Γούλα-Δημητρίου Μιχαήλα, 2018503 ΗΑ

Μπέλλος Φίλιππος, 2018511 ΗΑ

Εργασία 2019-20: Conway's Game of Life

MPI-OpenMp-Cuda

Στην παρούσα εργασία αναπτύχθηκε το Conway's Game of Life σε γλώσσα προγραμματισμού C χρησιμοποιώντας τρεις τρόπους παραλληλίας, το MPI, το OpenMP και το Cuda. Αρχικά, θεωρούμε σημαντικό να αναφέρουμε κάποιες σημαντικές θεωρητικές πληροφορίες που θα μας βοηθήσουν στη συνέχεια στην ανάλυση της απόδοσης των υλοποιήσεων που αναφέρθηκαν.

Performance

Speedup-Efficiency

Υπάρχουν δυο μεγέθη για την μέτρηση της απόδοσης. Το speedup και η αποδοτικότητα. Ιδανικά θέλουμε να χωρίσουμε την διεργασία σε όμοια κομμάτια όσα και οι επεξεργαστές(p) (load balancing). Αν μπορούμε να το κάνουμε αυτό τότε το πρόγραμμα μας θα είναι N φορές πιο γρήγορο δηλαδή αν ονομάσουμε Tserial τον χρόνο που κάνει ο καλύτερος σειριακός κώδικας και Tparallel τον χρόνο που χρειάζεται ο παράλληλος κώδικας, αυτοί συνδέονται με την σχέση του speedup

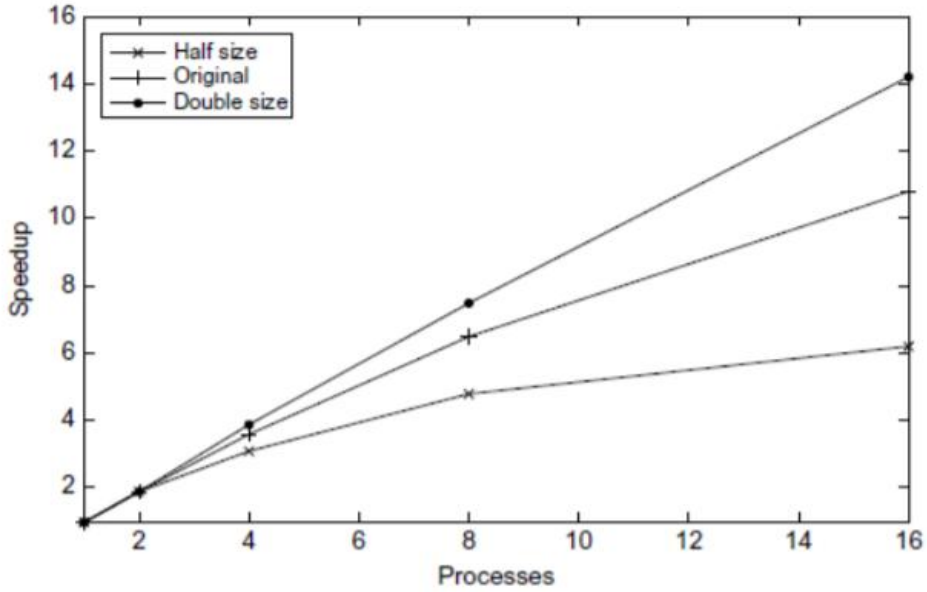
$$S = \frac{T_S}{T_P}$$

Κατά συνέπεια το ανώτατο όριο είναι το γραμμικό speedup. Επιπλέον όσο το p αυξάνεται αναμένουμε το S να αυξάνεται ανάλογα. Δυστυχώς αυτό είναι μόνο θεωρητικό διότι η εισαγωγή δικτύων και πολλών επεξεργαστών δημιουργεί επιπλέον χρόνο T overhead το οποίο αυξάνει τον χρόνο T parallel.

Εισάγουμε λοιπόν άλλο ένα μέγεθος που θα ονομάσουμε αποδοτικότητα (efficiency) που δίνεται από τον τύπο:

$$E = \frac{S}{p} = \frac{T_S}{pT_p}$$

Τα μεγέθη αυτά τα χρησιμοποιούμε σε κάθε παράλληλο πρόγραμμα και μεταβάλλονται στην πλειοψηφία των περιπτώσεων ανάλογα με τον αριθμό επεξεργαστών που διαθέτουμε, όπως φαίνεται και στο παρακάτω παράδειγμα.



Νόμοι Amdahl - Gustafson

Δυστυχώς η προσπάθεια μας δεν γίνεται να έχει άπειρη βελτίωση. Από το 1960 ο Gene Amdahl κατέγραψε ένα πάρα πολύ σημαντικό συμπέρασμα το οποίο είναι γνωστό ως Amdahl's Law. Κατέγραψε ότι, εάν όλο το πρόβλημα δεν γίνεται να γραφτεί σε απόλυτα παράλληλη μορφή τότε το συνολικό speedup έχει ανώτατο όριο και μάλιστα χαμηλό. Εάν για παράδειγμα υποθέσουμε ότι το 80% του κώδικα είναι παραλληλοποιήσιμο και το 20% όχι, τότε το μέγιστο speedup δίνεται από:

$$S = \frac{1}{0.8 / p + 0.2}$$

$$S \leq \frac{1}{0.2} = 5 \qquad \text{as} \qquad p \rightarrow \infty$$

Κατά συνέπεια ακόμα και αν το p είναι άπειρο έχουμε μέγιστο speedup 5. Αυτό αποτελεί ε΄να απ'τα limitations στην παράλληλη επεξεργασία. Αυτό όμως δεν σημαίνει ότι ακόμα και ένα Speedup 2 δεν είναι πολύ μεγάλη βελτίωση στον χρόνο. Επιπλέον ο νόμος Amdahl δεν λαμβάνει υπόψη του ένα σημαντικό ζήτημα το οποίο είναι το μέγεθος του προβλήματος και λαμβάνεται από τον νόμο Gustafson.

$$S = p - \alpha (p - 1)$$

Αν P είναι ο αριθμός των επεξεργαστών, και α το τμήμα που δεν παραλληλοποιείται, τότε ο νόμος μας λέει ότι, S(p) είναι ο λόγος του χρόνου που χρειάζεται η σειριακή μηχανή προς τον χρόνο που χρειάζεται μία επεξεργαστική μονάδα του παράλληλου συστήματος και με λίγα λόγια λέει ότι, λόγω του αυξανόμενου μεγέθους των δεδομένων οι προγραμματιστές μπορούν να χρησιμοποιήσουν την παραλληλία για να λύσουν στον ίδιο χρόνο μεγαλύτερα Αυτό έχει σαν αποτέλεσμα να περιορίζονται οι συνέπειες του νόμου του Amdahl.

Scalability

Ένα επιπλέον μέγεθος που πρέπει να έχουμε υπόψη είναι το scalability. Αυτό το βασικό μέγεθος μπορεί να γίνει κατανοητό με το εξής παράδειγμα. Έστω ότι έχουμε ένα πρόγραμμα με συγκεκριμένο αριθμό threads/core και συγκεκριμένο μέγεθος εισόδου. Έστω ότι έχουμε speedup S. Αν τώρα αυξήσουμε των αριθμό των επεξεργαστών και μπορούμε να βρούμε μια είσοδο ώστε το πρόγραμμα να παραμένει με speedup S τότε λέμε ότι το πρόβλημα είναι Scalable.

<u>Σχεδιασμός παράλληλων προγραμμάτων-Μεθοδολογία Foster</u>

Αναζητούμε λοιπόν έναν τρόπο να παραλληλοποιήσουμε ένα πρόγραμμα που γνωρίζουμε τον σειριακό του κώδικα. Γνωρίζουμε πχ ότι θα προσπαθήσουμε να χωρίσουμε την εργασία σε ίσα κομμάτια. Γνωρίζουμε επίσης ότι θα πρέπει κάπως να ορίσουμε την επικοινωνία μεταξύ των επεξεργαστών. Για τη σχεδίαση λοιπόν του παράλληλου κώδικα μας θα ακολουθήσουμε ένα σύνολο βημάτων που είναι γνωστό ως ***μεθοδολογία Foster***

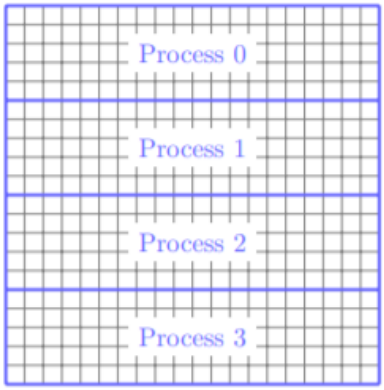
Ορίζουμε λοιπόν τα εξής βήματα.

1. Partitioning, δηλαδή επιμερισμός, δηλαδή ο λογικός χωρισμός του υπολογιστικού έργου σε επιμέρους υποέργα, αυτά που θα υλοποιούν οι παράλληλες εργασίες. Αυτό το βήμα ονομάζεται επιμερισμός ή καταμερισμός.
2. Communication. Ορίζουμε το σύνολο της επικοινωνίας που θα υπάρξει μεταξύ των processes.
3. Agglomeration.Συνδυασμός των 2 παραπάνω βημάτων σε μεγαλύτερες διεργασίες.
4. Mapping. Ορίζουμε τις διεργασίες από τα προηγούμενα σε κάθε Cpu με στόχο την ελαχιστοποίηση της επικοινωνίας και το βέλτιστο Load Balancing.

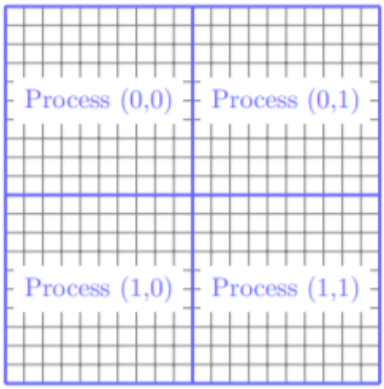
Ανάλυση διαμέρισης(επιμερισμού)

Για το πρώτο βήμα, Υπάρχουν δύο βασικές μέθοδοι επιμερισμού:επιμερισμός δεδομένων(*domain decomposition*) και επιμερισμός λειτουργιών (*functional decomposition*).

Ξεκινάμε με τον επιμερισμό δεδομένων , όπου εφαρμόσαμε επιμερισμό σε σειρές και επιμερισμό σε 2D blocks. Στην περίπτωση του 1D row-wise επιμερισμό, κάθε διεργασία έχει δύο γείτονες(πάνω και κάτω) στους οποίους πρέπει να στείλει ολόκληρες τις σειρές , όπως βέβαια και να λάβει απ' αυτούς αντίστοιχα.



Στη διαμέριση με blocks κάθε διεργασία έχει 8 γείτονες όπου στέλνει/λαμβάνει σειρές/στήλες/γωνιακά.



Για την τελική μας υλοποίηση καταλήξαμε στον δεύτερο τρόπο, καθώς αποδείχτηκε ότι είναι πιο αποδοτικός για το πρόβλημα μας.

Για να καταλήξουμε στο συμπέρασμα αυτό, κάναμε μια ανάλυση για scalability και performance των δύο τρόπων.

Υποθέτουμε λοιπόν ότι θέλουμε να γίνει το simulation του Game of Life σε ένα πεπερασμένο τετραγωνικό grid nxn με περιοδικά boundary conditions σε μια μηχανή με p processes.

Με τον πρώτο τρόπο(διαμέριση σε σειρές) , χωρίζουμε τις σειρές του grid σε p blocks των n/p σειρών το καθένα και αναθέτουμε σε κάθε block μια διεργασία. Με τον δεύτερο τρόπο, αναθέτουμε τις διεργασίες σε ένα δισδιάστατο mesh μεγέθους $\sqrt{p} * \sqrt{p}$ και χωρίζουμε τις σειρές και τις στήλες σε $\sqrt{p}$ blocks ίδιου μεγέθους. Έπειτα, αναθέτουμε κάθε $\sqrt{p} * \sqrt{p}$ =p intersections σε μια ξεχωριστή διεργασία. Με μια γρήγορη ματιά, μπορούμε να υποθέσουμε ότι πρώτον η μονοδιάστατη κατανομή οδηγεί σε μεγαλύτερη inter-process επικοινωνία και δεύτερον ότι δε μπορεί να υποστηρίξει τόσο πολλές δειργασίες όπως η δισδιάστατη κατανομή(n αντί για n^2). Μένει τώρα να το δείξουμε κιόλας.

1-D διαμέριση

Όπως είπαμε, θεωρούμε ένα τετραγωνικό grid nxn διανεμημένο σε p processes με μονοδιάστατη block row κατανομή. Σε κάθε process αντιστοιχεί ένα block row αποτελούμενο από n/p rows και ακριβώς n columns. Για να μπορεί μια process να υπολογίσει τον αριθμό των γειτόνων από την πάνω γραμμή στο local grid της , πρέπει να έχει πρόσβαση στην κάτω γραμμή του πάνω block. Αντίστοιχα, πρέπει να έχει πρόσβαση στην πάνω γραμμή του κάτω block για να υπολογίσει τους γείτονες της κάτω γραμμής. Ο χρόνος για την ανταλλαγή αυτών των boundary σημείων είναι

ts+tw*n, (1)

όπου ts είναι το κόστος της εκκίνησης(start-up ή latency cost) και tw είναι ο χρόνος μεταφοράς σε επίπεδο λέξης(word transfer time).

Μετά την ανταλλαγή των boundaries, οι processes κάνουν update τα local grids τους παράλληλα χωρίς περαιτέρω επικοινωνία. Ο χρόνος εκτέλεσης για αυτή τη φάση είναι ανάλογος του αριθμού των σημείων του local grid. Άρα computation time θα είναι

tc* n^2 /p, (2)

όπου tc είναι ο χρόνος να γίνει update ένα grid point. Προσθέτοντας τους χρόνους επικοινωνίας και εκτέλεσης βρίσκουμε ότι ο παράλληλος χρόνος εκτέλεσης σε συνάρτηση με το n και το p είναι

Tp(n,p)=tc* n^2 /p + ts+tw*n. (3)

Θεωρώντας τον σειριακό χρόνο

Ts(n)=tc* n^2 , (4)

βρίσκουμε ότι το overhead του παράλληλου είναι

To(n,p)=p*Tp(n,p)-Ts(n)=ts*p+tw*n*p. (5)

Ας δούμε τώρα και το scalability της 1-D διαμέρισης. Το παράλληλο efficiency είναι

Ep(n,p)=Ts(n)/(p*Tp(n,p))=Ts(n)/(Ts(n)+To(n,p))=1/(1+To(n,p)/Ts(n)). (6)

Η εξίσωση αυτή (6) μας λέει ότι αν αυξήσουμε τον αριθμό των processes , το efficiency πέφτει αφού το overhead αυξάνεται με την αύξηση του p, ενώ ο σειριακός χρόνος εκτέλεσης είναι σταθερός. Από την άλλη, όταν αυξάνουμε τη διάσταση του grid περιμένουμε να αυξηθεί το efficiency αφού ο σειριακός χρόνος εκτέλεσης αυξάνεται γρηγορότερα από το παράλληλο overhead(n^2 έναντι n).

Σύμφωνα με την (6), μπορούμε να κρατήσουμε σταθερό το efficiency ,ακόμα κι αν αυξήσουμε τις processes και το μέγεθος του grid, αν κρατήσουμε σταθερό τον λόγο To(n,p)/Ts(n) (7).

Αλλά μας αρκεί να βρούμε το n συναρτήσει του p έτσι ώστε ο λόγος να μην αυξάνεται.

Θα βρούμε άρα την συνάρτηση n=n(p) έτσι ώστε To(n,p)/Ts(n)=O(1). (8)

Αντικαθιστώντας με την (4) και την (5) έχουμε

(ts*p+tw*n*p)/tc* n^2 =O(1)=>ts*p/tc* n^2 +tw*n*p/tc* n^2 =O(1) (9)

Για το πρώτο μέλος στην πάνω εξίσωση(latency) συμπεραίνουμε ότι

n^2 =Ω(p) ή n^2 =Ω($\sqrt{p}$) (10)

Άρα βλέπουμε ότι το n πρέπει να αυξηθεί στη χειρότερη με τον ίδιο ρυθμό με το sqrt(p), έτσι ώστε το efficiency να μείνει σταθερό ή να αυξηθεί.

Για το δεύτερο μέλος(bandwith), αντίστοιχα καταλήγουμε ότι

n^2 =Ω(n*p) ή n^2 =Ω(p) (11).

Καταλήγουμε άρα ότι λόγω bandwith,το n πρέπει να αυξηθεί στη χειρότερη με τον ίδιο ρυθμό με το p, έτσι ώστε το efficiency να μείνει σταθερό ή να αυξηθεί.

Άρα αφού το αυξάνεται γρηγορότερα από το ,το bandwith είναι αυτό που καθορίζει το scalability στην 1-D διαμέριση.

Πέρα όμως από το latency και το bandwith, το scalability μπορεί να δυσκολευτεί να επιτευχθεί λόγω του concurrency, δηλαδή του αριθμού των processes που μπορούν να χρησιμοποιηθούν ταυτόχρονα. Στην 1-D διαμέριση, δε γίνεται να ξεπεράσουμε τις n processes αφού κάθε process μπορεί να έχει το λιγότερο 1 γραμμή στο local grid της.

Άρα καταλήγουμε ότι n=Ω(p)(12).

Σε αυτό που καταλήξαμε από τις (11) και (12) , στην ουσία σημαίνει ότι πρέπει να χρησιμοποιούμε ολοένα και περισσότερη μνήμη ανά process αν θέλουμε το n να έχει ίδιο ρυθμό αύξησης με το p. Επομένως, κάποια στιγμή θα μείνουμε από μνήμη.

2-D διαμέριση

Ας θεωρήσουμε ένα mesh των $\sqrt{p}$ * $\sqrt{p}$ processes. Όπως είπαμε και προηγουμένως, κάθε process πρέπει να ανταλλάξει boundary σημεία με 8 γείτονες. Ο αριθμός των σημείων αυτών είναι 4*n/ $\sqrt{p}$ Το latency μένει σταθερό αλλά το bandwith όχι αφού από Θ(n) που ήταν προηγουμένως γίνεται Θ(n/ $\sqrt{p}$). Επαναλαμβάνοντας την ίδια ανάλυση με πριν καταλήγουμε ότι για το bandwith μέλος έχουμε n^2 =Ω(n* $\sqrt{p}$)(13), δηλαδή για να μείνει σταθερό το efficiency πρέπει το n να αυξάνει με τον ίδιο ρυθμό με το $\sqrt{p}$. Αυτό αποτελεί σημαντική βελτίωση από την 1-D διαμέριση (ήταν p αντί για $\sqrt{p}$). Όσο για το concurrency, μπορούμε να χρησιμοποιήσουμε μέχρι n^2 processes. Άρα, έχουμε ένα όριο για το scalability που είναι n=Ω($\sqrt{p}$)(14) αφού p=O(n^2). Βλέπουμε και εδώ τη βελτίωση από p σε $\sqrt{p}$.

Όσον αφορά τη μνήμη, πάλι για κάθε process θέλουμε n^2 /p αλλά λόγω της (14) η μνήμη για κάθε process είναι σταθερή αφού n^2 /p και n=sqrt(p) δίνει ($\sqrt{p}$)^2/p=1.

Από τα παραπάνω λοιπόν συμπεραίνουμε ότι λόγω των ορίων που θέτουν στο scalability της 1-D διαμέρισης το bandwith και το concurrency δημιουργεί πρόβλημα με τη μνήμη, γεγονός που μας οδηγεί να προχωρήσουμε την υλοποίηση εφαρμόζοντας 2-D διαμέριση.

Επιμερισμός λειτουργιών

Η ευκολότερη ίσως μεθοδολογία σχεδιασμού παράλληλων προγραμμάτων η οποία βασίζεται στο μοντέλο υψηλού επιπέδου SPMD και η οποία εφαρμόσαμε στην υλοποίηση μας είναι η μεθοδολογία master-workers. Το πρόγραμμα αποτελείται από ένα σύνολο υπο-ρουτινών και όλοι οι επεξεργαστές εκτελούν το ίδιο εκτελέσιμο αλλά εκτελούν τμήμα του, ανάλογα με τη ταυτότητά τους (SPMD). Μια παράλληλη εργασία (νήμα ή διεργασία) θεωρείται ως Συντονιστής (Master Thread / Process).Ο συντονιστής που εφαρμόσαμε είναι υπεύθυνος για τη τελική συλλογή και παρουσίαση των αποτελεσμάτων και των χρόνων. Οι υπόλοιπες εργασίες εκτελούν το τμήμα του προγράμματος που τις αφορά. Το initiation του grid στην εργασία μας προτιμήθηκε να γίνεται από όλες τις processes,η καθεμιά το υπο-grid της.

Οι πιο σύνθετοι τύποι προβλημάτων συνήθως περιέχουν κάποια μορφή εξάρτησης δεδομένων ή συγχρονισμού. Έτσι και στη δική μας περίπτωση απαιτείται επικοινωνία, είτε απλά για συγχρονισμό επαναλήψεων ή/και για ανταλλαγή δεδομένων. Για την εργασία μας απαιτείται επικοινωνία μεταξύ των γειτονικών σημείων ώστε να υπολογιστούν οι τιμές των παραμέτρων σε όλο το grid. Επίσης επειδή το μοντέλο εξελίσσεται στο χρόνο, απαιτείται και συγχρονισμός μεταξύ διαδοχικών υπολογισμών στο ίδιο σημείο.

Επομένως, δημιουργήσαμε δύο υλοποιήσεις, μία για blocking(ανασταλτική) επικοινωνία όπου οι εμπλεκόμενες processes αναστέλλουν κάθε άλλη λειτουργία μέχρι την ολοκλήρωση της επικοινωνίας, και μία για non-blocking(μη-ανασταλτική) όπου οι εμπλεκόμενες processes μπορεί να εκτελούν άλλες λειτουργίες μέχρι την ολοκλήρωση της επικοινωνίας.

<u>Υλοποίηση</u>

MPI

Για την απλοποίηση της υλοποίησης του προγράμματος (όπως προτάθηκε στις διαλέξεις και το eclass) κάνουμε τις εξής παραδοχές:

1. Το πλέγμα πρέπει να είναι τετράγωνο (NxN).
2. Το πλήθος των processes πρέπει να είναι τέλειο τετράγωνο (x*x) και η ρίζα του (x) να διαιρεί τέλεια την κάθε πλευρά του πλέγματος (N % x = 0). Με αυτόν τον τρόπο διασφαλίζουμε τον ισομερισμό του προβλήματος στις διεργασίες.
3. Η αρχική γενιά παράγεται randomly, απαιτώντας όμως κάθε cell να έχει 40% πιθανότητα να είναι ζωντανό. Αυτό γίνεται για να αποφύγουμε τυχόν αποκλίσεις σε χρόνους εκτέλεσης που οφείλονται σε μεγάλη διαφορά ζωντανών ή νεκρών cells από τρέξιμο σε τρέξιμο.

Για τη βελτίωση της απόδοσης του κώδικα MPI υλοποιήσαμε τα εξής σημεία που προτείνετε:

1. Διαμερισμός σε 2-D blocks, που αποδείχτηκε πιο αποδοτικός από 1-D όπως δείξαμε.
2. Αυξήσαμε το μέγεθος των πινάκων «πριν» και «μετά» κατά 2 στήλες και 2 γραμμές για τα halo points για την αποστολή/λήψη ολόκληρων σειρών και στηλών και αντίστοιχη αποθήκευση τους

τοπικά.

Για την μείωση του αδρανούς χρόνου και overhead:

3. Εφαρμόσαμε επικάλυψη επικοινωνίας με υπολογισμούς. Πιο συγκεκριμένα,

- Αποστολές με Isend (non-blocking) της 1ης γραμμής στην άλω της διεργασίας στο «Βορρά» (Β), της τελευταίας γραμμής στην άλω της διεργασίας στο «Νότο» (Ν), της 1 ης στήλης στην άλω της διεργασίας της «Δύσης» (Δ) και της τελευταίας στήλης στην άλω της διεργασίας στην «Ανατολή» (Α). Επίσης αποστολή πράσινων γωνιακών στοιχείων στις διεργασίες ΒΑ, ΒΔ, ΝΑ, ΝΔ.
- Συμμετρικές λήψεις με Irecv (non-blocking) των γραμμών και στηλών της άλω της διεργασίας (κίτρινα) από τις γειτονικές Β, Ν, Δ, Α και γωνιακά σημεία.
- Ενώ γίνεται η non-blocking μεταβίβαση γραμμών, στηλών και γωνιακών στοιχείων, υπολογίζουμε τις νέες εσωτερικές τιμές που δεν εξαρτώνται από τα στοιχεία της άλω (λευκά).
- Ακολουθούν Wait για την ολοκλήρωση της λήψης γραμμών, στηλών και γωνιακών της άλω με τα αντίστοιχα Irecv.

Να σημειωθεί ότι κάναμε δύο υλοποιήσεις όπως ήδη αναφέραμε :

Μία με non-blocking Isend,Irecv και μια με Persistent Communication, MPI_Send_init, MPI_Start για να μην επαναϋπολογίζονται οι τιμές παραμέτρων των κλήσεων Isend, Irecv, όπως δηλαδή μας προτείνετε στο ***βήμα 8*** των βελτιώσεων. Παρατηρήσαμε σημαντική βελτίωση με persistent communication γι' αυτό και συνεχίσαμε τις μετρήσεις μας με αυτή την υλοποίηση. Παρόλα αυτά σας έχουμε στείλει και τις δύο υλοποιήσεις.

4. Πρώτα recv μετά send. Και για non-blocking και για persistent.
5. Χρήση datatypes στα Isend/Irecv(send-recv init αντίστοιχα) για αποστολή/λήψη σειρών και οπωσδήποτε στηλών, όπου έγινε με MPI_Type_create_subarray.
6. Υπολογισμός ranks των 8 σταθερών γειτόνων μια φορά έξω από κεντρική επανάληψη.
7. Τοποθέτηση διεργασιών που επικοινωνούν στον ίδιο κόμβο μέσω τοπολογιών διεργασιών Cartesian.
8. Όπως είπαμε προτιμήθηκε μείωση ακολουθιακών υπολογισμών στην κεντρική επανάληψη
9. Χρήση δεικτών, ούτως ώστε να κάνουμε απόδοση της τιμής δείκτη στον πίνακα «πριν» από τον «μετά» αποφεύγοντας την αντιγραφή τιμών πίνακα με διπλό for.
10. Μέσα στην κεντρική επανάληψη χρησιμοποιήσαμε ακολουθιακούς υπολογισμούς
11. Μετρήσεις με Compile με -O3

Για αποδοτικό υβριδικό κώδικα υλοποιήσαμε τις 12,13,14,15,16 από τις οδηγίες σας. Κάποιες λεπτομέρειες για μερικές από αυτές.

Στο 13 καλύτερος συνδυασμός ήταν οι 4 διεργασίες- 2 νήματα και με αυτόν συνεχίσαμε τις μετρήσεις για το hybrid MPI-OpenMP. Επίσης, το schedule(static,1) αποδείχτηκε πιο αποδοτικό, και προστέθηκε και collapse(2) για το for.

Στο 15 έγινε reduce και με το sum και με το diff. Προτιμήθηκε το sum.

Το 16 έγινε αλλά ήταν περιέργως μη αποδοτικό, γι' αυτό οι μετρήσεις έγιναν με ξεχωριστά parallel for πριν τις επαναλήψεις.

Λόγω της μη διαθεσιμότητας του GPU server, η υλοποίηση σε CUDA δοκιμάστηκε σε προσωπικό υπολογιστή. Συγκεκριμένα, χρησιμοποιήθηκε η nVidia κάρτα γραφικών GeForce GTX 1050 Ti, η οποία έχει μνήμη 4GB και είναι εφοδιασμένη με 768 CUDA cores. Αναλυτικά τα τεχνικά χαρακτηριστικά της:

GPU Engine Specs	
CUDA Cores	**768**
Graphics Clock (MHz)	**1290**
Processor Clock (MHz)	**1392**
Graphics Performance	**high-6747**
Memory Specs	
Memory Clock	**7 Gbps**
Standard Memory Config	**4 GB**
Memory Interface	**GDDR5**
Memory Interface Width	**128-bit**
Memory Bandwidth (GB/sec)	**112**
Feature Support	
Supported Technologies	**CUDA, 3D Vision, PhysX, NVIDIA G-SYNC™, Ansel**
Thermal and Power Specs	
Maximum GPU Temperature (in C)	**97**
Maximum Graphics Card Power (W)	**75**
Minimum System Power Requirement (W)	**300**

Για την υλοποίηση σε cuda εργαστήκαμε ως εξής: Καταρχάς, χρησιμοποιήσαμε 2 kernels halo_rows και halo_colums για να δημιουργήσουμε τις άλω γραμμές και στήλες στην μνήμη της GPU ή αλλιώς device memory αντιγράφοντας τις κατάλληλες γραμμές και στήλες της περιμέτρου. Έπειτα έχουμε τον βασικό kernel GOL που θα υλοποιεί ουσιαστικά το παιχνίδι. Χρησιμοποιούμε δισδιάστατα μεγέθη για το μέγεθος του μπλοκ και του πινάκα (blocksize kai gridsize). Στην main συνάρτηση υπολογίζεται ο νέος πίνακας καλώντας την GOL για κάθε γενιά (σύνολο γενεών 1000) αφού έχει γίνει η αντιγραφή των pointer των πινάκων από host σε device. Δηλώνουμε στον preprocessor το thread"(threads/block) όπου το χρησιμοποιούμε για την υπολογισμό των threads και των blocks. Στην περίπτωση που τρέχουμε με 4 threads/block ορίζουμε τα threads(ανά διάσταση) σε 2. Αντίστοιχα για 256 threads/block τα ορίζουμε στα 16 κ.ο.κ.

Να σημειωθεί ότι ανά 10 iterations ελέγχουμε αν δεν έχει γίνει κάποια αλλαγή ή ο νέος πίνακας είναι κενός. Αυτό γίνεται όπως και στο mpi-openmp με την μεταβλητή diff. Στην cuda όμως επειδή η device έχει τη δική της μνήμη πρέπει να δηλώσουμε device μεταβλητή και να αντιγράψουμε την μεταβλητή diff

που είναι στον host σε αυτή , και μετά από κάθε επανάληψη να αντιγράφουμε την τιμή που θα έχει πάρει πίσω στη μνήμη του host.

Για περαιτέρω βελτιστοποίηση της απόδοσης, δηλώσαμε ένα πίνακα στην μοιραζόμενη μνήμη, η οποία είναι προσβάσιμη από όλα τα threads μέσα σε ένα thread block, αλλά δεν είναι εμφανής στα άλλα blocks. Στην ουσία, μέσα σε κάθε block τα threads μοιράζονται data μέσω της μοιραζόμενης μνήμης. Με το caching αυτό των data σε μια fast on-chip memory ,όπως είναι η shared, αποφεύγουμε τα άσκοπα accesses στην global ή local μνήμη για data που έχουν ξαναχρησιμοποιηθεί. Χρησιμοποιώντας όμως shared μνήμη, πρέπει να είμαστε προσεκτικοί για να αποφύγουμε race conditions. Για να βεβαιωθούμε λοιπόν ότι θα έχουμε σωστά αποτελέσματα όταν παράλληλα threads συνεργάζονται , πρέπει να τα συγχρονίσουμε. Αυτό επιτυγχάνεται με την εντολή __syncthreads().

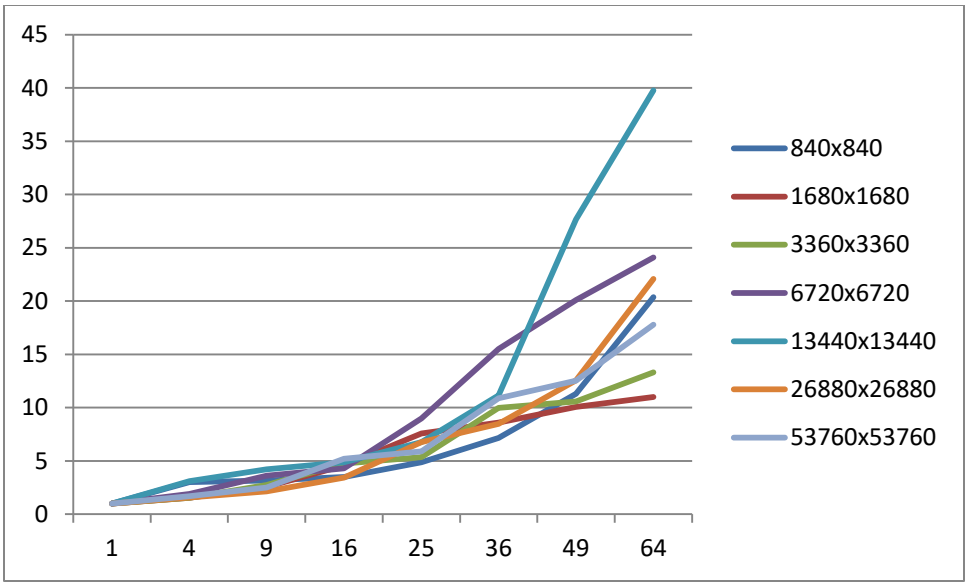
Μετρήσεις-Συμπεράσματα

MPI-Allreduce

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880	53760x53760
1	0.3545	0.7639	0.9498	1.8536	3.4301	6.8055	10.9584
4	0.1174	0.5003	0.6126	0.9709	1.1017	4.3016	6.5057
9	0.1120	0.2983	0.345	0.5115	0.8146	3.186	4.372
16	0.102	0.1674	0.2005	0.4307	0.7051	1.9794	2.1056
25	0.0729	0.1007	0.1776	0.206	0.5063	1.0062	1.8621
36	0.0496	0.089	0.0953	0.1197	0.3079	0.8018	1.0078
49	0.0314	0.076	0.0897	0.0922	0.124	0.5414	0.8743
64	0.0174	0.06952	0.0713	0.077	0.0863	0.3081	0.6159
80	fail	fail	fail	fail	fail	fail	fail

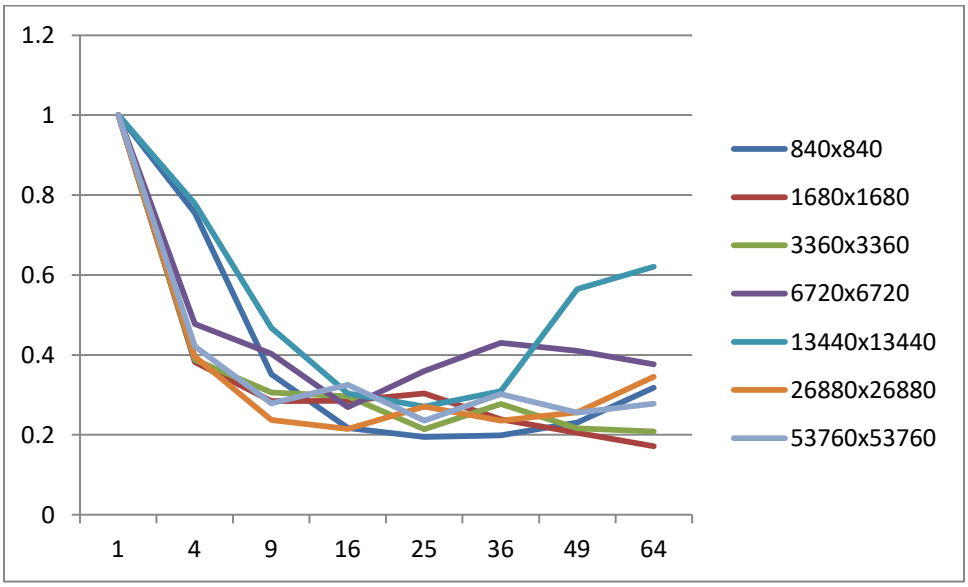
Speedup

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880	53760x53760
1	1	1	1	1	1	1	1
4	3.019591	1.526884	1.550441	1.909156	3.113461	1.582086	1.684431
9	3.165179	2.560845	2.753043	3.623851	4.210778	2.136064	2.506496
16	3.47549	4.563321	4.737157	4.303692	4.8647	3.438163	5.204407
25	4.862826	7.585899	5.347973	8.998058	6.774837	6.763566	5.884969
36	7.147177	8.583146	9.966422	15.48538	11.14031	8.487778	10.87359
49	11.28981	10.05132	10.58863	20.10412	27.6621	12.57019	12.53391
64	20.37356	10.9882	13.32118	24.07273	39.74623	22.08861	17.7925



Efficiency

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880	53760x53760
1	1	1	1	1	1	1	1
4	0.754898	0.381721	0.38761	0.477289	0.778365	0.395521	0.421108
9	0.351687	0.284538	0.305894	0.40265	0.467864	0.23734	0.2785
16	0.217218	0.285208	0.296072	0.268981	0.304044	0.214885	0.325275
25	0.194513	0.303436	0.213919	0.359922	0.270993	0.270543	0.235399
36	0.198533	0.238421	0.276845	0.430149	0.309453	0.235772	0.302044
49	0.230404	0.205129	0.216094	0.410288	0.564533	0.256534	0.255794
64	0.318337	0.171691	0.208143	0.376136	0.621035	0.345134	0.278008

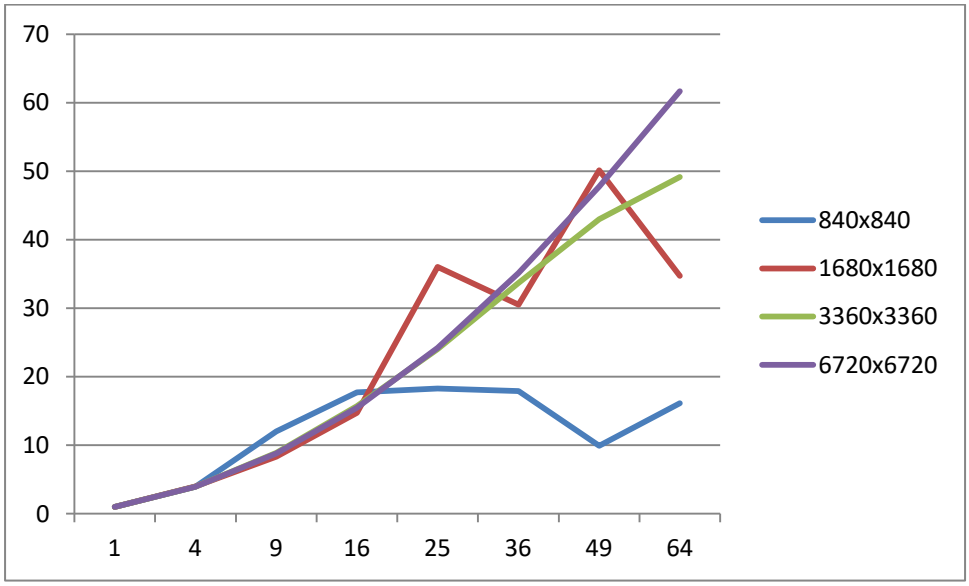


MPI χωρίς allreduce

	840x840	1680x1680	3360x3360	6720x6720
1	4.8273	19.2954	77.5028	309.103
4	1.2298	4.8562	19.5951	78.1121
9	0.4011	2.3127	8.7044	35.1535
16	0.2726	1.3113	4.938	20.0067
25	0.2645	0.5359	3.2289	12.7789
36	0.27	0.6321	2.3022	8.7955
49	0.4865	0.385	1.8041	6.4762
64	0.299	0.5557	1.5772	5.0121
80	fail	fail	fail	fail

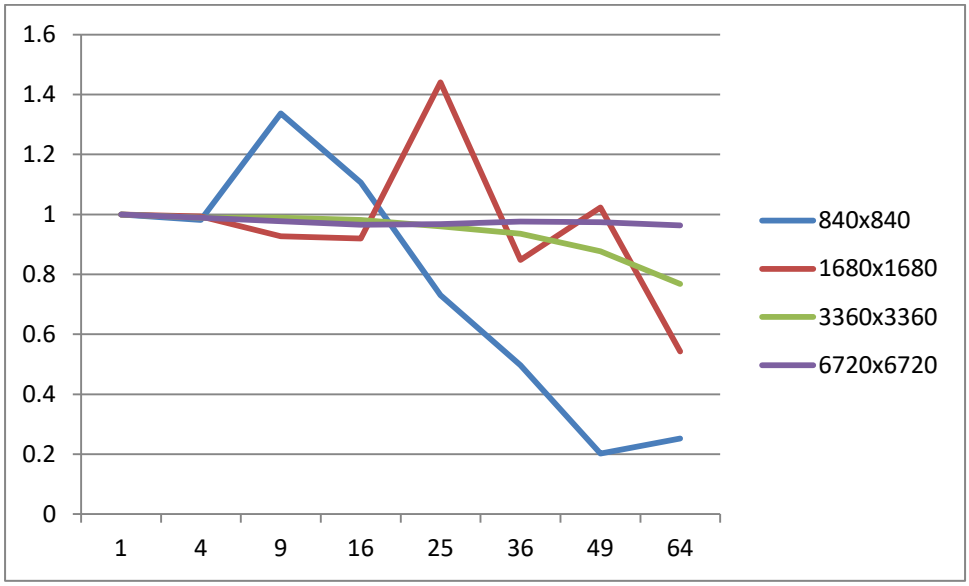
Speedup

	840x840	1680x1680	3360x3360	6720x6720
1	1	1	1	1
4	3.925272	3.9733537	3.9552133	3.9571718
9	12.03515	8.3432352	8.9038647	8.7929509
16	17.70836	14.714711	15.69518	15.449974
25	18.25066	36.005598	24.002849	24.188545
36	17.87889	30.525866	33.664669	35.143312
49	9.922508	50.117922	42.959259	47.72907
64	16.14482	34.722692	49.139488	61.671355



Efficiency

	840x840	1680x1680	3360x3360	6720x6720
1	1	1	1	1
4	0.981318	0.993338	0.988803	0.989293
9	1.337239	0.927026	0.989318	0.976995
16	1.106773	0.919669	0.980949	0.965623
25	0.730026	1.440224	0.960114	0.967542
36	0.496636	0.847941	0.93513	0.976203
49	0.2025	1.022815	0.87672	0.974063
64	0.252263	0.542542	0.767805	0.963615



Συγκρίνοντας τις αποδόσεις του προγράμματος MPI με τοπική επικοινωνία με αυτές του προγράμματος με global επικοινωνία, που επιτυγχάνεται με τη χρήση allreduce, παρατηρούμε ότι υπάρχει σαφής διαφορά των χρόνων εκτέλεσης για όλους τους συνδυασμούς αριθμού process και διαστάσεων πλέγματος. Πιο συγκεκριμένα, φαίνεται ξεκάθαρα η υπεροχή της global επικοινωνίας έναντι της τοπικής, γεγονός αναμενόμενο καθώς ελαχιστοποιείται τόσο ο χρόνος επικοινωνίας μεταξύ των process όσο και οι απαιτούμενοι έλεγχοι για τη σωστή λειτουργία του προγράμματος.

Παρατηρώντας, ωστόσο, την κλιμάκωση στις δύο υλοποιήσεις, αυτή είναι εξίσου εμφανής. Τα αποτελέσματα φαίνονται στις υπολογισμένες τιμές του speedup και για τις δύο περιπτώσεις. Αυτό που μπορούμε να συμπεράνουμε είναι ότι η κλιμάκωση είναι καλύτερη όσο αυξάνεται ο αριθμός των process, καθώς ο διαμοιρασμός των δεδομένων γίνεται πιο αποτελεσματικά και μπορεί να γίνει πλήρης χρήση της παραλληλίας των προγραμμάτων. Τα σημεία στα οποία το speedup δεν είναι τόσο καλό σε σχέση με τον αριθμό των process μπορούν επίσης να αιτιολογηθούν καθώς όσο περισσότερους worker χρησιμοποιεί το πρόγραμμα, τόσο μεγαλύτερο communication overhead έχουμε (dividing work and aggregating results back). Τελικά, ύπαρχει ένα «sweet spot», κατά το οποίο θέλουμε αρκετούς worker για την

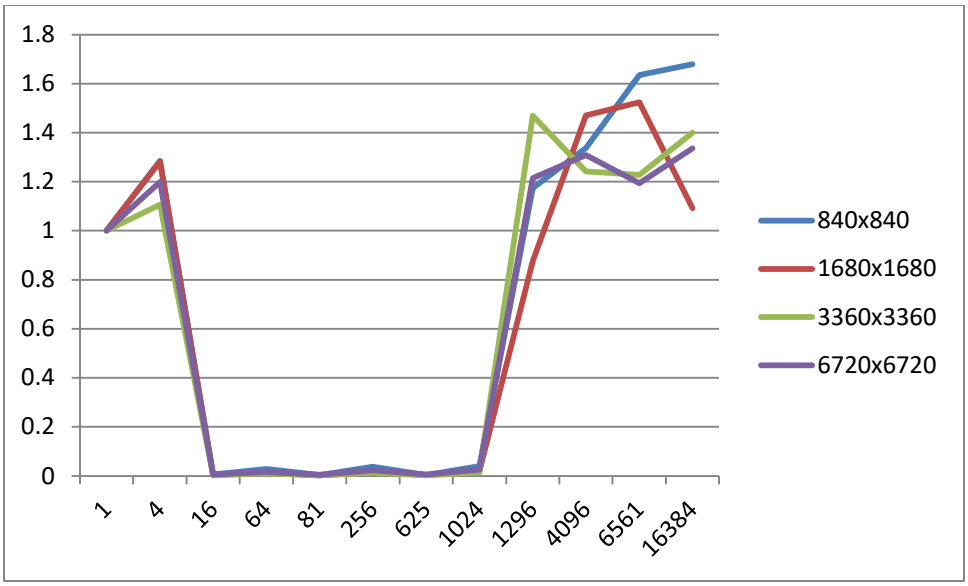
παραλληλοποίηση του προβλήματος αλλά όχι τόσους πολλούς ώστε το communication overhead να εμποδίζει την κλιμάκωση.

Cuda

	840x840	1680x1680	3360x3360	6720x6720
1	0.000875	0.001534	0.004409	0.014971
4	0.000681	0.001196	0.003984	0.012483
16	0.131844	0.482274	1.59629	2.456778
64	0.031194	0.123109	0.44654	0.876144
81	0.225371	0.755984	2.690596	3.246591
256	0.023116	0.090789	0.345533	0.653454
625	0.224752	0.513193	1.670535	2.275692
1024	0.021605	0.084947	0.323414	0.597341
1296	0.000745	0.001751	0.003001	0.012326
4096	0.000654	0.001043	0.00355	0.011444
6561	0.000535	0.001007	0.003593	0.012546
16384	0.000521	0.001404	0.003149	0.011207

Speedup

	840x840	1680x1680	3360x3360	6720x6720
1	1	1	1	1
4	1.284875	1.282609	1.106677	1.199311
16	0.006637	0.003181	0.002762	0.006094
64	0.02805	0.012461	0.009874	0.017087
81	0.003882	0.002029	0.001639	0.004611
256	0.037853	0.016896	0.01276	0.022911
625	0.003893	0.002989	0.002639	0.006579
1024	0.0405	0.018058	0.013633	0.025063
1296	1.174497	0.876071	1.469177	1.214587
4096	1.33792	1.470757	1.241972	1.308196
6561	1.635514	1.523337	1.227108	1.193289
16384	1.679463	1.092593	1.400127	1.335862

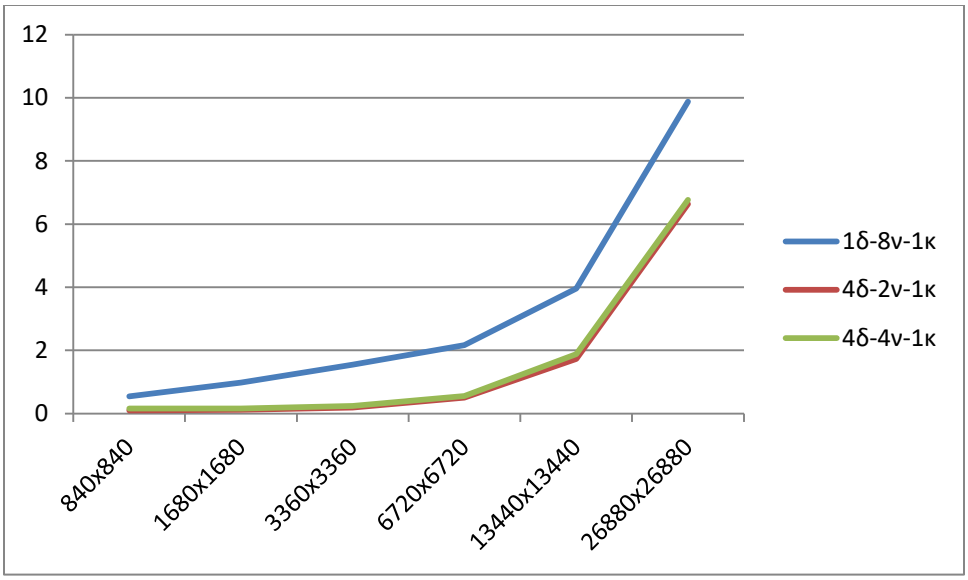


Μελετώντας τα αποτελέσματα των μετρήσεων παρατηρούμε ότι οι χρόνοι που επιτυγχάνονται είναι πολύ καλοί. Ωστόσο, βλέπουμε ότι το speedup για κάποιος αριθμούς process δεν είναι ικανοποιητικό. Αυτό συμβαίνει λόγω όχι τόσο αποτελεσματικού συνδυασμού αριθμού process και thread, καθώς φαίνεται ότι είναι ανεξάρτητο των διαστάσεων του πλέγματος.

Το efficiency στην περίπτωση του cuda δεν υπολογίστηκε, καθώς δε γνωρίζουμε ακριβώς πόσα process είναι σε χρήση κάθε φορά, παρότι γνωρίζουμε τον αριθμό των core.

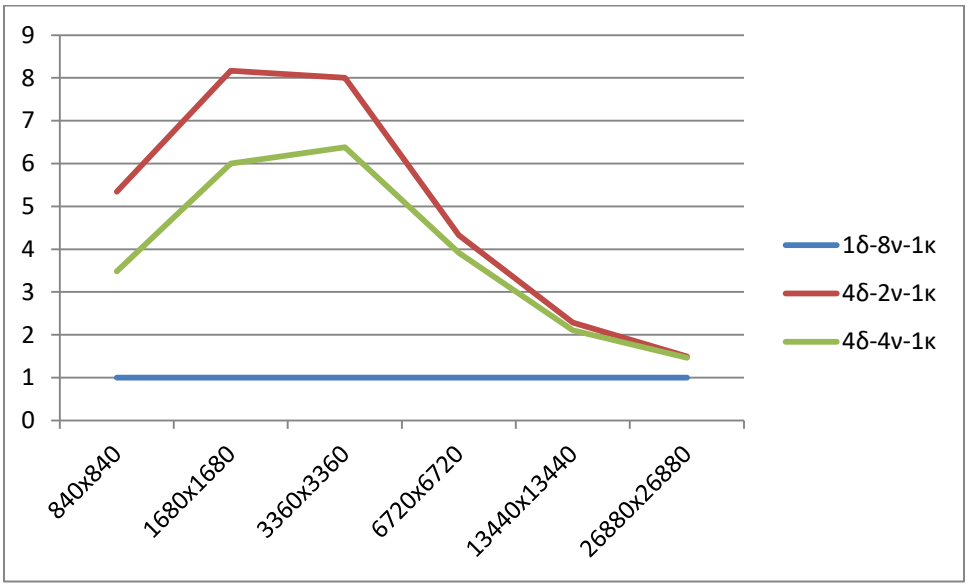
OpenMP

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880
1δ-8v-1κ	0.5436	0.9782	1.548	2.1667	3.9601	9.8767
4δ-2v-1κ	0.1018	0.1198	0.1934	0.5008	1.7327	6.6321
4δ-4v-1κ	0.1561	0.163	0.2426	0.554	1.8785	6.767



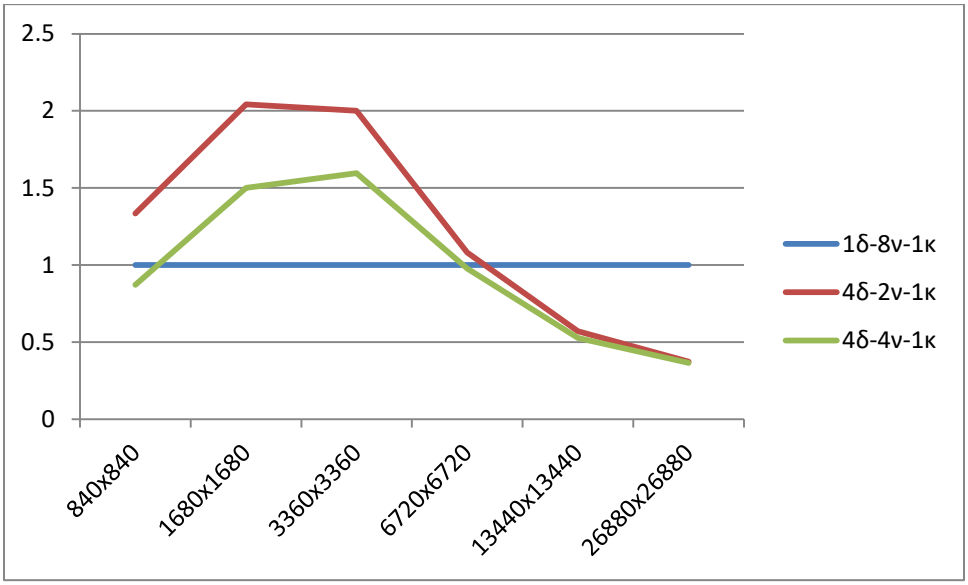
Speedup

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880
1δ-8v-1κ	1	1	1	1	1	1
4δ-2v-1κ	5.339882	8.165275	8.004137	4.326478	2.285508	1.489227
4δ-4v-1κ	3.482383	6.001227	6.380874	3.911011	2.108118	1.459539



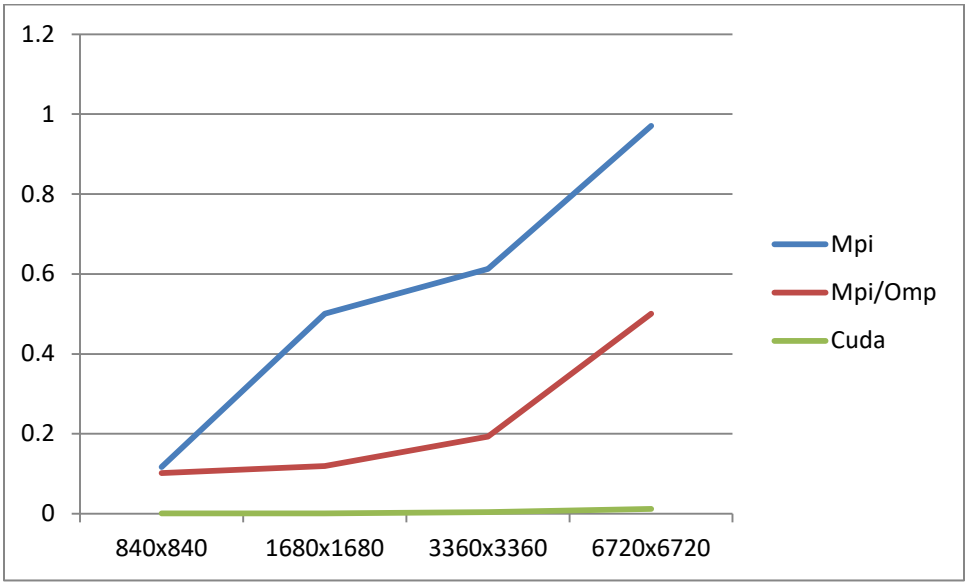
Efficiency

	840x840	1680x1680	3360x3360	6720x6720	13440x13440	26880x26880
1δ-8v-1κ	1	1	1	1	1	1
4δ-2v-1κ	1.334971	2.041319	2.001034	1.081619	0.571377	0.372307
4δ-4v-1κ	0.870596	1.500307	1.595218	0.977753	0.52703	0.364885



Παρατηρώντας τα αποτελέσματα συμπεραίνουμε ότι οι χρόνοι που επιτυγχάνονται είναι καλοί. Η αύξηση διεργασιών υπερισχύει έναντι των πολλών νημάτων, με αποτέλεσμα να έχουμε μεγαλύτερο speedup για περισσότερες διεργασίες με λιγότερα νήματα. Σίγουρα ρόλο σ' αυτό παίζει η επικοινωνία μεταξύ των νημάτων, αλλά και η πιθανή συμφόρηση του συστήματος λόγω της ύπαρξης περισσότερων χρηστών. Επιπλέον, βλέπουμε ότι καθώς οι διαστάσεις του πλέγματος αυξάνονται η κλιμάκωση δεν συνεχίζεται πράγμα που υπονοεί κάποιο θέμα στο διαμοιρασμό των δεδομένων.

Σύγκριση MPI-MPI/OpenMP-Cuda



Συγκρίνουμε τις τρεις υλοποιήσεις για τέσσερα process, καθώς για αυτόν τον αριθμό έχουμε δεδομένα απ' όλες. Το υβριδικό πρόγραμμα επιτυγχάνει καλύτερους χρόνους από την απλή MPI υλοποίηση, κάτι αναμενόμενο, καθώς πλέον γίνεται χρήση thread για τον ίδιο αριθμό process. Τέλος, παρατηρούμε ότι το cuda επιτυγχάνει ακόμα καλύτερους χρόνους από το MPI/Omp, καθώς στην περίπτωση του cuda γίνεται χρήση πολύ περισσότερων thread (μερικών εκατοντάδων), έναντι των δύο και των τεσσάρων που δοκιμάστηκαν στο υβριδικό πρόγραμμα. Επιπλέον, σημαντικό ρόλο έπαιξε η μνήμη της κάρτας γραφικών, που είναι κοινή για όλα τα thread, γεγονός που ελαττώνει το χρόνο επικοινωνίας μεταξύ τους.