High-Performance Computing

MPI: Convert color image to gray image

Anand Kamble
Department of Scientific Computing
Florida State University

1 Introduction

In this project, we will use the MPI (Message Passing Interface) to convert a color image into a grayscale one. We will also benchmark the program with a different number of processes.

2 Implementation

All the source code for this program will be written in the file `ColorToGray.cpp`. To compile this program you will need `Jpegfile.h` header file which is provided by the `JpegLib`. To compile the program, you will have to use the mpiCC compiler (use capital 'C' for C++ compiling) as shown in in listing below, which is available from the module `openmpi-x86_64`.

Listing 1: bash

```bash
mpiCC -o ColorToGray.exe Jpegfile.cpp ColorToGray.cpp JpegLib/libjpeg.a
```

If the module is not loaded, you can load it by running `module load mpi/openmpi-x86_64`.

2.1 Defining master process

To define the master process we are using the define directive. We have defined the master rank this way so the code becomes more readable and the master rank can be changed easily if required. It is implemented as shown in 2

Listing 2: ColorToGray.cpp

```cpp
#define MASTER 0
```

2.2 Initializing the MPI environment

The initialization of the MPI environment is done using a function named `MPI_Init`, which is provided by the `mpi.h` header file. In the below listing 3, we are also using the functions `MPI_Comm_rank` and `MPI_Comm_size`, to get the rank of the process and also the total number of processes. These numbers are stored in the addresses which are passed as the second argument.

Listing 3: ColorToGray.cpp

```cpp
MPI_Init(&argc, &argv);               // Initialize the MPI environment
MPI_Comm_rank(MPI_COMM_WORLD, &id);   // Get the rank of the process
MPI_Comm_size(MPI_COMM_WORLD, &p);    // Get the total number of processes
```

2.3 Reading the JPEG file

The JPEG file, which is the image, is only read by the master process. Implementation of this is shown in

Listing 4: ColorToGray.cpp

```cpp
if (id == MASTER) // Only the master rank reads the image file.
{
    dataBuf = JpegFile::JpegFileToRGB("test-huge.jpg", &width, &height);
}
```

2.4 Broadcasting the image height and width

Since only the master process reads the JPEG file, it alone has access to the image height and width. To ensure all other processes receive this information, the master process broadcasts the values using MPI_Bcast as shown in listing 5

Listing 5: ColorToGray.cpp

```cpp
MPI_Bcast(&height, 1, MPI_UINT16_T, MASTER, MPI_COMM_WORLD);
MPI_Bcast(&width, 1, MPI_UINT16_T, MASTER, MPI_COMM_WORLD);
```

This allows all processes in the MPI communicator to access the image dimensions locally, facilitating parallel processing without dependency on the master process.

2.5 Distributing the work

To distribute the workload among multiple processes, the image is divided into parts, with each process responsible for processing a portion of the image. The number of rows to be processed by each worker process (`workerRows`) is calculated by dividing the total height of the image (`height`) by the number of processes (`p`). Any remaining rows after the division (`extraRows`) are handled by the master process.

Listing 6: ColorToGray.cpp

```cpp
auto workerRows = height / p;
int extraRows = height % p;

auto rowsToProcess = workerRows;
if (id == MASTER) {
    rowsToProcess += extraRows;
}
```

The memory required for storing the data to be processed by each worker process (`workerData`) is allocated based on the calculated number of rows to process (`rowsToProcess`), the width of the image (`width`), and the size of each pixel (`3 * sizeof(BYTE)`).

2.6 Scattering the Data

The master process scatters the image data to all other processes using MPI_Scatter. This function divides the image data (`dataBuf`) into smaller, equally sized blocks and distributes them among the processes. Each process receives a portion of the image data in its `workerData` buffer.

Listing 7: ColorToGray.cpp

```cpp
auto scatterCount = workerRows * width * 3; // Number of elements to scatter
MPI_Scatter(dataBuf, scatterCount, MPI_BYTE, workerData, scatterCount,
    MPI_BYTE, MASTER, MPI_COMM_WORLD);
```

2.7 Gathering the Processed Data

After processing their respective portions of the image, all processes gather their results back to the master process using MPI_Gather. This function collects data from all processes and arranges it in the master process's buffer. In the following listing 9, each process sends its processed data (`workerData`) to the master process's buffer (`dataBuf`). Once all processes have completed their tasks, the master process has access to the complete processed image data.

Listing 8: ColorToGray.cpp

```cpp
MPI_Gather(workerData, scatterCount, MPI_BYTE, dataBuf, scatterCount,
    MPI_BYTE, MASTER, MPI_COMM_WORLD);
```

2.8 Writing the JPEG file

Finally, the master process writes the processed data to a new file using the `JpegFile::RGBToJpegFile` function. This function write the processed image data to a JPEG file. Only the master process performs this task. Once completed, the grayscale image file `testmono.jpg` contains the grayscale image.

Listing 9: ColorToGray.cpp

```cpp
if (id == MASTER) { // Only the master rank writes the image file.
    JpegFile::RGBToJpegFile("testmono.jpg", dataBuf, width, height, true, 75);
}
```

2.9 Finalizing the MPI Environment

After completing all MPI operations, the MPI environment is finalized using `MPI_Finalize`. This function ensures proper termination of MPI execution and releases any allocated resources. By calling `MPI_Finalize`, the MPI communication is concluded, and the program gracefully exits, allowing for the proper cleanup of MPI-related resources.

Listing 10: ColorToGray.cpp

```cpp
MPI_Finalize();
```

3 Errors and Debugging

While developing this, the extra rows after dividing the work should be processed by the master process. Making sure that the master thread performs the correct operation on the remaining rows was a bit tricky.

4 Results

This program was benchmarked with a number of processes ranging from 1 to 14.

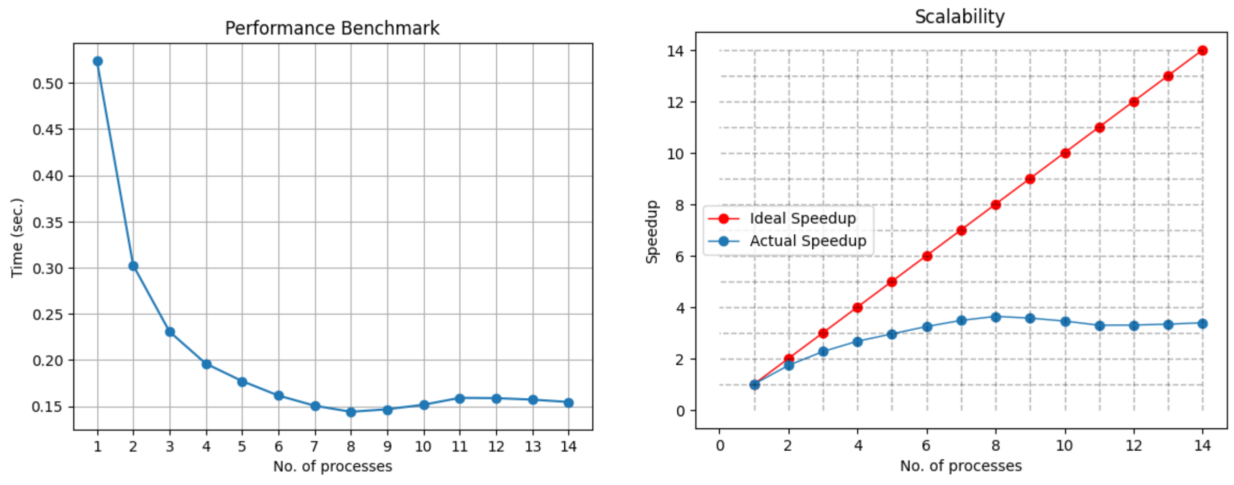

Figure 1: Timing and Scaling of the program

5 Conclusion

It is clear from the above graphs that after 8 processes, the execution time starts to increase a bit which means that the time required for communication between machines is increasing with the number of machines, and hence slowing down the program.