Jason Zhang
Parallel HW3 Written

1.

a. The deadlock-free approach of my solution came in the form of defining different send and receive ordering for even and odd processes. For even processes I have a send first and a receive following. For odd processes I have a receive first and send following. This reverse ordering for even and odd processes allows for an MPI_Send() and MPI_Recv() to be aligned and happen at the same time for contiguous processes. Since MPI_Send() will block until the sender can reuse the sender buffer, by putting MPI_Recv() first for the odd processes we will have posted the matching acknowledgement from the recv and our send will not block. Had we put MPI_Send() first for odd processes we would've encountered a block since our MPI_Send() from our even process will not get the recv it is looking for. (Note: even processes start with send and odd processes start with recv because we start with process 0.)

1b.

Let us define a square grid such that num_rows = num_cols = n = DIM, where DIM = 16. Let p = number of processes.

Starting with the number of messages passed, we only pass a constant number of messages (2 sent, 2 recv, 1 gather). Thus our big O notation will be constant which is O(1).

Now with the amount of memory used per process for a single iteration of the game, we know that each process will work with only n/p rows and n columns. This gives us a big O notation of O(n^2 / p).

Our total big O calculation will yield: O(1) + O(n^2 / p) = O(n^2 / p).

1c.

For our processes, we split our processes into even or odd processes. For even processes we send the first row to the previous process and send the last row to the next process (Even processes send and then receive). For odd processes we wait to receive the rows sent by the even processes and then send the top row to the previous process and the bottom row to the next process (Odd processes receive and then send). Each process has the dependency of getting the start and end rows of the previous and next processes. By have this reversed sending scheme for even and odd processes we can avoid cyclical dependencies which would cause deadlocks due to an even process sending and blocking infinitely due to not having a matching receive sending back an acknowledgement. We send the starts and ends of each block to the previous and next processes respectively in order to get a synchronization between our updates for each process. We then apply an MPI_gather such that our root process will get all the updates from each process and we have a fully synchronized and correctly computed cellular automata.

2

a.

Number of MPI messages will still be constant. However, in this case, each square now has 4 neighbors each, and must send and receive messages to all these neighbors. This gives us 1 send 1 recv from each neighbor and then an MPI_gather to get all of the information from all of the squares which is constant(9). This gives us O(1).

For each square, the size of each message is the size of each square. We define the rows and columns each process gets as n/sqrt(p) where n = DIM(dimension of 1 side of grid) and p = processes. Thus, the size of each square is (n/sqrt(p)) * (n/sqrt(p)) = n^2/p. Since every message we send is of this size, we know that the Big O will still be O(n^2/p).

The advantages and disadvantages of this decomposition when compared with the horizontal slicing of the space are that this decomposition still sends with the same Big O as our horizontal slicing. However even though both implementations send constant messages per iteration, the square decomposition sends more messages, since we need to consider more neighbors. This means that as our problem size grows and we either increase iterations or increase the number of processes, we will have to send far more messages for the square decomposition than we do for the horizontal slicing which could slow down our runtime considerably as we have far more dependencies to consider. This decomposition is also more difficult to implement as we need to consider more neighbors for message sending and it will be more difficult to resolve potential deadlocks.

2

b.

We let n = DIM(dimension of grid) and p = processes.

Our square decomposition will have n/sqrt(p) rows and columns which means each square is of size n^2/p. And given that it scales will our problem size we can expect to see that our overhead will be in the order of O(n^2 / p).

For advantages and disadvantages, the square decomposition has the same memory overhead as our horizontal slicing. A disadvantage however is that for square decomposition we could potentially have more neighbors to consider which means we have to consider more memory spaces as we need to get the data from all of the neighbors. So even though we have the same order of O(n^2/p) for both strategies, we ultimately have to consider more for the square decomposition and thus possibly taking up more memory per process.

2

c.

```
length_block = sqrt(num_procs);
prev = //previous process;
next = //next process;
for ( iters = 0; iters < num_iterations; iters++ ) {
  //sending for even/odd left and right neighbors
  if (ID % 2 == 0) {
    //send start row to previous process and sent end row to next process
    //recv end row from previous process and recv start row from next process
  }
  else{
    //recv end row from previous process and recv start row from next process
    //send start row to previous process and sent end row to next process
  }
  //sending for even/odd top and bottom neigbors
  if(ID / length_block % 2 == 0){
    //send start row to previous process and sent end row to next process
    //recv end row from previous process and recv start row from next process
  }
  else{
    //recv end row from previous process and recv start row from next process
    //send start row to previous process and sent end row to next process
  }
  //sending for left diagonal neighbors
  if((ID / length_block - 1) % 2 == 0){
    //send start row to previous process and sent end row to next process
    //recv end row from previous process and recv start row from next process
  }
```

```
    else{
        //recv end row from previous process and recv start row from next process
        //send start row to previous process and sent end row to next process
    }
    //sending for right diagonal neighbors
    if((ID / length_block + 1) % 2 == 0){
        //send start row to previous process and sent end row to next process
        //recv end row from previous process and recv start row from next process
    }
    else{
        //recv end row from previous process and recv start row from next process
        //send start row to previous process and sent end row to next process
    }
    //gather data, compute updates
    // Output the updated grid state
}
```

For this protocol we implement a similar even and odd scheme that we did for horizontal slicing. In this case we start differentiating even and odd processes in the left right direction, top bottom direction, and left diagonal right diagonal direction. We avoid deadlocks because for adjacent processes we have matching send and recv for the transmission such that the send will receive a corresponding acknowledgement at the same time so that we won't have the send blocking our program waiting for the matching recv. We index the left and right direction the same way as before in which we sequentially assign even or odd based on ID. For top bottom and diagonals, we start looking at 1 block size on top and one block size below(offset by one: left or right, for diagonals). Each block has n/sqrt(p) for rows and columns(n=DIM), and we look at ID/sqrt(p) and assign it either even or odd(offset by 1 left or right for diagonals).