CS/ECE 566 "Parallel Processing"

Two-dimensional Convolution using Fast Fourier Transform and MPI

Isaac Salvador
UIN 669834132
isalva2@uic.edu
Instructor: **Zhiling Lan**

May 5, 2024

Contents

1 Introduction **2**
1.1 Project Scope 2
1.2 Background 2

2 Design Details **2**
2.1 Custom MPI Datatypes 2
2.2 Model Details 3
2.2.1 Model 1: SPMD Send and Receive 3
2.2.2 Model 2: SPMD Collective Communication 3
2.2.3 Model 3: Task and Data Parallel 3

3 Correctness Verification **4**

4 Performance Report **5**
4.1 Experimental Setup 5
4.2 Recorded Performance 5

List of Tables

1 RMSE Values 4
2 Model Run Times 5

List of Figures

1 Model 1 Detailed Run Time 5
2 Model 2 Detailed Run Time 6
3 Speedup Comparison 6

1 Introduction

1.1 Project Scope

This design report details the implementation of two-dimensional convolution on parallel systems using MPI and fast Fourier transform. Three designs were developed using different parallelization techniques and models:

1. Single program multiple data (SPMD) model using **send and receive operations**
2. SPMD model using **collective communication**
3. **Task and Data Parallel** model using eight processes to perform four tasks

1.2 Background

For two images represented as matrices A and B of size $N \times N$, each element is a complex number whose imaginary component is zero and real component is non-zero. To perform a two-dimensional convolution on these nmatrices, a two-dimensional FFT must be performed as an intermediate step. A 2D-FFT can be obtained by performing N, N-point 1D-FFTs on the rows of a matrix followed by N additional N-point 1D-FFTs along the columns of the intermediate results of the row-wise FFT operation. Two-dimensional convolution can therefore be performed on A and B with the following steps:

1. Perform a 2D-FFT on A.
2. Perform a 2D-FFT om B.
3. Point-to-point matrix multiplication (Hadamard Product) of A and B, $A \odot B$.
4. Perform a 2D-IFFT (inverse) on $A \odot B$.

The goal of this report is to document the implementation of two-dimensional convolution on two sets of images and record the performance of the three different parallelization models.

2 Design Details

2.1 Custom MPI Datatypes

At the heart of these implementations is the C structure `complex`, which is used to represent complex numbers as two floating point values. An equivalent custom `MPI_Datatype` designated as `complex_type` was created using `MPI_Type_create_struct()`, and was used to move `complex` data structures between processes.

In addition to `complex_type`, Model 1 and 3 made use of a custom `row_type` datatype. This datatype was based on the `complex_type`, and was constructed using `MPI_Type_vector()`. This vector datatype was used to communicate consecutive rows of A, B, and the output matrix between processes.

2.2 Model Details

2.2.1 Model 1: SPMD Send and Receive

For this model `model1_pp.c`, inter-process **point-to-point** communication using `MPI_Send()` and `MPI_Recv()` routines were used exclusively. Matrices A and B were partitioned using static, row-wise blocked scheduling, such that for p processes, each process receives N/p rows of A and B using the `row_type` data structure.

Using a for loop for scheduling, $p-1$ `MPI_Send()` routines were sent from the root process to the corresponding $p-1$ other processes. This scheduling was designed such that consecutive processes receive the next N/p rows of A and B in order of rank. Since $N = 512$ and p is assumed to be a multiple of 2^n, the partitions of A and B contain the same integer-valued amount of rows.

`MPI_Send()` and `MPI_Recv()` routines were implemented between mathematical or data manipulation operations: intermediate calls of `c_fft1d()` (of which two are used to perform a 2D-FFT or 2D-IFFT), transposing matrices, and performing the point-to-point matrix multiplication. In total 12 send and receive operations occurred between the root and worker processes, resulting in an estimated $12 \cdot p$ point-to-point `MPI_Send()` and `MPI_Recv()` pairs.

2.2.2 Model 2: SPMD Collective Communication

The collective communication model, `model1_cc.c`, was structured similarly to the previous model but utilized collective routines for inter-process communication. The `MPI_Scatter()` routine was used to collectively distribute data before mathematical operations, and the processed dispatched data was returned back to the root process using `MPI_Gather()`. In total, 6 `MPI_Scatter()` and 6 `MPI_Gather()` routines were used to implement this model. While this implementation does indeed use less explicitly stated communications than Model 1, this collective nature of these communications implies that they will be slower than `MPI_Send()` and `MPI_Recv()` pairs.

This model similarly uses row-wise blocked scheduling, but was *implicitly* scheduled when using collective call routines. Each process (including root) was dispatched N^2/p elements of A, B, or the output matrix, which corresponds to the *row_type* data structure used in the previous model to send/receive N/p rows of A, B, and the output matrix. Dispatched data was stored in a buffer structures `buffer1` and `buffer2` of size `[N/p][N]`, as collective calls using in-place buffers did not appear to work correctly during development.

2.2.3 Model 3: Task and Data Parallel

Model 3 utilizes a task and data parallel model, which splits the tasks and data described in 1.2 between four groups of two processes each. Groups 1 and 2 perform a 2D-FFT on A and B separately but concurrently (task parallel), group 3 performs the point to point matrix multiplication, and group 4 performs the 2D-IFFT on the output vector from group 3. Each group has two processes, which split the work (data parallel) of the group's task. In order to direct groups to different tasks, the `MPI_Comm_split()` routine was used to partition the processes from global communicator `MPI_COMM_WORLD`.

In order to use `MPI_Comm_split()`, the `color` and key of each process must be determined with resepect to the original global communicator. The `color` (group number) of each process can be either 0, 1, 2, or 3, depending on the original rank of each process, and the `key` (new rank) of each process is either 0 or 1 within a group. These values are inputs of `MPI_Comm_split()`, which then generates the new MPI Communicator `NEW_COMM`.

Inter-group communication and operations are the similar to Model 1 and uses `MPI_Send()` and `MPI_Recv()` pairs using the `NEW_COMM` communicator. Once a group completes their work, the root process of the group switches to the old global communicator, `MPI_COMM_WORLD`, to send data from one group to another. This occurs when Group 1 and 2 sends there processed copies of A and B to Group 3, and Group 3 sending the output matrix to Group 4 for the final 2D-IFFT operation.

As opposed to the previous models, the task and data parallel implementation utilizes inter-group communication between only two processes.

3 Correctness Verification

Prior to the development of the three parallel models, a serial version `serial_convolution.c` was written and verified for accuracy. This serial version was used for debugging purposes and was referenced during the design of the parallel models. At the conclusion of one of steps described in 1.2, the in-progress matrices of the parallel models were exported and compared against the serial model at the same step.

Upon completion of a parallel model the outputs were visually compared to the provided output data for initial verification. Following this inspection, the root mean square error (RMSE) of the outputs of each model was calculated against the source output data using the `rmse_check_convolution.c` file. The RMSE of each model, compared against the source output `"out_2"`, for the second set of data is shown below:

Table 1: RMSE Values

Model	processes	RMSE
Serial	1	0.026509
1	4	0.026509
2	4	0.026509
3	8	0.026509

The identical RMSE across the parallel models and serial model suggests that the minimal error is systematic and related to computation, not MPI communication. Regardless, the models are sufficiently accurate for the purposes of this report.

4 Performance Report

4.1 Experimental Setup

The parallel models were run on single node on the Chameleon Compute Cluster. The instance was hosted on a Dell PowerEdge R730 rack server with two Intel(R) Xeon(R) Gold 6240R CPUs @ 2.40GHz each. The instance was running Ubuntu 20.04 with MPICH 3.3.2 installed using Ubuntu's package manager and the command `$ sudo apt install mpich`.

Model run time was evaluated solely on the computation and communication time for each run, strictly excluding input/output operations. The MPI utility `MPI_Wtime()` was used for portability, as this utility records wall-clock time, not cycle time. For Model 1 and 2, communication and calculation time was additionally recorded in addition to total run time.

4.2 Recorded Performance

Models were evaluated on a range of processes between 2 and 16, and were compared against the serial run time of Model 1 with one process. Interestingly, Model 2 exhibited segmentation faults for run time executions with less than four processes, and were therefore excluded from summary Table 2.

Table 2: Model Run Times

Model	1	2	4	8	12	16
1	0.0477	0.0271	0.0168	0.023	0.0973	0.8086
2	-	-	0.0179	0.0256	0.1083	0.3262
3	-	-	-	0.0131	-	-

Looking at the run times for Model 1 in Figure 1, it is apparent that as the number of processes is increased communication becomes exceedingly costly.

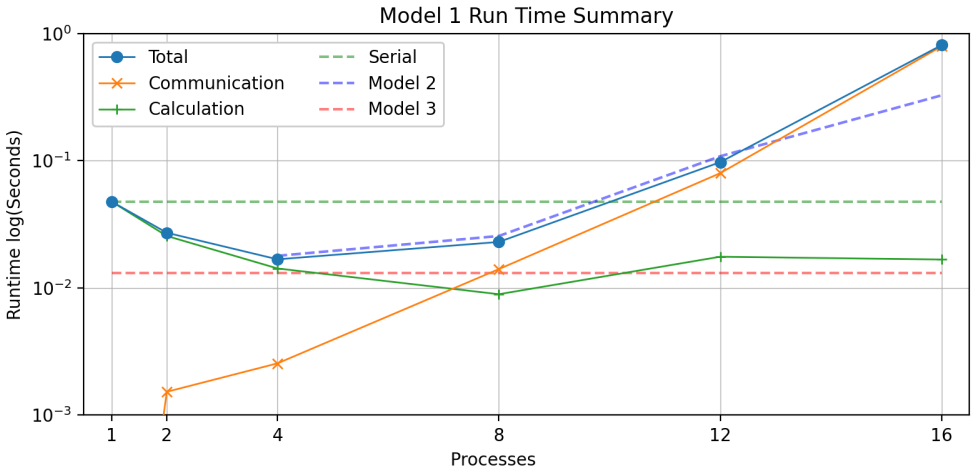

Figure 1: Model 1 Detailed Run Time

Similarly, Model 2 shown in Figure 2 exhibits increasing communication costs, and both models suffer from poor performance after 8 processes. Both Model 1 and Model 2 appear to asymptotically approach the limits of parallelization with respect to calculation time alone, suggesting that Model 3's architecture is well optimized for this type of problem.

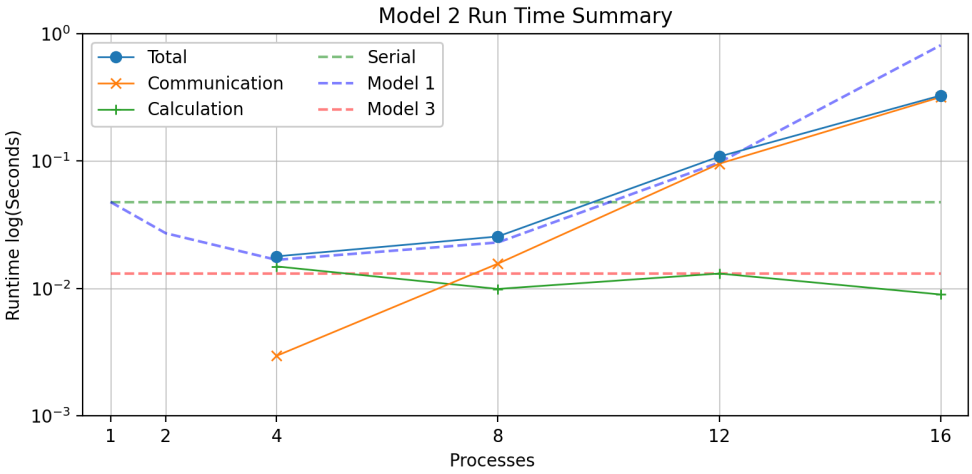

Figure 2: Model 2 Detailed Run Time

Looking at the speedup of all three models in Figure 3, All models exhibit sub-linear speedup, with best performance per process occuring around $p = 4$. However, Model 3, the Task and Data Parallel Model, had the best overall speedup of any model and processes configuration. Should this problem be explored with a larger N, the Task and Data Parllel Model incorporating more processes and possible more groups may be the best candidate for the task.

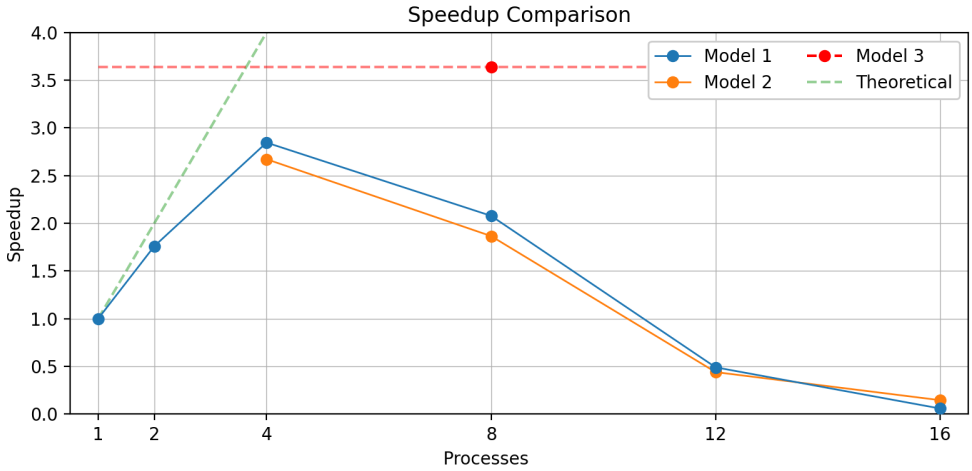

Figure 3: Speedup Comparison