Απόστολος Σουργκούνης Μεταλλίδης
ΑΜ 1115201400187
Μυρτώ Χουλιαρά
ΑΜ 1115201400227

Παραδοτέα :
gol_serial.c : Το πρόγραμμα υλοποιημένο ακολουθιακά σε C
gol_mpi.c : Το πρόγραμμα υλοποιημένο μονο με MPI
gol_openmp.c : Το πρόγραμμα υλοποιημένο μονο σε OpenMP
gol_mpi_omp.c : Το πρόγραμμα υλοποιημένο σε Hybric MPI + OpenMP
gol_cuda.cu : Το πρόγραμμα υλοποιημένο σε Cuda

Σχεδιασμός και Υλοποίηση:

**MPI:** Το πρόγραμμα δέχεται απο τον χρήστη κατα την εκτέλεση το μέγεθος του πίνακα που περιέχεται σε κάθε μπλοκ (μέγεθος N*N), καθώς και εναν αριθμό γενεών για τις οποίες θα τρέξει αν καμια απο τις συνθήκες τερματισμού δεν ισχύει. Επομένως, αν π.χ. Ο χρήστης δώσει ένα μέγεθος πίνακα 100*100 και 4 processes, ο τελικός πίνακας στον οποίο θα εφαρμοστεί το Game of life θα έχει μέγεθος 400*400. Σε περίπτωση που δεν δωθεί αριθμός γενεών οριζεταί σαν threshold το 999 ώστε να μην παρεμποδίζει τις συνθήκες τερματισμού αλλα να αποφευχθεί και η περίπτωση ατέρμον βρόγχου .

Οσον αφορά τα block, ο διαμερισμός γίνεται με MPI_Cart_create, οι διαστάσεις του οποίου βρίσκονται ανάλογα με τον αριθμό processes που δίνεται απο τη γραμμή εντολών. Οι πίνακες (δεικτες int**) που χρησιμοποιεί ο καθε process έχουν (N+2)*(N+2) και η δέσμευση τους γίνεται σε συνεχόμενες θέσεις μνήμης. Προσθέτοντας επιπλέον 2 σειρές και 2 στήλες, ο έξτρα χώρος που δημιουργείται θα γεμίζει κάθε φορα με τα δεδομένα των γειτόνων, ωστε να γίνουν οι υπολογισμοί των στοιχείων στις άκρες. Για λόγους ευκολίας, θεωρούμε αυτες τις σειρές ως την πρώτη και την τελευταία του πίνακα board, αντίστοιχα και με τις γραμμές. Ο υπόλοιπος πίνακας θα γεμίσει τυχαία με 0 και 1.

Η ανταλλαγή μηνυμάτων μεταξύ processes εφαρμόζεται με Isend και Irecv. Κάθε process στέλνει στους γείτονες της, που έχουν υπολογιστεί πιο πριν με MPI_Cart_shift, την πρώτη και τελευταία στήλη, καθώς και την πρώτη και τελευταία γραμμή και ξεχωριστά τις γωνίες. Για να σταλεί η στήλη δημιουργείται MPI Datatype. Η χρησιμότητα των Isend και Irecv ειναι η εξης: Ξεκινάει η αποστολή και η λήψη των απαραίτητων πληροφοριών απο τους γείτονες. Στο μεταξύ, ανεξαρτήτως του αν έχει ολοκληρωθεί η λήψη γίνονται οι υπολογισμοί των εσωτερικών κελίων, τα οποία γνωρίζουν ήδη τους γείτονες τους και δεν χρειάζονται πληροφορίες απο άλλους πίνακες. Στην συνέχεια, με την χρήση MPI_Wait, όταν μια λήψη έχει ολοκληρωθεί υπολογίζουμε τα άκρα που χρειάζονταν τις αντίστοιχες πληροφορίες.

Στο τέλος της διαδικασίας γίνεται έλεγχος για τις συνθήκες τερματισμού με μια συνάρτηση check η οποια λειτουργεί ως εξης: Δύο μεταβλητές, η same και η empty λειτουργούν ως έλεγχος. Σαρώνουμε τον πίνακα του κάθε process προσθέτοντας όλα τα στοιχεία στο empty. Αφού τα στοιχεία ειναι 0 ή 1, ενας “πεθαμένος” πίνακας εχει empty = 0. Ταυτόχρονα, γίνεται σύγκριση μεταξύ των στοιχείων του board (πίνακας πριν την αλλαγή) και next (πίνακας που εχει εφαρμοστεί η αλλαγή). Εάν κάποιο στοιχείο του next είναι διαφορετικό με το αντίστοιχο του board το same γινεται 0. Όλα τα αποτελέσματα των same και empty μαζεύονται με δυο MPI_Gather στο process με rank 0, το οποίο τα εξετάζει και καταλήγει σε ενα γενικό συμπέρασμα για τον πίνακα, το οποίο και στέλνει με MPI_Scatter σε ολα τα processes.

Όταν τελείωνει μια επανάληψη αυτής την διαδικασίας, και με προϋπόθεση οτι η διαδικασία θα συνεχίσει, γίνεται ανταλλαγή μεταξύ του next και του board. Όπως ειπώθηκε προηγουμένος, οι δυο πίνακες ειναι τύπου int**, επομένως η ανταλλαγή γίνεται εύκολα ανάμεσα τους με την χρήση ενός temporary δείκτη.

**OpenMP:** Το υβριδικό πρόγραμμα με χρήση MPI + OpenMP λειτουργεί ίδια με το MPI στην βασική του δομή. Τα threads του OpenMP χρησιμοποιούνται σε σημεία οπου διατρέχουμε μεγάλους πίνακες κελί-κελί, είτε για να κάνουμε αλλαγές  είτε για ελέγχους. Το μοναδικό σημείο που δεν υπάρχει χρήση thread είναι στον υπολογισμό της άνω γραμμής του πίνακα καθε process διότι παρουσίαζε πρόβλημα το οποίο δεν καταφέραμε να επιλύσουμε. Οι εκτελέσεις και οι μετρήσεις έχουν πραγματοποιηθεί μονο στα μηχανήματα της σχολής καθώς δεν καταφέραμε να έχουμε πρόσβαση σε άλλα κατάλληλα μηχανήματα που να είχαν το mpich.

**Cuda:** Αντιμετωπίσαμε μεγάλα προβλήματα με την εγκατάσταση του Cuda και τελικά δεν βρήκαμε κάποιον τρόπο να το κάνουμε να τρέχει στον ένα υπολογιστή, ενώ ο άλλος δεν είχε κάρτα γραφικών nvidia αρα δεν ήταν δυνατόν ούτως η άλλος. Ωστόσο, γράψαμε μια υλοποίηση η οποία θεωρητικά δουλεύει.
Το πρόγραμμα δέχεται κατα την εκτέλεση το μέγεθος του πίνακα (N*N) καθώς και των αριθμό γενεών. Οι πίνακες board και next περιέχουν ολόκληρο τον πίνακα του παιχνιδιού και αντι να υλοποιηθούν ως δισδιάστατοι είναι μονοδιάστατοι πίνακες μεγέθους N*N. Επομένως, το στοιχείο [N] ειναι το 1ο της 2ης γραμμής, το [2N-1] το τελευταίο της 2ης γραμμης και ούτω καθεξής. Θεωρούμε 256 threads per block και τρέχουμε το παραλληλοποιημένο κομμάτι του προγράμματος με (N+255)/256 blocks και 256 threads. Κάθε thread αναλαμβάνει ενα κελί του πίνακα, υπολογίζει τους γείτονες και εισάγει το αποτέλεσμα στον πίνακα των αποτελεσμάτων.

Υπολογισμός Επιτάχυνσης, Αποτελεσματικότητας μεταξύ MPI- ακολουθιακού προγράμματος:

| Processes | GoL Board Size | Speedup | Efficiency |
|---|---|---|---|
| 2 | 1200*1200 | 2.76042906488 | 1.38021453244 |
| 4 | 1200*1200 | 9.61424373204 | 2.40356093301 |
| 10 | 1200*1200 | 40.1539742115 | 4.01539742115 |
| 15 | 1200*1200 | 8.20544428913 | 0.54702961927 |
| 2 | 2400*2400 | 1.1272250751 | 0.56361253755 |
| 4 | 2400*2400 | 3.73590991906 | 0.93397747976 |
| 10 | 2400*2400 | 52.3487067883 | 5.23487067883 |
| 15 | 2400*2400 | 3.76493789004 | 0.25099585933 |
| 2 | 3600*3600 | 2.7787409878 | 1.3893704939 |
| 4 | 3600*3600 | 4.11955064528 | 1.02988766132 |
| 10 | 3600*3600 | 0.394823 | 60.7215815695 |
| 15 | 3600*3600 | 0.432989 | 55.3692518748 |

Μετρήσεις χρόνου:

Θεωρούμε ιδιο αριθμό γενεών gens = 50 και το τρέχουμε στα linux της σχολής

| | Processes | GoL Board size | Time |
|---|---|---|---|
| Serial | 1 | 1200*1200 | 2.653214 |
| Serial | 1 | 2400*2400 | 10.631970 |
| Serial | 1 | 3600*3600 | 23.974277 |
| MPI | 2 | 1200*1200 | 0.961160 |
| MPI | 4 | 1200*1200 | 0.275967 |
| MPI | 10 | 1200*1200 | 0.066076 |
| MPI | 15 | 1200*1200 | 0.323348 |
| MPI | 2 | 2400*2400 | 9.431985 |
| MPI | 4 | 2400*2400 | 2.845885 |
| MPI | 10 | 2400*2400 | 0.203099 |
| MPI | 15 | 2400*2400 | 2.823943 |
| MPI | 2 | 3600*3600 | 8.627748 |
| MPI | 4 | 3600*3600 | 5.819634 |
| MPI | 10 | 3600*3600 | 0.394823 |
| MPI | 15 | 3600*3600 | 0.432989 |
| OpenMP | 2 | 1200*1200 | 0.992286 |
| OpenMP | 4 | 1200*1200 | 0.288591 |
| OpenMP | 2 | 2400*2400 | 4.990230 |
| OpenMP | 4 | 2400*2400 | 1.939961 |
| OpenMP | 2 | 3600*3600 | 15.536709 |
| OpenMP | 4 | 3600*3600 | 2.244673 |

Συμπεράσματα:

Παρατηρήσαμε μεγάλη διαφορά μεταξύ του ακολουθιακού προγράμματος και της MPI υλοποίησης, καθώς υπήρξε μείωση του χρόνου εκτέλεσης εως και στο 1/60 του αρχικού. Μικρή ήταν η διαφορά μεταξύ Hybrid(MPI + OpenMp) και του MPI κυρίως λόγω της αδυναμίας μας να το εκτελέσουμε σε άλλο υπολογιστή εκτός απο αυτούς της σχολής. Επιπλέον παρατηρήσαμε αύξηση της επιτάχυνσης και της αποτελεσματικότητας μέχρι την χρήση 10 processes. Στις εκτελέσεις που περιλάμβαναν περισσότερα υπήρξε πτώση των παραπάνω όπως φαίνεται και στον 2ο πίνακα καθώς από εναν αριθμό και μετά δεν αξιοποιούντε πλέον οι διεργασίες.
Η αποτελεσματικότητα παραμένει σταθερη μόνο στην περίπτωση όπου μένουν ίδιες οι διεργασίες και αυξάνεται το μέγεθος του προβλήματος, επομένως το προγραμμα προσφερει μικρη δυνατότητα κλιμάκωσης. Λόγω ασυνεννοησίας τον αποτελεσμάτων μας ειναι δύσκολο να βγάλουμε σωστο συμπεράσμα για την κλιμάκωση όσον αφορά το OpenMP + MPI Hybrid πρόγραμμα