Midterm Part II of III - MPI IO & Grid Distribution

Adam Ross
adro4510@colorado.edu

Abstract

This project report addresses the implementation and correctness of my grid implementation of Conway's game of Life for both synchronous and asynchronous communication and parallel file output via MPI IO. To prove the correctness of my implementations I will be comparing the living bug counts for various processor counts and iteration counts to known values. These include 1000 and 10000 iteration counts and 4, 9, 25, and 36 processor counts, and the compared data will come from part one of this midterm. In addition I will be using code snippets to prove both the grid implementation and the MPI IO implementation. I will then detail scripts used to speed up repetitive development processes when developing on the computing cluster Comet.

1. Introduction

When processing large chunks of data it is useful to split this data inter smaller sets for processing. Building on the last midterm portion, which implemented row decomposition, we now look at a more scaleable solution of grid decomposition. We find value in the method with data sets that may expand as n^2, as a row decomposition would maintain the full width of the data.

Additionally, when moving and generating enormous sets of data, as is done on large computing clusters, it would be an extreme bottleneck to be doing serial IO to disk. Given this problem there is motivation for parallel file systems and IO operations. MPI IO provides a library to accomplish this much, and it is worthwhile to investigate these methods.

I first examine code snippets showing the respective methodologies, then I examine the output and bug count across a plethora of input permutations.

2. Overview

To implement a grid distribution we are more or less adding a second dimension to our parallel processing and data decomposition. First I will provide pseudo code for my row and column process assignment. This is detailed below in figure 2.1.

```
nrows = (int)sqrt(np);
ncols = (int)sqrt(np);
my_row = rank / nrows;
my_col = rank - my_row * nrows;
```

Figure 2.1 - Row and column process assignment.

Here we are assuming, and checking later, that we will have a square processor count where the data distributes evenly across all processes. This is checked in later code and returned if false.

It is then useful to define a column data type to pass out relevant column data to the vertical padding on surrounding each process's data set. This is done via the MPI vector datatype, where we pass only our local height as to implement two stage grid passing. Two stage passing details passing column data with local height padding horizontally and field width padding vertically allowing the passage of corner data. This allows only four communications per block as opposed to eight; one for each side opposed to one for each side and corner. This is outlined in figure 2.2 and used in figure 2.3.

```
          // Create data type to extract useful data out of padding
          MPI_Type_vector(local_height, local_width, field_width, MPI_UNSIGNED_CHAR,
&ext_array);
          MPI_Type_commit(&ext_array);
```

Figure 2.2 - Column data type detail.

```
// Send to right or recv from left
MPI_Sendrecv(&env_a[1 * field_width + 1], 1, column, left_dest, 0,
             &env_a[2 * field_width - 1], 1, column, left_source, 0,
             MPI_COMM_WORLD, &status);
// Send to left or recv from right
MPI_Sendrecv(&env_a[2 * field_width - 2], 1, column, right_dest, 0,
             &env_a[1 * field_width + 0], 1, column, right_source, 0,
             MPI_COMM_WORLD, &status);

// Send to below or recv from above
MPI_Sendrecv(&env_a[1 * field_width + 0], field_width, MPI_UNSIGNED_CHAR, top_dest, 0,
             &env_a[(field_height - 1) * field_width + 0], field_width,
             MPI_UNSIGNED_CHAR, top_source, 0, MPI_COMM_WORLD, &status);
// Send to above or recv from below
MPI_Sendrecv(&env_a[(field_height - 2) * field_width + 0], field_width,
            MPI_UNSIGNED_CHAR, bot_dest, 0,
             &env_a[0 * field_width + 0], field_width, MPI_UNSIGNED_CHAR, bot_source,
             0, MPI_COMM_WORLD, &status);
```

Figure 2.3 - Horizontal - vertical two-stage communication - synchronous.

Similarly to the synchronous code the asynchronous code also uses two stage communication. This, not unlike the asynchronous code from midterm part I, was also spread around various other pieces of computational work to allow processing in between and during communication. The process is outlined below in pseudo code.

```
<Start algorithm>

<calculate destinations and sources for padding exchanges>
// Initial exchange for horizontal communication
MPI_ISend(env_a data, row_length, MPI_CHAR, left_dest, tag, MPI_COMM_WORLD, &request);
MPI_ISend(env_a data, row_length, MPI_CHAR, right_dest, tag, MPI_COMM_WORLD,
&request);
```

```
<begin algorithm>
      <calculate destinations and sources for ghost row exchange>

      MPI_IRecv(env_a data, row_length, MPI_UNSIGNED_CHAR, right_source, tag,
      MPI_COMM_WORLD, &request);
      MPI_IRecv(env_a data, row_length, MPI_UNSIGNED_CHAR, left_source, tag,
      MPI_COMM_WORLD, &request);

      <calculate N + 1 state>

      // send the data we just calculated as soon as we know it
      MPI_Isend(env_b data, 1, column, left_dest, tag, MPI_COMM_WORLD, &lr);
      MPI_Isend(env_b data, 1, column, right_dest, tag, MPI_COMM_WORLD, &rr);


      <print to file if need be>
      <count if need be>
      <any other work>

          MPI_Irecv(env_b data, 1, column, left_source, tag, MPI_COMM_WORLD,
      &request);
          MPI_Irecv(env_b data, 1, column, right_source, tag, MPI_COMM_WORLD,
      &request);
          // Need the horizontal data before we send vertically
          MPI_Wait(&lr, &status);
          MPI_Wait(&rr, &status);

          MPI_Isend(env_b data, field_width, MPI_UNSIGNED_CHAR, top_dest, tag,
      MPI_COMM_WORLD, &request);
          MPI_Isend(env_b data, field_width, MPI_UNSIGNED_CHAR, bot_dest, tag,
      MPI_COMM_WORLD, &request);
```

Figure 2.4 - Asynchronous ghost row exchange pseudocode.

To accomplish the parallel IO required we first need to build two custom MPI datatypes; a distributed array and an array-padding extraction vector. Both are outlined below in figure 2.5.

```
    // Create darray and commit
    MPI_Type_create_darray(np, rank, 2, gsizes, distribs, dargs, psizes, MPI_ORDER_C,
            MPI_UNSIGNED_CHAR, &darray);
    MPI_Type_commit(&darray);

    // Create data type to extract useful data out of padding
    MPI_Type_vector(local_height, local_width, field_width, MPI_UNSIGNED_CHAR,
            &ext_array);
    MPI_Type_commit(&ext_array);
```

Figure 2.5 - MPI datatypes to distribute an array and to extract the data we want from our sub arrays.

In order to write only the data we care about to file we create a vector data type where the stride is the whole data width, the data chunk size is the useful data width and that we repeat for the row count of the relevant data. The distributed array knows how many processes we have, this process's rank, the global array size, and how we distribute this data. This is used when telling each process exactly where to write its chunk of the data into the current file. These two data types are then combined with the MPI_File_set_view and MPI_File_write calls to write our data out onto the parallel file system available on Comet. The important file writing code is included below in figure 2.6. Notice we write the pgm file header before writing the Conway data.

```
	MPI_File_open(MPI_COMM_WORLD, frame, MPI_MODE_CREATE|MPI_MODE_WRONLY,
MPI_INFO_NULL, &out_file);

	char header[15];
	sprintf(header, "P5\n%d %d\n%d\n", global_width, global_height, 255);
	int header_len = strlen(header);

	//write header
	MPI_File_set_view(out_file, 0,  MPI_UNSIGNED_CHAR, MPI_UNSIGNED_CHAR, "native",
MPI_INFO_NULL);
	MPI_File_write(out_file, &header, 13, MPI_UNSIGNED_CHAR, MPI_STATUS_IGNORE);

	// write data
	//MPI_File_set_view(out_file, 15 + rank * local_width + local_width,
MPI_UNSIGNED_CHAR, darray, "native", MPI_INFO_NULL);
	MPI_File_set_view(out_file, 13, MPI_UNSIGNED_CHAR, darray, "native",
MPI_INFO_NULL);

	//MPI_File_write(out_file, env_a, (local_height * local_width), ext_array,
&status);
	MPI_File_write(out_file, &env_a[field_width + 1], 1, ext_array, &status);
	MPI_File_close(&out_file);
```

Figure 2.6 - MPI IO set view and write calls.

In order to debug and develop faster several scripts were developed. The first I shall mention is a comet remake script, which made developing MPI IO many times less tedious. This script is detailed below in figure 2.7.

```
git pull && make clean && make && cp RossAdam_MT2 ../../bin && rm -f
/oasis/scratch/comet/adamross/temp_project/* && sbatch batch_files/RossAdam_testing.sh
&& watch squeue -u adamross && cat "dev/comet_out/$(ls -1rt dev/comet_out/ | tail
-n1)"
```

Figure 2.7 - Comet remake script. Pulls down the latest code, re-makes it, submits a testing batch file. Watches the queue, and cats the output when we exit the watch command.

Additionally to support the development and debugging on the grid distribution an animation script was created seen below in figure 2.8. These scripts reduced redundant keystroke operations many times over.

This is something I have experienced at work as well. If you need to perform a task over and over it will be well worth the time to script it up.

```
cp /oasis/scratch/comet/adamross/temp_project/* dev/data/. && for f in `ls -1
dev/data`; do xxd -p -c 16 ${f}; sleep 0.5; done
```

Figure 2.8 - Comet anim script. This script copies the parallel output into the local directory and pops out a terminal visualization of the frames produced.

3. Verification

Below is a table containing the various output information from the serial, synchronous and asynchronous implementations.

N	0	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
Serial	25301	18340	16512	16001	15449	14953	14953	14953	14953	14953	14953
Sync-Row	25301	18340	16512	16001	15449	14953	14953	14953	14953	14953	14953
Async-Row	25301	18340	16512	16001	15449	14953	14953	14953	14953	14953	14953
Sync-Grid	25301	18340	16512	16001	15449	14953	14953	14953	14953	14953	14953
Async-Grid	25301	18340	16512	16001	15449	14953	14953	14953	14953	14953	14953

Figure 3.1 - Serial and Synchronous to 10000, np = 9 counts.

Row Distribution

Method - np	Sync - 4	Sync - 9	Sync - 25	Sync - 36	Async - 4	Async - 9	Async - 25	Async - 36
Alive at 1000	18340	18340	18340	18340	18340	18340	18340	18340

Figure 3.2 - Asynchronous and Synchronous Counts for Row distribution varying np.

Grid Distribution

Method - np	Sync - 4	Sync - 9	Sync - 25	Sync - 36	Async - 4	Async - 9	Async - 25	Async - 36
Alive at 1000	18340	18340	18340	18340	18340	18340	18340	18340

Figure 3.3 - Asynchronous and Synchronous Counts for Grid distribution varying np.

In my last report I had seen some strange numbers coming back from my async communication. I had also seen this when debugging this time, which I came to find was that I had not completely initialized my whole array to 0s initially, which in some cases was producing strange numbers.

4. Conclusion/Learned

Given the nature of asynchronous communication, which allows us to do computation while running communication through the pipeline I would expect a reasonable performance increase. The limitation here being that we cannot be sending while actually iterating over the array. We need the current data to do the array calculations and can only send them once we have done them, hence there cannot be any asynchronous communication per process while we do the main chunk of our work.