Yannick Omar (352000) | Ting-Jui, Hsu (351218) |
Abdelrahman Abdelkawi (350125)

Assignment 3

Task 1:

a) see code in c)
b) i): parent p of a child c

$$p = c - \lfloor \frac{c}{2} \rfloor - 1 \tag{1}$$

b) ii): left child c_l of parent p

$$c_l = 2(p + 1) - 1 \tag{2}$$

b) iii): right child c_r of parent p

$$c_l = 2(p + 1) \tag{3}$$

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

#define TAG_Bcast_simple 1
#define TAG_Bcast_tree 2
#define TAG_Bcast_type 3
#define typeof __typeof__

//simple broadcast function
void Bcast_simple(void* buffer,int count, MPI_Datatype datatype, MPI_Comm comm){
    int i, NoP_comm, myID;
    MPI_Status status;

    MPI_Comm_size(comm,&NoP_comm);  //get comm size
    MPI_Comm_rank(comm,&myID);      //get own ID in comm

    if (myID==0){                   //only process zero is sender (i.e. 0 broadcasts)
        for (i=1;i<NoP_comm;i++){   //exclude 0 as it is the source
            //send to ith process
            MPI_Send(buffer,count,datatype,i,TAG_Bcast_simple,comm);
            printf("\nI am process %i and I sent to %i\n",myID,i);
        }
    }
    else{
        //reveivcer need to receive size first
        MPI_Probe(0,TAG_Bcast_simple,comm,&status);
        MPI_Get_count(&status,datatype,&count);
        MPI_Recv(buffer,count,datatype,0,TAG_Bcast_simple,comm,&status);
        }
}

int get_parent(int child, int num_nodes){
//compute parent node using the formula as shoow in the solution document
```

```
    if (child!=0){ //only rank zero node doesn't have a parent
        return (child-floor(child/((double) 2))-1);}
    else{
        return -1;}
}

int get_left_child(int parent,int num_nodes){
//compute left child node using the formula as shoow in the solution document
    int child_left;
    child_left=2*(parent+1)-1;

    if (child_left>num_nodes-1){ //return -1 if left child is greater than NoP
        return -1;}
    else {
        return child_left;}
}

int get_right_child(int parent,int num_nodes){
//compute right child node using the formula as shoow in the solution document
    int child_right;
    child_right=2*(parent+1);

    if (child_right>num_nodes-1){//return -1 if right child is greater than NoP
        return -1;}
    else {
        return child_right;}
}

void Bcast_tree(void* buffer,int count, MPI_Datatype datatype, MPI_Comm comm){
//function to broadcast a message using a tree structure for message propagation
    int myID,NoP, parent,child_left,child_right;
    MPI_Status status;

    MPI_Comm_rank(comm,&myID);           //get comm size
    MPI_Comm_size(MPI_COMM_WORLD, &NoP);//get own ID in comm

    parent=get_parent(myID,NoP);             //get corresponding parent element
    child_left=get_left_child(myID,NoP);     //get left child
    child_right=get_right_child(myID,NoP);   //get right child

    /*//just for checking purposes
    printf("\nparent of %d is %d",myID,parent);
    printf("\nleft child of %d is %d",myID,child_left);
    printf("\nright child of %d is %d",myID,child_right);*/

    if (parent!=-1){ //if parent is not -1, i.e. if myID!=0
        //probe first to get value
        MPI_Probe(parent,TAG_Bcast_tree,comm,&status);
        MPI_Get_count(&status,datatype,&count);
        MPI_Recv(buffer,count,datatype,parent,TAG_Bcast_tree,comm,
```

```c
    &status);
        //printf("\n%d:recv from parent: %d\n",myID,parent);
    }

    if (child_left!=-1){

        MPI_Send(buffer,count,datatype,child_left,TAG_Bcast_tree,comm);
        //printf("\n%d:send to left child: %d\n",myID,child_left);
    }
    if (child_right!=-1){

        MPI_Send(buffer,count,datatype,child_right,TAG_Bcast_tree,comm);
        //printf("\n%d:send to right child: %d\n",myID,child_right);
    }

}

int main(int args, char *argv[])
{
    int myRank, NoP;
    MPI_Init(&args, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&myRank);
    MPI_Comm_size(MPI_COMM_WORLD,&NoP);

    /*test value*/
    int N;
    N=3;
    int *sd;
    sd=malloc(N*sizeof(int));
    if (myRank==0){

        sd[0]=1;
        sd[1]=2;
        sd[2]=3;
        Bcast_tree(sd,N,MPI_INT,MPI_COMM_WORLD);
        //Bcast_simple(sd,N,MPI_INT,MPI_COMM_WORLD);
    }
    else{
        Bcast_tree(sd,N,MPI_INT,MPI_COMM_WORLD);
        //Bcast_simple(sd,N,MPI_INT,MPI_COMM_WORLD);
        printf("\nProcess %d: %d %d %d\n",myRank, sd[0], sd[1],sd[2]);
    }

    free(sd); //free test array

    MPI_Finalize();
    return 0;
}
```

Task 2: Glider Movement:

Generation 1:

```
. x . . . . . . . . . . . .
. . x . . . . . . . . . . .
x x x . . . . . . . . . . .
. . . . . . . . . . . . . .
. . . . . . . . . . . . . .
Generation 2:
. . . . . . . . . . . . . .
x . x . . . . . . . . . . .
. x x . . . . . . . . . . .
. x . . . . . . . . . . . .
. . . . . . . . . . . . . .
Generation 3:
. . . . . . . . . . . . . .
. . x . . . . . . . . . . .
x . x . . . . . . . . . . .
. x x . . . . . . . . . . .
. . . . . . . . . . . . . .
Generation 4:
. . . . . . . . . . . . . .
. x . . . . . . . . . . . .
. . x x . . . . . . . . . .
. x x . . . . . . . . . . .
. . . . . . . . . . . . . .
Generation 5:
. . . . . . . . . . . . . .
. . x . . . . . . . . . . .
. . . x . . . . . . . . . .
. x x x . . . . . . . . . .
. . . . . . . . . . . . . .
```

Code:

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

#define TAG_CycleRst 1  //tag for sending/receiving results after each cycle
#define TAG_Exchange 2  //tag for exchange of ghost points between different domains
#define TAG_CartPos 3   //tag for sending/receiving ranks in the cartesian grid

/*This program simulates Conway's game of life in parallel.
The domain is decompositioned, such that evaluation of
```

```
cells
can be done in parallel. This is accomplished by the workers.
A master process receives the results to facilitate correct
output of the result.
For reasons of simplicity, the initial configuration is implied
on a process level*/

//error depends on width!
int main(int args, char *argv[])
{
    int myID, NoP;
    MPI_Init(&args,&argv);                  //init MPI session
    MPI_Comm_rank(MPI_COMM_WORLD,&myID); //get own ID
    MPI_Comm_size(MPI_COMM_WORLD,&NoP);  //get number of processes

    //init
    int dims[0],period[0],reorder,w,domain_size[NoP-1],k,i,j,h,r;
    MPI_Status status;
    w=14; //board size (square board!)
    k=7; //number of timesteps


    //domain decomposition
    r=w; //remainder
    for (i=0;i<NoP-2;i++){ //-2 because one is master and one will fill up the rest
        domain_size[i]=floor(w/(double) (NoP-1));
        r-=domain_size[i];
    }
    domain_size[NoP-2]=r;

    dims[0]=NoP-1;   //number of domains, Number of process - master process
    period[0]=1;     //periodic boundary conditions
    reorder=1;       //reordering of processes in Cart

    int excl[0];     //process to exclude from new group
    excl[0]=0;       //exclude master(0)
    MPI_Group GroupWorker,GroupWorld;
    MPI_Comm CommWorker;

    MPI_Comm_group(MPI_COMM_WORLD,&GroupWorld);       //get group of MPI_COMM_WORLD
    MPI_Group_excl(GroupWorld,1,excl,&GroupWorker); //Create Group of worker without master

    //Create new communicator corresponding to group of workers
    if (myID!=0)
        MPI_Comm_create(MPI_COMM_WORLD,GroupWorker,&
CommWorker);
 else
     MPI_Comm_create(MPI_COMM_WORLD,MPI_GROUP_EMPTY,&CommWorker);


 if (myID!=0){

     MPI_Comm Cart;
     MPI_Cart_create(CommWorker,1,dims,period,reorder,&Cart);

     int sum,ngbor_top,ngbor_bot,jp1,jm1, crt_rank,crt_rank_var,mysize;

     MPI_Comm_rank(CommWorker,&crt_rank); //get rank in Cart

     mysize=domain_size[crt_rank];        //get size of own domain

     //own domain includes ghost points (no distinction here) to make computation simple
     int myConfig[mysize+2][w],ngborConfig_top[w],ngborConfig_bot[w],myNewConfig[mysize+2][w];

     //init to zero
     for (j=0;j<mysize+2;j++){
         for (i=0;i<w;i++){
             myConfig[j][i]=0;
             myNewConfig[j][i]=0;
         }
     }

     //initial configuration for the given special case
     //0 == dead, 1==alive
     if (crt_rank==0){
         myConfig[1][0]=0;myConfig[1][1]=1;myConfig[1][2]=0;
         myConfig[2][0]=0;myConfig[2][1]=0;myConfig[2][2]=1;
         myConfig[3][0]=1;myConfig[3][1]=1;myConfig[3][2]=1;
     }

     //get top and bottom neighbor
     MPI_Cart_shift(Cart,0,-1,&crt_rank_var,&ngbor_top);
     MPI_Cart_shift(Cart,0,1,&crt_rank_var,&ngbor_bot);


     for (i=0;i<k;i++){ //loop over number of generations to play

         /*send and recv ghost points: first send
```

```
to top and receive from bottom,
       the vice versa, to avoid deadlocks*/
       MPI_Sendrecv(myConfig[1],w,MPI_INT,
ngbor_top,TAG_Exchange,ngborConfig_bot,w,MPI_INT,
ngbor_bot,TAG_Exchange,CommWorker,&status);
       MPI_Sendrecv(myConfig[mysize],w,MPI_INT,
ngbor_bot,TAG_Exchange,ngborConfig_top,w,MPI_INT,
ngbor_top,TAG_Exchange,CommWorker,&status);

       //transfer ghost points to own domain
       memcpy(myConfig[0],ngborConfig_top,w*
sizeof(int));
       memcpy(myConfig[mysize+1],ngborConfig_bot,
w*sizeof(int));

       for (h=1;h<=mysize;h++){ //loop over row

          for (j=0;j<w;j++){    //loop over every
 cell in row

             //apply horizontal periodocity and
 find horizontal neighbors
             sum=0;
             if (j!=0)
                jm1=j-1;
             else
                jm1=w-1;

             if (j!=w-1)
                jp1=j+1;
             else
                jp1=0;

             //calculate the sum, because of
binary values, the sum indicates
             //the number of live cells
             sum+=myConfig[h-1][jm1]+myConfig[h
-1][j]+myConfig[h-1][jp1];
             sum+=myConfig[h][jm1]+myConfig[h][
jp1];
             sum+=myConfig[h+1][jm1]+myConfig[h
+1][j]+myConfig[h+1][jp1];

             //apply rules given in the task
             if ((myConfig[h][j]==1) && ((sum
<2) || (sum>3)))
                myNewConfig[h][j]=0;
             else if ((myConfig[h][j]==1) && ((
sum==2) || (sum==3)))
                myNewConfig[h][j]=1;
             else if ((myConfig[h][j]==0) && (
sum==3))
                myNewConfig[h][j]=1;
             } //for j

          } //for h
          if (i!=0){ //to print initial
```

```
configuration
                    for (j=0;j<mysize+2;j++){ //
transfer linewise
                        memcpy(myConfig[j],
myNewConfig[j],w*sizeof(int));
                    } //for j
            } //for i


            //just to check whether sending worked
 correctly
            /*for (h=1;h<mysize+1;h++){
                for (j=0;j<w;j++){
                    printf("%d%d ",crt_rank,
myConfig[h][j]);
                }
                printf("\n");
            }*/

            //send own rank in cartesian
communicator to allow correct printing
            MPI_Send(&crt_rank,1,MPI_INT,0,
TAG_CartPos,MPI_COMM_WORLD);
            for (h=1;h<=mysize;h++)  //send actual
 data linewise (facilitates easier printing)
                MPI_Send(myConfig[h],w,MPI_INT,0,
TAG_CycleRst,MPI_COMM_WORLD);
        }

        //free groups and communicator
        MPI_Comm_free(&CommWorker);
        MPI_Group_free(&GroupWorker);
        MPI_Group_free(&GroupWorld);

} //endif
 else{

        int rk,counter,recv_size,CurConfig[w][w],
RecvConfig[w];

        //int RecvConfig[recv_size][w];

        for (i=0;i<k;i++){ //iteration over all
generations
            counter=0;

            for (j=1;j<NoP;j++){

                recv_size=domain_size[j-1]; //domain
of process which sends
                //receive rank
                MPI_Recv(&rk,1,MPI_INT,j,TAG_CartPos,
MPI_COMM_WORLD,&status);

                for (h=0;h<recv_size;h++){  //
iteration over domain, receive linewise
                    MPI_Recv(RecvConfig,w,MPI_INT,j,
    TAG_CycleRst,MPI_COMM_WORLD,&status);
                    memcpy(CurConfig[counter],
    RecvConfig,sizeof(int)*w); //build global result
                    counter+=1;
               } //for h
          } //for j

          //print results
          for (j=0;j<w;j++){
               for (h=0;h<w;h++){
                    if (CurConfig[j][h]==0)
                         printf(" .");
                    else
                         printf(" x");
               } //for h
               printf("\n
    -------------------------------------------------\
    n");
          } //for j
          printf("\n
    -------------------------------------------------\
    n\n");
     } //for i
 } //else


    MPI_Finalize();
}
```