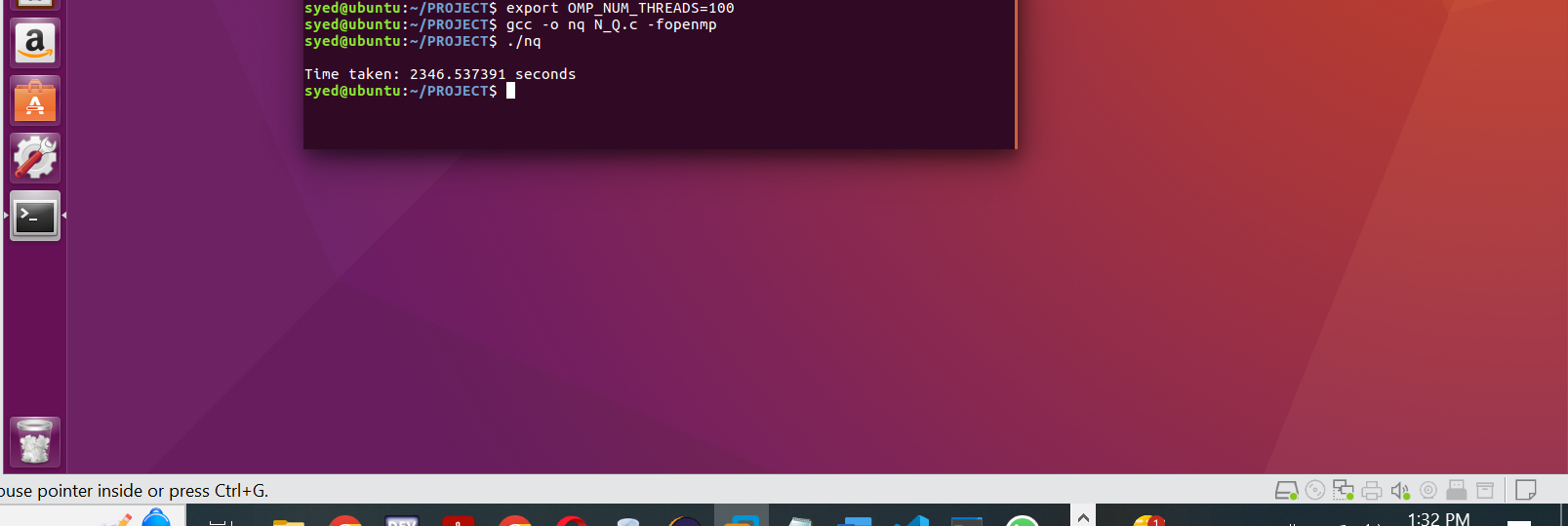
5. **Conclusion:**

This parallel quicksort implementation demonstrates the advantages of leveraging multiple threads to improve the efficiency of sorting large datasets. The use of OpenMP pragmas allows for the parallel execution of tasks, leading to a reduction in execution time compared to a single-threaded approach.

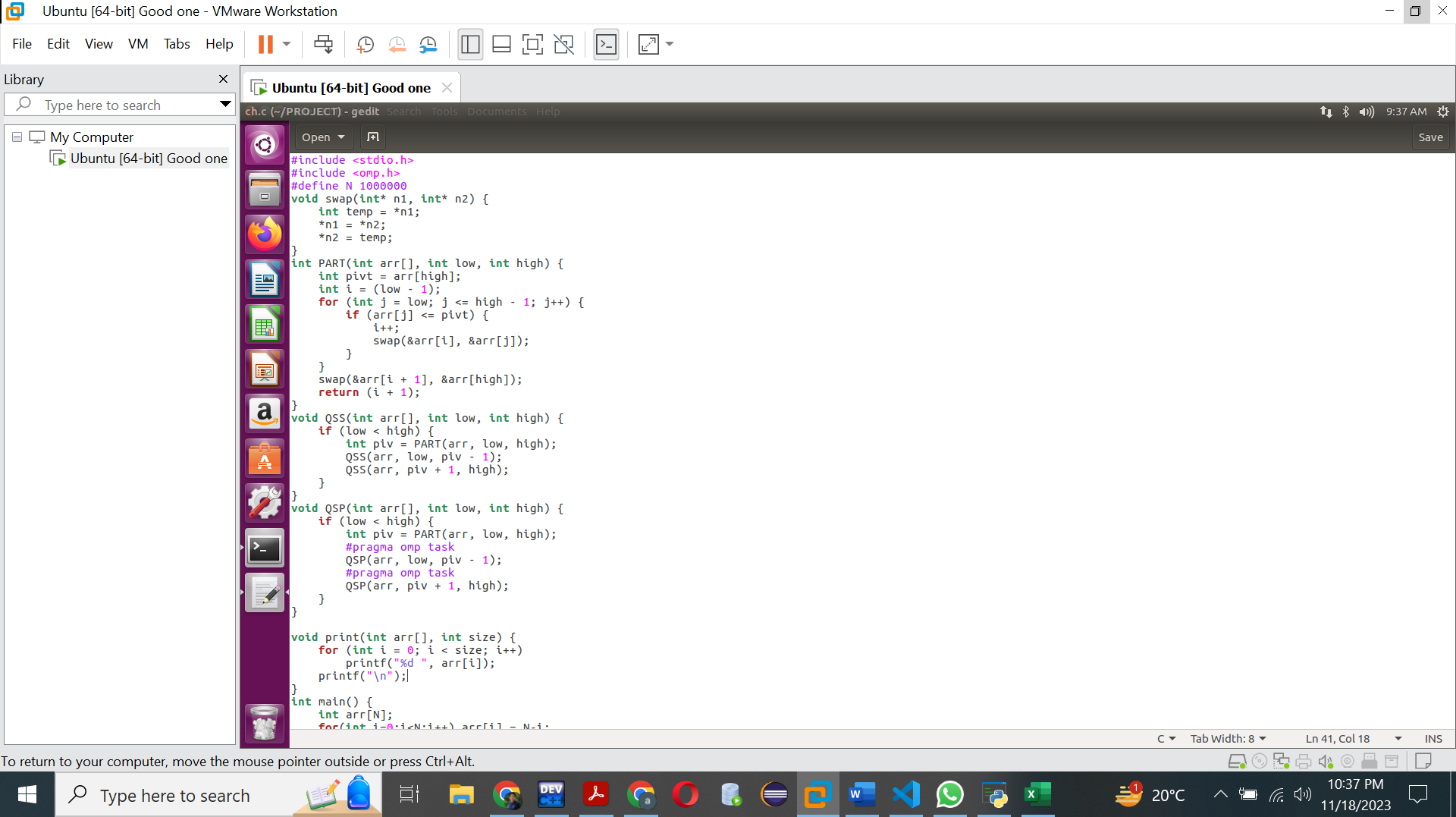
1000,000 Dataset with 1 thread:



1000,000 Dataset with 100 threads:



Code:



1. **Bubble sort:**

1. **Introduction:**

The provided code implements parallel bubble sort using OpenMP. Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. Parallelizing bubble sort involves distributing the work of comparing and swapping elements among multiple threads.

2. **OpenMP Pragmas:**

The OpenMP pragmas in the code are used to parallelize the bubble sort algorithm. Let's analyze the relevant parts:

- `#pragma omp parallel`: This pragma creates a parallel region where multiple threads can execute. Each thread will contribute to the parallel execution of the following code block.

- `#pragma omp for`: This pragma is used to distribute the outer loop among threads. Each thread will handle a different range of iterations, reducing the overall execution time.

3. **Performance Measurements:**

The program measures the execution time of the parallel bubble sort using the `omp\_get\_wtime()` function. The difference between the start and end times gives the total execution time. The results indicate the impact of varying the number of threads on performance.

- With 1 thread and a dataset size of 100,000, the execution time is 35.38 seconds.

- With 200 threads and the same dataset size, the execution time increases to 83 seconds.

- With 1 thread and a dataset size of 200,000, the execution time is 132.3 seconds.

- With 200 threads and the same dataset size, the execution time further increases to 230 seconds.

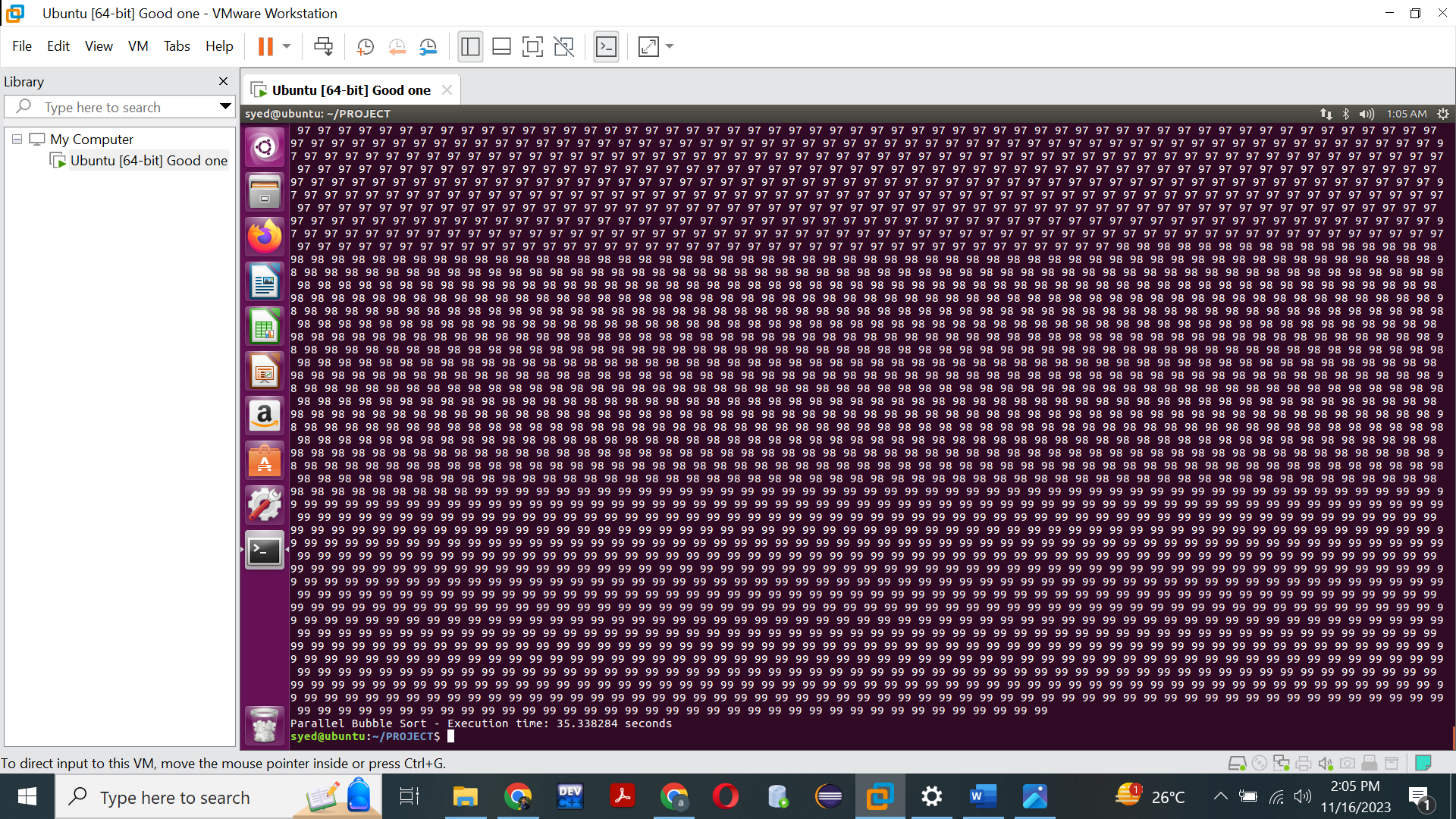
4. **Performance Analysis:**

The performance **degradation** observed with an increased number of threads can be explained by the fact that bubble sort has inherent sequential dependencies. While multiple threads can work on different parts of the array, they are still limited by the sequential nature of the algorithm.

5. **Conclusion:**

Parallelizing bubble sort using OpenMP may not provide significant performance improvements due to its inherent sequential nature. In some cases, as observed in the results, the overhead introduced by parallelization can even lead to increased execution times. For larger datasets and higher thread counts, alternative parallel sorting algorithms might be more suitable for achieving better performance.

Dataset: 100,000 1 thread



Dataset: 100,000 200 threads

A screenshot of a computer

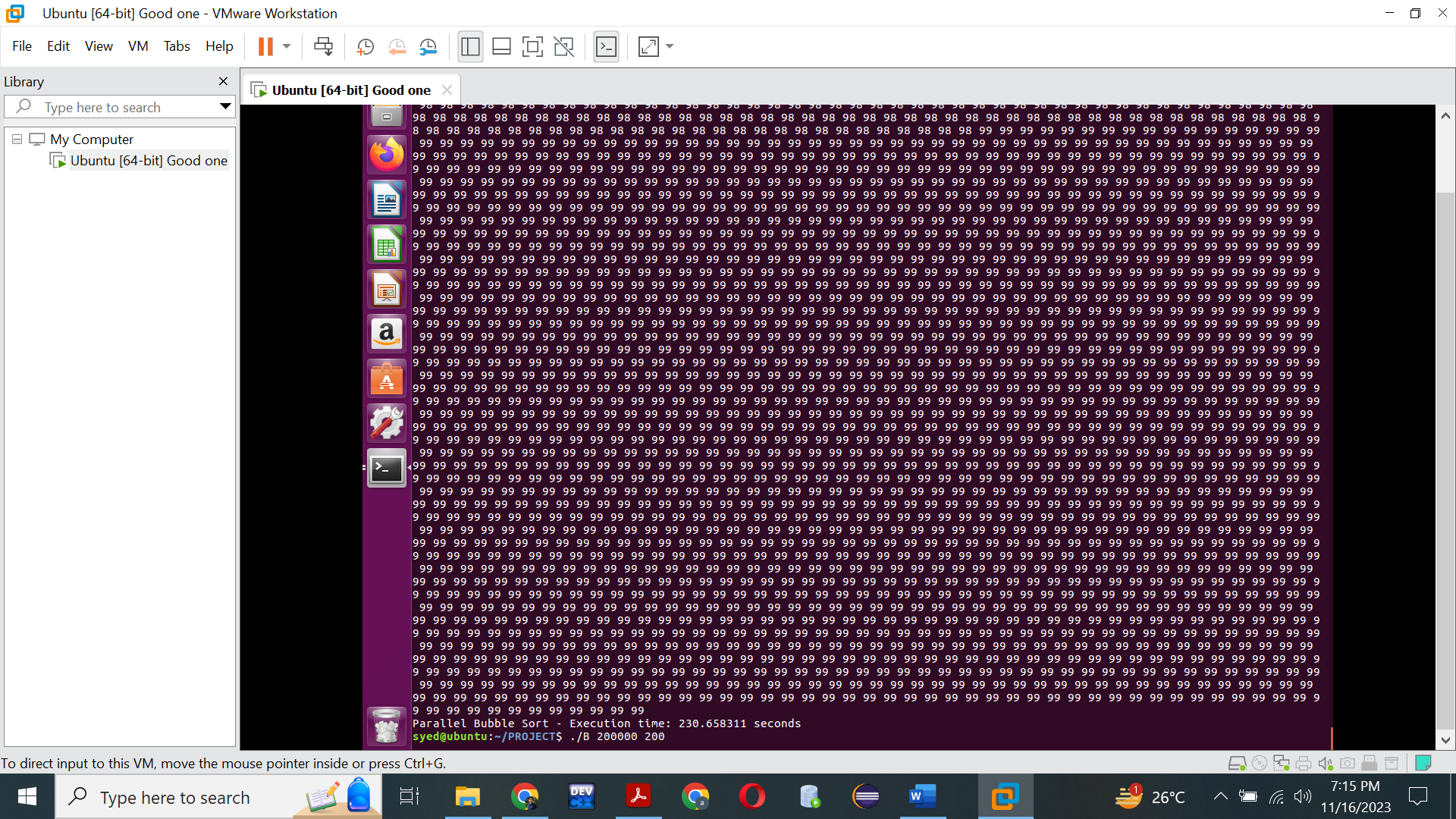
Description automatically generated

Dataset: 200,000 1 thread

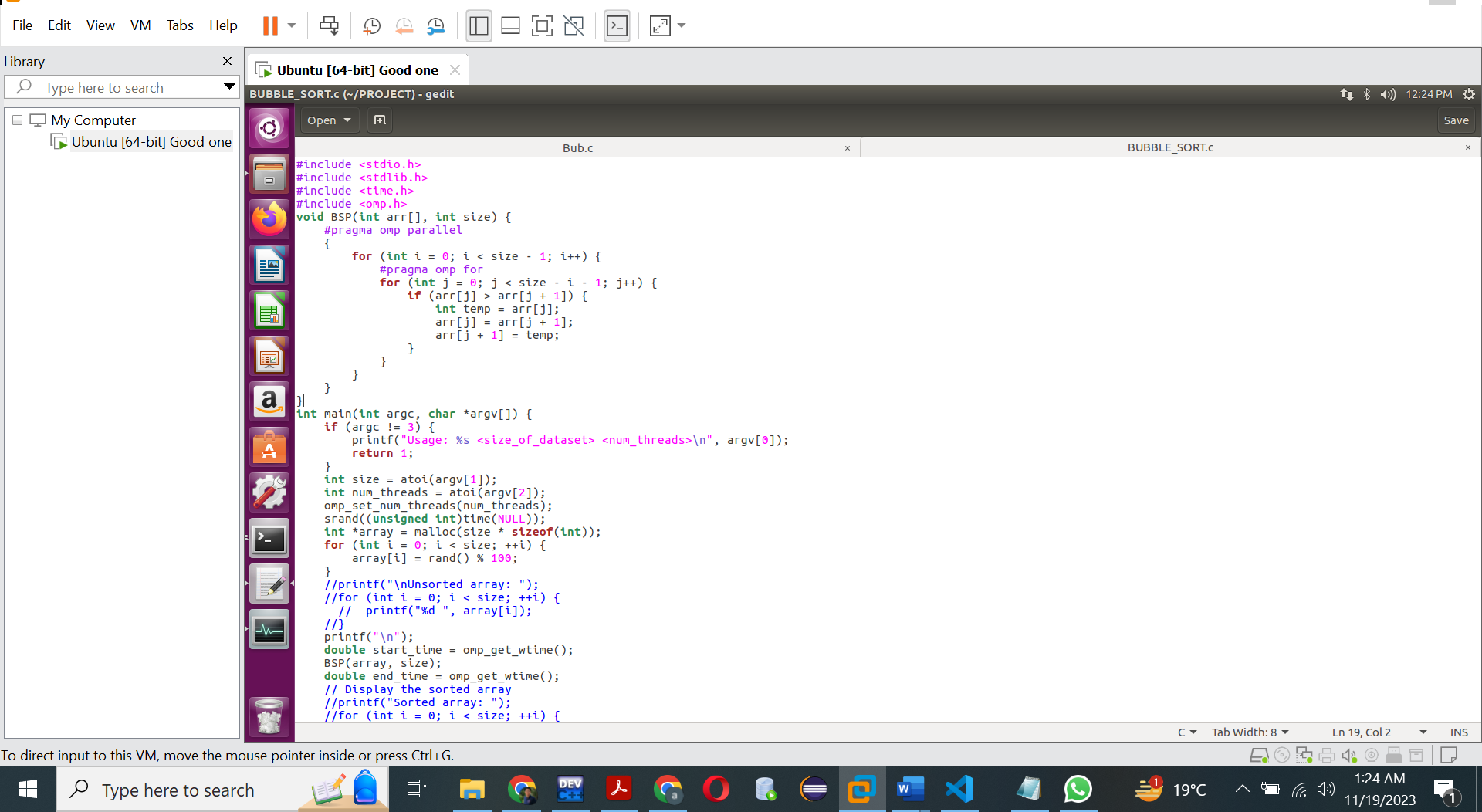
A screenshot of a computer

Description automatically generated

Dataset: 200,000 200 threads



Code:



1. **Selection Sort:**

1. **Introduction:**

The provided code implements parallel selection sort using OpenMP. Selection sort is a simple sorting algorithm that repeatedly selects the smallest (or largest, depending on sorting order) element from the unsorted part of the array and swaps it with the first unsorted element. Parallelizing selection sort involves distributing the work of finding the minimum element and swapping it among multiple threads.

2. **OpenMP Pragmas:**

The OpenMP pragmas in the code are used to parallelize the selection sort algorithm. Let's analyze the relevant parts:

#pragma omp parallel: This pragma creates a parallel region where multiple threads can execute. Each thread will contribute to the parallel execution of the following code block.

#pragma omp for: This pragma is used to distribute the outer loop among threads. Each thread will handle a different range of iterations, reducing the overall execution time.

#pragma omp single: This pragma ensures that the swap operation is executed by a single thread to avoid data races. The single thread is determined by the OpenMP runtime system.

3. **Performance Measurements:**

The program measures the execution time of the parallel selection sort using the omp\_get\_wtime() function. The difference between the start and end times gives the total execution time. The results indicate the impact of varying the number of threads on performance.

With 1 thread and a dataset size of 100,000, the execution time is 10.17 seconds.

With 200 threads and the same dataset size, the execution time increases to 109.651 seconds.

With 1 thread and a dataset size of 1,000,000, the execution time is 1393 seconds.

With 100 threads and the same dataset size, the execution time further increases to 1587 seconds.

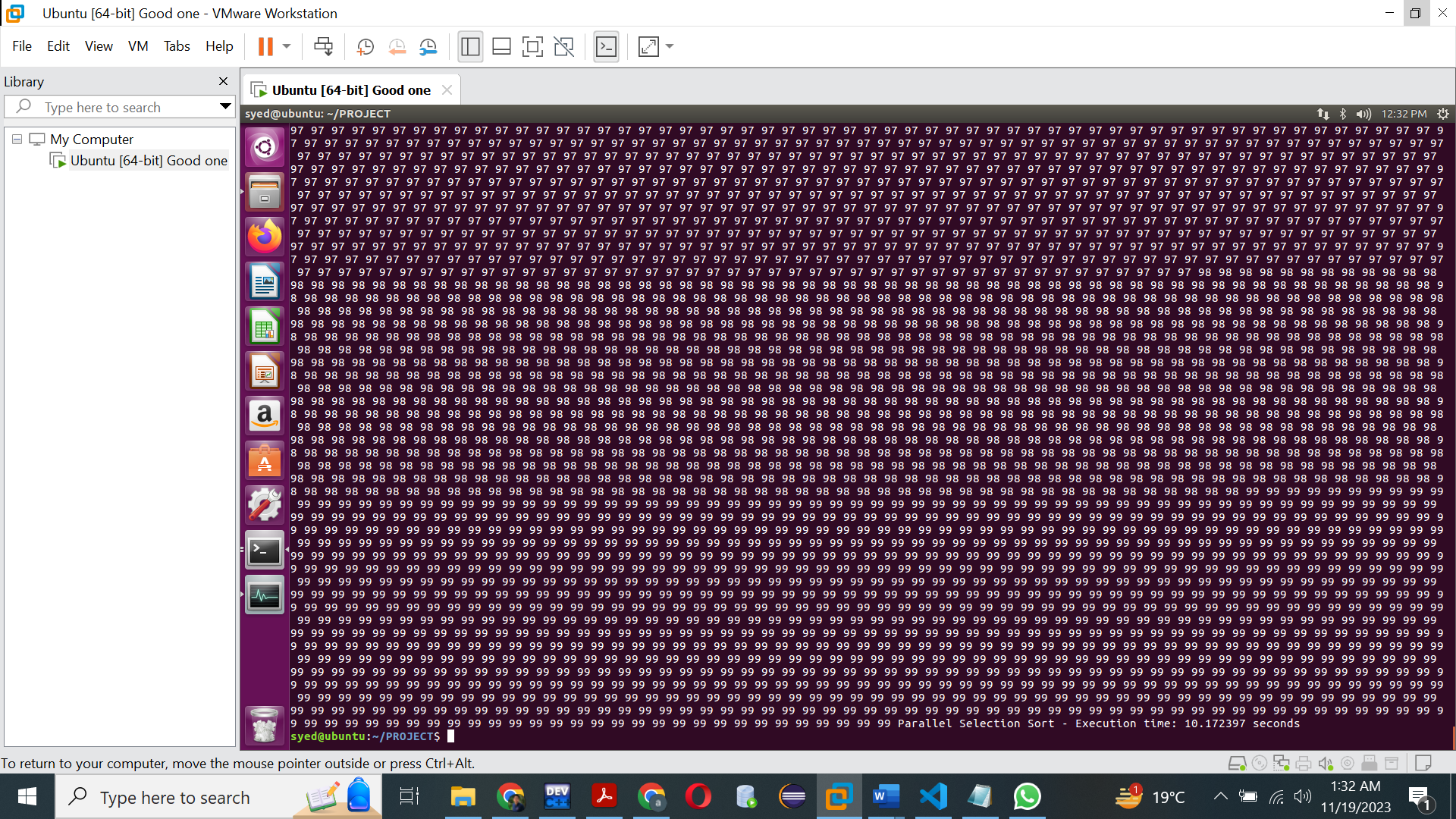
4. **Performance Analysis:**

The performance **degradation** observed with an increased number of threads can be attributed to the fact that selection sort has inherent sequential dependencies. The algorithm involves finding the minimum element and swapping it sequentially, limiting the effectiveness of parallelization.

5. **Conclusion:**

Parallelizing selection sort using OpenMP may not provide significant performance improvements due to its inherent sequential nature and the need for a single thread to perform the swapping operation. For larger datasets and higher thread counts, alternative parallel sorting algorithms might be more suitable for achieving better performance. The results suggest that the overhead introduced by parallelization can outweigh the benefits for this specific sorting algorithm.

Dataset: 100,000 1 thread:

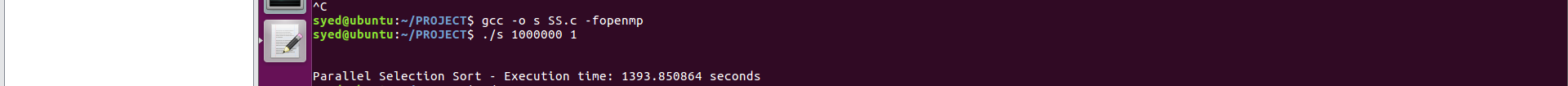


Dataset: 100,000 and 100 threads:

A screenshot of a computer screen

Description automatically generated

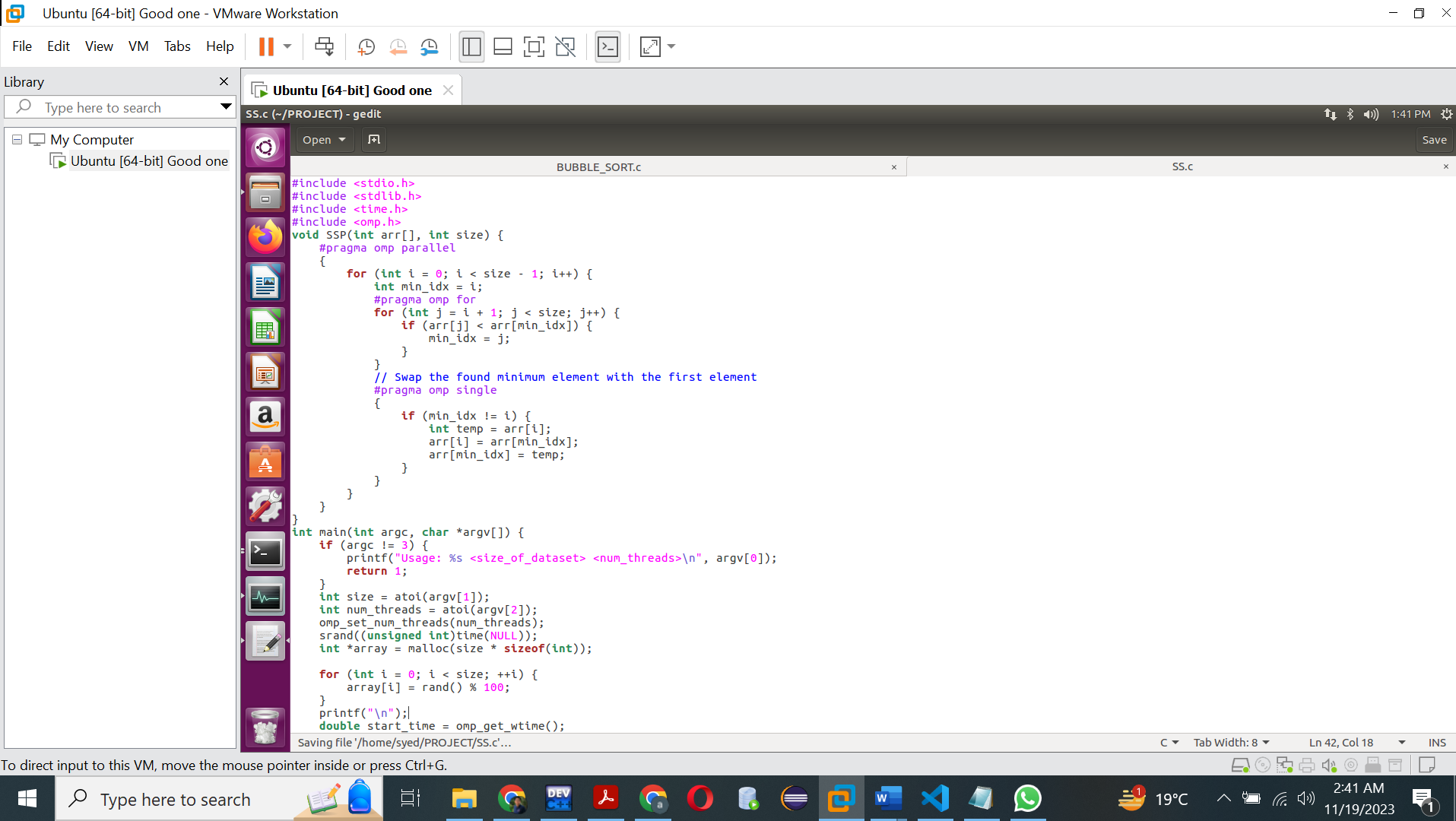
Dataset: 1000,000 1 thread:



Dataset: 1000,000 100 threads



Code:



**CONVERSION OF MERGE SORT INTO MPI:**

1. **Introduction**:

The provided code implements parallel merge sort using MPI (Message Passing Interface). MPI is a standard for writing parallel programs that can be executed on distributed-memory systems. In this case, the merge sort algorithm is parallelized across multiple processes to improve its performance.

2. **MPI Data Distribution:**

Unlike OpenMP, MPI programs involve multiple processes running independently and communicating through message passing. In this MPI merge sort implementation:

- Data Distribution (`DD` function): The `DD` function is responsible for distributing the dataset among different processes. It calculates the local size (`ls`), allocates memory for the local data (`ld`), and uses `MPI\_Scatter` to distribute portions of the dataset to each process.

- Data Gathering (`GD` function): After the local sorting is done, the `GD` function gathers the sorted local data from all processes back to the root process using `MPI\_Gather`.

3. **Performance Measurements:**

The program measures the execution time of the parallel MPI merge sort using the `MPI\_Wtime()` function. The difference between the start and end times gives the total execution time. The results indicate the impact of varying the number of processes on performance.

- With 1 process and a dataset size of 20,000,000, the execution time is 5.829387 seconds.

- With 4 processes and the same dataset size, the execution time decreases to 3.57 seconds.

4. **Performance Analysis:**

The performance **improvement** observed with an increased number of processes is due to the parallelization of the merge sort algorithm across multiple nodes. Each process independently sorts its portion of the dataset, and then the sorted data is merged in a distributed manner.

5. **Comparison with OpenMP:**

- MPI Merge Sort (20,000,000, 4 processes): 3.57 seconds.

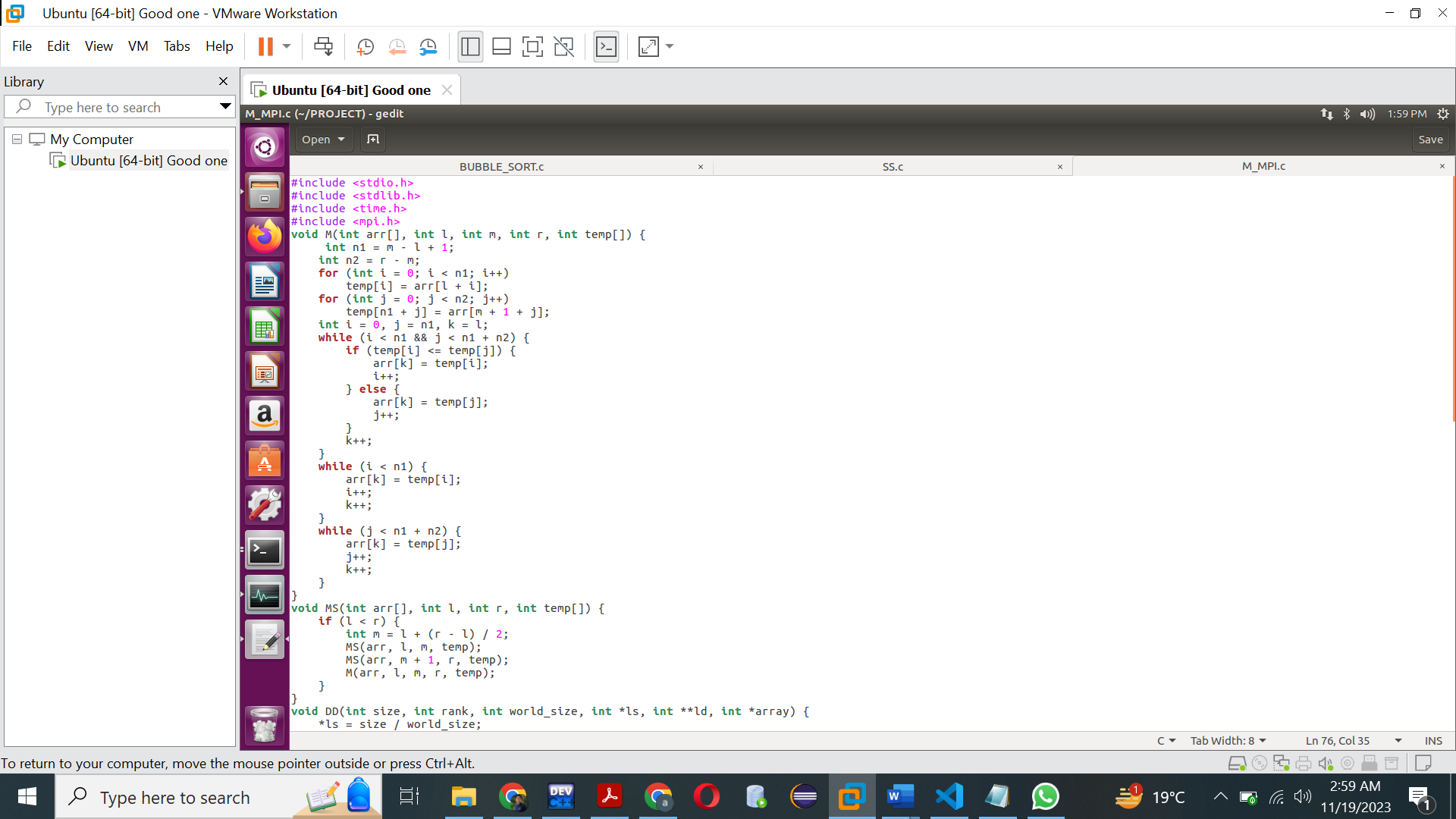
- OpenMP Merge Sort (20,000,000, 200 threads): 21.22 seconds.

MPI demonstrates better performance compared to OpenMP for this specific dataset size and parallelization level. This is likely due to the ability of MPI to efficiently scale across distributed memory systems.

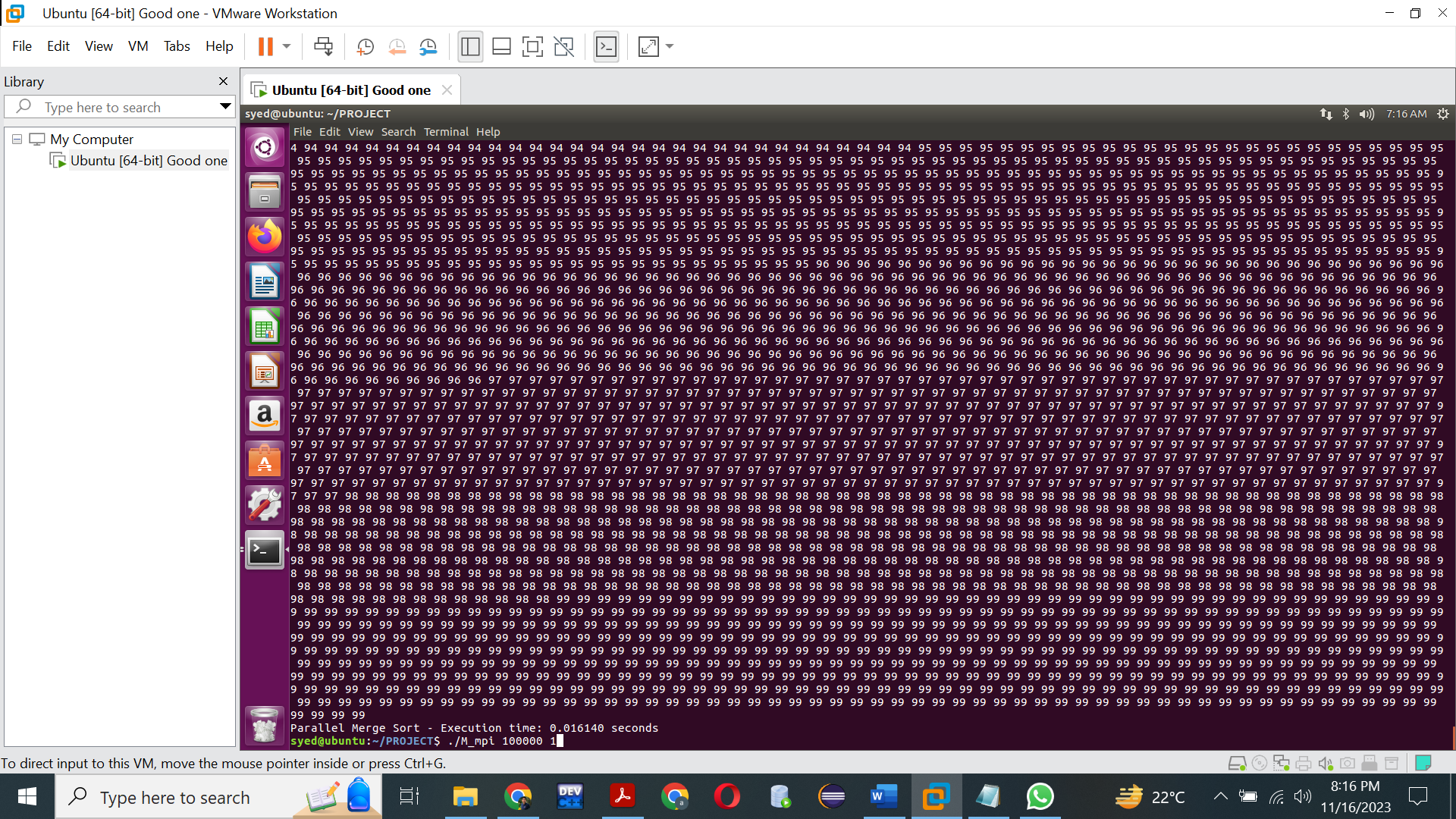
6. **Conclusion:**

MPI provides a powerful mechanism for parallelizing algorithms across multiple processes, making it suitable for distributed-memory systems. In the case of merge sort, MPI allows for efficient parallelization and distribution of data among processes, leading to improved performance compared to a single-process execution. The scalability of MPI across multiple nodes makes it a valuable tool for large-scale parallel computing.

Code:



100,000 1 Process:

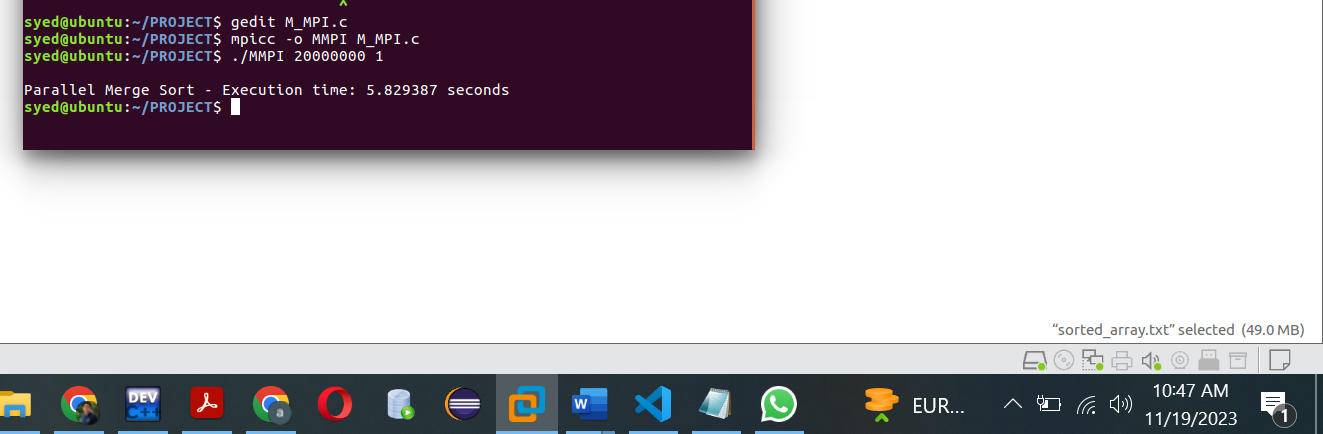


100,000 2 processes:

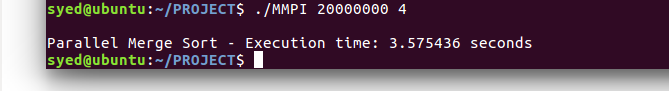
A screenshot of a computer

Description automatically generated

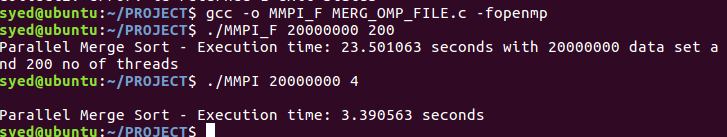
20000000 1 process:



20000000 4 processes:



DIFFERENCE:



With Cmp Command:



There is a difference between the two files (MPI, OMP) starting at byte 6367.

With Diff Command:

A screenshot of a computer screen

Description automatically generated

**Conclusion:**

So basically, sorting algorithms are not always trivial to parallelize, and some algorithms may introduce subtle differences in the order of elements when parallelized. Most parallel sorting algorithms aim to preserve the order of equal elements (stability), but there might be variations in the output due to the parallel nature of the computation.

Also, our dataset involves random numbers, the outcome could be different each time we generate a new set of random numbers. And lastly, the way parallelism is implemented, the order in which parallel tasks are executed, and other implementation-specific details can potentially lead to small variations so, subtle differences may arise due to the challenges of parallel computing.