Παράλληλα Συστήματα
Εργασία 2020-21

Σχεδιασμός, Ανάπτυξη και Αξιολόγηση Παραλλήλων Προγραμμάτων σε MPI, Υβριδικό Mpi+OpenMp Cuda που υλοποιούν την Jacobi with successive over-relaxation

Μέλη Ομάδας

Ονοματεπώνυμο	Αριθμός Μητρώου	Λογαριασμός argo
Ιωάννης Καπετανγεώργης	**1115201800061**	**argo187**
Δημήτρης Σιταράς	**1115201800178**	**argo220**

Παράδοση Εργασίας

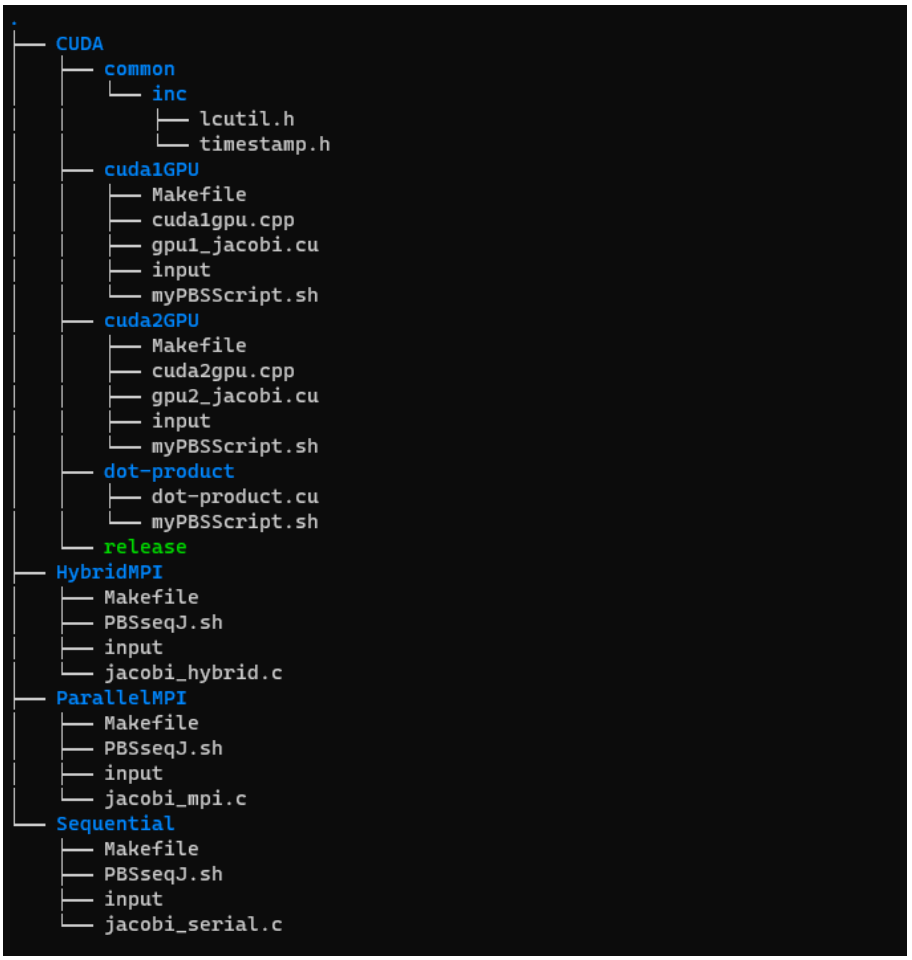
Ο παραδοτέος φάκελος έχει όνομα ProjectSubmission και έχει μεταφερθεί στο `/data/deliverables/argo187` από τον λογαριασμό argo187.

Περιεχόμενα

Εισαγωγή

Δομή Εργασίας

Ο παραδοτέος φάκελος έχει μεταφερθεί στο `/data/deliverables/argo187` και έχει όνομα **ProjectSubmission.**
Εσωτερικά, υπάρχουν 4 υποφακέλοι, ένας για κάθε κομμάτι της εργασίας. Οι φάκελοι έχουν τα εξής ονόματα Sequential, ParallelMPI, HybridMPI και CUDA.
Οι φάκελοι Sequential, ParallelMPI, HybridMPI περιέχουν το αντίστοιχο C αρχείο με τον κώδικα του προγράμματος, το αρχείο εισόδου input, ένα Makefile και το script. Εκτελώντας την εντολή make (έτσι ώστε να γίνει compile) και στην συνέχεια μέσω της qsub ακολουθούμενης από το όνομα του script εκτελείται με επιτυχία το αντίστοιχο πρόγραμμα.
Όσον αφορά τον φάκελο CUDA, περιέχει δύο υποφακέλους με ονόματα cuda1GPU και cuda2GPU οι οποίοι περιέχουν τις υλοποιήσεις για 1 gpu και 2 gpu αντιστοιχα. Δηλαδή η υλοποίηση για 1 και για 2 gpu έχει γίνει ξεχωριστά. Κάθε ένας από τους φακέλους cuda1GPU και cuda2GPU περιέχουν 2 αρχεία κώδικα (ένα .cpp και ένα .cu), το αρχείο input, το script και ένα Makefile.
Τέλος, στον φάκελο CUDA υπάρχουν οι φάκελοι common, dot-product και το αρχείο release τα οποία χρειάζονται για την εκτέλεση των προγραμμάτων CUDA.
Παρακάτω ακολουθεί μια εικόνα με την δομή του παραδοτέου φακέλου.



Πληρούνται όλες οι προϋποθέσεις/απαιτήσεις που αναφέρονται στην εκφώνηση της άσκησης και έχουν υλοποιηθεί όλα τα ζητούμενα.

Ακολουθιακό Πρόγραμμα

Στο πρώτο τμήμα της εργασίας βελτιώσαμε το ακολουθιακό προγραμμα έτσι ώστε να είναι έτοιμο για την παραλληλοποίηση του.
Οι βασικές βελτιώσεις που κάναμε σε σχέση με το αρχικό πρόγραμμα είναι οι εξής:

- Αφαιρέσαμε τις κλήσεις συναρτήσεων εσωτερικά του while loop καθώς είναι πολύ χρονοβόρες. Πιο συγκεκριμένα, ο κώδικας της συνάρτησης one_jacobi_iteration έχει τοποθετηθεί αυτούσιος εσωτερικά της επανάληψης και ώς αποτέλεσμα δεν υπάρχει καμία κλήση συνάρτησης.
- Παρατηρήσαμε πως οι μεταβλητές cx,cy και cc οι οποίες υπολογίζονταν σε κάθε κλήση της one_jacobi_iteration παραμένουν πάντα σταθερές, καθώς οι μεταβλητές deltaX, deltaY παραμένουν επίσης σταθέρες καθ' όλη την εκτέλεση του προγράμματος (έπειτα από τον αρχικό υπολογισμό τους). Έτσι, υπολογίζουμε μία φορά της μεταβλητές cx,cy και cc εξωτερικά του while loop αντί να υπολογίζονται σε κάθε επανάληψη.
- Ο υπολογισμός των fX και fY σε κάθε επανάληψη απαιτούσε μια πρόσθεση και έναν πολλαπλασιασμό καθώς κάθε φορά επαναυπολογιζόταν η νέα τιμή τους από την αρχή. Μετατρέψαμε τον υπολογισμό αυτό έτσι ώστε να χρειάζεται μόνο μια πρόσθεση, χρησιμοποιώντας την προηγούμενη τιμή των fX και fY.
- Αφαιρέσαμε περιττές αναθέσεις, όπως για παράδειγμα η ανάθεση της μεταβλητής f, μειώνοντας τες στις ελάχιστες δυνατές.
- Αφαιρέσαμε όλα τα defines, τοποθετώντας τις αντίστοιχες "εκφράσεις" των defines απευθείας στον κώδικα κάτι το οποίο προσφέρει μικρή μεν, αλλά αισθητή για μεγάλα μεγέθη προβλήματος, χρονική βελτίωση.

Στην συνέχεια ακολουθούν οι χρόνοι εκτέλεσης του βελτιωμένου ακολουθιακού προγράμματος για τα αντίστοιχα μεγέθη προβλήματος. Ο χρόνοι συγκρίνονται με το αρχικό ακολουθιακό πρόγραμμα.
Οι χρόνοι του παρακάτω πίνακα αναγράφονται σε δευτερόλεπτα.

Μέγεθος Προβλήματος	**Αρχικό πρόγραμμα**	**Βελτιωμένο πρόγραμμα**
840 x 840	0.837	0.657835
1680 x 1680	3.337	2.779478
3360 x 3360	13.336	12.684554
6720 x 6720	53.336	50.727522
13440 x 13440	213.313	202.956264
26880 x 26880	853.351	812.121603

MPI

Σχεδιασμός MPI Παραλληλοποίησης

Αρχικά, η διεργασία 0 διαβάζει όλες τις παραμέτρους από το αρχείο εισόδου μέσω της συνάρτησης readInputFile. Στην συνέχεια, μέσω broadcast (MPI_Bcast) ενημερώνει όλες τις υπόλοιπες διεργασίες για κάθε μία από τις παραμέτρους που διάβασε.
Στην συνέχεια ο αρχικός ενιαίος πίνακας διαμοιράζεται στις διεργασίες σε καρτεσιανή τοπολογία NxN, δηλαδή σε κάθε διεργασία αντιστοιχεί ένας πίνακας μεγέθους new_size x new_size, όπου:

new_size = n / $\sqrt{numberOfProcesses}$ (n : το μέγεθος του προβλήματος)

Επίσης, κάθε διεργασία χρειάζεται 2 γραμμές και 2 στήλες για την αποθήκευση των halos.
Άρα τελικα, κάθε διεργασία χρειάζεται δύο πίνακα μεγέθους (new_size+2)x(new_size+2), τον u και τον u_old.
Για τους πίνακες κάθε διεργασίας ισχύει:

- Η πρώτη γραμμή αντιστοιχεί στην τελευταία γραμμή του πίνακα του Βόρειου γείτονα
- Η τελευταία γραμμή αντιστοιχεί στην πρώτη γραμμή του πίνακα του Νότιου γείτονα
- Η πρώτη στήλη αντιστοιχεί στην τελευταία στήλη του πίνακα του Δυτικού γείτονα
- Η τελευταία στήλη αντιστοιχεί στην πρώτη στήλη του πίνακα του Ανατολικού γείτονα

Για τον σχεδιασμό του παράλληλου προγράμματος, πρέπει να ληφθεί υπόψη ότι αυτές οι γραμμές και στήλες (hallos) πρέπει να ανανεώνονται πριν από κάθε επανάληψη.
Έτσι, κάθε διεργασία πρέπει πριν από κάθε επανάληψη, να στέλνει τις “ακριανές” γραμμές/στήλες τις στους αντίστοιχους γείτονες, αλλά και να λαμβάνει τα hallos από τους αντίστοιχους γείτονες.
Το πως πραγματοποιείται αυτή η επικοινωνία εξηγείται στην αμέσως επόμενη παράγραφο.
Αφου κάθε διεργασία ενημερώσει τα hallos της ξεκινά τον υπολογισμό των στοιχείων του πίνακα που της αντιστοιχούν.
Έπειτα από τον υπολογισμό των άσπρων και πράσινων σημείων, γίνεται η εναλλαγή των πινάκων u και u_old.
Τέλος, υπολογίζεται συνολικό error από όλες τις διεργασίες μέσω ενός Allreduce.
Παραπάνω λεπτομέρειες για την υλοποίηση αλλα και για τις βελτιώσεις που κάναμε εξηγούνται στην αμέσως επόμενη παράγραφο Βελτιστοποίηση MPI κώδικα.

Βελτιστοποίηση MPI κώδικα

Η γενική ιδέα του παράλληλου προγράμματος δόθηκε στην παράγραφο Σχεδιασμός MPI Παραλληλοποίησης. Τώρα θα εξηγήσουμε τις λεπτομέρειες της υλοποίησης και ταυτόχρονα την βελτιστοποίηση του προγράμματος.

Τοπολογία διεργασιών

Για τον καθορισμό της καρτεσιανής τοπολογίας NxN των διεργασιών έχουμε χρησιμοποιήσει την συνάρτηση MPI_Cart_create.
Αρχικά ορίζουμε τις διαστάσεις μέσω της MPI_Dims_create με βάση τον αριθμό των διεργασιών.
Επίσης ενεργοποιούμε το rank reordering.
Στην συνέχεια (εκτός της κεντρικής επανάληψης) κάθε διεργασία βρίσκει τους 4 γείτονες της μέσω της MPI_Cart_shift αποθηκεύοντας τους σε έναν πίνακα τεσσάρων θέσεων.
Όσες διεργασίες δεν έχουν έναν γείτονα (π.χ. στον Βορρά) τότε η αντίστοιχη θέση του πίνακα περιέχει την τιμή MPI_PROC_NULL.

Χρήση datatypes για την αποστολή σειρών/στηλών

Όπως έχει ήδη αναφερθεί κάθε διεργασία πρέπει να “ανταλλάσει” σειρές και στήλες με τους γείτονες της.
Για να γίνεται αποδοτικά αυτή επικοινωνία έχουμε χρησιμοποιήσει προκαθορισμένα datatypes τόσο για τις στήλες όσο και για τις γραμμές.
Πιο συγκεκριμένα (πριν από την κεντρική επανάληψη) ορίζουμε δύο MPI_Datatype για τις γραμμές και τις στήλες μέσω των MPI_Type_contiguous και MPI_Type_vector.
Έτσι η αποστολή και η λήψη των hallos γίνεται αποκλειστικά και μόνο με την χρήση αυτών των datatypes.
Επίσης, λόγω αυτής της επιλογής, δεν χρειάζεται πρίν/μετά από κάθε αποστολή/λήψη να χρησιμοποιείται κάποιος ενδιάμεσος buffer κάτι που θα ήταν πολύ κοστοβόρο.

Αποστολή/Λήψη Μηνυμάτων

Η αποστολή και η λήψη των μηνυμάτων από τις διεργασίες γίνονται μέσω non-blocking κλήσεων των MPI_Isend και MPI_Irecv.
Οι εντολές Recv προηγούνται από τις Send έτσι ώστε οι διεργασίες να είναι έτοιμες να δεχτούν τα μηνύματα όταν αυτά σταλούν.
Μέσω των datatypes που έχουν οριστεί (και εξηγηθεί αμέσως παραπάνω) γίνεται η απευθείας αποστολή/λήψη των hallos στέλνοντας κάθε φορά ένα μήνυμα τύπου row_type ή collumn_type (όπου row_type και collumn_type τα δύο MPI_Datatype που έχουν οριστεί).
Τις κλήσεις MPI_Isend και MPI_Irecv ακολουθούν οι αντίστοιχες κλήσεις MPI_Wait στα Requests τα οποία προέκυψαν από τις πρώτες. Τα ακριβής σημεία όπου γίνονται οι κλήσεις MPI_Wait εξηγούνται στην συνέχεια.
Σε κάθε επανάληψη (του while loop) κάθε διεργασία στέλνει 4 μηνύματα (ένα σε κάθε γείτονα) και αντίστοιχα λαμβάνει 4 μηνύματα. Όσες διεργασίες δεν έχουν κάποιον γείτονα τότε επικοινωνούν με το MPI_PROC_NULL.

Επικάλυψη επικοινωνίας με υπολογισμούς

Λόγω της χρήσης των non-blocking συναρτήσεων MPI_Isend και MPI_Irecv (και των αντίστοιχων MPI_Wait) για την επικοινωνία μεταξύ των διεργασιών, μας δίνεται η δυνατότητα να επικαλύψουμε την επικοινωνία αυτή κάνοντας υπολογισμούς (αντί να “περιμένουμε” μέχρι να ολοκληρωθεί η επικοινωνία για να κάνουμε οποιονδήποτε υπολογισμό).
Πιο συγκεκριμένα, μέχρι να ολοκληρωθεί η μεταβίβαση των γραμμών/στηλών (hallos) μπορούμε να υπολογίσουμε όλα τα λευκά σημεία καθώς ο υπολογισμός τους δεν απαιτεί τιμές από τα σημεία των γραμμών/στηλών που περιμένει να λάβει η κάθε διεργασία.
Έτσι, έπειτα από τις 4 κλήσεις MPI_Irecv και τις 4 κλήσεις MPI_Isend μπορούμε να ξεκινήσουμε απευθείας τον υπολογισμό των άσπρων σημείων (ο οποίος γίνεται μέσω της κλήσης της συνάρτησης compute_whites) ενώ ταυτόχρονα πραγματοποιείται η μεταβίβαση των γραμμών στηλών.
Έπειτα από τον υπολογισμό των άσπρων σημείων, απομένει ο υπολογισμός των πράσινων. Έτσι, ακολουθούν 4 κλήσεις MPI_Wait στα αντίστοιχα receive request που προέκυψαν από τις κλήσεις MPI_Irecv έτσι ώστε να ολοκληρωθεί η λήψη των hallos.
Μόλις ληφθούν τα hallos, ξεκινά ο υπολογισμός των πράσινων σημείων από κάθε διεργασία μέσω της συνάρτησης compute greens.
Τέλος, ακολουθούν 4 κλήσεις MPI_Wait στα send requests που προέκυψαν από τις κλήσεις MPI_Isend και η εναλλαγή των πινάκων u και u_old (μέσω δεικτών).
Έτσι, έχουμε καταφέρει να μειώσουμε κατά πολύ τον αδρανή χρόνο, χωρίς να χρειάζεται να περιμένουμε να ολοκληρωθεί η επικοινωνία μεταξύ των διεργασιών κάνοντας όσο το δυνατόν περισσότερους υπολογισμούς γίνεται σε αυτό το χρονικό διάστημα.
Αυτό, έχει και την αντίστοιχη θετική επίπτωση στον χρόνο καθώς μειώνει κατά πολύ τον χρόνο εκτέλεσης ιδιαίτερα για μεγάλο μέγεθος προβλήματος όπου η επικοινωνία είναι αρκετά χρονοβόρα.

Σύνοψη Παράλληλου Προγράμματος

Συνοψίζοντας όλα όσα αναφέρθηκαν στις παραγράφους Σχεδιασμός MPI Παραλληλοποίησης και Βελτιστοποίηση MPI κώδικα, παρακάτω παρουσιάζουμε μια συνοπτική περιγραφή του συνολικού παράλληλου προγράμματος.
Με την σειρά, το πρόγραμμα ακολουθεί τα παρακάτω “βήματα”:

- Αρχικοποίηση του περιβάλλοντος MPI (MPI_Init).
- Η διεργασία 0 διαβάζει τις παραμέτρους από το αρχείο εισόδου και στην συνέχεια ενημερώνει μέσω broadcast τις υπόλοιπες διεργασίες.
- Κάθε διεργασία δεσμεύει χώρο για τους δύο πίνακες u και u_old. Το μέγεθος του κάθε πίνακα είναι (new_size+2)x(new_size+2) και περιγράφεται στην παράγραφο Σχεδιασμός MPI Παραλληλοποίησης.
- Δημιουργείται η τοπολογία των διεργασιών μέσω της MPI_Cart_create με reordering.
- Κάθε διεργασία αναγνωρίζει τους τέσσερις γείτονες της μεσω της MPI_Cart_shift.
- Δημιουργία των δύο datatypes για την αποστολή/λήψη στηλών και γραμμών μέσω των MPI_Type_contiguous και MPI_Type_vector.
- Ένα MPI_Barrier ακολουθούμενο από ένα MPI_Wtime έτσι ώστε να συγχρονιστούν οι διεργασίες και να ξεκινήσει η χρονομέτρηση της κεντρικής επανάληψης.
- Κεντρική επανάληψη (while loop) έως ότου εκτελεστεί ο μέγιστος αριθμός επαναλήψεων ή επιτευχθεί η επιθυμητή ακρίβεια στο error.
  Εσωτερικά της επανάληψης, έχουμε:
  - Τέσσερις κλήσεις MPI_Irecv, μια για κάθε γείτονα, έτσι ώστε να είναι έτοιμη η κάθε διεργασία να λάβει τις γραμμές/στήλες από τους γείτονες της.
  - Τέσσερις κλήσεις MPI_Isend, μια για κάθε γείτονα, έτσι ώστε να ξεκινήσει να στέλνει η κάθε διεργασία τις αντίστοιχες γραμμές/στήλες στους αντίστοιχους γείτονες.
  - Κλήση της συνάρτησης computeWhites η οποία υπολογίζει όλα τα εσωτερικά άσπρα σημεία έως ότου ληφθούν και ανανεωθούν τα hallos.
  - Τέσσερις κλήσεις MPI_Wait στο receive request που προέκυψαν από τις κλήσεις MPI_Irecv έτσι ώστε να ολοκληρωθεί η λήψη των hallos από τους γείτονες
  - Αφού ληφθούν οι hallos, μέσω της συνάρτησης computeGreens υπολογίζονται τα ακριανά πράσινα σημεία που χρειάζονται τις hallos.
  - Εναλλαγή των πινάκων u και u_old μέσω δεικτών.
  - Υπολογισμος του συνολικού error από όλες τις διεργασίες μέσω ενός MPI_Allreduce.
  - Τέσσερις κλήσεις MPI_Wait στο send request που προέκυψαν από τις κλήσεις MPI_Isend έτσι ώστε να ολοκληρωθεί η αποστολή των hallos στους τους γείτονες.
- Μόλις ολοκληρωθεί η κεντρική επανάληψη, ακολουθεί μια κλήση MPI_Wtime έτσι ώστε να ολοκληρωθεί η χρονομέτρηση του while loop.
- Κλήση της checkSolution από κάθε διεργασία στον αντίστοιχο της πίνακα u_old
- Υπολογισμός του συνολικού error της checkSolution μόνο στην διεργασία 0 αθροίζοντας τα τοπικά error κάθε διεργασίας μέσω μιας κλήσης MPI_Reduce
- Τέλος, η διεργασία 0 εκτυπώνει όλες τις πληροφορίες σχετικά με την εκτέλεση όπως: χρόνος εκτέλεσης, αριθμός επαναλήψεων, residual καθώς και το συνολικό error που προέκυψε από την checkSolution.

Μετρήσεις MPI Κώδικα

Στην συνέχεια παρουσιάζονται οι μετρήσεις του κώδικα MPI. Οι μετρήσεις έγιναν για κάθε συνδυασμό διεργασιών (4,9,16,25,36,49,64,80) και μεγέθους προβλήματος (840,1680,3360,6720,13440,26880). Επίσης, οι ίδιες μετρήσεις έχουν γίνει τόσο με τον έλεγχο σύγκλισης όσο και χωρίς αυτόν. Στις μετρήσεις χωρίς τον έλεγχο σύγκλισης έχει αφαιρεθεί η εντολή AllReduce μέσω της οποίας υπολογίζεται το συνολικό error καθώς και η συνθήκη σύγκλισης του while loop.

Μετρήσεις με τον έλεγχο σύγκλισης

Για κάθε μία μέτρηση παρουσιάζεται ο χρόνος (MPI Wall time) εκτέλεσης της κεντρικής επανάληψης (while loop), το τελικό residual καθώς και ο αριθμός των επαναλήψεων που εκτελέστηκαν.

Χρόνοι εκτέλεσης

	4	9	16	25	36	49	64	80
840	0.095925	0.055284	0.044244	0.059856	0.087488	0.064785	0.060223	0.080316
1680	0.445574	0.186315	0.138471	0.081581	0.095388	0.087403	0.071790	0.099623
3360	1.791409	1.116426	0.931440	0.553770	0.438372	0.247779	0.162015	0.135579
6720	7.034721	4.364373	3.604076	2.164947	1.644411	1.146974	0.961026	0.764825
13440	27.926241	17.230765	14.075085	8.433847	6.346712	4.376761	3.627915	2.909151
26880	112.247610	68.778193	33.475072	0.569369	0.504201	0.332833	0.294624	0.239936

Residual

	4	9	16	25	36	49	64	80
840	5.40306e-09	4.25e-09	3.40804e-09	3.06059e-09	2.76896e-09	2.56986e-09	2.74872e-09	2.52873e-09
1680	6.73635e-10	5.18552e-10	4.14811e-10	3.46421e-10	2.99123e-10	2.6517e-10	2.32168e-10	2.61635e-10
3360	8.42985e-11	6.45782e-11	5.12217e-11	4.22661e-11	3.59655e-11	3.13364e-11	2.78103e-11	2.89854e-11
6720	1.05493e-11	8.07429e-12	6.3931e-12	5.26093e-12	4.46074e-12	3.86936e-12	3.41623e-12	3.35999e-12
13440	1.3196e-12	1.00996e-12	7.99451e-13	6.57568e-13	5.57171e-13	4.82869e-13	4.2584e-13	3.82434e-13
26880	1.65015e-13	1.26302e-13	9.99988e-14	8.22976e-14	6.97349e-14	6.04336e-14	5.32903e-14	4.79158e-14

Αριθμός επαναλήψεων

	4	9	16	25	36	49	64	80
840	50	50	50	50	50	50	50	50
1680	50	50	50	50	50	50	50	50
3360	50	50	50	50	50	50	50	50
6720	50	50	50	50	50	50	50	50
13440	50	50	50	50	50	50	50	50
26880	50	50	30	1	1	1	1	1

Μετρήσεις χωρίς τον έλεγχο σύγκλισης

Για κάθε μία μέτρηση παρουσιάζεται ο χρόνος (MPI Wall time) εκτέλεσης της κεντρικής επανάληψης (while loop). Ο αριθμός των επαναλήψεων που εκτελέστηκαν είναι πάντα ίσος με 50 αφού δεν γίνεται ο έλεγχος σύγκλισης. Επίσης, το συνολικό residual δεν υπολογίζεται καθώς έχει αφαιρεθεί η εντολή Allreduce μέσω της οποίας γινόταν ο υπολογισμός του.

Χρόνοι εκτέλεσης

	4	9	16	25	36	49	64	80
840	0.061073	0.038000	0.032581	0.019704	0.051954	0.051653	0.030614	0.067295
1680	0.390233	0.195322	0.132075	0.068298	0.054132	0.056005	0.082742	0.099090
3360	1.786921	1.090756	0.887906	0.503353	0.386805	0.211315	0.141481	0.122678
6720	7.037339	4.315107	3.574513	2.149333	1.607281	1.092402	0.906838	0.758971
13440	27.889992	17.099075	14.063350	8.414589	6.305168	4.290970	3.570090	2.910039
26880	111.679246	67.967090	55.849755	33.265152	24.886841	17.075608	14.064306	0.228393

Σύγκριση μετρήσεων με και χωρίς τον έλεγχο σύγκλισης

Αρχικά, για τα μικρότερα μεγέθη (840,1680) παρατηρούμε ότι το πρόγραμμα χωρίς τον έλεγχο σύγκλισης είναι σημαντικά ταχύτερο. Αυτό είναι αναμενόμενο καθώς ο φόρτος των υπολογισμών είναι σχετικά μικρός κάνοντας έτσι τον χρόνο που ξοδεύεται για τις εντολές Allreduce να αποτελεί σημαντική καθυστέρηση. Αυτό μπορούμε να το επιβεβαιώσουμε και μέσω του profiling (mpiP).
Παρακάτω ακολουθούν δύο εικόνες από το mpiP των 2 εκτελέσεων με μέγεθος προβλήματος 1680 και αριθμό διεργασιών 4.

Πρόγραμμα με έλεγχο σύγκλισης

```
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
Call                 Site       Time    App%    MPI%      Count    COV
Allreduce              82       14.3    0.76   10.63         50   0.00
Allreduce              18       14.1    0.75   10.47         50   0.00
Allreduce              51         13    0.69    9.67         50   0.00
Allreduce             115       12.8    0.68    9.54         50   0.00
Wait                   10          5    0.27    3.72         50   0.00
Wait                   40       4.96    0.26    3.69         50   0.00
Wait                  104       4.96    0.26    3.68         50   0.00
Wait                   75       4.83    0.26    3.59         50   0.00
Cart_create            61       4.54    0.24    3.37          1   0.00
Cart_create           126       4.51    0.24    3.35          1   0.00
Cart_create            92       4.38    0.23    3.25          1   0.00
Cart_create            29       4.36    0.23    3.24          1   0.00
Wait                   79       2.77    0.15    2.06         50   0.00
Wait                   47       2.59    0.14    1.92         50   0.00
Isend                  49       2.06    0.11    1.53         50   0.00
Isend                 113       2.04    0.11    1.52         50   0.00
Isend                  85       2.01    0.11    1.49         50   0.00
Isend                  20          2    0.11    1.49         50   0.00
Wait                  109       1.66    0.09    1.23         50   0.00
Wait                   13       1.58    0.08    1.17         50   0.00
```

Πρόγραμμα χωρίς έλεγχο σύγκλισης

```
@--- Aggregate Time (top twenty, descending, milliseconds) ----------------
Call                 Site       Time    App%    MPI%      Count    COV
Wait                   93       7.87    0.47    7.67         50   0.00
Wait                  123       6.89    0.41    6.72         50   0.00
Wait                   28       5.78    0.35    5.63         50   0.00
Wait                   59       5.57    0.33    5.43         50   0.00
Cart_create           112       5.15    0.31    5.02          1   0.00
Barrier                68       5.08    0.30    4.95          1   0.00
Barrier                98       5.07    0.30    4.94          1   0.00
Cart_create            81       5.05    0.30    4.92          1   0.00
Cart_create            50       4.91    0.29    4.79          1   0.00
Cart_create            19       4.89    0.29    4.77          1   0.00
Reduce                  8       3.44    0.21    3.36          1   0.00
Isend                  67       2.56    0.15    2.50         50   0.00
Isend                   7       2.52    0.15    2.46         50   0.00
Wait                   62       2.52    0.15    2.45         50   0.00
Isend                  38       2.46    0.15    2.40         50   0.00
Isend                  99       2.41    0.14    2.35         50   0.00
Isend                  51       1.67    0.10    1.63         50   0.00
Wait                   30       1.65    0.10    1.61         50   0.00
Isend                  82       1.65    0.10    1.61         50   0.00
Isend                  13       1.63    0.10    1.58         50   0.00
```

Από τα παραπάνω στοιχεία συμπεραίνουμε πως οι κλήσεις Allreduce έχουν σημαντικό ποσοστό από τον χρόνο εκτέλεσης του MPI και είναι ο κύριος λόγος καθυστέρησης του προγράμματος. Έτσι, αφαιρώντας τις κλήσεις αυτές, εξαλείφουμε αυτές τις καθυστερήσεις μειώνοντας έτσι και τον χρόνο εκτέλεσης.
Άρα τόσο οι χρόνοι εκτέλεσης όσο και τα στοιχεία του mpiP επιβεβαιώνουν τα λεγόμενα μας.

Όσο αυξάνεται το μέγεθος του προβλήματος και οι υπολογισμοί καθώς και η αποστολή των μηνυμάτων γίνεται πιο χρονοβόρα, παρατηρούμε όλο και μικρότερες ποσοστιαίες διαφορές στους χρόνους εκτελέσεων των δύο μετρήσεων.
Πιο συγκεκριμένα για τα μεγέθη 3360, 6720, 13440 παρατηρούμε μικρότερες ποσοστιαίες διαφορές από ότι στα δύο πρώτα μεγέθη. Βέβαια το πρόγραμμα χωρίς τον έλεγχο σύγκλισης εξακολουθεί να είναι ταχύτερο.
Η μικρότερη διαφορά στους χρόνους οφείλεται στο γεγονός ότι ο χρόνος που ξοδεύεται για τις κλήσεις Allreduce είναι ποσοστιαία μικρότερος σε σχέση με τον συνολικό χρόνο εκτέλεσης, καθώς λόγω του μεγάλου μεγέθους του προβλήματος η αποστολή μηνυμάτων μεταξύ των διεργασιών είναι πιο χρονοβόρα.
Παρακάτω ακολουθούν δύο εικόνες από το mpiP των 2 εκτελέσεων με μέγεθος προβλήματος 13440 και αριθμό διεργασιών 16.

Πρόγραμμα με έλεγχο σύγκλισης

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

Call	Site	Time	App%	MPI%	Count	COV
Wait	10	140	0.12	19.08	50	0.00
Wait	107	133	0.11	18.03	50	0.00
Wait	13	98.1	0.08	13.33	50	0.00
Wait	47	92.1	0.08	12.51	50	0.00
Wait	77	89	0.08	12.09	50	0.00
Allreduce	114	41.7	0.04	5.67	50	0.00
Allreduce	18	36.6	0.03	4.97	50	0.00
Wait	111	19.9	0.02	2.71	50	0.00
Cart_create	29	18.6	0.02	2.52	1	0.00
Reduce	11	7.42	0.01	1.01	1	0.00
Allreduce	83	7.29	0.01	0.99	50	0.00
Allreduce	51	5.7	0.00	0.77	50	0.00
Wait	40	5.55	0.00	0.75	50	0.00
Cart_create	124	3.69	0.00	0.50	1	0.00
Barrier	23	2.94	0.00	0.40	1	0.00
Barrier	119	2.93	0.00	0.40	1	0.00
Bcast	52	2.49	0.00	0.34	1	0.00
Cart_create	61	1.7	0.00	0.23	1	0.00
Cart_create	94	1.63	0.00	0.22	1	0.00
Wait	16	1.46	0.00	0.20	50	0.00

Πρόγραμμα χωρίς έλεγχο σύγκλισης

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

Call	Site	Time	App%	MPI%	Count	COV
Wait	117	217	0.19	26.93	50	0.00
Wait	62	207	0.18	25.68	50	0.00
Wait	93	117	0.10	14.56	50	0.00
Wait	85	90.9	0.08	11.28	50	0.00
Wait	30	88.4	0.08	10.97	50	0.00
Wait	123	25	0.02	3.10	50	0.00
Cart_create	112	7.36	0.01	0.91	1	0.00
Wait	28	7.31	0.01	0.91	50	0.00
Cart_create	50	6.92	0.01	0.86	1	0.00
Cart_create	81	3.18	0.00	0.39	1	0.00
Reduce	8	2.3	0.00	0.29	1	0.00
Cart_create	19	1.48	0.00	0.18	1	0.00
Wait	124	1.44	0.00	0.18	50	0.00
Wait	31	1.24	0.00	0.15	50	0.00
Bcast	65	1.07	0.00	0.13	1	0.00
Isend	13	1.01	0.00	0.13	50	0.00
Isend	82	0.971	0.00	0.12	50	0.00
Wait	122	0.94	0.00	0.12	50	0.00
Bcast	97	0.94	0.00	0.12	1	0.00
Isend	51	0.932	0.00	0.12	50	0.00

Από τα παραπάνω στοιχεία συμπεραίνουμε ακριβώς όσα αναφέρθηκαν προηγουμένως. Δηλαδή, ότι ο περισσότερος χρόνος ξοδεύεται για τις κλήσεις Wait (δηλαδή για την αποστολή/λήψη των μηνυμάτων) και το ποσοστό του χρόνου που ξοδεύεται για τις κλήσεις Allreduce είναι σημαντικά μικρότερο.
Επομένως, είναι λογικό η ποσοστιαία μείωση του χρόνου να είναι μικρότερη όσο αυξάνεται το μέγεθος του προβλήματος.

Τέλος, όσον αφορά το μέγεθος προβλήματος 26880 και για αριθμό διεργασιών μεγαλύτερο των 9 παρατηρούμε πως το πρόγραμμα με τον έλεγχο σύγκλισης είναι ταχύτερο με τεράστια διαφορά.
Αυτό συμβαίνει καθώς το επιθυμητό tolerance επιτυγχάνεται σε λιγότερες από 50 επαναλήψεις. Αντίθετα, το πρόγραμμα χωρίς τον έλεγχο σύγκλισης εκτελεί και τις 50 επαναλήψεις.
Έτσι, προκύπτει και η μεγάλη διαφορά στους χρόνους εκτέλεσης των δύο μετρήσεων.

Μελέτη Κλιμάκωσης MPI

Για την μελέτη της κλιμάκωσης θα χρησιμοποιήσουμε την επιτάχυνση, την αποδοτικότητα καθώς και στοιχεία από το profiling των εκτελέσεων μέσω του mpiP.
Αρχικά, παραθέτουμε ξανά τους χρόνους εκτέλεσης του προγράμματος MPI (με τον έλεγχο σύγκλισης) οι οποίοι παρουσιαστηκαν και στην παράγραφο Μετρήσεις με τον έλεγχο σύγκλισης.

Χρόνοι εκτέλεσης

	4	9	16	25	36	49	64	80
840	0.095925	0.055284	0.044244	0.059856	0.087488	0.064785	0.060223	0.080316
1680	0.445574	0.186315	0.138471	0.081581	0.095388	0.087403	0.071790	0.099623
3360	1.791409	1.116426	0.931440	0.553770	0.438372	0.247779	0.162015	0.135579
6720	7.034721	4.364373	3.604076	2.164947	1.644411	1.146974	0.961026	0.764825
13440	27.926241	17.230765	14.075085	8.433847	6.346712	4.376761	3.627915	2.909151
26880	112.247610	68.778193	33.475072	0.569369	0.504201	0.332833	0.294624	0.239936

Για τον υπολογισμό της επιτάχυνσης, χρησιμοποιούμε τους παραπάνω χρόνους εκτέλεσης καθώς και τους χρόνους εκτέλεσης του σειριακού προγράμματος jacobi που μας δόθηκε.
Η επιτάχυνση προκύπτει από τον τύπο Speedup = T_serial / T_parallel.

Επιτάχυνση

	4	9	16	25	36	49	64	80
840	8.725566849	15.14000434	18.91781937	13.98356055	9.567026335	12.91965733	13.89834449	10.42133572
1680	7.489216157	17.91052787	24.0989088	40.90413209	34.98343607	38.17946752	46.48279705	33.49628097
3360	7.444419449	11.94526104	14.31761573	24.08220019	30.42165102	53.82215603	82.31336605	98.36331585
6720	7.581821653	12.22077031	14.79880003	24.63616892	32.43471371	46.50149001	55.49901876	69.73621416
13440	7.638442997	12.37977536	15.15536141	25.29249108	33.61000153	48.73763955	58.79768407	73.32482913
26880	7.602397949	12.4072902	25.49213337	1498.766178	1692.481768	2563.901416	2896.406946	3556.57758735

Αντίστοιχα υπολογίζουμε και την αποδοτικότητα μέσω του τύπου: Efficiency = T_serial / (T_parallel * n)
όπου n ο αριθμός των διεργασιών.

Αποδοτικότητα

	4	9	16	25	36	49	64	80
840	2,181391712	1,682222705	1,18236371	0,5593424218	0,2657507315	0,2636664761	0,2171616326	0.1302666965
1680	1,872304039	1,990058652	1,5061818	1,636165284	0,9717621131	0,7791728066	0,7262937039	0.418703512125
3360	1,861104862	1,327251226	0,8948509834	0,9632880077	0,8450458616	1,098411348	1,286146344	1.229541448125
6720	1,895455413	1,357863368	0,9249250016	0,9854467569	0,9009642696	0,9490100002	0,8671721681	0.871702677
13440	1,909610749	1,375530596	0,9472100879	1,011699643	0,9336111535	0,994645705	0,9187138136	0.916560364125
26880	1,900599487	1,3785878	1,593258336	59,95064712	47,01338245	52,32451869	45,25635853	44.45721984187

Σχολιασμός Επιτάχυνσης και Αποδοτικότητας

Παρατηρώντας τους πίνακες επιτάχυνσης και αποδοτικότητας παρατηρούμε πως:

- Για τα μικρότερα μεγέθη προβλήματος, αρχικά η επιτάχυνση αυξάνεται όσο αυξάνεται ο αριθμός των διεργασιών. Έπειτα από κάποιο σημείο η επιτάχυνση αρχίζει να μειώνεται όσο αυξάνεται ο αριθμός των διεργασιών.
  Αντίστοιχα παρατηρούμε ότι αρχικά η αποδοτικότητα είναι πολύ καλή (είναι μεγαλύτερη του 1) αλλά από κάποιο σημείο και μετά αρχίζει να μειώνεται σημαντικά.
  Αυτή η μείωση της επιτάχυνσης/αποδοτικότητας μπορεί να οφείλεται είτε στο γεγονός ότι δεσμεύονται πολλοί πόροι χωρίς να αξιοποιούνται πλήρως (λόγω του μικρού μεγέθους του προβλήματος), είτε σε μη αποδοτική επικοινωνία (Communication Efficiency).
  Τους λόγους αυτούς θα μελετήσουμε στην συνέχεια μέσω του mpiP.
- Για τα μεγαλύτερα μεγέθη παρατηρούμε η επιτάχυνση αυξάνεται συνεχώς όσο αυξάνεται ο αριθμός των διεργασιών, το οποίο σημαίνει ότι όσο αυξάνεται ο αριθμός των διεργασιών το πρόγραμμα μας χρειάζεται όλο και λιγότερο χρόνο. Όσον αφορά την αποδοτικότητα, παρατηρούμε ότι σε όλες τις περιπτώσεις είναι πολύ καλή καθώς είναι πολύ κοντά στο 1. Αυτό σημαίνει ότι αυξάνοντας τον αριθμό των διεργασιών το πρόγραμμα μας παραμένει πολύ αποδοτικό, μειώνοντας σημαντικά τον χρόνο εκτέλεσης και αξιοποιώντας (σχεδόν) πλήρως τους πόρους που δεσμεύονται.
  Επίσης παρατηρούμε πολύ μεγάλη απόδοση για μικρό αριθμό διεργασιών κάτι που σημαίνει ότι η μετάβαση από το σειριακό πρόγραμμα στο παράλληλο έχει πολύ σημαντικά οφέλη και μειώνει κατά πολύ τον χρόνο εκτέλεσης. Πιο συγκεκριμένα, με την μετάβαση από το σειριακό στο παράλληλο πρόγραμμα, δηλαδή με τον τετραπλασιασμό των διεργασιών από 1 σε 4, ο χρόνος εκτελέσεις είναι σχεδόν 8 φορές γρηγορότερος σε όλες τις περιπτώσεις ενώ θα περιμέναμε να είναι περίπου 4 φορές γρηγορότερος.
- Τέλος, όσον αφορά το μεγεθος 26880x26880 παρατηρούμε τεράστιες τιμές επιτάχυνσης και αποδοτικότητας. Αυτό οφείλεται (όπως αναφέρθηκε και παραπάνω) στο γεγονός ότι χρειάζεται μόνο μία επανάληψη για την επίτευξη του επιθυμητού residual ενώ το σειριακό εκτελεί 50 επαναλήψεις. Έτσι, υπάρχει τεράστια μείωση χρόνου όπως φαίνεται και από τις μετρήσεις.

Μελέτη Μέσω mpiP και δεικτών POP

Στην συνέχεια θα μελετήσουμε την αποδοτικότητα του προγράμματος μας χρησιμοποιώντας τα δεδομένα από το mpiP και από τους δείκτες POP που θα υπολογίσουμε από τα δεδομένα αυτα.

Αρχικά θα κάνουμε την μελέτη μας για τις περιπτώσεις όπου η αποδοτικότητα είναι χαμηλή έτσι ώστε να βρούμε τους λόγους στους οποίους οφείλεται.
Θα παρουσιάσουμε την μελέτη που κάναμε για την περίπτωση με μέγεθος προβλήματος 840x840 και 25 διεργασίες.
Στην περίπτωση αυτή παρατηρούμε ότι η αποδοτικότητα είναι σχετικά μικρή. Συγκεκριμένα είναι 0,559 δηλαδή αρκετά μικρότερη του 1.
Από τα δεδομένα του mpiP συλλέγουμε το AppTime, το MPITime και το MPI% για κάθε διεργασία.
Επίσης χρειαζόμαστε το RunTime δηλαδή τον συνολικό χρόνο εκτέλεσης του προγράμματος.
Στην συνέχεια υπολογίζουμε το User code, δηλαδή τον χρόνο εκτέλεσης χωρίς τις MPI εντολές, μέσω του AppTime και του MPITime.
Μέσω των παραπάνω μπορούμε να υπολογίσουμε το Load Balance, το Communication Efficiency καθώς και το Parallel Efficiency.
Τα αποτελέσματα που προκύπτουν για την συγκεκριμένη εκτέλεση είναι τα εξής:

RunTime	0,065856			
Load Balance	92,8627			
Communication	23,2325			
Parallel Efficienc	21,5743			
Task	AppTime	MPITime	MPI%	User code
0	0,0914	0,0772	84,4000	0,0142
1	0,0916	0,0764	83,4000	0,0152
2	0,0896	0,0756	84,3800	0,0140
3	0,0917	0,0779	84,9500	0,0138
4	0,0906	0,0760	83,9000	0,0146
5	0,0886	0,0753	84,9500	0,0133
6	0,0954	0,0814	85,2800	0,0140
7	0,0808	0,0666	82,4200	0,0142
8	0,0860	0,0719	83,5400	0,0141
9	0,0925	0,0778	84,0600	0,0147
10	0,0954	0,0814	85,3200	0,0140
11	0,0954	0,0815	85,3800	0,0139
12	0,0961	0,0816	84,8700	0,0145
13	0,0925	0,0786	84,9300	0,0139
14	0,0952	0,0810	85,0400	0,0142
15	0,0923	0,0774	83,8700	0,0149
16	0,0892	0,0747	83,6900	0,0145
17	0,0892	0,0748	83,8800	0,0144
18	0,0895	0,0752	84,0300	0,0143
19	0,0913	0,0760	83,2500	0,0153
20	0,0904	0,0756	83,6700	0,0148
21	0,0938	0,0799	85,1600	0,0139
22	0,0968	0,0834	86,1700	0,0134
23	0,0977	0,0843	86,3400	0,0134
24	0,0960	0,0823	85,7700	0,0137

Από τα παραπάνω αποτελέσματα παρατηρούμε τα εξής:

- Το Load Balance, δηλαδή ο διαμοιρασμός των δεδομένων στις διεργασίες είναι πολύ καλό (κοντά στο 100%) το οποίο σημαίνει ότι όλες οι διεργασίες έχουν σχεδόν ίδιο “φόρτο εργασιών”. Επομένως, δεν ευθύνεται ο διαμοιρασμός των δεδομένων για την όχι και τόσο καλή απόδοση του προγράμματος.
- Το Communication Efficiency είναι πολύ χαμηλό. Παρατηρώντας τα υπόλοιπα δεδομένα του mpiP, συμπεραίνουμε πως ένα πολύ μεγάλο μέρος του χρόνου εκτέλεσης οφείλεται στην επικοινωνία μεταξύ των διεργασιών. Δηλαδή οι διεργασίες ξοδεύουν περισσότερο χρόνο για την επικοινωνία παρά για τους υπολογισμούς. Σημαντικό ρόλο σε αυτό παίζει η εντολή AllReduce που εκτελείται σε κάθε επανάληψη για τον έλεγχο σύγκλισης, όπως έχει αναφερθεί και στην παράγραφο Σύγκριση μετρήσεων με και χωρίς τον έλεγχο σύγκλισης.
- Το Parallel Efficiency είναι επίσης χαμηλό. Για αυτό ευθύνεται το Communication Efficiency, καθώς το Load Balance είναι πολύ υψηλό.

Επομένως, συμπεραίνουμε πως για μικρά μεγέθη προβλήματος, από ένα σημείο και μετα όσο αυξάνεται ο αριθμός των διεργασιών τόσο αυξάνεται και ο χρόνος που ξοδεύουν οι διεργασίες για την επικοινωνία μεταξύ τους (κυρίως λόγω της εντολής AllReduce). Αντίθετα, όσο αυξάνεται ο αριθμός των διεργασιών μειώνεται ο χρόνος που ξοδεύει κάθε διεργασία για υπολογισμούς (αφού ανατίθεται όλο και μικρότερο μέρος του συνολικού πίνακα) και για μικρά μεγέθη προβλήματος ο χρόνος των υπολογισμών γίνεται αρκετά μικρός. Άρα, λόγω αυτής της αντίστροφης σχέσης χρόνου επικοινωνίας και χρόνου υπολογισμού κατα την αύξηση των διεργασιών για μικρά μεγέθη προβλήματος, η καθυστέρηση λόγω της επικοινωνίας έχει όλο και σημαντικότερη επίπτωση στην απόδοση του προγράμματος

Αντίστοιχα αποτελέσματα και συμπεράσματα βγάζουμε και για τις υπόλοιπες εκτελέσεις με μικρό μέγεθος προβλήματος και μικρή απόδοση, επομένως παραλείπεται η παρουσίαση τους.

Στην συνέχεια θα μελετήσουμε την περίπτωση όπου η αποδοτικότητα είναι πολύ καλή (μεγαλύτερη του 1). Συγκεκριμένα θα παρουσιάσουμε την μελέτη για την εκτέλεση με μέγεθος προβλήματος 13440x13440 και 4 διεργασίες όπου παρατηρούμε αποδοτικότητα ίση με 1,909.

Υπολογίζουμε τους δείκτες POP και παίρνουμε τα εξής αποτελέσματα:

RunTime	29,103000			
Load Balance	99,73			
Communication	99,97			
Parallel Efficienc	99,70			
Task	AppTime	MPITime	MPI%	User code
0	29,2	0,311	1,06	28,8890
1	29,2	0,105	0,36	29,0950
2	29,2	0,207	0,71	28,9930
3	29,2	0,113	0,39	29,0870

Από τα παραπάνω αποτελέσματα συμπεραίνουμε πως:

- Το Load Balance είναι σχεδόν άριστο (δηλαδή σχεδόν 100), άρα ο φόρτος είναι ομοιόμορφα καταμερισμένος σε κάθε διεργασία.
- Το Communication Efficiency είναι επίσης σχεδόν άριστο, το όποιο σημαίνει πως η επικοινωνία είναι και αυτή ομοιόμορφα κατανεμημένη στις διεργασίες και δεν ξοδεύεται περιττός χρόνος για την επικοινωνία μεταξύ διεργασιών.
- Κατ' επέκταση το Parallel Efficiency είναι επίσης σχεδόν άριστο, επιβεβαιώνοντας την πολύ καλή απόδοση που υπολογίσαμε προηγουμένως.

Σύνοψη Μελέτης Κλιμάκωσης

Με βάση όλα τα παραπάνω συμπεραίνουμε πως για τον κατάλληλο συνδυασμό μεγέθους προβλήματος και αριθμού διεργασιών το πρόγραμμα μας είναι πολύ αποδοτικό.
Η μόνες περιπτώσεις όπου το πρόγραμμα μας δεν είναι αποδοτικό είναι για μικρά μεγέθη προβλήματος (840 και 1680) και μεγάλο αριθμό διεργασιών για τους λόγους που εξηγήθηκαν παραπάνω.
Σε κάθε άλλη περίπτωση το πρόγραμμα μας είναι πολύ αποδοτικό.
Δηλαδή ο φόρτος εργασίας και ο διαμοιρασμός του πίνακα κατανέμεται ομοιόμορφα μεταξύ τον διεργασιών και η επικοινωνία μεταξύ των διεργασιών είναι πολύ αποδοτική.
Έτσι, με εξαίρεση τις περιπτώσεις μικρού μεγέθους προβλήματος, το πρόγραμμα μας κλιμακώνει πολύ καλα καθώς με την αύξηση του αριθμού των διεργασιών παρατηρούμε την αντίστοιχη μείωση στον χρόνο και μάλιστα πολλές φορές μεγαλύτερη μείωση από την αναμενόμενη.
Τέλος, όπως έχει ήδη αναφερθεί το πρόγραμμα μας έχει βέλτιστη απόδοση για το μεγαλύτερο μέγεθος προβλήματος 26880x26880 καθώς χρειάζεται μία μόνο επανάληψη για την επίτευξη του επιθυμητού residual.

MPI Vs Challenge

Στην συνέχεια θα συγκρίνουμε την απόδοση του δικού μας προγράμματος MPI σε σχέση με το πρόγραμμα Challenge που μας δόθηκε.
Στους δύο πίνακες που ακολουθούν παρουσιάζονται οι χρόνοι εκτέλεσης του προγράμματος Challenge καθώς και οι χρόνοι εκτέλεσης του δικού μας προγράμματος (οι οποίοι παρουσιάστηκαν στην παράγραφο Μελέτη Κλιμάκωσης MPI).

Χρόνοι εκτέλεσης Challenge

	4	9	16	25	36	49	64	80
840	0.22688	0.11971	0.10037	0.08954	0.08763	0.09641	0.08689	0.13570
1680	0.87016	0.47160	0.32298	0.22278	0.23462	0.21359	0.19984	0.26523
3360	3.40831	1.62819	1.04945	0.75847	0.59663	0.58142	0.53481	0.46837
6720	13.57835	6.39815	4.04134	2.77830	2.21629	2.00544	1.77323	1.66434
13440	54.28803	25.43402	15.88008	10.88747	8.58170	7.57285	6.65574	5.87089
26880	217.22396	113.30099	63.25852	42.85189	35.83707	29.40490	25.78698	22.73494

Χρόνοι εκτέλεσης MPI

	4	9	16	25	36	49	64	80
840	0.095925	0.055284	0.044244	0.059856	0.087488	0.064785	0.060223	0.080316
1680	0.445574	0.186315	0.138471	0.081581	0.095388	0.087403	0.071790	0.099623
3360	1.791409	1.116426	0.931440	0.553770	0.438372	0.247779	0.162015	0.135579
6720	7.034721	4.364373	3.604076	2.164947	1.644411	1.146974	0.961026	0.764825
13440	27.926241	17.230765	14.075085	8.433847	6.346712	4.376761	3.627915	2.909151
26880	112.247610	68.778193	33.475072	0.569369	0.504201	0.332833	0.294624	0.239936

Συγκρίνοντας τους χρόνους των παραπάνω πινάκων παρατηρούμε πως το δικό μας πρόγραμμα MPI έχει πολύ καλύτερη απόδοση από το Challenge.
Πιο συγκεκριμένα, σε όλες τις περιπτώσεις παρατηρείται σημαντική διαφορά στους χρόνους εκτέλεσης, με το δικό μας πρόγραμμα να χρειάζεται πολύ λιγότερο χρόνο για να εκτελεστεί.
Μάλιστα για μέγεθος 26880x26880 η διαφορά χρόνων είναι τεράστια καθώς το Challenge εκτελεί 50 επαναλήψεις ενώ το δικό μας μόνο 1.

Αυτό μπορεί να οφείλεται είτε σε άνιση κατανομή των δεδομένων στις διεργασίες καθώς και σε μη επικάλυψη της επικοινωνίας με υπολογισμούς.
Αντίθετα, το πρόγραμμα μας ακολουθεί την μεθοδολογία Foster και επίσης έχουν εφαρμοστεί διάφορες βελτιώσεις οι οποίες το κάνουν ακόμα αποδοτικότερο. Οι βελτιώσεις αυτές αναφέρονται αναλυτικά στην παράγραφο Βελτιστοποίηση MPI κώδικα, και όπως φαίνεται έχουν σημαντική επίπτωση στο χρόνο καθώς κάνουν το πρόγραμμα μας κατά πολύ γρηγορότερο.

Υβριδικό MPI+OpenMp

Σχεδιασμός υβριδικού προγράμματος

Για τον υβριδικό προγραμματισμό, η προσέγγιση που ακολουθήσαμε είναι να παραλληλοποιήσουμε τα for τα οποία υπάρχουν εσωτερικά της κεντρικής επανάληψης και είναι υπεύθυνα για τους υπολογισμούς.
Πιο συγκεκριμένα, παραλληλοποιήσαμε:

- Το διπλό for loop εσωτερικά της συνάρτησης computeWhites το οποίο είναι υπεύθυνο για τον υπολογισμό των εσωτερικών (άσπρων) σημείων.
- Τα δύο for loops εσωτερικά της συνάρτησης computeGreens τα οποία είναι υπεύθυνα για τον υπολογισμό των εξωτερικών (πράσινων) σημείων.

Οι τεχνικές που ακολουθήσαμε για την παραλληλοποίηση των for loops είναι:

- Για το διπλό for χρησιμοποιήσαμε collapse(2)
- Κάνουμε reduction στο error (reduction(+:temp_error)) έτσι ώστε να αθροίζεται συνεχώς από κάθε νήμα.
- Επιλέξαμε static scheduling καθώς έπειτα από δοκιμές μας έδινε τα καλύτερα αποτελέσματα. Όσον αφορά το chunk_size επιλέξαμε την default τιμή καθώς έδινε τα καλύτερα αποτελέσματα.
- Τα νήματα δημιουργούνται κάθε φορά πριν από κάθε for επανάληψη (εσωτερικά του κεντρικού while loop).
- Σχετικά με τον αριθμό διεργασιών-νημάτων παρατηρήσαμε καλύτερη απόδοση έχοντας δύο νήματα σε κάθε διεργασία. Για παράδειγμα για 16 νήματα επιλέγουμε 8 διεργασίες με 2 νήματα σε κάθε διεργασία κατανεμημένες σε δύο κόμβους. Δηλαδή η εντολή του script είναι:
  #PBS -l select=2:ncpus=8:mpiprocs=4:ompthreads=2:mem=16400000kb
- Στην παραλληλοποίηση κάθε for loop δηλώνουμε ποιές από τις μεταβλητές είναι κοινόχρηστες και ποιές ιδιωτικές μέσω των εντολών private και shared.

Μετρήσεις υβριδικού προγράμματος

Στην συνέχεια παρουσιάζονται οι μετρήσεις του υβριδικού κώδικα MPI+OpenMp. Οι μετρήσεις έγιναν για κάθε συνδυασμό νημάτων (4,16,36,64,80) και μεγέθους προβλήματος (840,1680,3360,6720,13440,26880). Σε κάθε εκτέλεση ο αριθμός των διεργασιών είναι ο μισός από αυτόν των νημάτων, δηλαδή σε κάθε διεργασία αντιστοιχούν 2 νήματα.

Χρόνοι εκτέλεσης

	4	16	36	64	80
840	0.110016	0.047876	0.453975	0.032924	0.069736
1680	0.467305	0.131415	0.077291	0.068106	0.055623
3360	1.839154	0.910388	0.417299	0.193659	0.200760
6720	7.227687	3.545727	1.624157	0.935773	0.754482
13440	28.675235	14.007106	6.308382	3.565517	2.855098
26880	208.105563	55.809435	0.488929	0.286512	0.235056

Residual

	4	16	36	64	80
840	6.00324e-09	4.33387e-09	5.2166e-11	2.78977e-09	2.64161e-09
1680	7.53883e-10	6.03107e-10	4.44457e-10	3.09668e-10	3.41247e-10
3360	9.44473e-11	7.17316e-11	5.28223e-11	4.16786e-11	3.86289e-11
6720	1.1819e-11	8.71461e-12	6.38292e-12	5.01141e-12	4.23963e-12
13440	1.47818e-12	1.07322e-12	7.7965e-13	6.07964e-13	5.63373e-13
26880	1.84823e-13	1.33134e-13	9.50932e-14	7.35811e-14	6.6421e-14

Αριθμός Επαναλήψεων

	4	16	36	64	80
840	50	50	50	50	50
1680	50	50	50	50	50
3360	50	50	50	50	50
6720	50	50	50	50	50
13440	50	50	50	50	50
26880	50	50	1	1	1

Μελέτη Κλιμάκωσης MPI+OpenMP

Για την μελέτη κλιμάκωσης ακολουθούμε την ίδια διαδικασία που περιγράφηκε στην Μελέτη Κλιμάκωσης MPI, δηλαδή θα χρησιμοποιήσουμε την επιτάχυνση και την αποδοτικότητα και στην συνέχεια στοιχεία από το mpiP.

Χρόνοι εκτέλεσης

	4	16	36	64	80
840	0.110016	0.047876	0.058862	0.032924	0.069736
1680	0.467305	0.131415	0.077291	0.068106	0.055623
3360	1.839154	0.910388	0.417299	0.193659	0.200760
6720	7.227687	3.545727	1.624157	0.935773	0.754482
13440	28.675235	14.007106	6.308382	3.565517	2.855098
26880	208.105563	55.809435	0.488929	0.286512	0.235056

Επιτάχυνση

	4	16	36	64	80
840	7,607984293	17,48266355	14,21970032	25,42218442	12.00240908
1680	7,140946491	25,39283948	43,17449638	48,9971515	59.99316829
3360	7,251160044	14,64869924	31,95790069	68,86331128	66.42757521
6720	7,379400907	15,04233123	32,83918981	56,99672891	70.69221001
13440	7,438927702	15,2289131	33,81421734	59,82666749	3.78418305
26880	4,100567941	15,29044327	1745,347484	2978,412772	3630.41573071

Αποδοτικότητα

	4	16	36	64	80
840	1,901996073	1,092666472	0,3949916754	0,3972216316	0.1500301135
1680	1,785236623	1,587052467	1,199291566	0,7655804922	0.749914603625
3360	1,812790011	0,9155437022	0,8877194636	1,075989239	0.830344690125
6720	1,844850227	0,9401457021	0,9121997169	0,8905738892	0.883652625125
13440	1,859731925	0,9518070685	0,9392838149	0,9347916796	0.047302288125
26880	1,025141985	0,9556527046	48,48187456	46,53769956	45.380196633875

Σχολιασμός Επιτάχυνσης και Αποδοτικότητας

Τα αποτελέσματα που παρατηρούμε από τους δύο παραπάνω πίνακες επιτάχυνσης και αποδοτικότητας είναι παρόμοια με αυτά που παρατηρήσαμε στο MPI. Δηλαδή:

- Όσον αφορά την επιτάχυνση, αυξάνεται συνέχεια όσο αυξάνεται ο αριθμός των διεργασιών/νημάτων. Με μόνη εξαίρεση την εκτέλεση για μέγεθος προβλήματος 840x840 και 36 διεργασίες. Σε κάθε άλλη περίπτωση το πρόγραμμα γίνεται όλο και γρηγορότερο με την αύξηση των νημάτων.
- Όσον αφορά την αποδοτικότητα, παρατηρούμε κατά κύριο λόγο πολύ καλές τιμές.
  Εξαίρεση αποτελεί το μέγεθος 840x840 για 36 και 64 νήματα.Αυτό μπορεί να οφείλεται είτε στο γεγονός ότι δεσμεύονται πολλοί πόροι χωρίς να αξιοποιούνται πλήρως (λόγω του μικρού μεγέθους του προβλήματος), είτε σε μη αποδοτική επικοινωνία (Communication Efficiency).
  Τους λόγους αυτούς θα μελετήσουμε στην συνέχεια μέσω του mpiP.
  Για μεγαλύτερα μεγέθη προβλήματος παρατηρούμε μεγάλη αποδοτικότητα (σχεδόν 1) και πολλές φορές και μεγαλύτερη της μονάδας. Έτσι, αυξάνοντας τον αριθμό των νημάτων το πρόγραμμα μας παραμένει πλήρως αποδοτικό αξιοποιόντας κατάλληλα όλους τους πόρους που δεσμεύει.
- Τέλος, για μέγεθος προβλήματος 26880x26880 και για 36 και 64 διεργασίες παρατηρούμε τεράστιες τιμές επιτάχυνσης και αποδοτικότητας. Αυτό οφείλεται στο γεγονός ότι το πρόγραμμα χρειάζεται μόνο 1 επανάληψη για να επιτύχει το επιθυμητό residual, όπως παρατηρήσαμε και στην μελέτη του MPI.

Μελέτη Μέσω mpiP και δεικτών POP

Στην συνέχεια θα μελετήσουμε την αποδοτικότητα του προγράμματος μας χρησιμοποιώντας τα δεδομένα από το mpiP και από τους δείκτες POP που θα υπολογίσουμε από τα δεδομένα αυτα.

Η διαδικασία που θα ακολουθήσουμε είναι αντίστοιχη με αυτή που ακολουθήσαμε στην μελέτη του καθαρού MPI.
Αρχικά θα κάνουμε την μελέτη μας για τις περιπτώσεις όπου η αποδοτικότητα είναι χαμηλή έτσι ώστε να βρούμε τους λόγους στους οποίους οφείλεται.
Θα παρουσιάσουμε την μελέτη που κάναμε για την περίπτωση με μέγεθος προβλήματος 840x840 και 64 νήματα. Στην συγκεκριμένη εκτέλεση η αποδοτικότητα είναι κατα πολύ μικρότερη του 1, συγκεκριμένα 0,3972.
Μέσω των δεδομένων από το mpiP, βρίσκουμε τους δείκτες POP και συγκεκριμένα υπολογίζουμε το Load Balance, το Communication Efficiency και το Parallel Efficiency.
Η συγκεκριμένη εκτέλεση αφορά 32 διεργασίες με 2 νήματα στην κάθε μία, δηλαδή συνολικά 64 νήματα.
Τα αποτελέσματα που προκύπτουν είναι τα εξής:

RunTime	0,032924			
Load Balance	88,67574257			
Communication	30,67671			
Parallel Efficienc	27,20280039			
Task	AppTime	MPITime	MPI%	User code
0	0,0592	0,0505	85,24	0,0087
1	0,0636	0,0548	86,16	0,0088
2	0,0601	0,0513	85,49	0,0088
3	0,0615	0,0523	85,11	0,0092
4	0,0592	0,0504	85,10	0,0088
5	0,0622	0,053	85,33	0,0092
6	0,0641	0,0552	86,12	0,0089
7	0,0592	0,0503	85,04	0,0089
8	0,061	0,0519	85,11	0,0091
9	0,0617	0,0529	85,64	0,0088
10	0,0608	0,0522	85,93	0,0086
11	0,0622	0,0534	85,85	0,0088
12	0,0609	0,0522	85,76	0,0087
13	0,0605	0,0512	84,60	0,0093
14	0,0612	0,0527	86,09	0,0085
15	0,0614	0,0527	85,81	0,0087
16	0,0617	0,0526	85,22	0,0091
17	0,0601	0,051	84,84	0,0091
18	0,0593	0,0504	85,01	0,0089
19	0,0631	0,0537	85,02	0,0094
20	0,0614	0,053	86,29	0,0084
21	0,061	0,0518	84,92	0,0092
22	0,0613	0,0522	85,20	0,0091
23	0,0619	0,0534	86,16	0,0085
24	0,0624	0,0536	85,85	0,0088
25	0,0614	0,0525	85,51	0,0089
26	0,0615	0,0525	85,37	0,0090
27	0,0623	0,0529	84,88	0,0094
28	0,0616	0,0526	85,41	0,0090
29	0,0591	0,049	82,91	0,0101
30	0,0586	0,0496	84,55	0,0090
31	0,0636	0,0547	86,05	0,0089

Από τα παραπάνω συμπεραίνουμε πώς:

- Το Load Balance, δηλαδή ο διαμοιρασμός των δεδομένων στις διεργασίες είναι αρκετά καλό άρα όλα τα νήματα έχουν σχεδόν ίδιο “φόρτο εργασιών”. Επομένως, δεν ευθύνεται ο διαμοιρασμός των δεδομένων για την όχι και τόσο καλή απόδοση του προγράμματος.
- Το Communication Efficiency είναι πολύ χαμηλό. Παρατηρώντας τα υπόλοιπα δεδομένα του mpiP, συμπεραίνουμε πως ένα πολύ μεγάλο μέρος του χρόνου εκτέλεσης οφείλεται στην επικοινωνία μεταξύ των διεργασιών αλλα και σε άλλες χρονοβόρες κλήσεις όπως ή Cart_create. Γενικότερα συμπεραίνουμε ότι στην συγκεκριμένη εκτέλεση οι διεργασίες/νήματα ξοδεύουν πολύ περισσότερο χρόνο σε κλήσεις MPI συναρτήσεων παρά σε υπολογισμούς.
- Το Parallel Efficiency είναι επίσης χαμηλό. Για αυτό ευθύνεται το Communication Efficiency, καθώς το Load Balance είναι αρκετά υψηλό.

Αντίστοιχα αποτελέσματα παίρνουμε και για την μελέτη της περίπτωσης 840x840 για 36 νήματα όπου επίσης παρατηρούμε μικρή αποδοτικότητα.

Στην συνέχεια θα μελετήσουμε την περίπτωση όπου η αποδοτικότητα είναι πολύ καλή (μεγαλύτερη του 1). Συγκεκριμένα θα παρουσιάσουμε την μελέτη για την εκτέλεση με μέγεθος προβλήματος 1680x1680 και 16 νήματα (δηλαδή 8 διεργασίες με 2 νήματα η κάθε μία) όπου παρατηρούμε αποδοτικότητα ίση με 1,587.

Υπολογίζουμε τους δείκτες POP και παίρνουμε τα εξής αποτελέσματα:

RunTime	0,131500			
Load Balance	96,66			
Communication	97,19			
Parallel Efficienc	93,94			
Task	AppTime	MPITime	MPI%	User code
0	0,168	0,0402	23,93	0,1278
1	0,164	0,0408	24,92	0,1232
2	0,169	0,0455	26,88	0,1235
3	0,164	0,0412	25,16	0,1228
4	0,166	0,0429	25,84	0,1231
5	0,167	0,0438	26,28	0,1232
6	0,167	0,0434	26,03	0,1236
7	0,167	0,0459	27,54	0,1211

Από τα παραπάνω συμπεραίνουμε πως τόσο το Load Balance όσο και το Communication Efficiency είναι πολύ υψηλά. Άρα τόσο τα δεδομένα προς υπολογισμό όσο και η επικοινωνία είναι ομοιόμορφα κατανεμημένη ανάμεσα στις διεργασίες/νήματα.
Έτσι επιβεβαιώνεται η πολύ καλή απόδοση του προγράμματος μας.

Σύνοψη Μελέτης Κλιμάκωσης

Με βάση όλα τα παραπάνω συμπεραίνουμε πως το πρόγραμμα μας είναι πολύ αποδοτικό σχεδόν σε όλες τις περιπτώσεις.
Η μόνες περιπτώσεις όπου το πρόγραμμα μας δεν είναι αποδοτικό είναι για το μέγεθος προβλήματος 840 και αριθμό νημάτων 36 και 64.
Σε κάθε άλλη περίπτωση το πρόγραμμα μας είναι πολύ αποδοτικό.
Δηλαδή ο φόρτος εργασίας και ο διαμοιρασμός του πίνακα κατανέμεται ομοιόμορφα μεταξύ τον διεργασιών/νημάτων και η επικοινωνία μεταξύ των διεργασιών είναι αποδοτική.
Έτσι το πρόγραμμα μας κλιμακώνει πολύ καλα καθώς με την αύξηση του αριθμού των διεργασιών/νημάτων παρατηρούμε την αντίστοιχη μείωση στον χρόνο και μάλιστα πολλές φορές μεγαλύτερη μείωση από την αναμενόμενη.

Σύγκριση Καθαρού MPI και Υβριδικού MPI+OpenMp

Στην συνέχεια θα συγκρίνουμε την απόδοση των δύο προγραμμάτων μας καθαρού MPI και υβριδικού MPI+OpenMp.
Παρακάτω παρουσιάζονται οι χρόνοι εκτέλεσης των δύο, οι οποίοι έχουν ήδη παρουσιαστεί στις αντίστοιχες παραγράφους.

Χρόνοι εκτέλεσης MPI

	4	9	16	25	36	49	64	80
840	0.095925	0.055284	0.044244	0.059856	0.087488	0.064785	0.060223	0.080316
1680	0.445574	0.186315	0.138471	0.081581	0.095388	0.087403	0.071790	0.099623
3360	1.791409	1.116426	0.931440	0.553770	0.438372	0.247779	0.162015	0.135579
6720	7.034721	4.364373	3.604076	2.164947	1.644411	1.146974	0.961026	0.764825
13440	27.926241	17.230765	14.075085	8.433847	6.346712	4.376761	3.627915	2.909151
26880	112.247610	68.778193	33.475072	0.569369	0.504201	0.332833	0.294624	0.239936

Χρόνοι εκτέλεσης MPI+OpenMp

	4	16	36	64	80
840	0.110016	0.047876	0.058862	0.032924	0.069736
1680	0.467305	0.131415	0.077291	0.068106	0.055623
3360	1.839154	0.910388	0.417299	0.193659	0.200760
6720	7.227687	3.545727	1.624157	0.935773	0.754482
13440	28.675235	14.007106	6.308382	3.565517	2.855098
26880	208.105563	55.809435	0.488929	0.286512	0.235056

Συγκρίνοντας τους παραπάνω πίνακες παρατηρούμε:

- Για μικρό αριθμό διεργασιών/νημάτων οι επιδόσεις είναι σχεδόν ίδιες.
  Πιο συγκεκριμένα συγκρίνοντας τις στήλες 4 και 16 διεργασιών του MPI με τις στήλες 4 και 16 νημάτων του Υβριδικού, παρατηρούμε ότι οι χρόνοι έχουν πολύ μικρές αποκλίσεις.
  Για 4 διεργασίες/νήματα το MPI έχει ελάχιστα καλύτερη απόδοση, ενώ για 16 διεργασίες/νήματα το Υβριδικό έχει ελάχιστα καλύτερη απόδοση
  Εξαίρεση σε αυτό αποτελεί το μέγεθος προβλήματος 26880x26880 όπου το MPI είναι αρκετά γρηγορότερο και στις δύο περιπτώσεις.
- Για 36 και 64 νήματα παρατηρούμε πως η απόδοση του υβριδικού προγράμματος βελτιώνεται. Πιο συγκεκριμένα, συγκρίνοντας τους χρόνους με τους αντίστοιχους χρόνους του καθαρού MPI για 36 και 64 διεργασίες, παρατηρούμε πως το υβριδικό είναι γρηγορότερο σε κάθε περίπτωση.

Τα παραπάνω μπορούμε να τα διαπιστώσουμε και συγκρίνοντας τους πίνακες επιτάχυνσης και αποδοτικότητας στις αντίστοιχες παραγράφους Μελέτη Κλιμάκωσης MPI και Μελέτη Κλιμάκωσης MPI+OpenMP

Προγραμματισμός CUDA

Πρόγραμμα CUDA σε 1 GPU

Σχεδιασμός Προγράμματος

Το πρόγραμμα μας αποτελείται από 2 αρχεία, το cuda1gpu.cpp και το gpu1_jacobi.cu.
Το αρχείο cuda1gpu.cpp είναι αντίστοιχο με το σειριακό πρόγραμμα. Πιο συγκεκριμένα:

- Διαβάζεται το αρχείο input.
- Δηλώνονται και αρχικοποιούνται όλες οι απαραίτητες μεταβλητές.
- Δεσμεύεται χώρος για τους πίνακες u και u_old.
- Εκτελείται η κεντρική επανάληψη (while loop).
    - Σε κάθε επανάληψη του while loop καλείται η συνάρτηση one_jacobi_iteration (του αρχείου gpu1_jacobi.cu) η οποία είναι υπεύθυνη για τον υπολογισμό των νέων τιμών του πίνακα.
    - Γίνεται η “αντιστροφή” των πινάκων u και u_old
- Μετά το τέλος της κεντρικής επανάληψης καλείται η συνάρτηση checkSolution.
- Εκτυπώνονται όλα τα αποτελέσματα και οι σχετικοί χρόνοι εκτέλεσης.

Όσον αφορά το αρχείο gpu1_jacobi.cu, υλοποιούνται δύο συναρτήσεις: η one_jacobi_iteration και η __global__ συνάρτηση compute.
Η one_jacobi_iteration είναι μια κλασική host code συνάρτηση (που τρέχει στην CPU) ενώ η compute είναι μια device code συνάρτηση,δηλαδή καλείται από την one_jacobi_iteration (host code) και εκτελείται στο device (GPU).
Η one_jacobi_iteration καλείται μία φορά σε κάθε επανάληψη του κεντρικού while loop και είναι υπεύθυνη για τον υπολογισμό όλων των νέων τιμών του πίνακα. Πιο συγκεκριμένα, τα “βήματα” που ακολουθεί είναι:

- Δέσμευση χώρου στο device για τους πίνακες u και u_old μέσω της cudaMalloc.
- Αντιγραφή των δεδομένων που περιέχουν οι πίνακες του host στους πίνακες του device που μόλις δεσμεύτηκαν, μέσω της cudaMemcpy.
- Δέσμευση χώρου και αρχικοποίηση της σε 0 για μία μεταβλητή η οποία είναι υπεύθυνη για τον υπολογισμό του συνολικού error. Περισσότερες λεπτομέρειες στην συνέχεια.
- Υπολογισμός των διαστάσεων του grid και των block. Ο υπολογισμός γίνεται ως εξής:

```
const int N = maxXCount*maxYCount;
const int BLOCK_SIZE = 512;
dim3 dimBl(BLOCK_SIZE);
dim3 dimGr(FRACTION_CEILING(N, BLOCK_SIZE));
```

- Κλήση της συνάρτησης compute με παραμέτρους <<<dimGr, dimBl>>>, έτσι ώστε να γίνει ο υπολογισμός των νέων σημείων από την GPU.
- Στην συνέχεια ακολουθεί μια κλήση της cudaDeviceSynchronize έτσι ώστε να περιμένουμε να ολοκληρωθούν όλες οι διεργασίες του device.
- Πλέον οι νέες τιμές που υπολογίστηκαν, βρίσκονται στον πίνακα του device. Έτσι, μέσω της cudaMemcpy αντιγράφουμε τις νέες τιμές από τον πίνακα του device στον πίνακα του host.
- Αποδεσμεύεται όλη η μνήμη που δεσμεύτηκε στο device, μέσω της cudaFree.
- Τέλος, υπολογίζεται το συνολικό error και επιστρέφεται. Περισσότερα για τον υπολογισμό του error εξηγούνται στην συνέχεια.

Η συνάρτηση compute εκτελείται στην GPU και είναι υπεύθυνη για τον υπολογισμό της νέας τιμής ενός σημείου του πίνακα.
Καλώντας την με τα κατάλληλα ορίσματα grid και blocks που εξηγήθηκαν παραπάνω, τελικά υπολογίζονται οι νέες τιμές για όλα τα στοιχεία του πίνακα. Πιο συγκεκριμένα, τα “βήματα” που ακολουθεί η compute είναι:

- Αρχικά εντοπίζεται σε ποιο index του πίνακα αντιστοιχεί η συγκεκριμένη κλήση. Το index υπολογίζεται ως εξης:

  $$\text{const unsigned int i} = \text{blockIdx.x} * \text{blockDim.x} + \text{threadIdx.x};$$
- Στην συνέχεια, γίνεται έλεγχος για το αν το index αυτό είναι εντός των ορίων του πίνακα που μας ενδιαφέρουν.
- Αν είναι, εκτελείται ο κώδικας για τον υπολογισμό της νέας τιμής του σημείου αυτού. Ο κώδικας για τον υπολογισμό της νέας τιμής, είναι αντίστοιχος με τον κώδικα που υπάρχει εσωτερικά του διπλού for της συνάρτησης one_jacobi_iteration στο σειριακό πρόγραμμα που μας δόθηκε.
- Τέλος ακολουθεί μια εντολή για τον υπολογισμό του error, η οποία εξηγείται στην συνέχεια.

Υπολογισμός του συνολικού error

Για τον υπολογισμό του συνολικού error χρησιμοποιείται μια κοινόχρηστη device μεταβλητή με όνομα totalError. Πιο συγκεκριμένα:

- Σε κάθε κλήση της one_jacobi_iteration (και πριν από την κλήση της compute) δεσμεύεται και αρχικοποιείται σε μηδέν η μεταβλητή totalError.
- Από τον υπολογισμό της νέας τιμής ενός σημείου προκύπτει ένα τοπικό error (error += updateVal*updateVal;). Έτσι, από κάθε κλήση της compute προκύπτει ένα τέτοιο error.
- Χρησιμοποιούμε μια ατομική συνάρτηση για το άθροισμα αυτών των τοπικών errors. Πιο συγκεκριμένα χρησιμοποιούμε την atomicAdd η οποία παρέχεται έτοιμη.
  Έτσι, αθροίζονται όλα τα τοπικά errors χωρίς να υπάρχει πρόβλημα λόγω του συγχρονισμού.
- Αφού ολοκληρωθούν όλες οι εκτελέσεις της compute και αθροιστούν όλα τα τοπικά errors το αποτέλεσμα αντιγράφεται σε μια μεταβλητή του host (tempErr), μέσω της cudaMemcpy.
- Το τελικό error, το οποίο και επιστρέφεται από την one_jacobi_iteration υπολογίζεται ως εξής:

  $$\text{sqrt(tempErr)/((maxXCount-2)*(maxYCount-2))}$$

Συνοψίζοντας, η διαδικασία για τον υπολογισμό του error είναι αντίστοιχη με αυτή του σειριακού προγράμματος. Δηλαδή, αρχικά αθροίζεται το error από τον υπολογισμό της νέας τιμής του κάθε σημείου (όπως γίνεται στο σειριακό πρόγραμμα εσωτερικά του διπλού for loop) και στην συνέχεια υπολογίζεται το τελικό error απο τον τύπο: sqrt(tempErr)/((maxXCount-2)*(maxYCount-2));
Το αθροισμα αυτο μας επιτρέπει να το κάνουμε με ασφάλεια και χωρίς λάθη η ατομική συνάρτηση atomicAdd.

Χρόνοι εκτέλεσης

Οι παρακάτω χρόνοι εκτέλεσης αντιστοιχούν σε δευτερόλεπτα.
Ο αριθμός επαναλήψεων σε όλες τις μετρήσεις είναι ίσος με 50.

	Χρόνος εκτέλεσης	Residual	Error of iterative solution
840	0.088641	5.40181e-09	0.000633904
1680	0.303162	6.75761e-10	0.000317239
3360	1.065935	8.44983e-11	0.000158679
6720	4.107931	1.05639e-11	7.93528e-05
13440	17.111026	1.32058e-12	3.96795e-05
26880	-	-	-

Παρατηρώντας τους χρόνους εκτέλεσης, βλέπουμε πως όσο κάθε φορά που αυξάνεται το μέγεθος του προβλήματος ο χρόνος εκτέλεσης τετραπλασιάζεται. Αυτή η συμπεριφορά είναι απολύτως λογική καθώς λόγω των δύο διαστάσεων του πίνακα, το μέγεθος του προβλήματος τετραπλασιάζεται.
Για παράδειγμα συγκρίνοντας τα μεγέθη 1680 και 840 έχουμε (1680*1680) / (840*840) = 4 , δηλαδή ότι το μέγεθος του προβλήματος είναι τετραπλάσιο.
Επομένως το πρόγραμμα μας έχει την αναμενόμενη συμπεριφορά.

Επίσης παρατηρούμε ότι το πρόγραμμα μας αδυνατεί να εκτελεστεί για μέγεθος προβλήματος 26880x26880 καθώς δεν επαρκεί η μνήμη της 1 GPU για να δεσμευτεί ο απαιτούμενος χώρος για τον πίνακα.

Πρόγραμμα CUDA σε 2 GPU

Σχεδιασμός Προγράμματος

Το πρόγραμμα μας αποτελείται από 2 αρχεία, το cuda2gpu.cpp και το gpu2_jacobi.cu.
Το αρχείο cuda2gpu.cpp είναι αντίστοιχο με το αρχείο cuda1gpu.cpp που περιγράφεται στην παράγραφο Πρόγραμμα CUDA σε 1 GPU.
Η μόνη διαφορά είναι ότι εσωτερικά του κεντρικού while loop, σε κάθε δημιουργούνται δύο threads (ένα για κάθε GPU). Στην συνέχεια κάθε ένα από τα threads καλεί την συνάρτηση one_jacobi_iteration δίνοντας σαν επιπλέον όρισμα το id του thread.
Επίσης οι πίνακες u και u_old χωρίζονται στην μέση, με το πρώτο thread να αναλαμβάνει το πρώτο μισό του πίνακα συν μία στήλη στα δεξιά και το δεύτερο thread να αναλαμβάνει το υπόλοιπο μισό συν μία στήλη στα αριστερά.
Έτσι, ανάλογα το thread, στην one_jacobi_iteration δίνεται σαν όρισμα οι διευθύνσεις των πινάκων u και u_old που αντιστοιχούν στο πρώτο και στο δεύτερο μισό του πίνακα.
Έπειτα από την ολοκλήρωση της one_jacobi_iteration τα threads καταστρέφονται και συνεχίζεται η σειριακή εκτέλεση του προγράμματος.

Όσον αφορά το αρχείο gpu2_jacobi.cu, ισχύουν επίσης όλα όσα αναφέρθηκαν την παράγραφο Πρόγραμμα CUDA σε 1 GPU, με τις εξής διαφορές:

- Με βάση το ID του thread στο οποίο εκτελείται η one_jacobi_iteration γίνεται το αντίστοιχο cudaSetDevice.
- Δεσμεύεται device χώρος ίσος με το μέγεθος του πίνακα το οποίο έχει ανατεθεί, δηλαδή το μισό από το μέγεθος του συνολικού προβλήματος συν μια επιπλέον στήλη.
- Έπειτα από τον υπολογισμό των νέων σημείων, δηλαδή από τις κλήσεις της compute, το πρώτο thread αντιγράφει όλες τις στήλες που του έχουν ανατεθεί πλην της τελευταίας στήλης που του έχει ανατεθεί (η οποία ανήκει στο δεύτερο thread, αλλά την χρειάζεται για τον υπολογισμό της πρό-τελευταίας στήλης) στον συνολικό πίνακα u του host. Αντίστοιχα το δεύτερο thread αντιγράφει όλες τις στήλες που του έχουν ανατεθεί (και για τις οποίες μόλις υπολόγισε τις νέες τιμές), πλην της πρώτης, στο δεύτερο μισό του πίνακα u του host.

Η συνάρτηση compute είναι ακριβώς ίδια με αυτή που περιγράφηκε παραπάνω για 1 GPU.

Υπολογισμός του συνολικού error

Για τον υπολογισμό του συνολικού error ακολουθείται η ίδια λογική με την 1 GPU.
Η μόνη διαφορά είναι ότι λόγω της ύπαρξης των 2 GPU:

- Κάθε gpu/thread υπολογίζει το δικό της error μέσω τον αθροισμάτων που γίνονται από τις atomicAdd που περιέχονται στην compute.
- Το error αυτό, που προκύπτει από το άθροισμα, επιστρέφεται από την συνάρτηση one_jacobi_iteration.
- Μέσω της επιλογής reduction(+:error) κατά την δημιουργία των thread, αθροίζονται τα δύο error που επιστρέφονται από τις κλήσεις της one_jacobi_iteration των 2 gpu/threads.
- Αφού καταστραφούν τα threads, υπολογίζεται το τελικό error μέσω της εντολής:

$$\text{error} = \text{sqrt}(\text{error})/((\text{n-2})*(\text{m-2}));$$

Χρόνοι εκτέλεσης

Οι παρακάτω χρόνοι εκτέλεσης αντιστοιχούν σε δευτερόλεπτα.
Ο αριθμός επαναλήψεων σε όλες τις μετρήσεις είναι ίσος με 50.

	Χρόνος εκτέλεσης	Residual	Error of iterative solution
840	0.04555	5.35402e-09	0.000634046
1680	0.162272	6.73007e-10	0.000317257
3360	0.549156	8.43127e-11	0.000158681
6720	2.075301	1.05523e-11	7.9353e-05
13440	8.21854	1.31985e-12	3.96796e-05
26880	33.58482	1.65033e-13	1.98405e-05

Αρχικά παρατηρούμε, όπως και στις εκτελέσεις με 1 GPU, πως όσο αυξάνεται το μέγεθος του προβλήματος οι χρόνοι τετραπλασιάζονται. Αυτό είναι απολύτως λογικό καθώς κάθε φορά στην ουσία τετραπλασιάζουμε το μέγεθος του προβλήματος λόγω των δύο διαστάσεων του πίνακα. Άρα το πρόγραμμα μας έχει την αναμενόμενη συμπεριφορά.

Συγκρίνοντας τους χρόνους με αυτούς που παρουσιάστηκαν στο πρόγραμμα για 1 GPU παρατηρούμε ότι έχουν μειωθεί στο μισό. Δηλαδή το πρόγραμμα μας έχει διπλάσια απόδοση, όπως είναι και αναμενόμενο καθώς διπλασιάσαμε τον αριθμό των GPU αλλά και κάναμε το κατάλληλο load balancing έτσι ώστε και οι 2 GPU να έχουν τον ίδιο φόρτο εργασίας.

Επίσης παρατηρούμε ότι το πρόγραμμα μας πλέον μπορεί να λύσει το πρόβλημα 26880x26880 και μάλιστα σε πολύ ικανοποιητικό χρόνο. Πλέον, εφόσον οι πίνακες χωρίζονται στις 2 gpu και χρειάζεται να αποθηκευτεί μόνο ο μισός στην κάθε μία, ο μνήμη επαρκεί και το πρόγραμμα μας εκτελείται με επιτυχία.