CmpE300 - Analysis of Algorithms
Fall 2021
MPI Programming Project

due: 17.01.2022 - 23:59

1 Introduction

In this project, you are going to experience parallel programming with C/C++/Python using MPI library. You will implement a parallel algorithm for a game called **"Lord of the Processors: The Two Towers"** which is a similar game to the combination of battleship and tower defense.

1.1 Cellular Automata

A cellular automaton is a discrete computational model used in the study of complex systems. It consists of an N-dimensional orthogonal grid called the "map", and rules of evolution. These alone define our complex environment. We then assign some initial values to the locations on the map, and make our system evolve by our set of rules via simulation:

- The map's state at time $t = 0$ is as initialized.
- To calculate the map's state at time $t = 1$, we look at the state at time $t = 0$. For each cell on the map at time $t = 1$, we look at the same cell and its neighbors at time $t = 0$.
- We keep on simulating like this until time $t = T$.

Now, you may think that this model is not complex at all, and you would be right! In analyzing complex systems, the key is to have a very simple model (i.e. a model with a very simple setup and rules of evolution), for an outcome that has a complex pattern. This will then allow the researcher to come up with simple explanations to the emergent patterns in a complex system.

2 Lord of the Processors: The Two Towers (LOTP: TTT)

Our focus will be on a particular cellular automaton which happens to be a game called Lord of the Processors: The Two Towers. In LOTP: TTT, we have a 2-dimensional orthogonal grid as a map (i.e. a matrix). Each cell on the map can either contain a certain type of tower (`'o'` or `'+'`) or be empty (`'.'`). The 8 cells that are immediately around a cell are considered as its neighbors. The following subsections explain the rules of the game.

2.1 Waves and Rounds

The game will be played for W waves. And each wave consists of 8 rounds. So the total game time will be $W \times 8$. At the start of each wave, new towers will be added to the map.

2.2 Towers

- Towers are the main items of the game.
- Towers have health points and attack powers.
- Towers can be placed at any empty place on the map.
- Towers attack their neighbours in each round simultaneously.
- If a tower is attacked, its health points are reduced.
- If a tower's health drops to 0 or below, then the tower gets destroyed.
- A tower can attack more than one tower in one round.
- A tower can be attacked by more than one tower in one round.
- Towers differentiate with each other by their attack patterns, health points and attack powers. There are two types of towers:
  - `‘o’` **tower:** Health Point: 6, Attack Power: 1, Attack pattern: Hits all its neighbors.
  - `‘+’` **tower:** Health Point: 8, Attack Power: 2, Attack pattern: Hits up, down, left and right neighbors like a plus (`‘+’`) shaped attack.
- A tower does not attack same type of towers. `‘+’` towers only attack `‘o’` towers, and `‘o’` towers only attack `‘+’` towers.

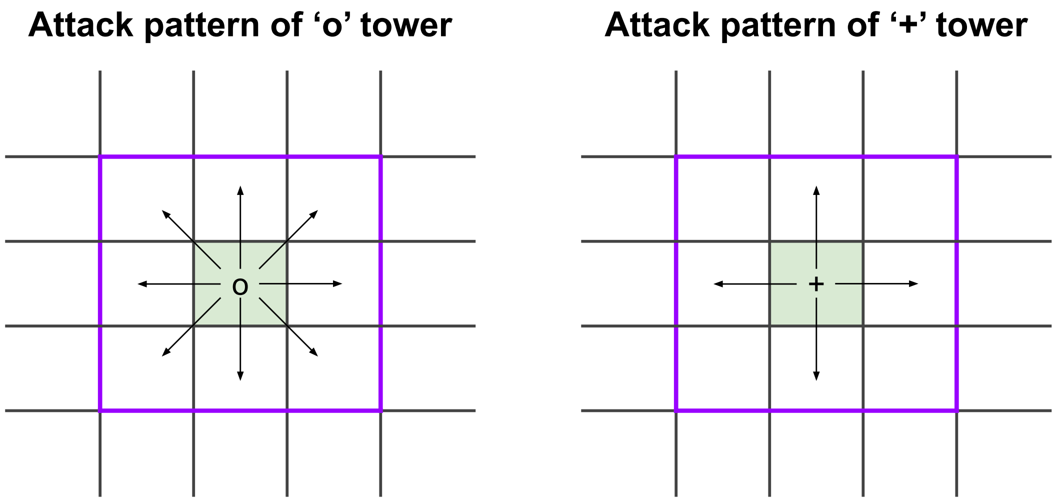

Figure 1: Attack patterns of `‘o’` tower and `‘+’` tower

2.3 Boundaries (Edges and Corners)

Since the cells at the boundaries do not have all the 8 neighbors, special considerations must be done for them. The missing neighbors are taken as `‘.’` i.e. empty. The corners have 3 neighbors, and the edges have 5 neighbors.

2.4 Game Progression

1. Initialize the map with empty cells. Use dot symbol (`'.'`) to remark empty cells on the map.
2. Iterate for W waves:
   (a) Put new towers to the map. Use tower signs to remark their places (`'o'` or `'+'` characters) on the map. If there is already a tower at the location where we want to put the new tower, then don't put the new tower and ignore it.
      - You can assume that, when placing towers for a new wave, there won't be any duplicate locations for the new towers. So, you can place the new towers in any order.

   (b) Iterate for 8 rounds:
      i. All the towers on the map attacks at the same time.
      ii. Reduce each tower's health points according to the total damage they received.
      iii. If a tower's health drops to 0 or below, then destroy the tower and make its location on the map as empty (`'.'`).
3. Print the final map.

Note: Health points of the towers are carried over to new rounds and new waves.

Note 2: The crucial part of the simulation is keeping track of the health of the towers. Because, health is the only variable that changes between rounds.

3 Parallel Simulation of LOTP: TTT

In this project, you are going to experience parallel programming with C/C++/Python using MPI library. Your task is to write an MPI program with 1 manager process and P worker processes. The manager will distribute the simulation task to the workers, and aggregate the results. The workers will, well, work on the task, and occasionally communicate with each other.

3.1 Compiling and Running

An MPI source code written in C Language, say `game.c`, can be compiled into the executable game with the following:

```
mpicc game.c -o game
```

For a C++ source code like `game.cpp`, use:

```
mpic++ game.cpp -o game
```

A compiled MPI program game, can be run with the following:

```
mpiexec -n [P] ./game input.txt output.txt
```

For a python code like `game.py`, you don't need to compile it, you directly run it with the following:

```
mpiexec -n [P] python game.py input.txt output.txt
```

`[P]` is the number of processes to run game on. If you want to have P = 8 worker processes, then you need 9 processes in total, accounting for the manager. Hence, you should write `-n 9` in the command line. Arguments `input.txt` and `output.txt` are passed onto game as command line arguments.

3.2 Input and Output Files

The `input.txt` will contain the size of the map (N), number of waves (W), the number of towers to be placed for each tower type per wave (T) and the coordinates of towers for each wave. Top left coordinate of the map is $(0, 0)$ and bottom right coordinate of the map is $(n-1, n-1)$. There will be T number of '`o`' towers and T number of '`+`' towers added to map per wave (if possible). Here is the line by line explanation of `input.txt`:

- First line consists of three integers separated by a space: $N\ W\ T$
- Coordinates of '`o`' towers for wave 1 separated by space: $co^1_{11}\ co^1_{12},\ co^1_{21}\ co^1_{22},\ co^1_{31}\ co^1_{32},\ \ldots,\ co^1_{T1}\ co^1_{T2}$
- Coordinates of '`+`' towers for wave 1 separated by space: $cp^1_{11}\ cp^1_{12},\ cp^1_{21}\ cp^1_{22},\ cp^1_{31}\ cp_{32},\ \ldots,\ cp^1_{T1}\ cp^1_{T2}$
- Coordinates of '`o`' towers for wave 2 separated by space: $co^2_{11}\ co^2_{12},\ co^2_{21}\ co^2_{22},\ co^2_{31}\ co^2_{32},\ \ldots,\ co^2_{T1}\ co^2_{T2}$
- Coordinates of '`+`' towers for wave 2 separated by space: $cp^2_{11}\ cp^2_{12},\ cp^2_{21}\ cp^2_{22},\ cp^2_{31}\ cp_{32},\ \ldots,\ cp^2_{T1}\ cp^2_{T2}$
- ...
- Coordinates of '`o`' towers for wave W separated by space: $co^W_{11}\ co^W_{12},\ co^W_{21}\ co^W_{22},\ co^W_{31}\ co^W_{32},\ \ldots,\ co^W_{T1}\ co^W_{T2}$
- Coordinates of '`+`' towers for wave W separated by space: $cp^W_{11}\ cp^W_{12},\ cp^W_{21}\ cp^W_{22},\ cp^W_{31}\ cp^W_{32},\ \ldots,\ cp^W_{T1}\ cp^W_{T2}$

The `output.txt` should be filled with the map's final state after W waves of simulation. There should be dot character '`.`' for the empty cells, little '`o`' for the '`o`' towers and plus sign for the '`+`' towers (without quotes) separated by one space. Example input-output files will be provided.

3.3 The Job of the Manager and Worker Processes

MPI is flexible, meaning that it is designed to be able to accommodate parallel computation models other than just the manager-worker model. Hence, it will not designate a process as the `manager` by itself. Instead, you should regard the rank-zero process as the manager yourself in your MPI code.

The job of the manager (rank = 0) process:

1. Reading input from `input.txt`.
2. Splitting the coordinates into completely separate (i.e. disjoint) parts and sending them to the workers. A coordinate should not be sent to more than one processor.
3. Receiving the results from the workers, and their aggregation.
4. Writing the final map to `output.txt`.

The job of the worker (rank $= 1,\ \ldots,\ P$**) processes:**

1. Receiving the data from the manager.
2. Carrying out the simulation. In the mean time, communicating with the other workers when necessary.
   - Each worker process should keep track of the health and the location of its towers.
3. Sending the data back to manager after W waves.

Note: You can send the coordinates for all the waves from the manager to workers at once or you can send the coordinates for each wave at the beginning of each wave.

Note 2: The worker processes should not make any file input/output.

3.4 Splits

There are many ways to split and distribute a $N \times N$ map to the workers. We will only consider two different approaches. We will assume that the map is a square of size $N \times N$. There will be 1 manager and P worker processors and we will assume that N is divisible by the number of worker processors P.

1. **Striped:** In the first approach, each processor is responsible from a group of N/P adjacent rows. Each processor works on $(N/P \times N)$ cells and these cells are stored locally by the processor. When a cell is inspected, the processor should inspect all of its neighbors. When the cell is not on the boundary of two adjacent processors, information about the neighbor cells are present to the processor. When a cell on the boundary is inspected, the processor needs to obtain information from the adjacent processor. Consider the black cell in the figure below. Processor 1 should communicate with Processor 2 in order to learn the status of the south neighbor of the black cell. Therefore, each processor should communicate with the adjacent processor at every round. Information about those neighbor cells of the boundary can be stored locally and updated at every round.

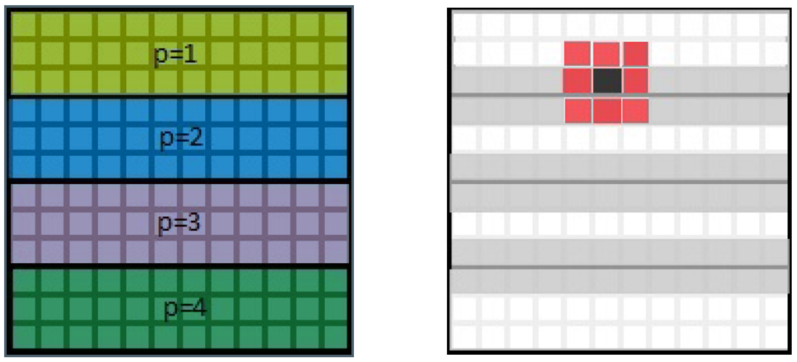

Figure 2: Striped split.

2. **Checkered (25 pts bonus):** In the second approach, the map is divided into $(N/P \times N/P)$ blocks and each process is responsible from a block. Now the updates are more trickier since each process has more than 1 adjacent process. If you choose this one, we will be testing your implementation with a P that is a perfect square, say $P = x^2$.

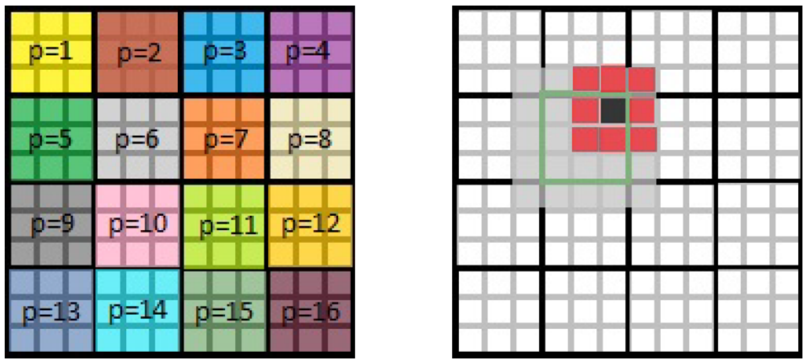

Figure 3: Checkered split.

Note: In your project, please pick either the striped- or checkered-split, and implement only that. In other words, do not implement both of them.

Note 2: *P* itself will always be an even number, which should help you in implementing the communication phase efficiently.

Warning: Splitting the map checkered is not much harder than splitting it striped. However, the checkered-split does make the communication phase very much harder! Please be aware of that before choosing to implement the checkered-split.

4 Communication Between Workers

After the split, each process will have a separate part of the map. To calculate the next state of some cells, they will need some information from each other.

When simulating the game, it is much easier to go over how much damage each tower takes than it is to go over who each tower damages.

For example, see the black-cell in the striped-split example in Figure 2. Assume that, there is a `'+'` tower at that location. It can only be attacked by `'o'` towers. Because of the attack pattern of `'o'` towers, we should check every neighbor of the black-cell. We can easily check if there is any `'o'` towers at left, right and upper 3 neighbor cells. However, lower 3 cells belong to rank 2 Process. Then, rank 2 Process should send those cells to rank 1 process. After receiving this information, rank 1 Process can count the number of attackers to `'+'` tower at black-cell and reduce its health accordingly.

It is obvious that, exchanging only 3 cells when it is needed is very inefficient. So, if you consider all the boundary cells within a part of the map:

- With the **striped-split**, each process will have to send their entire top/bottom row of their part of the map to the process above/below. On the other hand, each process will also have to receive the entire bottom/top row of the part of the map belonging to the process above/below, respectively.
- With the **checkered-split**, each process will have to make the similar exchanges as in the striped-split case, but also with their entire leftmost and rightmost columns, and with the processes on the left and on the right. Moreover, each process will have to send the corners of their part of the map to the process that is above-left, above-right, etc. On the other hand, each process will also have to receive the bottom-right, etc. corners of the part of the map belonging to the process that is above-left, etc.

This exchange of information will have to be done once in the beginning of every round of the simulation.

4.1 Deadlocks

Beware! You can easily have deadlocks, if you do not plan your communication structure well.

MPI does not come with spooling[1] capabilities. This means that the sender waits until the receiver receives, and the receiver waits until the sender sends.

In particular, functions `MPI_Send` and `MPI_Recv`, used in sending/receiving data to/from another process, blocks the execution from continuing until that process receives/sends the data. If two processes were to mutually attempt the same action, they would be waiting eternally.

[1]Queueing of sent information at bay, allowing recipient to receive/process them at a later time.

4.2 MPI Functions

Get familiar with the MPI framework. You will have to know about the following functions at the very least. These are also enough to complete this project:

- `MPI_Init` and `MPI_Finalize`
- `MPI_Comm_rank` and `MPI_Comm_size`
- `MPI_Send` and `MPI_Recv`

There are different "send" functions available in the MPI documentation. However, for this project, you are not allowed to use any other function than `MPI_Send` and `MPI_Recv` to send and receive messages between processes. Using other variants (such as `MPI_Isend`) is prohibited!

4.3 Performance

Your algorithm must be parallel and performant. We will readily tell you how you can achieve the required performance by the end of this section. Please implement your code accordingly, and otherwise, you may lose some points. Some communication schemes are more efficient than the others. Considering the striped-split's example, an example communication scheme that implements periodic boundaries without causing any deadlocks would be:

- Process 1 sends its bottom row to process 2.
- Process 2 receives from process 1. Then, sends its bottom row to process 3.
- Process 3 receives from process 2. Then, sends its bottom row to process 4.
- Process 4 receives from process 3.

Unfortunately, with this scheme, there is a waiting chain of 3 communications for the process 4 to receive its information. Not just the process with rank 4, but also any other process x will have to wait for the previous $x-1$ communications to be completed, before receiving the information.

We can say that this communication scheme performs in $O(P \cdot \sqrt{n})$ time, where n is the number of cells on the map. This means that each added worker comes with a time penalty.

Now, note that the following naive "simultaneous send" communication scheme would not be better, and in fact it can cause a deadlock, and therefore would not work:

- Every process sends its bottom row to the process below. Every process then waits to receive a row of information from the process above.

Fortunately, there still is a way to achieve $O(\sqrt{n})$ time for the communications. We cannot make all of our processes send at once; but we can make every other process send, while the remaining ones are waiting to receive. Since we are assuming that P (or $\sqrt{P}$ in the checkered-split) will be even, we can partition our processes as even/odd by their ranks, and establish the following scheme:

- Every process with an odd rank sends its bottom row to the process below. Then, they wait to receive a row of information from the process above.
- Every process with an even rank receives a row of information from the process above. Then, they send their bottom row to the process below.

It is very much important to us for you to notice the performance problem in the sequential scheme, the deadlock problem in the "simultaneous send" scheme, and how the even-odd scheme improves and achieves parallelism. You are welcome to implement your own ideas that runs as efficient. On the other hand, you will lose some points for implementing an inefficient communication scheme.

5 Submission

1. The deadline for submitting the project is 17.01.2022 - 23:59. The deadline is strict.
2. Write a project report with the following sections:
   (a) **Introduction:** Have you completed the project? Is your code running successfully?
   (b) **Structure of the Implementation:** You should talk about general progression of your implementation. How your processors communicate. How did you split the processors (Striped or Checkered) Your reasonings behind your decisions. Any assumptions you made. This is the most important section.
   (c) **Analysis of the Implementation:** Make an analysis of your implementation in terms of time complexity. Especially touch on the communication between processors. What slows down/ speeds up the implementation most?
   (d) **Test Outputs:** Put the outputs of the test inputs that we have provided to you.
   (e) **Difficulties Encountered and Conclusion**
3. Comment your names, your student ids, compilation status (Compiling/Not Compiling), working status (Working/Not working) and any additional notes that could be helpful in the grading process at the beginning of your main source code file.
4. This is a group project. Your code should be original. Any similarities between submitted projects or to a source from the web will be accepted as cheating.
5. Your code should be self explanatory by means of either choosing self-explanatory variable and function names or explicit comments.
6. Follow the good programming practices you have learned in your previous courses. Use functions and loops to prevent/eliminate duplicate patterns in your code.
7. Submit your deliverable as a zip file that includes your report and implementation codes through Moodle,
8. If you have any further questions, send an e-mail to *burak.suyunu@boun.edu.tr*