Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών

Τμήμα Πληροφορικής και Τηλεπικοινωνιών

ΘΠ04 - Παράλληλα Συστήματα

Εργασία - Game of Life

Μέλη:

- Βασίλειος Πουλόπουλος - 1115201600141
- Κωνσταντίνος Χατζόπουλος - 1115201300202

Εισαγωγή

Για την υλοποίηση της εργασίας, δημιουργήσαμε αρχεία εισόδου για τις μετρήσεις. Για κάθε τμήμα της εργασίας έχουμε ξεχωριστό φάκελο στον οποίο υπάρχει ο κώδικας, οι μετρήσεις, τα αποτελέσματα του profiling, τα input files και κάποια bash scripts που υλοποιήσαμε για να αυτοματοποιήσουμε τις παρακάτω διαδικασίες:

- μεταγλώττιση
- εκτέλεση στην argo
- συλλογή μετρήσεων
- υπολογισμό speedup
- υπολογισμό efficiency.

Τα scripts αρχικά μεταγλωττίζουν το πρόγραμμα με τις αντίστοιχες παραμέτρους κάθε φορά (μέγεθος προβλήματος **N**, αριθμό **διεργασιών** / **nodes** / **cpus** / **threads** / **gpus**) και στη συνέχεια για να το τρέξουν το τοποθετούν στην ουρά με την εντολή **qsub**.

Για όσο βρίσκεται στην ουρά και εκτελείται ή περιμένει να εκτελεστεί, το script ελέγχει σταδιακά με την εντολή **qstat** αν ολοκληρώθηκε το **job** και αμέσως μετά, με την εντολή append του bash (**>>**) δημιουργεί (αν δεν υπάρχουν ήδη) αρχεία που περιέχουν μόνο τους χρόνους με όνομα **times.txt**. Τέλος, για τα script που αφορούν το **MPI** ή **MPI + OpenMp**, το script διαβάζει τα αντίστοιχα αρχεία χρόνων και δημιουργεί τα **speedup.txt** και **efficiency.txt**.

Επίσης, υλοποιήσαμε script που δημιουργεί **επίπεδα αρχεία εισόδου** (όλος ο δισδιάστατος πίνακας χαρακτήρων σε μια γραμμή) τα οποία είναι ίδια με τα **παραγόμενα αρχεία κάθε βήματος**. Επιπλέον, για να ελέγξουμε αν κάθε βήμα έχει υπολογιστεί σωστά υλοποιήσαμε script που μετατρέπει τα επίπεδα αρχεία **εισόδου** / **εξόδου** σε δισδιάστατη αναπαράσταση μαύρων ή άσπρων κουτιών (μαύρο κουτί αν η τιμή είναι 0 και άσπρο αν είναι 1).

Στο **MPI** και **MPI + OpenMp** χρησιμοποιούμε command line arguments για να πάρουμε τα εξής:

- path για το input file
- path για το output file
- αριθμό γραμμών
- αριθμό στηλών

MPI

Η επιλογή των κόμβων και των διεργασιών έγινε με το σκεπτικό να γεμίζει ο κάθε κόμβος πριν χρησιμοποιήσουμε κάποιο καινούργιο. Αυτό το κάνουμε για να έχουμε όσο το δυνατό λιγότερη επικοινωνία ανάμεσα στους κόμβους ώστε να αποφεύγονται άσκοπες καθυστερήσεις. Δοκιμάζοντας και τις 2 προσεγγίσεις (**πληρότητα κόμβων**, **διασπορά διεργασιών σε κόμβους**) καταλήξαμε στην πρώτη διότι παρατηρήσαμε ότι ήταν πιο γρήγορη, πράγμα το οποίο είναι λογικό αφού στην αρχιτεκτονική κατανεμημένης μνήμης ο κάθε κόμβος δεν έχει πρόσβαση στη μνήμη άλλου κόμβου. Αυτό έχει ως αποτέλεσμα όσο τους αυξάνουμε να αυξάνεται και η επικοινωνία μεταξύ των κόμβων μέσω μηνυμάτων και συνεπώς να αυξάνεται η καθυστέρηση.

Σχεδιασμός

Ο δισδιάστατος πίνακας εισόδου θεωρείται **περιοδικός**, π.χ. αν είμαστε στην τελευταία γραμμή του δισδιάστατου και θέλουμε να πάμε στην ακριβώς από κάτω του, τότε αυτή θα είναι η πρώτη γραμμή κ.ο.κ.

Για την καλύτερη επίτευξη επικοινωνίας μεταξύ των κόμβων, δημιουργούμε μια τοπολογία όπου περιέχει ένα δισδιάστατο πλέγμα από blocks / διεργασίες διαστάσεων **sqrt(# of processes) * sqrt(# of processes)**, γι' αυτό το λόγο σχεδιάσαμε το πρόγραμμα να λειτουργεί μόνο με αριθμούς διεργασιών που είναι τέλεια τετράγωνα. Με αυτόν τον τρόπο, κάθε διεργασία είναι στοιχισμένη σε κάποια θέση στο δισδιάστατο πλέγμα και επεξεργάζεται ένα μόνο **block** / **υποπίνακα** του συνολικού προβλήματος.

Σχεδιασμός MPI κώδικα

Για να μπορέσει να τρέξει το MPI πρόγραμμα, πρέπει να γίνει αρχικοποίηση. Για να γινει αυτό καλούμε τη συνάρτηση **MPI_Init()**. Αμέσως μετά δημιουργούμε την τοπολογία μέσω της συνάρτησης **setupGrid()** όπου αρχικοποιεί τη δομή **gridInfo** με τα δεδομένα της αντίστοιχης διεργασίας.

struct gridInfo:

```
typedef struct gridInfo {
  MPI_Comm gridComm;     // communicator for entire grid
  Neighbors neighbors;   // neighbor processes
  int processes;         // total number of processes
  int gridRank;          // rank of current process in gridComm
  int gridDims[2];       // grid dimensions
  int gridCoords[2];     // grid coordinates
  int blockDims[2];      // block dimensions
  int localBlockDims[2]; // local block dimensions
  int stepGlobalChanges;
int stepLocalChanges; } GridInfo;
```

Τα δεδομένα που αρχικοποιεί η συνάρτηση **setupGrid()** σε κάθε διεργασία είναι:

- ο communicator που ανήκει στο πλέγμα, οι γειτονικές διεργασίες,
- ο συνολικός αριθμός διεργασιών
- η τάξη της διεργασίας στο πλέγμα,
- τις διαστάσεις του πλέγματος
- τις συντεταγμένες της στο πλέγμα καθώς και
- τις διαστάσεις του συνολικού πίνακα αλλά και του
- τοπικού (δικού της) πίνακα.

Για να δημιουργηθεί η τοπολογία καλούμε τη συνάρτηση **MPI_Cart_create()** και για να λάβουμε τις συντεταγμένες κάθε διεργασίας τη **MPI_Cart_coords()**.

Για τον υπολογισμό κάθε βήματος χρειάζεται να ξέρουμε την προηγούμενη κατάσταση του παιχνιδιού για να υπολογίσουμε την τρέχουσα. Για το λόγο αυτό δεσμεύουμε δυναμικά με χρήση της συνάρτησης **malloc()** 2 δισδιάστατους πίνακες χαρακτήρων (**old**, **current**) διαστάσεων `grid.localBlockDims[0] + 2, grid.localBlockDims[1] + 2` δηλαδή τις διαστάσεις που αντιστοιχούν στον κάθε υποπίνακα του συνολικού προβλήματος *αυξημένο κατά 22 διαστάσεις** για να συμπεριληφθούν και οι γειτονικές τιμές.

MPI Datatypes

Επειδή υπήρχε η ανάγκη να στέλνουμε και να λαμβάνουμε γραμμές και στήλες από τους τοπικούς αυτούς πίνακες σε γειτονικές διεργασίες, δημιουργήσαμε **datatypes** με τις κατάλληλες συναρτήσεις (**MPI_Type_vector()**, **MPI_Type_commit()**) γιατί χρειαζόμασταν ένα τρόπο με τον οποίο ή κάθε διεργασία θα αποθηκεύει στον δικό της πίνακα τις τιμές των πινάκων των γειτονικών **block** για να μπορει να υπολογίσει τις ακριανές τιμές του πίνακά της.

Το MPI θεωρεί πως για τα datatypes τα δεδομένα του buffer αποστολής / λήψης πρέπει να είναι σε συνεχόμενες θέσεις μνήμης, και τα offset για γραμμές και στήλες να είναι σταθερα, οπότε χρειαζόταν οι δισδιάστατοι πίνακες **old** και **current** να είναι σε συνεχόμενες θέσεις μνήμης. Γι αυτό το λόγο υλοποιήσαμε τη συνάρτηση **allocate2DArray()** με τέτοιο τρόπο ώστε να δημιουργεί ένα δισδιάστατο πίνακα σε συνεχόμενες θέσεις μνήμης.

Είσοδος δεδομένων αρχικής κατάστασης παιχνιδιού

Αν υπάρχει αρχείο εισόδου χρησιμοποιούμε ένα **subarray** datatype που δημιουργήσαμε για την ανάγνωση των σωστών **περιοχών** / **τμημάτων** του αρχείου εισόδου από το αντίστοιχο process μέσω της **MPI_File_set_view()** και το διαβάζουμε μέσω της **MPI_File_read()**. Στην περίπτωση που δεν υπάρχει αρχείο εισόδου, δημιουργείται ένας τυχαίος πίνακας με ολόκληρη την αρχική κατάσταση του παιχνιδιού μέσω της συνάρτησης **initialize_block()** και με τη βοήθεια της συνάρτησης **scatter2DArray()** (*κώδικας από το βιβλίο MPI ΘΕΩΡΙΑ ΚΑΙ ΕΦΑΡΜΟΓΕΣ*), ο πίνακας διασπείρεται σε όλες τις διεργασίες του δισδιάστατου πλέγματος. Με αυτόν τον τρόπο η κάθε διεργασία έχει στον τοπικό της υποπίνακα **old** το αντίστοιχο υποπρόβλημα που καλείται να υπολογίσει.

Επικοινωνία μεταξύ των διεργασιών του πλέγματος (Κεντρική επανάληψη)

Για την επίτευξη της αποστολής των **8 μηνυμάτων** (ένα για κάθε γείτονα) στις γειτονικές διεργασίες καθώς και τη **λήψη άλλων 8**, χωρίς την επιβάρυνση της κεντρικής επανάληψης με επιπλέον υπολογισμούς (offsets, κτλ), απαιτείται η αρχικοποίηση τους μέσω της συνάρτησης **MPI_Send_init()** ώστε να δημιουργηθούν **16 requests** τα οποία θα είναι έτοιμα προς εκκίνηση όταν αυτό χρειαστεί με τη χρήση της συνάρτησης **MPI_Startall()**. Στη συνέχεια γίνονται οι υπολογισμοί των εσωτερικών κυττάρων και κατόπιν, καλείται η **MPI_Waitall()** ώστε πριν υπολογιστούν και τα εξωτερικά κύτταρα να έχουν σταλθεί όλα τα γειτονικά σε κάθε block για να γίνουν σωστά οι υπολογισμοί.

Υπολογισμοί

Η συνάρτηση **calculate()** που έχουμε υλοποιήσει για τους υπολογισμούς είναι **inline** για να μην υπάρχουν καθυστερήσεις, επιπλέον, η συνάρτηση αυτή δέχεται ως όρισμα ένα δείκτη στην ακέραια μεταβλητή **changes** όπου αν υπάρξει αλλαγή στο συγκεκριμένο κύτταρο που εξετάζεται εκείνη τη στιγμή, η μεταβλητή αυτή αυξάνεται κατά 1.

Swap

Στο τέλος της επανάληψης, το αποτέλεσμα που υπολογίστηκε από τον πίνακα **'old'**, αποθηκεύεται στον πίνακα **'current'**. Έτσι, στην επόμενη επανάληψη θα πρέπει η παλιά κατάσταση του παιχνιδιού να αντικατασταθεί με την current του προηγούμενου βήματος. Για το λόγο αυτό, πρέπει να γίνει **swap** του old με τον current.

Μετρήσεις χρόνων εκτέλεσης (Χωρίς reduce)

	1 Process	4 Processes	16 Processes	64 Processes
320x320	0.745	**0.216**	0.202	0.292
640x640	3.354	0.770	**0.313**	0.310
1280x1280	13.268	3.383	0.851	**0.391**
2560x2560	53.610	13.459	3.466	**0.917**
5120x5120	213.743	53.943	18.438	**8.652**
10240x10240	-	215.087	54.659	**13.750**
20480x20480	-	-	217.871	**57.988**
40960x40960	-	-	-	**218.213**

Παρατηρούμε ότι όσο αυξάνεται το μέγεθος του προβλήματος, μεγαλώνοντας τον αριθμό των διεργασιών ότι στα πράσινα υπάρχει καλύτερη κλιμάκωση, συνεχίζοντας όμως να μεγαλώνουμε τον αριθμό, βλέπουμε ότι δεν συμφέρει (κίτρινα) διότι γίνεται σπατάλη πόρων για την επίτευξη του ίδιου περίπου χρόνου με τα πράσινα.

Επιπλέον, για μεγέθη μεγαλύτερα από **5120x5120** δεν έφτασε το όριο των 10 λεπτών και σταμάτησε η εκτέλεση.

Υπολογισμός speedup

	1 Process	4 Processes	16 Processes	64 Processes
320x320	1.0	**3.449**	3.681	2.542
640x640	1.0	4.357	**10.705**	10.815
1280x1280	1.0	3.921	15.600	**33.949**
2560x2560	1.0	3.983	15.468	**58.478**
5120x5120	1.0	3.962	11.593	**24.704**

Όπως αναφέραμε και παραπάνω, δεν υπάρχει λόγος να αυξήσουμε τις διεργασίες για τις περιπτώσεις που είναι με κίτρινο.

Υπολογισμός efficiency

	1 Process	4 Processes	16 Processes	64 Processes
320x320	1.0	**1.724**	0.920	0.318
640x640	1.0	2.178	**2.676**	1.352

	1 Process	4 Processes	16 Processes	64 Processes
1280x1280	1.0	1.961	3.900	**4.244**
2560x2560	1.0	1.992	3.867	**7.310**
5120x5120	1.0	1.981	2.898	**3.088**

Με τη βοήθεια του τύπου υπολογισμού του **efficiency** (**speedup** / **processes**), επιβεβαιώσαμε τον παραπάνω συλλογισμό μας.

Μετρήσεις χρόνων εκτέλεσης (Με reduce)

	1 Process	4 Processes	16 Processes	64 Processes
320x320	0.747	**0.216**	0.217	0.330
640x640	3.364	0.774	**0.333**	0.348
1280x1280	13.315	3.384	**0.879**	0.443
2560x2560	53.737	13.470	3.502	**0.977**
5120x5120	214.299	54.121	13.764	**3.594**

Υπολογισμός speedup

	1 Process	4 Processes	16 Processes	64 Processes
320x320	1.0	**1.729**	0.859	0.283
640x640	1.0	2.172	**2.525**	1.208
1280x1280	1.0	1.968	**3.786**	3.757
2560x2560	1.0	1.995	3.836	**6.874**
5120x5120	1.0	1.980	3.892	**7.454**

Υπολογισμός efficiency

	1 Process	4 Processes	16 Processes	64 Processes
320x320	1.0	**1.724**	0.920	0.318
640x640	1.0	2.178	**2.676**	1.352
1280x1280	1.0	1.961	3.900	**4.244**
2560x2560	1.0	1.992	3.867	**7.310**
5120x5120	1.0	1.981	2.898	**3.088**

Στο πρόγραμμά μας δεν παρατηρούμε κάποια σημαντική διαφορά στους χρόνους. Παρόλα αυτά είναι ένα παραπάνω κόστος που γίνεται σε κάθε επανάληψη. Όσο αυξάνονται οι διεργασίες, αυξάνονται και τα μηνύματα

λόγο επικοινωνίας στο server οπότε αυξάνεται και η καθυστέρηση.

MPI+OPENMP

Για την ανάπτυξη του **OpenMp** κώδικα χρησιμοποιήσαμε τον ίδιο κώδικα με το **MPI** που περιγράψαμε παραπάνω και προσθέσαμε εντολές που δημιουργούν και διαχειρίζονται νήματα.

Σχεδιασμός

Ο ρόλος των νημάτων είναι να παραλληλοποιηθεί όσο το δυνατόν περισσότερο το κάθε βήμα της κεντρικής επανάληψης. Για το λόγο αυτό, στο αρχικό νήμα (**master**) πριν ξεκινήσει η κεντρική επανάληψη, δημιουργούνται νήματα όπου το κάθε ένα από αυτά θα τρέξει όλα τα βήματα με σκοπό να παραλληλοποιηθούν οι εσωτερικές επαναλήψεις που κάνουν τα **calculations** και τα βήματα τους να μοιραστούν σε κάθε thread.

Σχεδιασμός MPI+OpenMp κώδικα

Κεντρική επανάληψη

Όταν θέλουμε να εκτελεστεί ένα τμήμα κώδικα της κεντρικής επανάληψης μόνο από το πρώτο νήμα που θα φτάσει σε εκείνο το σημείο, χρησιμοποιούμε το `#pragma omp single`, ενώ όταν θέλουμε να γίνει μόνο μία φορά, από το master νήμα, χρησιμοποιούμε το `#pragma omp master`. Αυτό χρειάζεται στις περιπτώσεις που καλούνται συναρτήσεις επικοινωνίας του MPI (**MPI_Startall()**, **MPI_Waitall()**, **MPI_Allreduce()**)

Συγχρονισμός νημάτων

Μετά από την υλοποίηση των υπολογισμών, χρειάζεται να γίνει συγχρονισμός νημάτων ώστε πριν γίνει το **swap** μεταξύ των δύο πινάκων (**old**, **current**) να είμαστε σίγουροι πως έχουν γίνει όλοι οι υπολογισμοί σε κάθε block. Το συγχρονισμό αυτό, επιτυγχάνουμε μέσω της `#pragma omp barrier`

Διασπορά κεντρικής επανάληψης βημάτων σε νήματα

Επιθυμούμε να παραλληλοποιήσουμε τις επαναλήψεις με τους υπολογισμούς μέσω των thread και γι αυτό χρησιμοποιούμε την εντολή `#pragma omp for`, ώστε κάποια βήματα των for να γίνουν παράλληλα.

Για τον υπολογισμό των εσωτερικών στοιχείων, που έχουμε **δύο εμφωλευμένες for**, συμπληρώσαμε στο `#pragma omp for` το `collapse(2)` ώστε να εφαρμοστεί η παραλληλοποίηση για το διπλό for. Δοκιμάσαμε να βάλουμε `#pragma omp for` μόνο στην εσωτερική έπανάληψη, χωρίς **collapse** και σε μεγάλο αριθμό κόμβων και threads παρατηρήσαμε πως οι χρόνοι ήταν πολύ μεγάλοι.

Έλεγχος τερματισμού

Για να υπάρχει έλεγχος τερματισμού, χρησιμοποιούμε τη μεταβλητή `stepLocalChanges`. Για να υπολογιστεί σωστά ο συνολικός αριθμός των τοπικών αλλαγών, κάνουμε `reduction(+:stepLocalChanges)` σε κάθε `#pragma omp for`.

Μετρήσεις χρόνων εκτέλεσης (collapse)

Καταλήξαμε στους συνδυασμούς **2 Threads** / **16 Processes** και **4 Threads** / **16 Processes**, καθώς παρατηρήσαμε πως ήταν οι πιο αποδοτικοί σε όλα τα μεγέθη με την επιλογή **8 κόμβων** και **2 διεργασιών** / **κόμβο**. Αυτό οφείλεται στο γεγονός ότι με τους συγκεκριμένους συνδυασμούς, εξαλείφεται εντελώς το **scheduling** επειδή κάθε νήμα θα εκτελεστεί σε διαφορετικό **πυρήνα**.

	2 Threads / 16 Process	4 Threads / 16 Processes
320x320	0.241	0.240
640x640	0.410	0.363
1280x1280	1.060	0.669
2560x2560	3.591	2.000
5120x5120	13.730	7.464

Υπολογισμός speedup

	2 Threads / 16 Process	4 Threads / 16 Processes
320x320	6,880	6,922
640x640	16,074	18,162
1280x1280	24,735	39,188
2560x2560	29,348	52,709
5120x5120	30,671	56,413

Υπολογισμός efficiency

Για τον υπολογισμό του efficiency χρησιμοποιήσαμε τον τύπο **Speedup / (Processes * Threads)**.

	2 Threads / 16 Process	4 Threads / 16 Processes
320x320	0,215	0.108
640x640	0.502	0.284
1280x1280	0,773	0.612
2560x2560	0,917	0,824
5120x5120	0,958	0,881

Δοκιμάσαμε να γεμίζουμε τους κόμβους, όπως στο **MPI** και λόγω του ότι τα **threads** δεν χωρούσαν σε κάθε κόμβο και γινόταν **scheduling**, άρα δεν γίνονταν παράλληλα, οι χρόνοι ήταν πολύ μεγάλοι. Αντίθετα, από την υλοποίηση με απλό MPI, στην περίπτωση του **υβριδικού MPI + openMP** δεν γεμίζαμε τους κόμβους που χρησιμοποιούσαμε, αλλά αντιθέτως χρησιμοποιούσαμε πολλούς κόμβους, για να μπορούν να χρησιμοποιηθούν πολλά **threads** σε κάθε κόμβο χωρίς να γίνεται **scheduling**. Έτσι τα threads εκτελούνταν παράλληλα.

Έξοδος script

Η έξοδος του script που τρέξαμε, είχε με το παρακάτω format: Όπου:

- **Select**, ο αριθμός των κόμβων,
- **Procs** ο αριθμός των διεργασιών που χρησιμοποίησε ο κάθε κόμβος,
- **Processes** τα συνολικά processes που τρέχει το πρόγραμμα και

- **Threads**, ο αριθμός των **νημάτων** ανά **διεργασία**.

Έξοδος για μέγεθος προβλήματος **320x320** (με collapse):

```
Select 1 Procs 1 Processes 1 Threads 1: 1.661166
Select 2 Procs 2 Processes 4 Threads 1: 0.574466
Select 8 Procs 2 Processes 16 Threads 1: 0.310549
Select 10 Procs 8 Processes 64 Threads 1: 0.342512

Select 1 Procs 1 Processes 1 Threads 2: 1.693688
Select 2 Procs 2 Processes 4 Threads 2: 0.372130
Select 8 Procs 2 Processes 16 Threads 2: 0.241463
Select 10 Procs 8 Processes 64 Threads 2: 10.313023

Select 1 Procs 1 Processes 1 Threads 4: 1.737526
Select 2 Procs 2 Processes 4 Threads 4: 0.267918
Select 8 Procs 2 Processes 16 Threads 4: 0.239978
Select 10 Procs 8 Processes 64 Threads 4: 30.845762

Select 1 Procs 1 Processes 1 Threads 8: 1.817256
Select 2 Procs 2 Processes 4 Threads 8: 0.608338
Select 8 Procs 2 Processes 16 Threads 8: 0.533826
Select 10 Procs 8 Processes 64 Threads 8: 3.604214

Select 1 Procs 1 Processes 1 Threads 16: 1.987536
Select 2 Procs 2 Processes 4 Threads 16: 0.819975
Select 8 Procs 2 Processes 16 Threads 16: 0.657694
Select 10 Procs 8 Processes 64 Threads 16: 4.968571
```

Έξοδος για μέγεθος προβλήματος **5120x5120** (Με collapse):

```
Select 1 Procs 1 Processes 1 Threads 1: 421.063685
Select 2 Procs 2 Processes 4 Threads 1: 106.055394
Select 8 Procs 2 Processes 16 Threads 1: 26.498416
Select 10 Procs 8 Processes 64 Threads 1: 6.825866

Select 1 Procs 1 Processes 1 Threads 2: 421.727274
Select 2 Procs 2 Processes 4 Threads 2: 54.166751
Select 8 Procs 2 Processes 16 Threads 2: 13.728413
Select 10 Procs 8 Processes 64 Threads 2: 17.354089

Select 1 Procs 1 Processes 1 Threads 4: 422.450984
Select 2 Procs 2 Processes 4 Threads 4: 28.378362
Select 8 Procs 2 Processes 16 Threads 4: 7.463992
Select 10 Procs 8 Processes 64 Threads 4: 43.181102

Select 1 Procs 1 Processes 1 Threads 8: (Το σταματήσαμε επειδή θα χρειαζόταν περίπου 422")
Select 2 Procs 2 Processes 4 Threads 8: 41.686177
Select 8 Procs 2 Processes 16 Threads 8: 10.623288
Select 10 Procs 8 Processes 64 Threads 8: 10.164924

Select 1 Procs 1 Processes 1 Threads 16: (Το σταματήσαμε επειδή θα χρειαζόταν  περίπου 422")
Select 2 Procs 2 Processes 4 Threads 16: 56.886217
Select 8 Procs 2 Processes 16 Threads 16: 15.242415
Select 10 Procs 8 Processes 64 Threads 16: 14.786688
```

Τα υπόλοιπα output files υπάρχουν στο times_omp.zip το οποίο βρίσκεται στο φάκελο mpi+openmp. Έχουμε κάνει μετρήσεις και με collapse και χωρίς.

Παραγόμενα αρχεία (MPI, MPI+OpenMp)

Παρατηρήσαμε υπερβολική καθυστέρηση όταν γράφαμε κάθε **generation** σε αρχείο. Αυτό συνέβαινε διότι τα αρχεία αποθηκεύονται στον φάκελο **mpi/generations/row** ο οποίος βρίσκεται στο **front end** με αποτέλεσμα κάθε διεργασία από κάθε **κόμβο** να επικοινωνεί με το front end, πράγμα το οποίο λόγω των μηνυμάτων έχει καθυστερήσεις.

CUDA

Για την υλοποίηση με **CUDA**, αρχικά έπρεπε να σκεφτούμε πως θα χωρίσουμε σε κάθε thread το πρόβλημα μιας και η **GPU** έχει χιλιάδες νήματα, έτσι αποφασίσαμε να χρησιμοποιήσουμε ένα δισδιάστατο πλέγμα από blocks δύο διαστάσεων κάνοντας την παραδοχή πως κάθε κύτταρο θα το επεξεργάζεται και ένα διαφορετικό νήμα.

Σχεδιασμός host κώδικα

Κατά την εκκίνηση του προγράμματος είναι απαραίτητο να γίνουν κάποιες αρχικοποιήσεις στον host κώδικα ώστε να γίνει κλήση της kernel συνάρτησης (**device**). Πιο συγκεκριμένα, γίνεται αρχικοποίηση των μεταβλητών **m**, **n** τύπου **dim3** όπου αποθηκεύονται οι διαστάσεις των blocks (m) καθώς και οι διαστάσεις του πλέγματος από μπλοκ (**n**). Μετά την παραπάνω αρχικοποίηση, πραγματοποιείται έλεγχος **assert** ώστε να εξασφαλιστεί πως το πρόγραμμα θα εκτελεστεί με τις σωστές διαστάσεις πλέγματος και νημάτων ανα μπλοκ.

Όταν περάσει ο έλεγχος, δεσμεύεται στη **host memory** δυναμικά ένας δισδιάστατος πίνακας χαρακτήρων (σε συνεχόμενες θέσεις μνήμης) ο οποίος θα αρχικοποιηθεί αργότερα από το αρχείο εισόδου με τα δεδομένα του πρώτου βήματος με τελικό σκοπό να αντιγραφεί στην **device memory** (device_old). Για να την επίτευξη της αντιγραφής αυτής, είναι απαραίτητο να γίνει δέσμευση `N * N * sizeof(char)` bytes στην device memory μέσω της συνάρτησης **cudaMalloc()** και αμέσως μετά να κληθεί η **cudaMemcpy()**.

Στην κεντρική επανάληψη, καλείται η συνάρτηση **kernel** που αναλαμβάνει να υπολογίσει ολόκληρο το βήμα χωρίζοντας το πρόβλημα σε blocks και νήματα όπου κάθε νήμα είναι ένα κύτταρο.

Κατά την ολοκλήρωση υπολογισμού του τρέχοντος βήματος μέσα στην κεντρική επανάληψη, είναι απαραίτητο να αρχικοποιηθεί ο πίνακας **device_old** με ‘0’ καθώς και να γίνει αντικατάσταση του **device_old** με τα δεδομένα του **device_current** πίνακα για την αρχικοποίηση του επόμενου βήματος.

Σχεδιασμός device κώδικα (kernel)

Αρχικά θεωρούμε πως ο **τετραγωνικός πίνακας μεγέθους N x N** αντιγράφεται στη μνήμη της κάρτας γραφικών και πως κάθε μπλοκ από νήματα θα έχει στη διάθεσή του και θα επεξεργάζεται μόνο ένα μέρος αυτού. Δεδομένου ότι κάθε block είναι δυνατό να έχει τη δική του κοινή ιδιωτική μνήμη, αρχικά δεσμεύουμε αυτή τη μνήμη με την εντολή `__shared__ char local[M + 2][M + 2];` δημιουργώντας έτσι για κάθε ένα από αυτά έναν πίνακα διαστάσεων `(M+2) * (M+2)` ώστε να αποθηκευτούν περιμετρικά και οι τιμές των κυττάρων από τα γειτονικά μπλοκ (πάνω, κάτω, αριστερά, δεξιά, διαγώνιες). Έπειτα, αντιγράφουμε το αντίστοιχο **2d** κομμάτι που αντιστοιχεί στην τοπική μνήμη του κάθε μπλοκ καθώς και τα γειτονικά του. Η αντιγραφή αυτή είναι πολύ σημαντικό να γίνει διότι την ώρα που θα γίνονται οι υπολογισμοί στο εκάστοτε **block**, δε θα χρειάζεται να γίνεται προσπέλαση της **Global Memory** (η οποία απαιτεί 400-800 κύκλους) αλλά θα έχει άμεση πρόσβαση στις τιμές των γειτονικών κυττάρων μέσω της τοπικής του μνήμης κάνοντάς το έτσι πιο γρήγορο.

Υπολογισμοί

Για τους υπολογισμούς που εκτελεί κάθε νήμα, όταν χρειαστεί να διαβάσει την παλιά κατάσταση ενός γειτονικού κυττάρου για να μπορέσει να ενημερώσει την καινούργια, δεν είναι απαραίτητο να αποκτήσει πρόσβαση στον πίνακα **device_old** που είναι στην **global memory**. Αντιθέτως, το μόνο που κάνει είναι να διαβάσει την αντίστοιχη τιμή που θέλει από την τοπική κοινή μνήμη (**local**) του block που ανήκει πολύ πιο γρήγορα από τι θα το έκανε διαβάζοντας την ίδια τιμή από τον πίνακα **device_old** (**global memory**). Παρόλα αυτά, για να γραφτεί το αποτέλεσμα του τρέχοντος βηματος, θα πρέπει να γίνει write στον πίνακα **device_current** (**global memory**), κάτι το οποίο προσθέτει καθυστέρηση αλλά είναι αναπόφευκτο.

Μετρήσεις χρόνων εκτέλεσης

	1 * 1 thread / block	2 * 2 threads / block	4 * 4 threads / block	8 * 8 threads / block	16 * 16 threads / block	32 * 32 threads / block	64 * 64 threads / block	128*128 threads / block
320x320	0.39	0.12	0.07	**0.06**	0.07	0.07	0.08	-
640x640	1.30	0.39	0.19	0.15	0.19	0.20	**0.09**	0.13
1280x1280	3.89	1.24	0.61	0.48	0.56	0.62	**0.29**	0.31
2560x2560	14.29	4.30	1.99	1.51	1.79	1.94	**0.83**	0.86
5120x5120	59.11	17.71	8.11	5.95	6.70	7.47	**3.10**	3.17

Παρατηρούμε, πως με **1 thread / block**, οι χρόνοι εκτέλεσης σε όλα τα μεγέθη του προβλήματος είναι πολύ αργοί. Αυτό οφείλεται στο γεγονός ότι κάθε thread, παίρνει τους γειτονές του από την global memory, με αποτέλεσμα να γίνεται προσπέλαση σε αυτή πολλές φορές αυξάνοντας έτσι τις καθυστερήσεις. Επίσης, όσο αυξάνεται το μέγεθος του προβλήματος, τόσο περισσότερο αυξάνεται η επιτάχυνση, όσο αυξάνουμε τα threads.

Δοκιμάσαμε να το τρέξουμε και με **256 * 256 threads / block** χωρίς επιτυχία, αφού λαμβάναμε λάθος αποτελέσματα, ενώ εμφανιζόταν το παρακάτω error:

> -ptxas error : Entry function '_Z6kernelPKcPc' uses too much shared data (0x10404 bytes, 0xc000 max)

Όπου **0x10404** (*66564 bytes*) είναι ο αριθμός των bytes (*66564 = (256 + 2) * (256 + 2) bytes*) που δεσμεύουμε στη shared memory, ενώ η μεγαλύτερη χωρητικότητα της κοινής μνήμης είναι **0xc000** (*49152 bytes*)

Συμπερασματα

Παρατηρήσαμε πως από τις τρεις υλοποιήσεις η υλοποίηση στην **GPU** με **CUDA** είναι η πιο γρήγορη με μεγάλη διαφορά, πράγμα το οποίο ήταν αναμενόμενο, αφού υπάρχει παραλληλοποίηση και σε επίπεδο **grid** (κάθε μπλοκ μέσα στο grid δουλεύει παράλληλα) και σε επίπεδο **block** (κάθε thread μέσα στο block δουλεύει παράλληλα με τα επόμενα). Επίσης, καταλήξαμε στο συμπέρασμα πως στην υλοποίηση **MPI** είναι καλύτερο να χρησιμοποιούμε όσο το δυνατόν λιγότερους κόμβους μπορούμε, ενώ στην υβριδική προσέγγιση, είναι καλύτερο να "**απλώνουμε**" τις διεργασίες σε πολλούς κόμβους.

** Όλες οι μετρήσεις πραγματοποιήθηκαν στην argo. ** Όλες οι μετρήσεις έγιναν για 1000 βήματα. *** Σε όλες τις μετρήσεις, έχουμε κάνει comment out το σημείο στο οποίο γράφεται το τρέχον generation σε αρχείο για να γίνουν πιο γρήγορα.*