Παράλληλα Συστήματα, Εργασία Μαθήματος 2020 - 2021

Ροβιθάκης Ιωάννης, 1115201800164
Γαλάνης Γεώργιος, 1115201800024

Εισαγωγή:

Η εργασία αυτή αφορά τον σχεδιασμό και την υλοποίηση παράλληλων προγραμμάτων, με στόχο την αύξηση της απόδοσης υπολογιστικών εφαρμογών. Κύριος στόχος, ήταν η εξοικείωση μας με τις βιβλιοθήκες MPI, OpenMP και Cuda, και η χρήση τους για παραλληλοποίηση υπολογισμών. Τα παραδοτέα της εργασίας μας αναπτύχθηκαν με βάση τις μεθοδολογίες σχεδιασμού που διδαχθήκαμε κατα τη διάρκεια της χρονιάς, και επιτυγχάνουν επιτάχυνση του σειριακού κώδικα εκμεταλλευόμενα τους παράλληλους πόρους (CPU ή GPU αντιστοιχα), που έχουμε διαθέσιμους στην “Αργώ¨.

Ακολουθιακό Πρόγραμμα:

Το πρώτο βήμα για μια καλή παράλληλη υλοποίηση, είναι η κατάλληλη προετοιμασία του σειριακού προγράμματος-βάσης. Όπως είδαμε και από το παραδειγμα με την παραλληλοποίηση του bubble sort, οι κατάλληλες τροποποιήσεις σε ένα σειριακό πρόγραμμα, διευκολύνουν σημαντικά τόσο τον σχεδιασμό του παράλληλου προγράμματος, όσο και την υλοποίηση του. Ακόμα, η διαδικασία βελτιστοποίησης και ανασχεδιασμού του σειριακού προγράμματος, οδηγεί τελικά σε ένα **βέλτιστο** (σε ιδανικές συνθήκες) εκτελέσιμο, το οποίο μπορεί να λειτουργήσει ως ένα ακόμα Benchmark για την απόδοση των παράλληλων υλοποιήσεών μας στη συνέχεια.

Όσον αφορά το παραδοτέο σειριακό πρόγραμμά μας, κάναμε τις εξείς βελτιώσεις:

1) Για να αποφύγουμε πλήρως το overhead κλήσεων συναρτήσεων, μεταφέραμε το σώμα της συνάρτησης “one_jacobi_iteration” μέσα στο κύριο loop του προγράμματος (με τις απαραίτητες τροποποιήσεις στα ονόματα των μεταβλητών).

2) Όπου ήταν δυνατό, αφαιρέσαμε εντολές “assignment” τιμών εσωτερικά του κυρίου loop. Παρά το μικρό κόστος μιας απλής εντολής “assignment”, όταν αυτή βρίσκεται εσωτερικά ενός βρόγχου ο οποίος εκτελείται εκατομμύρια φορές, όπως είναι λογικό, όλες οι μικρές καθυστερήσεις αθροίζονται σε μια μη αμελητέα ποσότητα.
(Μεταφορά του τύπου υπολογισμού των τιμων “f”, και ενσωμάτωση του στον τύπο υπολογισμού των “updateVal”)

3) Παρατηρήσαμε πως στον κώδικα της “one_jacobi_iteration” , οι ίδιες τιμές fX και fY, υπολογίζονται επανειλημμένα, και προσθέσαμε ένα απλό στάδιο προϋπολογισμών ωστε να υπολογίζονται μόνο μια φορά.

Κατα την αρχική μας υλοποίηση, προϋπολογίζαμε και εναν πίνακα “f”, ο οποίος αντιστοιχούσε στις τιμές “f” του δεδομένου σειριακού προγράμματος. Στην πράξη όμως, η παραπάνω επιλογή προσέφερε μηδαμινή επιτάχυνση και ταυτόχρονα αυξανε κατα πολύ την κατανάλωση μνήμης του προγράμματός μας (με αποτέλεσμα η αρχική μας υλοποίηση να μην μπορεί να εκτελεστεί για μέγεθος 26880x26880 για λόγους μνήμης). Οι τιμές “f” οντως υπολογίζονται παραπάνω απο μια φορες για κάθε κελί του πίνακα, ενώ είναι πρακτικά σταθερές, αλλα παρόλα αυτά, στην τελική πιστεύουμε πως η μικρή επιπλέον επιτάχυνση, δεν αξίζει το σημαντικά μεγαλύτερο κόστος σε μνήμης και κατα συνέπεια αφαιρέσαμε τους προϋπολογισμούς των “f”.
(Tradeoff λίγου χρόνου για οικονομία αρκετής μνήμης)

4) Τέλος, (αν και δεν αφορά την παραλληλοποίηση), μεταφέραμε και το σώμα της “checkSolution”, μέσα στην main, ωστε να εκμεταλλευτούμε και σε αυτή τις προϋπολογισμένες τιμές fX, fY. (Αναμενόμενα ελάχιστη επίδραση στους χρόνους)

Χρόνοι Εκτέλεσης:
Παρακάτω φαίνεται η σύγκριση των χρόνων (σε seconds) του αρχικού σειριακού προγράμματος, με το βελτιστοποιημένο:

Μέγεθος	Χρόνος Αρχικού	Χρόνος Βελτιστοποιημένου	Επιτάχυνση
840x840	0,87	0,50	1,74
1680x1680	3,37	1,89	1,78
3360x3360	13,49	7,40	1,82
6720x6720	53,32	29,53	1,81
13440x13440	213,29	118,43	1,80
26880x26880	853,23	476,37	1,79

Παρατηρούμε μια μέση επιτάχυνση της τάξης του 1.8x σε όλα τα μεγέθη. Αναμενόμενο δεδομένων των αλλαγών και της αφαίρεσης πολλών περιττών υπολογισμών.

Σχεδιασμός MPI Παραλληλοποίησης:

1) Διαμοιρασμός δεδομένων των πινάκων στις διαθέσιμες διεργασίες:
Βασίσαμε τον διαμοιρασμό των πινάκων σε blocks και την ανάθεση τους σε διεργασίες, στην μορφή της καρτεσιανής μας τοπολογίας. Στόχος μας ήταν ο κάθε αρχικός πίνακας να σπάσει σε **comm_sz** ισομεγέθη blocks (1 block ανα διεργασία) ωστε να κατανεμηθεί ισα ο υπολογιστικός φόρτος. Στην πράξη, οι διαστάσεις των blocks προκύπτουν ως συνάρτηση των διαστάσεων του πίνακα εισόδου και των διαστάσεων του Cartesian Topology. Με βάση τις διαστάσεις αυτές, υπολογίζουμε τα απαραίτητα displacements και offsets και με βάση αυτά διαχωρίζουμε τον πίνακα με χρήση Scatterv μεταξύ των διαθέσιμων διεργασιών, διατηρώντας πάντα τις σωστες σχετικές τους θέσεις με βάση το Cartesian Topology, ωστε να περνάνε σωστα τα Halo points αργότερα.

Όσον αφορά την δομή και την μεταφορά των Blocks, έχουμε χρησιμοποιήσει δύο ειδικούς τύπους δεδομένων, τον main block (**mblock**) ο οποίος αναλαμβάνει την σωστή λήψη των περιεχομένων του block από τον κύριο πίνακα (Διαστάσεις nxm => **Χωρίς padding** απο μηδενικά), και τον internal block (**iblock**) ο οποίος αναλαμβάνει την σωστη εναπόθεση των στοιχείων αυτών στον εσωτερικό πίνακα του block κάθε διεργασίας
(u : Διαστάσεις (block width+2) x (block_height+2) => **Με padding** απο μηδενικά)

Με βάση τον **Foster**, ο διαχωρισμός αυτός του πίνακα μας σε blocks, αποδίδει καλύτερα από άλλες μεθόδους διαχωρισμού πινάκων, πχ σε σειρές/στήλες, και για αυτό τον λόγο αναμένουμε και να ξεπεράσουμε σε απόδοση το εκτελέσιμο **challenge**.

Σημείωση: Εφόσον ο αρχικό πίνακας αρχικοποιείται σε μηδενικά πάντα, θα μπορούσαμε να μην τον κάνουμε καν allocate στην αρχή τον u_global και να στέλνουμε τα blocks του στις διαφορες διεργασίες. Θα στέλναμε δηλαδή μόνο τα απαραίτητα offsets, και οιι πίνακες u και u_old απλά θα αρχικοποιούνταν σε μηδέν και θα συνέχιζε κανονικά το πρόγραμμα. Πιθανόν με την μέθοδο αυτή να μπορούμε να τρέξουμε το πρόγραμμα μας και για μεγαλύτερη είσοδο από 26880x26880 καθώς δεν θα απαιτείται δεσμευση πολύ χώρου από την κεντρική διεργασία.

2) Εστιάζοντας στο main loop του προγράμματος:
Έχουμε διαχωρίσει τον πίνακα με λογική stencil, στα εσωτερικά σημεία, αυτά δηλαδή που δεν εξαρτώνται από κάποια άλω, και τα εξωτερικά, αυτά που απαιτούν την τιμή μιας άλου για τον υπολογισμό τους. Όπως θα δούμε και παρακάτω, με τον τρόπο αυτό μπορούμε να υπολογίζουμε τις δύο αυτές κλάσεις σημείων ανεξάρτητα μεταξύ τους και να μειώσουμε τον χρόνο αδράνειας των διεργασιών μας λόγω αναμονών.

Η επικοινωνία μεταξύ των διαφόρων διεργασιών γίνεται με δημιουργία persistent connections μεταξύ των διεργασιών, τα οποία απλά δημιουργούνται μια μονο φορά στην αρχή και χρησιμοποιούνται επανειλημμένα (Έτσι και αλλιως οι γείτονες παραμένουν σταθεροί καθόλη την διάρκεια της εκτέλεσης του προγράμματος). Ετσι γλιτώνουμε και το overhead των συνεχών δημιουργιών νεων requests κάθε φορα που θέλουμε να στείλουμε ένα μήνυμα. Είναι σημαντικό να αναφέρουμε πως δημιουργούμε συνολικά 16 persistent connections (8 send και 8 receive), τα μισά εκ των οποίων λειτουργούν με τον πίνακα u και τα άλλα μισά με τον πίνακα u_old, καθώς τους αλλάζουμε σε κάθε επανάληψη μεταξύ τους. Οι εναλλαγές μεταξύ requests γίνονται όμοια με την λογική ενος απλού lookup table, καθως αποθηκεύουμε ολα τα requests κάθε τύπου σε έναν πίνακα και στη συνεχεια με βάση την τιμή του iterationCount (even or odd), επιλέγουμε ποιο set of requests θα χρησιμοποιήσουμε τελικά. Με τον τρόπο αυτό αποφεύγουμε και την χρήση **if** για την επιλογή των requests.

3) Η κύρια διεργασία (rank=0) διαβάζει το input και ενημερώνει όλες τις υπόλοιπες, στέλνοντας το input που διάβασε με χρήση ενος custom τύπου δεδομένων και BroadCast.

Σχεδιασμός MPI Παραλληλοποίησης:

Το MPI αποτελεί μια βιβλιοθήκη για ανταλλαγή μηνυμάτων μεταξύ διεργασιών (**Message Passing Interface**). Συνεπώς για να επιτύχουμε όσο το δυνατόν καλύτερη απόδοση, θα πρέπει να περιορίσουμε όσο γίνεται τις καθυστερήσεις που προκύπτουν απο την ανταλλαγή των μηνυμάτων και τον απαραίτητο συγχρονισμό των διεργασιών. Για να ελαχιστοποιήσουμε τον “νεκρό” χρόνο - τον χρόνο που μια διεργασια αναγκάζεται να παραμείνει αδρανής επειδή απαιτεί δεδομένα από κάποια άλλη διεργασια, αρχικά χρησιμοποιήσαμε **persistent connections** για να αποφύγουμε το overhead των επαναλαμβανομένων νέων requests (όπως είδαμε και παραπάνω). Ακόμα, καναμε τα send και receive halos requests να **γράφουν/διαβάζουν απευθείας τους πίνακες u και u_old** ωστε να μην σπαταλάμε χρόνο σε περιττές αντιγραφες buffers. Επιπλέον, βασιζόμενοι στο σύστημα stencil που περιγράψαμε στο στάδιο του σχεδιασμού μας (Διαχωρισμός σε εσωτερικά/ανεξαρτητα και εξωτερικά/εξαρτημένα κελιά), βάλαμε τους **υπολογισμούς οι οποίοι δεν εξαρτώνται από halos**, **να εκτελούνται όσο η διεργασία αναμένει τα halo points από τους γείτονες της**, γεμίζοντας έτσι κενό χρόνο που θα χαραμιζόταν διαφορετικά. Επιπροσθέτως, σε όλες μας τις **αποστολές/λήψεις, έχουμε επιλέξει να χρησιμοποιούμε data types**, ωστε να μην σπαταλάμε άσκοπα χρόνο με “πληρώνοντας” το overhead απο πολλαπλά requests ενώ ένα μόνο αρκεί. Τέλος, στο Cartesian topology μας, χρησιμοποιούμε rank reordering ετσι ωστε το MPI να τοποθετήσει γειτονικά (με βάση πάντα την τοπολογία) ranks, όσο γίνεται πιο κοντα μεταξύ τους ώστε να ελαχιστοποιηθεί η καθυστέρηση των μηνυμάτων μεταξύ τους.

Μετρήσεις MPI Κώδικα:

i) Challenge:

Δεδομενα \ Διεργασιες	4	9	16	25	36	49	64	80
840x840	0.22	0.13	0.088	0.10	0.11	0.11	0.10	0.29
1680x1680	0.87	0.44	0.28	0.23	0.23	0.22	0.24	0.32
3360x3360	3.43	1.65	1.04	0.77	0.63	0.56	0.51	0.46
6720x6720	13.66	6.48	4.04	3.04	2.25	1.99	1.78	1.62
13440x13440	54.63	25.71	15.93	11.88	9.25	7.57	6.63	5.90
26880x26880	-	102.46	63.38	46.91	36.18	29.41	25.81	22.76

ii) MPI:

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Time	840x840	0,50	0,132	0,089	0,061	0,073	0,075	0,091	0,048	0,091
	1680x1680	1,89	0,65	0,25	0,16	0,12	0,12	0,137	0,078	0,137
	3360x3360	7,40	2,56	0,93	0,92	0,50	0,40	0,28	0,21	0,19
	6720x6720	29,53	10,193	3,59	3,52	1,86	1,42	1,17	0,98	0,79
	13440x13440	118,43	40,40	14,90	13,90	7,22	5,44	4,34	3,55	2,88
	26880x26880	476,37	161,72	56,28	55,29	28,47	21,42	16,99	13,95	11,20

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Speedup	840x840	1	3,79	5,62	8,20	6,85	6,67	5,49	10,42	5,49
	1680x1680	1	2,91	7,56	11,81	15,75	15,75	13,80	24,23	13,80
	3360x3360	1	2,89	7,96	8,04	14,80	18,50	26,43	35,24	38,95
	6720x6720	1	2,90	8,23	8,39	15,88	20,80	25,24	30,13	37,38
	13440x13440	1	2,93	7,95	8,52	16,40	21,77	27,29	33,36	41,12
	26880x26880	1	2,95	8,46	8,62	16,73	22,24	28,04	34,15	42,53

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Efficiency	840x840	1	0,95	0,62	0,51	0,27	0,19	0,11	0,16	0,069
	1680x1680	1	0,73	0,84	0,74	0,63	0,44	0,28	0,38	0,17
	3360x3360	1	0,72	0,88	0,50	0,59	0,51	0,54	0,55	0,49
	6720x6720	1	0,72	0,91	0,52	0,64	0,58	0,52	0,47	0,47
	13440x13440	1	0,73	0,88	0,53	0,66	0,60	0,56	0,52	0,51
	26880x26880	1	0,74	0,94	0,54	0,67	0,62	0,57	0,53	0,53

iii) MPI without AllReduce:

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Time	840x840	0,50	0,11	0,07	0,048	0,047	0,069	0,067	0,042	0,056
	1680x1680	1,89	0,64	0,22	0,14	0,10	0,083	0,10	0,060	0,072
	3360x3360	7,40	2,56	0,92	0,89	0,47	0,35	0,25	0,19	0,15
	6720x6720	29,53	10,18	3,57	3,51	1,85	1,43	1,12	0,91	0,73
	13440x13440	118,43	40,40	14,13	13,88	7,18	5,44	4,30	3,53	3,18
	26880x26880	476,37	162,10	56,45	55,30	28,55	21,49	16,95	13,95	11,91

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Speedup	840x840	1	4,55	7,14	10,42	10,64	7,25	7,46	11,90	1,96
	1680x1680	1	2,95	8,59	13,50	18,90	22,77	18,90	31,50	8,89
	3360x3360	1	2,89	8,04	8,31	15,74	21,14	29,60	38,95	17,07
	6720x6720	1	2,90	8,27	8,41	15,96	20,65	26,37	32,45	13,95
	13440x13440	1	2,93	8,38	8,53	16,49	21,77	27,54	33,55	12,70
	26880x26880	1	2,94	8,44	8,61	16,69	22,17	28,10	34,15	13,61

	Δεδομενα \ Διεργασιες	Σειριακό	4	9	16	25	36	49	64	80
Efficiency	840x840	1	1,14	0,79	0,65	0,43	0,20	0,15	0,19	0,02
	1680x1680	1	0,74	0,95	0,84	0,76	0,63	0,39	0,49	0,11
	3360x3360	1	0,72	0,89	0,52	0,63	0,59	0,60	0,61	0,21
	6720x6720	1	0,73	0,92	0,53	0,64	0,57	0,54	0,51	0,17
	13440x13440	1	0,73	0,93	0,53	0,66	0,60	0,56	0,52	0,16
	26880x26880	1	0,73	0,94	0,54	0,67	0,62	0,57	0,53	0,17

Σχόλια:

Όπως περιμέναμε με βάση τις βελτιώσεις που περιγράψαμε παραπάνω, η υλοποίηση του MPI μας νικάει με αισθητή διαφορά το challenge. Η κύρια διαφορά - πλεονέκτημα του MPI μας εναντι του challenge, είναι ότι έχουμε υλοποιήσει τον διαχωρισμό σε blocks αντι για σειρες που έχει το challenge, γεγονός που με βάση τον Foster μας δίνει πλεονέκτημα. Στην πράξη, το πλεονέκτημα αυτό φαίνεται καθαρά από τα στατιστικά του MPI Profiler:

Challenge (26880x26880 - 64 processes)

@--- Aggregate Time (top twenty, descending, milliseconds)

Call	Site	Time	App%	MPI%	Count	COV
Sendrecv	190	1.16e+04	0.89	2.81	50	0.00
Allreduce	74	1.16e+04	0.89	2.80	50	0.00
Allreduce	182	1.15e+04	0.88	2.77	50	0.00
Allreduce	188	1.13e+04	0.87	2.73	50	0.00

…

Our Implementation (26880x26880 - 64 processes)

@--- Aggregate Time (top twenty, descending, milliseconds)

Call	Site	Time	App%	MPI%	Count	COV
Waitall	157	173	0.02	1.59	50	0.00
Allreduce	1	142	0.02	1.30	50	0.00
Allreduce	316	137	0.02	1.26	50	0.00
Allreduce	6	133	0.01	1.22	50	0.00

Απο τα παραπάνω στοιχεία, γίνεται άμεσα εμφανής η υπεροχή των Blocks εναντι των σειρών σχεδιαστικά (όπως αναμέναμε), καθώς συγκρίνοντας του χρόνους (στήλη Time) των δύο υλοποιήσεων, παρατηρούμε ότι το challenge εκτελέσιμο έχει πολύ πιο χρονοβόρα communication calls, γεγονός που αποτυπώνεται και στις στήλες App% και MPI% οπου βλέπουμε την δική μας υλοποίηση να έχει σημαντικό πλεονέκτημα.

Σε γενικές γραμμές, οι χρόνοι μας κυμαίνονται στα επίπεδα που περιμέναμε, με κάποια πραγματικά εντυπωσιακά αποτελέσματα από πλευράς επιτάχυνσης. Όσο μεγαλώνει το πλήθος των διεργασιών, πίνακες μεγαλύτερου μεγέθους κλιμακώνουν καλύτερα καθώς δεν υπάρχει σημαντική σπατάλη χρόνου σε επικοινωνία και υπάρχει μια καλή ισορροπία μεταξύ χρόνων υπολογισμών και επικοινωνίας. Αντίστοιχα, για μικρά μεγέθη πινάκων, το overhead της επικοινωνιας καταλήγει να επικαλύπτει το κόστος των υπολογισμών, με αποτέλεσμα αδυναμια καλής απόδοσης/επιτάχυνσης.
Αντίστροφα, για μικρα πλήθη διεργασιών, δεν υπάρχει τόσο σοβαρό overhead επικοινωνιας (λιγοτερες διεργασιες => μικροτερη αναγκη για επικοινωνια), συνεπώς και δεν παρατηρούνται ιδιαιτερες καθυστερήσεις σε μικρά μεγέθη πινάκων. Ομως, όπως είναι λογικό, μικρά πλήθη διεργασιών είναι αδύνατο να ανταγωνιστούν μεγάλα πλήθη διεργασιών, όσον αφορα την κλιμάκωση τους για πολύ μεγάλους πίνακες.

Όσον αφορά την εκτέλεση του MPI χωρις την allreduce, βλέπουμε όντως διαφορά, αλλα μικρή, και αυτή κυρίως σε περιπτώσεις με μεγάλο πλήθος διεργασιών. Αυτό είναι απόλυτα λογικό, καθώς όσο περισσότερες διεργασίες χρησιμοποιούμε, τόσο περισσότερο καθυστερεί το σύστημα μια reduce.

Our Implementation (26880x26880 - 64 processes) - With Reduce

---

@--- Aggregate Time (top twenty, descending, milliseconds)

Call	Site	Time	App%	MPI%	Count	COV
Allreduce	288	201	0.02	1.43	50	0.00
Allreduce	186	192	0.02	1.37	50	0.00
Allreduce	1	190	0.02	1.36	50	0.00
Allreduce	308	187	0.02	1.33	50	0.00

Our Implementation (26880x26880 - 64 processes) - Without Reduce

---

@--- Aggregate Time (top twenty, descending, milliseconds)

Call	Site	Time	App%	MPI%	Count	COV
Waitall	105	813	0.09	3.55	50	0.00
Waitall	77	805	0.09	3.51	50	0.00
Waitall	201	758	0.08	3.31	50	0.00
Waitall	149	752	0.08	3.28	50	0.00

Ενδιαφέρουσα συμπεριφορά παρατηρούμε σε αυτό το παράδειγμα, (καθώς και σε αλλα παρόμοια test cases) οπου η αφαίρεση της allreduce που μαζεύει όλα τα errors εσωτερικά του jacobi iteration, φαίνεται να αυξάνει κατα λίγο τον συνολικό χρόνο εκτέλεσης, αυξάνοντας τους χρόνους αναμονής για send/receive. (Εκτός από comment out την allreduce, αφαιρέσαμε και τον έλεγχο του error στην συνθήκη του κυριου loop καθώς χωρις την reduce η τιμη του error δεν ειναι εγκυρη)

Hybrid MPI-OpenMP:

Κατα την υλοποίηση της υβριδικής version, αντιμετωπίσαμε ένα πρόβλημα κατα την αρχική δημιουργία των threads (ωστε να μην δημιουργούνται κάθε φορα): υπήρχε συγκρουση με τις send/receive του mpi. Δεν καταφέραμε να λύσουμε αυτό το προβλημα οπότε δυστυχώς πληρώνουμε στους χρόνους μας το overhead της δημιουργίας των threads καθε φορα που χρειαζονται.

Ακόμα, λόγω του τρόπου που έχουμε υλοποιήσει το εμφωλευμένο for στο α),
επρεπε να τροποποιήσουμε τον κώδικα του εσωτερικού for και να μεταφέρουμε εκει τις εξωτερικες εντολες για τα indexes (x_off, y_off), για να προσθέσουμε την επιλογή collapse(2) λόγω του overhead των τροποποιήσεων που κάναμε, δυστυχως με το collapse παρατηρήσαμε μικρες καθυστερήσεις οπότε τελικά δεν το χρησιμοποιήσαμε.

Αξιολόγηση απόδοσης Hybrid MPI σε έναν κόμβο:

Διεργασιες MPI	OpenMP Threads	Χρόνος Υβριδικού
1	8	0,480
2	4	0,247
2	8	0,246
4	2	0,136
4	4	0,134

Όπως αναμέναμε, για εναν κόμβο, φαίνεται να αποδίδει καλύτερα ο συνδυασμός 2 mpi processes - 4 omp threads.

Μελέτη Κλιμάκωσης σε πολλαπλούς κόμβους:

1 Κόμβος

Μέγεθος	Χρόνος Υβριδικού
840x840	0,247
1680x1680	0,968
3360x3360	3,833
6720x6720	15,292
13440x13440	61,844
26880x26880	245,940

2 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,134
1680x1680	0,517
3360x3360	1,971
6720x6720	7,724
13440x13440	30,684
26880x26880	123,817

3 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,102
1680x1680	0,365
3360x3360	1,322
6720x6720	5,158
13440x13440	20,499
26880x26880	82,792

4 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,087
1680x1680	0,278
3360x3360	1,018
6720x6720	3,894
13440x13440	15,421
26880x26880	62,108

5 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,080
1680x1680	0,245
3360x3360	0,834
6720x6720	3,145
13440x13440	12,354
26880x26880	49,685

6 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,081
1680x1680	0,214
3360x3360	0,709
6720x6720	2,630
13440x13440	10,299
26880x26880	41,097

7 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,082
1680x1680	0,179
3360x3360	0,625
6720x6720	2,265
13440x13440	8,865
26880x26880	35,713

8 Κόμβοι

Μέγεθος	Χρόνος Υβριδικού
840x840	0,068
1680x1680	0,162
3360x3360	0,527
6720x6720	1,996
13440x13440	7,766
26880x26880	30,921

Η υβριδική υλοποίηση μας φαίνεται να κλιμακώνει αρκετά καλά, αλλα παρόλα αυτά η υλοποίηση σκέτου MPI μας υπερισχύει. Ένας πιθανός λόγος που το υβριδικό υστερεί εναντι του καθαρού MPI είναι η απουσία των επιπλέον βελτιστοποιήσεων που προαναφέραμε.

CUDA:

Μέγεθος	Χρόνος Βελτιστοποιημένου	Χρόνος CUDA (1 GPU)	Επιτάχυνση
840x840	0,50	0,018	27,78
1680x1680	1,89	0,061	30,98
3360x3360	7,40	0,229	32,31
6720x6720	29,53	0,812	36,37
13440x13440	118,43	2,971	39,86
26880x26880	476,37	(Out of VRAM)	-

Η κεντρική ιδέα είναι ότι χωρίζουμε τον πίνακα σε blocks και κάθε thread του block αναλαμβάνει να ενημερώσει ένα συγκεκριμένο cell του u παίρνοντας τα στοιχεία που χρειάζεται για τον υπολογισμό της νέας τιμής. Την θέση του cell την δεικτοδοτούμε χρησιμοποιώντας το threadID και το blockID με παρόμοιο τρόπο όπως έχουμε δει από το μάθημα. Όλη αυτή την λειτουργικότητα την πραγματοποιεί η συνάρτηση jacobiIterations. Στην main κάνουμε όλες τις απαραίτητες δεσμεύσεις μνήμης όπως: fX και fY πίνακες για να μην ξαναυπολογίζουμε τις τιμές σε κάθε επανάληψη, cerror_in πίνακας για να αποθηκεύουμε το σφάλμα από τον υπολογισμό κάθε κελιού και στην συνέχεια να τα προσθέτουμε όλα μαζί και παίρνουμε το error κάθε επανάληψης.
Η sum συνάρτηση στην ουσία αθροίζει σε κάθε μπλοκ ενα μερικό άθροισμα και στην συνέχεια όταν ξανακαλείται αθροίζει τα μερικά αθροίσματα υπολογίζοντας το συνολικό άθροισμα. Προσπαθήσαμε στην ουσία να υλοποιήσουμε μια reduction συνάρτηση μοιράζοντας τον φόρτο σε αρκετά μπλοκς και νήματα για να αποφύγουμε σειριακή αντιμετώπιση του προβλήματος (η οποία σύμφωνα και με τον profiler πρόσθετε μεγάλη καθυστέρηση) , και με βοήθεια το συγκεκριμένο third party documentation (https://sodocumentation.net/cuda/topic/6566/parallel-reduction--e-g--how-to-sum-an-array-) εφαρμόσαμε την παραπάνω επιθυμητή λειτουργικότητα. Ειδικότερα, κάθε νήμα υπολογίζει ένα μερικό άθροισμα και έπειτα προσθέτει τα αθροίσματα που έχουν υπολογίσει ορισμένα νήματα μέσα στο ίδιο μπλοκ (επιλέγουμε ομοιόμορφα τα threads που προστίθενται). Αυτο γίνεται σε κάθε μπλοκ και στο τέλος προσθέτουμε τα αθροίσματα όλων των μπλοκς (τα οποία έχουν αποθηκευτεί στον πίνακα csum).Το τελικό άθροισμα βρίσκεται στην αρχή του πίνακα . Η initialisefXY είναι αρκετά απλή και απλά αρχικοποιεί τους πίνακες fX και fY με τις αντίστοιχες παραμέτρους.

Όσον αφορά την υλοποίηση για δύο gpus έχουμε φτιάξει ένα ξεχωριστό .cu αρχείο που τρέχει αποκλειστικά για 2 gpus χρησιμοποιώντας 2 threads openmp. Έχω χωρίσει τα δεδομένα στα δύο (συγκεκριμένα σε σειρές, το πάνω και το κάτω μισό του πίνακα). Πάνω κάτω είναι η ίδια σχεδίαση με την διαφορά ότι χρησιμοποιώ πίνακες μεγέθους δύο ώστε με βάση το thread id να

αποθηκεύει η id διεργασία τα δεδομένα της στους αντίστοιχους πίνακες. Στο τέλος κάθε επανάληψης ενημερώνω τα halo points που στην ουσία είναι οι δύο κεντρικές γειτονικές σειρές στο σημείο που διχοτομείται ο αρχικός πίνακας. Οι επιταχύνσεις, όπως φαίνεται παρακάτω, για μικρά μεγέθη είναι χειρότερη από ότι σε μια gpu γεγονός που πιθανότατα οφείλεται στο ότι έχουμε πολλά περιττά νήματα για ένα μικρό έργο. Αντίθετα, όσο μεγαλώνει το μέγεθος του πίνακα παρατηρούμε μεγαλύτερη επιτάχυνση καθώς αξιοποιούνται οι πόροι μας σε μεγαλύτερο βαθμό (χρειάζονται περισσότερα νήματα και μπλοκς για να καλυφθούν όλα τα κελιά). Να σημειωθεί ότι οι απαιτήσεις σε μνήμη ήταν αρκετά μεγάλες για να εκτελεστεί ο κώδικας μας οπότε για διαστάσεις 26880x26880 δεν έτρεξε ούτε με 2 gpu...

Όπως φαίνεται και απο τις μετρήσεις, επιτυγχάνονται πραγματικά εντυπωσιακές επιταχύνσεις, αναμενόμενες λόγω των χαρακτηριστικών των GPU ως hardware.

Μέγεθος	Χρόνος Αρχικού	Χρόνος CUDA (2 GPU)	Επιτάχυνση
840x840	0,87	0,089	9,78
1680x1680	3,37	0,120	28,08
3360x3360	13,49	0,235	57,40
6720x6720	53,32	0,695	76,72
13440x13440	213,29	2,217	96,21
26880x26880	853,23	-	-