Homework 2

Daniil Pakhomov

March 6, 2017

1 Question 1

1.1 Task

Please characterize (describe your answers):

1. The deadlock-free approach of your solution and its efficiency.
2. The dependencies between the decomposed parts of the problem.

1.2 Answer

In order to better explain our approach, we first refer the reader to the Figure 1, where we explain how the whole matrix was divided into blocks which were assigned to different processes.

Let the whole matrix be of size $D \times D$ and let the elements in it be of type "int32"(for our case it can be boolean because we store only 1s and 0s). Let's define d to represent number of bits that is required to store one element of a type "int32". For our specific case, $D = 16$ and $d = 32$.

Using the naming convention from the Figure 1., we follow up with the description of our algorithm. Our approach performs the updates in the game by following these rules:

1. Divide the matrix equally into M pieces, where M can be $1, 2, 4, 8, 16$ and assign each process to be responsible for the respective block. Therefore, number of process that we have is also equal to M.
2. For each process, allocate two additional arrays of size D to store values from previous and next block which are named "Previous row" and "Next row" in the Figure 1.
3. Loop for the number of iterations I:
   (a) Perform updates in each block according to the rules of the game.
   (b) Perform synchronization using barrier. We do this to make sure that updates were performed for all the blocks before we send messages.
   (c) Perform the deadlock-free message passing of bordering rows between adjacent blocks.
   (d) Perform gather operation to collect the updates in the main process (process number 0 in our case).

Each process has to store its part of matrix of size $d * D * (\frac{D}{M})$ and two additional rows $d * D * 2$, resulting in $d * D * (\frac{D}{M} + 2)$ overall.

On each iteration I a total number of $2 * M$ messages is sent, each of size $d * D$, making overall $2 * M * (d * D)$.

The decomposed parts are dependent in a way that they need to pass the bordering elements between each other to follow on the next stage of the game. This slows down the speed of the whole program, as we have to perform synchronization.

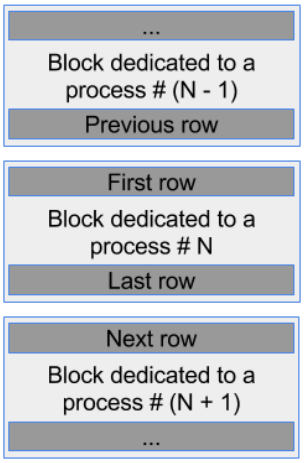

Figure 1: The figure demonstrates the problem that we have when performing the updates for the game. Each process has to copy row from process that is assigned to the previous block of a matrix and from the next block. The rows that it has to copy are named "Next row" and "Previous row" in our figure. At the same time the block has to send rows named "First row" and "Last row" to $(N-1)$ and $(N+1)$ blocks respectively. After these rows has been received from and sent to other processes, the blocks are updated independently by each process.

To demonstrate our deadlock-free approach we provide the following piece of original code. The names of the variables follow the same naming convention as in the Figure 1. The full code is available here: https://github.com/warmspringwinds/parallel_programming_jhu

```
// Don't send anything if we only run on one process
// Was tested with MPI_Ssend -- works.
// We avoid the deadlock here by pairing up operations.
// For this approach to work the number of nodes should be
// even.
if(num_procs != 1)
{
  if ( id % 2 == 0)
  {
    MPI_Send(row_first_to_send_pointer, /*..*/ prev_id, /*..*/);
    MPI_Recv(row_next_to_receive_pointer, /*..*/, next_id, /*..*/);
    MPI_Send(row_last_to_send_pointer, /*..*/, next_id, /*..*/);
    MPI_Recv(row_previous_to_receive_pointer, /*..*/, prev_id, /*..*/);
  }
  else
  {
    MPI_Recv(row_next_to_receive_pointer, /*..*/, next_id, /*..*/);
    MPI_Send(row_first_to_send_pointer, /*..*/, prev_id, /*..*/);
    MPI_Recv(row_previous_to_receive_pointer, /*..*/, prev_id, /*..*/);
    MPI_Send(row_last_to_send_pointer, /*..*/, next_id, /*..*/);
  }
}
```

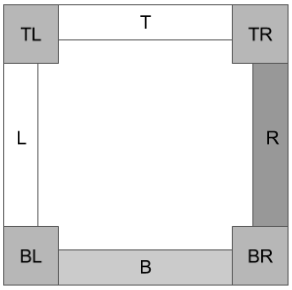

Figure 2: The figure demonstrates different parts of a block that is assigned to one process. Each named part of the block represents different parts that has to be sent over to other blocks, that need them in order to perform next step in the game. While the block needs to send this parts to other blocks, it also has to receive same parts from other bordering blocks. Take into account, that corner elements overlap with elements T, L, B, R.

2 Question 2

2.1 Task

Consider an alternative spatial decomposition in which we divide the grid recursively into smaller squares, e.g. an n x n grid may consist of 4 $(\frac{n}{2}) \times (\frac{n}{2})$ squares or 16 $(\frac{n}{4}) \times (\frac{n}{4})$ squares or 64 $(\frac{n}{8}) \times (\frac{n}{8})$ squares. What are the relative advantages and disadvantages of this decomposition when compared with the horizontal slicing of the space in your implementation:

1. On the number and size of MPI messages in the simulation? (Give an expression for each decomposition for the size and number of messages as a function of the number of processes)
2. On the memory overhead at each processor? (Again, give an expression for each decomposition)
3. On the flexibility of decomposition?
4. For the new decomposition, describe a deadlock-free messaging discipline for transmitting the top, bottom, left, and right sides and the top left, top right, bottom left and bottom right corners among neighboring partitions. Pseudocode or a drawing will be helpful in describing the protocol.

2.2 Answer

The number of MPI messages that are necessary for each iteration I is: $M * 2 * 8$, where M is a number of processes. The number 2 comes from the fact that messages come in pairs. For example, in order to perform next step in the game we have to send element TL and receive element BR, which can be done in a deadlock-free way about which we will elaborate later in this chapter. The number 8 represents the number of parts that are depicted at Figure 2.

Following the notation introduced in the previous section, we can derive the memory overhead: $d * (\frac{D}{S})^2$ for storing the block and $4 * d * \frac{D}{S}$, where $S = \sqrt{M}$, for storing the bordering rows and $4 * d$ for storing corner elements. Overall giving us $d * (\frac{D}{S})^2 + 4 * d * \frac{D}{S} + 4 * d$.

This scheme is better in a way that it makes it possible to break the problem down into bigger number of subproblems compared to the row-slicing approach and, therefore, spread it between greater number of nodes, overall, achieving faster . At the same time, it requires to pass more messages and overall store more additional elements.

The deadlock-free approach can be implemented following our example. Figure 3. shows an example of a matrix that has been split into blocks. Each block is dedicated to a separate process. In order to perform message passing in a deadlock-free fashion, we split this problem into three subproblems: message passing between **(i)**elements that are adjacent vertically, (ii) elements that are adjacent horizontally, **(iii)** elements that are adjacent diagonally.

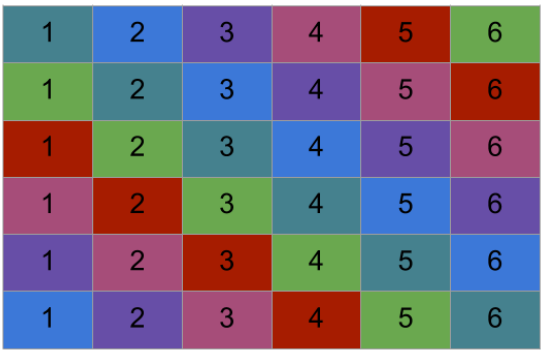

Figure 3: A figure that demonstrates our approach where we group all elements by the cycle that it belongs to. Each cycle is represented by a different color. Pay attention that we perform index wrapping when looking for a diagonal neighbour of a element. Number of elements in each cycle is even, given the fact that D is even. Numbers represent the id of an element in a cycle that we later on use to pair up message sending operations to avoid deadlock.

The problem of message passing for problems **(i)** and **(ii)** can be solved in the same way that the problem in the previous question was solved: by pairing elements up based on their column and row index alone.

The problem **(iii)** is a little bit more complicated as it it harder to group elements. We solve this problem in three stages.

First we change the coordinate system to better suite our solution. We alter it in the following way:

$$\bar{c} = c$$
$$\bar{r} = r - D$$

Where r and c represent the row index of current node and column index respectively (starting from 0).

Second, in our solution we find all the cycles that are present while passing messages diagonally. Take into account that we wrap the coordinates. This means that border elements also have diagonal neighbors. Left top diagonal neighbour is computed as $r_{lt} = r - 1$, $c_{lt} = c - 1$. For example, if $r = 0, c = 0$, then $r_{lt} = D - 1$, $c_{lt} = D - 1$. If we follow this rules when computing neighbours of border elements, we will find out that we have D cycles in our matrix, which is depicted with different colors on the Figure 3. In order to find to which cycle the element belongs to, we use the following equation:

$$Cycle = (\bar{r} + \bar{c})\%D$$

After we grouped all of our elements by cycles, we can determine the id of an element in a particular cycle by using the following equation:

$$Cycleid = \bar{c}\%D$$

In our picture the id of an element in a cycle group is represented by number (although our equation gives ids starting from zero, while on the picture it starts from one).

The number of elements in each cycle is always even, given the fact that D is even. Therefore, we can use the pairing technique that we used in the previous question to perform deadlock-safe message passing.