ΕΚΠΑ: Πληροφορικής και Τηλεπικοινωνιών
Λαζαρίδης Νίκος
M1485
Παράλληλα Συστήματα
2016-2017

Το Παιχνίδι της Ζωής

Εισαγωγή

Σε αυτή την εργασία έγινε η σχεδίαση και υλοποίηση του δημοφιλούς παιχνιδιού της ζωής (The Game of Life) που εφηύρε ο John Conway το 1970. Προκείται για ένα παιχνίδι χωρίς παίκτες, αφού η εξέλιξη του δεν απαιτεί καμιά είσοδο κατά τη διάρκεια του, αλλά αυτή καθορίζεται μονάχα από τις αρχικές συνθήκες. Το "παιχνίδι" είναι ένα κυτταρικό αυτόματο (cellular automaton). Ο σκοπός αυτής της εργασίας είναι η μελέτη τεχνικών και αλγορίθμων παράλληλου προγραμματισμού και γενικότερα η εξοικείωση με τις παράλληλες αρχιτεκτονικές υπολογιστών και έπειτα η εφαρμογή τους για την υλοποίηση του παιχνιδιού της ζωής σε περιβάλλον MPI, OpenMP & CUDA.

Πραγματοποίηθηκε υλοποίηση σειριακή και παράλληλη. Κρίθηκε αναγκαία η έναρξη με την σειριακή υλοποίηση του παιχνιδιού για την καλύτερη κατανόηση της ροής και λογικής του προγράμματος. Έπειτα εφαρμόστηκαν οι παράλληλες τεχνικές που μελετήθηκαν, με στόχο την παραλληλοποίηση του σειριακού προγράμματος. Αυτή η στρατηγική (σειριακό σε παράλληλο) βοήθησε (κατ' εμένα) στην ευκολότερη δημιουργία του τελικού παράλληλου προγράμματος.

Τα προγράμματα είναι 3, σειριακό, παράλληλο σε CPU και παράλληλο σε GPU. Το κάθε πρόγραμμα μπορεί να μεταγλωττιστεί με δύο τρόπους. Το σειριακό μπορεί να μεταγλωττιστεί σειριακά ή με OpenMP. Το CPU παράλληλο σε MPI ή MPI + OPENMP (Hybrid) και το GPU παράλληλο σε CUDA Global memory και σε CUDA Shared memory. Επομένως πρόκεται για τέσσερις στόχους του makefile αρχείου (makefile targets). Μεταγλωττίζετε με μια από τις εξής εντολές:

1. make serial
2. make serial_openmp
3. make mpi
4. make mpi_openmp
5. make cuda
6. make cuda_shared

Θεωρώ ότι η διάκριση μεταξύ τους είναι αυτονόητη.

Για την εκτέλεση των CPU προγραμμάτων απαιτούνται τα εξής ορίσματα:
... ./όνομα_προγράμματος γραμμές στήλες γενεές πυκνότητα

Συμπεριλαμβάνεται ο πηγαίος κώδικας για όλα τα προγράμματα και ένα αρχείο Makefile για την ταχύ μεταγλώττιση τους. Στον πηγαίο κώδικα βρίσκονται αρκετά λεπτομερή σχόλια. Στις παρακάτω όμως ενότητες θα επισημάνω και κάποια «καυτά» στοιχεία που χρήζουν εντονότερης προσοχής και στα οποία θεωρώ ότι πρέπει να κάνω μια εμπεριστατωμένη αναφορά.

Το σειριακό και φυσικά το CPU παράλληλο πρόγραμμα εκτελέστηκε σε 8 Η/Υ στο cluster του ΕΚΠΑ DIT (διότι 8 ήταν το μέγιστο πλήθος μηχανημάτων που βρέθηκαν λειτουργικά) για το σκοπό της λήψης των μετρήσεων κλιμάκωσης. Ο κάθε Η/Υ διαθέτει δύο πυρήνες επομένως λήφθηκαν μετρήσεις μέχρις και 16 CPUs.

Επίσης τα προγράμματα έχουν εκτελεστεί και μετρηθεί στον προσωπικό μου Η/Υ και έχουν ληφθεί και οι αντίστοιχες μετρήσεις. Πρόκεται για 8πύρηνο Η/Υ Intel x86_64 i7 3770 με hyperthreading. Ο έλεγχος του GPU προγράμματος έγινε επίσης στον προσωπικό μου Η/Υ με την κάρτα γραφικών NVIDIA GTX 660, καθώς η GPU του πανεπιστημίου δεν ήταν διαθέσιμη. Σημειώνω ότι ο Η/Υ έχει 2 λειτουργικά συστήματα Windows 8.1 & Linux 16.04. Έχει ληφθεί υπόψη στο makefile η μεταγλώττιση σε περιβάλλον Windows / Linux με επεξεργαστή x86 / x86_64 και η παραμετροποίηση της μεταγλώττισης για την κάθε περίπτωση γίνεται αυτομάτως, απλώς εκτελώντας τις παραπάνω εντολές. Δηλαδή η δυνατότητα cross platform μεταγλώττισης και εκτέλεσης επαληθεύτηκε στο προσωπικό μου σύστημα για όλα τα προγράμματα.

Επισυνάπτονται δύο αρχεία Word με τις προδιαγραφές της προσωπικής CPU και GPU για λόγους πληρότητας.

Επίσης σε Ubuntu χρησιμοποιήθηκε ο μεταγλωττιστής GCC 5.4.0 και η υλοποίηση MPICH3 του προτύπου MPI. Σε Windows ο μεταγλωττιστής της επιλογής ήταν το Visual Studio 2015 Community και η υλοποίηση MSMPI του προτύπου MPI για την CPU παράλληλη εκτέλεση. Για την μεταγλώττιση σε CUDA χρησιμοποιήθηκε το NVIDIA Toolkit έκδοση 8 και για τα λειτουργικά συστήματα.

Το Παιχνίδι της Ζωής

Το Παιχνίδι της ζωής είναι ένα κυτταρικό αυτόματο (cellular automaton). Ο όρος “κυτταρικό” (/κύτταρο) αφορά στην επέμβαση σε συγκεκριμένα στοιχεία / κύτταρα ενός πίνακα, ενώ ο όρος “αυτόματο” τον αυτόματο, ή προκαθορισμένο τρόπο που γίνεται αυτό. Τα κυτταρικά αυτόματα είναι σημαντικά εργασλεία στην επιστήμη και τη τεχνολογία, καθώς είναι χρήσιμα στη μοντελοποίηση φυσικών φαινομένων και συστημάτων. Ως παράδειγμα, χρησιμοποιούνται για να εξομοιώσουν την ανάπτυξη των μορίων με το πέρασμα του χρόνου. Στην οικολογία μοντελοποιούν ζωτικά και φυτικά είδη που ανταγωνίζονται μεταξύ τους για τους πόρους του περιβάλλοντος.

Ειδικότερα ως τεχνική προγραμματισμού στην Επιστήμη των Υπολογιστών επινοήθηκαν για την εκτέλεση διαφόρων ανεξάρτητων λειτουργιών σε ένα πλέγμα από διατεταγμένα στοιχεία (ενός πίνακα / διανύσματος), το κάθε ένα από τα οποία αποκαλείται κύτταρο. Τα κυτταρικά αυτόματα προηγήθηκαν φυσικά της δημιουργίας του παιχνιδιού της ζωής. Ο Conway ήθελε να δημιουργήσει κάτι το οποίο εξελίσσεται με ιδιαίτερα απροσδόκητο τρόπο και έπειτα από αρκούντως μεγάλο πλήθος γενεών να μπορεί να σχηματίσει “σχέδια” που είτε ταλλαντώνονται με σταθερό τρόπο (flip-flops), είτε μένουν ακίνητα, είτε “έρπουν” κατά μήκος του πλέγματος (glider ή διαστημόπλοιο) είτε γεννούν τα ίδια άλλα μοτίβα (guns) κλπ.

Για ένα μικρού μεγέθους πλέγμα οι γενιές μπορούν να σχεδιαστούν με χαρτί και μολύβι. Όσο όμως το μέγεθος του πίνακα μεγαλώνει, η πολυπλοκότητα του αυξάνεται ακόμα πιο πρήγορα, καθιστώντας απαγορευτικά δύσκολο το χειρονακτικό έργο. Οι Η/Υ μας σώσουν για άλλη μια φορά, κάνοντας αυτό το χαζό αλλά επίπονο έργο, τάχιστα. Το παιχνίδι της ζωής είναι ιδιαίτερα χρήσιμο για παραλληλοποίηση εξαιτίας των ιδιαίτερων πρακτικών προβλημάτων υλοποίησης του, η λύση των οποίων αυξάνει το επίπεδο προγραμματισμού μας και όχι μόνο.

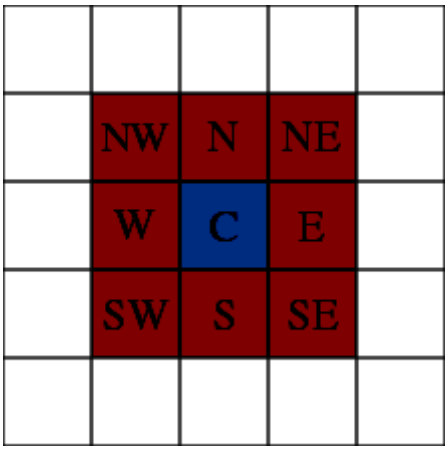
Ο John Conway ενδιαφερόταν για ένα πρόβλημα που προτάθηκε τη δεκαετία του 40 από τον μαθηματικό John Von Neumann, ο οποίος ήθελε να ανακαλύψει μια υποθετική μηχανή η οποία έχτιζε αντίγραφα του εαυτού της κάτω από συγκεκριμένες συνθήκες. Οι κανόνες όμως που έθετε ο Neumann ήταν εξαιρετικά περίπλοκοι. Ο διορατικός Conway ύστερα φυσικά από αρκετό πειραματισμό, κατάφερε να απλοποιήσει τη διαδικασία και τελικά θεμελίωσε τέσσερις απλούς

κανόνες που ακολουθεί το παιχνίδι του. Σε κάθε γενεά αξιολογούνται όλοι οι κανόνες για όλα τα κελιά του πλέγματος και αποφασίζουν σχετικά με το εάν ένας ζωντανός οργανισμός (κύτταρο) θα παραμείνει ζωντανός ή θα πεθάνει, ή εάν θα γεννηθεί ένας νέος οργανισμός από το πουθενά.

Οι κανόνες του Παιχνιδιού της Ζωής του John Conway είναι οι ακόλουθοι:

1. Εάν ένα κελί του πλέγματος είτε νεκρό, είτε ζωντανό έχει 3 ζωντανά γειτονικά κελιά τότε αυτό θα ζει στην επόμενη γενεά,
2. Εάν ένας ζωντανός οργανισμός του πλέγματος έχει 2 γειτονικούς ζωντανούς οργανισμούς τότε αυτός επιζεί στην επόμενη γενεά,
3. Εάν ένα κελί του πλέγματος έχει είτε λιγότερους από 2, είτε περισσότερους από 3 γειτονικούς ζωντανούς οργανισμούς, τότε αυτό θα είναι νεκρό στην επόμενη γενεά. Σαν από μοναξιά, ή υπερπληθυσμό αντίστοιχα.

Όταν αναφερόμαστε με τον όρο "γειτονικός" θα πρέπει να γίνουμε πιο συγκεκριμένοι. Αναφερόμαστε στη μαθηματική έννοια της γειτονιάς Moore, όπως φαίνεται παρακάτω. Γείτονες αποτελούν τα 8 γειτονικά κελιά.



Πηγή: Wikipedia

Με αυτούς τους 4 κανόνες έχει αποδειχτεί κάτι ιδιαίτερα ενδιαφέρον. Είναι δυνατό να προκύψουν ιδιαιτέρως περίτεχνα "μοτίβα" μετά από ένα αρκούντως μεγάλο αριθμό γενεών. Με άλλα λόγια, καθίσταται δυνατό να προκύψουν εξαιρετικά περίπλοκες "μορφές ζωής" με απλοικά υλικά και στοιχεία που δίδονται ως αρχικές συνθήκες.

Σειριακό πρόγραμμα – Serial

Παρακάτω περιγράφεται βήμα-βήμα σε ψευδογλώσσα ο αλγόριθμος που ακολουθήθηκε για την σειριακή υλοποίηση του παιχνιδιού της ζωής. Ένα μικρό μέρος του όλου αλγορίθμου (το βήμα 3) είναι παραλληλοποιήσιμο, αλλά αυτό είναι και το τμήμα στο οποίο γίνεται το μεγαλύτερο έργο - θυμηθείτε ότι το 90% του χρόνου εκτέλεσης ενός προγράμματος σπαταλάται προσεγγιστικά στο 10% του συνολικού κώδικα.

1. Λήψη και επεξεργασία παραμέτρων παιχνιδιού από τη γραμμή εντολών
2. Κατανομή πίνακα στη μνήμη και γέμισμα του με τυχαία boolean τιμή
3. Έναρξη παιχνιδιού. Εκτέλεση του παρακάτω βρόχου για τον προκαθορισμένο αριθμό γενεών που ορίστηκαν

a. 3.1 Για κάθε θέση του πίνακα A[y][x]:
    i. Μέτρα τους γείτονες του στοιχείου στη θέση A[y][x]
    ii. Εφαρμογή των 4 απλών κανόνων του παιχνιδιού στο στοιχείο A[y][x] για την εκτίμηση της κατάστασης του στην επόμενη γενεά (νεκρό ή ζωντανό)
4. [Εάν δεν υπήρξε αλλαγή στο πλέγμα τερματισμός του προγράμματος (experimental)]

Το πρόγραμμα υλοποιείται στο αρχείο gol_serial_openmp.c. Σε περίπτωση μορφής OpenMP στο βρόχο γεννούνται πολλάπλα νήματα (της ίδιας μοναδικής διεργασίας) τα οποία εκτελούν ένα ορισμένο αριθμό επαναλλήψεων του βρόχου. Περισσότερα για OpenMP στην επόμενη ενότητα.

Στο αρχείο gol_serial_openmp.c θα βρείτε τον σειριακό κώδικα. Μεταγλωττίστε με
make serial ή make serial_openmp.
Εκτελέστε ως εξής πχ.
./gol_serial 128 1024 10000 0.25 1
ή
./gol_serial_openmp 128 1024 10000 0.25 1
ή
mpiexec -p 1 ./gol_mpi 128 1024 10000 0.25

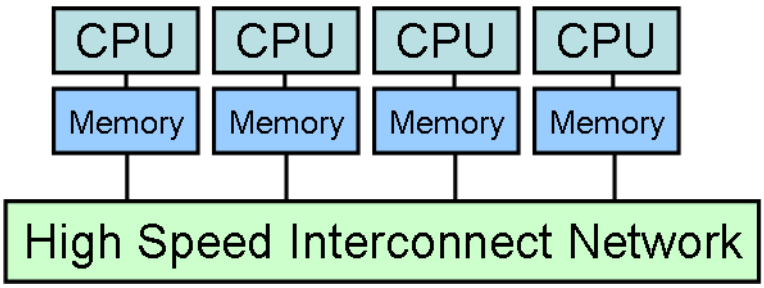
όπου 10,000 ο αριθμός γενεών και 0.25 ο λόγος αρχικού εν ζωή πληθυσμού οργανισμών.

Παραλληλοποίηση CPU

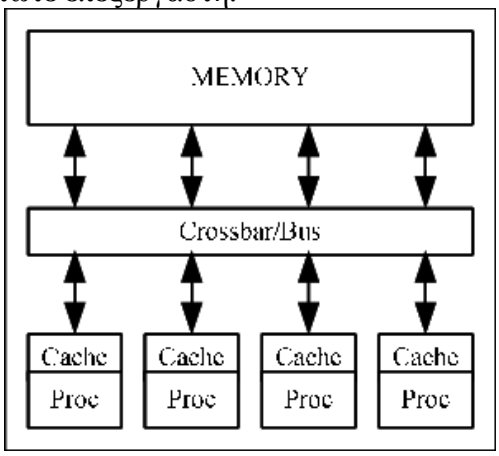
Στα πλαίσια αυτής της εργασίας αναπτύχθηκαν παράλληλα προγράμματα σε CPUs τύπου SIMD (Single Instruction Multiple Data), το οποίο είναι το πιο δημοφιλές της ταξονομίας Flynn που βρίσκουμε σε υπολογιστές τη σήμερον ημέρα. Με άλλα λόγια αυτό που παραλληλοποιείται σε κάθε περίπτωση είναι τα δεδομένα, όπου μια εντολή έχει τη δυνατότητα να εκτελείται σε διάφορες μονάδες δεδομένων ταυτόχρονα. Απαραίτητη προυπόθεση γι'αυτό είναι η ανεξαρτησία των δεδομένων και η ευελιξία στην παραλληλοποίηση τους.

Με παράλληλη επεξεργασία επιτυγχάνεται ταυτόχρονη εκτέλεση εντολών (μιας εντολής κάθε φορά) στους διάφορους επεξεργαστές / πυρήνες. Κάθε διεργασία εκτελείται στο δικό της "αφιερωμένο" χώρο εκτέλεσης έχοντας το δικό της αποκλειστικό χώρο μνήμης. Για να επικοινωνήσουν οι διεργασίες πρέπει να μεταφερθούν τα δεδομένα από τη μνήμη της μιας στη μνήμη της άλλης, μέσω ενός διαύλου επικοινωνίας (bus interconnect). Αυτή η παράλληλη αρχιτεκτονική έχει γίνει γνωστή ως αρχιτεκτονική κατανεμημένης μνήμης (distributed memory architecture), που η μνήμη είναι κατανεμημένη στους επεξεργαστές, δηλαδή κάθε επεξεργαστής διαθέτει τη δικιά του.

Αναπτύχθηκε το πρότυπο MPI για την εξυπηρέτηση τέτοιων υπολογιστών. Υλοποιήσεις του MPI αποτελούν το MPICH (η πιο γνωστή και αυτή που χρησιμοποιείται εδώ), το OpenMPI (μαζί με OpenMP), το MSMPI (Microsoft MPI το οποίο χρησιμοποιήθηκε με επιτυχία στην εκτέλεση στο Windows περιβάλλον) και το Intel MPI (ιδιότητο).



Η δεύτερη αρχιτεκτονική παράλληλης επεξεργασίας είναι αυτή της διαμοιρασμένης μνήμης (shared memory). Σύμφωνα με αυτή οι διεργασίες (που εδώ συνηθίζονται να αποκαλούνται νήματα) μοιράζονται ένα κοινό χώρο μνήμης. Τα νήματα μπορούν να επικοινωνούν μεταξύ τους χωρίς να στέλνουν "μηνύματα". Αυτό εφόσον μπορούν όλα να αναφέρονται σε κοινό χώρο διευθύνσεων; τη κύρια μνήμη (RAM) του εκάστωτε επεξεργαστή.



Για να επιτύχει η παραλληλοποίηση δεν αρκεί μονάχα το υλικό να την υποστηρίζει, αλλά πρέπει και να προγραμματίσουμε τα προγράμματα μας με κατάλληλο τρόπο ώστε να επωφεληθούν από τις επιδόσεις της αρχιτεκτονικής. Με άλλα λόγια πρέπει και το υλικό και το λογισμικό να συνεργάζονται αρμονικά για να παραχθεί το επιθυμητό αποτέλεσμα σε γρηγορότερο χρόνο σε σύγκριση με το σειριακό. Αλλά πρέπει και ο νούς μας να συνηθίσει σε έναν νέο τρόπο σκέψης, κάτι που αρχικά ίσως μας φαίνεται αντιδιαισθητικός, αλλά επιτυγχάνεται με πειθαρχία και με την εργασία που πραγματοποιήθηκε.

Γιατί θέλουμε να χτίσουμε παράλληλα προγράμματα;
Οι βασικοί λόγοι για να παραλληλοποιήσουμε τα προγράμματα μας είναι δύο, η επιτάχυνση (speedup) και η κλιμάκωση (scaling).

1. Επιτάχυνση είναι το ποσοστό ελάττωσης του χρόνου εκτέλεσης του παράλληλου προγράμματος σε σχέση με τον χρόνο εκτέλεσης του σειριακού προγράμματος. Εκφράζεται με τον παρακάτω νόμο:
   Sp = T_1 / T_N
   όπου
   T_1 : χρόνος εκτέλεσης σε 1 επεξεργαστή και
   T_N : χρόνος εκτέλεσης σε N επεξεργαστές
2. Κλιμάκωση είναι η ιδέα ότι όταν ν χειριστές (επεξεργαστές) εκτελούν ταυτόχρονα ένα έργο τότε θα το ολοκληρώσουν πιο γρήγορα από 1 χειριστή που εκτελεί το ίδιο έργο.

Υπάρχουν όμως παγίδες και προβλήματα που πρέπει να επισημάνουμε. Με γραμμική αύξηση του πλήθους των επεξεργαστών δεν έχουμε γραμμική μείωση του χρόνου εκτέλεσης. Υπάρχει ένα επιβαρύνων χρονικό διάστημα (overhead) που απαιτείται για την επικοινωνία και το συγχρονισμό μεταξύ των επεξεργαστών. Μάλιστα, αυτό το overhead αυξάνεται με την αύξηση των επεξεργαστών. Γενικότερα κρίνεται αναγκαίο να βρεθεί η χρυσή τομή μεταξύ του πλήθους των χειριστών και του όγκου εργασίας που πρέπει να εκτελέσει ο καθένας. Για προβλήματα με μικρό

όγκο δεδομένων πιθανόν να αποδειχτεί πιο γρήγορο να εκτελεστεί το πρόγραμμα από μονάχα έναν επεξεργαστή.

Έχουν θεσπιστεί δύο μέθοδοι για τη μέτρηση της κλιμάκωσης, η Ισχυρή και η Ασθενής κλιμάκωση.

1. Strong Scaling – Νόμος Amdahl:
Ο νόμος του Amdahl αναφέρει ότι η επιτάχυνση ενός παράλληλου προγράμματος περιορίζεται από τις σειριακές του περιοχές. Πάντα υπάρχει ένα τμήμα του προγράμματος το οποίο δεν μπορεί να παραλληλοποιηθεί. Γι'αυτό το λόγο καθώς αυξάνεται το πλήθος των επεξεργαστών τα πλεονεκτήματα της παραλληλοποίησης μοιραία μειώνονται, εφόσον το σειριακό κομμάτι είναι αυτό που θα καθορίσει τον τελικό χρόνο εκτέλεσης.

Καθώς αυξάνεται το μέγεθος ενός δεδομένου προβλήματος το χρονικό διάστημα μοντελοποίησης / εκτέλεσης του διατηρείται σταθερό εφόσον αυξάνεται και ο αριθμός των επεξεργαστών.

Εδώ συγκεκριμένα η επιτάχυνση υπολογίζεται με την παρακάτω εξίσωση:

$$Speedup = \frac{1}{S + \frac{P}{N}} \quad (1)$$

όπου
N: πλήθος επεξεργαστών
S: σειριακό μέρος του προγράμματος το οποίο δεν μπορεί να παραλληλοποιηθεί και
P: παράλληλο τμήμα προγράμματος
Φυσικά ισχύει ότι: S = 1 – P

Η αποδοτικότητα του νόμου (ποσοστιαία) υπολογίζεται θεωρητικά με τον ακόλουθο τύπο:

$$Efficiency = \frac{T_1}{T_N} * 100\ \% \quad (2)$$

2. Weak Scaling – Νόμος Gustafson:

Εδώ η επιτάχυνση υπολογίζεται θεωρητικά με την παρακάτω εξίσωση:

$$Speedup = S + N * P \quad (3)$$

Η αποδοτικότητα του νόμου υπολογίζεται με τον ακόλουθο τύπο:

$$Efficiency\ \frac{T_1}{T_N} * 100\ \% \quad (4)$$

[Βλέπετε ενότητα "Μετρήσεις" παρακάτω για περισσότερες λεπτομέρειες.]

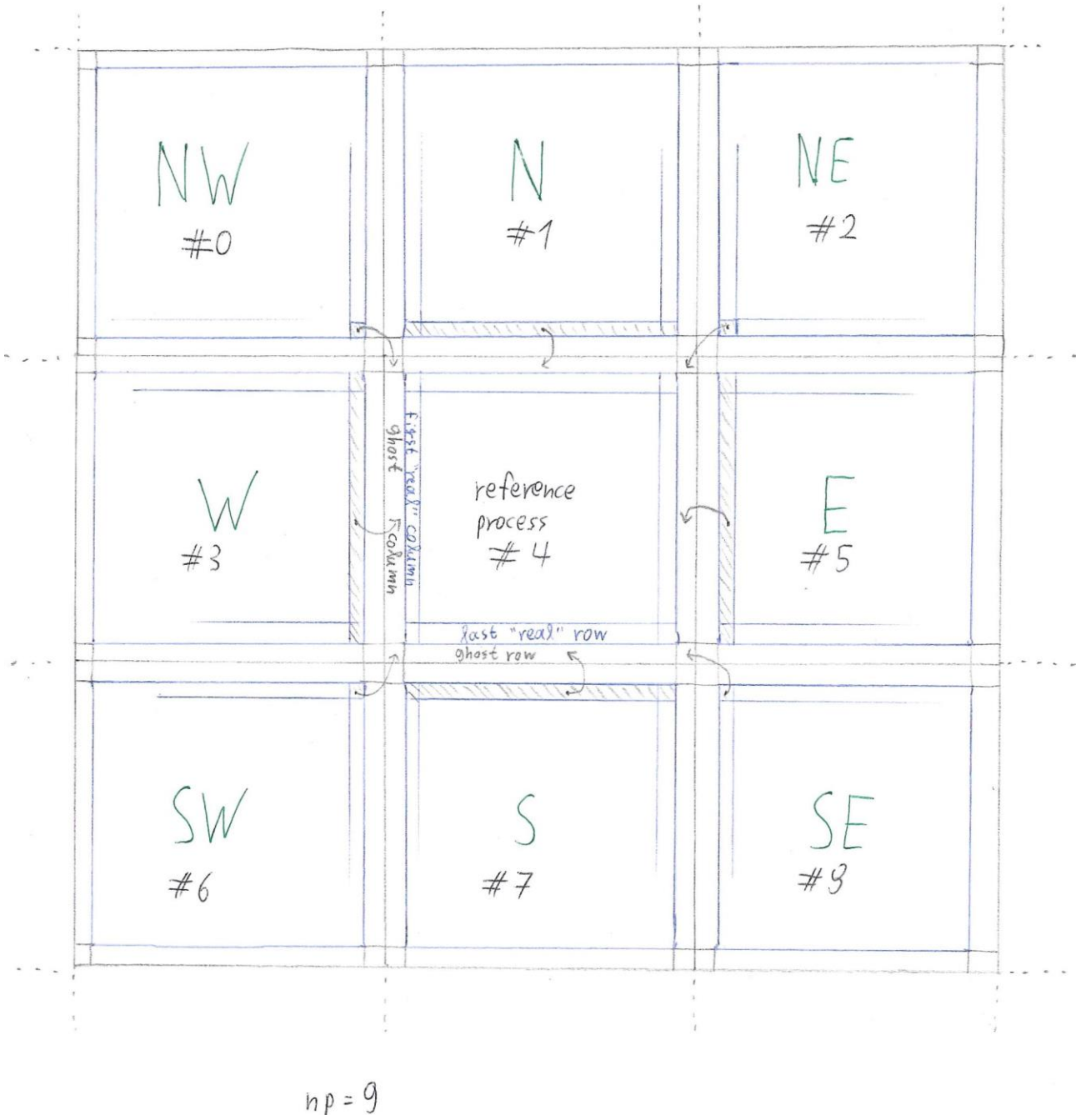
Σημαντική σημείωση που συχνά περνά απαρατήρητη:
Ας μην ξεχνάμε ότι με αύξηση των επεξεργαστών έχουμε και αύξηση του κόστους. Όταν εκτελούν N επεξεργαστές μια εργασία σε χρόνο T, τότε χρειαζόμαστε N φορές περισσότερη ισχύ για να τους λειτουργήσουμε για δεδομένο χρονικό διάστημα. Φυσικά ο 1 επεξεργαστής σίγουρα θα χρειαζόταν αρκετά επιπλέον χρόνο, που τότε θα δικαιολογούμε τη χρήση τους. Επίσης με την πρόοδο της τεχνολογίας η χρήση ισχύος των ηλεκτρονικών κυκλωμάτων ολοένα και μειώνεται, οπότε το πρόβλημα αντισταθμίζεται. Όλα αυτά πρέπει να λαμβάνονται υπόψη.

MPI (Message Passing Interface)

Ο παράλληλος προγραμματισμός είναι σε γενικές γραμμές πιο απλός, πιο «επίπεδος». Πολλοί καλοί ακολουθιακοί αλγόριθμοι είναι κακοί παράλληλοι αλγόριθμοι (πχ. quicksort) και πολλοί κακοί ακολουθιακοί αλγόριθμοι αποδίδουν αρκετά καλά όταν προγραμματισθούν παράλληλα (πχ. bubblesort).

Οι οδηγίες μας ήταν η κατανομή του πλέγματος του παιχνιδιού σε blocks. Πώς «διερμήνευσα» τις οδηγίες;
Το αρχικό πλέγμα του παιχνιδιού, οι διαστάσεις του οποίου καθορίζονται από την γραμμή εντολών διαιρείται σε np υποπλέγματα, όπου np το πλήθος των διεργασιών. Το κάθε υποπλέγμα είναι αφιερωμένο σε μία διεργασία. Επειδή ζητείται περιοδικό πλέγμα η κάθε διεργασία πρέπει να έχει 8 γείτονες. Επομένως σε κάθε γύρο επικοινωνεί με 8 το πολύ (στη περίπτωση που στο πρόγραμμα αποδόθηκαν λειγότερες από 8) διεργασίες. Οι γειτονικές διεργασίες αναφέρονται ως North, NorthEast, East, Southeast, South, Southwest, West, NorthWest και αναπαραστώνται στην επόμενο εικόνα (για μια «διεργασία αναφοράς»).

Διαπιστώνουμε ότι δημιουργείται ένας πίνακας από μπλόκ (/διεργασίες). Έχουμε ένα πλήθος σειρών από μπλόκ (prows) και ένα πλήθος στηλών από μπλόκ (pcols). Πρέπει δηλαδή prows * pcols = np. Για διευκόλυνση θέτω μία από δύο μεταβλητές prows και pcols να είναι ίση με 3. Η άλλη μεταβλητή απλά θα είναι np / 3. Το πλήθος γραμμών και στηλών κάθε μπλόκ πρέπει να είναι δύναμη του 2. Τέλος σε κάθε μπλοκ προστίθενται από δύο “φανταστικές” γραμμές (ghost rows) και στήλες (ghost columns). Αυτές χρησιμοποιούνται για την μετέπειτα λήψη και αποθήκευση των ακριανών δεδομένων των γειτονικών διεργασιών. Στην παρακάτω εικόνα παριστάνεται πως πραγματοποιείται η επικοινωνία δεδομένων μεταξύ διεργασιών. Βλέπουμε ποια δεδομένα μεταφέρονται από τη μια διεργασία στην άλλη. Σε κάθε επανάληψη πραγματοποιούνται 8 συνολικά αποστολές και 8 λήψεις. Για τις δύο γραμμές, δύο στήλες και τις 4 γωνιακές θέσεις του πίνακα.



Παρακάτω αναγράφονται τα βήματα του παράλληλου αλγορίθμου που αναπτύχθηκε. Αφότου είχε αναλυθεί ο ακολουθιακός αλγόριθμος η ανάπτυξη του παράλληλου έγινε πολύ πιο εύκολα. Αρκεί να λαμβανόταν το κομμάτι του προγράμματος το οποίο εκτελείται επαναλληπτικά στον κύριο βρόχο και να διαμορφωθεί καταλλήλως σύμφωνα με τους λογικούς και συντακτικούς κανόνες του MPI:

I. Αρχικοποίηση περιβάλλοντος και μεταβλητών,
II. Λήψη και επεξεργασία ορισμάτων προγράμματος (συνάρτηση parseCmd),
III. Για την κάθε διεργασία κατανέμεται χώρος για το τοπικό πλέγμα (grid) και το πλέγμα της επόμενης γενεάς (newGrid) (συνάρτηση allocContiguous2dArr) και έπειτα γίνεται αρχικοποίηση του πρώτου τοπικού πλέγματος με τυχαίο τρόπο (συνάρτηση randomlyPopulateGrid),
IV. Εύρεση γειτονικών διεργασίων
V. Έναρξη βασικού βρόχου (Game Loop). Κάθε διεργασία εκτελεί τα παρακάτω:

a. Αποστολή δεδομένων στις ακμές και γωνίες του τοπικού πλέγματος (συναρτήσεις MPI_Send*)
b. Λήψη δεδομένων στα φανταστικά κελιά (ghost cells) (συναρτήσεις MPI_Recv*)
c. Εκτίμηση του πλέγματος για την επόμενη γενεά (συνάρτηση evolve) η οποία για το κάθε κελί του τοπικού πλέγματος εκτελεί τα ακόλουθα:
    1. Μέτρηση γειτόνων (συνάρτηση countNeighbors)
    2. Εκτίμηση κατάστασης του αντίστοιχου κελιού για το πλέγμα της επόμενης γενεάς σύμφωνα με τους 4 κανόνες του παιχνιδιού και αποθήκευση της κατάστασης στο νέο πλέγμα. Επιστρέφεται το νέο πλέγμα (newGrid)
d. Το πλέγμα της επόμενης γενεάς (grid) λαμβάνει την νέα του κατάσταση από το newGrid (που λήφθηκε από τη συνάρτηση evolve). Το newGrid αποδεσμεύεται.

VI. Αποδέσμευση πινάκων και τερματισμός παράλληλου περιβάλλοντος

Πρέπει να βεβαιωθούμε ότι όλες οι διεργασίες είναι συγχρονισμένες σε κάθε επανάληψη. Γι'αυτό το λόγο τοποθετούμε τη συνάρτηση MPI_Barrier() αμέσως μετά το τέλος της επικοινωνίας. Σε εκείνο το σημείο η κάθε διεργασία περιμένει μέχρι να φτάσουν όλες εκεί.

Όταν διαμετακομίζεται ένας πίνακας (container object) μεταξύ δύο διεργασιών, το σύστημα του MPI το διαχειρίζεται κατάλληλα τοποθετώντας το σε έναν buffer και έπειτα το στέλνει στη διεργασία στόχο. Εάν το array container είναι πιο μεγάλο από τον buffer, τότε το σύστημα αυτόματα το διασπά κατάλληλα και το αποστέλει τεμαχισμένο (segmented). Στη διεργασία στόχο ανακατασκευάζεται, διαμορφώντας τον αρχικό πίνακα.

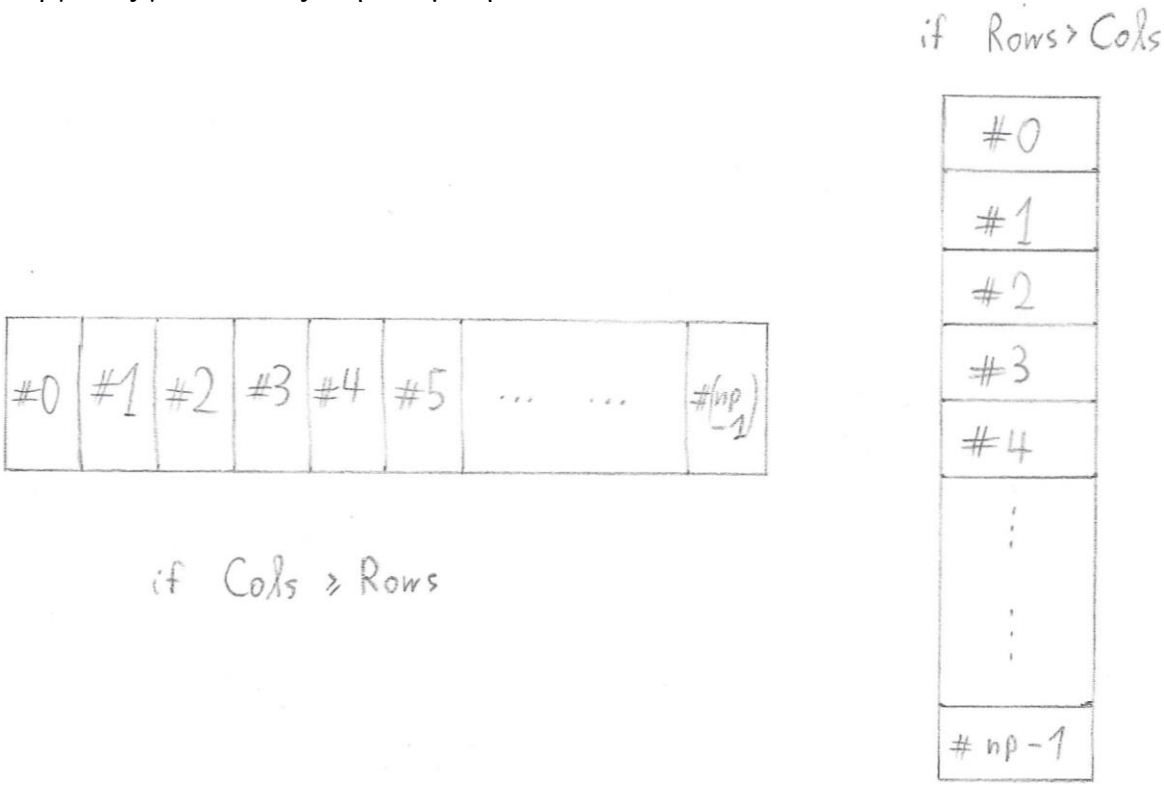
Βελτιστοποιήσεις

Όπως ίσως παρατηρήσατε γίνεται χρήση μιας βελτιστοποιημένης τεχνικής για την αποστολή και λήψη δεδομένων (αντί να χρησιμοποιούσαμε τους ξερούς τρόπους με MPI_Send() & MPI_Recv). Αυτή επιτελείται με τις συναρτήσεις MPI_Send_init() και MPI_Recv_init(). Το πλεονέκτημα που μας προσφέρουν είναι ότι αντί να υπολογίζονται σε κάθε επανάληψη τα ορίσματα επιστρέφεται ένα handle με όλες τις παραμέτρους για αυτό το σταθερό αίτημα επικοινωνίας. Έπειτα όποτε θέλουμε να στείλουμε / λάβουμε τα δεδομένα χρησιμοποιούμε αυτό το handle ως όρισμα στην MPI_Start() και MPI_Wait (/MPI_Waitall).
Αυτές οι συναρτήσεις εγκαθιδρύουν ένα σταθερό επικοινωνιακό δίαυλο μεταξύ διεργασιών. Με αυτό τον τρόπο αρκεί να στέλνουμε / λαμβάνουμε δεδομένα σε κάθε επανάληψη χωρίς πλεονέζοντα χρόνο (overhead) για θέματα όπως υπολογισμό και εκτίμηση παραμέτρων, παρακολούθηση μεταβλητών των παραμέτρων κτλ.
Υπάρχει και μια λιγότερο αποδοτική συνάρτηση την οποία αναφέρω για πληρότητα (και έχω επισυνάψει στον κώδικα σε σχολιασμένη μορφή). Αυτή πραγματοποιείται με τη συνάρτηση MPI_Sendrecv() η οποία εκτελεί: Send + Wait (Block μέχρι να επιστραφούν τα δεδομένα) + Receive. Δεν υπάρχει non-blocking εκδοχή της τελευταίας συνάρτησης.

Corner cases

Υπέθεσα εξ'αρχής ότι ένα τέτοιο πρόγραμμα θέλουμε να κλιμακώνει σε τουλάχιστον δεκάδες επεξεργαστές. Το cluster του πανεπιστημίου τελικά δεν αποδείχτηκε τόσο εύρωστο. Αρχικά είχα κατασκευάσει το πρόγραμμα υποθέτοντας ότι θα εκκινήσουν το λιγότερο 9 διεργασίες (για την εξυπηρέτηση της παρούσας και όλων των γειτονικών διεργασιών). Έπειτα όμως κρίθηκε ανάγκη για υποστήριξη και μικρότερου αριθμού επεξεργαστών όπως ορίζεται στην είσοδο με:
mpiexec -p [#] ./gol_mpi rows cols generations density
όπου [#] είναι ο αριθμός των επεξεργαστών. Αυτό διότι έπρεπε να πάρω μετρήσεις σε ένα node του πανεπιστημίου με μονάχα 2 επεξεργαστές, έπειτα σε 2 nodes με 4 συνολικά CPUs κ.ο.κ. Τελικά το πρόγραμμα μπορεί και ανταποκρίνεται κατάλληλα με οποιοδήποτε αριθμό επεξεργαστών από 1 έως και 8. Όταν όμως οι διεργασίες είναι περισσότερες από 9 τότε πρέπει το πλήθος τους να είναι πολλαπλάσιο του 3 για να εξυπηρετηθεί το domain decomposition που επελέγη.

Όταν οι διεργασίες είναι λιγότερες από 9 τότε η διαμέριση των δεδομένων και η ανάθεση τους σε διεργασίες γίνεται όπως στην επόμενη εικόνα:



Σε αυτή την περίπτωση η κατανομή γίνεται σε 1 σειρά * n στήλες από μπλόκς (εάν rows <= columns) ή σε n σειρές * 1 στήλη (εάν rows > columns). Η λογική αυτή υλοποιείται στη συνάρτηση επεξεργασίας ορισμάτων parseCmd() και στο main loop ρυθμίζεται αντίστοιχα.

OpenMP

Τα προγράμματα που κάνουν χρήση του προτύπου παράλληλου προγραμματισμού με κοινή μνήμη, το OpenMP, εκκινούν με ένα νήμα (η αρχική διεργασία). Όταν η εκτέλεση φτάσει σε εντολή προεπεργαστή της παρακάτω μορφής:
#pragma omp [code_construct]
τότε γεννούνται νήματα εργάτες (worker threads) στην έναρξη μιας παράλληλης περιοχής (parallel region). Μεταξύ παράλληλων περιοχών οι εργάτες διακόπτουν τη λειτουργία τους. Το πλαίσιο εκτέλεσης του OpenMP διαχειρίζεται στο παρασκήνιο όλες τις χαμηλού επιπέδου λεπτομέρειες για την εξυπηρέτηση των threads.

Κρίσιμο και σύνηθες σημείο για την έναρξη μιας παράλληλης περιοχής είναι οι βρόχοι. Ένας βρόχος με τα παρακάτω χαρακτηριστικά μπορεί να παραλληλοποιηθεί, διότι δεν υπάρχουν αλληλοεξαρτήσεις στα δεδομένα:

1. Όλες οι αναθέσεις σε κοινή μνήμη γίνονται σε πίνακες
2. Κάθε στοιχείο / συγκεκριμένη θέση μνήμης ελέγχεται από το πολύ μια επανάληψη του βρόχου
3. Καμιά επανάληψη δεν διαβάζει ή γράφει δεδομένα από άλλη επανάληψη του βρόχου

Με αυτές τις παραδοχές η εκτέλεση του βρόχου μπορεί να γίνει παράλληλα από πολλαπλές διεργασίες.

Τι διαμοιράζεται μεταξύ των threads;

1. Καθολικές μεταβλητές (file scope variables) είναι κοινές εξ'ορισμού

2. Οτιδήποτε δηλώνεται μέσα σε shared() κατά την έναρξη της παράλληλης περιοχής είναι κοινό

Τι είναι ιδιωτικό σε ένα νήμα;

1. Η μνήμη stack και οι αυτόματες μεταβλητές είναι ιδιωτικές στο νήμα
2. Οτιδήποτε βρίσκεται μέσα σε private() κατά την έναρξη μιας παράλληλης περιοχής είναι ιδιωτικό

Στη διάλεκτο του OpenMP η συλλογή νημάτων που εκτελούν ένα παράλληλο χωρίο αποκαλείται ομάδα (team).

Για την αποφυγή ορισμένων δυσάρεστων bugs στα προγράμματα προτείνεται η χρήση
#pragma omp parallel
ξεχωριστά από τη δήλωση της εκάστωτε λειτουργίας (πχ. εκτέλεση βρόχου for). Για παράδειγμα προτιμάται αυτό:
#pragma omp parallel num_threads(2)
;
i = 0;
#pragma omp for private(prow, pcol)
από αυτό:
#pragma omp parallel num_threads(2) for private(prow, pcol).

Επίσης προσοχή στο σημείο που δηλώνονται οι μεταβλητές σε ένα παράλληλο χωρίο. Εάν δηλωθούν έξω από αυτές τότε είναι κοινές στα νήματα. Διαφορετικά είναι ιδιωτικές στα νήματα.

Στο αρχείο gol_mpi_openmp.c θα βρείτε τον MPI, OpenMP κώδικα. Όπως ειπώθηκε, μπορεί να εκτελεστεί και με 1 διεργασία, που σε αυτή την περίπτωση η εκτέλεση θα γίνει με τον ίδιο τρόπο που γίνεται στο σειριακό πρόγρμαμα στο αρχείο gol_serial_opemp.c. Μεταγλωττίστε με make mpi ή make mpi_openmp.

Εκτελέστε με πχ.:
mpiexec -p 2 ./gol_mpi 128 1024 10000 0.3

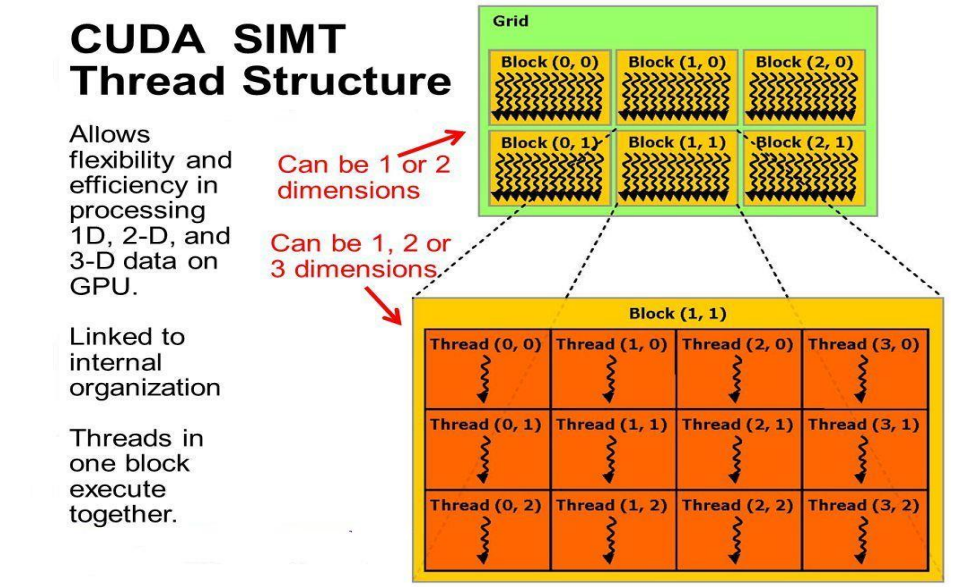
Επίσης στο αρχείο Intel Core i7 3770 Specifications.docx αναγράφονται οι προδιαγραφές του επεξεργαστή μου που επίσης χρησιμοποίησα για να πειραματισθώ με την εργασία.

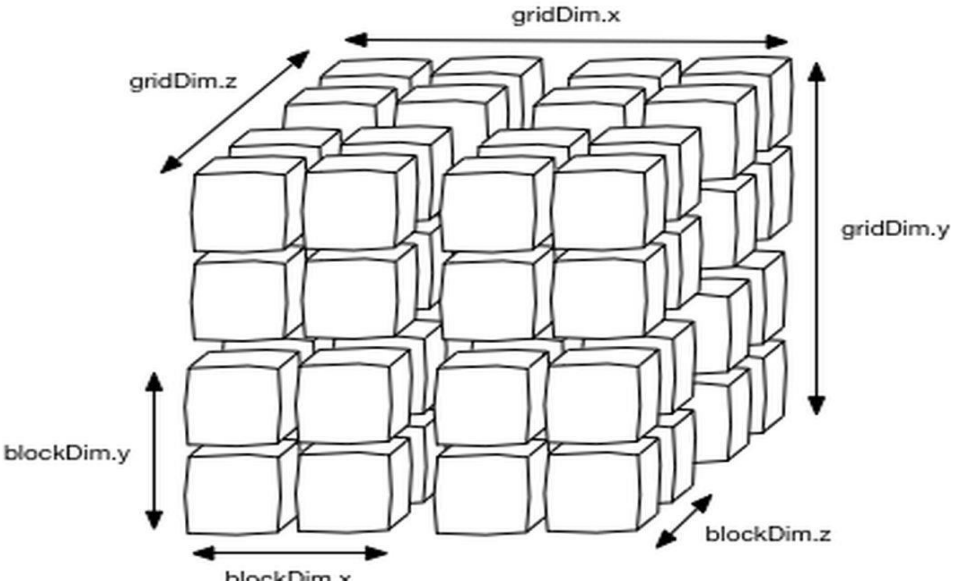
GPU Παράλληλο πρόγραμμα - CUDA

Οι «παραδοσιακές» γλώσσες προγραμματισμού δεν είναι επαρκής για τον προγραμματισμό γραφικών επεξεργαστών (GPUs). Η CUDA είναι μια επέκταση στη C/C++ που επιτρέπει τον προγραμματισμό καρτών γραφικών NVidia. Χρησιμοποιείται για τον ορισμό των kernels και διαχειρίζεται αυτόματα όλα τα διαφορετικά τμήματα μνήμης στην κάρτα γραφικών.

Οι GPUs προσφέρουν σημαντικά μεγαλύτερη απόδοση στην εκτέλεση προγραμμάτων με εξειδικευμένη εφαρμογή. Συνήθως όταν υπάρχουν πολλά δεδομένα ανεξάρτητα το ένα από το άλλο. Επισημαίνεται όμως ότι οι GPUs όμως δεν μπορούν να χρησιμοποιηθούν μόνες τους. Είναι συνεπεξεργαστές (coprocessors). Απαιτείται και ο βασικός επεξεργαστής του υπολογιστικού συστήματος (CPU) για την εκτέλεση πολύπλοκων λειτουργιών χάρις στα ποικίλου είδους μικροηλεκτρονικά κυκλώματα που βρίσκονται στο τσιπ των CPU.

Κάθε cuda thread έχει τη δικιά του ταυτότητα threadId.x, threadIdx.y & threadIdx.z που το ταυτοποιεί εντός ενός block και το δικό του blockIdx.x που ταυτοποιεί το εκάστωτε block εντός του Grid. Αυτή η ταυτότητα του thread χρησιμοποιείται κατά κόρον ως το index σε έναν βρόχο. Λέμε λοιπόν ότι τα blocks ταυτοποιούνται χρησιμοποιώντας 1, 2 ή 3 διαστάσεις. Η επόμενη εικόνα απεικονίζει τη δομή των threads και των blocks στο grid της GPU.





Ένα kernel εκκινεί με πολλαπλά νήματα σε ένα block. Αυτά τα νήματα μπορούν να επικοινωνήσουν μέσω της global ή της shared μνήμης.

The CUDA cores get only one instruction at a time by the SM (SIMD model), which means that inside the SM they all execute the same instruction on a set of data.
The goal is to choose a block size that maximizes the size of active threads in a SM.

Μεταφορά του CPU παράλληλου κώδικα στην GPU.
Κατά μεγάλο μέρος η ροή του κώδικα είναι παρόμοια. Οι διαφορές είναι οι παρακάτω:

1. Διατηρούμε τις CPU εκδόσεις των μεταβλητών / πινάκων και τις κάνουμε (a) malloc & (b) cudaMemcpy() για να τις αντιγράψουμε και στην GPU VRAM.
2. Φτιάχνουμε τα κατάλληλα kernels για τις συναρτήσεις που θα εκτελεστούν στη GPU. Το kernel είναι ένας συνδυασμός κανόνων εκτέλεσης, όπως πληθος νημάτων σε κάθε διάσταση

(x,y,z), δομή νημάτων, πλήθος διαμοιραζόμενης μνήμης και άλλα στοιχεία που θέτονται αυτόματα από τη μεταγλώττιση CUDA.

3. Κατά τον προγραμματισμό GPU προσέχουμε ιδιαίτερα τα παρακάτω που καταστρέφουν την απόδοση:
    a. Αντιγραφή δεδομένων από και προς τη GPU!!!
    b. GPU δεν χρησιμοποιείται συνεχώς - GPU device under-utilization. Επιφέρει ευδιάκριτο latency

Η διαμοιραζόμενη (shared) μνήμη είναι η κοινή μνήμη στα νήματα ενός block. Είναι έως και 2 τάξεις μεγέθους πιο γρήγορη από τη global μνήμη. Η CPU δεν έχει πρόσβαση στη shared memory. Πρέπει να γίνει αντιγραφή των δεδομένων από τη global στην shared μνήμη για να χρησιμοποιηθεί σωστά η shared μνήμη. Ακολουθούν επιπλέον θέματα που πρέπει να λάβουμε υπόψη για τον προγραμματισμό του προγράμματος ώστε να γίνεται χρήση της shared μνήμης:

1. Κάθε νήμα διαβάζει από τη global μνήμη στη shared μνήμη
2. Συγχρονισμός νημάτων (__syncthreads()) για να διασφαλίσουμε ότι όλα έχουν διαβάσει τα στοιχεία τους από τη global μνήμη
3. Έπειτα κάθε νήμα εκτελεί λειτουργίες χρησιμοποιώντας δεδομένα της shared μνήμης
4. Κάθε νήμα γράφει το αποτέλεσμα από τη shared μνήμη πίσω στη global μνήμη ώστε εντέλει να γίνει φανερό στη CPU.

Σημειώνεται ότι τα πλεονεκτήματα που προσφέρει η χρήση της shared μνήμης γίνονται εμφανή όταν γίνεται εκτενής χρήση αυτών των δεδομένων. Σε διαφορετική περίπτωση η απόδοση μπορεί να είναι χειρότερη από τη global μνήμη, εξαιτίας των επιπλέον αντιγραφών που απαιτούνται μεταξύ των τμημάτων μνήμης. Δυστυχώς η απόδοση της shared μνήμης στο πρόγραμμα μου ήταν ελάχιστα χειρότερη από της global μνήμης. Θα το διαπιστώσετε επειδή χρειάζεται μερικά επιπλέον δευτερόλεπτα για να ολοκληρωθεί. Δεν είμαι σίγουρος εάν φταίει αυτό ή κάτι άλλο το οποίο δεν έχω καταλάβει. Είναι το μοναδικό σημείο στο οποίο έχω αμφιβολία. Εάν μάθετε τι συμβαίνει ενημερώστε με σας παρακαλώ.

Στο αρχείο gol_cuda.cu θα βρείτε τον κώδικα για εκτέλεση στη GPU. Περιέχεται μαζί ο κώδικας για εκτέλεση με τη βοήθεια είτε της global, είτε της shared μνήμης. Ο έλεγχος αυτός γίνεται με εντολές προεπεξεργαστή. Μεταγλωττίστε με make cuda ή με make cuda_shared.

Έπειτα εκτελείτε με πχ. :
./gol_cuda 128 1024 10000 0.3
ή
./gol_cuda_shared 128 1024 10000 0.3

Επίσης στο αρχείο Nvidia GTX 660 GPU Specifications αναφέρονται οι προδιαγραφές της κάρτας γραφικών μου που χρησιμοποίησα για την εκτέλεση του προγράμματος.

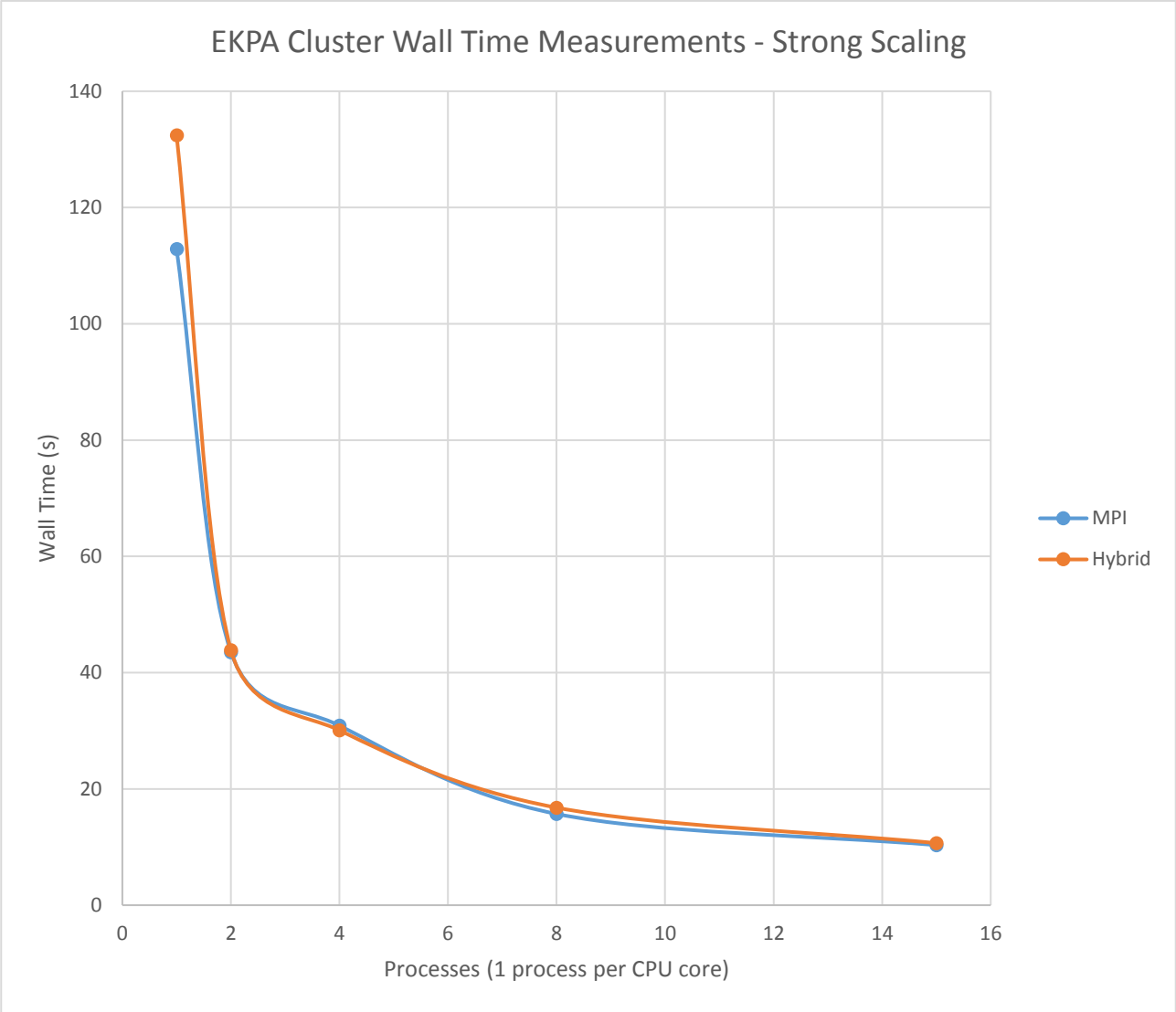
Μετρήσεις

Στο αρχείο Game_of_Life_M1485_metrics.xlsx περιέχεται φύλλο εργασίας με τις μετρήσεις για strong και weak scaling και για τη GPU και συνοδεύται από γραφήματα. Όπως προαναφέρθηκε έγιναν μετρήσεις στο cluster της σχολής και στο προσωπικό μου σύστημα.

Οι μετρήσεις αυτές έγιναν για να διαπιστώσουμε εάν το πρόγραμμα μας κλιμακώνει και εάν κλιμακώνει καλά. Ένα πρόγραμμα ονομάζεται κλιμακώμενο (scalable) εάν το μέγεθος του

προβλήματος προβλήματος μπορεί να αυξηθεί με ρυθμό τέτοιο ώστε η αποδοτικότητα του να μην μειωθεί, καθώς το πλήθος των επεξεργαστικών μονάδων αυξάνεται.

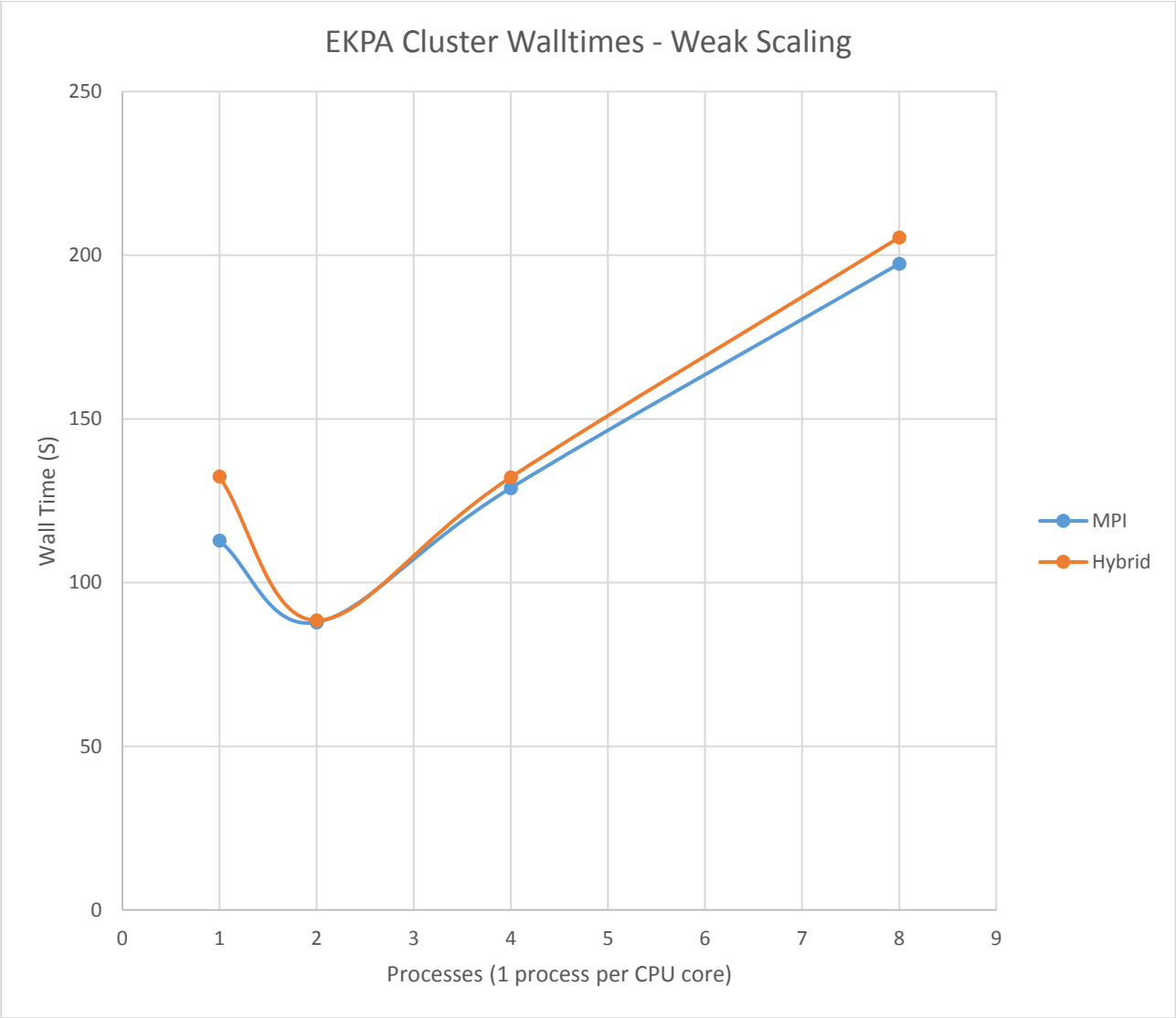
Πρακτικά και όπως φαίνεται και στα παρακάτω γραφήματα το κριτήριο που προσθέτει η ασθενής κλιμάκωση είναι πιο δύσκολο να επιτευχθεί. Γι'αυτό η ισχυρής κλιμάκωση είναι αυτή που χρησιμοποιείται σε μετρήσεις κατά κόρον.



Είναι προφανές και αναμενόμενο ότι με αύξηση του πλήθους των επεξεργαστών μειώνεται ο χρόνος εκτέλεσης.

Σε αντίθεση όμως, η ασθενής κλιμάκωση «ασθενεί» με αυξημένο αριθμό επεξεργαστών. Αποδεικνύεται αποδοτική μονάχα με 2 επεξεργαστές.

Φυσικά όλα αυτά διαφερούν και αποκλίνουν από το ένα σύστημα στο άλλο αλλά ο γενικός κανόνας παραμένει αμετάβλητος. Ο νόμος Amdahl είναι το καλύτερο κριτήριο αξιολόγησης της κλιμάκωσης προγραμμάτων σε παράλληλες αρχιτεκτονικές υπολογιστών.



Εξαιτίας των προβλημάτων του Cluster δεν ολοκληρώθηκε η εργασία με τα προγράμματα extrae και paraver. Ό,τι χρησιμοποίησα μπορείτε να το βρείτε αφού το επισυνάπτω στο φάκελο που συνοδεύει αυτό το έγγραφο.

Μεθοδολογία Foster

Ο Ian Foster ανέπτυξε μια διαδικασία 4 σταδίων για τη σχεδίαση παράλληλων προγραμμάτων. Γενικώς για τα περισσότερα προβλήματα υπάρχουν παράλληλες λύσεις. Τα στάδια αυτά είναι:

1. Διαμέριση - Partitioning: Διαχωρισμός του αρχικού προβλήματος σε πολλαπλά ανεξάρτητα υποπροβλήματα διακριτού μεγέθους και επίλυση του καθενός ξεχωριστά. Τελικά τα υποπροβλήματα συνδυάζονται σχηματίζοντας τη νέα μορφή του αρχικού προβλήματος.
2. Επικοινωνία - Communication: Συνίσταται πρωτίστως στην εύρεση ενός αποδοτικού αλγορίθμου κατάλληλου επικοινωνίας μεταξύ των διεργασιών. Ο αλγόριθμος συνήθως θα είναι στοχευμένος για το συγκεκριμένο πρόβλημα.
3. Συσσώρευση - Agglomeration: Ανασκόπηση βημάτων 1, 2 με σκοπό το σχηματισμό κατάλληλου αλγορίθμου για εκτέλεση σε κάποιο είδος παράλληλου υπολογιστή. Πολλές αποφάσεις πρέπει να ληφθούν σε αυτό το στάδιο, οι οποίες θα καθορίσουν την ποιότητα

εκτέλεσης του προγράμματος. Στόχος είναι η παραγωγή ορθού αποτελέσματος για κάθε είδος εισόδου στο πρόβλημα (soundness & completeness),

4. Mapping: Ανάθεση διεργασιών για την επίλυση των υποπροβλημάτων. Το πλήθος διαθέσιμων υπολογιστικών μονάδων / επεξεργαστών συνήθως διαφέρει από το πλήθος των υποπροβλημάτων. Επιζητούμε τον ισοζυγισμό των επεξεργαστών. Οι διεργασίες θα πρέπει να εκτελούν ταυτόχρονα, λαμβάνοντας υπόψη την χωρική και χρονική τοπικότητα. Στόχος είναι η ελαχιστοποίηση του χρόνου εκτέλεσης. Επίσης είναι επιθυμητό όλοι οι επεξεργαστές να επικοινωνούν και φυσικά μεταξύ τους.

Σε αντιπαραβολή με τη δική μας υλοποίηση του Παιχνιδιού της Ζωής, η διαμέριση έγινε σε διδιάστατα μπλόκς. Η κάθε διεργασία λάμβανε ένα μεγάλο πλήθος από δεδομένα και στο καθένα εκτελούσε ένα συγκεκριμένο αριθμό εντολών. Θα μπορούσε να γίνει διαμερισμός κατά σειρές. Δηλαδή κάθε διεργασία λαμβάνει ολόκληρο το πληθος στηλών του πλέγματος αλλά (Rows / np) πλήθος γραμμών, όπου Rows το συνολικό πλήθος γραμμών και np το πλήθος των διεργασιών (η τελευταία ή η πρώτη διεργασία μπορεί να λαμβάνει τις όποιες εναπομείναντες σειρές).

Η συγκεκριμένη υλοποίηση είναι υπολογιστικά πιο εύκολη. Επίσης το κόστος επικοινωνίας μειώνεται ριζικά διότι η κάθε εργασία έχει πλέον 2 (και όχι 8!) γείτονες, την διεργασία που είναι υπεύθυνη για τις «από πάνω» σειρές και την διεργασία υπεύθυνη για τις «από κάτω» σειρές. Πρόκειται για coarse grained parallelism, σε αντίθεση με την υλοποίηση σε μπλοκς που υλοποιήσαμε εμείς η οποία ήταν fine grained στη γενική περίπτωση. Ενδεχομένως ο χρόνος εκτέλεσης εδώ να ήταν σημαντικά μικρότερος, ιδιαίτερα με ηλεκτρονικούς υπολογιστές με συγκριτικά αυξημένη συχνότητα λειτουργίας του κάθε πυρήνα.

Απεικόνιση παιχνιδιού

Εκτός από τον προγραμματισμό της κλιμάκωσης του παιχνιδιού, προγραμματίσθηκε το παιχνίδι ώστε να μπορούμε να βλέπουμε την εξέλιξη του σε πραγματικό χρόνο. Μπορείτε να ενεργοποιήσετε αυτή την επιλογή κατά την μεταγλώττιση ως εξής:
make serial_display ή
make mpi_display ή
make mpi_openmp_display.

Εκτελείτε το πρόγραμμα κανονικά με τις ίδιες εντολές που αναφέρθηκαν στις προηγούμενες ενότητες.

Ο προγραμματισμός του παιχνιδιού για την απεικόνιση του real-time με δυσκόλεψε ιδιαιτέρως. Αποφάσισα όμως ότι ήθελα να το κάνω, αφού είδα διάφορα βίντεο (πχ. στο YouTube) με παρόμοια προγράμματα τα οποία με εντυπωσίασαν.

Όλος ο κώδικας για τον προγραμματισμό αυτό βρίσκεται εγκλεισμένος σε εντολές προεπεξεργαστή που ενεργοποιούνται εάν δηλωθεί ο όρος DISPLAY_RESULTS.

Αναφέρονται παρακάτω ορισμένα σημεία που χρίζουν ιδιαίτερης προσοχής. Το MPI υποθέτει ότι τα στοχεία των πινάκων που στέλνονται και λαμβάνονται μεταξύ διεργασιών βρίσκονται σε συνεχόμενες θέσεις μνήμης. Στη C/C++ δεδομένα στην ίδια σειρά ενός πίνακα βρίσκονται σε συνεχόμενες θέσεις μνήμης. Επομένως μπορεί να γίνει send / recv εύκολα. Οι στήλες όμως δεν βρίσκονται σε συνεχόμενες θέσεις μνήμης (κατ' αντιδιαστολή προς τη Fortran). Πρέπει να διαθέτουμε ένα ξεχωριστό τύπο δεδομένων για τις στήλες (πχ. vector).

Το MPI προσφέρει τέτοια δυνατότητα με MPI_Type_create_subarray() και MPI_Type_commit(). Φτιάχνουμε τον ιδιαίτερο πίνακα μη συμπεριλαμβάνοντας τα φανταστικά κελιά με το MPI_Type_create_subarray() τον οποίο θα στείλουν όλες οι διεργασίες στη master διεργασία 0. Έπειτα η διεργασία 0 απεικονίζει ολόκληρο τον πίνακα - ονομαζόμενος universe - με τη συνάρτηση display2dGrid(). Αφήνεται ένα περιθώριο μισού δευτερολέπτου μεταξύ διαδοχικών επαναλήψεων για να μπορέσουμε να δούμε τη διαδικασία.

Επεκτάσεις

Μια πιθανή ενδιαφέρουσα επέκταση του προγράμματος θα ήταν η πρόσθεση δυνατοτήτων για Parallel IO. Σύμφωνα με αυτό ο αρχικός πίνακας δεδομένων δε θα παράγεται τυχαία από τις διεργασίες αλλά θα διαβάζεται από πολλαπλές διεργασίες εν παραλλήλω από ένα αρχείο. Το MPICH προσφέρει τις παρακάτω συναρτήσεις για την εκπλήρωση αυτού του σκοπού:

- MPI_File_open(Communicator, char* filename, int accessMode = MPI_MODE_RDONLY, MPI_Info info, MPI_File &fh) : Open file for reading (accessMode = MPI_MODE_RDONLY) and produce output file fh
- MPI_File_get_size(MPI_File fh, MPI_Offset *outputSizeBytes) : Get size of the file. bufSize = fileSize / nProcesses. MPI_SEEK_SET : The pointer is set to offset
- MPI_File_seek(fileHandle, MPI_Offset offset (= rank * bufsize), int whence) : Each process reads in their own set of data from the file. Offset should be followed, read of written MPI_SEEK_SET
- MPI_File_read(fh, MPI_Offset offset (output), bufSize (sizeof(int)), MPI_Datatype (= MPI_INT), MPI_Status *status (output status))
- MPI_File_close (&fh)

Συμπεράσματα

Κατά την ολοκλήρωση αυτής της εργασίας απέκτησα αυτοπεποίηση με τον παράλληλο προγραμματισμό, τα οφέλη που παρέχει και τις επιπλέον τεχνικές δυσκολίες που προσθέτει. Εξεικοιώθηκα με παράλληλους αλγορίθμους και έμαθα να σκέπτομαι διαφορετικά, γεγονός το οποίο διεύρυνε τους ορίζοντες μου. Η λήψη λεπτομερών μετρήσεων και η μελέτη κλιμάκωσης μου δίδαξε την δυσκολία στη τέχνη των στατιστικών μετρήσεων. Τέλος κατανόησα τη φύση των manycore επεξεργαστών (GPUs), τις ομοιότητες και διαφορές τους με τους CPU.

Θεωρώ ότι έφερα εις πέρας το έργο και τους στόχους που διέπου το συγκεκριμένο μαθήμα και είμαι ικανοποιημένος από τις νέες γνώσεις τις οποίες μόχθησα για να αποκτήσω.