CMPE 300 PROJECT
PARALLEL MAZE SOLVER

UMUT AFACAN
2010400141

INTRODUCTION

The project “Parallel Maze Solving” has written by using C language with MPI on Mac OSX 10.10 x64 machine. The program solves a maze ,finds a path from given input maze. On contrary to sequential maze solving algorithm, this project aims to use multi processor to increase speed.

PROGRAM INTERFACE

The program has no special interface, it is used by command line. Only restriction is that if the matrice size [NxN], N must be divisible by proccesor count-1. The one for the master.

mpicc –g programname.c –o programname

mpiexec –n proc_count ./programname input.txt output.txt

INPUT OUTPUT

The program takes first argument as an input then read to internal memory. The input’s first line must be matrice size, the rest lines needs to represent matrice. The output file is the almost same as input, it has not contain matrice size in the first line.

PROGRAM STRUCTURE

The structure contains 2 main parts which are Master-Work, Slave-Work. Program is design to work with 1 master processor and multiple slaves. The slave count should divide the input matrice size to integer. General algorithm is likely to sequential maze solving however by splitting input to multiple slaves gains more speed. For the paralleling, the MPI_send and MPI_recv methotds are used.

Master’s algorithm:

Master has 5 main job to do, these are

-Reading input

-Distributing input to slaves

-Controlling Slaves

-Gathering outputs from slaves

-Writing output to a file

Firstly, master reads NxN matrice from input file to an 2-dim array. Then it splits the 2-dim array to smaller 2-dim sub arrays. After that by using MPI_send, it disributes data to slaves with size information. When the distribution is finished, it gives start signal for first iteration. Then it waits for messages from slaves about whether they marked a deadend or not. If a deadend exists, the master

gives signal of next iteration so on. Until none of the slaves find deadend, iterations continue. When the this part is completed, the master waits to collect outputs from the slaves. After all data collected, solution maze is written to a output file.

Slaves Algorithm:

Slaves' jobs can be listed as follows:

1-Take input from master

2-a) Wait for iteration

2-b)Communicate with neighboors

2-c)Traverse the sub-matrice

2-d)Give signal to the master

3-Send output data to master

Firstly, each slave takes an input from the master saves to a 2-dim array. After that, comes to while to wait iteration signals. When the signal is given by the master a slave start to iteration. The initial step of iteration is the communicate with neighboors. Each slave takes the border line from its neighboors. To avoid deadlock on this step, the data flow has a direction. Firstly, upper lines takes the input needed, then lowers.(Fig-1) After the communication, slaves starts to traverse the 2-dim arrays by calling checkNeighboor function for each cell. This functions duties is listed below:

-check cell's own value

-check upper cell

-check right cell

-check lower cell

-check left cell

-accourding to "0"'s count returns the result.

If the other cells is in border, looks border arrays which is provided as parameter after the communication step.

Accoring to checkNeighboors result, slaves marks a cell or not. Also saves a deadend exist message to send master.

After the traversing is finished, slaves sends messages to master and waits for the next signal. If next signal is 1, they start communicate again and so on.

If the signal is 0, they breaks the loop and starts to send data they have to master.

Conclusion

This program solves a maze by using multiple processors as it desired. By the using paralleling, the execution time is reduced almost 1 to 1/p. The communication part maybe slower a bit in real time however the basic operation is the checking cell is well distrubuted to slaves. The program has no input error detection. Also running by not divisible number of proccessor will crash program. As a result, the aim of project is completed. By communicating between slaves, storing sub-matrices into slaves, collecting data,waiting and signalling helps to understand concepts of parallel programming.

APPENDIX

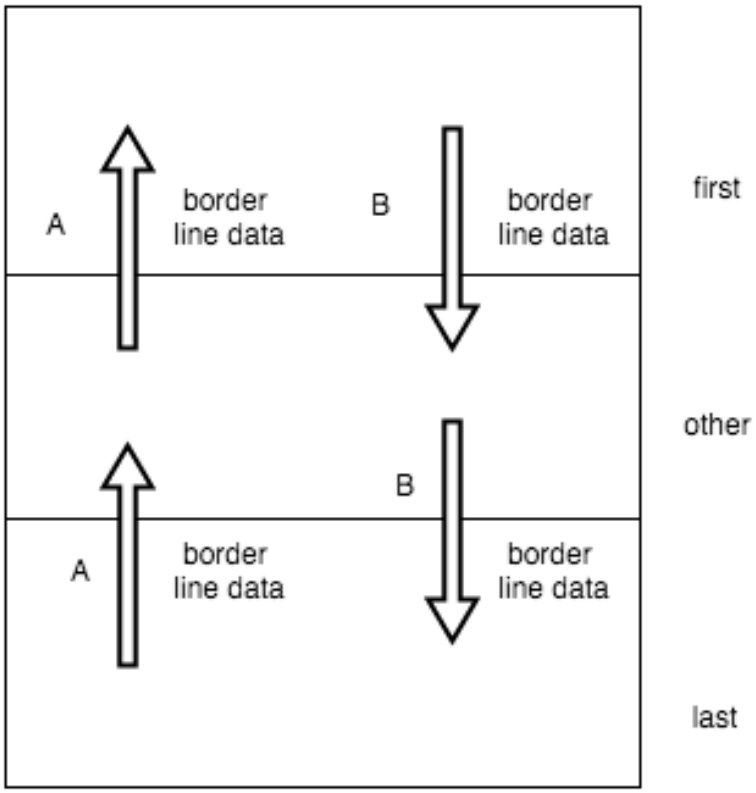

Fig-1

Firstly, A direction communication happens from top to below, then B direction runs. For ex: (Direction A)First slave takes info from below. Then second from its below, so on, until to the last slave.

CODE:

```
/*
Student Name: UMUT AFACAN
Student Number: 2010400141
Compile Status: Compiling
Program Status: Working
Notes: Matrice size should be divisible by proccessor count-1.
the 1 for master.
*/

#include <stdio.h>
#include <mpi.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include <limits.h>

#define A_DIRECTION 30
#define B_DIRECTION 31

#define NEXT_ITERATION 15
#define DEADEND_SIGNAL 14


int** maze;
int matrice_size, row_size, row_num_fproc;
int** maze_finish;
int** array_signal;



int checkCells(int i,int j,int proc_id,int **array, int row, int col,int *upperLine, int *lowerLine)
{
	if(array[i][j] == 0)
		return 0;

	int proc_count = col/row;
	int count=0;
	//check above
	if(i>0)
	{
		if(array[i-1][j] == 0)
			count++;
	}
	else if(proc_id>1)
	{
		if(upperLine[j]== 0)
			count++;
	}
	//check right
	if(j<col-1)
	{
		if(array[i][j+1] == 0)
			count++;
	}
	//check below
	if(i<row-1)
	{
		if(array[i+1][j] == 0)
			count++;
	}else if(proc_id < proc_count)
	{
		if(lowerLine[j] == 0)
			count++;
	}

		// check left
	if(j>0)
	{
		if(array[i][j-1] == 0)
			count++;
	}
	if (count == 3)
	{
		return 2;
	}else
		return 1;

}
```

```
int **alloc_2d_int(int rows, int cols) {
   int *data = (int *)malloc(rows*cols*sizeof(int));
   int **array= (int **)malloc(rows*sizeof(int*));
   for (int i=0; i<rows; i++)
      array[i] = &(data[cols*i]);

   return array;
}


int main (int argc, char **argv) {
                int num_procs, my_id;

                MPI_Init(&argc, &argv);

                MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
                MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

 if(my_id == 0) {
                                // MASTER - NODE

   FILE *file = fopen(argv[1], "r");
   fscanf(file, "%d", &matrice_size);
   maze = alloc_2d_int(matrice_size, matrice_size);

   for (int i = 0; i < matrice_size; i++)
      for (int j = 0; j < matrice_size; j++)
         if (!fscanf(file, "%d", &maze[i][j]))
            break;

   fclose(file);

   //output file
   char output[80] = "";
   char *val = strtok(argv[2], ".");
   strcat(output, val);
   val = strtok(NULL, ":");
   strcat(output, ".txt");

   // redirecting stdout to file.
   stdout = freopen(output, "w", stdout);



   row_num_fproc = matrice_size / (num_procs - 1);
  // printf("num_procs %d\nrow_num_fproc %d\nmatrice_size %d\n ",num_procs, row_num_fproc, matrice_size);

   for(int i = 1; i < num_procs; i++) {
     int** node_maze = alloc_2d_int(row_num_fproc, matrice_size);

     for(int k = 0; k < row_num_fproc; k++) {
       for(int l = 0; l < matrice_size; l++) {
         node_maze[k][l] = maze[k + ((i - 1) * row_num_fproc)][l];
       }
     }


     //MPI_Send(&i, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
     MPI_Send(&row_num_fproc, 1, MPI_INT, i, 0, MPI_COMM_WORLD);
     MPI_Send(&matrice_size, 1 , MPI_INT, i, 1, MPI_COMM_WORLD);
     MPI_Send(&(node_maze[0][0]), row_num_fproc * matrice_size, MPI_INT, i, 2, MPI_COMM_WORLD);
     //break;
   }

   array_signal = alloc_2d_int(matrice_size,matrice_size);


   int flag=1;

                for (int i = 1; i < num_procs ; ++i)
                {
                                MPI_Send(&flag,1,MPI_INT,i,15,MPI_COMM_WORLD);
   }

   int iteration = 0;
   while(flag == 1)
   {
                //printf("iteration count : %d ----\n",iteration++);
                //signals iteration
                flag=0;
                for (int i = 1; i < num_procs ; ++i)
                {
```

```
                int deadend=0;
                MPI_Recv(&deadend,1,MPI_INT,i,DEADEND_SIGNAL,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
                if (deadend == 1)
                {
                        //if any deadend exist signals next iteration
                        flag=1;
                }
        }

        for (int i = 1; i < num_procs ; ++i)
        {

                MPI_Send(&flag,1,MPI_INT,i,NEXT_ITERATION,MPI_COMM_WORLD);
        }


 }


 //collects datas from the slaves
 int** maze_finish = alloc_2d_int(matrice_size,matrice_size);

 for(int i = 1; i < num_procs; i++) {
  int** node_maze = alloc_2d_int(row_num_fproc, matrice_size);

  MPI_Recv(&(node_maze[0][0]), row_num_fproc*matrice_size, MPI_INT, i, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  for(int k = 0; k < row_num_fproc; k++) {
   for(int l = 0; l < matrice_size; l++) {
    maze_finish[k + ((i - 1) * row_num_fproc)][l] = node_maze[k][l];
   }
  }

 }




 for (int i = 0; i < matrice_size; ++i)
 {
  for (int j = 0; j < matrice_size; ++j)
  {
   printf("%d ", maze_finish[i][j]);
  }
  printf("\n");
 }



} else {
 int row_num_fproc, matrice_size;
 MPI_Recv(&row_num_fproc, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
 MPI_Recv(&matrice_size, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

 int** node_maze = alloc_2d_int(row_num_fproc, matrice_size);
 MPI_Recv(&(node_maze[0][0]), row_num_fproc * matrice_size, MPI_INT, 0, 2, MPI_COMM_WORLD, MPI_STATUS_IGNORE);


 //largest iteration parsing all data controlled by master
 int flag;
 MPI_Recv(&flag,1,MPI_INT,0,15,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
 while(flag)
 {

        int lowerLine[matrice_size];
        int upperLine[matrice_size];
        //if first slave
           if(my_id == 1){

                 //get lowerline from below neighboor
                MPI_Recv(&lowerLine,matrice_size,MPI_INT,my_id+1,30,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                //send to lower neighboor
                MPI_Send(&node_maze[row_num_fproc-1][0],matrice_size,MPI_INT,my_id+1,31,MPI_COMM_WORLD);


                }
                //other slaves
                else if(my_id<num_procs-1)
                {
                        //send upper neighboor
                        MPI_Send(&node_maze[0][0],matrice_size,MPI_INT,my_id-1,30,MPI_COMM_WORLD);
                        //get from lower neighboor
                        MPI_Recv(&lowerLine,matrice_size,MPI_INT,my_id+1,30,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

```
                                        //get from upper
                                        MPI_Recv(&upperLine,matrice_size,MPI_INT,my_id-1,31,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                                        //send to lower;
                                        MPI_Send(&node_maze[row_num_fproc-1][0],matrice_size,MPI_INT,my_id+1,31,MPI_COMM_WORLD);



                                }//last slave
                                else
                                {
                                        //send to upper
                                        MPI_Send(&node_maze[0][0],matrice_size,MPI_INT,my_id-1,30,MPI_COMM_WORLD);

                                        //get from upper
                                        MPI_Recv(&upperLine,matrice_size,MPI_INT,my_id-1,31,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

                                }


                int deadend=0;
                  for(int i = 0; i< row_num_fproc  ;i++)
                  {
                                for (int j = 0; j < matrice_size; j++)
                                {

                                        int res = checkCells(i,j,my_id,node_maze,row_num_fproc,matrice_size,upperLine,lowerLine);

                                        if(2 == res )
                                        {
                                                node_maze[i][j]=0;
                                                deadend=1; // deadend exists
                                        //      printf("deadend exists\n");
                                        }

                                }
                  }



                  //sends deadend data to master
                  MPI_Send(&deadend,1,MPI_INT,0,DEADEND_SIGNAL,MPI_COMM_WORLD);

                  //receives next iteration signal
                  MPI_Recv(&flag,1,MPI_INT,0,NEXT_ITERATION,MPI_COMM_WORLD,MPI_STATUS_IGNORE);


                }
                  //sends output to master
    MPI_Send(&(node_maze[0][0]),row_num_fproc * matrice_size, MPI_INT, 0, 3, MPI_COMM_WORLD);
  }


  MPI_Finalize();

  return 0;
}
```