Background Story

In this assignment, we need to write a MPI program to simulate Conway's Game of Life cellular automaton. For the Conway's game of life[1], there are basically 2 type of cell states: alive and dead.

The following are the 4 rules for simulate this game automatically:

- If a cell has fewer than 2 alive neighbors, it will be dead in the next time step
- If an alive cell has 2 or 3 alive neighbours, it will be alive in the next time step
- If a cell has more than 3 alive neighbors, it will be dead in the next time step
- If a dead cell has 3 alive neighbours, it will be alive in the next time step

Goal of Project

There are two principal methods of parallel computing: distributed memory computing and shared memory computing[2]. For distributed memory(processes), in order to pass messages, the program must communicate. For shared memory(threads), multiple computing nodes are sharing one network. To have multiple computing nodes with multiple cores the same time, it is the concept of hybrid. If there is only 1 thread with 1 process, it is just serial program.

- MPI(Message Passing Interface) is the standard of message passing in distributed memory environment,
- OpenMP is the API that supports multi-platform shared memory multiprocessing programming.

Research questions

1. What fraction of the overall running time is spent in inter-process communication?
2. How does the code scale with increasing grid size and increasing MPI process concurrencies?

Project Description

First of all, we need to consider a basic question: what data is parallelized? The answer is, each process receives a certain number of rows; each thread receives a certain number of columns. In this project, we only change the number of rows. The program should take as input two values, m (the grid dimension, use an mxm grid) and k (the number of time steps).The initial state of the system could be defined in the code. The m x m grid must be partitioned across multiple processes (i.e., each process is responsible for updating m x m/p grid values). Secondly, we need to consider what message passing occurs. In fact, the messaging passing is between different ranks in distributed memory environment.

Observations

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

According to the Amdahl's law,S_latency is the theoretical speedup of the execution of the whole task; s is the speedup of the part of the task that benefits from improved system resources; p is the proportion of execution time that the part benefiting from improved resources originally occupied[3]. This equation can basically indicates the fraction of the overall running time is spent in inter-process communication, because the communication latency is one of main cause in this project.

From figure 1, the values changings from 0.755 secs to 0.322 secs, it indicates that the running time of the program becomes twice faster, when first moving from the shared memory to distributed memory.

After the dramatic speedup, the changing speed becomes slow, which also follows the Amdahl's law. As more and more cores being added, there is less speedup for the computing performance.

In fact, the chart does not include other results of the serial and openmp code, it is because there is no meaningful result if they are running on distributed memory environment.

Figure 1: Moving from Shared Memory To Distributed Memory

Real Time(sec)				
# of nodes	# of processes	Serial	OpenMP	MPI
1	4	0.978	0.88	0.755
2	8			0.322
3	12			0.346
4	16			0.304
5	20			0.367
6	24			0.395

Observations

From figure 2 shown below, we can see that the running time changes gradually with the increment of the grid size. The reason why the running time increases is mainly due to the time spent in inter-process communication increases.

When increases the total number of cores, I have the number of nodes increased with same ratio, so that the same number of cores used per node can be ensured.

Figure 2: Vary the number of MPI processes, and determine running time for different values of m

Real Time(sec)					
# of nodes	#of cores(processes)	# of dimensions	Serial	OpenMP	MPI
1	4	400	0.524	0.423	0.31
2	8	800			0.32
3	12	1200			0.342
4	16	1600			0.35
5	20	2000			0.707
6	24	2400			0.901

Conclusion

There are some issues could not be ignored. In fact, the issue of communication overhead[4] can affect the the result of parallel computing algorithms a lot. As the more processes are added, the communication overhead increases. In other words, more time will be consumed for message passing.

Reference

[1]C. Lipa, "Conway's Game of Life'", Math.cornell.edu, 2017. [Online]. Available: http://www.math.cornell.edu/~lipa/mec/lesson6.html. [Accessed: 09- Nov- 2017]

[2]"Hybrid Parallelism: Parallel Distributed Memory and Shared Memory Computing | Intel® Software", Software.intel.com, 2017. [Online]. Available: https://software.intel.com/en-us/articles/hybrid-parallelism-parallel-distributed-memory-and-shared-memory-computing. [Accessed: 09- Nov- 2017]

[3]"Amdahl's law", En.wikipedia.org, 2017. [Online]. Available: https://en.wikipedia.org/wiki/Amdahl%27s_law. [Accessed: 09- Nov- 2017]

[4]"Cite a Website - Cite This For Me", People.eecs.berkeley.edu, 2017. [Online]. Available: https://people.eecs.berkeley.edu/~culler/papers/isca97.pdf. [Accessed: 09- Nov- 2017]

Code: gameoflife.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <assert.h>
```

```
#include <sys/time.h>
#include <time.h>

#ifdef USE_MPI
#include <mpi.h>
#endif

#ifdef _OPENMP
#include <omp.h>
#endif

#define alive 1
#define dead 0

static double timer() {
   struct timeval tp;
   gettimeofday(&tp, NULL);
   return ((double) (tp.tv_sec) + 1e-6 * tp.tv_usec);
}

int min_m = 1;
int min_k = 1;


void exit_if(int boolean_expression, char function_name[32], int rank)
{
   if(boolean_expression)
   {
#ifdef USE_MPI
      fprintf(stderr, "Rank %d ", rank);
#endif
#ifdef _OPENMP
      fprintf(stderr, "Thread %d ", omp_get_thread_num());
#endif
      fprintf(stderr, "ERROR in %s\n", function_name);
      exit(-1);
   }

   return;
}


int main(int argc, char **argv) {

   int rank=0, num_tasks = 1;

/* Initialize MPI */
#ifdef USE_MPI
   exit_if((MPI_Init(&argc, &argv) != MPI_SUCCESS), "MPI_Init", rank);
   exit_if((MPI_Comm_rank(MPI_COMM_WORLD, &rank) != MPI_SUCCESS),"MPI_Comm_rank", rank);
   exit_if((MPI_Comm_size(MPI_COMM_WORLD, &num_tasks)!= MPI_SUCCESS), "MPI_Comm_size", rank);
#endif

    if (argc != 3) {
      if (rank == 0) {
         fprintf(stderr, "%s <m> <k>\n", argv[0]);
         fprintf(stderr, "Program for parallel Game of Life\n");
         fprintf(stderr, "with 1D grid partitioning\n");
         fprintf(stderr, "<m>: grid dimension (an mxm grid is created)\n");
         fprintf(stderr, "<k>: number of time steps\n");
         fprintf(stderr, "(initial pattern specified inside code)\n");

   int m_p , our_current_row, my_current_column, my_neighbor_row, my_neighbor_column,
      my_number_of_alive_neighbors, c, return_value, next_lowest_rank,
      next_highest_rank;

   while((c = getopt(argc, argv, "r:c:t:")) != -1)
   {
      switch(c)
      {
         case 'r':
            m = atoi(optarg);
            break;
         case 'c':
            m = atoi(optarg);
            break;
         case 't':
            k = atoi(optarg);
```

```
            break;
          case '?':
          default:
#ifdef USE_MPI
            fprintf(stderr, "Usage: mpirun -np num_tasks %s [-r m] [-c m] [-t k]\n", argv[0]);
#else
            fprintf(stderr, "Usage: %s [-r m] [-c m] [-t k]\n", argv[0]);
#endif
            exit(-1);
      }
   }

   int m = 5,  k = 5;

   argc -= optind;
   argv += optind;


   if(return_value != 0)
      exit(-1);

   m_p = m / num_tasks;
   if(rank == num_tasks - 1)
   {
      m_p += m % num_tasks;
   }

   if (rank == 0) {
      fprintf(stderr, "Using m: %d, m_p: %d, k: %d\n", m, m_p, k);
      fprintf(stderr, "Requires %3.6lf MB of memory per task\n",
            ((2*4.0*m_p)*m/1e6));
   }

   int **grid_current, **grid_next;
   int current_time_step;

   exit_if(((grid_current = (int**)malloc((m_p + 2) * (m + 2) * sizeof(int))) == NULL),"malloc(grid_current)", rank);
   exit_if(((grid_next = (int**)malloc((m_p + 2) * (m + 2) * sizeof(int))) == NULL), "malloc(grid_next)", rank);
   for(our_current_row = 0; our_current_row <= m_p + 1;our_current_row++)
   {
      exit_if(((grid_current[our_current_row]= (int*)malloc((m + 2) * sizeof(int))) == NULL), "malloc(grid_current[some_row])", rank);
      exit_if(((grid_next[our_current_row]= (int*)malloc((m + 2) * sizeof(int)))== NULL), "malloc(grid_next[some_row])", rank);
   }


   for(our_current_row = 1; our_current_row <= m_p;our_current_row++)
   {
#ifdef _OPENMP
#pragma omp parallel for private(my_current_column)
#endif
      for(my_current_column = 1; my_current_column <= m;
            my_current_column++)
      {
         grid_current[our_current_row][my_current_column] =
            random() % (alive + 1);
      }
   }

   if(rank == 0)
      next_lowest_rank = num_tasks - 1;
   else
      next_lowest_rank = rank - 1;

   if(rank == num_tasks - 1)
      next_highest_rank = 0;
   else
      next_highest_rank = rank + 1;

   for(current_time_step = 0; current_time_step <= k - 1;
         current_time_step++)
   {
#ifdef USE_MPI
      exit_if((MPI_Send(grid_current[1], m + 2,MPI_INT, next_lowest_rank, 0, MPI_COMM_WORLD) !=MPI_SUCCESS),"MPI_Send(top row)", rank);

      exit_if((MPI_Send(grid_current[m_p], m + 2, MPI_INT, next_highest_rank,0, MPI_COMM_WORLD) != MPI_SUCCESS),"MPI_Send(bottom row)", rank);

      exit_if((MPI_Recv(grid_current[m_p + 1],m + 2, MPI_INT, next_highest_rank, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE) !=
MPI_SUCCESS),"MPI_Recv(bottom row)", rank);
```

```
      exit_if((MPI_Recv(grid_current[0], m + 2,MPI_INT, next_lowest_rank, 0, MPI_COMM_WORLD,MPI_STATUS_IGNORE) != MPI_SUCCESS),"MPI_Recv(top row)",
rank);
#else
#ifdef _OPENMP
#pragma omp parallel private(my_current_column)
#endif
      for(my_current_column = 0; my_current_column <= m + 1; my_current_column++)
      {
         grid_current[0][my_current_column]= grid_current[m_p][my_current_column];
         grid_current[m_p + 1][my_current_column]= grid_current[1][my_current_column];
      }
#endif
      for(our_current_row = 0; our_current_row <= m_p + 1; our_current_row++)
      {
         grid_current[our_current_row][0] =grid_current[our_current_row][m];

         grid_current[our_current_row][m + 1] =grid_current[our_current_row][1];
      }

      for(our_current_row = 1; our_current_row <= m_p;our_current_row++)
      {
#ifdef _OPENMP
#pragma omp parallel for private(my_current_column, my_neighbor_row, my_neighbor_column, my_number_of_alive_neighbors)
#endif
         for(my_current_column = 1; my_current_column <= m;my_current_column++)
         {
            my_number_of_alive_neighbors = 0;

            for(my_neighbor_row = our_current_row - 1;my_neighbor_row <= our_current_row + 1;my_neighbor_row++)
            {

               for(my_neighbor_column = my_current_column - 1;my_neighbor_column <= my_current_column + 1; my_neighbor_column++)
               {

                  if((my_neighbor_row != our_current_row|| my_neighbor_column != my_current_column)&& (grid_current[my_neighbor_row][my_neighbor_column]
== alive))
                  {
                     my_number_of_alive_neighbors++;
                  }
               }
            }

            if(my_number_of_alive_neighbors < 2)
            {
               grid_next[our_current_row][my_current_column] = dead;
            }

            if(grid_current[our_current_row][my_current_column] == alive && (my_number_of_alive_neighbors == 2 || my_number_of_alive_neighbors == 3))
            {
               grid_next[our_current_row][my_current_column] = alive;
            }

            if(my_number_of_alive_neighbors > 3)
            {
               grid_next[our_current_row][my_current_column] = dead;
            }

            if(grid_current[our_current_row][my_current_column] == dead && my_number_of_alive_neighbors == 3)
            {
               grid_next[our_current_row][my_current_column] = alive;
            }
         }
      }

      for(our_current_row = 1; our_current_row <= m_p; our_current_row++)
      {
#ifdef _OPENMP
#pragma omp parallel for private(my_current_column)
#endif
         for(my_current_column = 1; my_current_column <= m; my_current_column++)
         {
            grid_current[our_current_row][my_current_column] = grid_next[our_current_row][my_current_column];
         }
      }
   }

   for(our_current_row = m_p + 1; our_current_row >= 0; our_current_row--)
```

```
    {
        free(grid_next[our_current_row]);
        free(grid_current[our_current_row]);
    }
    free(grid_next);
    free(grid_current);

#ifdef USE_MPI
    exit_if((MPI_Finalize() != MPI_SUCCESS), "MPI_Finalize", rank);
#endif

    return 0;
}
```