# CS3210 Assignment 2 Report

*Nguyen Viet Dung – A0112068N*

# I. Design

The world is a matrix of size **N* N (N~3000)**. The pattern size is **K * K (K={3, 5})**. The world will change from time to time and we have to find all the positions that match the pattern. We are now to design a parallel system to deal with this problem.

### 1. Partitioning

Let **P** be the number of processes. We already know some common ways to partition a square matrix: by rows, by columns or by squares of some size $\frac{N}{\sqrt{P}}$. In any case, the world size will reduce **P** times. However, in this specific problem, processes are not independent as each cell will change basing on its 8 neighbors. Hence each process may need extra data from its neighbor processes:

- For dividing by rows, each process has at most 2 neighbors.
- For dividing by columns, each process has at most 2 neighbors.
- For dividing by squares, each process has at most 8 neighbors.

Remember that MPI_Send is only able of sending contiguous data, that is by rows (as designed in sequential SETL). So if we divide by columns or squares, we may need a lot more effort to transfer data to contiguous for sending. Hence dividing by rows seems to be the best solution in this case.

### 2. Communication

Let **numRows = N/P**. Process 0 will now cover rows [1 ... numRows]; process 1 will cover rows [numRows+1 ... numRows*2]; so on.

Now for process **i**, we need the last row of process **(i-1)** for calculating the next generation; we also need the first **(K-1)** rows of process **(i+1)** for calculating the next generation and checking for the pattern. Hence at the end of one generation, each process must send necessary data to its neighbors, to prepare for the next generation. There are 2 possible approaches:

- At the end of one generation, each process sends **K** rows to its 2 neighbors (except for the first and the last processes).
- Each process maintains extra **(K-1)** rows for checking pattern. Now at the end of one generation, a process only need to send 2 rows to its neighbors, for calculating the next generation.

Actually **K** is very small, communication cost is also small as nodes are in the same cluster. I did a quick check on node 1 (core i7, 8 cores) with 8 processes, world size 3000, pattern size 5 and 100 iterations. The first approach took **23.56s** and the second approach took **23.49s**. They are mostly the same. However, approach 1 may be troublesome when **K=N**, so that one process may not have enough **K** rows to send.

Therefore, I will stick with approach 2.

### 3. Data Gathering & Printing

Now each process has a list of results at a specific range of rows. Our last thing to do is to print all the results in order **(Generation, Direction/Rotation, Coordinate)**. We may come up with 2 approaches:

- Synchronous printing.
- Send everything to the root & print at the root.

For the first approach, each process has a list of results in order **(generation, direction, coordinate)** already. So we only need to make a for-loop of **(generation, direction)** and print in each process, in order of their ranks. Unfortunately, original MPI does not support such a thing. There are some tricks to do this with **MPI_Barrier** and a small **delay time**. However, there is nothing to make sure of the correct order; as **printf** is not scheduled by MPI. (Without the delay time, I made it works for >= 4 processes but not for 2 processes.)

Hence, I stick with the second approach, to print everything at the root. There are also some more things to consider with this approach. The question is how large the result can be and need to transferred to the root process. Consider the worst case: **3000*3000** world, **5*5** pattern of all dead cells and **100** generations. The world keeps the same for all generations. The number of matching patterns is about **3000*3000*4 directions*100 generations**. In each matching pattern, we also need to store the iteration, the direction and the coordinates **(~4*4=16 bytes).** Hence totally that is about **57.6 GB**. We of course cannot afford this number.

Remember that the result is discovered in order of **(generation, direction, coordinate)** in each process. So we can just store the result in each generation; then at the end of one generation, we send everything to the root for printing and reset the result. By this way, we do a tradeoff between communication rate and number of data stored. Anyway the communication is fast inside the device/cluster.

### 4. Number of Processes

First we follow the ideal condition, that is 1 process per core.

This algorithm is designed to be run with any number of processes (still should be smaller than **N**, or there will be redundant processes). So on NSCC, we take the maximum benefit with **48 cores (48 processes)** on dev-queue (still can be better with more cores).

For the 3-nodes cluster, we have one intel i7 with 8 cores, intel i5 with 4 cores and one Jetson with 1 core. A quick addition gives us 13 cores/processes totally. However, processes are dependent of each other (all processes need to finish generation i before going to generation i+1). The speeds of nodes are different and may increase the waiting time. Following is an experiment on 3 nodes with 1 process, world size 3000, pattern size 5 and 10 generations:

| Node | Time (seconds) |
|---|---|
| Node1 (intel i7) | 10.80 |
| Node2 (intel i5) | 12.42 |
| Jetson | 32.37 |

From the table, we can see that **node 1** is slightly faster than **node 2** and both are much faster than **Jetson**. Hence **Jetson** will probably slow down the system. To make sure, I also did

another experiment on different set of nodes, world size 3000, pattern size 5, 100 generations and following is the result:

| Setup (1 core = 1 process) | Time (seconds) |
|---|---|
| 8 cores from node 1 | 27.90 |
| 8 cores from node 1 + 4 cores from node 2 | 18.67 |
| 8 cores from node 1 + 4 cores from node 2 + 1 core from Jetson | 24.29 |

From the tables, we see that node 2 will support node 1 the best (as node 1 is the fastest) and Jetson slows down the work. Actually we still can take the advantage of 1 core from Jetson by giving it the work equals 1/3 work of others (as it is 3 times slower). However, it will remove the generality of the solution.

Therefore, I stick with 12 cores system (1 process each, 8 from node 1 + 4 from node 2).

# II. Evaluation

### 1. Experiments

We are going to do the experiments and evaluations on a world of size **3000 * 3000**, with pattern size **5 * 5** and **100** generations.

First, we have a quick estimation: a cell will be a live cell in the next generation only if 2 or 3 over 8 of its neighbors are live cells, hence the number of live cells is about 25% of the world. This estimation is of course not 100% correct. Actually I did a quick check on this, from the seconds generation, the average percentage of live cells is only about 20% (except for very-low-percentage-live-cells words or very-high-percentage-dead-cells words, all cells will become dead after some generation).

Hence the initialized percentage of live cells on the world does not matter much, as it will be normalized after some generations. So I will random some worlds with initial percentages of live cells are **10%, 50%** and **90%**.

The distribution of live cells somehow tells us about the form of pattern. If the live percentage of pattern is too high, it hardly matches the world of about only 20% of live cells. By experiments, with patterns of more than **30%** live cells, the matching ones appear rarely; with patterns of more than **50%** live cells, the matching ones mostly come from the first generation.

For the first experiment, I focus on the speedup of the parallel version. Hence, for the best evaluation, we only consider patterns with initial percentages of live cells are **10%, 15%, 20%, 25%, 30%** (too high percentage will give empty/very few result & spoil the average execution time; too low percentage will give too many results that the sequential SETL will run out of memory, so we cannot calculate the speedup of the parallel version. There will be another experiment for the executable of the parallel version.) So we will do experiments on different of percentages of live cells on both world and pattern. For each pair of values, we will do 5 experiments and calculate the averages. Following are the experiment results:

| | World 10% live | | | World 50% live | | | World 90% live | | |
|---|---|---|---|---|---|---|---|---|---|
| Pattern | seq | par | speedup | seq | par | speedup | seq | par | speedup |
| 10% | 125.58 | 20.97 | 5.99 | 114.15 | 18.32 | 6.23 | 131.39 | 22.09 | 5.95 |
| 15% | 98.05 | 16.11 | 6.09 | 93.96 | 14.88 | 6.31 | 101.48 | 16.76 | 6.05 |
| 20% | 96.32 | 16.46 | 5.85 | 94.76 | 14.53 | 6.52 | 98.89 | 16.35 | 6.05 |
| 25% | 84.52 | 13.86 | 6.10 | 84.57 | 13.19 | 6.41 | 85.97 | 14.11 | 6.09 |
| 30% | 64.48 | 10.95 | 5.89 | 70.33 | 10.85 | 6.48 | 63.37 | 10.06 | 6.30 |

*Table1. Experiments on the 3-nodes cluster with 12 cores (12 processes)*

| | World 10% live | | | World 50% live | | | World 90% live | | |
|---|---|---|---|---|---|---|---|---|---|
| Pattern | seq | par | speedup | seq | par | speedup | seq | par | speedup |
| 10% | 70.74 | 2.04 | 34.68 | 74.03 | 1.95 | 37.96 | 68.96 | 1.91 | 36.10 |
| 15% | 70.33 | 1.9 | 37.02 | 74.12 | 1.93 | 38.40 | 69 | 1.94 | 35.57 |
| 20% | 65.66 | 1.78 | 36.89 | 69.84 | 1.83 | 38.16 | 64.66 | 1.82 | 35.53 |
| 25% | 65.92 | 1.78 | 37.03 | 69.7 | 1.84 | 37.88 | 64.32 | 1.81 | 35.54 |
| 30% | 65.87 | 1.81 | 36.39 | 69.56 | 1.81 | 38.43 | 64.52 | 1.81 | 35.65 |

*Table2. Experiments on the NSCC with 48 cores (48 processes)*

For the second experiment, we examine the executable of the parallel version. There are some test cases that give very large results such as all-dead world and all-dead pattern (which can output up to 57.6GB of result as discussed above). The sequential SETL version of course cannot handle such cases as the memory is too large. In this experiment, we will do with a world size **3000*3000** with live-percent **0%** or **5%**, a pattern size **5 * 5** with **all dead cells** (so that this will maximize the number of results in each generation), and **10** generations only (to limit the data to write). Following is the results on NSCC with 2 nodes, 1GB memory and 24 cores on each node (As this is for checking the executable only, it is not necessary to run on the lab machines, so that we can speed up the experiment).

| World live cells | SETL_seq on NSCC | SETL_par on NSCC | Size of output file |
|---|---|---|---|
| 0% live-cells | Out of Mem | 592.63 sec | 4.5 GB |
| 5% live-cells | Out of Mem | 501.38 sec | 4.0 GB |

*Table3. Experiments on NSCC with 48 cores (48 processes)*

## 2. Discussions

From table 1 and table 2, we can see that both sequential and parallel versions give a slightly better time with increasing live-percent of the pattern. Running time with 30% live-cell pattern is much faster than the one with 10% live-cell pattern. This implies that higher live-percent gives less matching patterns, as the number of live cells in the world is only about 10 – 20% after some generations (as discussed above); so there are less result to process with higer live-percent of pattern.

The parallel version is running clearly faster on both lab machines and NSCC. For 3-nodes cluster with 12 processes, the speedup is about 6 times faster. So this is about 50% of the ideal speedup (12 times faster). This is quite good, as we have a lot of cost for

communication (after each generation, a process need to send data at least 5 times to neighbors/root). Moreover the speed of processes on node 1 and node 2 are different (node 1 is slightly faster than node 2); processes are dependent on their neighbors so we also have some waiting time here.

For NSCC, the speedup is about 35 – 38 times faster with 48 processes. This is very impressive with about 75% of the ideal speedup (48 times faster). This is because we have 48 processes of the same speed, the communication between processes is also fast as of the super computer.

From table 3, we can see it took a lot of times (about 5-6 mins) to complete. This is much slower than the results in experiment 1 (only 1-2 seconds). This is because there are a lot of matching patterns. We can see that the output file is about above **4GB**. This is large so that the sequential version cannot handle and run out of memory. Anyway, this proves the parallel version can handle the cases with very large number of results which the sequential version cannot.