# OpenMP and MPI Assignment

Qingxian Lu(14723697), Ingrid Marie Wølneberg(14580934),Yongqing Liang(14115603)

January 30, 2023

## 1 Summary

We were tasked with implementing Conway's game of life in C on a finite grid with a border of dead cells. Then we should parallelize the implementation with OpenMP and/or MPI to speedup our code.

In this report we first talk about our motivation for how we used MPI and OpenMP, then we shortly discuss the performance of our implementation. Finally we have the code documentation containing our full commented implementation and the script we used to run it on Lisa.

As a result, after running the grower pattern for 5000 generations, the population size was **3647**.

However, due to time constraints (because Lisa was heavily used by many people and jobs were pending), the data was not fully run for 5000 generations, but a more detailed discussion was carried out at generations=100.

Based on the discussions in latter sections, it can be concluded that increasing the number of tasks from 1 to 4 after 5000 generations resulted in a speed-up index of **12.7**. With 100 generations, the speed-up index was **39.6**.

## 2 Motivation

### 2.1 MPI

In the figure 1, suppose there are 6 rows of work (a, b, c, d, e, f) and two processes. Each process is assigned three rows of work. From the rules of the game of life, it can be known that a cell's value depends only on the values of the surrounding eight cells. Therefore, for the right side of the figure, the value of line c depends on line b (known in this process) and line d, and the value of line d depends on line c and line e (also known in this process). By exchanging one line of data between process 1 and process 2, each can iterate as required. During the iteration, each process can iterate after obtaining the necessary lines without communication with the master node. (Only the map is scattered from the host at the beginning, and the results are gathered to the host at the end to count the final survivors.)

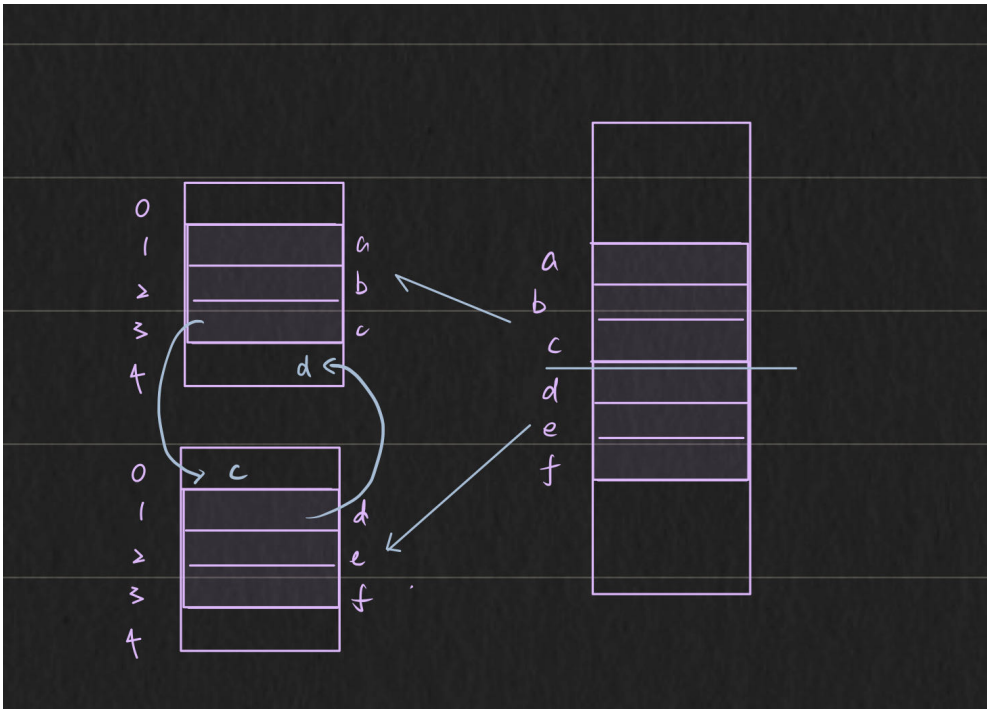


Figure 1: Split method

### 2.2 OpenMP

We started by first creating a program for game of life and noticed we had a lot of for loops in our code. These could be distributed and run in parallel. We therefore decided to use OpenMP for counting the live cells and calculating the next generation.

## 3 Discussion

We realized later on that our bath script should have been changed a bit to run on more nodes. Therefore we made some changes to our script to see how the performance would be then. Unfortunately we realized this too late, because when we wanted to run the new script on Monday ,Lisa was very busy and we did not manage to run it in time. So we can just analysis the result based on the data we have already had, so maybe not so sufficient.

The data collected shows the acceleration effect we have achieved. As shown in the Fig 2, when there is 1 node and each process has 4 CPUs, the acceleration effect increases as the number of processes increases. Specifically, when there is 1 process, the time required is approximately 1580s, while when there are 4 processes, the time required is approximately 124s, resulting in a 12.7-fold acceleration. This acceleration is based on MPI since the number of CPUs per process was kept constant. We believe that with more nodes and more CPUs per process, our program can achieve a 17-fold acceleration.

Due to time constraints as mentioned before, we ran the data with 100 generations in more details. Noticed that the first red dot represents the scenario where nodes=1, task=1, and cpu=1 is run as

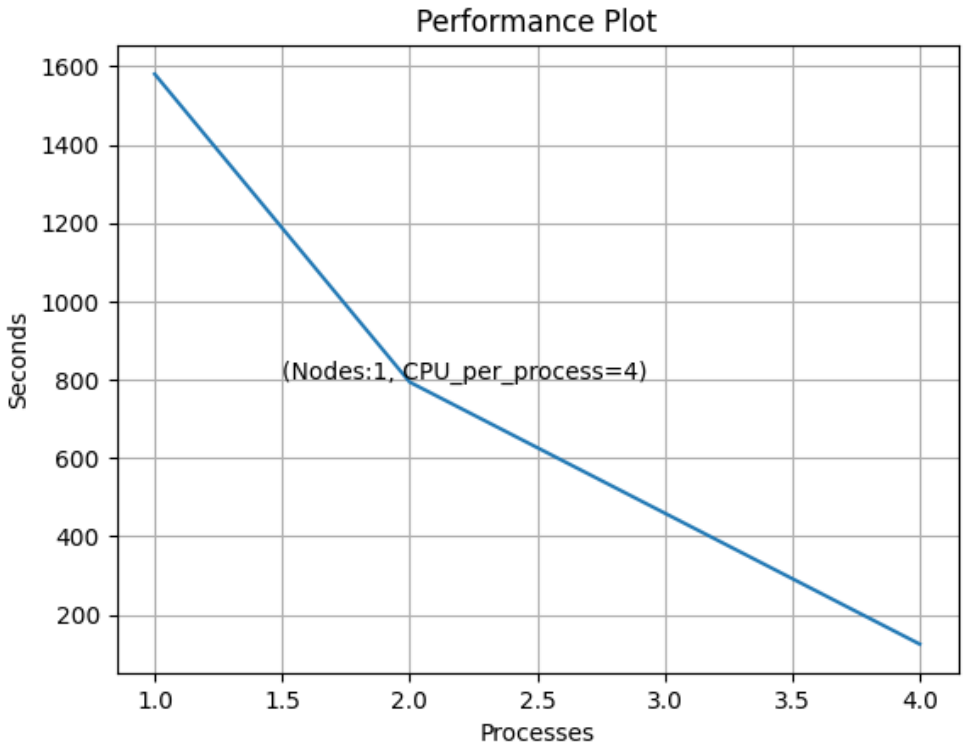


Figure 2: Experiment data

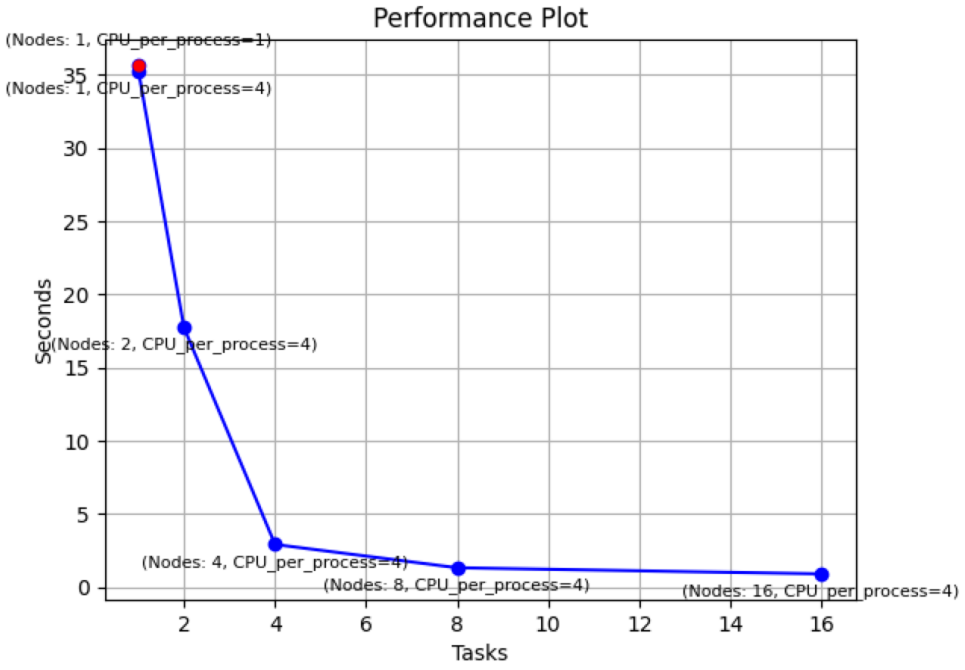


Figure 3: Result in 100 generations

a baseline. As in Fig 3, the first blue dot has an expanded number of available CPUs compared to the first red dot, but only a minimal improvement in performance. This may be due to the code's OpenMp not having the desired effect or a problem with the bash configuration file. In the points

connected by the blue line, we follow the principle of if tasks=n, then nodes=n, ensuring that each test case has the same node communication cost between different tasks (if nodestasks, then some nodes have multiple tasks, communicating via shared memory) and ensuring that each task has the same CPU, thus making the comparison more meaningful (emphasizing the role of MPI, while the role of OpenMp is not known why it is not shown, as discussed at the beginning of this paragraph). It can be seen that our acceleration effect is very significant, reducing from 35.22s to 0.89s, with a speed-up index of 39.6 times. It is shown in the following figure 4 which gives a more detailed representation of the acceleration ratio, making it more intuitive.

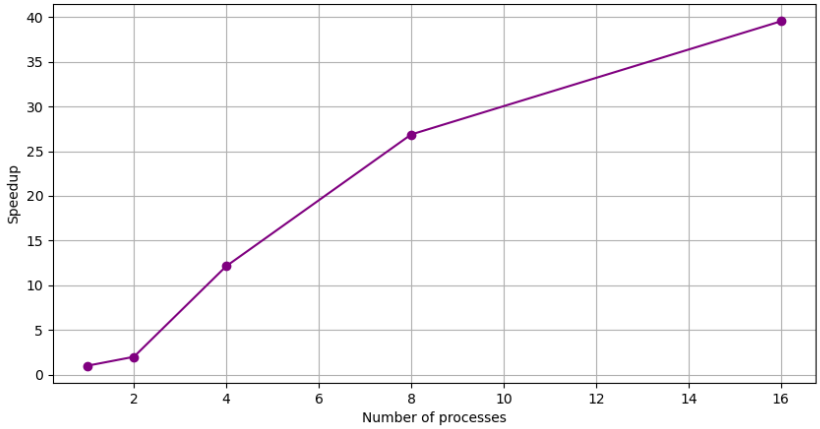


Figure 4: Speed up

# 4 Reflection

At the start, I used the combination of `export SLURM_NODES=$nodes` and

```
mpirun --npernode $SLURM_CPUS_PER_TASK -np $SLURM_NTASKS ./second_try
```

without setting the value for `#SBATCH --nodes.` I attempted to import the value of nodes into the environment variable `SLURM_NODES` so that mpirun would run the program using the specified number of nodes. However, I eventually discovered the problem - the program was only run using one node, resulting in only OpenMP acceleration. Ultimately, I found a detailed example in the Lisa documentation and learned that I should primarily refer to the documentation for knowledge and not rely solely on outdated code from other websites.

Another value worth reflecting on is the job requirement that 16 processes be used as mentioned in Markdown file. I initially thought this was a limit on the number of CPUs used, as one CPU typically corresponds to one process. However, in discussions with a teammate, I came to understand that the 16 processes referred to 16 tasks, or 16 MPI processes, and I wasted a lot of time due to this misunderstanding.

# 5 Code Implement

In our implementation we created several functions for different aspect of the game: checking if a cell is a border, counting all alive neigbouring cells of a given cell, counting all alive cells in a given board, initializing a board with only dead cells, inserting a pattern to a board, and calculating the next generation of a board.

Below is our full code implementation of the game of life in C with OpenMP and MPI:

```c
#include <stdio.h>
#include <omp.h>
#include <mpi.h>
#include <math.h>
#include <string.h>
#include <stdlib.h>
#include "./headers/beehive.h"
#include "./headers/glider.h"
#include "./headers/grower.h"

// number of cells per row and column
#ifndef N
#define N 3000
#endif

// Translating a (row, column) value into correct index in array
#define INDEX(i, j) (((i)*N)+(j))

#define ROW (N)
#define COL (N)
#define GENERATION (100)
#define BOARD_SIZE (N*N)

const int PATTERN_ROW=GROWER_HEIGHT;
const int PATTERN_COL=GROWER_WIDTH;

/**
 * @brief checking if cell (i,j) is a border
 *
 * @param i int row value
 * @param j int column value
 * @return int -1 if the given postition is not a border and 1 if it is a border
 */
int is_border(int i,int j){
    if (i == 0 || i == ROW-1 || j == 0 || j == COL-1){
        return 1;
    }
    return -1;
}

/**
 * @brief count the number of alive neigbouring cells to a given cell.
 * Uses a double for loop to go through the different row and column values around the
     given cell.
 * Check if the cell is not the given cell, actually int the board before checking it
    it is alive.
 *
 * @param a the board of the game
```

```
 * @param r int row value
 * @param c int column value
 * @return int, amount of alive neighbouring cells
 */
int count_live_neighbour_cell(int *a, int r, int c)
{
    int i, j, count = 0;
    for (i = r - 1; i <= r + 1; i++) {
        for (j = c - 1; j <= c + 1; j++) {
            if ((i == r && j == c) || (i < 0 || j < 0)
                || (i >= ROW || j >= COL)) {
                continue;
            }
            if (a[INDEX(i,j)] == 1) {
                count++;
            }
        }
    }
    return count;
}

/**
 * @brief counts all living cell on the board
 *
 * @param a board of the game
 * @param size amount of cells in the game
 * @return int, total amount of alive cells on the board
 */
int count_live_cell(int *a, int size) {
    int count = 0;
    #pragma omp parallel for
    for  (int i = 0; i < size; i++)  {
        if (a[i] == 1) {
            count++;
        }
    }
    return count;
}

/**
 * @brief initialize the game board.
 * Sets every cell on the game board to 0.
 *
 * @param a board to be initalized
 * @param size number of cells in the board
 */
void init_board(int *a, int size){
    for (int i = 0; i < size; i++)
```

```
    {
        a[i] = 0;
    }
}

/**
 * @brief Calculates the next generation of cells.
 * Double for loop that goes through all columns in a given amount of rows.
 * Then counting neighbours and checking if that cell should be alive or dead for the next generation.
 *
 * @param a current board generation
 * @param b next board generation
 * @param rows_per_proc the number of rows to calculate
 */
void calculate_next_generation(int *a, int *b, int rows_per_proc) {
    int neighbour_live_cell;

    #pragma omp parallel for private(neighbour_live_cell)
    for (int i = 1; i < rows_per_proc+1; i++) {
        for (int j = 0; j < COL; j++) {
            neighbour_live_cell = count_live_neighbour_cell(a, i, j);
            if (a[INDEX(i,j)] == 1 && (neighbour_live_cell == 2 || neighbour_live_cell == 3)) {
                if (is_border(i, j) == -1) {
                    b[INDEX(i,j)] = 1;
                }
            } else if (a[INDEX(i,j)] == 0 && neighbour_live_cell == 3) {
                if (is_border(i, j) == -1) {
                    b[INDEX(i,j)] = 1;
                }
            } else {
                b[INDEX(i,j)] = 0;
            }
        }
    }
}

/**
 * @brief Inserts a pattern of cells into the board
 *
 * @param pattern uint8_t pattern to be insertet in the board
 * @param board
 * @param pattern_row number of row in the pattern
 * @param pattern_col number of columns in the pattern
 * @param start_row row value of where the pattern should be inserted
 * @param start_col column value of where the pattern should be inserted
 */
```

```
void insert_pattern(uint8_t pattern[][PATTERN_COL], int *board, int pattern_row, int pattern_col, int start_row, int start_col) {
    for (int i = 0; i < pattern_row; i++) {
        for (int j = 0; j < pattern_col; j++) {
            board[INDEX(i + start_row,j + start_col)] = pattern[i][j];
        }
    }
}

/**
 * @brief Method for checking if the calculated amount of live cells are as expected for a generation
 *
 * @param a game board
 * @param generation
 */
void test_count_live_cell(int *a, int generation) {
    int final_output = count_live_cell(a,BOARD_SIZE);
  int expected_output;
  if(generation==10){
      expected_output = 49;
  }
  if(generation==100){
      expected_output = 138;
  }
  if (final_output == expected_output) {
      printf("Test passed: count_live_cell(a) returned %d\n", final_output);
        } else {
      printf("Test failed: count_live_cell(a) returned %d, expected %d\n", final_output, expected_output);
  }
}


int main(){
    // Initialize MPI
    int rank, num_procs;

    MPI_Init(NULL, NULL);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    //Error handling
    int error = MPI_Comm_set_errhandler(MPI_COMM_WORLD, MPI_ERRORS_RETURN);;

    //Compute rows per processes
    int rows_per_proc = (int)ceil((double)ROW / num_procs);
    int row_size = (rows_per_proc+2) * ROW;
```

```
186
187     int *A = NULL; //A is global board
188     int *C = NULL; //C and D is local board for each process
189     int *D = NULL;
190     A = (int*)malloc(BOARD_SIZE*sizeof(int));//global
191     B = (int*)malloc(BOARD_SIZE*sizeof(int));
192     C = (int*)malloc(row_size*sizeof(int));//local
193     D = (int*)malloc(row_size*sizeof(int));
194     // Error handling for malloc: if an error occurs with one of the memory
        allocations, stop running.
195     if (A == NULL || B == NULL || C == NULL || D == NULL){
196         perror("Error: ");
197         return EXIT_FAILURE;
198     }
199
200     init_board(C,row_size);
201     init_board(D,row_size);
202
203     //Divide data
204     if (rank == 0) {
205         //INIT: Initalizing start of a new game by initalizing a board and inserting
        pattern.
206         int i;
207         init_board(A,BOARD_SIZE);
208         insert_pattern(grower,A,PATTERN_ROW,PATTERN_COL,1500,1500);
209
210         memcpy(&C[N], &A[0], rows_per_proc * ROW * sizeof(int));
211         for (int proc = 1; proc < num_procs; proc++) {
212             MPI_Send(&A[proc*rows_per_proc*ROW], rows_per_proc * ROW, MPI_INT, proc,
        1, MPI_COMM_WORLD);
213         }
214     } else {
215         //For each processor, there is a local matrix.
216         MPI_Recv(&C[ROW], rows_per_proc * ROW, MPI_INT, 0, 1, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
217     }
218
219     double start,end;
220     start=MPI_Wtime();
221
222     // Gameplay
223     int top_neighbour = rank-1;
224     int bot_neighbour = rank+1;
225     for (int i = 0; i < GENERATION; i++)
226     {
227         // top
228         if (top_neighbour >= 0){
229
```

```
230             MPI_Sendrecv(&C[ROW], ROW, MPI_INT, top_neighbour, 0,C, ROW, MPI_INT,
    top_neighbour, MPI_ANY_TAG,MPI_COMM_WORLD, MPI_STATUS_IGNORE);
231         }
232         // bottom
233         if (bot_neighbour < num_procs)
234         {
235             MPI_Sendrecv(&C[(rows_per_proc)*ROW], ROW, MPI_INT, bot_neighbour, 0,&C[(
    rows_per_proc+1)*ROW], ROW, MPI_INT, bot_neighbour, MPI_ANY_TAG,MPI_COMM_WORLD,
    MPI_STATUS_IGNORE);
236         }
237         //game
238         calculate_next_generation(C,D,rows_per_proc);
239         {
240             // Swap the role of the two arrays.
241             int *tmp = C;
242             C = D;
243             D = tmp;
244         }
245     }
246
247     //Merge data
248     if (rank == 0) {
249         memcpy(A, &C[ROW], rows_per_proc * ROW * sizeof(int));
250         for (int proc = 1; proc < num_procs; proc++) {
251             MPI_Recv(&A[proc*rows_per_proc*ROW], rows_per_proc*ROW, MPI_INT, proc, 1,
    MPI_COMM_WORLD, MPI_STATUS_IGNORE);
252         }
253         //test
254         if(GENERATION!=5000){
255                 test_count_live_cell(A,GENERATION);
256         }
257         else{
258             printf("# of cell alive in the 5000th generation: %d\n",count_live_cell(A,
    BOARD_SIZE));
259         }
260     }
261     else {
262         MPI_Send(&C[ROW], rows_per_proc*ROW, MPI_INT, 0, 1, MPI_COMM_WORLD);
263     }
264
265     end=MPI_Wtime();
266     // Print result
267     if (rank==0){
268         printf("Obtained in %f seconds, rank=%d\n",end - start,rank);
269     }
270
271     free(A);
272     free(C);
```

```
273     free(D);
274 
275     // Finalize MPI
276     MPI_Finalize();
277     return 0;
278 }
```