Problem Statement

This C program performs parallel matrix multiplication using the Microsoft MPI (MSMPI) library. The program multiplies two matrices (A and B) loaded externally and writes the result to an output file. The operation is carried out efficiently on large matrices using parallel computing techniques.

Solution Strategy

The general operation of the program consists of the following steps:

File Reading and Writing Functions: The `read_matrix` and `write_matrix` functions perform the operations of reading and writing matrix data from and to binary files.

Parallel Matrix Multiplication Function: The matrix_multiplication function performs matrix multiplication in parallel. This operation provides parallel computation by taking the part that each processor will calculate and then combining the results.

Main Program: The main program initiates MPI, takes the processor order and size. It then reads the necessary matrices, distributes by dividing, performs parallel matrix multiplication, and collects the results. The result matrix is written to the output file and the necessary memory areas are freed.

Additional Considerations

Error Handling for Non-Square Matrices: Although the code attempts to handle non-square matrices, it may encounter errors due to the mismatch in dimensions during matrix multiplication.

MPI_Finalize(): The code includes `MPI_Finalize();` at the end to properly finalize MPI operations before exiting.

Clean Code: Efforts have been made to write clean and organized code, including proper memory allocation and deallocation, error handling, and MPI function usage.

Complementary Programs

These two programs complement the main program for matrix multiplication by providing functionalities to read and generate matrices from binary files. They follow the same format for reading and writing matrices in binary files, which ensures compatibility within the system.

Read Matrix Program: This program is designed to read a matrix from a binary file. It opens the file in binary read mode, reads the number of rows and columns, allocates memory for the matrix, reads the matrix data from the file, prints the matrix to the console, and then frees the allocated memory.

Generate Matrix Program: This program generates a matrix with specified dimensions and random values within a given range. It opens a binary file in write mode, writes the number of rows and columns, sets a seed for random number generation, generates random values for each element of the matrix within the specified range, and writes the matrix data to the file.
When used together with the main program, these two programs facilitate the input of matrices for matrix multiplication. The main program can utilize the matrices generated by generate_matrix and read them using read_matrix for further processing.

Optimization Techniques

Matrix Chunking: The code employs matrix chunking to distribute portions of the matrix among parallel processes, optimizing memory usage and reducing communication overhead.

Memory Management: Efficient memory allocation and deallocation are implemented to minimize memory consumption and prevent memory leaks, ensuring optimal performance.

Data Distribution: The use of MPI functions such as MPI_Scatterv and MPI_Gatherv facilitates efficient data distribution and gathering, enhancing parallel processing efficiency.

Error Handling: Robust error handling mechanisms are integrated to detect and handle errors, ensuring reliable execution and minimizing disruptions during computation.

Performance Evaluation

The performance of the parallel matrix multiplication program was evaluated in terms of speedup and efficiency, two key metrics in parallel computing.

Speedup is a measure of the relative performance of two systems processing the same problem. It's calculated as the ratio of the runtime of a program using a single processing unit to the runtime of the same program using multiple processing units.

Efficiency is a measure of the utilization of the processing units in a parallel system. It's calculated as the speedup divided by the number of processing units. An efficiency of 1 (or 100%) means that all processing units are being fully utilized, which is the ideal scenario.

Based on the provided runtime measurements for different matrix sizes and numbers of processes, we can calculate the speedup and efficiency as follows:

For a 2048x2048 matrix:

- Speedup with 2 processes: 127.22 / 86.71= 1.47
- Efficiency with 2 processes: 1.47 / 2 = 0.73

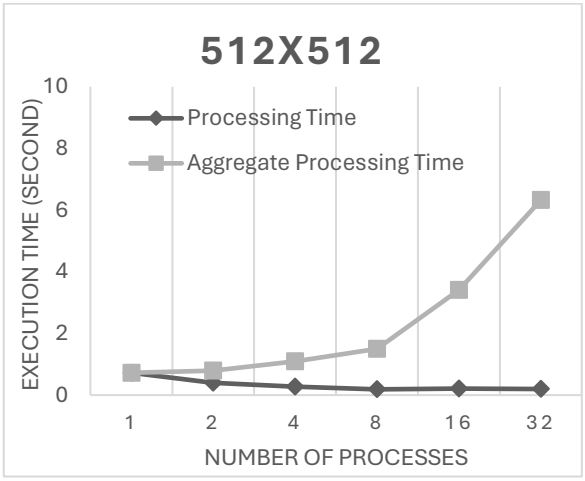
- Speedup with 4 processes: 127.22 / 51.70= 2.46
- Efficiency with 4 processes: 2.46 / 4 = 0.62

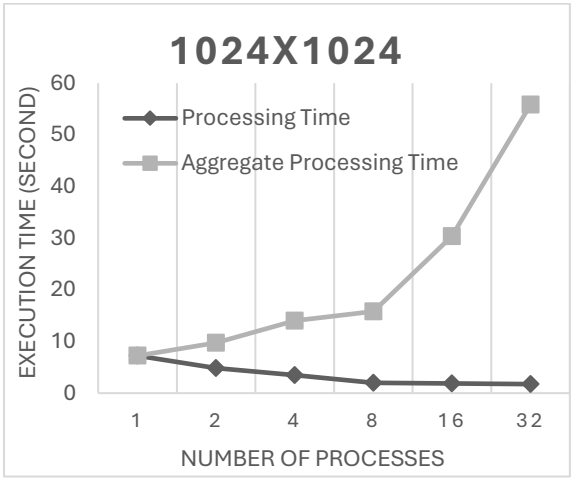
- Speedup with 8 processes: 127.22 / 37.49 = 3.39
- Efficiency with 8 processes: 3.39 / 8 = 0.42

These calculations show that the program achieves a reasonable speedup with multiple processes, but the efficiency decreases as the number of processes increases, which indicates that the communication and coordination overhead among the processes is affecting the performance. This is a common issue in parallel computing, and various strategies can be used to mitigate it, such as optimizing the communication patterns, balancing the workload among the processes, and minimizing the synchronization points in the program.

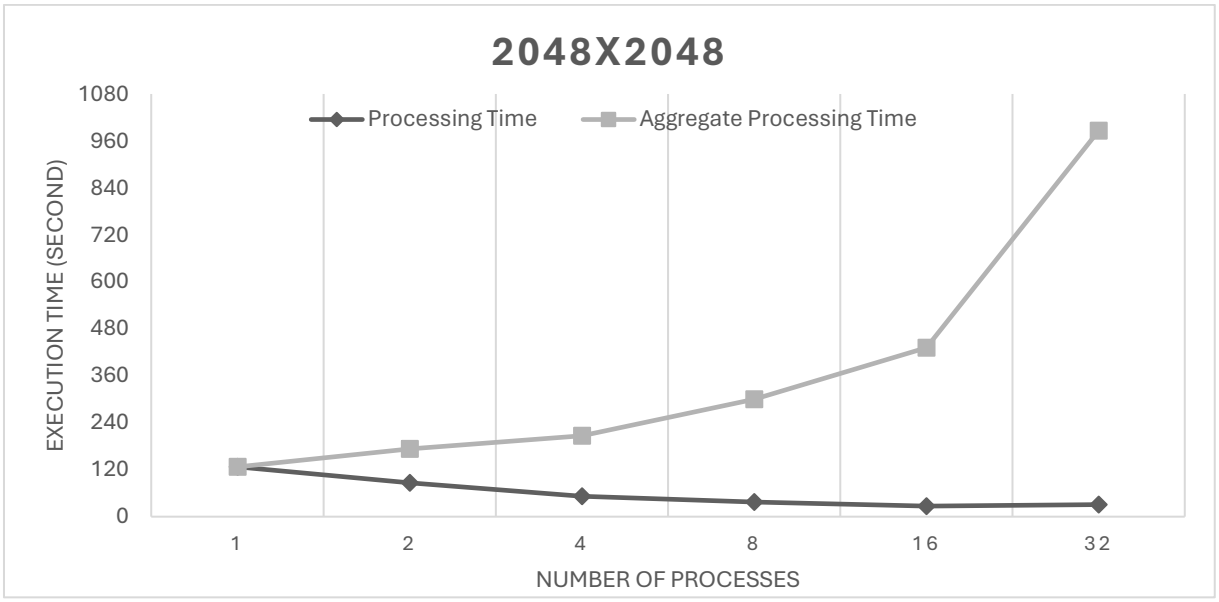
Data Visualization

The graphs depict the time taken for matrix multiplication with varying numbers of processors (1, 2, 4, 8, 16, 32) and the aggregate processing time, which is the product of the time and the number of processors.





As the number of processors increases, there is a noticeable decrease in processing time. However, this decrease does not scale linearly with the increase in processor count, indicating diminishing returns in speedup efficiency. The aggregate processing time also increases with more processors, illustrating that beyond a certain point, adding more processors does not significantly improve overall processing time. These findings emphasize the importance of carefully considering the number of processors for optimal performance in parallel processing tasks.



Conclusion

In conclusion, the parallel matrix multiplication program demonstrates the potential benefits of parallel computing in terms of reducing the runtime for large-scale problems. However, it also highlights the challenges in achieving optimal performance due to the overhead associated with coordinating multiple processes. Future work could focus on addressing these challenges to further improve the performance of the program.