**ΕΘΝΙΚΟ ΚΑΙ ΚΑΠΟΔΙΣΤΡΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ**

**ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ**
**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ**

## Παράλληλα Συστήματα-Παράλληλα Υπολογιστικά Συστήματα

# Conway's Game of Life
**MPI, MPI/OPENMP, CUDA**

**Δρέττας Βασίλειος 1115201300042**

**Εργασία 2016-2017**

# ΠΕΡΙΕΧΟΜΕΝΑ


**1. Εισαγωγή....3**

1.1 Εκφώνηση....3

1.2 Οργάνωση Αρχείων....4

1.3 Μεταγγλώτιση και Εκτέλεση....5

**2. Σχεδιασμός Διαμοιρασμού Δεδομένων....6**

2.1 Διαμοιρασμός σε Block....6

2.2 Επικοινωνία....6

2.3 Τοπολογία Διεργασιών....8

**3. Σχεδιασμός και υλοποίηση MPI....9**

3.1 Επιλογές Επικοινωνίας....9

3.2 Χρήση Datatypes....10

3.3 Αποφυγή Αντιγραφών....11

3.4 Έλεγχος αλλαγής πλέγματος....11

3.5 Είσοδος από αρχείο....11

3.6 Υπολογισμός απόδοσης....12

**4. OPENMP/MPI Υλοποίηση....21**

4.1 MPI αλλαγές....21

4.2 Χρήση OPENMP....21

4.3 Υπολογισμός απόδοσης....22

4.4 Σύγκριση με απλό MPI....26

**5. CUDA....27**

5.1 Λεπτομέρειες υλοποίησης....27

5.2 Υπολογισμός απόδοσης....27

5.3 Σύγκριση με απλό MPI....29

**6. Συμπεράσματα....29**

# 1. ΕΙΣΑΓΩΓΗ

Η εργασία υλοποιήθηκε στα πλαίσια του μαθήματος των Παράλληλων συστημάτων και σκοπό έχει την εξοικείωση με τον παράλληλο προγραμματισμό σε CPU με την χρήση του MPI, του MPI σε συνδυασμό με το OPENMP και σε GPU με την χρήση του CUDA.

## 1.1 Εκφώνηση

Το Παιχνίδι της Ζωής αναπτύχθηκε το 1970 από τον John Conway στο Πανεπιστήμιο του Καίμπριτζ. Το παιχνίδι καταδεικνύει ότι μερικοί απλοί τοπικοί κανόνες μπορούν να οδηγήσουν σε ενδιαφέρουσα μεγάλης κλίμακας συμπεριφορά. Παίζεται σε ένα περιοδικό δισδιάστατο πλέγμα (NxM) από κελιά ή κύτταρα, τα οποία μπορούν να βρίσκονται σε μια από δυο καταστάσεις: ζωντανά ή νεκρά. Το παιχνίδι δεν έχει παίκτες, δηλαδή δεν απαιτεί εισαγωγή δεδομένων κατά την εξέλιξή του, αλλά αυτή εξαρτάται αποκλειστικά από το αρχικό σχέδιο του πλέγματος Κάθε κελί θεωρείται πως έχει οκτώ γείτονες, οι οποίοι επηρεάζουν την κατάστασή του. Το σύνολο των ζωντανών κελιών αποτελεί τον πληθυσμό του πλέγματος. Η εξέλιξη γίνεται σε διακριτά βήματα, τις γενεές και η ανανέωση του πλέγματος από γενεά σε γενεά γίνεται ταυτόχρονα. Η κατάσταση κάθε κελιού στην επόμενη γενιά εξαρτάται αποκλειστικά από την κατάσταση του ιδίου και των οκτώ γειτόνων του στη παρούσα γενιά, βάση ορισμένων κανόνων. Το παιχνίδι αρχίζει είτε με μια τυχαία επιλογή των κατειλημμένων θέσεων είτε με ένα σχέδιο που διαβάζεται από ένα αρχείο. Από αυτήν την αρχική γενεά, η επόμενη γενεά υπολογίζεται χρησιμοποιώντας τους ακόλουθους κανόνες:

1. Εάν ένας οργανισμός (κατειλημμένη θέση) έχει 0 ή 1 γειτονικούς οργανισμούς, ο οργανισμός πεθαίνει από μοναξιά.

2. Εάν ένας οργανισμός έχει 2 ή 3 γειτονικούς οργανισμούς, ο οργανισμός επιζεί στην επόμενη γενεά.

3. Εάν ένας οργανισμός έχει 4 έως 8 γειτονικούς οργανισμούς, ο οργανισμός πεθαίνει λόγω υπερπληθυσμού.

4. Εάν μία μη κατειλημμένη θέση έχει ακριβώς 3 γειτονικούς οργανισμούς, αυτή η θέση θα καταληφθεί στην επόμενη γενεά από έναν νέο οργανισμό, δηλαδή ένας οργανισμός γεννιέται

Το πρόγραμμά σταματάει εάν δεν υπάρχει κανένας οργανισμός στο πλέγμα. Επίσης σταματάει αν το πλέγμα παραμένει αμετάβλητο για δύο συνεχόμενες γενεές.

## 1.2 Οργάνωση αρχείων

Η εργασία είναι χωρισμένη σε 5 φακέλους

**timers:** Περιέχει το timer.h είναι για την μέτρηση της γραμμικής(ακολουθιακής) υλοποίησης και της υλοποίησης του CUDA όπως δόθηκε από το βιβλίο του pacheco. Επιστρέφει το χρόνο σε second.

**MPI: Περιέχει** τα αρχεία της υλοποίησης mpi.

Τα mpiFunctions έχουν συναρτήσεις που κάνουν κλήσεις του MPI για την αποστολή και λήψη μηνυμάτων, για το setup της καρτεσιανής τοπολογίας, την είσοδο των δεδομένων από αρχείο και την εύρεση των γειτονικών διεργασιών.

Τα gameOfLife έχουν συναρτήσεις για την εύρεση των γειτόνων, την αρχικοποίηση του πίνακα άμα δεν δοθεί από αρχείο, την δυναμική δέσμευση του πίνακα κ.α.

Το αρχείο main.c εχει την main για το game of life με mpi.

Το mainCreateFile έχει την main για την δημιουργία ενός αρχείου εισόδου τυχαία.

**OPENMP**: Περιέχει τα αρχεία της υλοποίησης με mpi και openmp. Οι αλλαγές έχουν γίνει στο αρχείο της main με την προσθήκη των εντολών προς το openmp. Υπάρχουν δυο υλοποιήσεις

1. Η openmpParrallelCommunication όπου η επικοινωνία γίνεται από όλα τα threads
2. Η openmpMasterThreadCommunication όπου η επικοινωνία γίνεται μόνο από το master thread.

**CUDA:** Περιέχει τα αρχεία της υλοποίησης με cuda. Στο αρχείο gameOfLifeGPU βρίσκονται οι κλήσεις cuda για την αντιγραφή της CPU μνήμης και η κλήση στο device kernel για τον υπολογισμό της τιμής κάθε κελιού.

**Linear:** Περιέχει την ακολουθιακή υλοποίηση και χρησιμοποιείται για τον υπολογισμό του σειριακού χρόνου.

**Σημείωση:**Τα αρχεία gameOfLife.c gameOfLife.h βρίσκονται και στους τέσσερις φακέλους υλοποίησης(εκτός από τον timers) και είναι ίδια. Τα έβαλα σε κάθε φάκελο για λόγους ευκολίας και να μην ψάχνετε σε ποιο φάκελο βρίσκεται ο κώδικας κλήσης κάποιας συνάρτησης.

### 1.3 Μεταγλώττιση και Εκτέλεση

Για την μεταγλώττιση έχει υλοποιηθεί makefile. Επομένως άπλα πληκτρολογείτε την εντολή make στον εκάστοτε φάκελο(MPI, OPENMP, CUDA, Linear).

Για την εκτέλεση πληκτρολογείτε στον αντίστοιχο φάκελο

**MPI:**

mpiexec -n <NumberOfTasks> ./gameoflife -r <number of rows> -c <number of columns> -f <inputFileName(optional)>

Παράδειγμα mpiexec -n 4 ./gameoflife -r 1000 -c 1000

Μετά το -r ακολουθεί το πλήθος των γραμμών του πίνακα, μετά το -c ακολουθεί το πλήθος των στηλών του πίνακα.

Επίσης μπορεί να γίνει είσοδος του αρχικού terrain από αρχείο. Άμα δεν δοθεί τότε ο αρχικός πίνακας είναι τυχαίος

Για την δημιουργία τυχαίου αρχείου μπορείτε να τρέξετε

./createRandomFile -r <number of rows> -c <number of columns> -f <outFileName>

**OpenMP/MPI:**

mpiexec -n <NumberOfTasks> ./gameoflife -r <number of rows> -c <number of columns> -f <inputFileName(optional)>

**Cuda:**

./gameoflife -r <number of rows> -c <number of columns>

**Linear:**

./gameoflife -r <number of rows> -c <number of columns>

# 2. Σχεδιασμός Διαμοιρασμού Δεδομένων

## 2.1 Διαμοιρασμός σε Block

Ο διαμοιρασμός του αρχικού NxM πίνακα στις διεργασίες γίνεται ανά blocks αντί για σειρές καθώς έτσι επιτυγχάνονται καλύτερα αποτελέσματά. Για παράδειγμα άμα έχουμε ένα πίνακα 1000x1000 και έχουμε 4 διεργασίες τότε κάθε διεργασία θα πάρει ένα πίνακα 500x500. Δηλαδή η πρώτη από [0-499][0-499], η δεύτερη [0-499][500-999], η τρίτη [500-999][0-499], και η τέταρτη [500-999][500-999].

Στο MPI ο πίνακας που διατηρεί κάθε διεργασία έχει 2 επιπλέον γραμμές και στήλες. Αυτός ο επιπλέον χώρος χρησιμοποιείται για την ευκολότερη επικοινωνία των διεργασιών, έτσι ώστε τα δεδομένα που λαμβάνονται από τις 8 γειτονικές διεργασίες να τοποθετούνται στον πίνακα.

Σημείωση: Δεν απαιτούμε ο πίνακας να είναι τετραγωνικός. Θα μπορούσε να είναι και ο 2000x1000, αρκεί να μπορεί να χωριστεί σωστά στις δοθείσες διεργασίες.

## 2.2 Επικοινωνία

Κάθε διεργασία προκειμένου να υπολογίζει τα όρια του πίνακα χρειάζεται να επικοινωνήσει με τις γειτονικές διεργασίες της. Κάθε διεργασία έχει 8 γείτονες, έναν πάνω και ζητά την κάτω γραμμή , ένα κάτω και ζητά την πάνω γραμμή, ένα δεξιά και ζητά την αριστερή στήλη, ένα αριστερά και ζητά την δεξιά στήλη. Τέλος για να υπολογιστούν οι γωνίες η διεργασία επικοινωνεί με τις διαγώνιες. Δηλαδή με την πάνω αριστερά διεργασία και ζητά το κάτω δεξιά στοιχείο, την πάνω δεξιά διεργασία και ζητά το κάτω αριστερά στοιχείο, την κάτω αριστερά διεργασία και ζητά το πάνω δεξιά στοιχείο, την κάτω δεξιά διεργασία και ζητά το πάνω αριστερά στοιχείο .

Έστω ότι έχουμε τις διεργασίες με τα ακολουθά νούμερα που έχουν χωρίσει τον πίνακα σε blocks όπως περιγράψαμε προηγουμένως.

| | | |
|---|---|---|
| 1 | 2 | 3 |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

σχήμα 1

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |

σχήμα 2

Τότε στο σχήμα 1, η διεργασία 5 έχει τους εξής γείτονές.

- Πάνω αριστερά την 1
- Πάνω την 2
- Πάνω δεξιά την 3
- Αριστερά στην 4
- Δεξιά την 6
- Κάτω αριστερά την 7
- Κάτω την 8
- Κάτω δεξιά την 9

Για το σχήμα 2 η διεργασία 1 για να υπολογίσει την γραμμή 0 (εκτός από τις γωνίες) επικοινωνεί με την διεργασία που έχει την γραμμή 999 που είναι η 3. Για να υπολογίσει την γραμμή 499 επικοινωνεί πάλι με την 3 και ζητά την 500. Για την στήλη 0 επικοινωνεί με την αριστερή διεργασία που είναι η 2 και ζητά την 999 στήλη. Επίσης επικοινωνεί με την διεργασία δεξιά δηλαδή την 2 και ζητά την 500 στήλη. Τέλος για να υπολογιστούν οι γωνίες η διεργασία επικοινωνεί με τις διαγώνιες. Δηλαδή με την διεργασία 4 και ζητά το [500][500], [500][999], [999][500], [999][999].

δηλαδή είναι σαν να προεκτείνουμε τον πίνακα έτσι

1 2 1 2
3 4 3 4

1 2 1 2
3 4 3 4

Και μετά μπορούμε εύκολα να δούμε τους γείτονες του 1.

Στην περίπτωση του σχήματος 2 είδαμε ότι η μια διεργασία μπορεί να ζητά από διαφορετικά στοιχειά(πχ από την 4 ζητά τα γωνιακά στοιχειά της). Για να μπορεί να πάρει και να τοποθετήσει το στοιχείο που έλαβε στην σωστή θέση χρησιμοποιείται το πεδίο tag για να μπορούν να διαχωριστούν τα δεδομένα που ελήφθησαν ποια θέση του πίνακα αφορούν. Για παράδειγμα όταν η διεργασία 1 θέλει το πάνω δεξιά στοιχείο, στέλνει στην διεργασία 4 αίτημα με tag LEFTDOWN.

Έχει υλοποιηθεί Enumeration για να είναι πιο ευανάγνωστος ο κώδικας.

enum Positions

{LEFTUP, UP, RIGHTUP, LEFT, RIGHT, LEFTDOWN, DOWN, RIGHTDOWN}

Αξίζει να σημειωθεί ότι στην περίπτωση των γραμμών και στηλών στέλνουμε όλα τα δεδομένα με ένα request καθώς είναι πολύ πιο γρήγορο από το να κάνουμε πολλά αιτήματα.

## 2.3 Τοπολογια Διεργασιεων

Στην υλοποίηση εχει χρησιμοποιηθεί η καρτεσιανή τοπολογία. Το setup του νέου communicator γίνεται στην συνάρτηση

void createCartesianTopology(MPI_Comm* gridComm, int axisSize)

Εκεί ορίζουμε οτι θα έχουμε ενα τετραγωνικό grid δηλαδή ο άξονας χ θα είναι ίσος με τον άξονα y. Έτσι το πλήθος των διεργασιών πρέπει να είναι 1, 4, 9, 16, 25 …

Αυτό γίνεται για λόγους ευκολίας στους υπολογισμούς.

Επίσης ορίζουμε ότι θα είναι περιοδικός και στις δυο διαστάσεις, καθώς ο πινάκας μας είναι περιοδικός και θέλουμε οι διεργασίες που βρίσκονται στα άκρα να βρίσκουν τις γειτονικές τους. Για παράδειγμα στο σχήμα 1 η διεργασία 6 θέλουμε να γνωρίζει ότι δεξιά της έχει την διεργασία 4 και είναι γειτονικές.

Τους γείτονες τους βρίσκουμε στην αρχή με την συνάρτηση

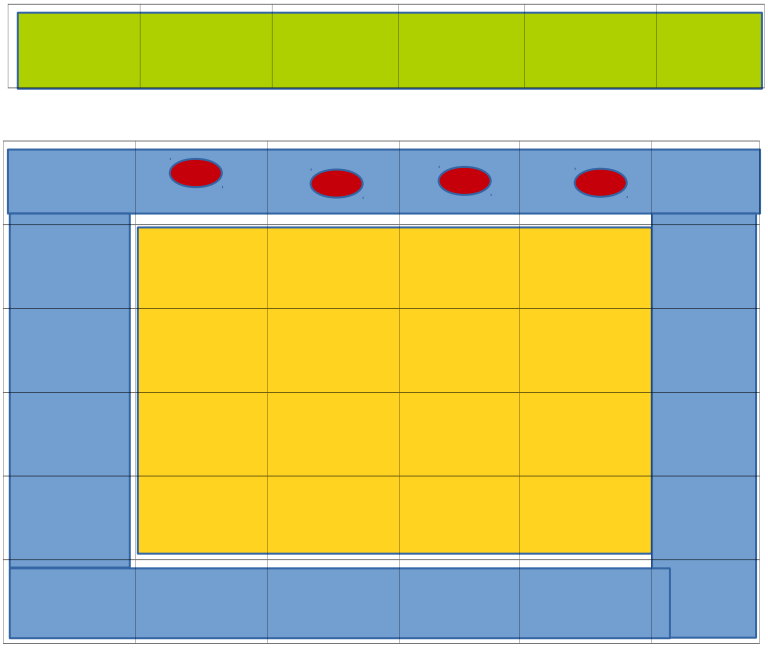
void neighborProcess(MPI_Comm communicator,int processRank, int* neighbors)

και αποθηκεύουμε το id τους στον πίνακα neighbors. Στην συνέχεια χρησιμοποιούμε τον πίνακα για να στείλουμε και να λάβουμε τα δεδομένα προς/από τους γείτονες.

# 3. Σχεδιασμός και υλοποίηση MPI

## 3.1 Επιλογές Επικοινωνίας

Προκείμενου να μπορούμε να έχουμε επικάλυψη της επικοινωνίας δεν θέλουμε να χρησιμοποιήσουμε της άπλες εντολές send και recv καθώς οι κλήσεις αυτές είναι blocking. Έτσι αρχικά επιλέξαμε τις non blocking μορφές τους isend και irecv. Έτσι μπορούμε να κάνουμε αυτές τις κλήσεις και στην συνεχεία να υπολογίζουμε τον εσωτερικό πίνακα, δηλαδή όλα τα στοιχειά εκτός από αυτά που βρίσκονται στα όρια του πίνακα. Δηλαδή στον παρακάτω πίνακα θα υπολογίσουμε το κίτρινο κομμάτι.



Για τον εσωτερικό πίνακα έχουμε όλα τα δεδομένα που χρειαζόμαστε ενώ για τα στοιχειά που βρίσκονται στα όρια χρειάζεται να λάβουμε δεδομένα από τις γειτονικές διεργασίες.

Αφού υπολογίσουμε τα κίτρινα περιμένουμε να λάβουμε τα δεδομένα από τις γειτονικές διεργασίες. Για να έχουμε ακόμα μεγαλύτερη επικάλυψη χρησιμοποιούμε την MPI_Waitany όπου περιμένει οποιοδήποτε recv request. Έτσι αν λάβουμε για παράδειγμα την πάνω γραμμή(την πράσινη στο σχήμα) απο τον πάνω γείτονα θα μπορούμε να υπολογίσουμε τα πάνω τέσσερα κελιά με τις κόκκινες κουκκίδες. Αντίστοιχα και για τα υπόλοιπα. Τον υπολογισμό των γωνιών τον κάνουμε αφού λάβουμε τις δυο γραμμές και τις δυο στήλες για να έχουμε όλες τις πληροφορίες που χρειαζόμαστε.

Τέλος για να έχουμε ακόμη μεγαλύτερη βελτίωση χρησιμοποιήσαμε τις συναρτήσεις

MPI_Send_init

MPI_Recv_init

MPI_Start

Αντί για τις isend και irecv. Αυτό το κάνουμε για να αποφύγουμε τον επανυπολογισμό του overhead για κάθε γύρο του παιχνιδιού. Ωστόσο τώρα θα χρειαστούμε 16 MPI_Request για send και 16 MPI_Request για receive. Αυτό συμβαίνει καθώς διατηρούμε δυο πίνακες. Έτσι χρειαζόμαστε τα 8 send και recv request για όταν χρησιμοποιείται ο πρώτος ως δεδομένα και τα αποτελέσματα γράφονται στον δυο, και άλλα 8 send και recv request για όταν χρησιμοποιείται ο δεύτερος ως δεδομένα και τα αποτελέσματα γράφονται στον πρώτο. Θα μπορούσαμε να έχουμε πάντα τον ίδιο για δεδομένα και τον ίδιο που θα γράφουμε τα αποτελέσματα αλλά τότε θα χρειαζόταν να αντιγράψουμε όλα τα δεδομένα του ενός πίνακα στον άλλο πράγμα που είναι απαγορευτικό.

## 3.2 Χρήση Datatypes

Για την επικοινωνία μεταξύ των διεργασιών υλοποιήθηκαν δυο Datatypes ROW και COLUMN.

Το datatype ROW είναι μια γραμμή του πίνακα, δηλαδή διατηρεί Μ συνεχόμενα στοιχειά(Μ πλήθος στηλών). Έτσι όταν μια διεργασία ζητά την πάνω ή κάτω γραμμή χρησιμοποιείται αυτό το Datatype.

Το datatype COLUMN χρησιμοποιείται για την στήλη του πίνακα, δηλαδή διατηρεί Ν στοιχεία(Ν πλήθος γραμμών) όπου το καθένα έχει απόσταση από το προηγούμενο του κατά Μ. Έτσι όταν μια διεργασία ζητά την στήλη από κάποια άλλη χρησιμοποιείται από το datatype. Με αυτό το τρόπο αποφεύγουμε μια επιπλέον αντιγραφή των στοιχείων σε ένα buffer έτσι ώστε να τα στείλουμε μετά.

### 3.3 Αποφυγή Αντιγραφών

Πέρα από την αποφυγή της αντιγραφής των δεδομένων σε buffer όταν θέλουμε να στείλουμε μια στήλη αποφεύγουμε και την αντιγραφή των στοιχείων από τον ένα πίνακα στον άλλο.

Οι δυο πίνακες δεσμεύονται δυναμικά και όλα τα στοιχειά τους βρίσκονται σε συνεχόμενες θέσεις μνήμης έτσι ώστε να είναι εφικτό να φτιαχτεί ένα COLUMN datatype.

Έτσι έχουμε τους πίνακες prevBlock και currentBlock. Αφού υπολογίσουμε τις νέες τιμές του πίνακα με βάση τον prevBlock και τις αποθηκεύσουμε στον currentBlock απλά κάνουμε swap τον pointer έτσι ώστε ο prevBlock να δείχνει όπου έδειχνε το currentBlock (και αρα να έχει τις πιο πρόσφατες τιμές) και τον currentBlock εκεί που έδειχνε ο prevBlock (δηλαδή στα παλιά δεδομένα). Έτσι συνεχίζουμε κανονικά τους υπολογισμούς του επόμενου γύρου χωρίς να αντιγράψουμε όλα τα δεδομένα από τον ένα πίνακα στον άλλο.

### 3.4 Έλεγχος αλλαγής πλέγματος

Κάθε 5 γύρους γίνεται έλεγχος αν το πλέγμα έχει παραμείνει σταθερό. Μέτα με την χρήση της MPI_Allreduce με την πράξη OR ελέγχεται αν έστω και ένα πλέγμα έχει αλλάξει. Αμα έχει αλλάξει κάποια συνεχίζουν όλες οι διεργασίες για άλλους 5 γύρους μέχρι να ξαναγίνει ο έλεγχος. Σε περίπτωση που όλες παρέμειναν σταθερές τότε σταματάνε την λειτουργιά τους.

Ο έλεγχος δεν είναι ενεργοποιημένος.

Ο κώδικας αυτού του ελέγχου βρίσκεται σε **σχόλιο** στην γραμμή 194-205 του αρχείου main.c στον φάκελο mpi. Αν θέλετε να ενεργοποιήσετε άπλα αφαιρέστε το σχόλιο.

### 3.5 Είσοδος από αρχείο

Για την είσοδο από binary αρχειο έχει υλοποιηθεί η συνάρτηση

void readFromFile(char* fileName, MPI_Comm communicator, int processRank, int** block, int nRows, int nColumns, int totalRows, int totalColumns)

Στην συνάρτηση αυτή έχει χρησιμοποιηθεί το MPI_Type_create_subarray για να διαβάσει ένα block δεδομένων κάθε διεργασία και collective read με την χρήση της συνάρτησης MPI_File_read_all

Επίσης υλοποιήθηκε και η συνάρτηση

void createFile(char* fileName, MPI_Comm communicator, int processRank, int** block, int nRows, int nColumns)

Που αφού γίνει initialiaze ο πίνακας block μπορεί να χρησιμοποιηθεί για να εγγραφή ένα αρχείο με όνομα fileName με τυχαία δεδομένα.

## 3.6. Υπολογισμός Απόδοσης

### 3.6.1 Μετρήσεις Χρόνου Εκτέλεσης

Όλες οι μετρήσεις έγιναν με 100 επαναλήψεις δηλαδή 100 γύρους παιχνιδιού. Επίσης οι μετρήσεις είναι σε δευτερόλεπτα(sec).

Οι μετρήσεις έγιναν στα μηχανήματα της σχολής. Σε κάθε μηχάνημα δίνονταν 2 διεργασίας και ήταν διαθέσιμα 14 μηχανήματα.

Στις γραμμές φαίνονται οι διαστάσεις του αρχικού πίνακα και στις στήλες το πλήθος των Process.

Χωρίς allreduce (έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 0.100794 | 0.070141 | 0.070905 | - |
| 720x720 | 1.504735 | 0.690599 | 0.417985 | - |
| 800x800 | 1.784570 | - | 0.502802 | 0.33712 |
| 1008x1008 | 2.897363 | 1.348336 | 0.779038 | - |
| 1440x1440 | 5.870196 | 2.673587 | 1.540778 | - |
| 1600x1600 | 7.148357 | - | 1.870464 | 1.206793 |
| 1728x1728 | 8.387243 | 3.822500 | 2.188497 | - |
| 2880x2880 | 23.013436 | 10.406107 | 5.908510 | - |

Με allreduce ανά 5 loop( έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 0.102238 | 0.084639 | 0.066455 | - |
| 720x720 | 1.513625 | 0.702143 | 0.426673 | - |
| 800x800 | 1.821961 | - | 0.503302 | 0.344061 |
| 1008x1008 | 2.897776 | 1.361862 | 0.783542 | - |
| 1440x1440 | 5.860196 | 2.681197 | 1.532482 | - |
| 1600x1600 | 7.159163 | - | 1.977332 | 1.215407 |
| 1728x1728 | 8.397117 | 3.91671 | 2.19975 | - |
| 2880x2880 | 23.26343 | 10.70160 | 6.20150 | - |

Προκείμενου να κάνουμε πιο εμφανής την καθυστέρηση που προκαλεί η allreduce αλλάξαμε την συνθήκη και κάνουμε έλεγχο ανά μια επανάληψη.

Με allreduce ανά 1 loop( έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 0.115071 | 0.079736 | 0.093048 | - |
| 720x720 | 1.661560 | 0.760655 | 0.574618 | - |
| 800x800 | 2.002432 | - | 0.580469 | 0.709569 |
| 1008x1008 | 3.108752 | 1.522692 | 0.876465 | - |
| 1440x1440 | 6.135598 | 2.899502 | 1.621463 | - |
| 1600x1600 | 7.533361 | - | 2.097110 | 1.379990 |
| 1728x1728 | 8.787898 | 4.110186 | 2.430924 | - |
| 2880x2880 | 24.226376 | 11.116334 | 6.420953 | - |

Ενώ στην υλοποίηση όπου ο έλεγχος γινόταν ανά 5 loop οι χρόνοι ήταν αρκετά κοντά με την υλοποίηση που δεν είχε έλεγχο πλέγματος όταν αλλάζουμε την συνθήκη και την κάνουμε να κάνει έλεγχο σε κάθε επανάληψη η καθυστέρηση είναι εμφανής. Σε όλες τις μετρήσεις μας παρατηρούμε μια επιβράδυνση.

## Linear

Χρειαζόμαστε την ακολουθιακή υλοποίηση προκείμενου να υπολογίσουμε ποια θα είναι η επιτάχυνση της παράλληλης σε σύγκριση με την ακολουθιακή.

Με την ακολουθιακή υλοποίηση στο μηχάνημα της σχολής linux01 έχουμε τα ακόλουθα αποτελέσματα.

| Matrix Size | Time |
|---|---|
| 144x144 | 0.208812 |
| 720x720 | 5.245523 |
| 800x800 | 6.494685 |
| 1008x1008 | 10.240248 |
| 1440x1440 | 20.897311 |
| 1600x1600 | 25.760992 |
| 1728x1728 | 30.029273 |
| 2880x2880 | 83.226259 |

### 3.6.2 Επιτάχυνση(speedup)

Για να υπολογίσουμε την επιτάχυνση χρησιμοποιούμε τον τύπο S = Tserial/Tparallel.

Στους παρακάτω πίνακες βλέπουμε την επιτάχυνση ανάλογα με το μέγεθος του πίνακα και το πλήθος των διεργασιών.

Επομένως για το MPI θα ελέγξουμε τους χρόνους που βρήκαμε με τους χρόνους που κάνει η ακολουθιακή υλοποίηση στο μηχάνημα της σχολής linux01.

Όπως είδαμε από τους χρόνους με allreduce είναι αρκετά κοντά με την απλή υλοποίηση άρα και η επιτάχυνση θα βρίσκεται πολύ κοντά. Για να έχουμε αισθητή μείωση θα δούμε την επιτάχυνση όταν ο έλεγχος γίνεται ανά μια επανάληψη.

Άρα έχουμε:

Χωρίς allreduce (έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 1.99 | 2.98 | 2.94 | - |
| 720x720 | 3.49 | 7.6 | 12.55 | - |
| 800x800 | 3.64 | - | 12.92 | 19.27 |
| 1008x1008 | 3.53 | 7.59 | 13.14 | - |
| 1440x1440 | 3.56 | 7.82 | 13.57 | - |
| 1600x1600 | 3.6 | - | 13.77 | 21.35 |
| 1728x1728 | 3.58 | 7.86 | 13.72 | - |
| 2880x2880 | 3.61 | 8 | 14.08 | - |

Με allreduce ανά 1 loop( έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 1.81 | 2.62 | 2.24 | - |
| 720x720 | 3.16 | 6.9 | 9.3 | - |
| 800x800 | 3.24 | - | 11.2 | 9.15 |
| 1008x1008 | 3.29 | 6.73 | 11.69 | - |
| 1440x1440 | 3.4 | 7.2 | 12.89 | - |
| 1600x1600 | 3.42 | - | 12.28 | 18.67 |
| 1728x1728 | 3.42 | 7.3 | 12.35 | - |
| 2880x2880 | 3.43 | 7.49 | 12.96 | - |

### 3.6.3 Αποδοτικότητα(efficiency)

Για τον υπολογισμό την αποτελεσματικότητας έχουμε τον τύπο E= S/p, όπου S είναι η επιτάχυνση και p το πλήθος των πυρήνων.

Χωρίς allreduce (έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 0.498 | 0.33 | 0.18 | - |
| 720x720 | 0.87 | 0.84 | 0.78 | - |
| 800x800 | 0.91 | - | 0.80 | 0.77 |
| 1008x1008 | 0.88 | 0.84 | 0.82 | - |
| 1440x1440 | 0.89 | 0.87 | 0.85 | - |
| 1600x1600 | 0.9 | - | 0.86 | 0.85 |
| 1728x1728 | 0.9 | 0.87 | 0.86 | - |
| 2880x2880 | 0.9 | 0.89 | 0.88 | - |

Με allreduce ανά 1 loop( έλεγχο αν παραμένει ίδιο το πλέγμα)

| | | **Process** | | |
|---|---|---|---|---|
| **Matrix Size** | 4 | 9 | 16 | 25 |
| 144x144 | 0.45 | 0.29 | 0.14 | - |
| 720x720 | 0.79 | 0.77 | 0.58 | - |
| 800x800 | 0.81 | - | 0.7 | 0.37 |
| 1008x1008 | 0.82 | 0.75 | 0.73 | - |
| 1440x1440 | 0.85 | 0.8 | 0.81 | - |
| 1600x1600 | 0.86 | - | 0.77 | 0.75 |
| 1728x1728 | 0.86 | 0.81 | 0.77 | - |
| 2880x2880 | 0.86 | 0.83 | 0.81 | - |

### 3.6.4 Κλιμάκωση(scalability)

Από τις παραπάνω μετρήσεις για την υλοποίηση χωρίς την χρήση allreduce παρατηρούμε ότι για μικρο πίνακα δηλαδή για τον πίνακα 144x144 η κλιμάκωση είναι κακή καθώς η αποτελεσματικότητα των 3 μετρήσεων μας είναι [4,9,16]=[0.498, 0.33, 0.18]. Δηλαδή παρατηρούμε ότι η αποδοτικότητα μειώνεται αισθητά καθώς αυξάνουμε το πλήθος των διεργασιών και διατηρούμε σταθερό το μέγεθος του προβλήματος. Ωστόσο για μεγάλους πίνακες όπως 1600x1600 οπου έχουμε την μετρήσεις [4,16,25] = [0.9,0.86, 0.85] παρατηρούμε οτι η αποδοτικότητα μειώνεται ελάχιστα όσο αυξάνουμε το πλήθος των διεργασιών. Ακόμα πιο εμφανές παράδειγμα είναι για τον πίνακα 2880x2880 όπου είχαμε τις μετρήσεις [4,9,16]=[0.9,0.89,0.88] δηλαδή η αποδοτικότητα παραμένει σχεδόν σταθερή πράγμα που σημαίνει ότι το πρόβλημα εμφανίζει καλή κλιμάκωση. Επομένως για μεγάλο πλήθος δεδομένων που είναι και το κύριο ζητούμενο του προβλήματος το πρόβλημα εμφανίζει καλή κλιμάκωση.

Με την χρήση της allreduce παρατηρούμε οτι έχουμε λίγο χειρότερη κλιμάκωση για παράδειγμα για πίνακα 2880x2880 έχουμε τις μετρήσεις [4,9,16]=[0.86,0.83, 0.81] που παρουσιάζουν μεγαλύτερη αστάθεια σε σύγκριση με τις προηγούμενες μετρήσεις μας. Αυτό το περιμέναμε καθώς όσο αυξάνουμε το πλήθος διεργασιών η allreduce απαιτεί μεγάλη επικοινωνία μεταξύ τους, καθώς σε κάθε γύρω πρέπει να επικοινωνούν όλες μεταξύ τους προκείμενου να διαπιστωθεί αν θα συνεχιστεί το παιχνίδι ή θα σταματήσει.

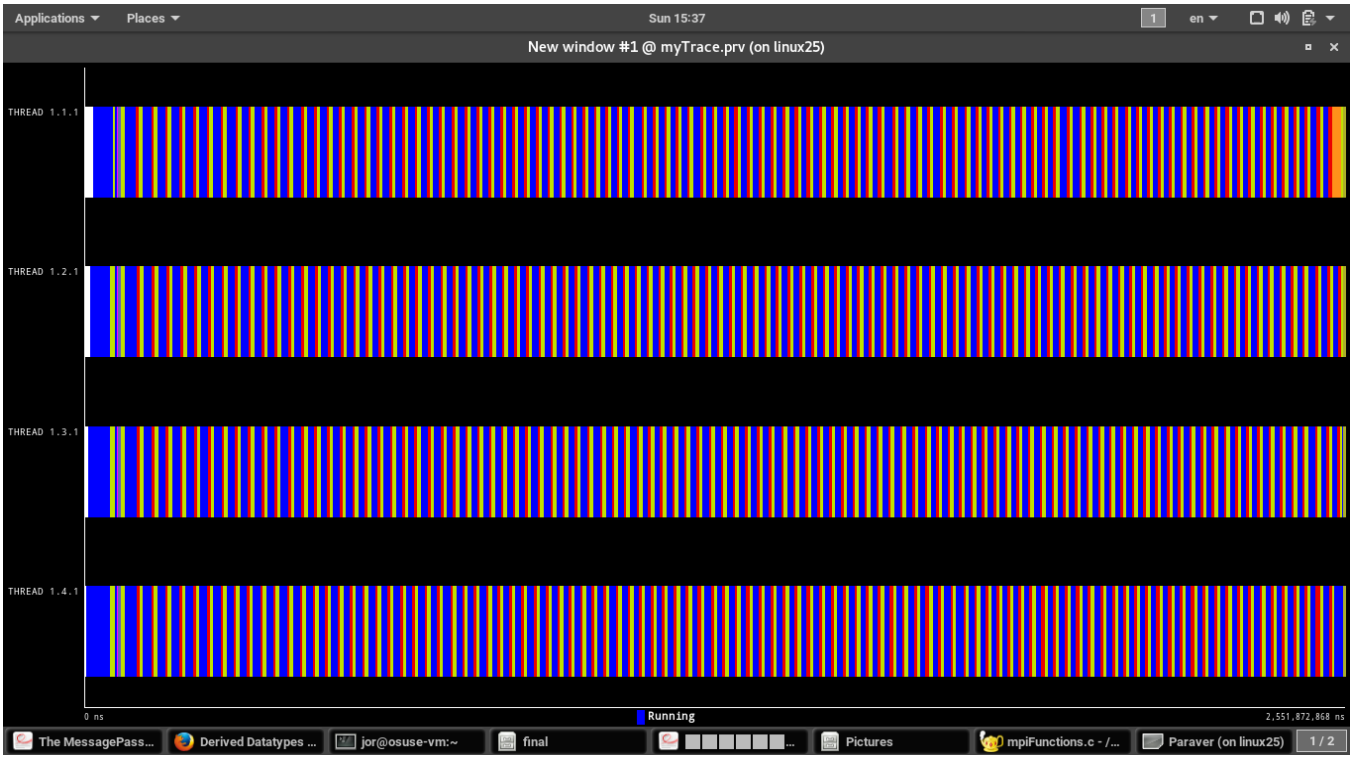
### 3.6.5 Συμπεράσματα

Γενικά παρατηρούμε ότι όσο αυξάνουμε το μέγεθος του προβλήματος οι μετρήσεις μας βελτιώνονται. Όπως είδαμε για μεγάλους πίνακες η αποδοτικότητα φτάνει ακόμα και το 0.9 που είναι αρκετά κοντά στο ιδανικό 1. Όταν χρησιμοποιούμε την allreduce ανά 1 loop τότε η καλύτερη αποδοτικότητα που βρήκαμε είναι 0.86 πράγμα που σημαίνει οτι έχουμε περισσότερες καθυστερήσεις.

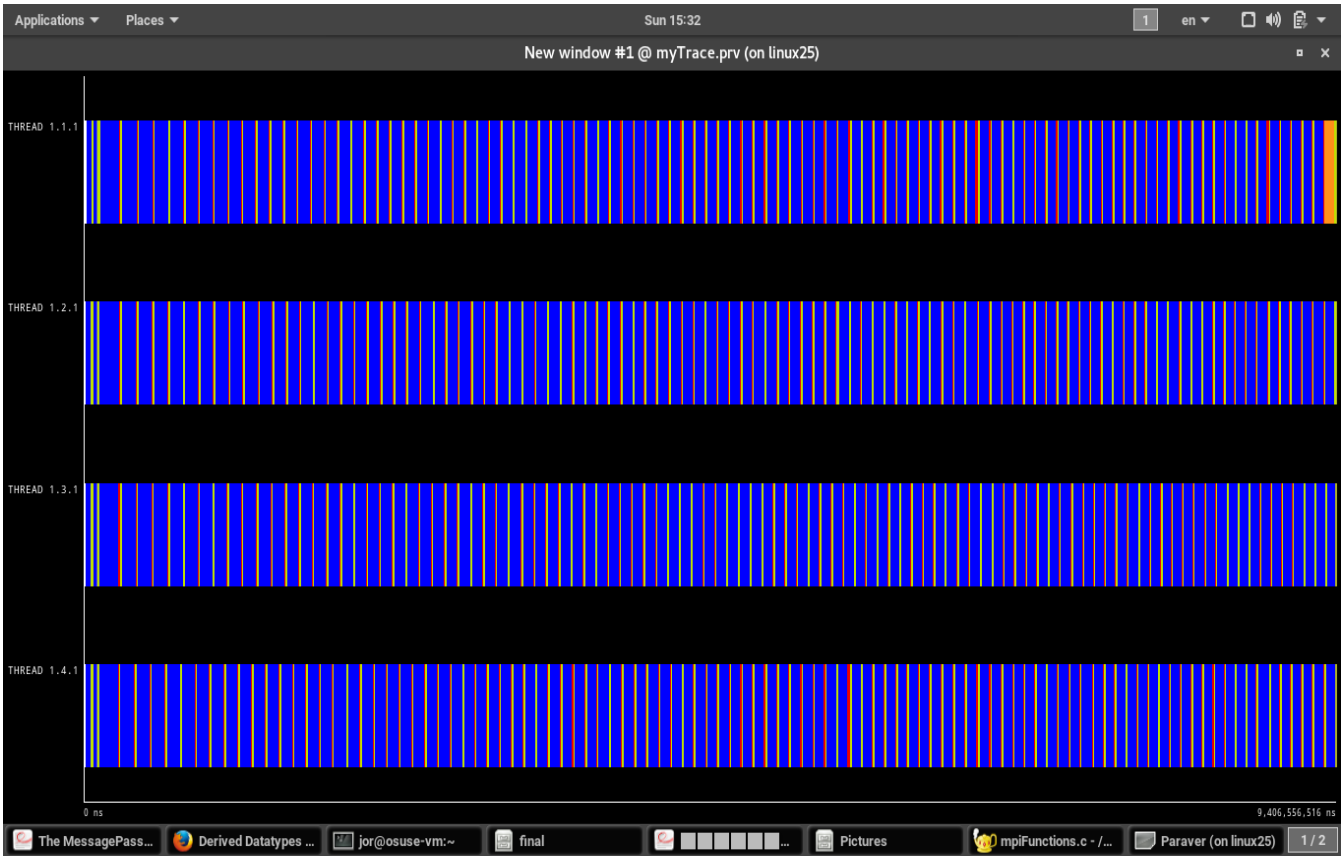
### 3.6.6 Paraver

Δοκιμάσαμε το paraver μόνο για 4 process, όταν είχαμε περισσότερα process αντιμετωπίζαμε προβλήματα και δεν παίρναμε αποτελέσματα.

Για πίνακα 720x720 έχουμε τα ακόλουθο σχήμα

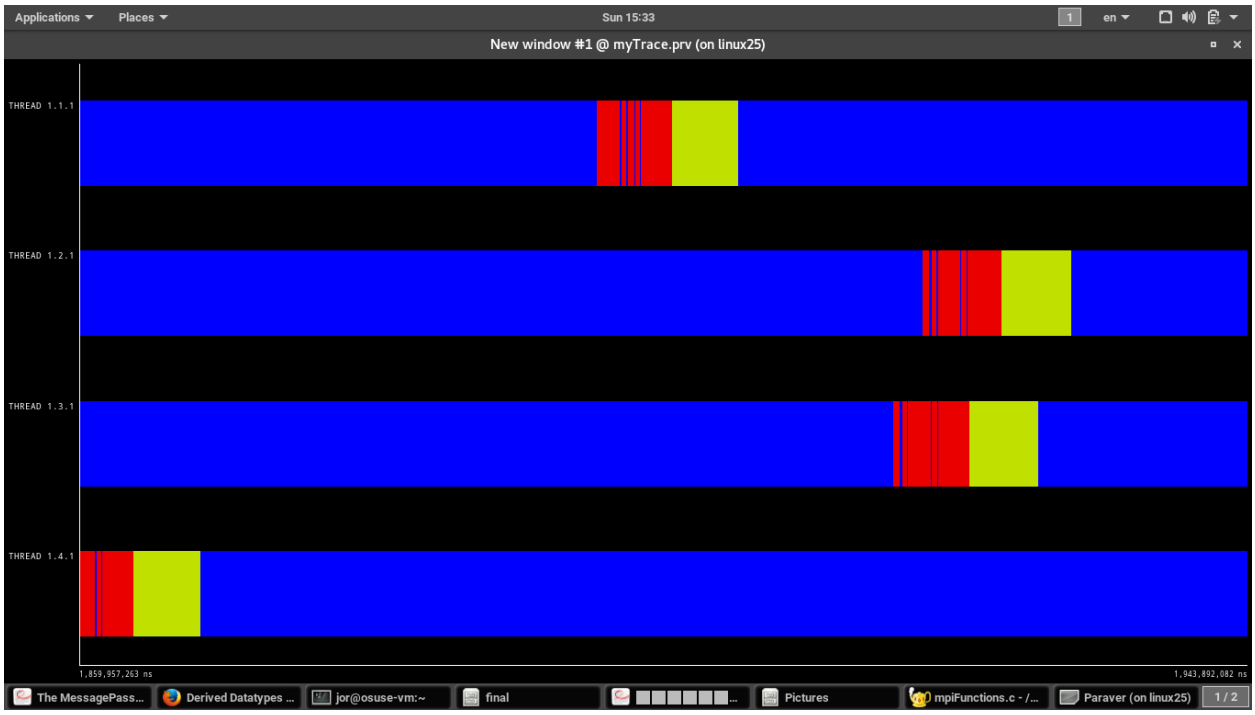



Για πίνακα 1728x1728 έχουμε το ακόλουθο σχήμα



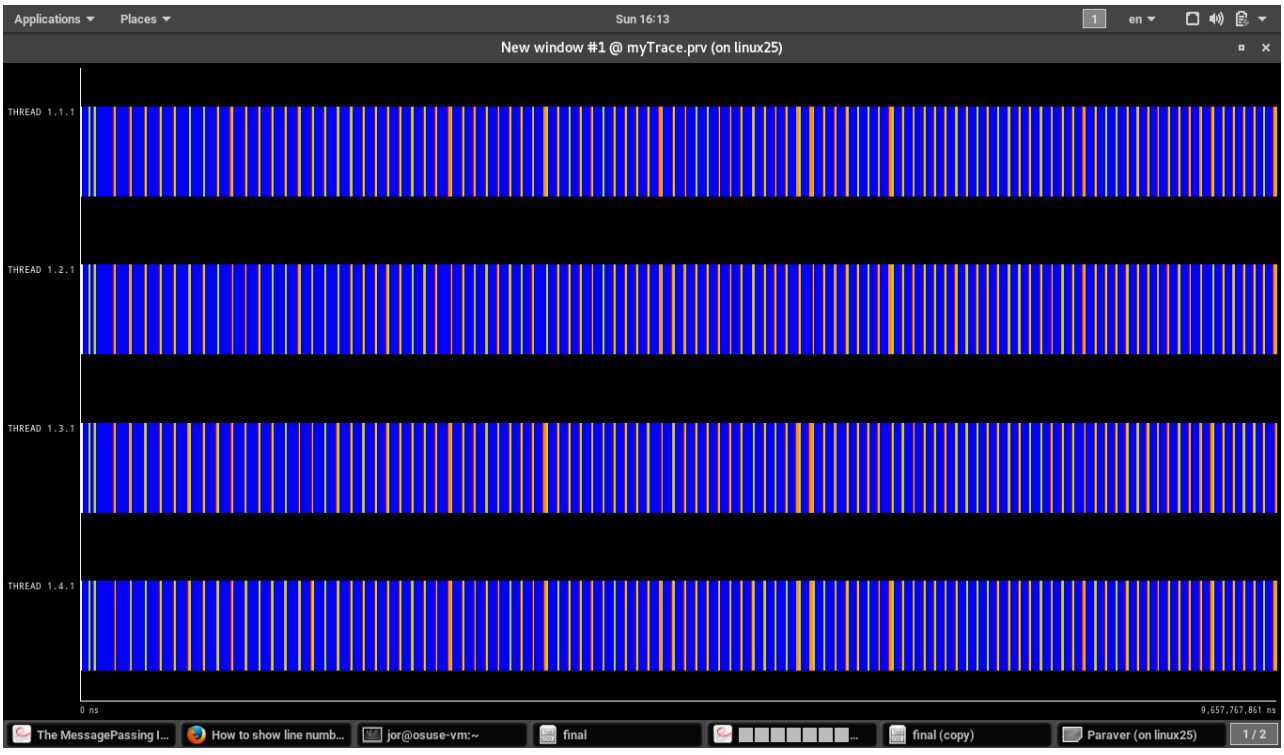
Είναι εμφανές ότι ο μπλε χρωματισμός είναι πιο έντονος στην δεύτερη περίπτωση όπου το μέγεθος του προβλήματος είναι μεγαλύτερο. Αυτό σημαίνει ότι έχουμε λιγότερες καθυστερήσεις λόγο επικοινωνίας. Ο υπολογισμός του εσωτερικού πίνακα στην δεύτερη περίπτωση παίρνει παραπάνω χρόνο και έτσι ο χρόνος αδράνειας είναι μικρότερος.

Αν μεγεθύνουμε την εικόνα βλέπουμε οτι

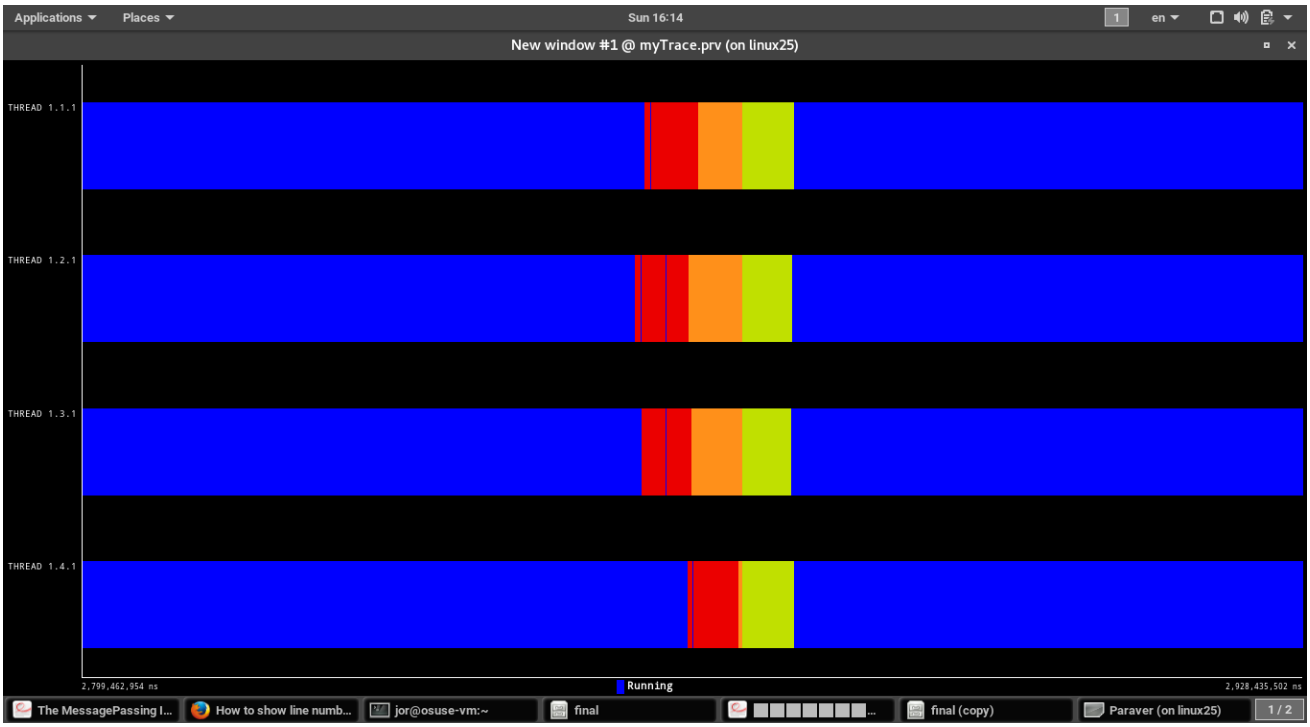



Όπως βλέπουμε ακόμα και τώρα δεν έχουμε πλήρη επικάλυψη της επικοινωνίας καθώς και πάλι χρειάζεται να περιμένουμε την λήψη των μηνυμάτων (φαίνεται από το κόκκινο χρώμα). Οι μπλε γραμμές είναι ανάμεσα στις κόκκινες είναι οι υπολογισμοί των ορίων με την χρήση της waitany. Διαπιστώνουμε ότι αυτή η βελτίωση δεν προσφέρει μεγάλη βελτίωση στο τελικό χρόνο καθώς ο υπολογισμός είναι πολύ μικρός.

Με την allreduce με πίνακα 1728x1728 έχουμε το ακόλουθο.



Και με μεγέθυνση βλέπουμε




Το πορτοκάλι χρώμα οφείλεται στην allreduce. Επομένως αυτή είναι η επιβάρυνση που έχουμε σε κάθε επανάληψη από την χρήση της.

Οι εικόνες μπορούν να βρεθούν σε μεγαλύτερη ανάλυση στο φάκελο images.

# 4. OPENMP/MPI Υλοποίηση

## 4.1 MPI αλλαγές

Προκειμένου να συνδυάσουμε το OPENMP με το MPI χρειάστηκε να κάνουμε μια αλλαγή στην MPI_Init έτσι ώστε το MPI να γνωρίζει οτι δουλεύουμε με threads. Για αυτό το λόγο δοκιμάσαμε δυο εκδοχές της MPI_Init_thread με MPI_THREAD_MULTIPLE και MPI_THREAD_FUNNELED.

Το MPI_THREAD_MULTIPLE ουσιαστικά λέει στο MPI ότι όλα τα threads μπορούν να κάνουν κλήσεις με το MPI. Επίσης οι κλήσεις μπορούν να γίνουν ταυτόχρονα από όλα τα threads και δεν υπάρχει κανένας περιορισμός στην επικοινωνία. Η λύση αυτή επιλέχθηκε για λόγους αποδοτικότητας.

Το MPI_THREAD_FUNNELED ουσιαστικά λέει στο MPI οτι μόνο το master thread μπορεί να κάνει κλήσεις του MPI. Έτσι όλη η επικοινωνία μεταξύ των διεργασιών επιβαρύνει το master thread κάθε process.

## 4.2 Χρήση OPENMP

Όπως αναφέραμε στο τμήμα των μετρήσεων χρησιμοποιήσαμε είτε 2 είτε 4 threads. Η δημιουργία των threads γίνεται έξω από το loop των γύρων του παιχνιδιού. Έτσι αποφεύγουμε το overhead δημιουργίας των threads σε κάθε γύρο.

Για να βρίσκονται όλα τα threads στον ίδιο γύρο του παιχνιδιού χρησιμοποιείται η εντολή **#pragma omp barrier** στο τέλος κάθε γύρου. Άμα δεν το χρησιμοποιούσαμε θα μπορούσε κάποιο thread να συνεχίσει σε επόμενο γύρο ενώ το άλλο βρίσκεται σε προηγούμενο και να είχαμε λάθος αποτελέσματα.

Για τον υπολογισμό του εσωτερικού πίνακα χρησιμοποιείται μια εντολή

**#pragma omp for** στο εσωτερικό loop. Έτσι ο υπολογισμός κάθε σειράς του πίνακα μοιράζεται στα διαθέσιμα threads. Δεν χρειάστηκε να εφαρμόσουμε κάποιο χρονοπρογραμματισμό (scheduling) στο loop καθώς κάθε επανάληψη απαιτεί ίδιο χρόνο δηλαδή υπολογισμό των 8 γειτόνων και έλεγχο αν η νέα τιμή του κελιού πρέπει να είναι μηδέν ή ένα.

Έγινε δοκιμή χρήσης και του **#pragma omp for collapse(2)** που παρέχεται από το OPENMP 3.0 για την παραλληλοποίηση και του εξωτερικού loop ωστόσο οι χρόνοι που περνάμε ήταν χειρότεροι από την παραλληλοποιήση του εσωτερικού loop πιθανόν επειδή οι υπολογισμοί μας είναι άμεσα συνδεδεμένοι με την πρόσβαση στην μνήμη για αυτό η λύση αυτή απορρίφθηκε.

Για την επικοινωνια δοκιμαστηκαν δυο εκδοχες.

1. Με MPI_THREAD_FUNNELED οπου μονο το master thread κανει κλησεις του MPI. Η υλοποιηση βρισκεται στον φακελο openmpMasterThreadCommunication.
2. Με MPI_THREAD_MULTIPLE οπου η επικοινωνια γινεται απο ολα τα threads. Αυτο γινεται με την χρηση της **#pragma omp for** για την αποστολη και της ληψη μηνυματων. Η υλοποιήση βρισκεται στο φακελο openmpParrallelCommunication.

Τέλος χρησιμοποιήθηκε η εντολή **#pragma omp single** για κάποια σειριακά κομμάτια κώδικα όπως ο υπολογισμός της τιμής των τεσσάρων γωνιών(θα μπορούσε να γίνει παράλληλα αλλά ο υπολογισμός δν είναι σε loop και απαιτούνταν αλλαγές στον κώδικα χωρίς να έχουν σημασία 4 υπολογισμοί). Επίσης χρησιμοποιήθηκε και στην αλλαγή του πίνακα από previous σε current καθώς αυτή η αλλαγή αφορά όλα τα threads. Το barrier χρησιμοποιήθηκε πριν από αυτή την αλλαγή καθώς δν θέλουμε κάποιο thread να κάνει υπολογισμούς και ένα άλλο να αλλάξει τον pointer του πίνακα current και previous και τα μισά δεδομένα να αποθηκευτούν στον έναν πίνακα και τα άλλα στον άλλο καθώς θα έχουμε λάθος αποτελέσματα στους υπολογισμούς.

## 4.3 Υπολογισμός απόδοσης

### 4.3.1 Μετρήσεις Χρόνου Εκτέλεσης

Όλες οι μετρήσεις έγιναν με 100 επαναλήψεις δηλαδή 100 γύρους παιχνιδιού. Επίσης οι μετρήσεις είναι σε δευτερόλεπτα(sec).

Οι μετρήσεις έγιναν στα μηχανήματα της σχολής. Σε κάθε μηχάνημα δίνονταν 2 διεργασίας και ήταν διαθέσιμα 14 μηχανήματα.

Για το openmp αλλάξαμε τις ρυθμίσεις και κάθε process βρίσκεται σε διαφορετικό pc αλλά τα threads που δημιουργούνται βρίσκονται στο ίδιο pc. Αφού είχαμε 14 μηχανήματα μπορούσαμε να δοκιμάσουμε μέχρι 9 process.

Ο φάκελος openmpParrallelCommunication που έχει την υλοποίηση openmp/mpi όπου όλα τα threads στέλνουν και λαμβάνουν δεδομένα.

| **Threads per MPI process/Total MPI processes** | | | | |
|---|---|---|---|---|
| **Matrix Size** | 2/4 | 2/9 | 4/4 | 4/9 |
| 144x144 | 0.084780 | 0.069479 | 0.210899 | 0.152935 |
| 720x720 | 0.890845 | 0.432456 | 1.585625 | 0.900429 |
| 1008x1008 | 1.662262 | 0.763934 | 2.666067 | 1.488612 |
| 1440x1440 | 3.229790 | 1.509470 | 4.608766 | 2.64824 |
| 1728x1728 | 4.716810 | 2.338118 | 6.260947 | 3.322313 |
| 2880x2880 | 12.607271 | 5.668418 | 15.76464 | 7.598292 |

Ο φάκελος openmpMasterThreadCommunication που έχει την υλοποίηση openmp/mpi οπού μόνο το master thread κάνει κλήσεις του MPI. Δεν κάναμε μετρήσεις για 4 threads καθώς όπως παρατηρήσαμε πριν οι χρόνοι είναι πολύ πιο αργοί.

| **Threads per MPI process/Total MPI processes** | | |
|---|---|---|
| **Matrix Size** | 2/4 | 2/9 |
| 144x144 | 0.063591 | 0.054963 |
| 720x720 | 0.914514 | 0.447684 |
| 1008x1008 | 1.590618 | 0.739927 |
| 1440x1440 | 3.198775 | 1.458225 |
| 1728x1728 | 4.423657 | 2.123745 |
| 2880x2880 | 12.458520 | 5.281486 |

Παρατηρούμε ότι η υλοποίηση όπου η επικοινωνία γίνεται αποκλειστικά από το master thread είναι λίγο πιο αποδοτική σε σχέση με την υλοποίηση όπου η επικοινωνία γίνεται από όλα τα threads. Επίσης τα 4 threads ανά process είναι αισθητά πιο αργό σε σχέση με 2 threads ανά process.

## Linear

Με την ακολουθιακή υλοποίηση στο μηχάνημα της σχολής linux01 έχουμε τα ακόλουθα αποτελέσματα. Οι μετρήσεις είναι ίδιες με αυτές που είχαμε στο MPI άπλα της εμφανίζομε πάλι εδώ για να είναι πιο εύκολο να γίνουν οι επόμενοι υπολογισμοί.

| **Matrix Size** | **Time** |
|---|---|
| 144x144 | 0.208812 |
| 720x720 | 5.245523 |
| 800x800 | 6.494685 |
| 1008x1008 | 10.240248 |
| 1440x1440 | 20.897311 |
| 1600x1600 | 25.760992 |
| 1728x1728 | 30.029273 |
| 2880x2880 | 83.226259 |

### 4.3.2 Επιτάχυνση(speedup)

Για να υπολογίσουμε την επιτάχυνση χρησιμοποιούμε τον τύπο S = Tserial/Tparallel.

Στους παρακάτω πίνακες βλέπουμε την επιτάχυνση ανάλογα με το μέγεθος του πίνακα και το πλήθος των διεργασιών.

Επομένως για το OPENMP/MPI θα ελέγξουμε τους χρόνους που βρήκαμε με τους χρόνους που κάνει η ακολουθιακή υλοποίηση στο μηχάνημα της σχολής linux01.

Ο φάκελος openmpParrallelCommunication που έχει την υλοποίηση openmp/mpi όπου όλα τα threads στέλνουν και λαμβάνουν δεδομένα.

| **Threads per MPI process/Total MPI processes** | | | | |
|---|---|---|---|---|
| **Matrix Size** | 2/4 | 2/9 | 4/4 | 4/9 |
| 144x144 | 2.46 | 3 | 0.99 | 1.37 |
| 720x720 | 5.89 | 12.3 | 3.3 | 5.83 |
| 1008x1008 | 6.16 | 13.4 | 3.84 | 6.88 |
| 1440x1440 | 6.47 | 13 | 4.53 | 7.89 |
| 1728x1728 | 6.36 | 12.84 | 4.8 | 9.04 |
| 2880x2880 | 6.6 | 14.68 | 5.28 | 10.95 |

Ο φάκελος openmpMasterThreadCommunication που έχει την υλοποίηση openmp/mpi όπου μόνο το master thread κάνει κλήσεις του MPI.

| **Threads per MPI process/Total MPI processes** | | |
|---|---|---|
| **Matrix Size** | 2/4 | 2/9 |
| 144x144 | 3.28 | 3.8 |
| 720x720 | 5.74 | 11.72 |
| 1008x1008 | 6.44 | 13.84 |
| 1440x1440 | 6.53 | 14.33 |
| 1728x1728 | 6.79 | 14.13 |
| 2880x2880 | 6.68 | 15.79 |

### 4.3.3 Αποδοτικότητα(efficiency)

Για τον υπολογισμό την αποτελεσματικότητας έχουμε τον τύπο E= S/p, όπου S είναι η επιτάχυνση και p το πλήθος των πυρήνων.

Ο υπολογισμός της αποδοτικότητας θα γίνει με τον ίδιο τύπο ωστόσο στον παρανομαστή θα βάλουμε το πλήθος των threads αντί για το πλήθος των process. Έτσι όταν χρησιμοποιούσαμε 4 process όπου το καθένα είχε 2 threads έχουμε σύνολο 8 threads. Αντίστοιχα για τα υπόλοιπα

Ο φάκελος openmpParrallelCommunication που έχει την υλοποίηση openmp/mpi όπου όλα τα threads στέλνουν και λαμβάνουν δεδομένα.

| **Total threads/process** | | | | |
|---|---|---|---|---|
| **Matrix Size** | 8/4 | 18/9 | 16/4 | 36/9 |
| 144x144 | 0.31 | 0.17 | 0.06 | 0.04 |
| 720x720 | 0.74 | 0.68 | 0.20 | 0.16 |
| 1008x1008 | 0.77 | 0.74 | 0.24 | 0.19 |
| 1440x1440 | 0.81 | 0.72 | 0.28 | 0.22 |
| 1728x1728 | 0.8 | 0.71 | 0.3 | 0.25 |
| 2880x2880 | 0.83 | 0.82 | 0.33 | 0.31 |

Ο φάκελος openmpMasterThreadCommunication που έχει την υλοποίηση openmp/mpi όπου μόνο το master thread κάνει κλήσεις του MPI.

| **Threads per MPI process/Total MPI processes** | | |
|---|---|---|
| **Matrix Size** | 8/4 | 18/9 |
| 144x144 | 0.41 | 0.21 |
| 720x720 | 0.72 | 0.65 |
| 1008x1008 | 0.81 | 0.77 |
| 1440x1440 | 0.82 | 0.8 |
| 1728x1728 | 0.85 | 0.79 |
| 2880x2880 | 0.84 | 0.88 |

### 4.3.4 Κλιμάκωση(scalability)

Ας ελέγξουμε την κλιμάκωση που παρουσιάζει η υβριδική υλοποίηση όπου όλα τα threads κάνουν την επικοινωνία. Τις μετρήσεις όπου έχουμε 4 threads ανα process τις αγνοούμε γιατί είναι χειρότερες από αυτές με 2 threads ανά process. Από τις παραπάνω μετρήσεις παρατηρούμε ότι για μικρο πίνακα δηλαδή για τον πίνακα 144x144 η κλιμάκωση είναι κακή καθώς η αποτελεσματικότητα των 2 μετρήσεων μας είναι [4,9]=[0.41, 0.21]. Δηλαδή παρατηρούμε ότι η αποδοτικότητα μειώθηκε στο μισό καθώς σχεδόν διπλασιάσαμε το πλήθος των διεργασιών και διατηρούμε σταθερό το μέγεθος του προβλήματος. Ωστόσο για μεγάλους πίνακες όπως 2880x2880 είχαμε τις μετρήσεις [4,9]=[0.83,0.82] δηλαδή η αποδοτικότητα παραμένει σχεδόν σταθερή πράγμα που σημαίνει ότι το πρόβλημα εμφανίζει καλή κλιμάκωση. Επομένως για μεγάλο πλήθος δεδομένων που ειναι και το κύριο ζητούμενο του προβλήματος το πρόβλημα εμφανίζει καλή κλιμάκωση.

### 4.3.5 Συμπεράσματα

Παρατηρούμε ότι η υλοποίηση όπου μόνο το master thread αναλαμβάνει την επικοινωνία είναι η πιο αποδοτική. Η καλύτερη απόδοση που είδαμε στις μετρήσεις είναι 0.85. Επισης η χρήση 4 thread παρουσιάζει χειρότερα αποτελέσματα σε σχέση με την χρήση 2 thread.

## 4.4 Σύγκριση με απλό MPI

Παρατηρούμε ότι η αποδοτικότητα του υβριδικού προγράμματος είναι χειρότερη σε σύγκριση με το απλό MPI, αφού στο υβριδικό είχαμε καλύτερη αποδοτικότητα 0.85 ενώ στο MPI φτάνει το 0.9. Για το ίδιο πλήθος process για παράδειγμα για 4 process και στον μεγάλο πίνακα 2880x2880 είχαμε τις μετρήσεις 12.607271 και 23.013436 αντίστοιχα. Από αυτό βλέπουμε όντως ότι η χρήση 2 threads βελτιώνει σημαντικά τον χρόνο της τάξεως του 1.82 T(openmp) = T(MPI) ωστόσο όταν μετράμε την αποδοτικότητα έχουμε συνολικά 2*4=8 threads που έχει ως αποτέλεσμα αυτή να είναι χειρότερη.

Όσων άφορα την κλιμάκωση και οι δυο μέθοδοι παρουσιάζουν παρόμοια συμπεριφορά όταν αυξάνουμε το πλήθος των process και διατηρούμε τα threads 2. Όταν αυξάνουμε το πλήθος των threads σε διπύρηνο επεξεργαστή η κλιμάκωση είναι κακή.

# 5. CUDA

## 5.1 Λεπτομέρειες υλοποίησης

Για την υλοποίηση του κώδικα σε Cuda ακολουθήθηκαν τα παραδείγματα που είχαν δοθεί στο μάθημα. Αρχικά αντιγράφουμε τα δεδομένα από την CPU memory στην GPU memory και στην συνεχεια σε καθε loop καλούμε τον κώδικα kernel. Για το πλήθος των blocks και το μέγεθος των threads per blocks έγιναν αρκετές δοκιμές.

Ο κώδικας kernel βρίσκεται στην συνάρτηση

__global__ void updatedValueGPU(int* prevBlock, int* currentBlock, int nRows, int nColumns)

Αυτό που κάνει είναι να βρει ποιο cell υπολογίζει με βάση το threadIdx και blockIdx και στην συνέχεια να βρει ποια είναι τα 8 γειτονικά cells. Στην συνέχεια ελέγχει πόσα είναι ενεργά και υπολογίζει την νέα τιμή του cell.

Οι υπολογισμοί των γειτόνων με την χρήση του mod είναι αρκετά κοστοβοροι σύμφωνα με το documentation του cuda για αυτό και γίνονται μια φορά και στην συνέχεια αποθηκεύονται στις μεταβλητές.

## 5.2 Υπολογισμός απόδοσης

### 5.2.1 Μετρήσεις Χρόνου Εκτέλεσης

Οι μετρήσεις έγιναν στο μηχάνημα 195.134.67.205 που έχει κάρτα γραφικών NVidia GTX-480. Οι μετρήσεις έγιναν για διάφορα μεγέθη thread ανά block για να δούμε ποιος συνδυασμός έχει τα καλύτερα αποτελέσματα. Όλα τα μεγέθη είναι πολλαπλάσια του 32 (wrap size).

| | **Threads per Block** | | | |
|---|---|---|---|---|
| **Matrix Size** | 128 | 256 | 512 | 1024 |
| 144x144 | 0.000943 | 0.001007 | 0.001007 | 0.001327 |
| 720x720 | 0.016915 | 0.016725 | 0.017793 | 0.020903 |
| 800x800 | 0.020640 | 0.020415 | 0.0215 | 0.025289 |
| 1008x1008 | 0.032842 | 0.032355 | 0.034394 | 0.040544 |
| 1440x1440 | 0.066105 | 0.065289 | 0.068764 | 0.081176 |
| 1600x1600 | 0.081610 | 0.080552 | 0.08486 | 0.100022 |
| 1728x1728 | 0.095428 | 0.093898 | 0.099102 | 0.116652 |
| 2880x2880 | 0.265818 | 0.260359 | 0.277119 | 0.324600 |

Με την ακολουθιακή υλοποίηση στο μηχάνημα 195.134.67.205 με CPU i7 2600K έχουμε τα ακόλουθα αποτελέσματα.

| Matrix Size | Time |
|---|---|
| 144x144 | 0.064138 |
| 720x720 | 1.575883 |
| 800x800 | 1.950608 |
| 1008x1008 | 3.107099 |
| 1440x1440 | 6.338924 |
| 1600x1600 | 7.846581 |
| 1728x1728 | 9.147075 |
| 2880x2880 | 25.242224 |

### 5.2.2 Επιτάχυνση(speedup)

Για να υπολογίσουμε την επιτάχυνση χρησιμοποιούμε τον τύπο S = Tserial/Tparallel.

Στους παρακάτω πίνακες βλέπουμε την επιτάχυνση ανάλογα με το μέγεθος του πίνακα και το πλήθος των διεργασιών.

Επομένως για το Cuda θα ελέγξουμε τους χρόνους που βρήκαμε με τους χρόνους που κάνει η ακολουθιακή υλοποίηση.

| | Threads per Block | | | |
|---|---|---|---|---|
| **Matrix Size** | 128 | 256 | 512 | 1024 |
| 144x144 | 68.01 | 63.69 | 63.69 | 48.33 |
| 720x720 | 93.14 | 94.22 | 88.57 | 75.39 |
| 800x800 | 94.5 | 95.54 | 90.72 | 77.13 |
| 1008x1008 | 94.6 | 96.03 | 90.34 | 76.64 |
| 1440x1440 | 95.89 | 97.09 | 92.18 | 78.09 |
| 1600x1600 | 96.15 | 97.41 | 92.47 | 78.44 |

| | | | | |
|---|---|---|---|---|
| 1728x1728 | 95.85 | 97.42 | 92.3 | 78.41 |
| 2880x2880 | 94.9 | 96.95 | 91.09 | 77.76 |

Από τις παραπάνω μετρήσεις μπορούμε να διαπιστώσουμε ότι όταν έχουμε 256 threads ανά block έχουμε την μεγαλύτερη επιτάχυνση. Επίσης παρατηρούμε ότι όσο αυξάνουμε το μέγεθος του προβλήματος η επιτάχυνση βελτιώνεται μέχρι που φτάνει σε κάποιο σημείο σύγκλισης. Επίσης από τους χρόνους μπορούμε να διαπιστώσουμε ότι όταν τετραπλασιάζουμε τον πίνακα αντίστοιχα και ο χρόνος τετραπλασιάζεται. Πχ από 1440χ1440 σε 2880χ2880 ο χρόνος πήγε από 0.065289 σε 0.260359 που είναι 3.99 φόρες μεγαλύτερο.

### 5.3 Σύγκριση με απλό MPI

Οι μετρήσεις έγιναν σε διαφορετικά μηχανήματα καθώς τα μηχανήματα linux της σχολής δεν υποστηρίζουν το cuda και το μηχάνημα 195.134.67.205 δεν υποστηρίζει το MPI. Ωστόσο θα μπορούσαμε να συγκρίνουμε την επιτάχυνση των δυο μεθόδων. Το MPI το δοκιμάσαμε σε 4, 9, 16, 25 διεργασίες όπου η μέγιστη επιτάχυνση ήταν 21.35 όταν είχαμε 25 διεργασίες σε πίνακα 1600χ1600. Με την υλοποίηση του cuda καταφέραμε να έχουμε επιτάχυνση 97.42 σε σύγκριση με το ακολουθιακό πρόγραμμα. Αυτό συμβαίνει καθώς η GPU μπορεί να παραλληλοποιήση πάρα πολύ το προβλήματα και ουσιαστικά κάθε thread της να υπολογίζει ένα κελί του πίνακα. Έτσι διαπιστώνουμε ότι η υλοποίηση cuda είναι πολύ πιο γρήγορη σε σύγκριση με αυτή του MPI.

## 6. Συμπεράσματα

Γενικά και από τις 3 υλοποιήσεις παρατηρούμε ότι όσο αυξάνουμε το μέγεθος του προβλήματος η επιτάχυνση βελτιώνεται. Η υλοποίηση με παράλληλο προγραμματισμό σε GPU με την χρήση του cuda μας έδωσε την μεγαλύτερη επιτάχυνση που φτάνει μέχρι και 97.42 φόρες σε σύγκριση με την ακολουθιακή υλοποίηση. Με το mpi καταφέραμε να έχουμε και εκεί καλή παραλληλοποίηση που για μεγάλους πίνακες έχει αποδοτικότητα ακόμα και 0.9 που είναι πολύ κοντά στο ιδανικό 1. Επίσης παρατηρούμε ότι η κλιμάκωση είναι πολύ καλή. Η χρήση της allreduce για τον έλεγχο του πλέγματος προκαλεί μια μικρή επιβάρυνση στους χρόνους καθώς επίσης και η κλιμάκωση είναι λίγο χειρότερη καθώς όσο αυξάνονται οι πυρήνες τόσο μεγαλύτερο θα είναι το κόστος της επικοινωνίας των πυρήνων. Η υβριδική υλοποίηση με openmp και mpi παρατηρούμε οτι έχει λίγο χειρότερη αποδοτικότητα σε σύγκριση με την απλή υλοποίηση του mpi. Ωστόσο και σε αυτή την περίπτωση επιτυγχάνουμε 0.85 αποδοτικότητα που είναι πολύ καλή.