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

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

25 Φεβρουαρίου 2018

Περιεχόμενα

1 Εισαγωγή

Έχουν υλοποιηθεί όλα τα ζητούμενα της εργασίας (MPI, $MPI-OPENMP$, $CUDA$). Επίσης έχουν ικανοποιηθεί οι απαιτήσεις που ανέλυσε ο διδάσκοντας κατά την παρουσίαση της εργασίας. Συγκεκριμένα:
- Τα δεδομένα διαμοιράζονται σε *blocks*.
- Αποφυγή των *duplicates*.
- Κάθε διεργασία διατηρεί μονάχα τον πίνακα/*block* που της αναλογεί με 2 παραπάνω γραμμές και 2 παραπάνω στήλες προκειμένου να αποθηκέυσει τις γραμμές/στήλες των γειτόνων.
- Ο πίνακας της κάθε διεργασίας δεσμεύεται σε συνεχόμενη μνήμη και στην συνέχεια εμφανίζεται ως δισδιάστατος με αναθέσεις δεικτών, όπως εξηγείται παρακάτω.
- $Non-blocking$ 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.

- Έλεγχος για μη αλλαγή του πίνακα με *Allreduce*. Κάθε διεργασία στέλνει ένα 0/1 που υποδηλώνει αν ΔΕΝ άλλαξε κάτι στο *block* της. Έτσι γίνεται ένα ομαδικό άθροισμα που αν φτάσει στο πλήθος των διεργασιών, οδηγεί σε πρόωρο τερματισμό.
Επιπλέον έχει προστεθεί η εξής λειτουργικότητα στοχεύοντας κυρίως στην καλύτερη διεξαγωγή μετρήσεων και δοκιμών γενικότερα:
- Δυνατότητα διαβάσματος *input* αρχείο το οποίο γίνεται *scatter* στις διάφορες διεργασίες απο τον *master* με δικιά μας συνάρτηση. - Δυνοτότητα αυτόματης παραγωγής *input* αρχείων μέσω ̈ προγραμμάτων στον κατάλογο *test_generators* ή μέσω του ίδιου του προγράμματος αν δεν δωθεί *input file*.

```
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 ευρείας παραμετροποίησης του εκτελέσιμου MPI ή MPI-$OPENMP$ ή $Cuda$ προγράμματος.
Το *make* παράγει και τα 3 ζητούμενα εκτελέσιμα καθώς και το σειριακό $Game\ Of\ Life$ (κώδικας C). Αν δεν υπάρχει εγκατεστημένος ο *nvicc* η μεταγλώττιση του *Cuda* αποτυγχάνει αλλά τα προηγούμενα 2 μεταγλωττίζονται κανονικά.

```
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>

./gol_cuda -f <filename> -l <N> -c <M> -n <max_loops> -r <reduceRate> -b <blocks> -t <
```

όπου *reduce_rate* το πλήθος των επαναλήψεων μετά το οποίο ελέγχεται για αλλαγή πλέγματος. Αν δωθεί μη θετική τιμή (π.χ. 0), τότε δεν γίνεται ποτέ έλεγχος για αλλαγή πλέγματος. Πάρθηκαν μετρήσεις και με και χωρίς έλεγχο για αλλαγή.

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 //for MPI-OPENMP only
```

- Για τα *-l -c*, αν δοθούν θα πρέπει να δοθούν και τα δύο αλλίως παίρνουν $default$ τιμές.
- Αν δεν δωθεί αρχείο, παράγεται ένα παιχνίδι (η παραγωγή δεν συμμετέχει στην χρονομέτρηση) αυτόματα με βάση το μέγεθος, με περίπου του μισούς οργανισμούς ζωντανούς, και αν το $define$ $SAVE\_GENERATED$ είναι 1, αποθηκεύεται (για μελλοντική αναφορά) στον κατάλογο *generated_tests* με όνομα σχετικό με το μέγεθος και το *timestamp* της δημιουργίας.
Στην συγκεκριμένη συνάρτηση η *master* διεργασία παράγει τυχαία συντεταγμένες ζωντανών οργανισμών, τις οποίες είτε τις κρατάει για τον δικό του τοπικό (μικρό) πίνακα/*block*, είτε τις στέλνει στην κατάλληλη διεργασία για να κάνει το ίδιο.
- Αποστολή των κομματιών του *input* 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. Η λειτουργία είναι όμοια με την αυτόματη παραγωγή που αναφέρθηκε παραπάνω.
- Αποστολή τελικών *blocks* απο όλες τις διεργασίες στον *master* με μια δικιά μας συνάρτηση τύπου *gather*, με σκοπό την εκτύπωση του τελικού πίνακα μόνο αν το $flag$ $PRINT\_FINAL$ είναι 1 (παρακάτω). Δεν συμμετέχει στην χρονομέτρηση της εκτέλεσης. Για τον σκοπό αυτό ορίσαμε το βλοςκ ως *derived data type* ώστε να στέλνουν οι διεργασίες ολόκληρο το *block* τους στον *master* και όχι απλά τις συντεταγμένες μία μία:

```
//Define block_array data type
//In case we want to gather all arrays to master
MPI_Datatype derived_type_block_array;

MPI_Type_vector(rows_per_block,
cols_per_block,
cols_per_block + 2,
MPI_SHORT,
&derived_type_block_array);

MPI_Type_commit(&derived_type_block_array);
```

- Το παραπάνω *gather* μπορεί να γίνεται και στο τέλος κάθε *loop* με σκοπό την εκτύπωση των βημάτων του παιχνιδιού αν το $flag$ $PRINT\_STEPS$ είναι 1. Χρησιμοποιήθηκε για να βεβαιωθούμε ότι δουλεύει σωστά το πρόγραμμα. Διευκρινίζεται ότι για τον σκοπό της εκτύπωσης δεσμεύεται (μόνο αν είναι 1 τα αντίστοιχα $flags$) ολόκληρος ο πίνακας και κάθε διεργασία στέλνει στον *master* το δικό της *block*.
- 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 (διαμοιρασμός 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 διαμοιρασμό των δεδομένων σε *blocks* (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 τον αριθμό των διεργασιών. Αξίζει να αναφερθεί ότι υπολογίζονται έτσι τα *line_div* και *col_div*, που μπορουμε να σπάσουμε σε *blocks* διάφορους αριθμούς διεργασιών, και όχι απλά τετράγωνα. Προφανώς όταν οι διεργασίες είναι τετράγωνο:
$line\_div = col\_div = \sqrt{processes}$

```
...
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;
...
row_start = 1; // row 0 is the neigbours row
row_end = rows_per_block;
col_start = 1; // col 0 is the neigbours col
col_end = cols_per_block;
```

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.

```
...
//get this process's coordinates in the virtual topology
ret = MPI_Cart_coords(virtual_comm, my_rank, ndimansions, my_coords);

//get the ranks of this process's (8) neighbours
neighbour_coords[0] = my_coords[0] - 1; //go up
neighbour_coords[1] = my_coords[1];
ret = MPI_Cart_rank(virtual_comm, neighbour_coords, &rank_u);

neighbour_coords[0] = my_coords[0] + 1; //go down
neighbour_coords[1] = my_coords[1];
ret = MPI_Cart_rank(virtual_comm, neighbour_coords, &rank_d);
...
```

3 Σχεδιασμός και υλοποίηση MPI κωδικα

Με σκοπό την καλύτερη οργάνωση του κώδικα και τον διαχωρισμό του κώδικα που αφορά το παιχνίδι, και του κώδικα που αφορά το MPI (διαμοιρασμός δεδομένων, επικοινωνία, τοπολογία, *derived data types* κλπ) το ίδιο παιχνίδι αντιμετωπίζεται σε ξεχωριστά αρχεία.
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 κλπ). Οργανώνεται σε 3 τμήματα:

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

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_Start)
x8 MPI_IRecv (MPI_Start)
Calculate 'inner' cells
Wait for IRecvs to complete
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 (it will return immediatly if the above is used)
```

```
//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 όσο και του $MPI-OPENMP$ χρησιμοποιήθηκαν 29 διαθέσιμοι υπολογιστές της σχολής. Ο αριθμός επαναλήψεων που χρησιμοποιήθηκε για τις μετρήσεις ήταν 500 και έλεγχος για αλλαγή πλέγματος αν γινόταν, ήταν κάθε 20 επαναλήψεις. Τα αρχεία εισόδου που χρησιμοποιήθηκαν παράχθηκαν απο τον δικό μας *test_genarator*. Σε μετρήσεις που διατηρούσαμε το μέγεθος ίδιο (5040 επι 5040) χρησιμοποιήθηκε το ίδιο ακριβώς αρχείο σε όλες τις μετρήσεις για να έχουμε καλύτερο μέτρο σύγκρισης.

Table 1: 100 processors - Variable size with and without reduce

Size/Reduce	Without Reduce	Reduce rate 20
630x630	19.536960s	19.963401s
1260x1260	20.375856s	19.916006s
2520x2520	20.895833s	20.383997s
5040x5040	22.997950s	23.295994s
10080x10080	78.647988s	79.663951s

Table 2: Size 5040

Reduce	Without Reduce	Reduce rate 20
1 Processor	393.994528s	579.663185s
4 Processors	201.764549s	200.678950s
9 Processors	92.947688s	93.228045s
16 Processors	71.343499s	71.159935s
25 Processors	49.446506s	49.999688s
36 Processors	37.187947s	35.947987s
49 Processors	30.898386s	30.247963s
64 Processors	29.966213s	29.883991s
81 Processors	24.310714s	24.315209s
100 Processors	23.488344s	22.783994s

Table 3: Size 5040 Speedups

Reduce	Without Reduce	Reduce rate 20
1 Processor	1	1
4 Processors	1.95274	2.8885
9 Processors	4.23889	6.2177
16 Processors	5.52249	8.14592
25 Processors	7.9681	11.59334
36 Processors	10.59468	16.12505
49 Processors	12.7513	19.16371
64 Processors	13.14795	19.39711
81 Processors	16.20662	23.83952
100 Processors	16.77405	25.44168

- Διπλασιάζοντας το μέγεθος κάθε φορά απο $630x630$ μέχρι $10080x10080$ φαίνεται αρχικά ο χρόνος να παραμένει σταθερός στα 20 δευτερόλεπτα παρά τον διπλασιασμό του μεγέθους, πράγμα που σημαίνει ότι η επικάλυψη επικοινωνίας δουλεύει και οι πράξεις που γίνονται είναι λίγο-πολύ ίδιες. Ωστόσο στο τελευταίο μέγεθος ($10080x10080$), απο κεί που για $5040x5040$ αυξήθηκε ελάχιστα και έγινε 23 δευτερόλεπτα, ο χρόνος αυξήθηκε απότομα στα 78 δευτερόλεπτα. Το μεγάλο μέγεθος του πίνακα οδηγεί σε καθυστερήσεις λόγω της επικοινωνίας. Δηλαδή μετά απο ένα μέγεθος για σταθερό αριθμό διεργασιών, η επικάλυψη επικοινωνίας δεν είναι αρκετή. Ακόμα και μετά τον υπολογισμόυ των εσωτερικών σημείων ενός *block*, η επικοινωνία έχει αρκετά μεγάλο όγκο ώστε να μην έχει ολοκληρωθεί ακόμα. Συμπεραίνουμε λοιπόν ότι ακόμα και με την επικάλυψη που κάνουμε με *non-blocking* συναρτήσεις, το οερηεαδ της επικοινωνίας δεν μπορεί παρά να καθυστερεί την εκτέλεση όσο αυξάνουμε το μέγεθος διατηρώντας σταθερό αριθμό διεργασιών.

- Η επιτάγχυνση είναι ίκανοποιητική αρχικά αυξάνωντας διεργασίες, αλλά όσο περισσότερο αυξάνονται οι διεργασίες τόσο λιγότερο αυξάνεται η επιτάγχυνση που επιτυγχάνεται. Χαρακτηριστικό είναι ότι στις 81 διεργασίες η επιτάγχυνση είναι 16.2 ενώ με 100 διεργασίες 16.7 (χρόνοι 24.3 και 23.4 αντίστοιχα). Δηλαδή η διαφορά είναι ελάχιστη και φαίνεται ήδη ότι το πρόβλημα δεν είναι ισχυρά επεκτάσιμο. Όσο αυξάνονται οι διεργασίες, το κάθε *block* γίνεται όλο και μικρότερο, με αποτέλεσμα τελικά να τελειώνει η επεξεργασία των εσωτερικων σημείων πιο γρήγορα απ ότι η επικοινωνία με τους γείτονες. Επίσης είναι προφανές ότι περισσότερες διεργασίες σημαίνει περισσότερη επικοινωνία και άρα μεγαλύτερη συμφόρηση του δικτύου. Τελικά οι υπολογισμοί στο πρόβλημα μας είναι τόσο απλοί που όταν η επικοινωνία καθυστερεί θα γίνονται πιο γρήγορα και επομένως ο τελικός χρόνος θα εξαρτάται κατα κύριο λόγο απο την καθυστέρηση της επικοινωνίας.

- Η αποτελεσματικότητα μειώνεται όσο αυξάνουμε τις διεργασίες, γεγονός που επιβεβαιώνει τα παραπάνω. Το πρόβλημα δεν είναι ισχυρά επεκτάσιμο και όσο αυξάνεται ο αριθμός των διεργασιών τόσο λιγότερο εκμεταλευόμαστε την επεξεργαστική ισχύ των υπολογιστών αφού είναι αναγκασμένοι να περιμένουν να ολοκληρωθεί η επικοινωνία πρωτού συνεχίσουν με τα εξωτερικά σημεία του *block*.

-Όσον αφορά τον έλεγχο για αλλαγή κάθε n επαναλήψεις, τα συμπεράσματα είναι ίδια όσον αφορά την επεκτασιμότητα. Από τους χρόνους φαίνεται ότι το *Allreduce* καθυστερεί την εκτέλεση αλλά όχι σε τεράστιο βαθμό. Βέβαια κάτι τέτοιο δεν μας φαίνεται λογικό και μάλλον οφείλεται στην αστάθεια των μετρήσεων. Θέλαμε να ξανακάνουμε μετρήσεις χωρίς τον έλεγχο αλλά τις τελευταίες ημέρες λειτουργούσαν οι 14 απο τους 30 υπολογιστές.

Table 4: Size 5040 Efficiency

Reduce	Without Reduce	Reduce rate 20
1 Processor	1	1
4 Processors	0.48817	0.72212
9 Processors	0.47098	0.69084
16 Processors	0.34515	0.50911
25 Processors	0.31871	0.46373
36 Processors	0.2943	0.4479
49 Processors	0.26022	0.39108
64 Processors	0.20543	0.30307
81 Processors	0.20007	0.29431
100 Processors	0.16774	0.25441

Χρησιμοποίησαμε και το εργαλείο *mpip* που το ενεργοποιήσαμε με βάση την ανακοίνωση μόνο για το κεντρικό *loop* του παιχνιδιού. Παραθέτουμε τα αποτελέσματα για 64 διεργασίες, μέγεθος 5040 (με και χωρίς έλεγχο).
Όπως είναι αναμενόμενο το μεγαλύτερο ποσοστό απο τον χρόνο εκτέλεσης καταλαμβάνει η *Waitall* (75% χωρίς *reduce* και 41% με) καθώς οι διεργασίες είναι πολλές. Αξιοσημείωτο είναι ότι η αναμονή καταλαμβάνει μεγαλύτερο χρόνο απο το *Allreduce* όσο αυξάνεται ο αριθμός των διεργασιών.

mpiP - **Οι πλήρεις έξοδοι βρίσκονται στον φάκελο** *mpip*

Without Reduce

//16 processes

@— Aggregate Time (top twenty, descending, milliseconds) —

Call	Site	Time	App%	MPI%	COV
Waitall	4	2.43e+06	75.05	99.93	0.24
Startall	3	1.58e+03	0.05	0.06	0.69
Waitall	1	132	0.00	0.01	4.42
Startall	2	92.2	0.00	0.00	1.08

// 4 Processes

@— Aggregate Time (top twenty, descending, milliseconds) —

Call	Site	Time	App%	MPI%	COV
Waitall	4	1.1e+05	37.17	99.96	0.67
Startall	3	46.3	0.02	0.04	0.17
Startall	2	1.29	0.00	0.00	0.29
Waitall	1	1.12	0.00	0.00	0.32

With Reduce (Every 20 generations)

@— Aggregate Time (top twenty, descending, milliseconds) —

Call	Site	Time	App%	MPI%	COV
Waitall	5	4.87e+05	41.37	77.93	0.32
Allreduce	3	1.36e+05	11.59	21.83	0.16
Startall	4	1.31e+03	0.11	0.21	0.51
Startall	2	94.8	0.01	0.02	1.45
Waitall	1	43.5	0.00	0.01	1.50

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

Ο αριθμός επαναλήψεων που χρησιμοποιήθηκε για τις μετρήσεις ήταν 500 και έλεγχος για αλλαγή πλέγματος γινόταν κάθε 20 επαναλήψεις.
Οι μετρήσεις έχουν μια κάποια αστάθεια γιατί προσπαθήσαμε να λειτουργούμε με όσα μηχανήματα είχαμε διαθέσιμα εκείνη την στιγμή (σε ορισμένες περιπτώσεις). Προσπαθήσαμε να πάρουμε τις καλύτερες κάνοντας πολλαπλές εκτελέσεις. Στο αρχείο *machines* είχαμε αναθέσει μια διεργασία ανα υπολογιστή όπως προτάθηκε απο τον διδάσκοντα.
Γενικά το $OPENMP$ σε συνδυασμό με το MPI έχει ανταγωνιστικούς χρόνους στην περίπτωση των 8 *threads* αλλά δεν βελτιώνει και τόσο τους χρόνους για πάνω απο 9 διεργασίες. Για 1, 4 και 9 διεργασίες η χρήση των *threads* είναι ιδιαίτερα ευεργετική για τους χρόνους, καθώς με 16 *threads* οι χρόνοι πέφτουν στο μισό και λίγο παρακάτω (σε σχέση με 1 *thread*). Βέβαια για 2 και 4 *threads* η διαφορά φαίνεται να είναι ελάχιστη ενώ για απο τις 25 διεργασίες και μετά τα οι χρόνοι είναι τελικά χειρότεροι. Συγκεκριμένα στις 16 διεργασίες με 16 *threads* κερδίσαμε μονάχα 3 δευτερόλεπτα ενώ στις 25 διεργασίες ο χρόνος ανέβηκε απο τα 30 δευτερόλεπτα στα 42.

Συνεπώς βλέπουμε ότι με το $OPENMP$ δεν κερδίζουμε και πολλά η επεκτασιμότητα δεν είναι καλή, καθώς για μεγάλο αριθμό διεργασιών τα *threads* μάλλον επιβαρύνουν την εκτέλεση.
Ωστόσο σε περίπτωση που έχουμε λίγους υπολογιστές στην διάθεση μας, έχουμε καλύτερους χρόνους.
Για το *reduce* και πάλι οι χρόνοι είναι λίγο μεγαλύτεροι όταν έχουμε τον έλεγχο, αλλά φαίνεται να επηρεάζει λιγότερο την εκτέλεση σε σύγκριση με το απλό MPI.

Table 5: Variable size - 16 processes 16 threads

Size \Reduce	Without Reduce	Reduce Rate 20
2520x2520	23.104s	24.787s
5040x5040	65.705s	67.558s
10080x10080	202.175s	202.730s
20160x20160	631.254s	657.247s

Table 6: MPI-OPENMP 5040x5040 timings without reduce

Threads \Processes	1	4	9	16	25	36
1	376.751s	94.892s	42.987s	40.262s	32.533s	21.305s
2	275.510s	49.104s	22.217s	46.486s	50.253s	61.976s
4	493.096s	81.060s	60.697s	98.111s	101.432s	105.730s
8	237.091s	40.121s	18.824s	36.839s	28.054s	21.323s
16	142.060s	40.278s	19.400s	38.683s	44.215s	66.888s

Table 7: MPI-OPENMP 5040x5040 speedups without reduce

Threads \Processes	1	4	9	16	25	36
1	1	3.9703	8.7643	9.35748	11.58058	17.68369
2	1	5.61075	12.40086	5.92673	5.48245	4.44543
4	1	6.0831	8.12389	5.02591	4.86134	4.66373
8	1	5.9094	12.59514	6.43587	8.45123	11.11903
16	1	3.527	7.32268	3.67242	3.21294	2.12386

Table 8: MPI-OPENMP 5040x5040 efficiency without reduce

Threads \Processes	1	4	9	16	25	36
1	1	0.99257	0.9738	0.58484	0.46321	0.55261
2	1	1.40268	1.37787	0.3704	0.21928	0.13892
4	1	1.52077	0.90265	0.31412	0.19444	0.14574
8	1	1.47734	1.39946	0.40224	0.33804	0.34746
16	1	0.88174	0.81363	0.22952	0.12851	0.06636

Table 9: MPI-OPENMP 5040x5040 timings with reduce

Threads \Processes	1	4	9	16	25	36
1	362.931s	94.793s	43.053s	40.496s	30.641s	19.075s
2	341.629s	49.092s	22.157s	42.379s	46.153s	62.127s
4	361.116s	81.205s	60.896s	97.635s	100.054s	104.318s
8	216.193s	39.858s	18.728s	36.026s	27.541s	21.714s
16	152.294s	40.778s	19.422s	37.855s	42.165s	69.260s

Table 10: MPI-OPENMP 5040x5040 speedups with reduce

Threads \Processes	1	4	9	16	25	36
1	1	3.82867	8.42987	8.96214	11.84462	19.02654
2	1	6.95895	15.41856	8.06128	7.4021	5.49889
4	1	4.44696	5.93004	3.69864	3.6092	3.46169
8	1	5.42409	11.54382	6.00102	7.84985	9.95638
16	1	3.73471	7.84131	4.02309	3.61186	2.19887

Table 11: MPI-OPENMP 5040x5040 efficiency with reduce

Threads \Processes	1	4	9	16	25	36
1	1	0.95717	0.93665	0.56012	0.47377	0.59457
2	1	1.73973	1.71317	0.50383	0.29608	0.17183
4	1	1.11174	0.65889	0.23116	0.14436	0.10817
8	1	1.35602	1.28264	0.37506	0.31398	0.31113
16	1	0.93367	0.87125	0.25143	0.14447	0.06871

Και εδώ χρησιμοποιήσαμε το εργαλείο *mpiP* για 16 διεργασίες και 8 *threads*, όπου βλέπουμε ότι πλέον δεν καταλαμβάνει η *Waitall* τόσο μεγάλο ποσοστό του προγράμματος όσο με το απλό *MPI* το οποίο είχε 75% χωρίς έλεγχο για αλλαγή πλέγματος.

Without reduce

Call	Site	Time	App%	MPI%	COV
Waitall	4	6.22e+05	58.82	99.91	0.49
Startall	2	491	0.05	0.08	0.47
Startall	1	31.8	0.00	0.01	0.21
Waitall	3	26.7	0.00	0.00	0.37

With reduce

@— Aggregate Time (top twenty, descending, milliseconds) —

Call	Site	Time	App%	MPI%	COV
Waitall	5	1.89e+06	62.73	90.75	0.34
Allreduce	2	1.92e+05	6.37	9.22	0.35
Startall	3	655	0.02	0.03	0.95
Waitall	4	61.5	0.00	0.00	2.58
Startall	1	26.8	0.00	0.00	0.20

6 Αυτόνομο πρόγραμμα $CUDA$

Έχουμε υλοποιήσει την εργασία και με την χρήση $Cuda$ ($gol\_cuda.cu$). Χρησιμοποιήσαμε κάρτα γραφικών $GTX9604Gb$.

Στην αρχή όπως και στις άλλες υλοποιήσεις μετατρέπουμε τον πίνακα του παιχνιδιού σε μονοδιάστατο πίνακα ώστε να είναι συνεχόμενη η μνήμη.
Έπειτα δεσμεύουμε την απαιτούμενη μνήμη μέσα στη κάρτα γραφικών $cudaMalloc()$.
Σε κάθε $for\ loop$ πριν αρχίσει παράλληλη εκτέλεση μεταφέρουμε τα δεδομένα που χρειαζομαστε στο $device$ και μετα την εκτέλεση τα ξαναφέρνουμε πίσω στην κυρία μνήμη $cudaMemcpy()$.
Για το κύριο κομμάτι που είναι η παράλληλη εκτέλεση το κάθε $thread$ γνωρίζοντας την θέση του μέσα στο $grid$ των $threads$ του $Cuda$ και έτσι παίρνει ενα κελί επεξεργάζεται, ελέγχει τους γείτονες του και αποφασίζει αν τελικά στον επόμενο γυρώ θα είναι ζωντανό ή όχι. Χρησιμοποιούμε δύο πίνακες όπου στον δεύτερο αποθηκεύεται η επόμενη καταστασή ώστε να μην υπάρχουν προβλήματα συγχρονισμού ή λάθος υπολογιμού λόγω αλλαγής της κατάστασης των γειτόνων.

Παρακάτω είναι οι μετρήσεις που έχουν γίνει. Μέγιστος αριθμός $thread$ που έχουμε χρησιμοποιήσει είναι 64 * 256 = 16.384 $threads$.

Table 12: Reduce Rate = 1

Size	**8 b *128 t**	**8 b *256 t**	**16 b *128 t**	**16 b *256 t**	***32 b *128 t***	**32 b *256 t**
512x512	0.357s	0.294s	0.299s	0.278s	0.280s	0.281s
1024x1024	1.284s	0.999s	0.992s	0.885s	0.869s	0.854s
2048x2048	4.603s	3.557s	3.400s	3.132s	3.056s	3.034s
4096x4096	17.927s	13.956s	14.021s	12.561s	12.037s	11.602s
8192x8192	68.040s	54.916s	55.593s	48.822s	47.901s	45.158s

Table 13: No reduce rate

Size	**8 b *128 t**	**8 b *256 t**	**16 b *128 t**	**16 b *256 t**	***32 b *128 t***	**32 b *256 t**
512x512	0.317s	0.260s	0.261s	0.245s	0.244s	0.239s
1024x1024	1.143s	0.889s	0.885s	0.787s	0.770s	0.767s
2048x2048	4.247s	3.247s	3.239s	2.770s	2.755s	2.683s
4096x4096	16.326s	12.870s	12.899s	11.322s	11.053s	10.572s
8192x8192	64.835s	50.964s	50.073s	44.987s	43.409s	41.558s

Μπορούμε να παρατηρήσουμε οτι 8 b * 256 t == 16 b * 128 t και 16 b * 256 t == 32 b *128 t. Η εκτέλεση γίνεται με ίδιο συνολικό αριθμό $threads$ αλλά σε διαφορετική κατανομή, δηλαδή διαφορετικό αριθμο $block$ και $thread$ ανα $block$. Κάναμε αυτές τις μετρήσεις για να δούμε μήπως υπάρχει διαφορά στον χρόνο εκτέλεσης αναλογα την κατανομή των νημάτων και των $blocks$ στο $grid$ του $Cuda$. Απο τις μετρήσεις βλέπουμε ότι δεν υπάρχει κάποια εμφανές και ουσιαστική διαφόρα όποτε συμπεράναμε οτι για την λύση αυτη δεν παίζει ρόλο ή κατανομή των νημάτων.

Πιστεύουμε ότι για το προβλημά μας το $Cuda$ έχει πολύ καλά αποτελέσματα. Οι πράξεις που γίνονται σε κάθε $loop$ είναι απλές και παρόμοιες με αποτέλεσμα η GPU να τις εκτελεί πολύ γρήγορα. Μέχρι τα 1024 $threads$ το MPI με 100 διεργασίες είχε καλύτερο χρόνο ($23.48sec$), αλλά στην συνέχεια αυξάνοντας τον αριθμό των $threads$ το $Cuda$ κατέβηκε στα $15.97sec$. Ο έλεγχος για αλλαγή εδώ είναι προφανώς λιγότερο χρονοβόρος.
Επίσης:
- Όσο αυξάνουμε τα $threads$ η επιτάγχυνση αυξάνεται και λιγότερο αφού για παράδειγμα με 8192 είχαμε επιτάγχυνση 24.08 ενώ με 16384 24.66. Οπότε ούτε εδώ έχουμε πολύ καλή επεκτασιμότητα
- Για μεγάλους πίνακες το $Cuda$ φαίνεται να είναι περισσότερο αποτελεσματικό απ΄ οτι το MPI. Παρατηρούμε ότι σε κάθε περίπτωση, διπλασιάζοντας το μέγεθος, ο χρόνος γίνεται περίπου τετραπλάσιος. Στο MPI για μικρότερα μεγέθη οι χρόνοι αυξανόντουσαν ελάχιστα ακόμα και διπλασιάζοντας το μέγεθος, αλλά για το μεγαλύτερο μέγεθος που δοκιμάσαμε παρατηρήσαμε μεγάλη μεταβολή στον χρόνο. Πιστεύουμε ότι το $overhead$ της επικοινωνίας θα χειροτερεύουν τους χρόνους του MPI για πολύ μεγάλα μεγέθη. Στο $Cuda$ η μεταβολή του χρόνου όταν διπλασιάζεται το μέγεθος φαίνεται να είναι πιο σταθερή, ενώ αυξάνοντας τα $threads$ οι χρόνοι έχουν ικανοποιητική βελτίωση.

Table 14: Cuda 5040x5040 size With reduce rate

Blocks*Threads	***Time Elapsed***
8x8	183.738s
9x9	149.621s
10x10	125.142s
16x16	58.966s
32x32	23.452s
8x64	38.001s
8x128	24.548s
16x128	19.150s
32x128	16.502s
8x256	19.071s
16x256	16.851s
32x256	16.360s
64x256	15.974s

Table 15: Speedups - Cuda 5040x5040 size With reduce rate

Blocks*Threads	***Speedup***
8x8	2.14433
9x9	2.63329
10x10	3.14838
16x16	6.68172
32x32	16.80005
8x64	10.36801
8x128	16.04996
16x128	20.57413
32x128	23.87556
8x256	20.65935
16x256	23.38107
32x256	24.0828
64x256	24.66473

Table 16: Cuda 5040x5040 size No reduce rate

Blocks*Threads	***Time Elapsed***
8x8	178.881s
9x9	149.094s
10x10	123.454s
16x16	58.350s
32x32	25.526s
8x64	37.992s
8x128	25.501s
16x128	19.895s
32x128	17.032s
8x256	20.327s
16x256	17.352s
32x256	16.745s
64x256	16.136s

Table 17: Speedups - Cuda 5040x5040 size No reduce rate

Blocks*Threads	***Speedup***
8x8	3.2405
9x9	3.88791
10x10	4.69537
16x16	9.93425
32x32	22.70874
8x64	15.2575
8x128	22.73099
16x128	29.13611
32x128	34.03377
8x256	28.5169
16x256	33.40611
32x256	34.6171
64x256	35.92358