To compile the code:

1d version:

make 1d

2d version:

make 2d

2d version data generator:

make inputgen

There is a data generation file to convert the x, y coordinates info to a grid.

To run the code,

1d version:

mpirun -np <number of processes> ./a.out <input file name> <# of generations> <X_limit> <Y_limit>

The output file name is <input file name>.csv, e.g., life.1.250x250.data.csv

2d version:

Run data generator first:

./a.out <data name> <X_limit> <Y_limit>

e.g., ./a.out life.1.250x250.data 250 250

The output will be <data name>.txt, e.g., life.1.250x250.data.txt

Then run 2d mpi:

mpirun -np <number of processes> ./a.out <generated input file name> <# of generations> <X_limit> <Y_limit>

e.g., mpirun -np 4 ./a.out life.1.250x250.data.txt 100 250 250

The output file name is life.2d.csv

Experiments:

Input: final.500*500.data, 500 steps, 500*500 grid

Version	Time Min	Time Avg	Avg Max
1d - 1 processes	10.4508	10.4508	10.4508

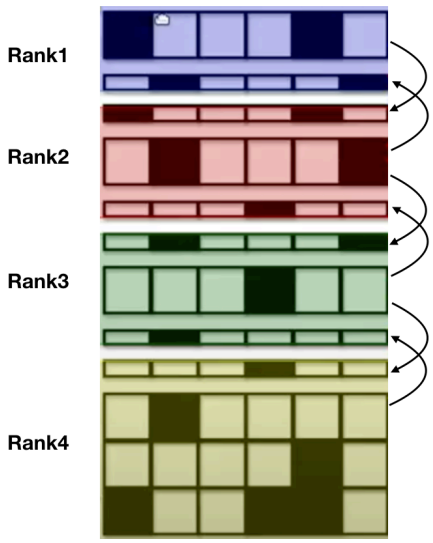
2d - 1 processes	0.751583	0.751583	0.751583
1d - 4 processes	3.18594	3.18673	3.18765
2d - 4 processes	0.268466	0.269179	0.269776
1d - 8 processes	1.8284	1.83013	1.83158
1d - 16 processes --ntasks-per-node=16	1.18584	1.18767	1.18976
2d - 16 processes --ntasks-per-node=16	0.097235	0.098428	0.099325
1d - 16 processes --ntasks-per-node=8	1.21805	1.22031	1.22272
2d - 16 processes --ntasks-per-node=8	0.102047	0.103292	0.104225

The 2d version runs significantly faster than 1d version. I think it's because 2d version is written in C while 1d is in C++. A more advanced partition in 2d also helps to further improve the speed up the process with more processes. I did some value by value swap in vectors in 1d version, which may slow down the procedure a lot.

With more processes, the time per processes is almost cut to 1 process time/# processes, that's is the ideal case. But time also spends on transferring data. Speed of 16 processes is definitely better than 4 in both of these two versions.

Less tasks per node slightly slow down the process. Maybe the data transfer across nodes costs more than within a node. It really depends on how much data is across the nodes.

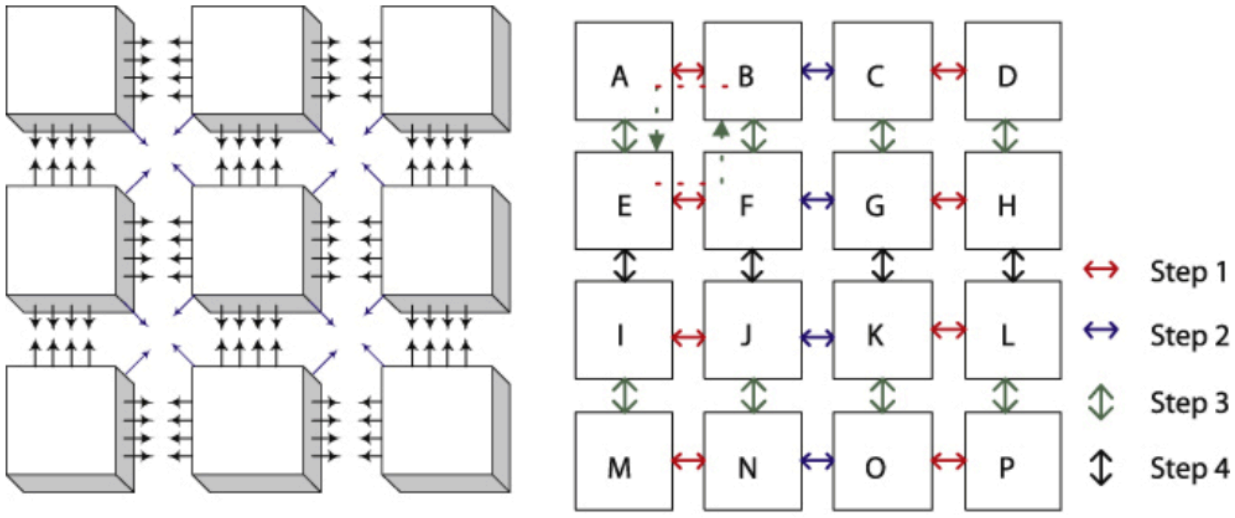
Decompose in 1d version:



The universal grid is divided into subgrids to each processes along the rows. For example, if there are 8 rows and 4 processes, each subgrid will have 2 rows. But if 9 rows totally, the last row will have 3 rows. Data that are in the first and the last row are send to their neighbors.

Data distribution: data is firstly read by the root process and stored in a 2-d vector. Then data are sent to each processes to initialize their local grid. Then for each iteration, info of first and last row of subgrids are transferred. After the loop ends, data is sent back to root rank to output.

2d version:



Global grids are divided into sqrt(nRows*nCols/mpiSize) sub blocks(grids). Each middle block has south, north, east, west neighbors to transfer an edge, and have four corner neighbors to transfer only a single corner token. While each edge block and corner blocks are more special. They follows the arrows to transfer data on the right side. In each iteration, update function is called only if all neighbor data is received.