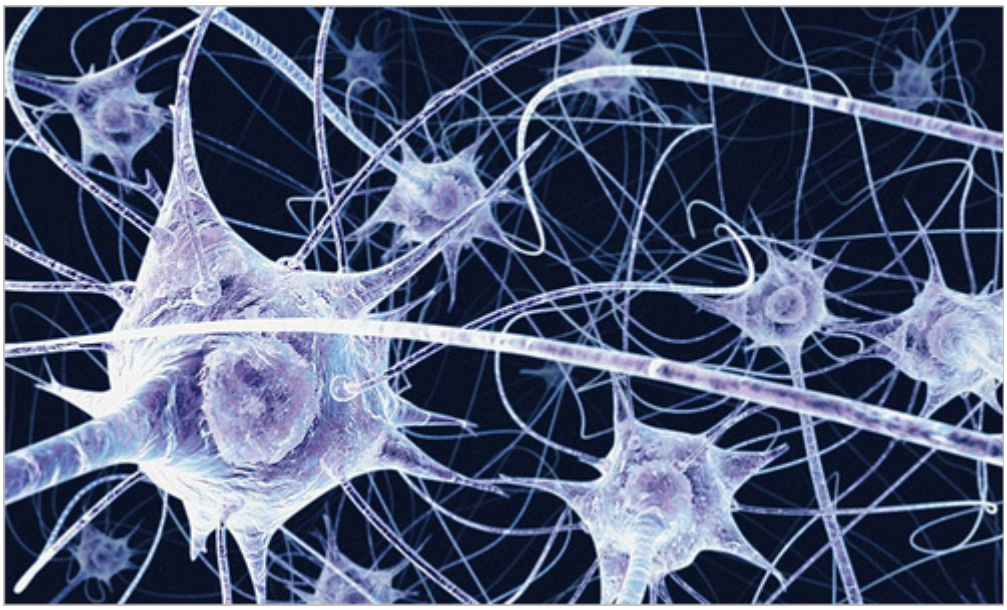
Εθνικό και Καποδιστριακό
ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ

Εαρινό Εξάμηνο 2013 – 2014



Μέλη Ομάδας

Μιχαηλίδης Θεόδωρος, Α.Μ.: 1115201000068
Μπαμίδης Κωνσταντίνος, Α.Μ.: 1115201000087

1. Εισαγωγή

Στην εργασία ασχοληθήκαμε με τον παράλληλο προγραμματισμό με MPI,OpenMP και CUDA. Αρχικά μας δόθηκε ένα πρόγραμμα που ήταν υλοποιημένο σε MPI,στο οποίο κάναμε όσες βελτιώσεις μπορούσαμε ώστε να επιτύχουμε καλύτερους χρόνους εκτέλεσης. Παρακάτω παρουσιάζουμε τις επιλογές υλοποίησης που κάναμε και αντιπαραβάλουμε τις μετρήσεις απόδοσης που κάναμε για κάθε μια από τις 3 τεχνικές παράλληλου προγραμματισμού.

2. Γενικός Σχεδιασμός και Υλοποίηση

- **MPI:** Όσον αφορά τη δική μας υλοποίηση του προβλήματος σε MPI χωρίσαμε τα στοιχεία του δοθέντος πίνακα ανάμεσα στις διεργασίες ισόποσα με χρήση τετραγωνικών blocks. Κάνοντας την πράξη NXPROB x NYPROB / numtasks βρίσκουμε τον αριθμό των στοιχείων για τα οποία είναι υπεύθυνη η κάθε διεργασία. Η τετραγωνική ρίζα αυτού του αριθμού (μιας και τα blocks, όπως είπαμε, είναι τετράγωνα) μας δίνει τις διαστάσεις X και Y που θα χρησιμοποιήσει η κάθε διεργασία για τους πίνακες της. Έπειτα, αφού η κάθε διεργασία έχει αρχικοποιήσει τον αρχικό της πίνακα μέσω της συνάρτησης **inidat**, ελέγχουμε με διαδοχικές if την αποστολή/λήψη των δεδομένων μεταξύ των διεργασιών. Αυτό είναι απαιραίτητο, αφού, άμα φανταστούμε ένα πλέγμα διεργασιών, βλέπουμε ότι κάποιες διεργασίες (πχ αυτές που βρίσκονται στις 4 γωνίες αυτού του πλέγματος) δεν έχουν κάποια γειτονική διεργασία στο πάνω όριο τους, κάποιες στο κάτω κ.ο.κ. ώστε να τους στείλουν ή να λάβουν δεδομένα από αυτές. Η επικοινωνία μεταξύ των διεργασιών είναι **non-blocking**, χρησιμοποιώντας τις συναρτήσεις **MPI_Isend**, **MPI_Irecv** και **MPI_Wait**, όπου αυτή χρειάζεται. Η αποστολή και λήψη των δεδομένων γίνεται με τη χρήση datatypes. Οι γραμμές των πινάκων στέλνονται και λαμβάνονται ως **MPI_Type_contiguous** και οι στήλες ως **MPI_Type_vector**. Στη συνέχεια, πρέπει να καλέσουμε την συνάρτηση **update**. Εδώ, πρέπει να αναφέρουμε ότι χωρίσαμε την λειτουργικότητα της **update** σε 2 συναρτήσεις: στην **inner_update**, η οποία επενεργεί στα εσωτερικά δεδομένα των πινάκων της κάθε διεργασίας, που δεν εξαρτώνται από τα δεδομένα των γειτονικών διεργασιών, και στην **outer_update**, η οποία επενεργεί στα δεδομένα της 1ης και της τελευταίας γραμμής και στήλης, που, αντίθετα, εξαρτώνται από τα δεδομένα των γειτονικών

διεργασιών. Αυτή η πρακτική είναι πολύ χρήσιμη, αφού καλούμε την **inner_update** πριν τις κλήσεις των **MPI_Wait**, που είναι απαραίτητες για τη σωστή επικοινωνία μεταξύ των διεργασιών, και έτσι εκμεταλλευόμαστε την ταχύτητα της **non-blocking** επικοινωνία εκτελώντας, ταυτόχρονα, χρήσιμο έργο. Αφού, πλέον, έχουν εκτελεστεί και οι **MPI_Wait**, είμαστε σίγουροι ότι η αποστολή/λήψη δεδομένων μεταξύ των διεργασιών έχουν γίνει σωστά, άρα μπορούμε να καλέσουμε την συνάρτηση **outer_update**, της οποίας η λειτουργικότητα έχει αναφερθεί παραπάνω. Τέλος, σημειώνουμε ότι δεν ακολουθήσαμε το master/slave πρότυπο του αρχικού προγράμματος,αλλά προτιμήσαμε να αξιοποιήσουμε όλες τις διεργασίες,οι οποίες έχουν τον ίδιο φόρτο εργασίας.

- **OpenMP:** Για το OpenMP χρησιμοποιήσαμε το δικό μας MPI πρόγραμμα,αλλάζοντας μόνο τη συνάρτηση update που υπολογίζει σε κάθε step τα δεδομένα του πίνακα. Αρχικά,ο χρήστης επιλέγει τον αριθμό των threads δίνοντας το ως πρώτο όρισμα στο πρόγραμμα ως παράμετρο. Δοκιμάσαμε 3 διαφορετικές προσεγγίσεις για τον διαμοιρασμό των επαναλήψεων της update στα διαθέσιμα threads: απλή παραλληλοποίηση με τη χρήση της **#pragma parallel omp for**, δυναμική παραλληλοποίηση με τη χρήση της **#pragma parallel omp for schedule(dynamic,1)**, και τέλος στατική παραλληλοποίηση με τη χρήση της **#pragma parallel omp for schedule(static,1)**. Επίσης, στις περιπτώσεις που έπρεπε, απλή και στατική παραλληλοποίηση, απενεργοποιήσαμε το δυναμικό ορισμό των threads μέσω της συνάρτησης **omp_set_dynamic()** και θέτοντας την σε 0. Επιπλέον, χρησιμοποιώντας τη συνάρτηση **omp_set_nested()** και θέτοντας τη σε 1, επιτρέπουμε την υλοποίηση nested for και τη σωστή αξιοποίησή τους. Όσον αφορά τα threads, χρησιμοποιήσαμε τη συνάρτηση **MPI_Init_thread()** με level of thread support **MPI_THREAD_MULTIPLE** ώστε όλα τα threads να μπορούν να κάνουν MPI calls. Για την **update**, μετά από πολλές δοκιμές στην υλοποίηση και παραλλαγές αυτών καταλήξαμε στο -φαινομενικά- βέλτιστο χρησιμοποιώντας το δυναμικό scheduling ως προς τον αριθμό των threads.

- **CUDA:** Για την υλοποίηση του προγράμματος σε CUDA μετατρέψαμε το MPI πρόγραμμά μας σε σειριακό C++ πρόγραμμα (για λόγους συμβατότητας με το CUDA) και υλοποιήσαμε την **update** σε ξεχωριστό .cu αρχείο. Αρχικά, δίνουμε από το C++ πρόγραμμα στο

CUDA πρόγραμμα τους 2 πίνακες που χρειαζόμαστε για την **update**. Στη συνέχεια δεσμεύουμε χώρο για τους 2 μονοδιάστατους arrays που θα χρησιμοποιηθούμε από την global function/kernel και περνάμε με **memcpy** τα στοιχεία του ενός από τους 2 αρχικούς πίνακες σε έναν από τους arrays της GPU (ο άλλος αρχικός περιέχει μηδενικά οπότε δεν είχε νόημα να τον αντιγράψουμε). Στο τέλος κάνουμε το αντίστροφο. Όσον αφορά τα blocks και τα threads που χρησιμοποιούνται από τη GPU καταλήξαμε, μετά από δοκιμές, να χρησιμοποιήσουμε αντίστοιχα το NXPROBLEM και το NYPROBLEM των αρχικών πινάκων (όπου NXPROBLEM ο αριθμός των γραμμών και NYPROBLEM ο αριθμός των στηλών των αρχικών πινάκων). Τέλος,επιστρέφουμε τον πίνακα με τα τελικά αποτελέσματα στο C++ πρόγραμμα για εκτύπωση.

3. Μετρήσεις Απόδοσης – Μελέτη Κλιμάκωσης

Για να κάνουμε τις μετρήσεις στα 3 προγράμματα χρησιμοποιήσαμε:

- **MPI_Wtime()** στο MPI και στο MPI+OMP,διότι μας ενδιέφερε το wall-clock time και όχι το CPU time. (το οποίο θα παίρναμε με τις περισσότερες συναρτήσεις της C).
- **CudaEvents** στο CUDA ώστε να πάρουμε με καλύτερη ακρίβεια το συνολικό χρόνο που χρειάχονται η CPU και η GPU να εκτελέσουν το κομμάτι που τους αντιστοιχεί. Ενδείκνυται ιδιαίτερα σε περιπτώσεις χρήσης ασύγχρονου kernel.

Οι μετρήσεις για το MPI και το OpenMP έγιναν σε 19 μηχανήματα linux της σχολής (ακριβώς όσα ήταν διαθέσιμα εκείνη τη χρονική στιγμή) και οι μετρήσεις του CUDA πραγματοποιήθηκαν σε local μηχάνημα σε κάρτα nVidia GeForce GTX 460 (όπως και η υλοποίηση του CUDA). Για τις συγκρίσεις χρησιμοποιήσαμε διαφορετικά μεγέθη προβλήματος, διαφορετικό αριθμό διεργασιών, και, σε κάποιες περιπτώσεις διαφορετικό αριθμό steps. Τέλος, προσθέτουμε πως αν και προσπαθήσαμε να είμαστε όσο περισσότερο ακριβείς μπορούσαμε, οι μετρήσεις δεν είναι 100% αντιπροσωπευτικές, διότι τα μηχανήματα στα οποία παίρναμε τους χρόνους ήταν δεσμευμένα και από άλλους συναδέλφους.

Αρχικό (δοθέν) πρόγραμμα – mpi_heat2D:

Οι χρόνοι εκτέλεσης του αρχικού προγράμματος που μας δόθηκε, φαίνονται στους παρακάτω πίνακες. Οι δοκιμές μας είναι για 4 και 9 διεργασίες και για 3 διαφορετικά μεγέθη λόγω των περιορισμών που θέτει το ίδιο το πρόγραμμα. Επίσης, μετρήσαμε το χρόνο του προγράμματος για 3 διαφορετικά μεγέθη των STEPS: 100, 400, 1000.

Για STEPS = 100

processes\size	240 x 240	600 x 600	960 x 960
4	0.1184	0.5840	1.4273
9	0.1024	0.4322	1.0100

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Για STEPS = 400

processes\size	240 x 240	600 x 600	960 x 960
4	0.3706	1.7900	4.3283
9	0.2500	0.9750	2.2310

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

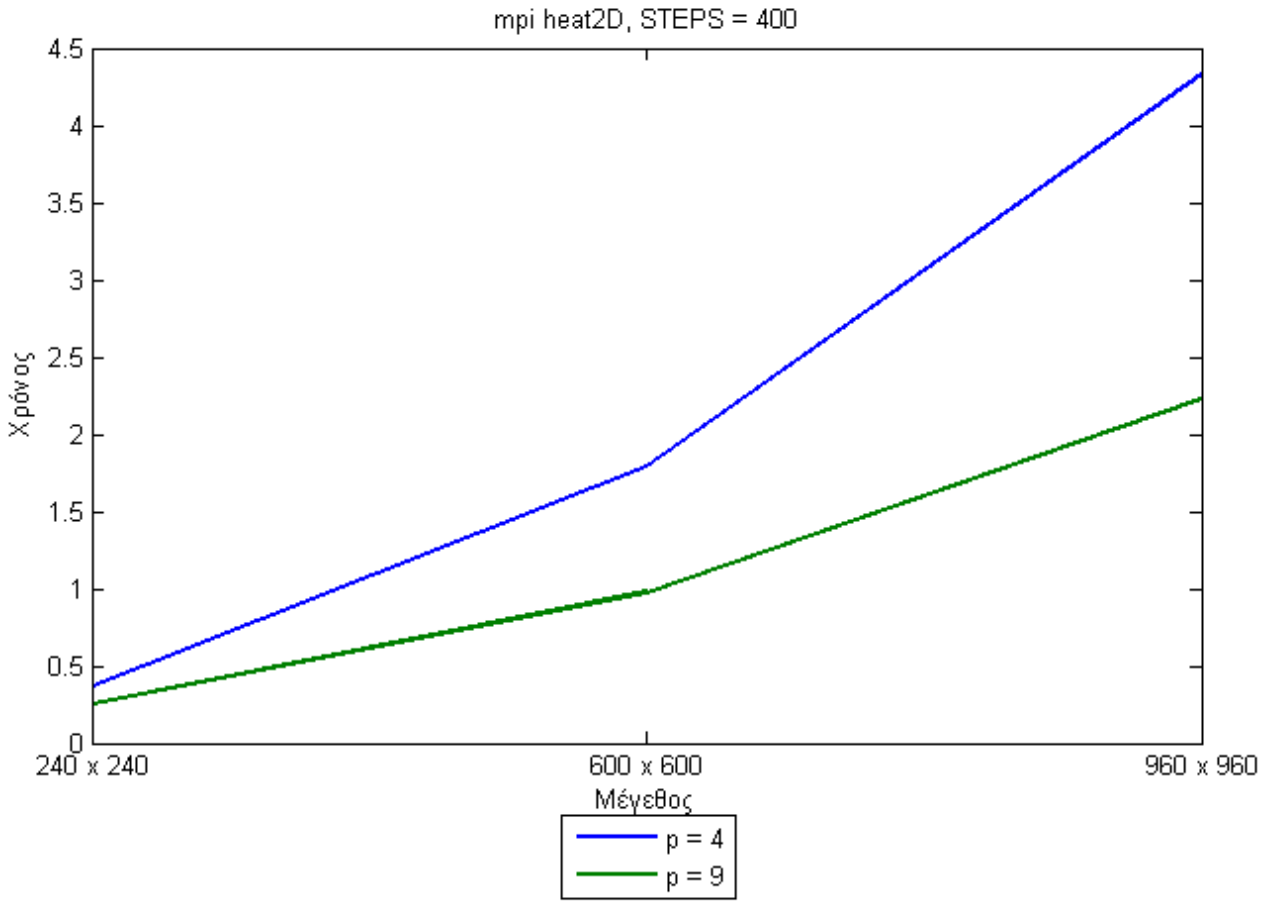
Για STEPS = 1000

processes\size	240 x 240	600 x 600	960 x 960
4	0.8755	4.2280	10.1280
9	0.5562	2.060	4.6858

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Όπως βλέπουμε από τα παραπάνω αποτελέσματα, είναι προφανές ότι όσο μεγαλώνουμε το μέγεθος του προβλήματος, τόσο μεγαλώνουν και οι χρόνοι εκτέλεσης. Ταυτόχρονα όμως βλέπουμε ότι μεγαλώνει και το utilization των παραπάνω διεργασιών.

Στο παρακάτω διάγραμμα βλέπουμε, ενδεικτικά για αριθμό STEPS = 400, πως αναπαρίσταται η συμπεριφορά του προγράμματος.



Τροποποιημένο MPI πρόγραμμα - MPI_Opt:

Για το τροποποιημένο MPI πρόγραμμα κάναμε δοκιμές με αρκετά διαφορετικά μεγέθη προβλήματος, όπως και για διαφορετικό αριθμό διεργασιών. Φτάσαμε μέχρι τις 36 διεργασίες, επειδή το πλήθος των υπολογιστών στο cluster είναι 19 και ο καθένας μπορεί να τρέχει μέχρι 2 διεργασίες ταυτόχρονα. Επίσης, όπως και πριν, μετρήσαμε το χρόνο του προγράμματος για 3 διαφορετικά μεγέθη των STEPS: 100, 400, 1000. Τα αποτελέσματα φαίνονται παρακάτω.

Για STEPS = 100

p\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
1	0.1600	1.009	2.548	5.7632	10.2559	25.2977	36.0346
4	0.0529	0.3231	0.6788	1.4984	2.6107	5.9308	9.1739
9	0.0445	0.1348	0.3896	0.6845	1.1946	2.6503	4.1337
16	0.0492	0.1369	0.2980	0.5601	0.7654	1.5483	2.3988
25	0.0600	0.1303	0.2566	0.3799	0.5453	1.0692	1.5890
36	0.1119	0.1957	0.2202	0.3090	0.4223	0.8509	1.1601

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Για STEPS = 400

p\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
1	0.6260	3.9719	10.2756	23.1779	41.3431	101.8220	144.1457
4	0.1922	1.0528	2.6933	5.9552	10.5174	23.5017	36.9946
9	0.1783	0.5673	1.2953	2.7978	4.8515	10.5529	16.5286
16	0.1237	0.4461	0.7916	1.8378	2.8508	6.0068	9.3405
25	0.1702	0.3627	0.6355	1.0980	1.9628	4.0425	6.1773
36	0.1793	0.3942	0.4747	0.8083	1.4631	2.9438	4.0424

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

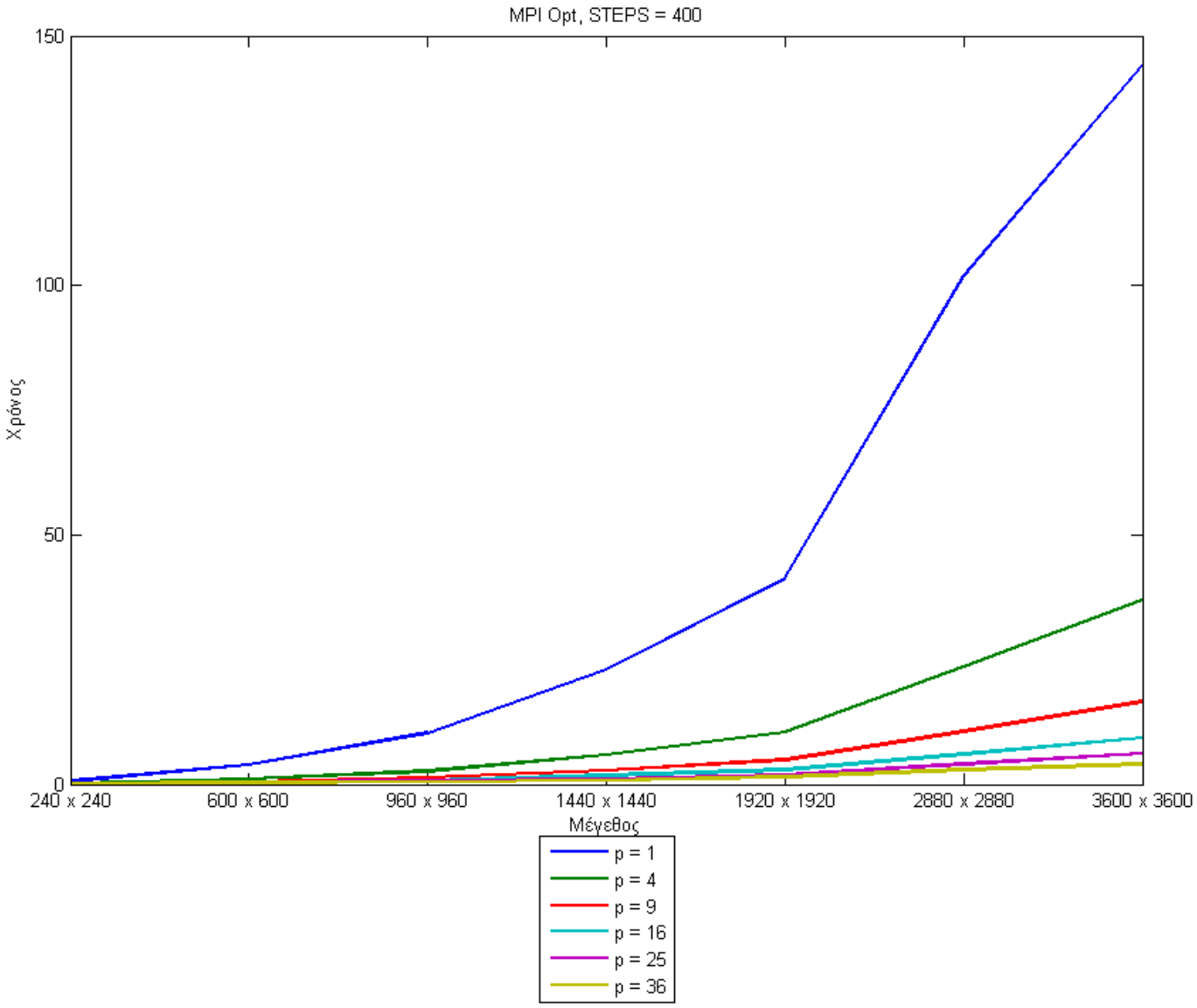
Για STEPS = 1000

p\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
1	1.5698	10.0129	25.6338	57.9609	102.6096	252.4966	360.4700
4	0.4654	2.6168	6.7240	14.9186	26.3535	59.3436	92.5613
9	0.4201	1.2735	3.0751	6.9296	12.0272	26.5529	41.4309
16	0.3122	0.9300	1.8360	4.6407	7.0345	15.0738	23.3165
25	0.3210	0.7028	1.2663	2.6977	4.7863	9.9697	15.2245
36	0.4009	0.6901	1.0679	2.0500	3.2789	7.1626	10.0753

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Όπως παρατηρούμε από τους παραπάνω πίνακες, η μετάβαση από 1 διεργασία σε περισσότερες, ειδικά σε μεγάλα μεγέθη προβλήματος, δίνει θεαματικά καλύτερα αποτελέσματα. Αυτό δείχνει ότι το πρόγραμμα μας κλιμακώνει αρκετά ικανοποιητικά. Υπάρχει, όμως, και μια ακόμα αρκετά ενδιαφέρουσα παρατήρηση στα αποτελέσματα των εκτελέσεων. Βλέπουμε ότι για μικρά μεγέθη προβλήματος (πχ 240 x 240, 600 x 600) όσο αυξάνουμε τον αριθμό των διεργασιών τόσο η μείωση στο χρόνο εκτέλεσης ελαττώνεται, μέχρι που φτάνουμε στο σημείο ο χρόνος εκτέλεσης να έχει αυξηθεί όταν έχουμε 36 διεργασίες. Αυτό συμβαίνει επειδή πλέον τα εσωτερικά δεδομένα, που είναι ανεξάρτητα της επικοινωνίας, έχουν μειωθεί αρκετά, και αντίστοιχα τα δεδομένα που εξαρτώνται από την επικοινωνία των διεργασιών έχουν αυξηθεί. Έτσι, το κόστος επικοινωνίας που "πληρώνουμε" για την αποστολή/λήψη δεδομένων μεταξύ των διεργασιών αυξάνεται, δημιουργώντας αντίστοιχη αύξηση στο χρόνο εκτέλεσης.

Παρακάτω, παραθέτουμε ένα διάγραμμα που απεικονίζει τους χρόνους για αριθμό STEPS = 400 σε σύστημα αξόνων. Σε αυτό μπορούμε καλύτερα να καταλάβουμε την κλιμάκωση του προγράμματος.


MPI Opt, STEPS = 400
150
100
50
0
Χρόνος
240 x 240
600 x 600
960 x 960
1440 x 1440
1920 x 1920
2880 x 2880
3600 x 3600
Μέγεθος
p = 1
p = 4
p = 9
p = 16
p = 25
p = 36

Τροποποιημένο MPI & OMP πρόγραμμα:

Για το πρόγραμμα αυτό επιλέξαμε κατά βάση να κρατήσουμε σταθερό τον αριθμό των STEPS και ίσο με 400 ώστε να δείξουμε περισσότερο τις διαφορές που παρουσίαζε ο διαφορετικός αριθμός threads και διεργασιών σε κάθε εκτέλεση για ίδια δεδομένα. Επίσης, επιλέξαμε να, όπως αναφέραμε και στο 1ο Κεφάλαιο αυτής της εργασίας, να χρησιμοποιήσουμε δυναμικό διαμοιρασμό των επαναλήψεων της **update** στα διαθέσιμα thread, αφού θεωρούμε ότι είναι η πιο αποτελεσματική τακτική. Παρακάτω, παραθέτουμε, ενδεικτικά, πίνακες με χρόνους εκτέλεσης και για τις άλλες 2 πρακτικές που δοκιμάσαμε, το στατικό διαμοιρασμό και την απλή παραλληλοποίηση.

Δυναμικό scheduling

Για PROCESSES = 4

t\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
2	0.1852	1.0490	2.7099	5.9519	10.5141	23.5626	36.9765
4	0.1884	1.0502	2.7016	5.9678	10.5417	23.6022	36.9723
8	0.1867	1.0443	2.7016	5.9654	10.5289	23.6013	36.8017
16	0.2281	1.0461	2.7061	5.9581	10.5050	23.5867	36.8662

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Για PROCESSES = 16

t\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
2	0.1323	0.5298	0.8056	1.9612	2.9801	6.3508	9.7716
4	0.1888	0.4878	0.8767	1.8695	3.0159	6.2857	9.7934
8	0.2129	0.3787	0.8640	1.9443	2.9676	6.2289	9.5112
16	0.1886	0.3815	0.8100	1.8305	2.8784	6.0949	9.3892

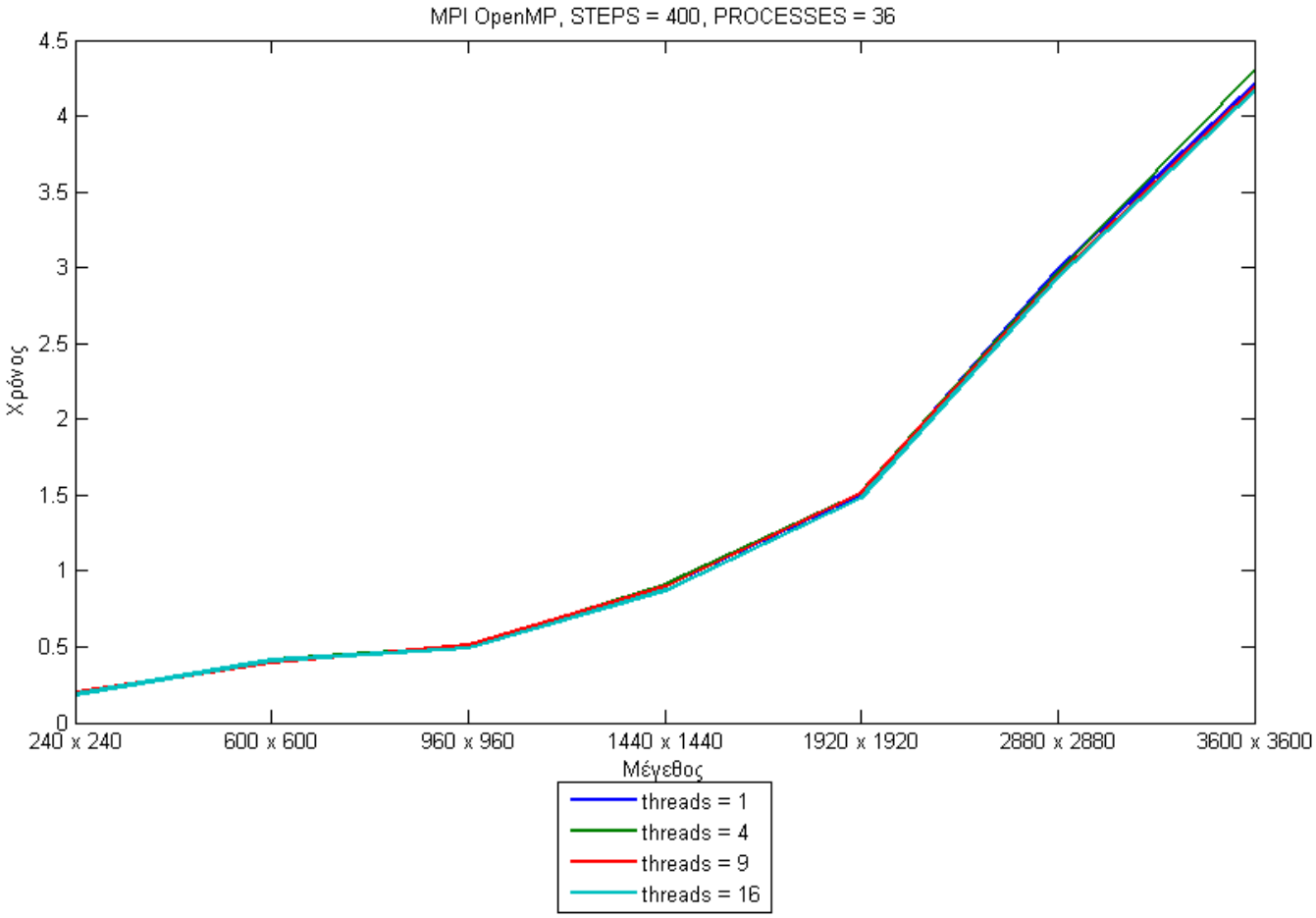
οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Για PROCESSES = 36

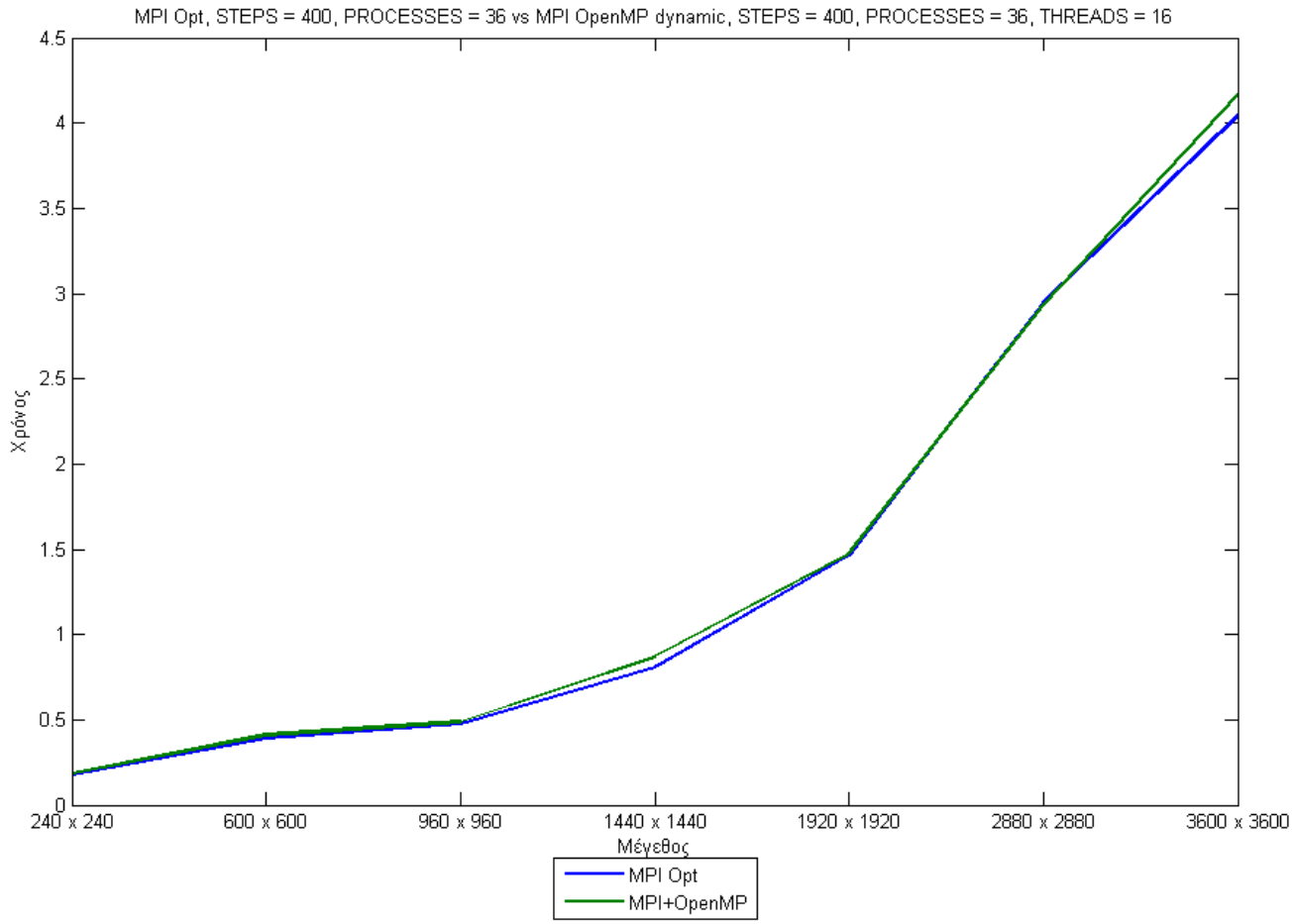
t\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
2	0.1867	0.4155	0.4986	0.9002	1.4993	2.9812	4.2066
4	0.1923	0.4091	0.5031	0.9084	1.5156	2.9677	4.3005
8	0.1972	0.3948	0.5132	0.8965	1.5102	2.9430	4.1886
16	0.1848	0.4099	0.4900	0.8733	1.4779	2.9316	4.1669

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Η κύρια παρατήρηση που έχουμε, όσον αφορά το πρόγραμμα που χρησιμοποιεί MPI και OpenMP μαζί, είναι ότι όπως φαίνεται στον πίνακα η διαφορές στους χρόνους των εκτελέσεων με διαφορετικό αριθμό threads είναι ελάχιστες. Μάλιστα, σε κάποιες περιπτώσεις βλέπουμε ότι αν αυξήσουμε τον αριθμό των threads, η απόδοση του προγράμματος είναι χειρότερη. Το παρακάτω διάγραμμα παρουσιάζει καλύτερα αυτό το γεγονός.



Μια, ακόμα, όμως παρατήρηση που έχουμε είναι ότι αν συγκρίνουμε τους χρόνους εκτέλεσης του προγράμματος όπου χρησιμοποιούμε μόνο MPI με αυτούς μιας οποιασδήποτε εκτέλεσης του προγράμματος που χρησιμοποιεί MPI + OpenMP, θα δούμε ότι και εκεί η διαφορά είναι ελάχιστη. Αυτή η παρατήρηση φαίνεται και στο διάγραμμα παρακάτω.



Στατικό scheduling

Για PROCESSES = 36

t\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
2	0.1837	0.4193	0.4945	0.9589	1.5006	2.9909	4.3011
4	0.1928	0.4109	0.5067	0.9134	1.5166	2.9498	4.2905
8	0.1985	0.3986	0.5019	0.8973	1.5140	2.9356	4.1886
16	0.1852	0.4117	0.4952	0.8882	1.4883	2.9324	4.1669

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

Απλή παραλληλοποίηση

Για PROCESSES = 36

t\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
2	0.1921	0.4245	0.4921	0.9583	1.5200	2.9870	4.2897
4	0.1934	0.4169	0.5067	0.9066	1.5178	2.9500	4.2891
8	0.2005	0.3997	0.5067	0.8864	1.5298	2.9404	4.1913
16	0.1916	0.4153	0.4980	0.8905	1.4959	2.9327	4.1684

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε sec

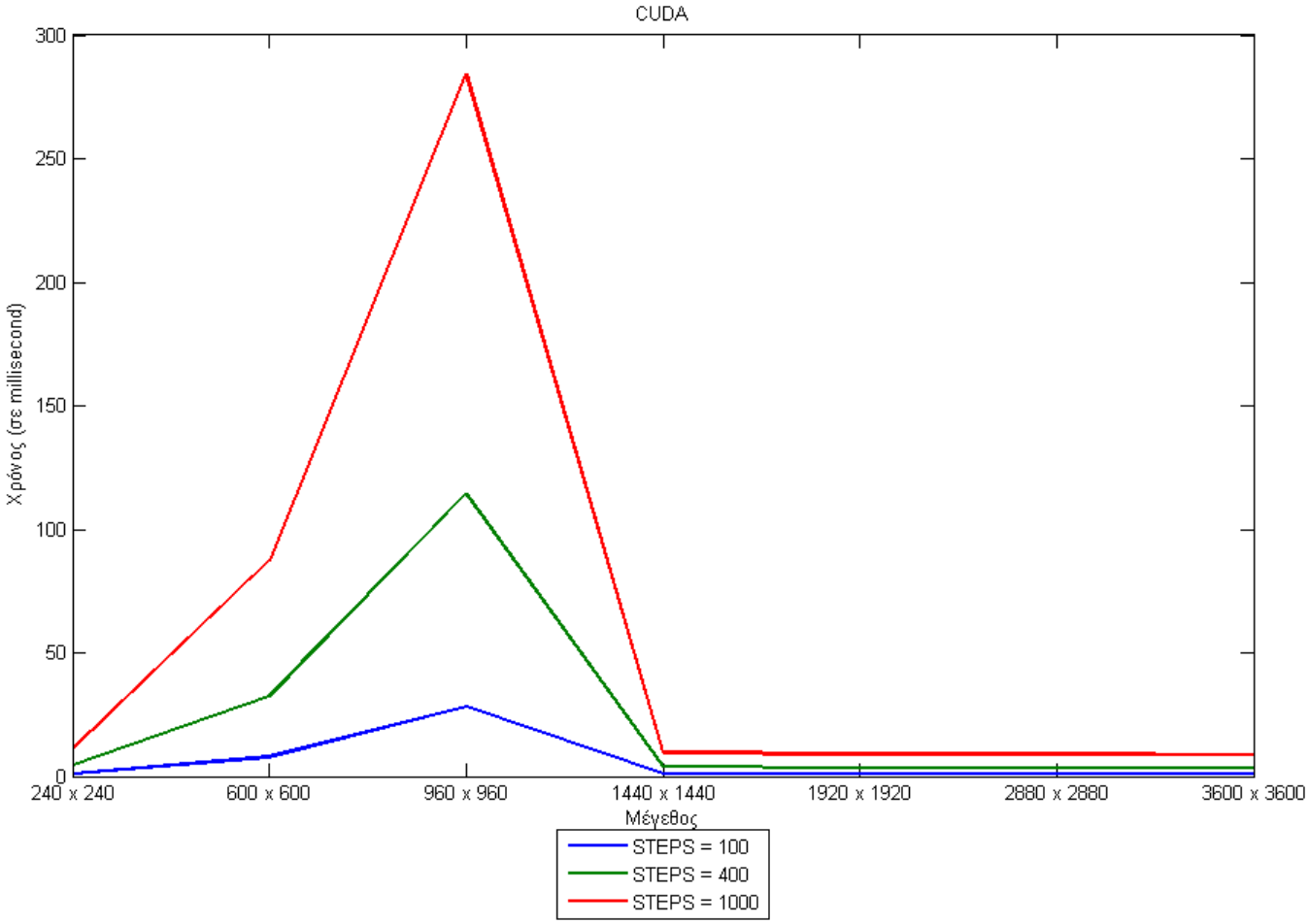
CUDA πρόγραμμα:

Για το πρόγραμμα αυτό χρειάστηκε να χρησιμοποιήσουμε ως μονάδα για τη μέτρηση των χρόνων τα milliseconds, διότι, όπως ήταν αναμενόμενο, ήταν πολύ πιο γρήγορο από τα υπόλοιπα. Όπως αναφέρθηκε παραπάνω, χρησιμοποιήσαμε ως αριθμό block και αριθμό threads/block τον αριθμό γραμμών και τον αριθμό των στηλών του συνολικού αρχικού πίνακα, αντίστοιχα. Μιας και δεν έχουμε έλεγχο για το πόσους πυρήνες CUDA θα χρησιμοποιήσει η GPU για να τρέξει το πρόγραμμα, αποφασίσαμε να κάνουμε τις δοκιμές αλλάζοντας το μέγεθος του προβλήματος, όπως και σε όλες τις προηγούμενες μετρήσεις, αλλά και τον αριθμό των STEPS. Τα αποτελέσματα των εκτελέσεων είναι:

s\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
100	1.0344	7.9452	28.3680	0.9685	0.9121	0.9105	0.9032
400	4.6136	32.2594	114.4725	3.9026	3.3839	3.3681	3.3259
1000	11.1520	87.2770	284.0628	9.7530	9.0016	8.8977	8.7147

οι χρόνοι εκτέλεσης είναι εκφρασμένοι σε millisec

Όπως ήταν αναμενόμενο οι χρόνοι είναι καλύτεροι από κάθε άλλη πρακτική παράλληλου προγραμματισμού χρησιμοποιήσαμε. Τα αποτελέσματα είναι θεαματικά και μας δείχνουν το πόσο ισχυρές σε υπολογισμούς είναι πλέον οι κάρτες γραφικών. Αυτό που παρατηρούμε από τα παραπάνω αποτελέσματα είναι ότι έχουμε μια αναλογική αύξηση του χρόνου όσο μεγαλώνει το πρόβλημα, μέχρι όμως ένα σημείο. Όπως δείχνει ο πίνακας βλέπουμε ότι από ένα μέγεθος και μετά έχουμε μια εκπληκτική μείωση του χρόνου. Γραφικά μπορούμε να δούμε αυτή τη συμπεριφορά του προγράμματος στο παρακάτω διάγραμμα.


CUDA
300
250
200
150
100
50
0
Χρόνος (σε millisecond)
240 x 240
600 x 600
960 x 960
1440 x 1440
1920 x 1920
2880 x 2880
3600 x 3600
Μέγεθος
STEPS = 100
STEPS = 400
STEPS = 1000

4. Speedup & Efficiency

Για να υπολογίσουμε το Speedup & το Efficiency των προγραμμάτων μας, χρησιμοποιήσαμε τους 2 τύπους:

- $S_p = T_{serial} / T_{parallel}$
- $E_p = S_p / p$

όπου:

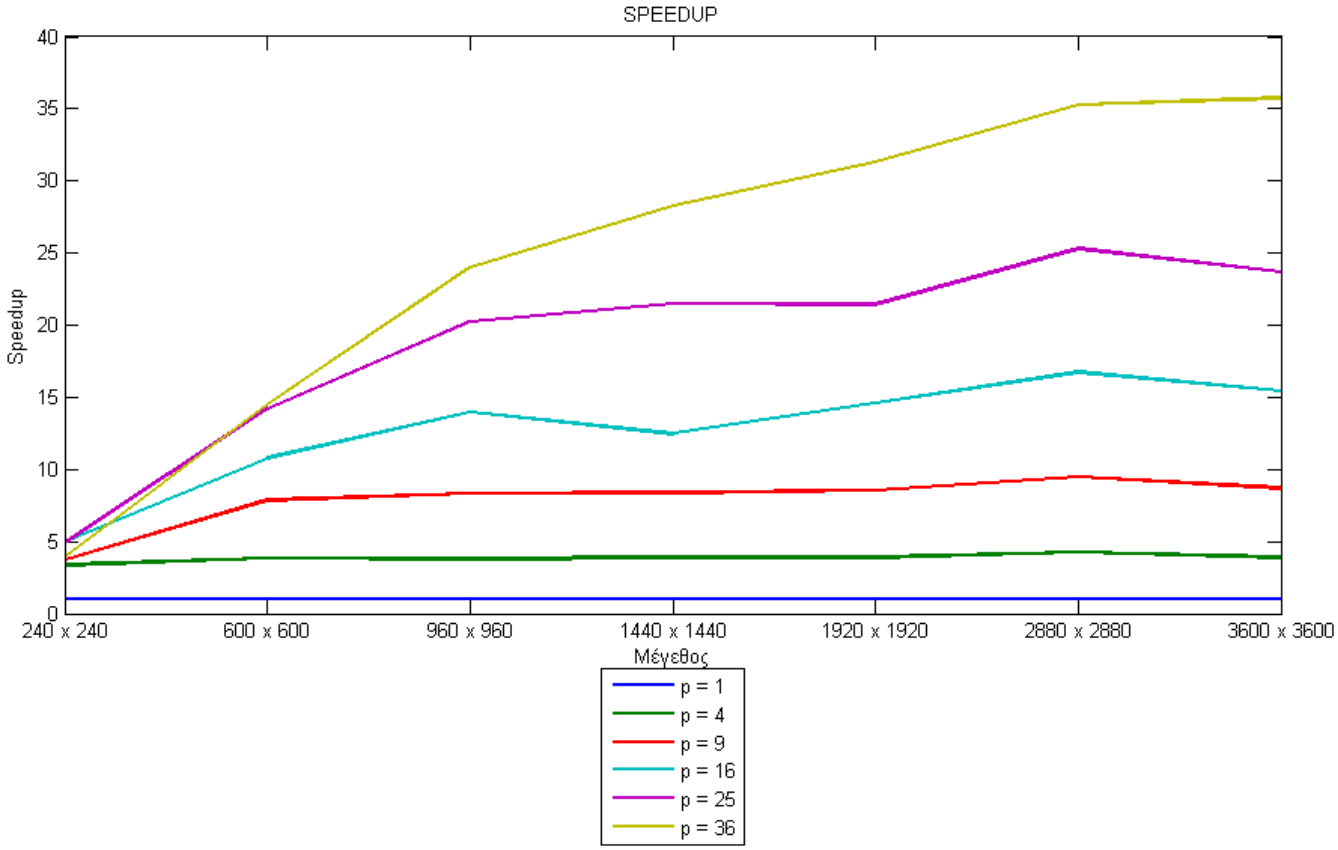
S_p : speedup
T_{new} : Χρόνος τροποποιημένου προγράμματος
T_{old} : Χρόνος αρχικού προγράμματος
E_p : Efficiency

Για τον υπολογισμό των παραπάνω μεγεθών πήραμε ενδεικτικούς χρόνους από κάθε πρόγραμμα, οι οποίοι είχαν υπολογιστεί για κοινά μεγέθη με αριθμό STEPS = 1000.

Speedup

Υπενθυμίζουμε ότι ο τύπος είναι $S_p = T_{serial} / T_{parallel}$

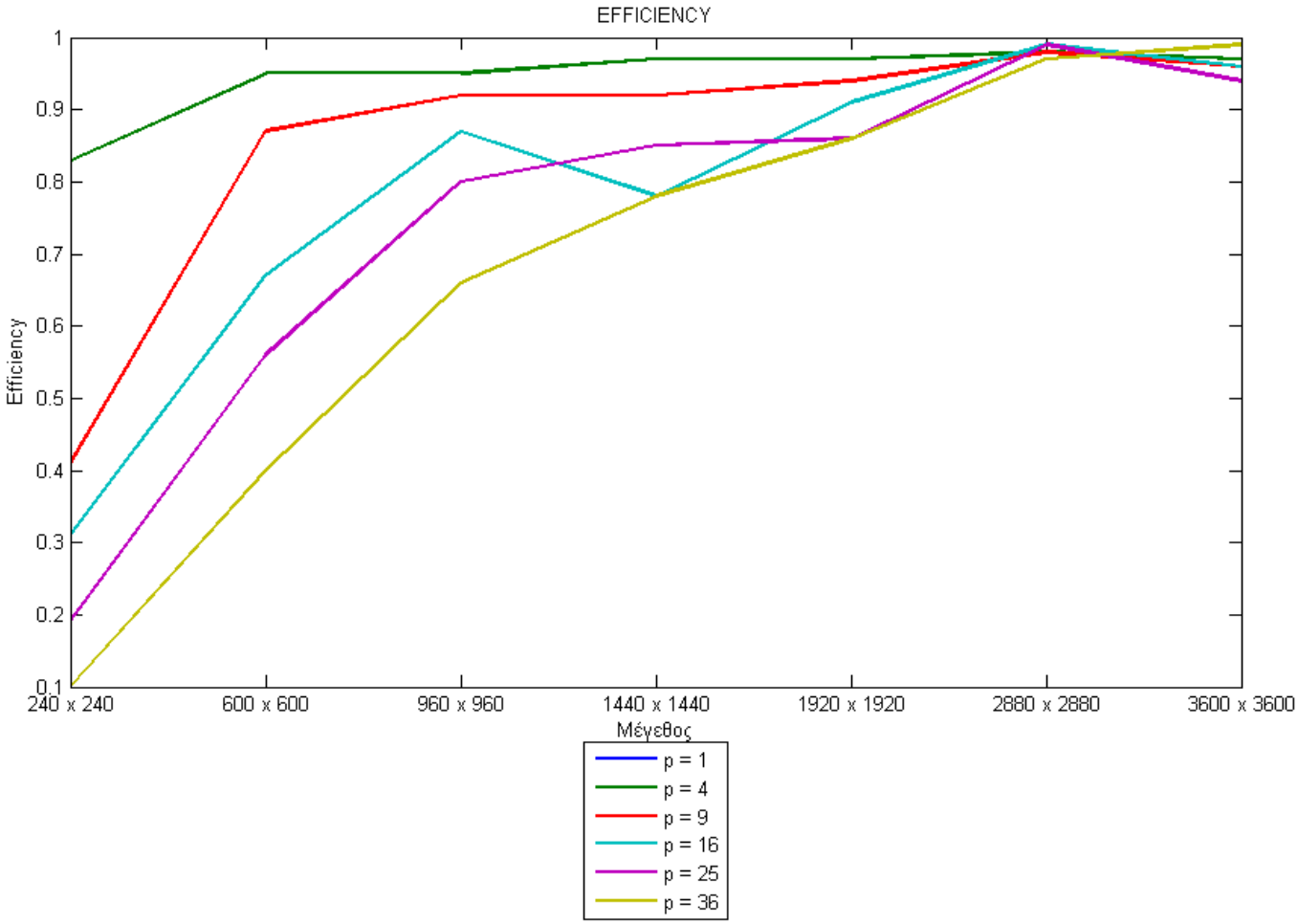
p\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
1	1	1	1	1	1	1	1
4	3.35	3.82	3.81	3.88	3.89	3.95	3.89
9	3.73	7.86	8.33	8.36	8.53	8.90	8.70
16	5.02	10.76	13.96	12.48	14.58	15.98	15.45
25	4.89	14.24	20.24	21.48	21.43	24.99	23.67
36	3.91	14.50	24	28.27	31.29	35.25	35.77



Efficiency

Υπενθυμίζουμε ότι ο τύπος είναι $E_p = S_p / p$

p\size	240 x 240	600 x 600	960 x 960	1440 x 1440	1920 x 1920	2880 x 2880	3600 x 3600
1	1	1	1	1	1	1	1
4	0.83	0.95	0.95	0.97	0.97	0.98	0.97
9	0.41	0.87	0.92	0.92	0.94	0.98	0.96
16	0.31	0.67	0.87	0.78	0.91	0.99	0.96
25	0.19	0.56	0.80	0.85	0.86	0.99	0.94
36	0.10	0.40	0.66	0.78	0.86	0.97	0.99


EFFICIENCY
Efficiency
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
240 x 240
600 x 600
960 x 960
1440 x 1440
1920 x 1920
2880 x 2880
3600 x 3600
Μέγεθος
p = 1
p = 4
p = 9
p = 16
p = 25
p = 36

5. Σύγκριση αναλυτικών υπολογισμών και πραγματικών μετρήσεων.

Συγκρίνοντας το αρχικό με τα προγράμματα που αναπτύξαμε έχουμε παρατηρήσει τα εξής:

- Σε σχέση με το αρχικό πρόγραμμα και τα τροποποιημένα πρόγραμματα που περιείχαν MPI, φαίνεται πως η (1) προσθήκη μιας επιπλέον συνάρτησης για τον υπολογισμό του πίνακα, (2) η χρήση datatypes, η (3) non-blocking επικοινωνία και (4) η τοπολογία διεργασιών δίνει σημαντική βελτίωση, ιδιαίτερα σε πολύ μεγάλα μεγέθη.
- Σε σχέση με το αρχικό πρόγραμμα και το πρόγραμμα που ήταν υλοποιημένο σε CUDA, βλέπουμε μια αναμενόμενη και τρομακτική βελτίωση. Από τους παραπάνω πίνακες βλέπουμε και τη χρονική υπεροχή του CUDA προγράμματος σε κάθε μέγεθος και αριθμό βημάτων για την update.

6. Συμπεράσματα.

Μετά από πολλές δοκιμές και πειραματισμούς, καταλήξαμε πως υπάρχουν πολλές επιλογές υλοποίησης για την παραλληλοποίηση ενός προβλήματος. Προφανώς, το καλύτερο κριτήριο για τις οποιεσδήποτε επιλογές υλοποίησης είναι ο ίδιος ο αλγόριθμος τον οποίο καλούμαστε να παραλληλοποιήσουμε. Ωστόσο, πρέπει να λάβουμε υπόψιν και το περιβάλλον υλοποίησης και τα μέσα που έχουμε στη διάθεση μας ώστε να πετύχουμε το βέλτιστο αποτέλεσμα. Για παράδειγμα, για το δοθέν υπό λύση πρόβλημα, σε 2 διαφορετικά μηχανήματα με διαφορετικές αρχιτεκτονικές, οι καλύτεροι χρόνοι, με ίδια μεγέθη και αριθμό βημάτων, προέκυπταν στο πρώτο με 4 πυρήνες και στο δεύτερο με 6. Τέλος, η άποψη που αποκομίσαμε από την έρευνα και την υλοποίηση της εργασίας αυτής ήταν ότι η χρήση της GPU αποτελεί μεγάλο πλεονέκτημα για τον παράλληλο προγραμματισμό, μιας και είναι το καλύτερο εργαλείο για τη βέλτιστη και -υπο όρους- έγκυρη πραγματοποίηση υπολογισμών.

Περιβάλλοντα & Εργαλεία που χρησιμοποιήθηκαν

Η εργασία πραγματοποιήθηκε κατά βάση σε Linux Ubuntu 14.04 και ελέγχθηκε στα μηχανήματα της σχολής, σε Linux Mint 17 και σε Mac OS X v10.9. Όσον αφορά το CUDA κομμάτι δοκιμάστηκε και στο μηχάνημα CUDA που είχαμε στη διάθεση μας από το διδάσκοντα, αν και οι μετρήσεις έγιναν σε local host με κάρτα **nVidia GeForce GTX 460**.

specs: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-460/specifications

βαθμολογία: https://developer.nvidia.com/cuda-gpus (2.1)