## Parallel-Processing-Assignment-2

I've coded 4 different modes:

- Serial mode -> serial.c
- Parallel mode (Option 1) -> parallel\_v1.c Uses OpenMP static scheduling
- Parallel mode (Option 2) -> parallel\_v2.c Uses OpenMP dynamic scheduling
- Parallel mode (Option 3) -> parallel\_v3.c Uses OpenMP dynamic scheduling

In the experiments, I've repeated every mode 10 times and take the average of the execution times. The results are in Results.xlsx file.

For small N value, serial mode has better timing performance. BUT when N=1000, parallel modes improves the performance. Parallel\_v2 has the best timing for large iteration counts. Dynamic scheduling is faster than static ones. Parallelize random initialization of the matrix has no positive effect.

For execution, please run:

```
.\playGame.sh
```

The above sh file is for MacOS. For Unix environments, you can use:

```
#!/bin/bash
gcc -o playGame -fopenmp main.c
export OMP_NUM_THREADS=4
./playGame
```

Samples of threads assignments for playGame method: (N=4) - Serial

• Parallel v1

```
0
        0
                 0
                         0
1
        1
                1
                        1
2
        2
                2
                        2
3
        3
                3
                        3
```

• Parallel v2

```
1 1 1 1
1 1 1 1
0 0 0 0
2 2 2 2
```

• Parallel v3

```
    1
    1
    1
    1

    2
    2
    2
    2

    1
    1
    1
    1

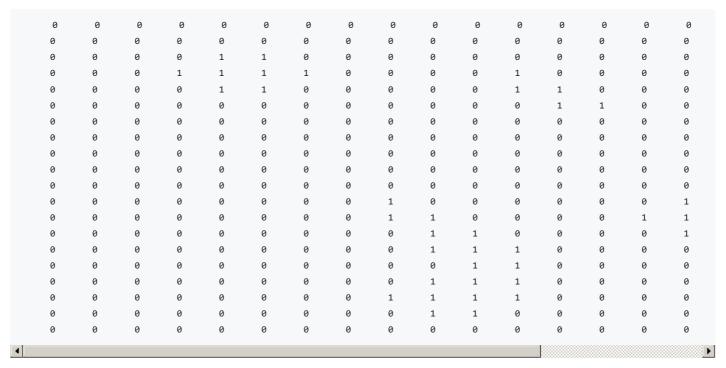
    1
    1
    1
    1
```

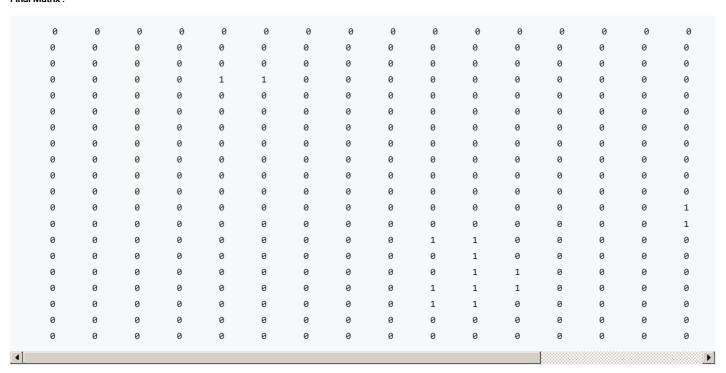
A sample output of Parallel v2 option: Program started with 4 threads and 10 iterations. After some iterations, the matrix become a sparse matrix that has lots of 0s.

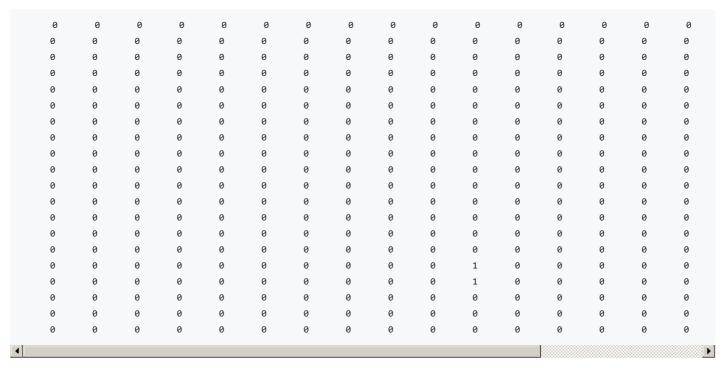
Initial Matrix

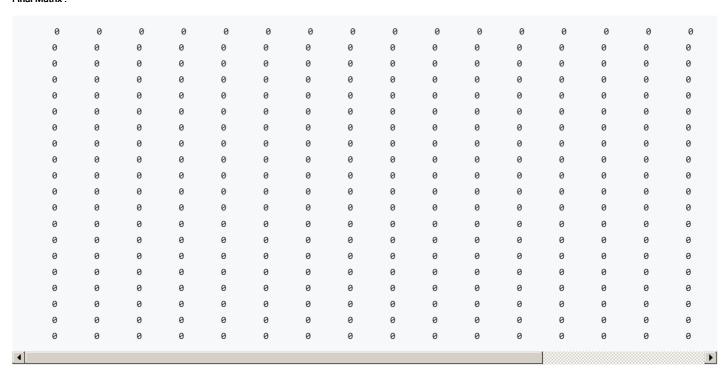
1	1	1	0	0	0	0	0	1	1	0	1	0	0	1	1
1	1	1	1	0	1	1	0	0	1	1	0	0	1	0	1
1	0	1	0	1	1	0	1	0	1	1	1	0	0	0	0
0	0	1	0	1	1	1	0	0	1	0	1	1	1	1	0
1	0	1	1	1	1	1	0	1	0	0	1	1	0	0	0
1	1	1	0	1	0	1	1	0	0	1	1	1	1	1	0
0	1	0	0	1	0	1	0	0	0	0	1	0	0	1	1
0	1	0	1	0	0	0	1	1	0	0	0	1	1	0	1
1	0	1	0	0	0	0	0	0	0	0	0	1	0	0	0
1	1	0	1	1	1	0	1	1	0	0	0	1	1	1	0
0	1	0	0	0	1	1	1	1	1	1	1	1	0	0	1
1	0	1	0	1	1	0	1	0	1	0	0	0	0	1	1
0	1	1	0	1	0	0	0	1	1	0	0	0	1	1	1
1	1	1	0	1	1	0	0	1	1	1	1	0	0	0	1
0	0	0	1	0	1	0	1	0	1	1	1	1	1	1	1
1	0	0	0	0	0	0	1	0	0	1	1	1	0	0	0
1	1	1	1	0	0	1	1	1	0	1	1	1	0	0	0
0	0	1	0	1	0	0	0	1	1	1	0	1	0	0	0
1	0	1	1	1	1	1	1	1	1	1	1	0	1	0	1
1	1	1	0	0	1	0	1	0	0	1	0	1	0	0	0
												#n			
4															<u>}</u>

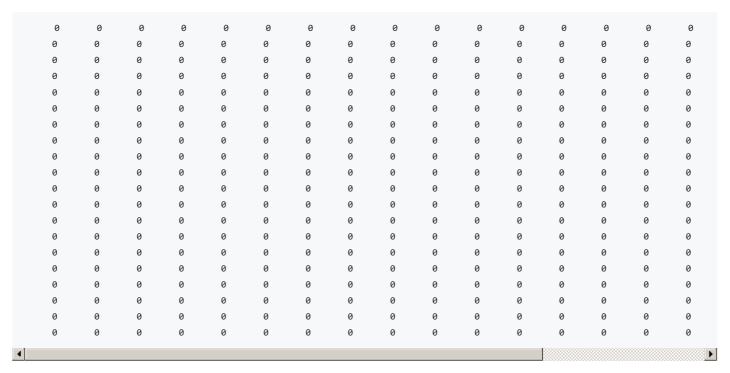
e	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	0
0	0	0	0	0	1	0	0	0	1	1	1	1	0	1	0
0	0	0	1	1	1	0	0	0	1	1	1	1	0	0	0
0	0	1	1	1	0	0	0	0	1	1	1	1	0	1	0
0	0	0	1	1	0	0	0	0	0	1	0	1	0	1	0
1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
6	0	0	0	0	0	0	1	1	1	1	1	0	0	0	0
1	1	0	0	0	0	1	1	0	0	0	0	0	0	0	0
1	1	0	0	0	0	1	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	1	1	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	1	1	0	0	0	1	1	1	0
0	0	0	0	1	1	1	1	0	0	0	0	1	0	1	0
6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

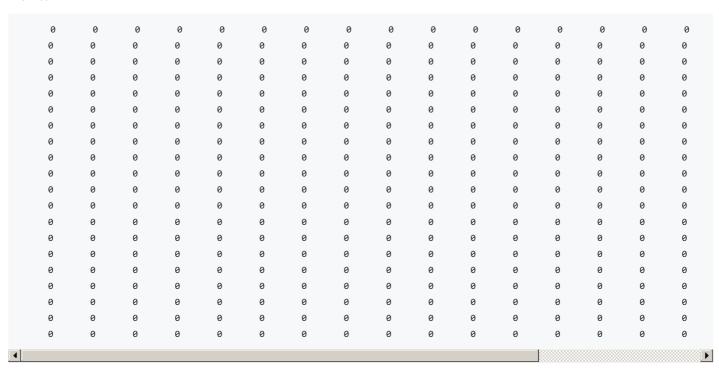


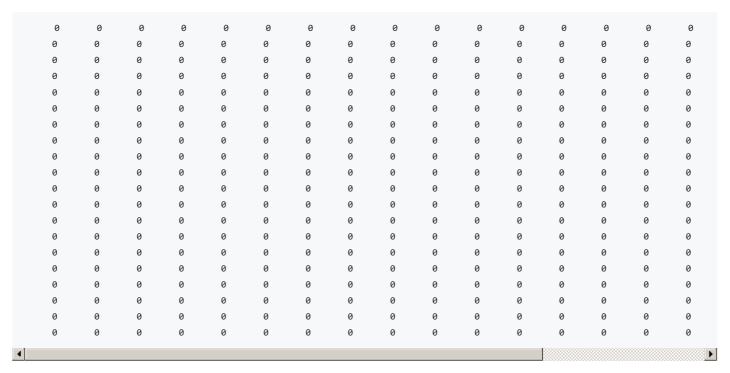


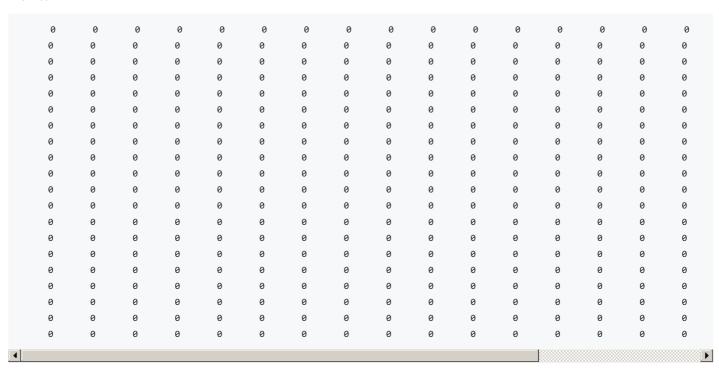












0	0														
		0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Program finished. Total execution time: 0.305444