Assignment-1 Report

Minghan Yu

1.At the beginning of the assignment, I firstly create a 1d array of size X_limit*Y_limit as my send buffer. Then I create a 1d array of size interval*Y_limit where interval is X_limit/numprocs as my receive buffer. These two arrays should be declared in all processors so that I can use MPI_Scatter. In root, I read the input file and get the life** matrix. Then I flattened the **life matrix into 1d and then stored in send buffer in root. Then MPI_Scatter is called for every processor. It divides send buffer and every processor get a receive buffer respectively. If the life** matrix is flattened row dominantly, then I don't need to worry about indices here.

After each processor stored the data from receive buffer to a new 2d array, called block, which is smaller and has size (interval)*(Y_limit). I then create empty 2d array of size (interval+2)*(Y_limit+2) and call it previous_block. If I send block and previous_block into the function "compute" which comes from serial.C, I can directly compute game of life without modifying the function "compute". Then before "compute" in each generation, I do MPI_Send/Isned and MPI_Recv/Irecv to get the upper and lower rows in previous_block.

2. I add an if condition to firstly separate processor with odd ranks and even ranks. Then even ones firstly send their first rows to the previous processor, and the odd ones firstly receive data from the next processor.

Then even ones receive from the previous processor and odd ones send their last row to the next processor.

The next two pairs of send and receive operations happen with the same rule. Odd ones send then even ones receive, and vice versa. One point is that processor 0 doesn't send its first row or receive anyone's last row, and the last processor doesn't send its last row or receive anyone's first row.

3. Performance result

The graphs of execution time v.s. number of processors are shown below for both blocking and nonblocking cases. Nonblocking one runs a little bit faster than the blocking ones. This is reasonable because processors and utilize the waiting time to do other tasks. The trends behind the two graphs are identical. From 4 processors to 64 processors, the execution time drops nearly exponentially. This is reasonable because the parallel part occupies most of the this game of life code, i.e. ,time of sending and receiving only 4 rows can be ignored. Then the number of operations will depend linearly on the number of processors. Thus, the number of processors increase exponentially, the execution time should also decreases exponentially.

There is a big leap of execution time form 64 processors to 128 processors. This is also interpretable because the overhead of too many communications and synchronization might overweigh the benefits of parallelization.