Game of Life Implemented Using OpenMPI System

CMPE 300, Analysis of Algorithms, Mehmet Erdinç Oğuz, Programming project, 22/10/2019

Introduction

In this project, I implemented a cellular automaton called "Game of Life" using parallelism. I used OpenMPI Interface to apply the parallelism to the project, and I handled the deadlock situations without using the functions that MPI provided but with arranging the order of "receive" and "send" calls.

Program Interface

One can run this program by compiling the source file and run the executable file. However, since we use OpenMPI system, one should use the *"mpic++"* compiler to compile our code. To run the project, one can use these two consecutive commands below:

"mpic++ main.cpp -o [name of executable you prefer]"

"mpirun -np [number of processors to use] –oversubscribe ./[name of executable] [inputfile] [outputfile] [number of iterations]"

The first command is straightforward, we just change the compiler to "mpic++". On the other hand, to run the executable, we should declare number of processors since we are using a parallel algorithm and parallelism interface. It should be [a square number] + 1 and the sqrt of this square number should divide 360. The "+1" is the master process and the rest are the slave processes. Since the focus of this project is to implement a parallel algorithm, the program doesn't work with one slave. The "—oversubscribe" flag is to create more processes than the CPU has processors since this is also a part of the project. Final 3 arguments are to get the input, print the output and simulate the game with right amount of iterations.

Program Execution

To run this program properly, one need to txt files; one for which to get the 2d matrix that the game will play on, and second is to print the final state of the game. And user should also give the number of iterations during the execution part at the end of the command.

Input and Output

Input and output files are ".txt" files and number of iterations should be given as an integer. Input should be given as a txt file in which each line has the values (should be 360 values) of a row, and consists of 360 lines. These values should be "0" or "1" because our game shows the alive creatures as "1" and dead ones as "0". If user wants to play this game with a different format of input, one should be careful that they should give a square array and they should change the "GRID_SIZE" variable in the code to the length of a side of this 2D array. Also, one should be careful about the process numbers as it's explained above that it has some constraints.

Program Structure

In this program, we first initialize the MPI System. After that, we declare some variables used in the rest of the code. Then, we split the code into two blocks using if/else. One block is for the master process, and the other for slave processes.

In the master process, we take the input from the input file. Then I start to send the data to other processes by splitting the data into equal sized smaller arrays. I didn't find a right way to send a 2D array from one process to another using MPI_Send method so I decided to convert the array into one-dimensional and then convert back when it is necessary while I send it. After I send it, master waits for the modified data from other processes and when it gets the data, it uses that data to create the final big array. At the end of that part, master prints the final 2D array to the output file.

In the other block of the code, we implement how a slave behaves. A slave first waits for the data from the master. When it gets the data, it turns that 1D data into 2D array to make changes on it later easily. Then, the slave starts the simulation in a while loop for the number of iterations that is given. The slave first initializes the data needed during that state while sending and receiving. After that, it behaves according to its process rank. There are a couple of if/else blocks to make the process behave in the right way. During if/else blocks, the slave process receives and sends the data from and to the other processes. The communication here is implemented in a way to prevent the deadlocks. I was inspired from the solution in the project documentation of the project and modified it a bit when I was writing the project. All ranks need to communicate with the other ranks in 8 directions. First, one process should calculate the ranks of the processes it needs to communicate. Here, each process needs to know the outer side of itself because of the game rules. For example. the top right corner cell needs to know the row above it, the corner next to it, and the column at the right of it to calculate its state. So, each process sends and receives rows and columns and single cell values to one another. I created a pattern for sending and receiving. First, each process handles the right-left communication. Here, processes send and receive 1D array of columns. Rank with an odd value first sends and then receives the data After that, they handle top communication. This also includes top-left and top-right since it is easier to handle them while we know the address of the upper rank. Here, I also split the processes into two part to prevent the deadlock. The processes in 1^{st}, 3^{rd}, 5^{th}… row behave the opposite way of the processes in 2^{nd}, 4^{th}, 6^{th}… row. When one sends the data, the other receives and vice versa. To the top, each process sends the row of its top side. Then the process sends its right and left corner values to the other processes after calculating the rank of them using the rank of the process above. The same applies to the bottom side. Then they wait for receiving. When they receive the data, they create a bigger 2D array to make the calculations of the current 2D array. They add the received data to that bigger array and make the calculations. When the calculations are done, the current data in the process rank is updated and the next state is executed. If there is no next state, then the process converts the data to 1D array and sends it back to the master for merging.

Difficulties Encountered

In this project, I learned the MPI system during my development and it was a quite new concept, but I really enjoyed it. I had a deadlock problem at the beginning and then I realized that it was because of the tags that I use with send and receive functions. Then I tried a couple of ways to figure out how to solve this problem by searching from the internet. Then, I found a way and prevented the deadlock. Then I had a problem with the arrays I send. I first tried to send the arrays in 2D form but then I realized

that some of the arrays had random values inside of it. So, I tried some more ways to send the arrays in 2D form but then, I gave up and just converted it into 1D array because that was the way I learned from the internet and I knew that it was working properly. However, I am still curious about sending a 2D array using MPI. I also had problems with saving the code since I made experiments on it a lot. It would be better if I used a version control system to save my code and check later. Also, I will use a pure Linux OS next time since I had a hard time setting up the MPI to Windows.

Conclusion

I get the right output when I test my code. My implementation works fast enough in terms of time it takes to execute with 145 processes. That was a good project. I learned a new great concept. I will try to apply parallelism if I can when I work on some programming project. I also improved my C++ skills during that project. I believe my project works in a fast way, and I tried to make my code as readable as possible. If I made this project again, I would use tools like GitHub to save my code and a pure Linux OS since I had a hard time setting up the MPI to Windows.