Παράλληλα Συστήματα
Εργασία 2016-17: *Conway's Game of Life*

Μανούσος Τοράκης
A.M. 1115201400201
Valentin Ivanov
A.M. 1115201400049

8 Οκτωβρίου 2017

1 Εισαγωγή

Έχουν υλοποιηθεί όλα τα ζητούμενα της εργασίας εκτός απο την επίδειξη συμπεριφοράς με το *paraver* και την χρονομέτρηση του προγράμματος *Cuda* λόγω τεχνικών προβλημάτων. Επίσης έχουν ικανοποιηθεί οι απαιτήσεις που ανέλυσε ο διδάσκοντας κατά την παρουσίαση της εργασίας.
Συγκεκριμένα:
- Τα δεδομένα διαμεροίζονται σε *blocks*.
- $Non - blocking$ επικοινωνία.
- Χρήση *derived data type* για την αποστολή στήλης.
- Χρήση $Virtual\ topology$ για βελτιστοποίηση της επικοινωνίας.
- Προκαθορισμός των επικοινωνιών που γίνονται σε κάθε επανάληψη με MPI_Send_init και MPI_Recv_init.
- Μαζική έναρξη και αναμονή των παραπάνω επικοινωνιών με $MPI\_Startall$ και $MPI\_Waitall$. - $MPI\_Allreduce$ για τον έλεγχο μη αλλαγή πλέγματος κάθε n επαναλήψεις.

Επιπλέον έχει προστεθεί η εξής λειτουργικότητα στοχεύοντας κυρίως στην καλύτερη διεξαγωγή μετρήσεων και δοκιμών γενικότερα:
- Δυνοτότητα αυτόματης παραγωγής *input* αρχείων μέσω ̈ προγραμμάτων στον κατάλογο *test_generators*.

```
cd test_generators
make
./fgen lines columns alive_percent file_name
```

Το *gui_gen.c* παρέχει γραφική παρουσιάση του *input* που δημιουργείται χειροκίνητα απο τον χρήστη, αλλά δεν είναι πρακτικό για μεγάλα μεγέθη πινάκων και χρησιμοποιήθηκε για αρχικές ¨μικρές' δοκιμές ωστέ να επαληθευτεί η ορθή λειτουργία του προγράμματος.
- Δυνατότητα ευροίας παραμετροποίησης του εκτελέσιμου MPI ή $MPI\text{-}OPENMP$ προγράμματος.

```
make
mpiexec -n <process_num> ./gol_mpi -f <filename> -l <N> -c <M>
-n <max_loops> -r <reduce_rate>

mpiexec -n <process_num> ./gol_mpi_openmp -f <filename> -l <N> -c <M>
-n <max_loops> -r <reduce_rate> -t <threads>
```

όπου *reduce_rate* το πλήθος των επαναλήψεων μετά το οποίο ελέγχεται για αλλαγή πλέγματος. Αν δωθεί αρνητικό τιμή (π.χ. -1), τότε δεν γίνεται ποτέ έλεγχος για αλλαγή πλέγματος.

Κανένα όρισμα δεν είναι υποχρεωτικό και για όποιο δεν δωθεί χρησιμοποιείται η $default$ τιμή του:

```
#define DEFAULT_N 420
#define DEFAULT_M 420
#define MAX_LOOPS 200
#define REDUCE_RATE 1
#define NUM_THREADS 2
```

Για τα *-l -c*, αν δοθούν θα πρέπει να δοθούν και τα δύο αλλίως παίρνουν $default$ τιμές.
Αν δεν δωθεί αρχείο, παράγεται ένα αυτόματα με βάση το μέγεθος, με περίπου του μισούς οργανισμούς ζωντανούς, και αποθηκεύεται στον κατάλογο *generated_tests* με όνομα σχετικό με το μέγεθος και το *timestamp* της δημιουργίας.

- Αποστολή των κομματιών του *input* πίνακα στις σωστές διεργασίες απο τον *master* (δεν συμμετέχει στην χρονομέτρηση της εκτέλεσης), δηλαδή δεν παράγει απλά η κάθε διεργασία τον δικό της πίνακα, αλλά εκτελείται κανονικά το πρόγραμμα με το *input* που έχει δωθεί ή παραχθέι.
- Αποστολή τελικών *blocks* απο όλες τις διεργασίες στον *master* με μια συνάρτηση τύπου *gather*, με σκοπό την εκτύπωση του τελικού πίνακα αν είναι $flag\ PRINT\_FINAL$ είναι 1 (παρακάτω). Δεν συμμετέχει στην χρονομέτρηση της εκτέλεσης. - Τέλος υπάρχουν κάποια $defined\ flags$ στον κώδικα που σχετίζονται με τις εκτυπώσεις και είναι ιδιαίτερα χρήσιμα για *debug* αλλά και για την επίδειξη της σωστής λειτουργίας του προγράμματος (διαμεροισμός δεδομένων, *blocks*, εκτύπωση χρόνων, εκτύπωση του πίνακα κ.α.)

```
#define DEBUG 0
#define INFO 0
#define STATUS 1
#define TIME 1
#define PRINT_INITIAL 0
#define PRINT_STEPS 0
#define PRINT_FINAL 0
```

Το $DEBUG$ ενεργοποιεί εκτυπώσεις σχετικές με τον διαμεροισμό των δεδομένων σε βλοςκς (το κομμάτι κάθε διεργασίας), την *virtual* τοπολογία που δημιουργείται (*coordinates* και *neighbours*) και την αρχική αποστολή του πίνακα απο τον *master* στις υπόλοιπες διεργασίες (εκτύπωση σε αρχεία στον κατάλογο *test_outs*. Δεν δουλεύει τόσο καλά για μεγάλους πίνακες λόγω ταυτόχρονης εκτύπωσης)
Το $INFO$ ενεργοποιεί γενικού σκοπού εκτυπώσεις σχετικά με την κατάσταση της εκτέλεσης του προγράμματος.
Το $STATUS$ ενεργοποιεί λίγες εκτυπώσεις σχετικά με την κατάσταση του τρέχοντος παιχνιδιού (πότε ξεκινάει, πότε/γιατί τερματίζει).

2 Σχεδιασμός Διαμοιρασμού Δεδομένων

Όπως αναφέρεται και παραπάνω ο διαμοιρασμός δεδομένων γίνεται σε *blocks*. Κάθε διεργασία αναλαμβάνει το κομμάτι του πίνακα απο την γραμμή *row_start* εώς και *row_end* και απο την στήλη *col_start* εώς και *col_end*, τα οποία υπολογίζονται με βάση:
- Την διαμέριση του πίνακα σε *blocks*
- Τις συντεταγμένες της εκάστοτε διεργασίες στην *virtual topology*

```
...
int rows_per_block = N / line_div;
int cols_per_block = M / col_div;
int blocks_per_row = N / rows_per_block;
int blocks_per_col = M / cols_per_block;
...
MPI_Comm virtual_comm;
int ndimansions, reorder;
int dimansion_size[2], periods[2];
ndimansions = 2;
dimansion_size[0] = blocks_per_row;
dimansion_size[1] = blocks_per_col;
periods[0] = 1;
periods[1] = 1;
reorder = 1;

int ret = MPI_Cart_create(MPI_COMM_WORLD, ndimansions, dimansion_size, periods, reorde
...
ret = MPI_Cart_coords(virtual_comm, my_rank, ndimansions, my_coords);
...
row_start = my_coords[0] * rows_per_block;
row_end = row_start + rows_per_block - 1;
col_start = my_coords[1] * cols_per_block;
col_end = col_start + cols_per_block - 1;
```

Επίσης, ο καθορισμός των 8 γειτονικών *ranks* για κάθε διεργασία γίνεται με την χρήση των συντεταγμένων της στην τοπολογία, και με την συνάρτηση *MPI_Cart_rank*, μιας και είναι ενεργοποιημένο το *reorder* για την τοπολογία μας και τα *ranks* δεν μπορούν να υπολογιστούν με προφανή τρόπο.

```
...
//calculate the rank of the up-left neighbour
neighbour_coords[0] = my_coords[0] - 1;//go up
neighbour_coords[1] = my_coords[1] - 1;//go left
ret = MPI_Cart_rank(virtual_comm, neighbour_coords, &rank_ul);
...
```

3 Σχεδιασμός και υλοποίηση MPI κωδικα

Στον κατάλογο *gol_lib* υπάρχει κώδικας που αφορά το *logic* του *Game of Life* καθώς και ορισμένες βοηθητικές συναρτήσεις/δομές για την εκτέλεση (δέσμευση/αρχικοποίηση/απελευθέρωση δομής, *populate* του πίνακα για κάθε *loop* κ.α.)
Αξίζει να σημειωθεί ότι ο πίνακας έχει δεσμευτεί σε συνεχόμενη μνήμη, και στην συνέχεια ¨φαίνεται' στην *main* να είναι δισδιάστατος χάρη σε κατάλληλες αναθέσεις με *pointers*.

```
//file: gol_lib/gol_array.c
gol_array* gol_array_init(int lines, int columns)
{
        //allocate one big flat array, so as to make sure that the memory
        //is continuous in our 2 dimensional array
        short int* flat_array = calloc(lines*columns, sizeof(short int));
        assert(flat_array != NULL);

        short int** array = malloc(lines*sizeof(short int*));
        assert(array != NULL);
        int i;
        //make a 2 dimension array by pointing to our flat 1 dimensional array

        for (i=0; i<lines; i++)
                array[i] = &(flat_array[columns*i]);
        ...
}
```

Ο βασικός κώδικας για το MPI βρίσκεται στο αρχείο *gol_mpi.c* και ικανοποιεί τα ζητούμενα (μείωση αδρανούς χρόνου/περιττών υπολογισμών, επικάλυψη επικοινωνίας, χρήση *datatypes*, έλεγχος για μη αλλαγή πλέγματος κάθε ν επαναλήψεις). Οργανώνεται σε 3 τμήματα:

```
SECTION A
MPI Initialize
Blocks computation
Matrix allocation, initialization and 'scatter' to processes
Column derived data type

SECTION B
Create our virtual topology
For the current proccess in the new topology
Find the ranks of its 8 neighbours
Calculate its piece/boundaries of the game matrix

SECTION C
Define communication (requests) that will be made in EVERY loop with MPI_Init
Game of life loop (with timing)
x8 MPI_ISend (MPI_Startall)
x8 MPI_IRecv (MPI_Startall)
Calculate 'inner' cells
Wait for IRecvs to complete (MPI_Waitall)
Calculate 'outer' cells
MPI_Allreduce for master to see if there was a change or not
If (change && repeats < max_repeats)
        repeat
else
        finish
Wait for ISends to complete (MPI_Waitall)
```

```
//file: gol_lib/gol_array.h

#ifndef GOL_ARRAY_H
#define GOL_ARRAY_H

#include <stdio.h>
#include <string.h>
#include <assert.h>
#include <stdlib.h>
#include <time.h>
#include "functions.h"

struct gol_array
{
        short int* flat_array;
        short int** array;
        int lines;
        int columns;
};

typedef struct gol_array gol_array;

gol_array* gol_array_init(int lines, int columns);
void gol_array_free(gol_array** gol_ar);
void gol_array_read_input(gol_array* gol_ar);
void gol_array_read_file(char* filename, gol_array* gol_ar);
void gol_array_generate(gol_array* gol_ar);

#endif
```

```
//file: gol_lib/functions.h

#ifndef FUNCTIONS_H
#define FUNCTIONS_H

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "gol_array.h"

void print_array(short int** array, int N, int M);
int populate(short int** array1, short int** array2, int N, int M, int i, int j);
int num_of_neighbours(short int** array, int N, int M, int row, int col);
void print_neighbour_nums(short int** array, int N, int M);
void get_date_time_str(char* datestr, char* timestr);

#endif
```

4 Μετρήσεις χρόνου εκτέλεσης

Για τις εκτελέσεις τόσο του MPI χρησιμοποιήθηκαν 8 διαθέσιμοι υπολογιστές της σχολής. Ο αριθμός επαναλήψεων που χρησιμοποιήθηκε για τις μετρήσεις ήταν 500 και έλεγχος για αλλαγή πλέγματος γινόταν κάθε 20 επαναλήψεις. Τα αρχεία εισόδου που χρησιμοποιήθηκαν παράχθηκαν απο τον δικό μας *test_genarator* αλλά ήταν πολύ μεγάλα για να συμπεριληφθούν στο παραδοτέο. Κάθε *input* χρησιμοποιήθηκε σε 3 εκτελέσεις, και κρατήσαμε τους καλύτερους/σταθερότερους χρόνους.

Table 1: MPI timings with reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	0.902603	0.497823	0.477089	0.292184	0.957296
256 x 256	3.596160	1.816599	1.048347	0.678046	1.156213
512 x 512	14.417825	7.387163	3.858929	2.075757	2.696575
1024 x 1024	57.755004	29.398613	14.959850	7.545417	4.936497
2048 x 2048	416.464362	165.001508	69.308270	85.217432	21.392594
4096 x 4096	1068.252632	703.898145	361.591368	161.884114	94.790958

Table 2: MPI speedups with reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	1	1.89607	2.44034	3.1141	1.00488
256 x 256	1	1.9509	3.492	5.41487	3.228
512 x 512	1	1.95181	3.82771	7.2018	5.27432
1024 x 1024	1	1.86618	3.37505	7.67126	10.33691
2048 x 2048	1	0.3981	0.84901	1.47421	3.29755
4096 x 4096	1	1.72066	2.90387	5.13623	15.09407

Table 3: MPI efficiency with reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	1	0.94803	0.61008	3.1141	1.00488
256 x 256	1	0.97545	0.873	0.67685	0.20174
512 x 512	1	0.9759	0.95692	0.90022	0.32964
1024 x 1024	1	0.93309	0.84375	0.95891	0.64606
2048 x 2048	1	0.19905	0.21225	0.18427	0.20609
4096 x 4096	1	0.86032	0.72597	0.64203	0.94337

Table 4: MPI timings without reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	0.904479	0.477023	0.370649	0.290453	0.900041
256 x 256	3.595032	1.842699	1.029464	0.663919	1.113688
512 x 512	14.420917	7.388497	3.767523	2.002380	2.734167
1024 x 1024	57.923903	31.038880	17.162390	7.550767	5.603595
2048 x 2048	381.608061	145.501710	68.225202	39.291550	17.565808
4096 x 4096	1027.920754	597.398473	353.983091	200.131577	68.100946

Table 5: MPI speedups without reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	1	1.89607	2.44034	3.1141	1.00488
256 x 256	1	1.9509	3.492	5.41487	3.228
512 x 512	1	1.95181	3.82771	7.2018	5.27432
1024 x 1024	1	1.86618	3.37505	7.67126	10.33691
2048 x 2048	1	2.62271	5.59335	9.71222	21.72449
4096 x 4096	1	1.72066	2.90387	5.13623	15.09407

Table 6: MPI efficiency without reduce

size \processors	***1***	***2***	***4***	***8***	***16***
128 x 128	1	0.94803	0.61008	3.1141	1.00488
256 x 256	1	0.97545	0.873	0.67685	0.20174
512 x 512	1	0.9759	0.95692	0.90022	0.32964
1024 x 1024	1	0.93309	0.84375	0.95891	0.64606
2048 x 2048	1	1.31136	1.39833	1.21402	1.35777
4096 x 4096	1	0.86032	0.72597	0.64203	0.94337

Αν και παρατηρείται γενικά μια αστάθεια στις μετρήσεις, απο τα αποτελέσματα δεν μπορούμε να συμπεράνουμε ότι το πρόγραμμα είναι ισχυρά επεκτάσιμο, καθώς η αποδοτικόττα ανα μέγεθος διαφέρει αρκετά.
Ωστόσο το πρόγραμμα φαίνεται να είναι ασθενώς επεκτάσιμο καθώς στην πλειοψηφία των αποτελεσμάτων, η αποδοτικότητα διατηρείται λίγο πολύ σταθερή όταν διπλασιάζουμε το μέγεθος του προβλήματος μαζί με το μέγεθος του προβλήματος.

5 Ενσωμάτωση *OPENMP*

Παραλληλοποίηση του υπολογισμού των έσωτερικών' αλλά και των έξωτερικών' κελιών του πίνακα με χρήση:
-pragma omp parallel for
-pragma omp parallel sections

```
...
//calculate/populate 'inner' cells
#pragma omp parallel for collapse(2)
for (i=row_start + 1; i<= row_end - 1; i++)
{
  for (j= col_start + 1; j <= col_end - 1; j++)
  {
    //for each cell/organism

    //for each cell/organism
    //see if there is a change
    //populate functions applies the game's rules
    //and returns 0 if a change occurs
    if (populate(array1, array2, N, M, i, j) == 0)
        no_change = 0;
  }
}

//wait for recvs
MPI_Waitall(8, recv_request[communication_type], statuses);

//calculate/populate 'outer' cells

//up/down row
#pragma omp parallel for
for (j= col_start; j <= col_end; j++) {
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      if (populate(array1, array2, N, M, row_start, j) == 0)
      no_change = 0;
    }
    #pragma omp section
    {
      if (populate(array1, array2, N, M, row_end, j) == 0)
      no_change = 0;
    }
  }
}
```

```
//left/right col
#pragma omp parallel for
for (i=row_start; i<= row_end; i++)
{
  #pragma omp parallel sections
  {
    #pragma omp section
    {
    if (populate(array1, array2, N, M, i, col_start) == 0)
    no_change = 0;
        }
    #pragma omp section
    {
      if (populate(array1, array2, N, M, i, col_end) == 0)
      no_change = 0;
    }
  }
}

//corners
#pragma omp parallel sections
{
  #pragma omp section
  {
    if (populate(array1, array2, N, M, row_start, col_start) == 0)
    no_change = 0;
  }
  #pragma omp section
  {
    if (populate(array1, array2, N, M, row_start, col_end) == 0)
    no_change = 0;
  }
  #pragma omp section
  {
    if (populate(array1, array2, N, M, row_end, col_start) == 0)
    no_change = 0;
  }
  #pragma omp section
  {
    if (populate(array1, array2, N, M, row_end, col_end) == 0)
    no_change = 0;
  }
}
...
```

5.1 Χρόνοι

Για τις εκτελέσεις τόσο του *OpenMP* χρησιμοποιήθηκαν 7 διαθέσιμοι υπολογιστές της σχολής (ο 26 δεν μπορούσε να τρέξει *OpenMP*). Ο αριθμός επαναλήψεων που χρησιμοποιήθηκε για τις μετρήσεις ήταν 500 και έλεγχος για αλλαγή πλέγματος γινόταν κάθε 20 επαναλήψεις. Τα αρχεία εισόδου που χρησιμοποιήθηκαν είναι τα ίδια με αυτά που χρησιμοποιήθηκαν στο *MPI*.

Table 7: MPI-OPENMP timings with reduce

WITH REDUCE	**processor**	**1**	**2**	**4**	**8**
threads & size					
1 & 128x128		*0.948789*	*8.124007*	*8.339107*	*5.132730*
2 & 128x128		*1.034578*	*7.732336*	*13.061874*	*4.413136*
4 & 128x128		*1.107532*	*4.622765*	*8.573740*	*13.688686*
8 & 128x128		*0.903353*	*8.323989*	*6.484052*	*4.860042*
1 & 256x256		*4.879794*	*4.361695*	*5.446872*	*2.410867*
2 & 256x256		*6.344141*	*4.042780*	*3.994405*	*8.841831*
4 & 256x256		*5.155313*	*5.102790*	*5.679992*	*0.978468*
8 & 256x256		*4.296948*	*4.459619*	*1.260573*	*2.191813*
1 & 512x512		*23.000883*	*7.850280*	*8.979746*	*4.313928*
2 & 512x512		*20.332571*	*8.006090*	*8.321848*	*8.537243*
4 & 512x512		*16.545502*	*16.836772*	*7.980021*	*2.511396*
8 & 512x512		*14.460122*	*7.960438*	*18.614641*	*2.067840*
1 & 1024x1024		*69.003258*	*58.671678*	*63.152333*	*25.216681*
2 & 1024x1024		*58.097262*	*60.780002*	*65.197339*	*15.536028*
4 & 1024x1024		*68.193377*	*62.447546*	*33.323935*	*15.698366*
8 & 1024x1024		*57.863566*	*56.895953*	*35.863364*	*4.15202*
1 & 2048x2048		*749.217094*	*280.694984*	*77.869525*	*34.637301*
2 & 2048x2048		*464.614370*	*170.796409*	*68.570618*	*34.478967*
4 & 2048x2048		*325.549141*	*176.109375*	*69.335642*	*34.625216*
8 & 2048x2048		*502.272161*	*178.353288*	*73.258658*	*34.365944*

Table 8: MPI-OPENMP timings without reduce

size \processors	***1***	***2***	***4***	***8***
128 x 128 1 thread	0.910214	7.825845	9.667980	4.195511
128 x 128 2 threads	0.964804	0.882477	8.359953	4.128068
128 x 128 4 threads	1.175129	11.959990	11.896956	2.220348
128 x 128 8 threads	1.160117	5.472357	3.083855	4.860042
256 x 256 1 thread	5.386563	2.220249	3.377850	8.861572
256 x 256 2 threads	4.843557	4.625820	5.119800	10.785162
256 x 256 4 threads	5.384033	4.627199	5.979208	8.230306
256 x 256 8 threads	5.508845	5.131799	2.488058	2.008650
512 x 512 1 thread	22.744841	8.108089	8.943500	4.127979
512 x 512 2 threads	16.127130	9.917318	8.211972	2.064010
512 x 512 4 threads	14.670824	7.851447	8.975328	6.546400
512 x 512 8 threads	14.735914	15.512435	7.763762	2.067840
1024 x 1024 1 thread	75.635061	61.248265	39.181009	15.197785
1024 x 1024 2 threads	80.154459	43.430616	100.343958	15.884065
1024 x 1024 4 threads	61.737023	71.346417	31.032796	111.227995
1024 x 1024 8 threads	58.040129	46.106482	37.496978	40.260046
2048 x 2048 1 thread	598.514129	256.462453	69.054985	34.821293
2048 x 2048 2 threads	352.763843	137.800705	69.549843	34.432549
2048 x 2048 4 threads	318.183089	163.128221	75.876063	34.395857
2048 x 2048 8 threads	433.596394	256.225831	69.573756	34.353958

Και πάλι οι μετρήσεις είναι αρκετά ασταθείς αλλά τα συμπεράσματα που μπορούμε να βγάλουμε είναι ότι για μικρά μεγέθη (μικρότερα του 512) το *overhead* λόγω των τηρεαδς είναι πολύ μεγαλύτερο απο τον χρόνο που κερδίζεται στους υπολογισμούς με αποτέλεσμα να είναι οι χρόνοι χειρότεροι. Ωστόσο όσο μεγαλώνουμε το μέγεθος, τόσο μεγαλύτερη επιτάγχυνση έχουμε, αλλά χωρίς ακόμα να ξεπεράσουμε τις επιταγχύνσεις του MPI (τουλάχιστον με λιγότερα απο 8 *threads*).
Με 8 επεξεργαστές οι χρόνοι για μέγεθος 2048 είναι λίγο καλύτεροι απο το MPI αλλά δεν μπορούμε να εξάγουμε κάποιο σίγουρο συμπέρασμα χωρίς να επεκταθούμε σε μεγαλύτερα μεγέθη. Επίσης παρατηρούμε ότι για μεγάλο μέγεθος (2048), ακόμα και να αυξήσουμε τα *threads*, οι διαφορές στους χρόνους είναι ελάχιστες, πράγμα που λογικά οφείλεται στο ότι όσο γρήγορα και να τελειώσουν τα *threads*, θα πρέπει να περιμένουν να ολοκληρωθεί η επικοινωνία για να προχωρήσουν στον υπολογισμό των έξωτερικών΄ κελιών.

6 Αυτόνομο πρόγραμμα $CUDA$

Δείτε το αρχείο *gol_cuda.cu*, δεν μπορέσαμε να τρέξουμε μετρήσεις κυρίως λόγω τεχνικών προβλημάτων.