Παράλληλη Υλοποίηση Game Of Life

Ομαδική εργασία για το μάθημα *Παράλληλα Συστήματα* από τους:

Ανδρέα Τσόλκα: 1115201400212

Κυριάκο Στυλιανόπουλο: 1115201400194

Οδηγίες μεταγλώττισης και εκτέλεσης:

Σημείωση: Αναμένουμε πως η μορφή του πίνακα θα είναι τετράγωνη και πως ο αριθμός των διεργασιών θα είναι τέλειο τετράγωνο. Θα μπορούσαμε να είχουμε αποφύγει τον περιορισμό αυτό αλλά θεωρήσαμε πως δεν εμπλέκεται τόσο στον σκοπό της εργασίας.

- Για την μεταγλώττιση του MPI σε συνδυασμό με το openMP:

```
$ make clean && make game
```

Αυτό θα δημιουργήσει το εκτελέσιμο **game**.
Για εκτέλεση (με ενδεικτικές τιμές):

```
$ mpiexec -f machines -n 4 game -s 28 -t 3
```

Για προβολή όλων των ορισμάτων μπορείτε να τρέξετε:

```
$ mpiexec -n 1 game --help
```

- Για την μεταγλώττιση μόνο του MPI (χωρίς το openMP):

```
$ make clean && make mpi
```

Αυτό θα δημιουργήσει το εκτελέσιμο **game_mpi**.
Για εκτέλεση (με ενδεικτικές τιμές):

```
$ mpiexec -f machines -n 4 game -s 28
```

Για προβολή όλων των ορισμάτων μπορείτε να τρέξετε:

```
$ mpiexec -n 1 game --help
```

- Για την μεταγλώττιση του Cuda:

```
$ make clean && make cuda
```

Αυτό θα δημιουργήσει το εκτελέσιμο **game_cuda**.
Και για εκτέλεση (ενδεικτικά):

```
$ ./game_cuda -s 1000 --blocks 50 --threads 100
```

Για προβολή όλων των ορισμάτων μπορείτε να τρέξετε:

```
$ ./game_cuda --help
```

Υλοποίηση σε MPI + openMP:

Γενική εικόνα:

- Η υλοποίηση είναι ένα *hybrid* παράλληλο σύστημα που περιέχει τόσο *MPI* για τον διαμοιρασμό του προβλήματος σε πολλαπλά processes και την μεταξύ τους επικοινωνία, όσο και *openMP* για την δημιουργία και την εκτέλεση παράλληλων threads μέσα στο κάθε process.

- Ο διαχωρισμός του πίνακα και η επικοινωνία μεταξύ των διεργασίων έγινε όπως ζητήθηκε, δηλαδή ο πίνακας χωρίζεται σε blocks (αντί για στήλες ή σειρές), το κάθε process αναλαμβάνει ένα τέτοιο block και επικοινωνεί με τα οχτώ γειτονικά του.

- Το *openMP* χρησιμοποιείται για τον υπολογισμό των τιμών των κελιών των εσωτερικών κελιών των blocks από κάθε διεργασία και σε κάθε επανάληψη - στιγμιότυπο. Το *openMP*, δηλαδή, δεν λειτουργεί ως αυτόνομο module του προγράμματος αλλά δρα σε συνδυασμό με το MPI.

Σημεία του κώδικα που χρίζουν επεξήγησης:

- **Μονοδιάστατος πίνακας**:
  Κάναμε χρήση μονοδιάστατων πινάκων για την αναπαράσταση των υποπινάκων του πλέγματοw διότι έτσι δεν πραγματοποιούνταν περιττές αντιγραφές κατά την επικοινωνία μεταξύ των γειτόνων.

- **Η οδηγία** `#pragma omp paraller for`:
  Τοποθετήσαμε αυτήν την οδηγία μέσα στην κεντρική επανάληψη του παιχνιδιού (η οποία μετράει τις γενιές) και ακριβώς πάνω από την επανάληψη που υπολογίζει την νέα κατάσταση των εσωτερικών κελιών. Εκ πρώτης όψεως φαίνεται σπάταλο καθώς μοιάζει να δημιουργέι και να καταστρέφει σε κάθε κατάσταση threads. Όμως, όπως διαβάσαμε σε διάφορες πηγές, οι υλοποιήσεις του openMP τυπικά δεν καταστρέφουν “ελαφρά τη καρδία” τα threads και αντίθετα τα χρησιμοποιούν στην συνέχεια.

- **Παράλληλος πίνακας** `sums`:
  Αναγκαστικά έπρεπε να αποθηκεύουμε σε κάθε επανάληψη είτε την μετέπειτα κατάσταση είτε τον αριθμό των γειτόνων κάθε στοιχείου, αλλιώς δεν θα ήταν σαφώς ορισμένη η γενία λόγω της μη σύγχρονης ενημέρωσης των τιμών των κελιών.

- **Έλεγχος τερματισμού κάθε 10 επανλήψεις**:

Ελέγχουμε αν έχει υπάρξει τελική κατάσταση κάθε 10 επαναλήψεις, διότι έτσι γλιτώνουμε μεγάλο αριθμό από `MPI_AllReduce()` με συμβιβασμό ότι υπάρχει ένα μικρό ενδεχόμενο να κτυπωθεί 10 φορές η ίδια (τελική) κατάσταση του προγράμματος.

- **Vectors για επικοινωνία**:
  Χρησιμοποιώντας τύπο δεδομένων **vector**, σε συνδυασμό με τον μονοδιάστατο πίνακα, καταφέραμε να πετύχουμε ουσιαστική εξάλειψη των περιττών αντιγραφών κατά την επικοινωνία μεταξύ των processes.

- **Parsing ορισμάτων από κάθε διεργασία**:
  Αντί να διαβάζει, να αναλαμβάνει το parsing και να μοιράζεται τις τιμές τους με τις υπολοιπες διεργασίες ο master, προτιμήσαμε να αφήσουμε κάθε διεργασία να κάνει μόνη της το parsing. Αν και φαίνεται πως όλες οι διεργασίες επαναλαμβάνουν άσκοπα τον ίδιο κώδικά, εντούτους γλιτώνουμε χρόνο καθώς δεν υπάρχει το overhead της επικοινωνίας σε συνδυασμό ότι στην άλλη περίπτωση οι υπόλοιπες διεργασίες θα παρέμεναν αδρανείς μέχρι να λάβουν τις παραμέτρους.

- **Λήψη υποπινάκων για εκτύπωση από τον master**:
  Ο τρόπος με τον οποίο βάζουμε τον master να λάβει τους υποπίνακες για την εκτύπωση είναι μέσα σε ένα for loop της μορφής `for (i=1; i<numprocs; i++)` σε κάθε επανάληψη

του οποίου διαβάζουμε τα δεδομένα της i-οστής διεργασιας. Αυτό εκ πρώτης όψεως δείχνει να μην είναι πολύ πρακτικό διότι όσο περιμένουμε να λάβουμε από τον `i`-οστό, μπορεί ο `i+1` να έχει ήδη στείλει.

Ένας εναλλακτικός τρόπος θα ήταν να χρησιμοποιησουμε το `MPI_ANY_SOURCE` κατά την κλήση της `MPI_Receive` και έτσι να μετατραπεί ο βρόγχος σε `while(not_everyone_sent)` και σε κάθε επανάληψη να διαβάζουμε από τον οποιονδήποτε. Το αρνητικό αυτής της υλοποίησης είναι πως κατά την κλήση της συνάρτησης δεν ξέρουμε την ταυτότητα του αποστολέα και άρα είμαστε αναγκασμένοι να αποθηκέυσουμε τα δεδομένα σε προσωρινή μεταβλητή και να τα αντιγράψουμε στον καθολικό πίνακα όταν (στην επόμενη γραμμή) βρούμε το id του αποστολέα.

Θεωρήσαμε πως ο πρώτος τρόπος είναι πιο πρακτικός καθώς ακόμα κι όταν ο `i+j` στέλνει πριν τον `i`, τα δεδομένα του αποθηκεύονται σε κάποιον εσωτερικό buffer κι έτσι όταν θα καταλήξουμε να διαβάσουμε από τον `i+j` ο χρόνος θα είναι πολυ μικρότερος, ενώ αντίθετα ο δεύτερος τρόπος περιέχει πολλές αντιγραφές στην μνήμη (αξίζει να αναφέρουμε πως έτσι κι αλλιώς όλες οι υλοποιήσεις έχουν ως κάτω χρονικό φράγμα τον χρόνο αποστολής της πιο αργής διαδικασίας).

Επιδόσεις και μελέτη κλιμάκωσης:

Μία γενική παρατήρηση είναι ότι οι χρόνοι εκτέλεσης των παράλληλων προγραμμάτων υπερέχουν των χρόνων των σειριακών από κάποια τάξη μεγέθους προβλήματος και μετά (π.χ. μετά τα 10.000

στοιχεία). Από εκεί και έπειτα, παρατηρήθηκε ότι το πλήθος των διεργασιών θα πρέπει να αυξάνει λογαριθμικά σχετικά με το μέγεθος του προβλήματος, ούτως ώστε να προκύπτει βέλτιστος χρόνος.
Όσον αφορά στα threads, παρατηρήθηκε (με κάποιο σφάλμα) ότι συνήθως μικρός αριθμός από threads βελτιώνει την επίδοση (π.χ.3-5 threads), αλλά το αποτέλεσμα αυτό δεν ήταν σταθερό όσο αύξανε το μέγεθος του προβλήματος (παρουσιάστηκαν σχετικές και ασταθείς διακυμάνσεις στην επίδοση ανάλογα με την τάξη μεγέθους του πλήθους των threads).
Επίσης, θα θέλαμε να αναφέρουμε κάτι το οποίο περιμέναμε και από την αρχή, το ότι δηλαδή οι χρόνοι εκτέλεσης του παράλληλου προγράμματος για μικρά μεγέθη του προβλήματος (π.χ. κάτω απο 10000 στοιχεία) είναι μεγαλύτεροι από τους χρόνους εκτέλεσης του σειριακού. Όπως φαίνεται, το overhead της επικοινωνίας και των επιπρόσθετων κλήσεων των συναρτήσεων για μικρές τιμές είναι πολύ περισσότερο από την προσφορά των επιπλέον διεργασιών/νημάτων.

Στη συνέχεια σας παρουσιάζουμε σε πίνακα τις τιμές των μετρικών `Speedup` κι `Efficiency` για χαρακτηριστικές τιμές διεργασιών και μεγεθών προβλήματος.
Τα πρόγραμμά μας εκτελούταν με σταθερές (100) επαναλήψεις, 1 μόνο thread (βρήκαμε πως αυτή ήταν η πιο αποδοτική τιμή αν και άλλαζε συχνά ανάλογα με την μέρα), και με απενεργοποιημένες τις εκτυπώσεις, σε 7 διαθέσιμα μηχανήματα της σχολής. Θα θέλαμε να επισημάνουμε πως οι χρόνοι εκτέλεσης των προγραμμάτων μας (με τις ίδιες παραμέτρους κάθε φορά) διέφεραν αρκετά από εκτέλεση σε εκτέλεση, ιδιαίτερα όταν οι εκτελέσεις είχαν διαφορές ωρών ή ημερών μεταξύ τους. Οι τιμές που λάβαμε υπόψην μας αποτελούν έναν

προσεγγιστικό μέσο όρο (με περισσότερη βαρύτητα στα αποτελέσματα των τελευταίων εκτελέσεων κι ας ήταν πιο αργές λόγου του ότι έγιναν την τελευταία μέρα και άρα τα μηχανήματα ήταν περισσότερο απασχολημένα).

Number of processes

Number of cells (board dimension)	36		49		64		81	
	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency	Speedup	Efficiency
1000000 (1000x1000)	10.4	0.29	10.82	0.22	10.02	0.16	6.29	0.08
4000000 (2000x2000)	15.63	0.43	14.02	0.28	13.18	0.21	15.11	0.19
16000000 (4000x4000)	9.32	0.26	12.51	0.26	7.18	0.11	10.43	0.13
25000000 (5000x5000)	15.27	0.42	12.42	0.25	15.27	0.24	13.33	0.16

Πάνω σε αυτό θα θέλαμε να επισημάνουμε το ότι (αν εξαιρέσουμε την τελευταία στήλη με τις 81 διεργασίες) το speedup, αυξάνοντας το πλήθος των διεργασιών και κρατώντας σταθερό το μέγεθος του προβλήματος, παραμένει επίσης σταθερό (και αντίστοιχα η αποδοτικότητα πέφτει). Δεν μπορούμε ωστόσο, βάση των μετρήσεών μας, να αντλήσουμε πληροφορία για την συμπεριφορά του προγράμματος κρατώντας σταθερό τον αριθμό των διεργασιών και αυξάνοντας το μέγεθος του προβλήματος.

Υλοποίηση με την χρήση Cuda:

Γενικές παρατηρήσεις:

- Η στρατηγική της ανάπτυξής μας ήταν να πρόγραμμα και να το τροποποιήσουμε ώστε να εκμεταλλευτεί τις δυνατότητες των πολλαπλών πυρήνων και threads που υπάρχουν στις κάρτες γραφικών. Πρακτικά τροποποιήσαμε μόνο την συνάρτηση που

υπολογίζει την νέα κατάσταση και την ορίσαμε ως `__kernel__`. Σπάσαμε την αντίστοιχη δομή επανάληψης σε κατάλληλα άκρα έτσι ώστε κάθε δημιουργούμενο thread να υπολογίζει τον ίδιο αριθμό στοιχείων.

- Θα θέλαμε να αναφέρουμε πως δεν είχαμε στην διάθεσή μας κάποιο μηχάνημα με συμβατή κάρτα γραφικών NVidia και ο sevrer της σχολής είχε τεράστια αναμονή ώστε να μην εκτελεστεί ποτέ ο κώδικας που υποβάλαμε και έτσι δεν έχουμε ελέγξει την λειτουργικότητα του προγράμματος που σας παραδίδουμε.
  (Τρέξαμε μία προηγούμενη έκδοση του κώδικά μας σε μηχάνημα ενός συμφοιτητή με διαστάσεις 1000x1000, 100 επαναλήψεις, 50 blocks και 200 threads ανά μπλοκ και ο χρόνος εκτέλεσής του ήταν της τάξης του 0.0001 δευτερολέπτου, αλλά δεν ξέρουμε κατά πόσο η μία εκτέλεση είναι αρκετή για να βγάλουμε αξιόπιστα συμπεράσματα σε επίπεδο μελέτης. Ωστόσο αυτό που μπορούμε να αποφανθούμε είναι ότι ο χρόνος εκτέλεσης ήταν πολλές τάξεις μεγέθους μικρότερος από τον αντίστοιχο χρόνο για το MPI+openMP).