ΠΑΡΑΛΛΗΛΑ ΣΥΣΤΗΜΑΤΑ
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ 2017-2018
ΕΡΓΑΣΙΑ ΣΥΝΕΛΙΞΗ ΕΙΚΟΝΩΝ

ΣΥΜΜΕΤΕΧΟΝΤΕΣ ΣΤΗΝ ΟΜΑΔΑ:

ΕΥΑΓΓΕΛΙΟΥ ΠΑΝΑΓΙΩΤΗΣ **ΑΜ:1115201500039**
ΜΑΣΤΟΡΑΚΗΣ ΕΥΣΤΑΘΙΟΣ-ΑΝΔΡΕΑΣ **ΑΜ:1115201500092**

Το παρόν pdf περιέχει:

Οι **μετρήσεις καθαυτές** βρίσκονται στο pdf με όνομα **Timings.pdf** .

Εντολές για μεταγλώττιση των προγραμμάτων:

- Πρόγραμμα για εφαρμογή φίλτρου σε ασπρόμαυρη εικόνα *gray*
  - mpicc gray.c -o gray -lm
- Πρόγραμμα για εφαρμογή φίλτρου σε έγχρωμη εικόνα *gray*
  - mpicc color.c -o color -lm

Εντολές για εκτέλεση των προγραμμάτων:

- Πρόγραμμα για εφαρμογή φίλτρου σε ασπρόμαυρη εικόνα *gray*
  - mpiexec -n <number_of_processes> gray [-i inputFileName] [-o outputFileName] [-s rowsNumber colsNumber] [-f filterApplications]

    είτε αν θέλουμε να το τρέξουμε σε πολλές μηχανές

  - mpiexec -f ../machines -n <number_of_processes> gray [-i inputFileName] [-o outputFileName] [-s rowsNumber colsNumber] [-f filterApplications]

- Πρόγραμμα για εφαρμογή φίλτρου σε έγχρωμη εικόνα *color*
  - mpiexec -n <number_of_processes> color [-i inputFileName] [-o outputFileName] [-s rowsNumber colsNumber] [-f filterApplications]

    είτε αν θέλουμε να το τρέξουμε σε πολλές μηχανές

  - mpiexec -f ../machines -n <number_of_processes> color [-i inputFileName] [-o outputFileName] [-s rowsNumber colsNumber] [-f filterApplications]

Όπου:

- inputFileName είναι το όνομα του αρχείου που περιέχει την αρχική εικόνα

- outputFileName είναι το όνομα του αρχείου που θα περιέχει την τελική εικόνα μετά το τέλος του προγράμματος
- rowsNumber colsNumber είναι ο αριθμός των γραμμών και ο αριθμός των στηλών της εικόνας δηλαδή οι διαστάσεις της
- filterApplications είναι το πόσες φορές θέλουμε να εφαρμοστεί το φίλτρο στην εικόνα

Όλες οι παράμετροι που δίνονται κατά την εκτέλεση των προγραμμάτων gray και color είναι προαιρετικές και μπαίνουν με οποιαδήποτε σειρά. Αν δεν χρησιμοποιηθεί κάποια μεταβλητή τότε στο πρόγραμμα θα έχει μια default τιμή.

Default τιμές είναι οι εξείς:

- inputFileName είναι *waterfall_grey_1920_2520.raw* για το gray και *waterfall_1920_2520.raw* για το color, όπου για να λειτουργήσει η default αυτή τιμή θα πρέπει τα αντίστοιχα αρχεία των εικονών να είναι τοποθετημένα μέσα στον φάκελο
- outputFileName είναι outputGrey.raw για το gray και outputColor.raw για το color
- rowsNumber είναι 2520 και colsNumber είναι 1920 που αντιστοιχούν στις διαστάσεις των default inputFileNames
- filterApplications είναι 100

Μέσα στον παραδοτέο φάκελο υπάρχει ο φάκελος “Versions” όπου μέσα σε αυτόν υπάρχουν 5 φάκελοι, καθένας από τους οποίους αντιστοιχεί σε 1 version, 1 version από τα οποία είναι και το openMP, και καθένας περιέχει 1 πηγαίο αρχείο .c για το gray και 1 για το color. **Όλα τα versions βγάζουν τα ίδια και σωστά αποτελέσματα χωρίς κάποιο memory leak** (μετά από έλεγχο που έγινε με την χρήση του valgrind). **Όλα τα versions μεταγλωττίζονται και εκτελούνται με τον ίδιο τρόπο - *εκτός του Openmp που απαιτεί -openmp flag καθώς και ../openmpmachines ,όπου δίπλα από κάθε μηχανή υπάρχει :1 αντί για :4*** -, ο οποίος περιγράφτηκε παραπάνω.

Σημειώνεται ότι για τυχόν διευκόλυνση υπάρχει και ένα αρχείο Makefile μέσα σε κάθε φάκελο – version όπου περιλαμβάνει σχόλια για κάθε επιλογή. Ενδεικτικές επιλογές είναι οι παρακάτω:

- make: για μεταγλώττιση του κώδικα
- make mpip: για μεταγλώττιση του κώδικα για χρήση MPIP
- make rungray: για εκτέλεση του κώδικα για ασπρόμαυρες εικόνες
- make runcolor: για εκτέλεση του κώδικα για έγχρωμες εικόνες
- Αν οι δύο τελευταίες εντολές χρησιμοποιηθούν ως έχει θα εκτελέσουν το πρόγραμμα για τις default τιμές, δηλαδή με input αρχείο το waterfall_grey_1920_2520.raw για το gray και waterfall_1920_2520.raw για το color, output το outputGrey.raw για το gray και outputColor.raw για το color, για εικόνα 1920x2520 pixels, για 4 processes και εφαρμογή του φίλτρου 100 φορές.
- Για εκτέλεση με άλλες τιμές μπορείτε να αλλάξετε τις παρακάτω τιμές κατά τη χρήση των εντολών rungray/runcolor:
  - procs : αριθμός διεργασιών
  - filter : αριθμός εφαρμογών φίλτρου
  - input : όνομα/θέση input αρχείου
  - output : όνομα/θέση output αρχείου
  - rows : ύψος της input εικόνας
  - cols : πλάτος της input εικόνας
- Η αλλαγή αυτών γίνεται ως εξής:
  - make <rungray/runcolor> <όνομα παραμέτρου>=<τιμή>

- Παράδειγμα:
  - Για να τρέξει το πρόγραμμα 'gray' σε 16 διεργασίες αρκεί το εξής:
    - make rungray procs=16
  - Αντίστοιχα για να τρέξει με διαφορετικό input αρχείο σε 64 διεργασίες:
    - make rungray input=newImage.raw procs=64
- Κατά τη χρήση των rungray και runcolor εμφανίζονται και κάποια βοηθητικά μηνύματα για επιβεβαίωση σωστού input.

Δομή κώδικα σε κάθε version:

Original

- gray.c
  - Συνάρτηση unsigned char ** get2DArray(int rows, int cols) :
    - Η συνάρτηση αυτή χρησιμοποιείται προκειμένου να γίνει δέσμευση ενός “δισδιάστατου” πίνακα *με συνεχόμενες θέσεις μνήμης* ώστε να μπορεί να λειτουργήσει το MPI_Scatterv καθώς και τα MPI_Isend και MPI_Irecv που απαιτούν οι buffers που περνιούνται να είναι σε συνεχόμενες θέσεις μνήμης. Για να το πετύχουμε αυτό δεσμεύουμε έναν “μονοδιάστατο” πίνακα από unsigned char μεγέθους rows * cols και στην συνέχεια δεσμεύουμε έναν “μονοδιάστατο” πίνακα δεικτών σε unsigned char μεγέθους rows. Έπειτα πάμε και τοποθετούμε κάθε δείκτη του δεύτερου πίνακα να δείχνει στην αντίστοιχη “γραμμή” που θα είχαμε αν ο πρώτος πίνακας ήταν δισδιάστατος. Τέλος επιστρέφει των δεύτερο πίνακα, ο οποίος πετύχαμε να λειτουργεί σαν δισδιάστατος πίνακας με συνεχόμενες θέσεις μνήμης.

  - Δομές που χρησιμοποιούμε:
    - MPI_Datatype type,resizedtype για την αναπαράσταση του subarray κάθε process
    - MPI_Datatype column για την αναπαράσταση της στήλης που θα στέλνει κάθε process σε κάποιους γείτονές του
    - MPI_Datatype row για την αναπαράσταση της γραμμής που θα στέλνει κάθε process σε κάποιους γείτονές του
    - unsigned char **array2D και **final2D για την αναπαράσταση ολόκληρου του αρχικού πίνακα (που θα διαβάσει από το inputFileName το process 0) και ολόκληρου του τελικού πίνακα (που θα γράψει στο outputFileName το process 0) αντίστοιχα
    - unsigned char **myArray και **myFinalArray και **temp για την αναπαράσταση του αρχικού subarray που θα πάρει κάθε process μέσω του Scatterv και το οποίο είναι μέρος του αρχικού μεγάλου array2D και για την αναπαράσταση του τελικού subarray που θα προκύψει από την εφαρμογή του φίλτρου πάνω στον myArray και το temp το χρησιμοποιούμε προκειμένου να γίνει swap μεταξύ των 2 αυτών πινάκων ώστε να ξαναεφαρμοστεί το φίλτρο
    - unsigned char *topRow,*bottomRow,*leftCol,*rightCol και unsigned char leftUpCorn,rightUpCorn,rightDownCorn,leftDownCorn για την αναπαράσταση των γραμμών,στηλών και γωνιών που θα ανταλλάσσουν οι γείτονες μεταξύ τους
    - struct neighbor και struct allNeighbors για την αναπαράσταση των γειτόνων κάθε process στο οποίο για κάθε γείτονα κρατάμε πληροφορίες όπως το rank του (int rank), το αν λάβαμε ή όχι την πληροφορία που μας έστειλε (char recieved), το request που κάναμε για να λάβουμε (MPI_Request recieveRequest) και το request που κάναμε για να στείλουμε (MPI_Request sendRequest) .

- Αρχικά ορίζουμε τις default τιμές των ορισμάτων και έπειτα κάνουμε έλεγχο για τα ορίσματα που δόθηκαν, ώστε να αντικαταστήσουμε κάποια default τιμή με αυτή που δόθηκε αν χρειαστεί.
- Στην συνέχεια υπολογίζουμε την ρίζα του πλήθους των διεργασιών (δηλαδή του comm_sz) διότι θα μας χρειαστεί στο να υπολογίσουμε πόσες στήλες και πόσες γραμμές θα πάρει κάθε process.
- Έπειτα δημιουργούμε τους MPI_Types που θα χρησιμοποιήσουμε και είναι οι εξείς:
  - MPI_Type_create_subarray όπου παίρνει σαν ορίσματα το 2 (αφού είναι 2διάστατος ο πίνακας που θέλουμε να αναπαραστήσουμε), τον πίνακα sizes (όπου έχει τις διαστάσεις του αρχικού μεγάλου array2D), τον πίνακα subsizes (όπου έχει τις διαστάσεις του subarray κάθε process δηλαδή rowsNumber/sqrt_comm_sz και colsNumber/sqrt_comm_sz), τον πίνακα starts (όπου δηλώνει ότι ο πρώτος subarray ξεκινάει από την διεύθυνση [0][0]) καθώς και τα υπόλοιπα ορίσματα για το τι τύπος ήταν ο παλιός και που να αποθηκευτεί ο καινούριος.
  - Έπειτα χρειάζεται να κάνουμε MPI_Type_create_resized για τον τύπο που μόλις δημιουργήσαμε ώστε να ορίσουμε ότι το lower bound του data_type θα είναι το 0 καθώς και το extent του που θα είναι (colsNumber/sqrt_comm_sz)*sizeof(unsigned char).
  - MPI_Type_contiguous για την αναπαράσταση της row που θα είναι colsNumber/sqrt_comm_sz συνεχόμενοι MPI_UNSIGNED_CHAR.
  - MPI_Type_vector για την αναπαράσταση της column, όπου θα έχει count rowsNumber/sqrt_comm_sz (δίοτι κάθε στήλη αποτελείται από στοιχεία όσο είναι το πλήθος των γραμμών), blocklength 1 και stride colsNumber/sqrt_comm_sz ώστε προκειμένου να φτάσουμε στο επόμενο στοιχείο να παραλείπουμε τα επόμενα colsNumber/sqrt_comm_sz στοιχεία.
- Εν συνεχεία το process 0 διαβάζει και αποθηκεύει ολόκληρη την εικόνα από το inputFileName.
- Με την χρήση του MPI_Scatterv μοιράζεται η εικόνα στις διεργασίες. Σημαντικό να αναφερθεί είναι ότι ο πίνακας counts έχει 1 παντού γιατί κάθε process παίρνει από 1 block, ο πίνακας displs έχει displs[k] = i*rowsNumber+j όπου i=0 μέχρι sqrt_comm_sz – 1 και j=0 μέχρι sqrt_comm_sz – 1, διότι αν για παράδειγμα έχουμε 4 processes, τότε:
  - το πρώτο block θα ξεκινάει παραλείποντας 0* rowsNumber + 0 = 0 extent block (μεγέθους colsNumber/sqrt_comm_sz*sizeof(unsigned char) όπως ορίστηκε παραπάνω)
  - το δεύτερο block θα ξεκινάει παραλείποντας 0* rowsNumber + 1 = 1 extent block
  - το τρίτο block θα ξεκινάει παραλείποντας 1* rowsNumber + 0 = rowsNumber extent blocks
  - το τέταρτο block θα ξεκινάει παραλείποντας 1* rowsNumber + 1 = rowsNumber + 1 extent blocks
- Καθορίζουμε ποιοι είναι οι γείτονες του κάθε process και στην συνέχεια βλέπουμε ποιοι γείτονες είναι MPI_PROC_NULL ανάλογα με το my_rank του κάθε process.
- Βάζουμε στους topRow,bottomRow κλπ buffers default τιμές που θα χρησιμοποιηθούν σε περίπτωση που κάποιο process έχει κάποιον MPI_PROC_NULL γείτονα αλλιώς θα αντικατασταθούν από την τιμή που θα πάρει από τον αντίστοιχο γείτονα.Έχουμε κάνει την παραδοχή ότι οι default τιμές, για παράδειγμα για την topRow, θα είναι η ίδια η topRow του subarray του process κ.ο.κ.
- Καλούμε την MPI_Barrier ώστε να περιμένουμε όλα τα processes να τελειώσουν με την προετοιμασία.
- “Ξεκινάμε” το χρονόμετρο (δηλαδή μετράμε τον χρόνο πριν ξεκινήσει το loop) .
- Στο for loop με το πόσες φορές θα εφαρμοστεί το φίλτρο κάνουμε κάθε φορά τις εξείς ενέργειες:

        - αρχικοποιούμε τα recieved flags σε 0
        - “στέλνουμε” 8 μηνύματα στους 8 γείτονες μας με χρήση non-blocking επικοινωνίας MPI_Isend ώστε να έχουμε **επικάλυψη επικοινωνίας με υπολογισμούς**
        - “λαμβάνουμε” 8 μηνύματα από τους 8 γείτονες μας με χρήση πάλι non-blocking επικοινωνίας MPI_Irecv
        - εφαρμόζουμε το φίλτρο στα εσωτερικά σημεία του πίνακα μας για τα οποία δε χρειαζόμαστε τις πληροφορίες από τους γείτονες
        - εφόσον όπως είπαμε χρησιμοποιούμε non-blocking επικοινωνίας έχουμε ένα while loop που κάνουμε κάθε φορά έλεγχο στο τι λάβαμε, μέχρι να λάβουμε και τα 8 μηνύματα (δηλαδή να γίνει counterItems == 8) και κάνουμε τις απαραίτητες εφαρμογές φίλτρου
        - αξίζει να σημειωθεί ότι αν λάβουμε κάποια γραμμή ή στήλη θα κάνουμε εφαρμογή του φίλτρου μόνο στο “εσωτερικό” της γραμμής ή στήλης αυτής και όχι στις άκρες/γωνίες της διότι μπορεί να μην έχουμε λάβει τις γωνίες από τους γείτονες
        - για την εφαρμογή του φίλτρου στις γωνίες, χρειάζεται να έχουμε λάβει και τις άλλες 2 πλευρές που χρειαζόμαστε
        - τέλος, περιμένουμε να σταλθούν όλα τα Isend και έπειτα κάνουμε swap τους 2 array ώστε αν χρειαστεί να ξαναεφαρμοστεί το φίλτρο
    - “Σταματάμε” το χρονόμετρο (δηλαδή μετράμε τον χρόνο μετά το τέλος του loop) και εκτυπώνουμε την διαφορά, δηλαδή πόσο χρόνο τελικά διήρκεσε το loop σε κάθε process.
    - Με την χρήση της Gatherv με παρόμοια ορίσματα με την Scatterv όπως περιγράφτηκε παραπάνω, “ενώνουμε” τους subarrays κάθε process σε έναν τελικό μεγάλο τον final2D πίνακα ο οποίος και γράφεται στο outputFileName από το process 0.
    - Τέλος κάνουμε free ό,τι χρειάζεται.

- <u>color.c</u>
    - Το color.c λειτουργεί με τον ίδιο τρόπο με το gray.c, αλλά πλέον θεωρούμε ότι κάθε “κελί” του πίνακα έχει 3 bytes (ένα για το κάθε χρώμα) οπότε οι μόνες διαφορές είναι οι παρακάτω:
        - η συνάρτηση δέσμευσης πινάκων πλέον είναι η unsigned char ** getRGBArray(int rows, int cols) όπου δεσμεύει rows * cols * (3 * sizeof(unsigned char)), δηλαδή όπως είπαμε πλέον μπορούμε να “φανταστούμε” ότι κάθε “κελί” έχει 3 unsigned char
        - οι buffers των γωνιών είναι πλέον πίνακες μεγέθους 3 unsigned char και οι buffers των γραμμών και των στηλών δεσμεύονται ώστε να αποθηκεύουν 3 unsigned char σε κάθε “κελί” (στην πραγματικότητα δηλαδή τριπλάσιο μέγεθος αφού δεν υπάρχουν “κελιά”, απλώς το 3 σε κάθε “κελί” το χρησιμοποιούμε για λόγους ευκολότερης κατανόησης)
        - παρόμοιες αλλαγές και για τα sizes και τα subsizes όπου πλέον οι “στήλες” θα έχουν τριπλάσιο μέγεθος
        - πλέον το block extent στο create_resized θα είναι (colsNumber/sqrt_comm_sz)*(3 * sizeof(unsigned char)) και δεν χρειάζεται να αλλάξουμε κάτι στα displacements του Scatterv αφού πηγαίνουν βάση του block extent
        - στο MPI_Type_contiguous το μέγεθος θα είναι 3*(colsNumber/sqrt_comm_sz) συνεχόμενοι unsigned char
        - στο MPI_Type_vector θα αλλάξει το blocklength και πλέον θα είναι 3, και το stride γίνεται 3*(colsNumber/sqrt_comm_sz)
        - στο διάβασμα και γράψιμο του αρχείου, ο αριθμός των γραμμών παραμένει ο ίδιος με πριν απλώς οι στήλες 3πλασιάζονται

- κατάλληλες αλλαγές για τον 3πλασιασμό κάνουμε και στα Isend και Irecv όπου χρειάζεται
- στην εφαρμογή του φίλτρου οι γραμμές παραμένουν ίδιες σε αριθμό, οι στήλες τριπλασιάζονται και κάθε φορά εφαρμόζουμε το φίλτρο στο κάθε pixel, έστω στο pixel [i][j] το εφαρμόζουμε 3 φορές, μια για το j+0, μια για το j+1 και μια για το j+2, και προχωράμε στο επόμενο pixel με j = j +3
- παρόμοια λογική ακολουθούμε και όταν χρειάζεται να χρησιμοποιήσουμε κάποια γραμμή ή στήλη του γείτονα

InitStart

- Παρόμοια με την Original. Εδώ όμως επειδή οι γείτονες θα είναι οι ίδιοι σε όλες τις εφαρμογές του φίλτρου, αντί να έχουμε 8 x Isend και 8 x Irecv μέσα στο for loop, έχουμε 8 x Send_init και 8 x Recv_init πριν το for loop και 8 x Start (για τα sendRequest) και 8 x Start (για τα recieveRequest) μέσα στο for loop. Έτσι θεωρητικά γλιτώνουμε τον χρόνο που χρειάζεται κάθε φορά μέσα στο loop για την εγκαθίδρυση της επικοινωνίας μια και αυτή έχει γίνει πριν το loop.

Reduce

- Παρόμοια με την Original. Εδώ όμως ανά 15 επαναλήψεις στο for loop ελέγχουμε αν υπάρχουν αλλαγές μεταξύ του myArray και του myFinalArray σε κάθε process, αν υπάρχουν αλλαγές το process ορίζει την τιμή της μεταβλητής localChanges σε 1 ενώ αν δεν υπάρχουν σε 0. Στην συνέχεια με την χρήση της MPI_Allreduce και με operator MPI_LOR γίνεται λογικό OR μεταξύ των τιμών localChanges κάθε process και επιστρέφεται η τιμή globalChanges σε όλα τα process, που είναι 0 αν δεν υπάρχει αλλαγή σε κανένα process και 1 αν υπάρχει αλλαγή έστω και σε 1 process. Στο συγκεκριμένο πρόγραμμα δεν γίνεται κάποια ενέργεια προκειμένου να σταματήσει πιο νωρίς το for loop, αν και υπάρχει τέτοια δυνατότητα, πχ να γίνεται κάποιος break αν διαπιστωθεί οτι globalChanges == 0.

Cartesian

- Παρόμοια με την Original. Εδώ όμως δημιουργούμε έναν καινούριο communicator τον new_comm μέσω της συνάρτησης MPI_Cart_create(MPI_COMM_WORLD, 2, dim, period, reorder, &new_comm) δίνοντας τα κατάλληλα ορίσματα ώστε να ακολουθείται η καρτεσιανή τοπολογία και έτσι οι γείτονες που βρίσκονται πάνω,κάτω,αριστερά και δεξιά (όχι όμως διαγώνια) τοποθετούνται κοντά στον επεξεργαστή.

Openmp

- Παρόμοια με την Original. Εδώ όμως το for στο οποίο πραγματοποιείται η εφαρμογή του φίλτρου στα εσωτερικά σημεία παραλληλοποιείται με την χρήση της οδηγίας *#pragma omp parallel for* . Προκειμένου όμως να εκμεταλλευτούμε αυτή την δυνατότητα, θα πρέπει το αρχείο machines που πριν είχε :4 δίπλα από κάθε μηχάνημα με αποτέλεσμα να τρέχουν 4 διεργασίες σε κάθε CPU, να έχει :1 ώστε πλέον να τρέχει 1 διεργασία σε κάθε CPU, και όταν φτάσει η ώρα του for που αναφέραμε, να μπορεί να παραλληλοποιηθεί στους 4 πυρήνες του συγκεκριμένου CPU.
- Σημείωση: Ο λόγος που χρησιμοποιείται η οδηγία αυτή μόνο στο “μεγάλο” for της εφαρμογής φίλτρου στα εσωτερικά σημεία, είναι ότι αν εφαρμοστεί και για την εφαρμογή φίλτρου στα Halo points υπάρχει μια αισθητή μείωση πιθανότατα επειδή η εφαρμογή φίλτρου στα σημεία αυτά είναι αρκετά σύντομη, οπότε η προσπάθεια να παραλληλοποιηθεί το καθυστερεί περισσότερο.

Μετρήσεις

- Οι μετρήσεις βρίσκονται στο αρχείο Timings.pdf και συμπεριλαμβάνουν στατιστικά για την επεξεργασία:
  - της έγχρωμης εικόνας που αναφέρεται στην εκφώνηση
  - της ασπρόμαυρης εικόνας που αναφέρεται στην εκφώνηση
  - του μισού της έγχρωμης
  - του 1/4 της έγχρωμης
  - του μισού της ασπρόμαυρης
  - του 1/4 της ασπρόμαυρης
- Η επεξεργασία μεγαλύτερης εικόνας θα ήταν αδύνατη λόγω του περιορισμένου χώρου στους λογαριασμούς των υπολογιστών της σχολής.
- Στο παραπάνω αρχείο βρίσκονται και οι μετρήσεις για το OpenMP.
- Το κάθε πρόγραμμα χρονομετρήθηκε >= 10 φορών και χρησιμοποιείται ο μέσος όρος χωρίς τον υπολογισμό των χρόνων που αποκλίνουν από τους συνήθεις. Αν για παράδειγμα το πρόγραμμα τις περισσότερες φορές έτρεχε στα ~3.3 sec και κάποια φορά έτρεξε στα 5.8 sec τότε το 5.8 sec δεν συνυπολογίζεται.
- Οι μετρήσεις έγιναν όσο το δυνατόν πιο κοντά η μια με την άλλη για να υπάρχει συνέπεια αποτελεσμάτων και σε μη ώρα αιχμής.
- Όλες οι μετρήσεις είναι με το φίλτρο να εφαρμόζεται 100 φορές.
- Δεν έγινε μεταγλώττιση με την επιλογή του optimization -O2 ή -O3 όπου συνήθως υποδιπλασιάζουν ή υποτετραπλασιάζουν τον χρόνο, ανάλογα με το μέγεθος της εκτέλεσης.
- Όλα τα προγράμματα έχουν εκτελεστεί για 1,4,9,16,25,36 και 64 διεργασίες επειδή:
  - την ώρα που κάναμε τις μετρήσεις οι διαθέσιμοι υπολογιστές ήταν 22, πράγμα που μας επιτρέπει να έχουμε μέχρι 88 ταυτόχρονες διεργασίες, μια σε κάθε πυρήνα
  - απορρίφθηκαν οι περιπτώσεις εκτελέσεις για 7*7 = 49 και 9*9 = 81 διεργασίες, επειδή δε θα ήταν δυνατό να χωριστεί σε ίσα κομμάτια η εικόνα και θα απαιτούσε ειδική μεταχείριση που όπως είχε διευκρινιστεί δεν ήταν απαραίτητο

Παρατηρήσεις

- Παρατηρούμε ότι, στις περισσότερες περιπτώσεις, όσο αυξάνεται ο αριθμός των διεργασιών μειώνεται ο χρόνος εκτέλεσης. Γιατί συμβαίνει αυτό;
  - Αυτό συμβαίνει διότι όσο αυξάνεται ο αριθμός των διεργασιών μπορούμε να ισομοιράσουμε το πρόβλημα στις διεργασίες αυτές. Έτσι κάθε διεργασία δουλεύει σε ένα μόνο μέρος της εικόνας και αφού όλες οι διεργασίες δουλεύουν ταυτόχρονα, πέφτει ο συνολικός χρόνος εκτέλεσης.

- Παρατηρούμε ότι οι μεγαλύτερες εικόνες έχουν καλύτερο efficiency και κλιμακώνουν καλύτερα από τις μικρότερες. Γιατί συμβαίνει αυτό;
  - Ο βασικός λόγος που συμβαίνει αυτό είναι ότι για μικρό μέγεθος δεδομένων (στην περίπτωσή μας για μικρότερες εικόνες) το κόστος της επικοινωνίας μεταξύ των διεργασιών αποτελεί μεγαλύτερο μέρος του συνολικού κόστους. Αντίθετα για μεγάλα δεδομένα (μεγάλες εικόνες) το μεγαλύτερο μέρος του κόστους αποδίδεται στην εφαρμογή του φίλτρου.
  - Για αυτό όσο αυξάνεται ο αριθμός των διεργασιών για μικρή εικόνα, θα βελτιώνεται ελαφρώς ο χρόνος εφαρμογής φίλτρου αλλά θα επιβραδύνεται περισσότερο ο χρόνος

επικοινωνίας. Έτσι δε βελτιώνεται πολύ ο συνολικός χρόνος εκτέλεσης και σε ορισμένες περιπτώσεις γίνεται χειρότερος όσο αυξάνεται ο αριθμός των διεργασιών.
    - Αντίθετα για μεγάλες εικόνες το επιπρόσθετο κόστος επικοινωνίας που προκύπτει από την αύξηση των διεργασιών, είναι πολύ μικρό σε σύγκριση με την μεγάλη βελτίωση του χρόνου εφαρμογής του φίλτρου.
    - Για αυτό το λόγο όταν εφαρμόζουμε το φίλτρο σε ολόκληρη την έγχρωμη εικόνα παρατηρούμε efficiency >1 ακόμα και μέχρι 36 διεργασίες, ενώ όταν εφαρμόζουμε το φίλτρο στο 1/4 της ασπρόμαυρης εικόνας έχουμε <1 efficiency για οποιαδήποτε εκτέλεση σε περισσότερες από μια διεργασίες.

- Παρατηρούμε ότι από ένα σημείο και μετά μειώνεται το efficiency για κάθε μέγεθος εικόνας. Γιατί συμβαίνει αυτό;
    - Αυτό συμβαίνει διότι όσο αυξάνεται ο αριθμός των διεργασιών τόσο αυξάνεται και το κόστος επικοινωνίας μεταξύ αυτών. Έτσι όταν εκτελείται το πρόγραμμα σε 4 ή 9 διεργασίες, ο συνολικός χρόνος που απαιτείται για την αποστολή και λήψη μηνυμάτων από και προς κάθε διεργασία είναι σαφώς μικρότερος από όταν εκτελείται για 64 διεργασίες.

- Παρατηρούμε ότι σε ορισμένες περιπτώσεις μειώνεται ακόμα και η επιτάχυνση, δηλαδή με αύξηση του αριθμού των διεργασιών έχουμε χειρότερο χρόνο. Γιατί συμβαίνει αυτό;
    - Όπως προαναφέρθηκε και στην 2η παρατήρηση, με την αύξηση του αριθμού των διεργασιών και τον ισομοιρασμό του προβλήματος σε αυτές προκύπτουν:
        - μείωση του χρόνου που απαιτείται για την εφαρμογή του φίλτρου
        - αύξηση του χρόνου που απαιτείται για την ανταλλαγή μηνυμάτων μεταξύ των διεργασιών
    - Έτσι όταν η αύξηση του χρόνου επικοινωνίας υπερκαλύπτει τη μείωση του χρόνου εφαρμογής φίλτρου, τότε ο συνολικός χρόνος εκτέλεσης αυξάνεται με κάθε περαιτέρω αύξηση του αριθμού των διεργασιών.

- Παρατηρούμε ότι το πρόγραμμα που περιλαμβάνει MPI_Allreduce ανά συγκεκριμένο αριθμό επαναλήψεων, δεν κλιμακώνει όπως τα αλλά. Γιατί συμβαίνει αυτό;
    - Όταν δεν έχουμε χρήση MPI_Allreduce στο πρόγραμμα μας, όσο και να αυξηθεί ο αριθμός των διεργασιών, το πλήθος των διεργασιών που θα επικοινωνούν μεταξύ τους θα είναι ένας συγκεκριμένος αριθμός. Στην περίπτωση μας, όσο μεγάλος και να είναι ο αριθμός των διεργασιών πάντα κάθε διεργασία θα στέλνει μηνύματα σε 8 γείτονές της.
    - Αντίθετα όταν έχουμε χρήση MPI_Allreduce, χρειάζεται όλες οι διεργασίες να επικοινωνήσουν μεταξύ τους, τόσο για να στείλουν την δική τους τιμή όσο και για να λάβουν την τελική τιμή, με αποτέλεσμα όσο αυξάνεται ο αριθμός των διεργασιών να αυξάνεται και ο χρόνος επικοινωνίας αποστολής και συλλογής αυτών των τιμών.
    - Άλλος ένας πιθανών λόγος είναι ότι οι υλοποιήσεις των συναρτήσεων reduce, συνήθως περιλαμβάνουν crash recovery το οποίο είναι αρκετά ακριβό, με αποτέλεσμα ακόμα και αν ένα task ολοκληρωθεί μέσα σε δευτερόλεπτα, αν προκύψει κάποιο error να χρειαστεί να ξαναστείλει το μήνυμα από την αρχή.

- Παρατηρούμε ότι σε ορισμένες περιπτώσεις (ειδικά για τα Color Full Size) φαίνεται να έχουμε superlinear speedup και efficiency > 1; Γιατί συμβαίνει αυτό;
    - Ίσως ο βασικότερος λόγος που συμβαίνει αυτό είναι ο τρόπος με τον οποίο ο επεξεργαστής χειρίζεται την cache. Ένας από τους τρόπους που η cache βοηθάει το

παραλληλισμένο πρόγραμμα είναι ότι αν εκτελείται σε ένα πυρήνα ενός τετραπύρηνου επεξεργαστή τότε η μέγιστη διαθέσιμη cache θα είναι περίπου το ένα τέταρτο της συνολικής ενώ αν εκτελείται σε όλους τους τέσσερις πυρήνες του επεξεργαστή η μέγιστη διαθέσιμη cache είναι – θεωρητικά – ολόκληρη η cache του συστήματος.

- Οι εκδόσεις Cartesian και InitStart έχουν σε γενικές γραμμές παρόμοια απόδοση με την original. Γιατί συμβαίνει αυτό και γιατί επιλέξαμε να τις συμπεριλάβουμε ως ξεχωριστές εκδόσεις;
  - Συμπεριλάβαμε τις ξεχωριστές αυτές εκδόσεις επειδή είχαν προταθεί ως μέθοδοι βελτιστοποίησης τους προγράμματος αλλά δεν είχαν εμφανή και άμεση επιρροή στο χρόνο εκτέλεσης του 'original' κώδικα. Οπότε αποφασίσαμε να τις κρατήσουμε ως αυτοτελείς εκδόσεις με σκοπό να δούμε αν για διάφορα μεγέθη θα απέδιδαν καλύτερα από την αρχική.
  - Θεωρητικά, με το να χρησιμοποιήσουμε καρτεσιανή τοπολογία περιμένουμε να δούμε βελτίωση στο χρόνο εκτέλεσης εξαιτίας του γεγονότος ότι οι πυρήνες που θα ανταλλάσσουν μηνύματα κατά την εκτέλεση του προγράμματος θα βρίσκονται τοπολογικά 'κοντά' ο ένας στον άλλο και η επικοινωνία θα είναι πιο γρήγορη. Με τη χρήση InitStart, ομοίως, περιμένουμε βελτίωση στο χρόνο επειδή τα μηνύματα που στέλνει και λαμβάνει η κάθε διεργασία κάθε φορά μεταδίδονται από και προς τις ίδιες διεργασίες, κάτι που η χρήση του InitStart εκμεταλλεύεται.
  - Όπως φαίνεται όμως από τις μετρήσεις, οι τρεις αυτές εκδόσεις (Original, Cartesian και InitStart) δεν έχουν μεγάλες διαφορές. Όσο όμως αυξάνονται οι διεργασίες και ιδίως όσο μεγαλώνει σε μέγεθος η εικόνα οι Cartesian και InitStart φαίνονται να κλιμακώνουν καλύτερα. Κάτι τέτοιο φαίνεται στην εφαρμογή του φίλτρου στην έγχρωμη εικόνα όπου για 64 διεργασίες διατηρούν καλύτερη επιτάχυνση.
  - Βέβαια στην πλειοψηφία των περιπτώσεων οι τρεις αυτές εκδόσεις είναι πολύ κοντά από άποψη απόδοσης και κλιμάκωσης. Αυτό συμβαίνει επειδή για το μέγεθος δεδομένων και το πλήθος των επεξεργαστών που έχουμε στη διάθεση μας τα πλεονεκτήματα των τεχνικών Cartesian και InitStart δε μπορούν να εμφανιστούν. Πιθανότατα για μεγάλο πλήθος δεδομένων και περισσότερες διεργασίες θα αποδίδουν καλύτερα από την απλή έκδοση λόγω του ότι θα στέλνονται περισσότερα μηνύματα και κατ επέκταση θα παίζουν μεγαλύτερο ρόλο στη μείωση του συνολικού χρόνου εκτέλεσης. Γιαυτό και εμφανίζεται μία ένδειξη αυτού όταν εφαρμόζεται το φίλτρο για το Full μέγεθος της έγχρωμης εικόνας και μόνο για 64 διεργασίες.

- Είναι το πρόγραμμα επεκτάσιμο; Αν ναι, είναι ισχυρά επεκτάσιμο ή ασθενώς;
  - Για να είναι ένα πρόγραμμα ισχυρά επεκτάσιμο πρέπει με την αύξηση του αριθμού των διεργασιών στις οποίες εκτελείται να διατηρείται σταθερή η αποδοτικότητα χωρίς την αύξηση του μεγέθους του προβλήματος. Αυτό δε συμβαίνει. Σε κάποιες περιπτώσεις έχουμε αύξηση της αποδοτικότητας με την αύξηση του αριθμού διεργασιών χωρίς να αλλάζει το μέγεθος του προβλήματος, αλλά όπως αναφέρθηκε και παραπάνω αυτό οφείλεται κυρίως στον τρόπο διαχείρισης της cache από τους σύγχρονους επεξεργαστές.
  - Για να είναι ένα πρόγραμμα ασθενώς επεκτάσιμο πρέπει να μπορούμε να διατηρήσουμε σταθερή την αποδοτικότητα αυξάνοντας το μέγεθος του προβλήματος κατά το ίδιο ποσοστό με το οποίο αυξήσαμε το πλήθος των διεργασιών. Αυτό είναι κάτι που συμβαίνει στο πρόγραμμα μας και υπάρχουν πολλά παραδείγματα.
  - Να γίνει μία διευκρίνηση: Τα προγράμματα των μετρήσεων με ονόματα Color Half Size και Gray Half Size είναι στην πραγματικότητα το ένα τέταρτο (¼) του μεγέθους των Color και Gray αντίστοιχα. Τα Color και Gray είναι εικόνες μεγέθους 1920x2520 ενώ τα Half Size είναι 960x1260. Δηλαδή η κάθε διάσταση διαιρείται δια του 2 άρα το συνολικό μέγεθος διαιρείται δια του 4. Ομοίως, τα Quarter Size (480x640) είναι το ένα τέταρτο (¼) των Half Size και το ένα δέκατο έκτο (1/16) των αρχικών Full Size. Άρα τα

Color και Gray είναι τετραπλάσια σε μέγεθος των αντίστοιχων Half Size και δέκα εξαπλάσια των αντίστοιχων Quarter Size. Τα Half Size είναι με τη σειρά τους τετραπλάσια των Quarter Size.

- Οπότε μπορούμε για παράδειγμα να συγκρίνουμε το efficiency του Color Half Size για 4 διεργασίες με εκείνη του Color Full Size για 16, η οποία θα έχει τετραπλάσιο αριθμό διεργασιών και τετραπλάσιο μέγεθος δεδομένων. Όντως η αποδοτικότητα του είναι 1,24 και 1,28 αντίστοιχα, κάτι που υποδεικνύει ότι το πρόγραμμα είναι ασθενώς επεκτάσιμο. Περισσότερα παραδείγματα ακολουθούν παρακάτω:

  - Color Half Size με 9 διεργασίες : 1.03
  - Color Full Size με 36 διεργασίες : 1.07

  - Color Half Size με 16 διεργασίες : 0.70
  - Color Full Size με 64 διεργασίες : 0.68

  - Color Quarter Size με 4 διεργασίες : 0.69
  - Color Half Size με 16 διεργασίες : 0.70

  - Color Quarter Size με 9 διεργασίες : 0.40
  - Color Half Size με 36 διεργασίες : 0.42

  - Color Quarter Size με 16 διεργασίες : 0.22
  - Color Half Size με 64 διεργασίες : 0.20

  - Gray Half Size με 4 διεργασίες : 1.06
  - Gray Full Size με 16 διεργασίες : 0.92

  - Gray Half Size με 9 διεργασίες : 0.57
  - Gray Full Size με 36 διεργασίες : 0.54

  - Gray Half Size με 16 διεργασίες : 0.30
  - Gray Full Size με 64 διεργασίες : 0.28

  - Gray Quarter Size με 4 διεργασίες : 0.43
  - Gray Half Size με 16 διεργασίες : 0.30

  - Gray Quarter Size με 9 διεργασίες : 0.13
  - Gray Half Size με 36 διεργασίες : 0.12

  - Gray Quarter Size με 16 διεργασίες : 0.07
  - Gray Half Size με 64 διεργασίες : 0.06

Συμπεράσματα

1. Σε γενικές γραμμές όταν ένα παραλληλισμένο πρόγραμμα τρέχει σε περισσότερους πυρήνες, περιμένουμε να τρέξει σε λιγότερο χρόνο.
2. Τα προβλήματα με μεγάλο μέγεθος δεδομένων κλιμακώνουν καλύτερα από τα αντίστοιχα προβλήματα με μικρότερο μέγεθος.

3. Η αποδοτικότητα του συγκεκριμένου παράλληλου προγράμματος πέφτει αναπόφευκτα για μεγάλο αριθμό διεργασιών λόγω του αυξανόμενου κόστους επικοινωνίας. Αυτό εμφανίζεται πιο νωρίς σε προγράμματα με μικρό μέγεθος δεδομένων.
4. Ακόμα και η ταχύτητα και το speedup ενός προγράμματος μπορεί να μειωθεί με την αύξηση του αριθμού διεργασιών στις οποίες εκτελείται, αν το επιπλέον κόστος της επικοινωνίας υπερκαλύψει τη μείωση κόστους λόγω παραλληλισμού. Αυτό εμφανίζεται πάλι πιο νωρίς σε προγράμματα με μικρό μέγεθος δεδομένων.
5. Όταν ένα πρόγραμμα σχεδιάζεται με σκοπό τον παραλληλισμό του σε MPI δεν θα προσφέρει ιδιαίτερη αύξηση της απόδοσης η χρήση OpenMP, τουλάχιστον όχι χωρίς περαιτέρω μελέτη.
6. Η χρήση MPI_Allreduce όχι μόνο μειώνει την ταχύτητα του προγράμματος αλλά επιδεινώνει και την κλιμάκωση του σε περισσότερες διεργασίες.
7. Λόγω του σχεδιασμού των σύγχρονων επεξεργαστών είναι εφικτό να έχουν superlinear επιτάχυνση, ειδικά για εκτέλεση σε 4 ή 9 διεργασίες.
8. Οι εκδόσεις Cartesian και InitStart μπορούν να βελτιώσουν την συνολική απόδοση του προγράμματος αλλά η διαφορά είναι αισθητή μόνο για μεγάλο μέγεθος δεδομένων και διεργασιών.
9. Το πρόγραμμα αυτό είναι ασθενώς επεκτάσιμο.