CS 566: Parallel Processing
Homework 3 - Design Document

Zohair Hashmi – UIN 668913771

Part (a)

Parallel Program (gauss-mpi.c) is attached in the final submission.

Static interleaved scheduling is a parallelization technique commonly used in distributed memory systems to distribute computational workloads evenly among multiple processing elements. It involves partitioning the data or tasks into equal-sized chunks and assigning each chunk to a different processing element for concurrent execution. This approach ensures load balance and minimizes communication overhead by statically assigning tasks to processing elements before the computation begins.

The provided code utilizes static interleaved scheduling for parallel execution, ensuring efficient utilization of resources and balanced workload distribution among processing elements. Key aspects include:

Data Distribution: Matrix A and vector B are partitioned into equal-sized chunks, with each process receiving a distinct portion. This facilitates an equitable distribution of computational tasks.

Parallel Execution: Following data distribution, each process executes the Gaussian elimination algorithm concurrently with others. This simultaneous operation across processes optimizes resource utilization and expedites computation.

Minimal Communication Overhead: Static interleaved scheduling minimizes communication overhead by predetermining task assignments. Communication primarily occurs during essential operations like broadcasting normalization rows and gathering updated data, keeping overhead to a minimum.

Workload Balance: Through static interleaved scheduling, the workload is evenly spread across processes, ensuring each process handles a comparable amount of computation. This promotes load balance and prevents resource underutilization or overload.

Part (b)

In this experiment, we modify and execute the gaussian elimination code using Message Passing Interface (MPI) and compare the performance with the sequential code with a matrix size of N = 5000 and varying number of MPI processes.

The following sections describe the tests that were conducted and their respective observed results. We discuss the difference in the results for varying parameters and the optimization and enhancements made over time.

MPI Parallel Processing Steps:

Libraries included: #include <mpi.h>

Modified gauss function:

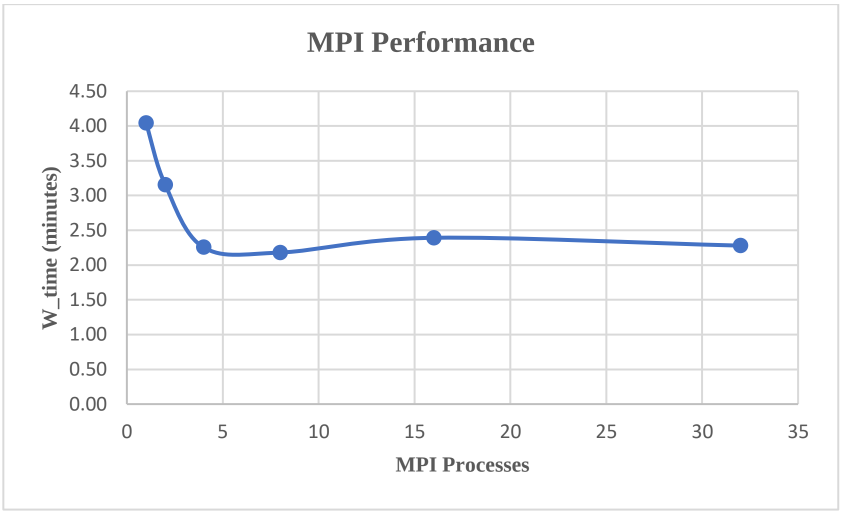
- The **total number of processes** engaged in the computation is acquired using *MPI_Comm_size(MPI_COMM_WORLD, &numtasks)*. This command retrieves the number of available processes within the MPI communicator.
- The calculation of **rows per process** is based on the total number of rows (N) and the specified number of processes during the execution of the compiled executable file.
- The **rank** of the current process is identified *utilizing MPI_Comm_rank(MPI_COMM_WORLD, &rank)*. This command assigns a unique identifier to each process within the MPI communicator.
- Memory is dynamically allocated to generate data chunks destined for each process, facilitating parallel computation using MPI.
- Matrix A and vector B are disseminated across all processes employing *MPI_Scatter*. This function accepts parameters including the matrix, the count of rows per process, the data type of matrix elements, chunks allocated for each process, the source of scattered data, and the communicator used for the scatter operation.

- Each process undertakes Gaussian elimination on its designated rows. The normalization row is disseminated to all processes via *MPI_Bcast*. Subsequently, each process conducts operations on its assigned rows while excluding the normalization row.
- Post each iteration of Gaussian elimination, the modified *local_A* and *local_B* arrays are aggregated back to the root process utilizing *MPI_Gather*.
- Following Gaussian elimination, back substitution is executed sequentially across all processes.
- The duration of the computation is gauged employing *MPI_Wtime()* prior to initiation and subsequent to completion of the computational task.

Experimentation Results:

The following table and graph document the impact of exponentially increasing the number of processes on *W_time*. It defines *W_time* as the duration from initiating parallel processes to aggregating results in the primary thread (0th thread).

Processes	W_time (seconds)	Wtime (mins)
1	242.44	4.04
2	189.35	3.16
4	135.29	2.25
8	130.80	2.18
16	143.52	2.39
32	136.77	2.28



Performance Evaluation:

As the number of processes increases from 1 to 8, there is a notable decrease in the execution time. With a single process (1), the execution time is the highest at 242.44 seconds, indicating sequential execution. With the introduction of parallelism via MPI and increasing the number of processes, the workload is distributed among multiple processors, leading to reduced execution times. The execution time continues to decrease as the number of processes increases up to 8, indicating effective parallelization.

The decrease in execution time is not entirely linear with the increase in the number of processes. After 8 processes, there is a slight increase in execution time when using 16 and 32 processes compared to 8 processes. This increase could be attributed to the overhead incurred due to communication between a larger number of processes.

The most significant reduction in execution time occurs when moving from 1 to 2 processes, indicating that even a small degree of parallelization can yield notable benefits. Subsequent increases in the number of processes lead to diminishing returns in terms of reducing execution time.

Conclusion:

The provided code demonstrates reasonably efficient parallelization through MPI for Gaussian elimination, as indicated by the decreasing execution times with an increasing number of processes.

The code achieves proper load balancing as it evenly distributes the workload among processes by dividing the data into equal-sized chunks using static interleaved scheduling. This ensures that each process receives a fair share of the computational tasks, promoting load balance. There is minimal communication overhead by carefully distributing data into chunks and gathering to a primary process using MPI_Scatter and MPI_Gather.

The Gaussian elimination algorithm is parallelized effectively, with each process executing its portion of the computation concurrently. By leveraging parallelism, the code harnesses the computational power of multiple processes to expedite the solution process. The code exhibits scalability, as evidenced by the decreasing execution times with an increasing number of processes. This indicates that the parallelization scheme is capable of effectively utilizing additional resources to improve performance.

In summary, the code demonstrates efficient parallelization through MPI, characterized by proper load balancing, minimal communication overhead, optimized parallel execution, and scalability. These design choices and methodologies contribute to the code's effectiveness in leveraging distributed memory systems for parallel computing tasks.