Εργασία στο μάθημα ‘Παράλληλα Συστήματα’

Γεωργία Κουτίβα – 1115201700060
Αλέξανδρος Νεοφώτιστος - 1115201700270

I. Εισαγωγή

Το αντικείμενο της εργασίας αφορά την βελτιστοποίηση ακολουθιακού προγράμματος που υλοποιεί την jacobi μέθοδο με την παραλλαγή successive over-relaxation προς αριθμητική επίλυση της εξίσωσης Poisson, και την παραλληλοποίησή του χρησιμοποιώντας τα APIs MPI, OpenMP και CUDA.

Ο κώδικας της εργασίας βρίσκεται στον φάκελο /data/deliverables/argo189.

Περιεχόμενα

Επιπλέον, θα θέλαμε να αναφέρουμε ότι το πλήθος των αρχείων στον φάκελο στην ΑΡΓΩ διαφέρει από αυτό που αναφέρεται στην εκφώνηση. Θέλοντας να είμαστε περισσότερο σαφείς, κρατήσαμε 2 εκδοχές του ParallelMPI προγράμματος (ώστε να φαίνεται ο τρόπος με τον οποίον χειριστήκαμε την απενεργοποίηση της AllReduce). Το πρόγραμμα CUDA έχει γραφτεί σε 2 διαφορετικά αρχεία λόγω έλλειψης χρόνου (ένα για 1 κάρτα γραφικών και ένα για 2). Έχουμε επίσης κρατήσει ένα script για κάθε περίπτωση τρεξίματος όλων των προγραμμάτων: κάθε πλήθος διεργασιών και κάθε πλήθος νημάτων, καθώς και 2 διαφορετικά scripts στο CUDA.

II. Ακολουθιακό Πρόγραμμα

Αρχικά, το ακολουθιακό πρόβλημα τρέχει για 0.875 δευτερόλεπτα για το μέγεθος πίνακα 840x840. Μεταφέροντας το σώμα της συνάρτησης one_jacobi_iteration στο εσωτερικό του βρόχου while, και μεταφέροντας τον υπολογισμό των cx, cy, cc ακριβώς έξω από αυτόν, παίρνουμε χρόνο 0.838 δευτερολέπτων. Άλλος περιττός επαναϋπολογισμός είναι αυτός των fX^2, fY^2 και f. Η βελτιστοποίηση που επιλέξαμε ήταν η εξής: κρατήσαμε τα fX^2 και fY^2 σε δύο διανύσματα μήκους 1xn και 1xm αντίστοιχα. Ιδανικά, θα θέλαμε να κρατούσαμε το f για κάθε θέση του nxm πίνακα, όμως αυτό είναι ασύμφορο από πλευράς μνήμης, αφού χρειάζεται ένας ακόμη nxm πίνακας και αυτό είναι αδύνατον να συμβεί για το μεγαλύτερο μέγεθος προβλήματος. Μόνο με τα δύο διανύσματα αυτά πετυχαίνουμε και πάλι εξαιρετική μείωση στην απόδοση. Η αρχικοποίησή τους και η πρώτη επανάληψη της Jacobi SOR γίνονται ταυτόχρονα. Κατόπιν, ο βρόχος while κάνει 49 επαναλήψεις, στις οποίες χρησιμοποιούνται τα στοιχεία από τα διανύσματα fXsquared και fYsquared.

Οι χρόνοι για όλα τα μεγέθη προβλήματος ακολουθούν:

Μέγεθος Προβλήματος	Χρόνος
840x840	0.486 sec
1680x1680	1.802 sec
3360x3360	7.064 sec
6720x6720	28.085 sec
13440x13440	112.305 sec
26880x26880	449.683 sec

Δοκιμάσαμε ο υπολογισμός της f να μην ανατίθεται σε ενδιάμεση μεταβλητή, και αντί αυτού να γίνεται μέσα στον υπολογισμό της updateVal. Για το μικρότερο μέγεθος πίνακα είχαμε αύξηση από τα 0.486 στα 0.504 δευτερόλεπτα, επομένως δεν συνεχίσαμε τις μετρήσεις για τα μεγαλύτερα μεγέθη και απορρίψαμε την αλλαγή.

III. Σχεδιασμός MPI Παραλληλοποίησης

Θεωρώντας δεδομένο ότι σχεδόν όλα τα μεγέθη προβλήματος είναι τετράγωνα ενός αριθμού N, δημιουργούμε πάντοτε έναν Καρτεσιανό communicator διαστάσεων NxN, και παίρνουμε την περίπτωση των 80 διεργασιών ως ειδική περίπτωση, όπου ο communicator έχει 8 σειρές και 10 στήλες, όπως υποδεικνύεται από την εκφώνηση.

Ο χωρισμός των πινάκων u, u_old ακολουθεί πιστά την τοπολογία των διεργασιών στον communicator. Ο τρόπος με τον οποίον βρίσκουμε το μέγεθος του τμήματος που αναλαμβάνει μία δεδομένη διεργασία είναι διαιρώντας τις παραμέτρους n, m με τον ίδιο αριθμό N όπως πριν. Στην περίπτωση των 80 διεργασιών, το τοπικό n ισούται με το καθολικό διαιρεμένο με το 10. Ενώ ο communicator είναι 8X10, τα σχόλια της εργασίας ανέφεραν ότι το n δείχνει το μήκος του πλέγματος στην διάσταση των x, επομένως αναφέρεται στον αριθμό των **στηλών** που έχει το πλέγμα. Άρα για να συμφωνούν τα n_local, m_local κάθε διεργασίας με το σχήμα του καρτεσιανού communicator, έχουμε n_local = n_global / 10 και m_local = m_global / 8.

Κατόπιν, μένει να βρούμε το μήκος του διαστήματος των λύσεων που αναλαμβάνει κάθε διεργασία, δηλαδή τα επιμέρους xLeft, yBottom της κάθε διεργασίας. Το deltaX, deltaY είναι κοινό για όλες τις διεργασίες: δείχνει το ‘μέγεθος’ μίας γραμμής ή στήλης του πλέγματος, και αυτό δεν αλλάζει από διεργασία σε διεργασία. Το xLeft κάθε διεργασίας είναι η μετατόπιση από το καθολικό xLeft κατά το πλήθος διεργασιών που βρίσκονται αριστερά της στην γραμμή που ανήκει στον

καρτεσιανό communicator, επί το πλήθος των στηλών που αναλαμβάνει η κάθε μία από αυτές, επί το μέγεθος της κάθε στήλης, που είναι το deltaX. Με παρόμοιο τρόπο υπολογίζουμε το yBottom της κάθε διεργασίας.

Φυσικά, κάθε διεργασία δεσμεύει τώρα πίνακες (n_local+2)*(m_local+2) για να κρατά το κομμάτι των u, u_old που της αντιστοιχεί και την άλω για τα στοιχεία που θα δεχθεί από τους γείτονές της. Όσον αφορά τις συντεταγμένες του πλέγματος τις οποίες έχει αναλάβει να υπολογίσει η κάθε διεργασία, δηλαδή τα κομμάτια των fXsquared, fYsquared που της αντιστοιχούν, δεν απαιτείται άλως, επομένως δεσμεύονται διανύσματα μήκους n_local και m_local αντίστοιχα από κάθε διεργασία για αυτά.

Αποφασίσαμε να εκμεταλλευτούμε τον διαμοιρασμό δεδομένων και για την checkSolution, αν και δεν την λαμβάνουμε υπ’όψιν μας στις μετρήσεις μας (η 2η MPI_Wtime παραμένει μετά το τέλος του while). Καλούμε την συνάρτηση δίνοντάς της ως δεδομένο τον u_old. Δεν χρειάζεται αποστολή εκ νέου της άλω. Κάθε διεργασία υπολογίζει το absoluteError στο δικό της κομμάτι, αυτά γίνονται reduce στην εργασία 0 και υπολογίζεται το τελικό error.

IV. Βελτιστοποίηση MPI Κώδικα

Δεν έχουμε πολλά να αναφέρουμε σε αυτή την ενότητα, καθώς εφαρμόσαμε ρητά και κατά γράμμα *όλες* τις βελτιώσεις που αναφέρονται στις Οδηγίες Σχεδιασμού και τίποτα περισσότερο. Το μόνο που κρίναμε ότι πρέπει να αναφερθεί εδώ είναι το πως υλοποιήσαμε την εκδοχή του προγράμματος με την MPI_Allreduce απενεργοποιημένη. Σχολιάσαμε τόσο τον έλεγχο στο while για το κριτήριο σύγκλισης (ελέγχεται μόνο το πλήθος των επαναλήψεων), όσο και τον εκ νέου υπολογισμού του error σε κάθε επανάληψη. Για να είμαστε όμως σίγουροι ότι το residual εξακολουθούσε να λειτουργεί σωστά, κάναμε ένα και μοναδικό MPI_Reduce κατόπιν των 50 επαναλήψεων το οποίο δεν συμπεριλαμβάνεται στις μετρήσεις του wall time, που συγκεντρώνει τα επιμέρους error των διεργασιών.

V. Μετρήσεις MPI Κώδικα

Α) Με την MPI_Allreduce ενεργοποιημένη

1) Μέγεθος προβλήματος: 840x840

# Proc.	4	9	16	25	36	49	64	80
Time(msec)	132	71	55	64	90	71	52	87
Challenge(msec)	222	119	104	97	97	111	108	108

2) Μέγεθος προβλήματος: 1680x1680

# Proc.	4	9	16	25	36	49	64	80
Time(msec)	513	241	162	128	120	95	80	116
Challenge(msec)	865	422	279	217	219	221	218	326

3) Μέγεθος προβλήματος: 3360x3360

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	1,989	1,179	0,926	0,560	0,423	0,257	0,194	0,190
Challenge(sec)	3,413	1,631	1,054	0,762	0,596	0,543	0,503	0,505

4) Μέγεθος προβλήματος: 6720x6720

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	7,838	4,656	3,549	2,168	1,640	1,153	0,956	0,801
Challenge(sec)	13,580	6,398	4,033	2,796	2,265	2,002	1,768	1,614

5) Μέγεθος προβλήματος: 13440x13440

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	31,204	18,430	14,023	8,428	6,308	4,356	3,597	2,897
Challenge(sec)	54,220	25,418	15,840	10,841	8,570	7,593	6,650	5,913

6) Μέγεθος προβλήματος: 26880x26880

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	126,594	73,795	55,917	33,360	24,914	17,192	14,083	11,314
Challenge(sec)	216,962	101,604	63,314	42,723	33,665	29,507	25,767	22,914

Η επιτάχυνση και η αποδοτικότητα του προγράμματος με την εντολή Allreduce ενεργοποιημένη για όλα τα μεγέθη προβλήματος, έχει ως εξής:

Επιτάχυνση								
# Proc. _____ Problem Size	4	9	16	25	36	49	64	80
840x840	3,68	6,84	8,83	7,59	5,4	6,84	9,34	5,58
1680x1680	3,51	7,47	11,12	14,07	15,01	18,96	22,50	15,53
3360x3360	3,55	5,99	7,62	12,61	16,69	27,48	36,41	37,17
6720x6720	3,58	6,03	7,91	12,95	17,12	24,35	29,37	35,06
13440x13440	3,59	6,09	8,00	13,32	17,80	25,78	31,22	38,76
26880x26880	3,55	6,09	8,04	13,47	18,04	26,15	31,93	39,74

Αποδοτικότητα								
# Proc. _____ Problem Size	4	9	16	25	36	49	64	80
840x840	0,92	0,76	0,55	0,30	0,15	0,13	0,14	0,06
1680x1680	0,87	0,83	0,69	0,56	0,41	0,38	0,35	0,19
3360x3360	0,88	0,66	0,47	0,50	0,46	0,56	0,56	0,46
6720x6720	0,89	0,67	0,49	0,51	0,47	0,49	0,45	0,43
13440x13440	0,89	0,67	0,50	0,53	0,49	0,52	0,48	0,48
26880x26880	0,88	0,67	0,50	0,53	0,50	0,53	0,49	0,49

Το πρόγραμμα δεν κλιμακώνει. Ας πάρουμε το εξής παράδειγμα: Για 4 διεργασίες και για μέγεθος προβλήματος 3360x3360, έχουμε αποδοτικότητα ίση με 0,88. Διπλασιάζοντας την πλευρά του τετραγώνου, τετραπλασιάζουμε το μέγεθος του προβλήματος, άρα το μέγεθος 6720x6720 είναι 4 φορές μεγαλύτερο από το προηγούμενο. Τετραπλασιάζοντας και τον αριθμό των διεργασιών, παρατηρούμε ότι η απόδοση πέφτει στο 0,49. Όμως, για μέγεθος προβλήματος 13440x13440 και 64 διεργασίες, παρατηρούμε ότι η αποδοτικότητα παρουσιάζει σχεδόν αμελητέα πτώση, πηγαίνοντας στο 0,48. Επομένως, παρατηρούμε κάποιο weak scaling για μεγαλύτερα μεγέθη προβλήματος και περισσότερες διεργασίες, αλλά για τα περισσότερα μεγέθη προβλήματος, ο τετραπλασιασμός των διεργασιών δεν διατηρεί την απόδοση σε σταθερά επίπεδα. Ένα ακόμη παράδειγμα προς αυτή την κατεύθυνση είναι η πτώση της απόδοσης από 0,92 σε 0,69 από 4 διεργασίες για 840x840 σε 16 διεργασίες για 1680x1680.

Β) Με την MPI_Allreduce απενεργοποιημένη

1) Μέγεθος προβλήματος: 840x840

# Proc.	4	9	16	25	36	49	64	80
Time(msec)	122	63	54	55	51	42	42	61
Challenge(msec)	222	119	104	97	97	111	108	108

2) Μέγεθος προβλήματος: 1680x1680

# Proc.	4	9	16	25	36	49	64	80
Time(msec)	502	231	144	101	91	79	64	75
Challenge(msec)	865	422	279	217	219	221	218	326

3) Μέγεθος προβλήματος: 3360x3360

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	1,982	1,127	0,885	0,522	0,378	0,214	0,154	0,147
Challenge(sec)	3,413	1,631	1,054	0,762	0,596	0,543	0,503	0,505

4) Μέγεθος προβλήματος: 6720x6720

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	7,830	4,567	3,526	2,115	1,597	1,083	0,888	0,716
Challenge(sec)	13,580	6,398	4,033	2,796	2,265	2,002	1,768	1,614

5) Μέγεθος προβλήματος: 13440x13440

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	31,190	17,513	14,000	8,410	6,253	4,282	3,523	2,834
Challenge(sec)	54,220	25,418	15,840	10,841	8,570	7,593	6,650	5,913

6) Μέγεθος προβλήματος: 26880x26880

# Proc.	4	9	16	25	36	49	64	80
Time(sec, msec)	126,513	71,254	55,762	33,300	24,796	16,922	13,955	11,168
Challenge(sec)	216,962	101,604	63,314	42,723	33,665	29,507	25,767	22,914

Η επιτάχυνση και η αποδοτικότητα του προγράμματος με την εντολή Allreduce απενεργοποιημένη για όλα τα μεγέθη προβλήματος, έχει ως εξής:

Επιτάχυνση								
# Proc. _____ Problem Size	4	9	16	25	36	49	64	80
840x840	3,98	7,71	9,00	8,83	9,52	11,57	11,57	8,10
1680x1680	3,58	7,80	12,51	17,84	19,80	22,81	28,15	24,02
3360x3360	3,56	6,26	7,98	13,53	18,68	33,00	45,87	48,05
6720x6720	3,58	6,14	7,96	13,27	17,58	25,93	31,62	54,78
13440x13440	3,60	6,41	8,02	13,35	17,96	26,22	31,87	39,62
26880x26880	3,55	6,31	8,06	13,50	18,13	26,57	32,22	40,26

Αποδοτικότητα								
# Proc. _____ Problem Size	4	9	16	25	36	49	64	80
840x840	0,99	0,85	0,56	0,35	0,26	0,23	0,18	0,10
1680x1680	0,89	0,86	0,78	0,71	0,55	0,46	0,43	0,30
3360x3360	0,89	0,69	0,49	0,54	0,51	0,67	0,71	0,60
6720x6720	0,89	0,69	0,49	0,53	0,48	0,52	0,49	0,68
13440x13440	0,90	0,71	0,50	0,53	0,49	0,53	0,49	0,49
26880x26880	0,88	0,70	0,50	0,54	0,50	0,54	0,50	0,50

Παρότι η κλιμάκωση του προγράμματος παραμένει ποιοτικά ίδια (ανάλογη πτώση της απόδοσης στα ίδια παραδείγματα με την περίπτωση της ενεργοποιημένης Allreduce), παρατηρούμε ότι επιτυγχάνεται βελτίωση της αποδοτικότητας για τα μεγαλύτερα πλήθη διεργασιών, κυρίως στα πρώτα 3 μεγέθη προβλήματος, και στο 4ο μέγεθος στην περίπτωση των 80 διεργασιών. Στους παρακάτω πίνακες σημειώνονται με πράσινο όλες οι περιπτώσεις στις οποίες το πρόγραμμα χωρίς allreduce βελτίωσε την απόδοσή του. Σημειώνεται ότι στα μη επισημασμένα κελιά, δεν υφίσταται πτώση στην απόδοση, απλά στασιμότητα.

Ο λόγος για τον οποίον παρατηρείται βελτίωση στην απόδοση για τα μικρότερα μεγέθη προβλήματος είναι προφανώς γιατί το overhead της Allreduce είναι συγκρίσιμο με τον χρόνο που χρειάζεται για τους υπολογισμούς όσο τα στοιχεία είναι σχετικά λίγα. Στην περίπτωση των 80 διεργασιών, όπου η επικοινωνία είναι και η πιο χρονοβόρα λόγω του μεγάλου πλήθους των διεργασιών στον communicator, αυτό το overhead είναι τόσο μεγάλο ώστε να εξακολουθεί να είναι συγκρίσιμο με τον υπολογισμό και στο 4ο μέγεθος προβλήματος, με βελτίωση της απόδοσης της τάξης του 60% περίπου.

Αποδοτικότητα για 4 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,92	0,99
1680x1680	0,87	0,89
3360x3360	0,88	0,89
6720x6720	0,89	0,89
13440x13440	0,89	0,90
26880x26880	0,88	0,88

Αποδοτικότητα για 9 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,76	0,85
1680x1680	0,83	0,86
3360x3360	0,66	0,69
6720x6720	0,67	0,69
13440x13440	0,67	0,71
26880x26880	0,67	0,70

Αποδοτικότητα για 16 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,55	0,56
1680x1680	0,69	0,78
3360x3360	0,47	0,49
6720x6720	0,49	0,49
13440x13440	0,50	0,50
26880x26880	0,50	0,50

Αποδοτικότητα για 25 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,30	0,35
1680x1680	0,56	0,71
3360x3360	0,50	0,54
6720x6720	0,51	0,53
13440x13440	0,53	0,53
26880x26880	0,53	0,54

Αποδοτικότητα για 36 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,15	0,26
1680x1680	0,41	0,55
3360x3360	0,46	0,51
6720x6720	0,47	0,48
13440x13440	0,49	0,49
26880x26880	0,50	0,50

Αποδοτικότητα για 49 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,13	0,23
1680x1680	0,38	0,46
3360x3360	0,56	0,67
6720x6720	0,49	0,52
13440x13440	0,52	0,53
26880x26880	0,53	0,54

Αποδοτικότητα για 64 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,14	0,18
1680x1680	0,35	0,43
3360x3360	0,56	0,71
6720x6720	0,45	0,49
13440x13440	0,48	0,49
26880x26880	0,49	0,50

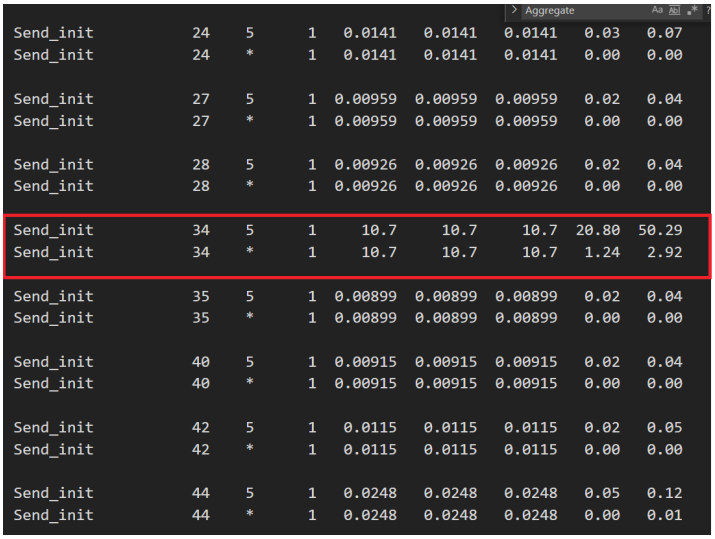
Αποδοτικότητα για 80 διεργασίες		
	Με Allreduce	Χωρίς Allreduce
840x840	0,06	0,10
1680x1680	0,19	0,30
3360x3360	0,46	0,60
6720x6720	0,43	0,68
13440x13440	0,48	0,49
26880x26880	0,49	0,50

Γ) Συμπεράσματα από τα profiling αρχεία

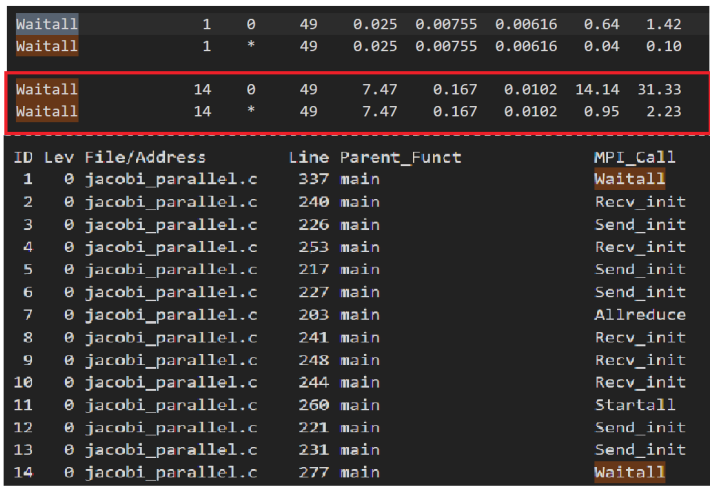
1) Σύγκριση profiling δικού μας προγράμματος, με και χωρίς Allreduce

Εφόσον βλέπαμε ότι μεγαλύτερη βελτίωση στην απόδοση έχουμε για τα πρώτα 3 μεγέθη προβλήματος με την απενεργοποίηση της Allreduce, εξετάζουμε τα profiling αρχεία για αυτά τα μεγέθη προβλήματος, για όλα τα πλήθη διεργασιών. Πράγματι, βλέπουμε ότι μόνο για μικρά πλήθη διεργασιών στα Top Twenty Callsites' Aggregate Times υπάρχουν και άλλες κλήσεις συναρτήσεων πέρα από την Allreduce. Συνήθως, από τις 25 διεργασίες και μετά, η Allreduce καταλαμβάνει αν όχι όλες τις θέσεις της εν λόγω λίστας, το μεγαλύτερο πλήθος από αυτές. Μόνο για τα δύο μεγαλύτερα μεγέθη προβλήματος, η επικοινωνία για την μεταφορά των στοιχείων της άλω είναι συγκρίσιμη με το χρονικό κόστος της Allreduce, και άρα δεν περιμένουμε ιδιαίτερη βελτίωση με την απενεργοποίηση αυτής.

Άλλες κλήσεις του MPI που είναι χρονοβόρες είναι ορισμένα Send_init (π.χ. για μέγεθος προβλήματος 840x840, στο profiling των 16 διεργασιών, παρατηρούμε ότι η διεργασία με rank 5 έχει ένα Send_init callsite που διαρκεί 10,7 msecs σε σχέση με τα υπόλοιπα Send_init callsites του ίδιου rank), καθώς και ορισμένες Startall και Waitall (για μέγεθος προβλήματος 840x840, στο profiling 16 διεργασιών, rank 0, callsite 14 που αντιστοιχεί στην Waitall για τα send requests της άλω στο τέλος του while loop).



```
                                                                    > Aggregate
Send_init          24    5      1   0.0141   0.0141   0.0141   0.03   0.07
Send_init          24    *      1   0.0141   0.0141   0.0141   0.00   0.00

Send_init          27    5      1  0.00959  0.00959  0.00959   0.02   0.04
Send_init          27    *      1  0.00959  0.00959  0.00959   0.00   0.00

Send_init          28    5      1  0.00926  0.00926  0.00926   0.02   0.04
Send_init          28    *      1  0.00926  0.00926  0.00926   0.00   0.00

Send_init          34    5      1     10.7     10.7     10.7  20.80  50.29
Send_init          34    *      1     10.7     10.7     10.7   1.24   2.92

Send_init          35    5      1  0.00899  0.00899  0.00899   0.02   0.04
Send_init          35    *      1  0.00899  0.00899  0.00899   0.00   0.00

Send_init          40    5      1  0.00915  0.00915  0.00915   0.02   0.04
Send_init          40    *      1  0.00915  0.00915  0.00915   0.00   0.00

Send_init          42    5      1   0.0115   0.0115   0.0115   0.02   0.05
Send_init          42    *      1   0.0115   0.0115   0.0115   0.00   0.00

Send_init          44    5      1   0.0248   0.0248   0.0248   0.05   0.12
Send_init          44    *      1   0.0248   0.0248   0.0248   0.00   0.01
```



```
Waitall             1    0     49    0.025  0.00755  0.00616   0.64   1.42
Waitall             1    *     49    0.025  0.00755  0.00616   0.04   0.10

Waitall            14    0     49     7.47    0.167   0.0102  14.14  31.33
Waitall            14    *     49     7.47    0.167   0.0102   0.95   2.23

ID Lev File/Address        Line Parent_Funct             MPI_Call
 1   0 jacobi_parallel.c    337 main                    Waitall
 2   0 jacobi_parallel.c    240 main                    Recv_init
 3   0 jacobi_parallel.c    226 main                    Send_init
 4   0 jacobi_parallel.c    253 main                    Recv_init
 5   0 jacobi_parallel.c    217 main                    Send_init
 6   0 jacobi_parallel.c    227 main                    Send_init
 7   0 jacobi_parallel.c    203 main                    Allreduce
 8   0 jacobi_parallel.c    241 main                    Recv_init
 9   0 jacobi_parallel.c    248 main                    Recv_init
10   0 jacobi_parallel.c    244 main                    Recv_init
11   0 jacobi_parallel.c    260 main                    Startall
12   0 jacobi_parallel.c    221 main                    Send_init
13   0 jacobi_parallel.c    231 main                    Send_init
14   0 jacobi_parallel.c    277 main                    Waitall
```

Εξετάζοντας τα profiling αρχεία του προγράμματος για το ίδιο μέγεθος προβλήματος χωρίς την Allreduce, παρατηρούμε ότι την θέση της Allreduce ως πιο χρονοβόρα κλήση συνάρτησης λαμβάνει η Waitall, ιδιαίτερα για τα μεγαλύτερα πλήθη διεργασιών, όπου καταλαμβάνει ολόκληρη την εικοσάδα πιο χρονοβόρων callsites (2η εικόνα που ακολουθεί: 36 διεργασίες, 840x840). Για τα μικρότερα, η Startall και η Send_Init (αλλά ποτέ η Recv_Init!) εμφανίζονται επίσης περιστασιακά στις πιο χρονοβόρες (1η εικόνα που ακολουθεί: 4 διεργασίες, 840x840). Κατά πάσα πιθανότητα αυτό έχει να κάνει με την λήψη δεδομένων από γείτονες που δεν βρίσκονται στον ίδιο κόμβο με την τρέχουσα διεργασία. Ακόμα και με reorder = true στην MPI_Cart_create, δεν παρατηρείται αλλαγή στην τοποθέτηση διεργασιών στους κόμβους, και αυτό γίνεται γιατί πράγματι, για τετράγωνο καρτεσιανό πλέγμα, ο βέλτιστος τρόπος τοποθέτησης είναι με την σειρά, όπου κάθε διεργασία συνορεύει με δύο γείτονες στον ίδιο κόμβο και με δύο γείτονες σε άλλον κόμβο. Η επικοινωνία με αυτούς τους γείτονες καθορίζει και την απόδοση του προγράμματος.

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

Call	Site	Time	App%	MPI%	Count	COV
Waitall	6	1.15	0.23	9.63	49	0.00
Startall	52	1.06	0.22	8.89	49	0.00
Waitall	46	1.05	0.21	8.83	49	0.00
Startall	12	1.03	0.21	8.65	49	0.00
Waitall	66	1.01	0.21	8.50	49	0.00
Startall	32	0.98	0.20	8.22	49	0.00
Startall	72	0.939	0.19	7.88	49	0.00
Waitall	26	0.898	0.18	7.53	49	0.00
Startall	53	0.335	0.07	2.81	49	0.00
Startall	13	0.329	0.07	2.76	49	0.00
Startall	73	0.322	0.07	2.70	49	0.00
Startall	33	0.321	0.07	2.70	49	0.00
Waitall	58	0.309	0.06	2.60	49	0.00
Waitall	18	0.307	0.06	2.58	49	0.00
Waitall	78	0.302	0.06	2.53	49	0.00
Waitall	38	0.3	0.06	2.52	49	0.00
Send_init	19	0.171	0.03	1.43	1	0.00
Send_init	59	0.166	0.03	1.39	1	0.00
Send_init	39	0.164	0.03	1.38	1	0.00
Send_init	79	0.148	0.03	1.24	1	0.00

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

Call	Site	Time	App%	MPI%	Count	COV
Waitall	286	25.5	1.95	3.19	49	0.00
Waitall	246	25	1.91	3.13	49	0.00
Waitall	26	24.5	1.87	3.06	49	0.00
Waitall	226	24.3	1.86	3.05	49	0.00
Waitall	408	24.3	1.85	3.04	49	0.00
Waitall	68	24.2	1.85	3.03	49	0.00
Waitall	365	23.6	1.80	2.96	49	0.00
Waitall	345	23.5	1.79	2.94	49	0.00
Waitall	425	23.4	1.78	2.93	49	0.00
Waitall	148	23.3	1.78	2.92	49	0.00
Waitall	266	23.3	1.78	2.91	49	0.00
Waitall	386	22.3	1.70	2.79	49	0.00
Waitall	565	22.1	1.68	2.76	49	0.00
Waitall	46	21.7	1.66	2.72	49	0.00
Waitall	486	21.2	1.62	2.65	49	0.00
Waitall	706	20.5	1.57	2.57	49	0.00
Waitall	206	20.3	1.55	2.54	49	0.00
Waitall	506	19.8	1.51	2.48	49	0.00
Waitall	605	19.8	1.51	2.48	49	0.00
Waitall	525	19.1	1.46	2.39	49	0.00

2) Σύγκριση profiling δικού μας προγράμματος, με Allreduce, με το Challenge πρόγραμμα.

Τόσο το κανονικό πρόγραμμα όσο και αυτό χωρίς την Allreduce σημειώνουν σημαντικά καλύτερους χρόνους από το Challenge πρόγραμμα, ιδιαίτερα για τα μικρά μεγέθη προβλήματος. Συγκεκριμένα, παρατηρούμε για το κανονικό πρόγραμμα ότι παρουσιάζεται βελτίωση οπουδήποτε από 8% έως 65%! Η συνήθης βελτίωση είναι γύρω στο 50%.

Ας εξετάσουμε τα profiling αρχεία. Συγκρίνοντας το profiling αρχείο του Challenge για μέγεθος προβλήματος 840x840 με το αντίστοιχο profiling αρχείο του καθαρού MPI με Allreduce, παρατηρούμε ότι, στα top twenty callsites' aggregate times, οι Allreduce καταλαμβάνουν τις πρώτες θέσεις της λίστας και στα δύο. Μάλιστα, οι Allreduce του δικού μας προγράμματος λαμβάνουν περισσότερο χρόνο αθροιστικά σε σχέση με το challenge (32,9 msecs έναντι 29,22)! Σημαντική διαφορά όμως παρατηρείται στην επικοινωνία. Το άθροισμα των sendrecv του Challenge προγράμματος μας δίνει 23,78 msecs, ενώ το άθροισμα των startall/waitall του δικού μας προγράμματος δίνει μόλις 9,681 msecs. Αυτό δείχνει τον θετικό αντίκτυπο του persistent & non blocking communication στην απόδοση του προγράμματός μας.

Δικό μας πρόγραμμα:

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                Site       Time    App%    MPI%      Count    COV
Allreduce             30       6.68    1.27   15.11          1   0.00
Allreduce             19       6.67    1.27   15.08         49   0.00
Allreduce             63       6.41    1.22   14.49         49   0.00
Allreduce             73        6.4    1.22   14.48          1   0.00
Allreduce             41       3.45    0.66    7.80         49   0.00
Allreduce             85       3.29    0.63    7.43         49   0.00
Startall              55       1.04    0.20    2.35         49   0.00
Startall              77       1.02    0.19    2.29         49   0.00
Waitall               81          1    0.19    2.26         49   0.00
Startall              33      0.987    0.19    2.23         49   0.00
Waitall               59      0.949    0.18    2.14         49   0.00
Waitall               14      0.922    0.18    2.08         49   0.00
Startall              11      0.912    0.17    2.06         49   0.00
Waitall               38      0.908    0.17    2.05         49   0.00
Startall              80      0.399    0.08    0.90         49   0.00
Startall              37      0.352    0.07    0.80         49   0.00
Startall              60      0.314    0.06    0.71         49   0.00
Startall              15        0.3    0.06    0.68         49   0.00
Waitall               45       0.29    0.06    0.66         49   0.00
Waitall               23      0.288    0.05    0.65         49   0.00
```

Challenge πρόγραμμα:

```
---------------------------------------------------------------------------
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
---------------------------------------------------------------------------
Call                Site       Time    App%    MPI%      Count    COV
Allreduce              2       7.73    0.87   14.60         50   0.00
Allreduce              8       7.32    0.83   13.83         50   0.00
Allreduce             11       7.28    0.82   13.75         50   0.00
Allreduce              5       6.83    0.77   12.90         50   0.00
Sendrecv               7       6.35    0.72   12.00         50   0.00
Sendrecv              10       4.19    0.47    7.91         50   0.00
Sendrecv               3        3.8    0.43    7.17         50   0.00
Sendrecv               4       3.77    0.43    7.13         50   0.00
Sendrecv               6       2.71    0.31    5.11         50   0.00
Sendrecv               9       1.28    0.14    2.43         50   0.00
Sendrecv               1      0.915    0.10    1.73         50   0.00
Sendrecv              12      0.765    0.09    1.45         50   0.00
```

Τα profiling αρχεία για όλα τα μεγέθη προβλημάτων και με τους δύο τρόπους μετρήσεων, καθώς και ένας ξεχωριστός φάκελος με τα αρχεία από τα οποία συμπεριλάβαμε screenshots βρίσκονται στον φάκελο 'profiling' του φακέλου ParallelMPI.

VI. Υβριδικό MPI + OpenMP

1) Αλλαγές στην δομή του προγράμματος σε σχέση με το καθαρό MPI πρόγραμμα.

Για να μας είναι ευκολότερη η παραλληλοποίηση, η πρώτη επανάληψη του σώματος του while loop παύει να είναι ενσωματωμένη στον υπολογισμό των fY_squared, fX_squared. Επίσης, ο υπολογισμός αυτών ‘σπάει’ σε δύο ξεχωριστά for loops, τα οποία επίσης παραλληλοποιούνται. Επομένως, η δημιουργία των νημάτων γίνεται ακριβώς πριν τον υπολογισμό αυτό (οδηγία 16). Με τον ίδιο τρόπο παραλληλοποιούνται τα 4 for loops για τον υπολογισμό των στοιχείων της άλω. Όσον αφορά την παραλληλοποίηση του διπλού for, δοκιμάσαμε δύο τρόπους: με collapse και με παραλληλοποίηση του εξωτερικού for loop. Τα αποτελέσματα της μελέτης κλιμάκωσης σε έναν κόμβο για ορισμένους συνδυασμούς πλήθους διεργασιών – πλήθους νημάτων με τους δύο τρόπους παραλληλοποίησης του διπλού for ακολουθούν. Το master νήμα εκτελεί τα MPI operations (οδηγία 16), επομένως έχουμε 3 ζεύγη MPI_Barrier: ένα που περικλείει τις δύο MPI_Startall, ένα που περικλείει την MPI_Waitall μετά το διπλό for και ένα που περικλείει τις εναπομείνασες MPI λειτουργίες, καθώς και την ανταλλαγή των πινάκων u, u_old, send_requests_current και send_requests_former.

Μετρήσεις χωρίς ιδιαίτερο scheduling, με collapse στο διπλό for.

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	2δ-8ν Έκαστος	4δ-2ν Έκαστος	4δ-4ν Έκαστος
840x840	0,202	0,106	0,208	0,109	1,215	0,117	1,280
1680x1680	0,809	0,433	0,829	0,451	1,923	0,463	1,746
3360x3360	3,218	1,765	3,264	1,803	3,379	1,805	3,396
6720x6720	12,850	7,033	12,912	7,152	8,820	7,075	8,984
13440x13440	51,306	27,980	51,536	28,36	30,682	28,208	31,106
26880x26880	205,701	113,582	206,302	112,453	117,208	112,807	118,015

Μετρήσεις χωρίς ιδιαίτερο scheduling, με παραλληλοποίηση του εξωτερικού for μόνο στο διπλό for.

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	2δ-8ν Έκαστος	4δ-2ν Έκαστος	4δ-4ν Έκαστος
840x840	0,115	0,066	0,120	0,069	1,470	0,100	1,370
1680x1680	0,470	0,427	0,490	0,443	1,830	0,446	1,567
3360x3360	1,862	1,753	1,898	1,785	3,118	1,796	3,081
6720x6720	7,413	6,966	7,489	7,101	8,615	7,052	8,516
13440x13440	29,610	27,830	29,739	28,093	30,545	28,077	29,965
26880x26880	118,389	112,448	119,495	112,042	116,311	112,087	116,463

Τα κελιά που έχουν πράσινο χρώμα σημαίνει ότι η παραλληλοποίηση του εξωτερικού for τα πήγε καλύτερα για αυτόν τον συνδυασμό μεγέθους προβλήματος – πλήθους διεργασιών – πλήθους νημάτων απ’ότι το collapse. Ιδιαίτερα στην περίπτωση που υπάρχουν 4 νήματα συνολικά στον κόμβο, δηλαδή στις περιπτώσεις 1δ-4ν και 2δ-2ν, υπάρχει σημαντική βελτίωση με αυτόν τον τρόπο παραλληλοποίησης του διπλού for. Επομένως, καθώς τα 4 νήματα ανά κόμβο θα χρειαστούν στην

περίπτωση των 4 και 36 νημάτων συνολικά, θα προτιμήσουμε να κρατήσουμε αυτόν τον τρόπο παραλληλοποίησης και να πειραματιστούμε με το είδος του scheduling από εδώ και στο εξής.

Επίσης είναι προφανές ότι οι περιπτώσεις όπου έχουμε συνολικά 16 νήματα στον κόμβο δεν είναι καθόλου αποδοτικές, διότι προφανώς το overhead για την δημιουργία των νημάτων είναι υπερβολικά μεγάλο ακόμα και στα μεγαλύτερα μεγέθη προβλήματος. Επομένως στις περιπτώσεις νημάτων που ζητήθηκε να μελετήσουμε στην μελέτη κλιμάκωσης που είναι πολλαπλάσια του 16 (για 16 και 64 νήματα, συγκεκριμένα) δεν πρόκειται να αξιοποιήσουμε τους συνδυασμούς 2 διεργασίες με 8 νήματα έκαστος και 4 διεργασίες με 4 νήματα έκαστος.

2) Επιλογή τρόπου scheduling

Ακολουθεί μία μικρή μελέτη κλιμάκωσης για τους διάφορους τρόπους scheduling. Θεωρήσαμε ότι, καθώς κάθε επανάληψη εκτελεί το ίδιο πλήθος δουλειάς, το static scheduling θα έπρεπε να είναι το καλύτερο. Κρίναμε επίσης ότι το ιδανικότερο chunksize θα ήταν το πλήθος των επαναλήψεων δια το πλήθος των νημάτων που διαθέτει η συγκεκριμένη διεργασία, έτσι ώστε ο φόρτος των επαναλήψεων να μοιραστεί ομοιόμορφα στα διάφορα νήματα. Όμως, θελήσαμε να πειραματιστούμε με την κυκλική ανάθεση επαναλήψεων επίσης. Περιμέναμε ότι το dynamic και το guided scheduling θα είχαν χειρότερη απόδοση σε σχέση με το static scheduling, λόγω του overhead της επικοινωνίας των νημάτων για τον διαμοιρασμό του φόρτου δυναμικά.

Μετρήσεις με static scheduling και chunksize = 1

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	4δ-2ν Έκαστος
840x840	0,183	0,169	0,151	0,144	0,111
1680x1680	0,785	0,665	0,665	0,655	0,596
3360x3360	3,099	2,594	2,325	2,564	2,429
6720x6720	12,287	10,346	9,222	10,497	8,585
13440x13440	49,068	41,243	36,738	40,286	28,331
26880x26880	195,991	164,152	159,785	161,090	136,170

Εδώ παρατηρούμε σημαντικά χειρότερη απόδοση σε σχέση με τον τρόπο χωρίς ιδιαίτερο scheduling, ιδιαίτερα στα μεγαλύτερα μεγέθη προβλήματος! Προφανώς η κυκλική ανάθεση επαναλήψεων αποκλείεται.

Μετρήσεις με static scheduling και chunksize = (m_local – 2)/num_threads
(Εδώ το scheduling αφορούσε μόνο την παραλληλοποίηση του διπλού for, καθώς εκεί βρισκόταν το μεγαλύτερο μέρος του υπολογιστικού φόρτου του προγράμματος)

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	4δ-2ν Έκαστος
840x840	0,118	0,067	0,123	0,070	0,071
1680x1680	0,476	0,440	0,495	0,449	0,438
3360x3360	1,882	1,754	1,912	1,779	1,778
6720x6720	7,485	6,958	7,557	7,094	7,044
13440x13440	29,872	27,848	30,002	28,227	28,077

Δεν παρατηρήσαμε κάποια ουσιαστική διακύμανση στην απόδοση με αυτή την αλλαγή. Πιθανολογήσαμε ότι αυτό συμβαίνει γιατί ο τρόπος με τον οποίον γινόταν το scheduling ήταν by default αυτός.

Μετρήσεις με dynamic scheduling και chunksize = 1

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	4δ-2ν Έκαστος
840x840	0,189	0,150	0,165	0,128	0,138
1680x1680	0,787	0,657	0,659	0,677	0,598
3360x3360	3,098	2,601	2,556	2,660	2,354

Όπως ήταν αναμενόμενο, παρατηρήσαμε σημαντική πτώση στην απόδοση, αφού μετά το πέρας κάθε επανάληψης υπάρχει overhead επικοινωνίας των νημάτων ώστε να αποφασιστεί η επόμενη επανάληψη που θα τους ανατεθεί. Δεν συνεχίσαμε τις μετρήσεις για μεγαλύτερα μεγέθη προβλήματος.

Μετρήσεις με dynamic scheduling και chunksize = (m_local – 2)/omp_get_num_threads
(Εδώ το scheduling αφορούσε μόνο την παραλληλοποίηση του διπλού for, καθώς εκεί βρισκόταν το μεγαλύτερο μέρος του υπολογιστικού φόρτου του προγράμματος)

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	4δ-2ν Έκαστος
840x840	0,136	0,117	0,127	0,091	0,082
1680x1680	0,477	0,440	0,491	0,460	0,457
3360x3360	1,865	1,756	1,900	1,780	1,795
26880x26880	119,832	114,408	120,692	112,886	111,914

Εδώ, όσο μεγάλωνε το μέγεθος προβλήματος, οι χρόνοι ήταν πιο κοντά στον χωρίς scheduling τρόπο, ενώ στα μικρότερα μεγέθη τα πήγαινε χειρότερα, άρα δεν υφίσταται κάποιος λόγος να προτιμήσουμε αυτού του είδους το scheduling.

Μετρήσεις με guided scheduling, χωρίς συγκεκριμένο chunksize.

	1δ-4ν	1δ-8ν	2δ-2ν Έκαστος	2δ-4ν Έκαστος	4δ-2ν Έκαστος
840x840	0,136	0,108	0,126	0,098	0,080
1680x1680	0,480	0,454	0,496	0,459	0,457
3360x3360	1,874	1,775	1,898	1,803	1,779

Παρατηρώντας παρόμοιους χρόνους με αυτούς της προηγούμενης περίπτωσης, κρίναμε ότι δεν χρειαζόταν να συνεχίσουμε την μελέτη.

Κατόπιν των ανωτέρω μελετών, κρίναμε ότι ο λόγος για τον οποίον διαφέρουν ελάχιστα οι χρόνοι των διαφόρων τρόπων scheduling με εξαίρεση το μικρό overhead στα dynamic και guided scheduling, ήταν ότι οι επαναλήψεις είχαν τον ίδιο φόρτο εργασίας. Μετρήσαμε δειγματοληπτικά ορισμένα μεγέθη προβλήματος και ορισμένους συνδυασμούς διεργασιών & νημάτων για την

περίπτωση όπου το scheduling είναι τύπου static και ΔΕΝ ορίζουμε εμείς το chunksize, και πήραμε παρόμοιους χρόνους με την περίπτωση μετρήσεων χωρίς ιδιαίτερο scheduling και την περίπτωση μετρήσεων scheduling με chunksize πλήθος_επαναλήψεων/πλήθος_threads, επομένως αποφασίσαμε να κρατήσουμε αυτού του είδους την ανάθεση επαναλήψεων στα διάφορα νήματα.

3) Χωρισμός νημάτων σε διεργασίες

Σε κάθε πλήθος νημάτων, διαλέξαμε τον αποδοτικότερο από τους ανωτέρω συνδυασμούς που μπορούσε να διαιρέσει ακριβώς το πλήθος των νημάτων, και τον αξιοποιήσαμε σε πολλαπλούς κόμβους. Συγκεκριμένα:

- 4 νήματα: Οι επιλογές μας ήταν οι συνδυασμοί 1δ-4ν ή 2δ-2ν έκαστος. Παρατηρήσαμε ότι για όλα τα μεγέθη προβλήματος, ο συνδυασμός 1δ-4ν απέδιδε λίγο καλύτερα, άρα προτιμήσαμε αυτόν.
  - Η αντίστοιχη γραμμή του script γίνεται
    #PBS -l select=1:ncpus=8:mpiprocs=1:ompthreads=4:mem=16400000kb
- 36 νήματα: Ίδιο σκεπτικό με πάνω. 9 διεργασίες με 4 νήματα έκαστος.
  - Η αντίστοιχη γραμμή του script γίνεται
    #PBS -l select=9:ncpus=8:mpiprocs=1:ompthreads=4:mem=16400000kb
- 16 νήματα: Εδώ μπορούσαμε να εκμεταλλευτούμε όλους τους συνδυασμούς, αφού διαιρείται ακριβώς με το 8 και το 4. Θα προτιμήσουμε έναν συνδυασμό που δίνει 8 νήματα. Φαίνεται η περίπτωση 1δ-8ν να τα πάει ελάχιστα καλύτερα σε σχέση με τις άλλες, επομένως έχουμε 2 διεργασίες με 8 νήματα έκαστος.
  - Η αντίστοιχη γραμμή του script γίνεται
    #PBS -l select=2:ncpus=8:mpiprocs=1:ompthreads=8:mem=16400000kb
- 64 νήματα: Ίδιο σκεπτικό με πάνω. 8 διεργασίες με 8 νήματα έκαστος
  - Η αντίστοιχη γραμμή του script γίνεται
    #PBS -l select=8:ncpus=8:mpiprocs=1:ompthreads=8:mem=16400000kb
- 80 νήματα: Εδώ περιοριζόμαστε μόνο στους συνδυασμούς που δίνουν 8 νήματα στον κόμβο, που ούτως ή άλλως είναι αποδοτικότεροι. Άρα, 10 διεργασίες με 8 νήματα έκαστος.
  - Η αντίστοιχη γραμμή του script γίνεται
    #PBS -l select=10:ncpus=8:mpiprocs=1:ompthreads=8:mem=16400000kb

4) Μελέτη κλιμάκωσης

Χρόνοι (sec)					
#Threads ---------- Problem Size	4	16	36	64	80
840x840	0,116	0,043	0,035	0,027	0,035
1680x1680	0,470	0,156	0,074	0,077	0,065
3360x3360	1,863	0,894	0,234	0,171	0,135
6720x6720	7,406	3,516	0,901	0,975	0,764
13440x13440	29,630	13,987	3,434	3,608	2,914
26880x26880	120,350	56,591	14,329	14,232	11,514

Επιτάχυνση					
#Threads ---------- Problem Size	4	16	36	64	80
840x840	4,189	11,302	13,885	18,000	13,885
1680x1680	3,83	11,55	24,35	23,4	27,72
3360x3360	3,79	7,9	30,19	41,31	52,33
6720x6720	3,79	7,99	31,17	28,81	36,76
13440x13440	3,79	8,03	32,7	31,13	38,54
26880x26880	3,74	7,95	31,38	31,6	39,06

Αποδοτικότητα					
#Threads ---------- Problem Size	4	16	36	64	80
840x840	1,05	0,7	0,380	0,28	0,17
1680x1680	0,95	0,72	0,67	0,36	0,34
3360x3360	0,94	0,49	0,83	0,64	0,65
6720x6720	0,94	0,5	0,86	0,45	0,45
13440x13440	0,94	0,5	0,9	0,48	0,48
26880x26880	0,93	0,5	0,87	0,49	0,48

5) Σύγκριση απόδοσης με καθαρό MPI

Σύγκριση απόδοσης		
	4 διεργασίες καθαρό	4 νήματα υβριδικό
840x840	0,92	1,05
1680x1680	0,87	0,95
3360x3360	0,88	0,94
6720x6720	0,89	0,94
13440x13440	0,89	0,94
26880x26880	0,88	0,93

Μεγαλύτερη απόδοση με το υβριδικό για κάθε μέγεθος προβλήματος

Σύγκριση απόδοσης		
	16 διεργασίες καθαρό	16 νήματα υβριδικό
840x840	0,55	0,7
1680x1680	0,69	0,72
3360x3360	0,47	0,49

6720x6720	0,49	0,5
13440x13440	0,50	0,5
26880x26880	0,50	0,5

Μεγαλύτερη απόδοση το υβριδικό για τα πρώτα 4 μεγέθη προβλήματος, αν και δεν έχουμε μικρότερη βελτίωση σε σχέση με τα 4 νήματα.

Σύγκριση απόδοσης		
	36 διεργασίες καθαρό	36 νήματα υβριδικό
840x840	0,15	0,380
1680x1680	0,41	0,67
3360x3360	0,46	0,83
6720x6720	0,47	0,86
13440x13440	0,49	0,9
26880x26880	0,50	0,87

Καλύτερη απόδοση το υβριδικό για όλα τα μεγέθη προβλήματος, και με ιδιαίτερα βελτιωμένες επιδόσεις!

Σύγκριση απόδοσης		
	64 διεργασίες καθαρό	64 νήματα υβριδικό
840x840	0,14	0,28
1680x1680	0,35	0,36
3360x3360	0,56	0,64
6720x6720	0,45	0,45
13440x13440	0,48	0,48
26880x26880	0,49	0,49

Καλύτερη απόδοση το υβριδικό για τα πρώτα 3 μεγέθη προβλήματος.

Σύγκριση απόδοσης		
	80 διεργασίες καθαρό	80 νήματα υβριδικό
840x840	0,06	0,17
1680x1680	0,19	0,34
3360x3360	0,46	0,65
6720x6720	0,43	0,45
13440x13440	0,48	0,48
26880x26880	0,49	0,48

Καλύτερη απόδοση το υβριδικό για τα πρώτα 3 μεγέθη προβλήματος. Μικρή πτώση για το τελευταίο.

Συνολικά παρατηρούμε ότι με το υβριδικό πρόγραμμα πετυχαίνουμε καλύτερες επιδόσεις για τα τρία πρώτα μεγέθη προβλήματος, με στάσιμες μάλλον επιδόσεις για τα υπόλοιπα, σε σχέση με το καθαρό MPI. Εξαίρεση όμως αποτελούν τα 4 και 36 νήματα όπου παρουσιάζεται σημαντική βελτίωση για όλα τα μεγέθη προβλήματος! Υπενθυμίζουμε ότι αυτά τα πλήθη νημάτων είναι πολλαπλάσια μόνο του 4 και επομένως ο συνδυασμός 1δ-4ν ίσως είναι αποδοτικότερος πολλαπλασιαζόμενος σε πολλούς κόμβους σε σύγκριση με τον 1δ-8ν που χρησιμοποιήθηκε για τις υπόλοιπες περιπτώσεις.

VII) Πρόγραμμα CUDA

1) Πρόγραμμα με μία κάρτα γραφικών

Ο σχεδιασμός του προγράμματος CUDA έγκειται στην παραλληλοποίηση των εξής περιοχών του παλιού μας κώδικα: στην παραλληλοποίηση του υπολογισμού των συντεταγμένων (των τετραγώνων των fX, fY δηλαδή), στην παραλληλοποίηση του υπολογισμού του πίνακα u και στην παραλληλοποίηση του reduction του error.

Ξεκινώντας από την παραλληλοποίηση του υπολογισμού των fX, fY, επιλέξαμε να χρησιμοποιήσουμε 128 νήματα ανά μπλοκ (παντού χρησιμοποιούμε νήματα που είναι πολλαπλάσια του 32, όπως αναφέρεται ότι το CUDA ‘προτιμάει’ εδώ: Programming Guide :: CUDA Toolkit Documentation (nvidia.com) *“The multiprocessor creates, manages, schedules, and executes threads in groups of 32 parallel threads called warps.”*), και να βρούμε τον ελάχιστο αριθμό μπλοκ που χρειάζεται για να ανατεθεί ένα στοιχείο του αντίστοιχου πίνακα σε κάθε νήμα. Το τελευταίο μπλοκ δεν έχει πλήρες thread utilization, αλλά είναι ένα μικρό τίμημα γιατί έτσι τα υπόλοιπα έχουν. Ο kernel ελέγχει αν το στοιχείο είναι έγκυρο (δηλαδή, αποκλείει την χρήση ενός από τα ‘περιττά’ τελευταία) και υπολογίζει την αντίστοιχη τιμή.

Για τον υπολογισμό των πινάκων u, u_old, αποφασίσαμε να χρησιμοποιήσουμε δισδιάστατο πλέγμα τόσο για τα νήματα όσο και για τα μπλοκ, καθώς αυτά προσομοιάζουν στον τρόπο με τον οποίον τα δεδομένα χωρίζονταν σε διεργασίες στο καθαρό MPI πρόγραμμά μας. Αποφασίζοντας να αυξήσουμε τον αριθμό των νημάτων ανά μπλοκ, καθώς έχουμε περισσότερα στοιχεία να υπολογίσουμε, επιλέξαμε 256 νήματα ανά μπλοκ διατεταγμένα σε 16 επί 16 πλέγμα. Διαιρώντας την πλευρά του πίνακα u με το 16 βρίσκουμε και το αντίστοιχο πλήθος block *ανά διάσταση*. Σημειώνεται ότι ούτε εδώ έχουμε πλήρες thread utilization αλλά και πάλι αυτό συμβαίνει μόνο στα τελευταία block της μίας διάστασης. Κατόπιν ανάλογου ελέγχου εγκυρότητας, πάλι κάθε στοιχείο του πίνακα υπολογίζεται από ένα νήμα, και το error αποθηκεύεται στην αντίστοιχη θέση ενός πίνακα ιδίων διαστάσεων με των n, m που δίνονται.

Τα στοιχεία αυτού του πίνακα γίνονται reduce. Για να αξιοποιήσουμε τον κώδικα που δόθηκε στο eclass, κάνουμε zero padding στον πίνακα του error, καθώς ο κώδικας δουλεύει για πλήθος στοιχείων που είναι δύναμη του 2. Βρίσκουμε την πρώτη δύναμη του 2 που είναι μεγαλύτερη από το μέγεθος του προβλήματός μας. Τα στοιχεία που αντιστοιχούν σε κάποιο έγκυρο index του πίνακα u/u_old θα είναι μη μηδενικά, και τα υπόλοιπα από το στοιχείο n*m θα είναι μηδενικά. Τώρα, μπορούμε να αντιμετωπίσουμε τον πίνακα error ως μονοδιάστατο και επιλέγοντας να αναθέσουμε 2048 στοιχεία του zero-padded error πίνακα σε κάθε μπλοκ βρίσκουμε το ανάλογο πλήθος block κάθε φορά. Κάθε block αθροίζει τα 2048 στοιχεία που του έχουν ανατεθεί, και αποθηκεύει το αποτέλεσμα αυτό στην θέση του πίνακα error που αντιστοιχεί στο blockId του. Κατόπιν επαναϋπολογίζουμε τον αριθμό τον μπλοκ που χρειάζεται ώστε κάθε block να αθροίσει 2048 στοιχεία, δεδομένου ότι τώρα το πλήθος των στοιχείων είναι ίσο με το πλήθος των μπλοκ στον προηγούμενο γύρο και επομένως και πάλι δύναμη του 2. Αν το πλήθος αυτό γίνει μικρότερο από το 2048, δημιουργείται κατάλληλο zero padding στο πρώτο μπλοκ του πίνακα error και καλείται η reduce μία και τελευταία για αυτή την επανάληψη φορά και λαμβάνεται το τελικό error.

Σημειώνουμε ότι δεν έχει υλοποιηθεί παραλληλοποίηση της checkSolution.

2) Πρόγραμμα για 2 κάρτες γραφικών

Ξεκινάμε αυτή την ενότητα της αναφοράς μας αναφέροντας προκαταβολικά ότι το πρόγραμμα αυτό ΔΕΝ υπολογίζει σωστά το Residual. Όμως, καθώς λειτουργεί χωρίς segmentation fault, θελήσαμε να παρουσιάσουμε την δουλειά μας, έστω και ατελή.

Η σκέψη μας εδώ ήταν να μην διαμοιράσουμε την παραλληλοποίηση των fX, fY σε δύο κάρτες. Διατηρώντας τους υπολογισμούς σε μία κάρτα, χωρίζουμε μόνο τον πίνακα fY σε δύο κομμάτια, καθώς ο fX χρειάζεται ολόκληρος και στις δύο κάρτες. Οι πίνακες u και u_old μοιράζονται κατά σειρές: η πρώτη κάρτα παίρνει τις σειρές 0 έως m/2 – 1 και η δεύτερη τις σειρές m/2 έως m-1. Κάθε κάρτα έχει και μία σειρά άλω.

Κατόπιν, εφαρμόζουμε zero-padding στο μισό του μεγέθους προβλήματος, ώστε ο επιμέρους πίνακας σε κάθε κάρτα να είναι zero padded. Καλείται ο υπολογισμός της jacobi και στις δύο, και η CPU περιμένει να συγχρονιστούν, το ίδιο και για κάθε γύρο του error reduction. Αθροίζονται τα δύο reduced errors, και μετά ανταλάσσονται οι γραμμές της άλω: η τελευταία γραμμή της κάρτας 1 δίδεται στην κάρτα 2, και η πρώτη γραμμή της κάρτας 2 δίδεται στην 1. Για τις αντιγραφές μεσολαβεί η CPU και δεν έχει χρησιμοποιηθεί unified memory.

3) Χρόνοι εκτέλεσης για 1 κάρτα

	Χρόνος (sec)
840x840	0,022
1680x1680	0,081
3360x3360	0,305
6720x6720	1,027
13440x13440	3,750

Το μέγεθος προβλήματος 26880x26880 δεν τρέχει λόγω μνήμης

Μελετώντας τα profiling αρχεία, παρατηρούμε ότι σε όλα ανεξαιρέτως bottleneck στην απόδοση είναι οι κλήσεις προς την cudaMemcpy ή/και την cudaMalloc, και όχι οι κλήσεις προς τους δικούς μας kernels (φυσικά, ο kernel jacobi καταλαμβάνει ένα σημαντικό ποσοστό χρόνου, αλλά στην περίπτωση του μεγαλύτερου μεγέθους προβλήματος, εξακολουθούν οι κλήσεις στην cudaMemcpy να διαρκούν ένα τρίτο του χρόνου παραπάνω απ’ότι οι κλήσεις του εν λόγω kernel).

4) Χρόνοι εκτέλεσης για 2 κάρτες

	Χρόνος (sec)
840x840	0,016
1680x1680	0,049
3360x3360	0,18
6720x6720	0,608
13440x13440	2,046
26880x26880	0,035

Το τελευταίο μέγεθος προβλήματος φαίνεται να κάνει μόνο μία επανάληψη, καθώς φαίνεται να μην υπάρχει αρκετή μνήμη όπως εμφανίζεται στο αντίστοιχο profiling αρχείο. Δεν νομίζουμε ότι θα έπρεπε να συμβαίνει αυτό, αλλά δεν προλάβαμε να διαπιστώσουμε γιατί. Παρατίθεται απλά τυπικά.

Μελετώντας τα profiling αρχεία, παρατηρούμε παρόμοιους χρόνους GPU utilization μεταξύ των 2 devices, τόσο για τον jacobi kernel όσο και για τον error reduction kernel, κάτι το οποίο προσδοκούσαμε. Σε σύγκριση με τους χρόνους σε μία GPU, δεν παρατηρούμε ακριβή υποδιπλασιασμό του χρόνου. Αυτό είναι αναμενόμενο όμως, καθώς υπάρχει ένα overhead από τις κλήσεις για συγχρονισμό και από τις κλήσεις για την αντιγραφή δεδομένων από την GPU 0 στην GPU 1 (οι πίνακες για τα fY_squared, fX_squared) και για την ανταλλαγή της άλω.

Τα profiling αρχεία και για τις δύο περιπτώσεις βρίσκονται στους φακέλους res1GPU και res2GPU αντίστοιχα.

VII. Συμπεράσματα

Κατόπιν των μελετών που εκπονήθηκαν στα πλαίσια της εργασίας, εξάγαμε τα ακόλουθα συμπεράσματα.

Η μεθοδολογία του Foster έχει σημαντικό αντίκτυπο στην απόδοση του προγράμματός μας: πέρα από το είδος communication το οποίο επιλέξαμε, η σωστή διαχείριση του βήματος ένα (partitioning) βελτίωσε πολύ την απόδοση του προγράμματός μας σε σχέση με του Challenge προγράμματος, που δεν υλοποιεί τον ίδιο διαμοιρασμό δεδομένων με το δικό μας, και που παρότι τρέχει γρηγορότερα ακολουθιακά δεν αποδίδει εξίσου καλά παράλληλα.

Οι περισσότεροι πόροι δεν προσφέρουν πάντοτε καλύτερη απόδοση. Αυτό είναι έκδηλο σε πολλαπλά σημεία των μετρήσεών μας, όπως η χρήση 80 διεργασιών για όλα τα μεγέθη προβλήματος, ή η χρήση 16 νημάτων σε έναν κόμβο στο υβριδικό πρόγραμμα. Το overhead της επικοινωνίας δεν είναι αμελητέο και θα πρέπει να είναι το πολύ συγκρίσιμο με το κόστος του υπολογισμού, όχι μεγαλύτερο.

Ο παράλληλος προγραμματισμός στις κάρτες γραφικών διευκολύνει κατά πολύ τον πολυνηματικό προγραμματισμό, καθώς στην περίπτωση του OpenMP έπρεπε να εξεταστούν προσεκτικά οι συνδυασμοί διεργασιών και νημάτων και να αξιολογηθεί αν το overhead δημιουργίας νημάτων ήταν συμφέρον από πλευράς κόστους σε σχέση με το αντίστοιχο υπολογιστικό κόστος. Αυτή η διαδικασία δεν μας απασχόλησε καθόλου στην ανάπτυξη του CUDA προγράμματος, και αν συγκρίνουμε τους καθαρούς χρόνους εκτέλεσης των kernels στα profiling αρχεία του CUDA με το elapsed time στο υβριδικό πρόγραμμα έχουμε βελτίωση της απόδοσης.