File : error

```c

/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : error.h                                           */
/*------------------------------------------------------------ */

#ifndef ERROR_H
#define ERROR_H

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

/**
 * If Debug Flag is on, create a maccro to print debug information
 * %param MSG : String to print
 * %param ... : List of param [ for example if want to print variable value ]
 */
#ifdef DEBUG
    #define DEBUG_MSG(MSG, ...)   \
    do {                          \
        fprintf(stderr, "\n\t[DEBUG] File : %s - Line : %d - Function : %s() : " MSG "\n",
            __FILE__, __LINE__, __func__, ## __VA_ARGS__);   \
    } while(0);
#else
    #define DEBUG_MSG(MSG, ...)
#endif

/**
 * Create a maccro for quit the program
 * %param MSG : String to print
 * %param ... : List of param [ for example if want to print variable value ]
 */
#define QUIT_MSG(MSG, ...)                          \
    do {                                            \
        DEBUG_MSG(MSG, ##__VA_ARGS__)               \
        fprintf(stderr, "[FATAL ERROR] ");          \
        fprintf(stderr, MSG, ## __VA_ARGS__);       \
        perror(NULL);                               \
        exit(EXIT_FAILURE);                         \
    }while(0);

#endif /* ERROR_H included */
```

File : main

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <time.h>
#include <mpi.h>

#include "matrix.h"
#include "rows.h"
#include "memory.h"
#include "error.h"
#include "game.h"
#include "option.h"


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

    Option o;
    Game *g, *s = NULL;
    double time_taken = 0.0;
    int total_proc, my_id, my_x, my_y, proc_slice, slice_size, size_tick[4];

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD,&total_proc);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_id);

    /* The process 0 get all parameters, load board if needed etc */
```

```
28      if ( my_id == 0 ) {
29          srand(time(NULL));
30          o = getOption(argc, argv);  /* Get all option     */
31
32          if ( *o.file_path != '\0' ) /* If path file is not empty */
33              if ( (g = loadBoard(o.file_path)) == NULL ) /* then use the given file [load id
                    ] */
34                  fprintf(stderr, "Can't load file %s\n", o.file_path);
35
36          if ( g == NULL ) /* If load of file fail Or no grid given */
37              g = generateRandomBoard(o); /* then create one */
38
39          time_taken = MPI_Wtime();
40
41          /* Fill value that will be needed for other process */
42          size_tick[0] = g->rows;
43          size_tick[1] = g->cols;
44          size_tick[2] = o.max_tick;
45          size_tick[3] = o.method;
46
47      }
48
49      /* Broadcast all the needed value ( in a array for compacting data) */
50      MPI_Bcast(size_tick, 4, MPI_INT, 0, MPI_COMM_WORLD);
51
52      /*********************/
53      /*     init part     */
54      /*********************/
55
56      if ( size_tick[3] == DIVIDE_MATRICE) {
57          proc_slice = sqrt(total_proc);
58          slice_size = size_tick[0] / proc_slice;
59          my_x = my_id / proc_slice;
60          my_y = my_id % proc_slice;
61
62          if ( my_id == 0 && ( fmod(sqrt(total_proc), 1.0f) != 0
63                      || size_tick[1] != size_tick[0] ) ) {
64
65              QUIT_MSG("Grid could no be devided by the total number of process %d - %d\n",
                        size_tick[1], total_proc);
66          }
67
68          if ( my_id == 0 )
69              s = sendAllSubMatrice(g, slice_size, proc_slice);
70          else
71              s = receivedMatrix(my_x, my_y, slice_size, proc_slice);
72      } else {
73          if ( my_id == 0 && size_tick[1] % total_proc != 0 )
74              QUIT_MSG("Grid could no be devided by the total number of process\n");
75
76          /* Lets allocated the memory for the shared buffer at the same moment */
77          slice_size = size_tick[1] / total_proc;
78
79          if ( total_proc > 1 ) /* Next formular only work if there is more than 1 proc */
80              s = newGame(size_tick[0], slice_size + (my_id != 0) + (my_id != total_proc - 1)
                        );
81          else
82              s = newGame(size_tick[0], slice_size);
83
84          MPI_Scatter( g->board, size_tick[0] * slice_size, MPI_CHAR,
85                  __posBufferRecv(my_id, s->board, size_tick[0]),
86                  size_tick[0] * slice_size, MPI_CHAR,
87                  0, MPI_COMM_WORLD);
88      }
89
90      /*********************/
91      /*     Init end      */
92      /* ------------------*/
93      /*   Process tick    */
94      /*********************/
95
96      /* This pre-process indication is defined by the make display command */
97      #if PRINT
```

```
98      if ( my_id == 0 )
99          gamePrintInfo(g, size_tick[2]);
100     MPI_Barrier(MPI_COMM_WORLD);
101     #endif
102
103     for ( ; size_tick[2] > 0; size_tick[2]--) {
104
105         if ( size_tick[3] == DIVIDE_MATRICE ) {
106             shareMatrixBorder(s, my_x, my_y, slice_size, proc_slice);
107             processMatrixGameTick(s, my_x, my_y, slice_size);
108             gatherMatrix(g, s, my_x, my_y, slice_size, proc_slice, total_proc);
109         } else {
110             shareGetBorder(s, slice_size, my_id, total_proc);
111             processRowsGameTick(s);
112             MPI_Gather( __posBufferRecv(my_id, s->board, size_tick[0]),
113                         size_tick[0] * slice_size, MPI_CHAR,
114                         g->board, size_tick[0] * slice_size, MPI_CHAR,
115                         0, MPI_COMM_WORLD);
116         }
117
118         /* If we need to display, Then we going to print */
119         #if PRINT
120         if ( my_id == 0 )
121             gamePrintInfo(g, size_tick[2] - 1);
122         MPI_Barrier(MPI_COMM_WORLD);
123         #endif
124     }
125
126     if ( my_id == 0 ) {
127         time_taken =  MPI_Wtime() - time_taken;
128         printf("Time : %f\n", time_taken);
129
130         if ( o.save_file )
131             saveBoard(g);
132
133         freeGame(g);            /* Free space we are not in Java */
134     }
135
136     freeGame(s);
137     MPI_Finalize();
138
139     exit(EXIT_SUCCESS);
140 }
```

File : game_struct

```
1  /*------------------------------------------------------------ */
2  /* AUTEUR : REYNAUD Nicolas                                    */
3  /* FICHIER : game_struct.h                                     */
4  /*------------------------------------------------------------ */
5
6
7  #ifndef GAME_STRUCT_H
8  #define GAME_STRUCT_H
9
10 /**
11  * Struct that represent a game
12  */
13 typedef struct {
14     char *board; /* The board as an array of 0's and 1's. */
15     unsigned int cols;   /* The number of columns. */
16     unsigned int rows;   /* The number of rows. */
17 } Game;
18 #endif
```

File : game

```
1  /*------------------------------------------------------------ */
2  /* AUTEUR : REYNAUD Nicolas                                    */
3  /* FICHIER : game.h                                            */
4  /*------------------------------------------------------------ */
5
6  #ifndef GAME_H
7  #define GAME_H
8
```

```c
#include "game_struct.h"
#include "option_struct.h"

/**
 * First need to define all the constante
 * thoses one are usefull for generate a random board if needed
 */
#define MIN_COLS_SIZE 3 /* Minimum number of cols */
#define MIN_ROWS_SIZE 3 /* Minimum number of rows */

#define POURCENT_BEEN_ALIVE 50 /* Pourcentage of cell to keep alive during generation */

#define DEAD_CELL 0
#define ALIVE_CELL 1

/* Define constant to identify which method we use for dividing the grid */
#define DIVIDE_ROWS 0
#define DIVIDE_MATRICE 1

/**
 * Given X, and Y this function output the position into the board.
 * For example POS(0,0,G) return 0, cause the cell in 0 on X, and 0 on Y is the cell 0 of the board
 * %param X : Position on the X coordinate
 * %param Y : Position on the Y coordinate
 * %param G : Board on which we need to compute the position
 * %return  : The associate position on the board
 */
#define POS(X, Y, G) (__position(X,Y,G))
int __position(unsigned int x, unsigned int y, Game *g);

/**
 * Function that print the board, this function determine if we need to print it or not
 * i.e if the programme is make with make display
 * This function also determine which function we need to use to display the board, and print the
 * number of generation left.
 *
 * %param g : The game which contains the board to print
 * %param tick_left : total of tick game left to do
 */
void gamePrintInfo ( Game* g, int tick_left);

/**
 * Function that create a new game
 * %param rows : Total number of rows onto the new board
 * %param cols : Total number of Column onto the new board
 * %return     : Allocated game structure which contains all the information
 */
Game* newGame(unsigned int rows, unsigned int cols);

/**
 * Function that free the memory associate with a game
 * %param g : Game to free
 */
void freeGame(Game* g);

/**
 * Function that generate a random board if no are given
 * %param o : Option for generating the board
 * %return  : a random board
 */
Game* generateRandomBoard(Option o);

/**
 * Function that process the board if the method used it by rows
 * %param g : Game which contains the Board to process
 */
void processRowsGameTick(Game *g);

/**
 * Function that process the board if the method used is by matrix
 * %param g : Game which contains the board to process
```

```
 * %param my_x : My process id on x
 * %param my_y : My process id on y
 * %param slice_size : Size of the slice either on rows or columns
 */
void processMatrixGameTick(Game *g, int my_x, int my_y, int slice_size);

/**
 * Load in memory a game / board contains into a file
 * %param name : path to the file to load
 * %return     : The game structure associate with the contenant of the file
 *               Or NULL if that fail [i.e the file is not valide
 */
Game* loadBoard(char* name);

/**
 * Function that save a game into a file
 * %param g : the board to save
 * %return  : true if it succeed
 *            false otherwise
 */
bool saveBoard(Game *g);

#endif
```

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <mpi.h>

#include "error.h"
#include "game.h"
#include "game_struct.h"
#include "memory.h"

/**
 * Private function that compute the position of the board given a x and a y
 * %param x : Position on the X coordinate
 * %param y : Position on the Y coordinate
 * %param g : Game where we need to compute the cell position
 * %return  : Position of the cell associate with the X and Y coordinate
 */
int __position(unsigned int x, unsigned int y, Game* g) {
    return g->rows * x + y;
}

/**
 * Private function that print a simple line
 * %param g : Game structure which contains the information relative to the game
 */
void __printLine(Game* g) {
    unsigned int i = 0;

    printf( "+");
    for ( i = 0; i < g->rows + 2; i++ ) /* the 2 '+'  */
        printf( "-");
    printf( "+\n");

}

/**
 * Private function that really print the board contenant
 * %param g : Game struct which contains the board to print
 * %param pf : Pointer to a printing function
 */
void __gamePrint (Game* g) {
    unsigned int x, y;

    printf( "Board size : \n");
    printf( "  %d Columns\n", g->cols);
    printf( "  %d rows\n", g->rows);
    __printLine(g);

    for ( x = 0; x < g->cols; x++) {
```

```
50          printf( "| ");
51          for ( y = 0; y < g->rows; y++) {
52              printf( "%c", ((g->board[POS(x, y, g)] == DEAD_CELL) ? '.' : '#'));
53          }
54
55          printf( " |\n");
56      }
57
58      __printLine(g);
59
60      DEBUG_MSG("Print board finish\n");
61  }
62
63  void gamePrintInfo(Game* g, int tick_left) {
64
65      DEBUG_MSG("Board : %s\n", g->board);
66
67      #ifndef PRINT
68           return;
69      #endif
70
71      if ( tick_left >= 0 )
72          printf("%d Generation left.\n", tick_left);
73
74      __gamePrint(g);
75  }
76
77  /**
78   * Private function that allocate a new board
79   * %param rows : Total number of rows onto the board
80   * %param cols : Total number of column onto the board
81   * %return     : Allocated array of char which will contains the board
82   */
83  char* __newBoard(unsigned int rows, unsigned int cols) {
84      char* board = NEW_ALLOC_K(rows * cols, char);
85      memset(board, DEAD_CELL, rows * cols);
86      return board;
87  }
88
89  Game* newGame(unsigned int rows, unsigned int cols) {
90      Game* g = NEW_ALLOC(Game);
91
92      g->rows = rows;
93      g->cols = cols;
94
95      g->board = __newBoard(rows, cols);
96      return g;
97  }
98
99  void freeGame(Game* g)  {
100     if ( g == NULL )
101         return;
102
103     free(g->board);
104     free(g);
105 }
106
107 Game* generateRandomBoard(Option o) {
108
109     unsigned int rows = 0, cols = 0;
110     Game* g;
111
112     g = newGame(o.rows, o.cols);
113
114     DEBUG_MSG("Ligne : %d, Cols : %d\n", o.rows, o.cols);
115     for(cols = 0; cols < g->cols; cols++)
116         for (rows = 0; rows < g->rows; rows++)
117             g->board[POS(cols, rows, g)] = (
118                 ( rand() % 100 >= POURCENT_BEEN_ALIVE ) ?
119                     DEAD_CELL:
120                     ALIVE_CELL
121                 );
122     DEBUG_MSG("Generate random finish");
```

```
123     return g;
124 }
125 
126 /**
127  * Private function which compute the total number of neighbour of a cell
128  * %param x : X position of the cell on the board
129  * %param y : Y position of the cell on the board
130  * %param g : Game struct wich contains all information relative to the game
131  * %return  : Total number of neighbour of this cell
132  */
133 int __neighbourCell(unsigned int x, unsigned int y, Game *g) {
134     unsigned int total = 0;
135     char *b = g->board;
136     bool isTop, isBot;
137 
138     isTop = (x == 0);
139     isBot = (x == g->cols - 1);
140 
141     if ( y % g->rows != g->rows - 1) {
142         total +=               (b[POS(x, y + 1, g)]     == ALIVE_CELL); /* Right */
143         if ( !isBot ) total += (b[POS(x + 1, y + 1, g)] == ALIVE_CELL); /* Right - Down */
144         if ( !isTop ) total += (b[POS(x - 1, y + 1, g)] == ALIVE_CELL); /* Up - Right */
145     }
146 
147     if ( y % g->rows != 0 ) {
148         total +=               (b[POS(x, y - 1, g)]     == ALIVE_CELL); /* Left */
149         if ( !isBot ) total += (b[POS(x + 1, y - 1, g)] == ALIVE_CELL); /* Left - Down */
150         if ( !isTop ) total += (b[POS(x - 1, y - 1, g)] == ALIVE_CELL); /* Up - Left */
151     }
152 
153     if ( !isBot ) total += (b[POS(x + 1, y, g)] == ALIVE_CELL); /* Down */
154     if ( !isTop ) total += (b[POS(x - 1, y, g)] == ALIVE_CELL); /* Up   */
155 
156     return total;
157 }
158 
159 /**
160  * Private function which process a cell, i.e update the cell on the other board according
        to ome rules
161  * %param x : Position on X of the cell on the board
162  * %param y : Position on Y of the cell on the board
163  * %param g : Game struct which contains all information relative to the game
164  * %return  : New state of the cell in x / y coordinate.
165  */
166 char __process(unsigned int x, unsigned int y, Game* g) {
167     unsigned int neightbour = __neighbourCell(x, y, g);
168 
169     if ( neightbour < 2 || neightbour > 3 ) return DEAD_CELL;
170     else if ( neightbour == 3 )             return ALIVE_CELL;
171     else                                    return g->board[POS(x, y, g)];
172 }
173 
174 void processRowsGameTick(Game *g) {
175     int my_id, total_proc;
176     unsigned int x, y;
177     char* next;
178 
179     MPI_Comm_rank(MPI_COMM_WORLD,&my_id);
180     MPI_Comm_size(MPI_COMM_WORLD,&total_proc);
181 
182     next = __newBoard(g->rows, g->cols);
183 
184     for ( x = (my_id != 0); x < g->cols - (my_id != total_proc - 1); x++ )
185         for ( y = 0; y < g->rows; y++ )
186             next[POS(x, y, g)] = __process(x, y, g);
187 
188     free(g->board);
189     g->board = next;
190 }
191 
192 void processMatrixGameTick(Game *g, int my_x, int my_y, int slice_size) {
193     char *next;
194     int x, y, startx, starty;
```

```
195 
196     startx = (my_x != 0);
197     starty = (my_y != 0);
198 
199     next = __newBoard(g->rows, g->cols);
200     for ( x = 0; x < slice_size; x++)
201         for ( y = 0; y < slice_size; y++)
202             next[POS(x + startx, y + starty, g)] = __process(x + startx, y + starty, g);
203 
204     free(g->board);
205     g->board = next;
206 }
207 
208 Game* loadBoard(char* name) {
209     char reader = ' ';
210     unsigned int rows = 0, cols = 0;
211     FILE* fp = NULL;
212     Game *g = NULL;
213 
214     if ( (fp = fopen(name, "r")) == NULL ) return NULL;
215     if ( fscanf(fp, "Rows : %d\nCols : %d\n", &cols, &rows) != 2) { fclose(fp); return NULL
        ; }
216 
217     g = newGame(rows, cols);
218 
219     DEBUG_MSG("Rows : %d, Cols : %d\n", rows, cols);
220     rows = 0; cols = 0; /* Reinit variable */
221 
222     while ( (reader = fgetc(fp)) != EOF ) {
223         if ( reader == '.' ) reader = DEAD_CELL;
224         if ( reader == '#' ) reader = ALIVE_CELL;
225 
226         if ( reader == '\n') ++cols;
227         else g->board[POS(cols, rows, g)] = reader;
228 
229         if ( ++rows > g->rows ) rows = 0;
230     }
231 
232     fclose(fp);
233 
234     if ( cols != g->cols && (reader == '\n' && rows != 0) ) { freeGame(g); return NULL; }
235     return g;
236 }
237 
238 bool saveBoard(Game *g) {
239     unsigned int i;
240     FILE *fp = NULL;
241 
242     if ( (fp = fopen("output.gol", "w")) == NULL ) return false;
243 
244     fprintf(fp, "Rows : %d\nCols : %d\n", g->rows, g->cols);
245     for ( i = 0; i < g->cols * g->rows; i++ ) {
246 
247         fprintf(fp, "%c", ((g->board[i]) ? '#' : '.') );
248         if ( i % g->cols == g->cols - 1 ) fprintf(fp, "\n");
249     }
250 
251     #ifdef PRINT
252         printf("File saved into : output.gol\n");
253     #endif
254 
255     fclose(fp);
256     return true;
257 }
```

File : memory

```
1 /*------------------------------------------------------------ */
2 /* AUTEUR : REYNAUD Nicolas                                    */
3 /* FICHIER : memory.h                                          */
4 /*------------------------------------------------------------ */
5 
6 
7 #ifndef MEMORY_H
```

```c
#define MEMORY_H

#include <stdlib.h>

/**
 * Function that allocate a single object
 * %param OBJECT : Object type to allocate
 * %return       : Pointer in memory associate with the object Type.
 */
#define NEW_ALLOC(OBJECT) (NEW_ALLOC_K(1, OBJECT))

/**
 * Function that allocate an array of the same Object
 * %param K      : Total number to allocate
 * %param OBJECT : Object type to allocate
 * %return       : Pointer in memory associate with the object type.
 */
#define NEW_ALLOC_K(K, OBJECT) (__memAlloc(K, sizeof(OBJECT)))

/**
 * Private function that shouldn't be used
 * The definition of this function is in memory.c
 */
void *__memAlloc(int total, size_t object_size);

#endif
```

```c
#include "error.h"
#include "memory.h"

/**
 * Private function that board the allocation of an object
 * %param total : Total number of object that we need to allocate
 * %param object_size : Size of the object which we need to allocate
 * %return : Pointer on the memory associate with the new object
 */
void *__memAlloc(int total, size_t object_size) {

    void *p = calloc(total, object_size);

    if ( p == NULL )
        QUIT_MSG("Canno't allocate new object\n");

    return p;

}
```

File : option_struct

```c
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : error.h                                           */
/*------------------------------------------------------------ */

#ifndef OPTION_STRUCT
#define OPTION_STRUCT

#include <stdbool.h>

/**
 * Structure that will contains all of the option
 */
typedef struct Option {
    int max_tick;             /* How much tick we need to do            - Default : 100 */
    char* file_path;          /* Path to the file to load               - Default : "" */
    unsigned int rows;        /* Number of rows to generate             - Default : Random
        */
    unsigned int cols;        /* Number of columns to generate          - Default : Random
        */
    bool save_file;           /* Do we need to save the last grid ?     - Default : false
        */
    int method;               /* Divide by grid or by rows              - Default :
        DIVIDE_GRID */
} Option;
```

```
22
23 #endif
```

File : option

```
1  /*-------------------------------------------------------------- */
2  /* AUTEUR : REYNAUD Nicolas                                     */
3  /* FICHIER : error.h                                            */
4  /*-------------------------------------------------------------- */
5
6
7  #ifndef OPT
8  #define OPT
9
10 #include "option_struct.h"
11
12 /* List of possible option */
13 #define OPT_LIST "hf:t:r:c:sm"
14
15 /** Use the definition defined by David Titarenco
16  *  On StackOverFlow http://stackoverflow.com/questions/3437404/min-and-max-in-c
17  */
18 #define MAX(a,b) \
19    ({ __typeof__ (a) _a = (a); \
20        __typeof__ (b) _b = (b); \
21      _a > _b ? _a : _b; })
22
23 /**
24  * Print the usage of the program
25  * %param name : name of the program
26  */
27 void usage(char* name);
28
29 /**
30  * Function that get all command line option and return those one into a structure
31  * %param argc : Total number of argument onto the command line
32  * %param argv : Contenant of all the command line
33  * %return     : Structure which contains all option given onto command line into this
      structure
34  */
35 Option getOption(int argc, char** argv);
36
37 #endif
```

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <getopt.h>
4
5  #include "game.h"
6  #include "option.h"
7
8  void usage(char* name) {
9      printf("%s [-h]\n\t\t [-f <filePath>] [-t <maxTick>] [-c <number cols] [-r <number rows
       ] [-m] [-s]\n\n", name);
10     printf("\t\t -h : print this help\n");
11     printf("\t\t -f filePath : path to the file to use for the grid\n");
12     printf("\t\t -t maxTick : max time to make the game tick, set it to negative for
       infinite tick\n");
13     printf("\t\t -c : total numner of column\n");
14     printf("\t\t -r : total number of rows\n");
15     printf("\t\t -m : If here we use division by matrice if not we use division by rows\n")
       ;
16     printf("\t\t -s : if -s is use the final grid will be saved\n");
17
18     exit(EXIT_SUCCESS);
19 }
20
21 /**
22  * Private function that define the default value for the option
23  * %return : The option struct with the default value
24  */
25 Option __setDefaultValue() {
26     Option o;
27
```

```
28      o.file_path = "\0";
29      o.max_tick = 100;
30      o.method = DIVIDE_ROWS;
31      o.save_file = false;
32
33      o.rows = MIN_ROWS_SIZE;
34      o.cols = MIN_COLS_SIZE;
35
36      return o;
37  }
38
39  Option getOption(int argc, char **argv) {
40      int opt = 0;
41      Option o = __setDefaultValue();
42
43      while ( (opt = getopt(argc, argv, OPT_LIST)) != -1 ) {
44          switch(opt) {
45              case '?':
46              case 'h':
47                  usage(argv[0]);
48                  break;
49              case 'f':
50                  if ( optarg != 0 )
51                      o.file_path = optarg;
52                  break;
53              case 't':
54                  o.max_tick = atoi(optarg);
55                  break;
56              case 'r':
57                  o.rows = MAX(atoi(optarg), MIN_ROWS_SIZE);
58                  break;
59              case 'c':
60                  o.cols = MAX(atoi(optarg), MIN_COLS_SIZE);
61                  break;
62              case 's':
63                  o.save_file = true;
64                  break;
65              case 'm':
66                  o.method = DIVIDE_MATRICE;
67                  break;
68              default:
69                  exit(EXIT_FAILURE);
70          }
71      }
72
73      if ( argc == 1)
74          fprintf(stderr, "Remember to use -h for help\n");
75
76      return o;
77  }
```

File : Matrix

```
1   /* AUTEUR : REYNAUD Nicolas                                    */
2   /* FICHIER : rows.h                                           */
3   /*------------------------------------------------------------ */
4
5   #ifndef ROWS_H
6   #define ROWS_H
7
8   #include "game_struct.h"
9
10  /**
11   * Function which return the pointer to the string s with the offset "offset"
12   * %param s : Object on which you need to do the offset
13   * %param offset : Size of the offset that need to be done
14   * %param object_size : Size of an object in the object
15   * %return : The offset pointer
16   */
17  char *__offset(char *s, int offset);
18
19  /**
20   * I choose to use the notation for private function for this one, since ONLY main should
         use it
```

```
 *
 * Halt private Function which offset the input buffer, only if needed (i.e. if process == 0 )
 * Cause all buffer ( except the first one ) got a buffer which offset the value of the rows
 * %param my_id : id of the current process
 * %param s : String to offset if needed
 * %param offset : Amount of offset that need to be done
 * %return : The offset pointer of the string
 */
char *__posBufferRecv(int my_id, char* s, int offset);

/**
 * Function which share the border to all other process and get border of other process
 * %param s : Game where you going to share your border and get some
 * %param slice_size : Size of the slice of each subprocess [all should be equal ]
 * %param total_proc : total number of processus
 */
void shareGetBorder(Game *s, int slice_size, int my_id, int total_proc);

#endif
```

```
#include <stdlib.h>
#include <mpi.h>

#include "rows.h"
#include "game_struct.h"

char *__offset(char *s, int offset) {
    return &(*(s + (offset * sizeof(char)) ));
}

char *__posBufferRecv(int my_id, char* s, int offset) {
    if ( my_id != 0 )
        return __offset(s, offset);
    return s;
}


void shareGetBorder(Game *s, int slice_size, int my_id, int total_proc) {

    if ( my_id != 0 ) { /* send bottom row to process on top */
        MPI_Send(__offset(s->board, s->rows),
                 s->rows, MPI_CHAR, my_id - 1, 0, MPI_COMM_WORLD);
    }

    if ( my_id != total_proc - 1) {
        /* Received the top row of the bottom process */
        MPI_Recv(__offset(s->board, s->rows * (slice_size + (my_id != 0))),
                 s->rows, MPI_CHAR, my_id + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        /* Send the bottom row of our slice */
        MPI_Send(__offset(s->board, s->rows * (slice_size - (my_id == 0))),
                 s->rows, MPI_CHAR, my_id + 1, 0, MPI_COMM_WORLD);
    }

    if ( my_id != 0 ) { /* Recv the bottom row of the process at top*/
        MPI_Recv(s->board, s->rows, MPI_CHAR, my_id - 1, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    }

    MPI_Barrier(MPI_COMM_WORLD);
}
```

File : Rows

```
/*------------------------------------------------------------ */
/* AUTEUR : REYNAUD Nicolas                                    */
/* FICHIER : matrix.h                                          */
/*------------------------------------------------------------ */

#ifndef MATRIX_H
#define MATRIX_H

```

```
9  #include "game_struct.h"
10
11 /**
12  * Private function which return a sub-matrix according to a start on x and y and slice
        size
13  * %param src : Source matrix which we extract data
14  * %param startx : Where to start on x
15  * %param starty : Where to start on y
16  * %param xslice_size : Slice size on x
17  * %param yslice_size : Slice size on y
18  * %return : The submatrix of size xslice_size * yslice_size starting at startx, starty
19  */
20 Game *__subMatrix(Game *src, int startx, int starty, int xslice_size, int yslice_size);
21
22 /**
23  * Merge a submatrix with a destination matrix
24  * %param src : Source matrix
25  * %param dest : Destination matrix
26  * %param startx : Where to start on X in dest matrix
27  * %param starty : Where to start merge in dest matrix
28  */
29 void __mergeMatrix(Game *src, Game *dest, int startx, int starty);
30
31 /**
32  * Big function that share all border to other process, share each needed part to
        neightbour and get other border
33  * %param s : Game where the border need to go and need to be shared
34  * %param my_x : My process id on x
35  * %param my_y : My process id on y
36  * %param slice_size : Size of the slice into the first big matrix
37  * %param proc_slice : Number of proc on the columns or rows
38  */
39 void shareMatrixBorder(Game *s, int my_x, int my_y, int slice_size, int proc_slice );
40
41 /**
42  * Gather all submatrix into the original one (after the compute ), this function MUST be
        done by all process
43  * %param g : Game where the final grid will be
44  * %param s : Submatrix to send
45  * %param my_x : My process id on x
46  * %param my_y : My process id on y
47  * %param slice_size : Size of the slice of the principale board
48  * %param proc_slice : Total number of process on a column or a row
49  * %param total_proc : total number of process
50  */
51 void gatherMatrix(Game *g, Game *s, int my_x, int my_y, int slice_size, int proc_slice, int
        total_proc);
52
53
54 /* Schematix of a matrix split into submatrix */
55 /*  Matrix : 3 x 3, Np = 9
56  *  [V][S] [S][V][S] [S][V] < x: 0
57  *  [S][S] [S][S][S] [S][S]
58  *
59  *  [S][S] [S][S][S] [S][S] < x: 1
60  *  [V][S] [S][V][S] [S][V] <
61  *  [S][S] [S][S][S] [S][S] <
62  *
63  *  [S][S] [S][S][S] [S][S] < x: 2
64  *  [V][S] [S][V][S] [S][V] <
65  *   y: 0    y: 1     y: 2
66  *
67  *  S = Buffer
68  *  V = Value of original 3 x 3 matrix
69  */
70 /**
71  * Function which send matrix to other process
72  * %param g : Original game board to split
73  * %param slice_size : Size of the slice of the submatrix
74  * %param proc_slice : Total number of processus by rows or columns
75  * %return : SubMatrix of process 0
76  */
77
```

```h
Game* sendAllSubMatrice(Game *g, int slice_size, int proc_slice);

/**
 * Received a matrix send by process 0
 * %param my_x : My processus id on x
 * %param my_y : My processus id on y
 * %param slice_size : Size of a slice of the principal grid
 * %param proc_slice : Total number of proc on a rows or column
 * %return : return the board including the offset for border sharing
 */
Game* receivedMatrix(int my_x, int my_y, int slice_size, int proc_slice);

#endif
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <mpi.h>

#include "matrix.h"
#include "memory.h"
#include "error.h"
#include "game.h"

Game *__subMatrix(Game *src, int startx, int starty, int xslice_size, int yslice_size) {
    Game *dest = NULL;
    int x, y;

    dest = newGame(yslice_size, xslice_size);
    for ( x = 0; x < xslice_size; x++ )
        for ( y = 0; y < yslice_size; y++ )
            dest->board[POS(x, y, dest)] = src->board[POS(x + startx, y + starty, src)];

    return dest;
}

void __mergeMatrix(Game *src, Game *dest, int startx, int starty) {
    unsigned int x, y;

    for (x = 0; x < src->cols; x++ )
        for (y = 0; y < src->rows; y++ )
            dest->board[POS(x + startx, y + starty, dest)] = src->board[POS(x, y, src)];
}

void shareMatrixBorder(Game *s, int my_x, int my_y, int slice_size, int proc_slice ) {
    int my_id;
    Game *tmp, *buf;
    tmp = NULL;
    buf = newGame(1, slice_size);

    my_id = my_y + my_x * proc_slice;
    /* Send Right - Left*/
    if ( my_y != proc_slice - 1) { /* Send right column */
        tmp = __subMatrix(s, (my_x != 0), slice_size - (my_y == 0), slice_size, 1);
        MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, my_id + 1, 0, MPI_COMM_WORLD)
            ;
        freeGame(tmp);
    }

    if ( my_y != 0 ) { /* get right column and send our left */
        MPI_Recv(buf->board, buf->cols * buf->rows, MPI_CHAR, my_id - 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
        tmp = __subMatrix(s, (my_x != 0), 1, slice_size, 1); /* Start at 1 due to buffer */

        MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, my_id - 1, 0, MPI_COMM_WORLD)
            ;
        __mergeMatrix(buf, s, (my_x != 0), 0);
        freeGame(tmp);
    }

    if ( my_y != proc_slice - 1) { /* get the left column of neighbours */
        MPI_Recv(buf->board, buf->cols * buf->rows, MPI_CHAR, my_id + 1, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
```

```
56          __mergeMatrix(buf, s, (my_x != 0), slice_size + 1 - (my_y == 0));
57      }
58
59      freeGame(buf);
60      buf = newGame(slice_size, 1);
61
62      /* Send Top-Bottom */
63      if ( my_x != proc_slice - 1) {
64          tmp = __subMatrix(s, slice_size - (my_x == 0), (my_y != 0), 1, slice_size);
65          MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, my_id + proc_slice, 0,
                MPI_COMM_WORLD);
66          freeGame(tmp);
67      }
68
69      if ( my_x != 0 ) {
70          MPI_Recv(buf->board, buf->cols * buf->rows, MPI_CHAR, my_id - proc_slice, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
71          tmp = __subMatrix(s, 1, (my_y != 0), 1, slice_size); /* Start at 1 due to buffer */
72
73          MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, my_id - proc_slice, 0,
                MPI_COMM_WORLD);
74          __mergeMatrix(buf, s, 0, (my_y != 0));
75          freeGame(tmp);
76      }
77      if ( my_x != proc_slice - 1 ) {
78          MPI_Recv(buf->board, buf->cols * buf->rows, MPI_CHAR, my_id + proc_slice, 0,
                MPI_COMM_WORLD, MPI_STATUS_IGNORE);
79          __mergeMatrix(buf, s, slice_size + 1 - (my_x == 0), (my_y != 0));
80      }
81
82      freeGame(buf);
83
84      /* Diagonales */
85      if ( my_y != 0 && my_x != 0 ) /* Send to top left */
86          MPI_Send(&s->board[POS(1, 1, s)], 1, MPI_CHAR, my_id - proc_slice - 1, 0,
                MPI_COMM_WORLD);
87
88
89      if ( my_y != proc_slice - 1 && my_x != proc_slice - 1) { /* Send to bottom right */
90          /* Get the bottom right element */
91          MPI_Recv(&s->board[POS(slice_size + (my_x != 0), slice_size + (my_y != 0), s)], 1,
                MPI_CHAR,
92                   my_id + proc_slice + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
93
94          MPI_Send(&s->board[POS(slice_size - (my_x == 0), slice_size - (my_y == 0), s)], 1,
                MPI_CHAR,
95                   my_id + proc_slice + 1, 0, MPI_COMM_WORLD);
96      }
97
98      if ( my_y != 0 && my_x != 0 ) {
99          /* Get the top left element */
100         MPI_Recv(&s->board[POS(0, 0, s)], 1, MPI_CHAR,
101                  my_id - proc_slice - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
102     }
103
104     if ( my_y != proc_slice - 1 && my_x != 0 ) /* Send to top right */
105         MPI_Send(&s->board[POS(1, slice_size - (my_y == 0), s)], 1, MPI_CHAR, my_id -
                proc_slice + 1, 0, MPI_COMM_WORLD);
106
107     if ( my_y != 0 && my_x != proc_slice - 1 ) { /* Send  to bottom left */
108         /* Get the bottom left element */
109         MPI_Recv(&s->board[POS(slice_size + (my_x != 0) ,0, s)], 1, MPI_CHAR,
110                  my_id + proc_slice - 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
111
112         MPI_Send(&s->board[POS(slice_size - (my_x == 0),1, s)], 1, MPI_CHAR, my_id +
                proc_slice - 1, 0, MPI_COMM_WORLD);
113     }
114
115     if ( my_y != proc_slice - 1 && my_x != 0 ) {
116         /* Get the top right element */
117         MPI_Recv(&s->board[POS(0, slice_size + (my_y != 0), s)], 1, MPI_CHAR,
118                  my_id - proc_slice + 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
119     }
```

```
120 
121 }
122 
123 void gatherMatrix(Game *g, Game *s, int my_x, int my_y, int slice_size, int proc_slice, int
        total_proc) {
124     Game* tmp = NULL;
125 
126     if ( my_x != 0 || my_y != 0 ) {
127         tmp = __subMatrix(s, (my_x != 0), (my_y != 0), slice_size, slice_size);
128         MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, 0, 0, MPI_COMM_WORLD);
129         freeGame(tmp);
130     } else {
131         int total_recv;
132         MPI_Status status;
133 
134         /* First merge matrix of process 0 */
135         tmp = __subMatrix(s, 0, 0, slice_size, slice_size);
136         __mergeMatrix(tmp, g, 0, 0);
137         freeGame(tmp);
138 
139         tmp = newGame(slice_size, slice_size);
140         for ( total_recv = 1; total_recv < total_proc; total_recv++ ) {
141             MPI_Recv(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, MPI_ANY_SOURCE, 0,
                    MPI_COMM_WORLD, &status);
142             __mergeMatrix(tmp, g,
143                     (status.MPI_SOURCE / proc_slice) * slice_size,
144                     (status.MPI_SOURCE % proc_slice) * slice_size);
145         }
146 
147         freeGame(tmp);
148     }
149 
150 
151     /* Nobody will go out since process 0 end recv */
152     MPI_Barrier(MPI_COMM_WORLD);
153 }
154 
155 Game* sendAllSubMatrice(Game *g, int slice_size, int proc_slice) {
156     Game *tmp = NULL, *buf = NULL;
157     int total_proc, i, is_x, is_y;
158 
159     MPI_Comm_size(MPI_COMM_WORLD, &total_proc);
160 
161     for ( i = 1; i < total_proc; i++) {
162         is_x = i / proc_slice;
163         is_y = i % proc_slice;
164 
165         tmp = __subMatrix(g, is_x * slice_size, is_y * slice_size, slice_size, slice_size )
                ;
166         MPI_Send(tmp->board, tmp->rows * tmp->cols, MPI_CHAR, i, 0, MPI_COMM_WORLD);
167         freeGame(tmp);
168     }
169 
170     /* Get matrix for the process 0
171      * Add buffer for exchange
172      * Place the subMatrix in place of matrix with buffer
173      */
174     buf = newGame(slice_size + 1, slice_size + 1);
175     tmp =  __subMatrix(g, 0, 0, slice_size, slice_size);
176     __mergeMatrix(tmp, buf, 0, 0);
177     free(tmp);
178 
179     return buf;
180 }
181 
182 Game* receivedMatrix(int my_x, int my_y, int slice_size, int proc_slice) {
183     Game *s, *tmp;
184 
185     tmp = newGame(slice_size, slice_size );
186     s = newGame(slice_size + (my_y != 0) + (my_y != proc_slice - 1),
187                 slice_size + (my_x != 0) + (my_x != proc_slice - 1));
188 
189     MPI_Recv(tmp->board, slice_size * slice_size, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
```

```
            MPI_STATUS_IGNORE);
    __mergeMatrix(tmp, s, (my_x != 0), (my_y != 0));
    freeGame(tmp);

    return s;
}
```