Πανεπιστήμιο Ιωαννίνων

Τμήμα Μηχανικών Ηλεκτρονικών
Υπολογιστών & Πληροφορικής

Παράλληλα Συστήματα & Προγραμματισμός

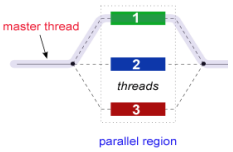
Εαρινό Εξάμηνο 2024

OpenMP

Χρήστος Δημητρέσης 4351 cs04351@uoi.gr

Περιεχόμενα

Περιβάλλον Μετρήσεων..... 3
Παραλληλοποίηση Εύρεσης Πρώτων Αριθμών.....4
Δεδομένα - Ζητούμενα.....4
Παραλληλοποίηση Loop.....4
Χρονομέτρηση.....5
Συμπέρασμα.....7
Θόλωση Εικόνων..... 8
Δεδομένα - Ζητούμενα.....8
Παραλληλοποίηση Loop - Υλοποίηση..... 8
Παραλληλοποίηση με χρήση Tasks.....9
Χρονομέτρηση..... 11
Συμπέρασμα.....13
Task Dependencies.....15
Επεξήγηση Κώδικα..... 17
Παράδειγμα Εκτέλεσης.....18
Ορθή εκτέλεση με χρήση της οδηγίας depend(xxx : buffer)..... 18
Λανθασμένη εκτέλεση χωρίς την χρήση της οδηγίας depend(xxx : buffer)..... 19



Περιβάλλον Μετρήσεων

Όλες οι πειραματικές μετρήσεις χρονομέτρησης - απόδοσης των υλοποιήσεων σε υπολογιστή του τμήματος με τα εξής χαρακτηριστικά.

Όνομα Υπολογιστή	opti7020ws11
Μοντέλο Επεξεργαστή	Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz
Πλήθος Πυρήνων	4
Μεταφραστής	gcc 11.2.0

Παραλληλοποίηση Εύρεσης Πρώτων Αριθμών

Δεδομένα - Ζητούμενα

Δίνεται ένα σειριακό πρόγραμμα στο οποίο, δεδομένου του N, η συνάρτηση serial_primes() υπολογίζει το πλήθος των πρώτων αριθμών καθώς και τον μεγαλύτερο πρώτο αριθμό μέχρι και το N.

Ζητείται να συμπληρωθει η συνάρτηση openmp_primes() ώστε να κάνει τους ίδιους υπολογισμούς παράλληλα, χρησιμοποιώντας το OpenMP.

Δεν επιτρέπεται αλλαγή στον υλοποιημένο αλγόριθμο.

Παραλληλοποίηση Loop

Προκειμένου να παραλληλοποιήσουμε το δοθέν σειριακό πρόγραμμα χρησιμοποιήθηκαν οδηγίες που χρησιμοποιούν το API του OpenMP.

Συγκεκριμένα με την οδηγία :
#pragma omp parallel for private(num, divisor, quotient , remainder) reduction(+ : count) reduction(max : lastprime) schedule(runtime)
παραλληλοποιούμε τον πρώτο βρόγχο που συναντάμε στο σειριακό μας πρόγραμμα.

Οι μεταβλητες ***num, divisor, quotient, remainder*** πρέπει να είναι ιδιωτικές για το κάθε νήμα που θα δημιουργηθει διότι σύμφωνα με τον αλγόριθμο του κώδικα , αλλάζουν- ενημερώνονται σε κάθε επανάληψη του βρόχου. Αρά φροντίζουμε να συμβεί αυτό ρήττα με την οδηγία private.

Όλα τα νήματα θα πρέπει να ενημερώσουν τις **αρχικά** κοινόχρηστες μεταβλητές ***count*** και ***lastprime*** άρα δημιουργούνται προβλήματα τύπου race conditions. Για την επίλυση αυτού του προβλήματος χρησιμοποιούμε τις οδηγίες ***reduction(max : lastprime)*** και ***reduction(+ : count)*** Με την ***reduction(max : lastprime)*** δημιουργείται μια ξεχωριστή τοπική μεταβλητή για το κάθε νήμα και στο τέλος της εκτέλεσης όλων των νημάτων επιλέγεται ως τιμή της μεταβλητής ***lastprime*** η μεγαλύτερη αριθμητικά τιμή από αυτές.

Με την οδηγία ***reduction(+ : count)*** δημιουργείται μια ξεχωριστή τοπική μεταβλητη count στο κάθε νήμα και στο τέλος της εκτέλεσης όλων των νημάτων επιλέγεται ως τιμή της μεταβλητής ***count*** το αριθμητικό άθροισμα των όλων των τοπικών μεταβλητών αυτών .

Δοκιμάστηκε και η υλοποίηση να δημιουργηθεί κρίσιμη περιοχή ***#pragma omp critical*** και κλειδαριά στο σημείο που ενημερώνεται η τιμή της μεταβλητής ***count*** στον κώδικα , όμως δεν προσέφερε κάποια αισθητή χρονική βελτίωση σε σχέση με το ***reduction*** .

Η οδηγία ***schedule(runtime)*** μας επιτρέπει να καθορίζουμε μέσω της μεταβλητής περιβάλλοντος του τερματικού την πολιτική διαμοιράσης των νημάτων κάνοντας πιο ευέλικτη την διαδικασία χρονομέτρησης

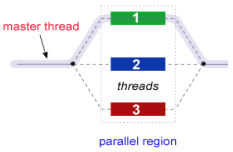
Χρονομέτρηση

Τα προγράμματα εκτελέστηκαν σε υπολογιστή με τα τεχνικά χαρακτηριστικά που αναφέρθηκαν στην εισαγωγή και για την χρονομέτρηση χρησιμοποιήθηκε η συνάρτηση ***gettimeofday()*** απο την βιβλιοθήκη ***<sys/time.h>***

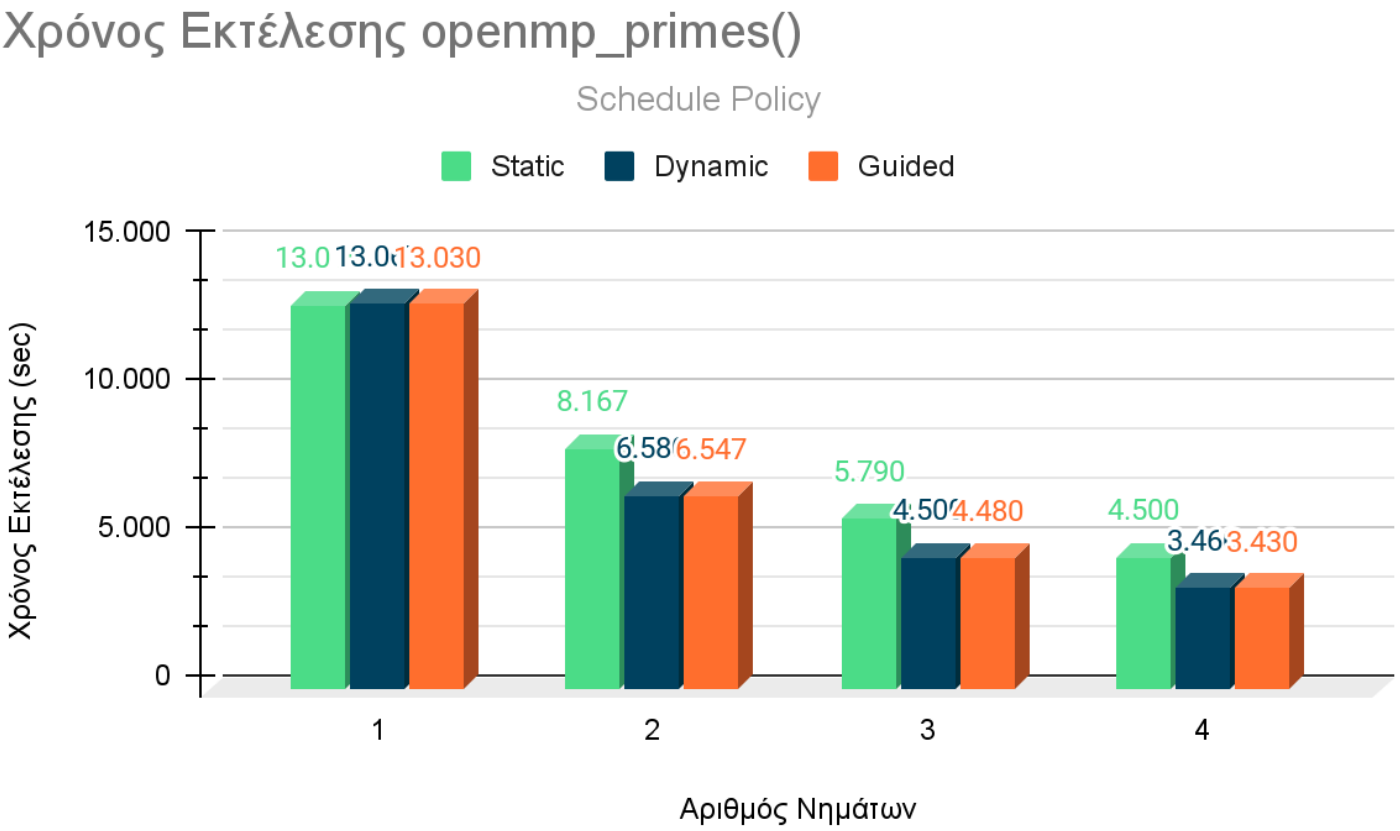
Τονίζεται ότι σε κάθε πολιτική διαμοιράσης δεν έχει καθοριστεί ρητά το chunk και άρα χρησιμοποιείται το προκαθορισμένο από το σύστημα.

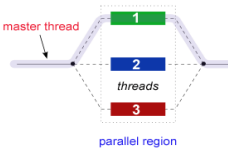
Static Schedule					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	13.01 sec	13.02 sec	13.02 sec	13.01 sec	13.015 sec
2	8.17 sec	8.16 sec	8.17 sec	8.17 sec	8.167 sec
3	5.79 sec	5.79 sec	5.79 sec	5.79 sec	5.790 sec
4	4.50 sec	4.50 sec	4.50 sec	4.50 sec	4.500 sec

Dynamic Schedule					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	13.07 sec	13.07 sec	13.07 sec	13.06 sec	13.067 sec
2	6.58 sec	6.58 sec	6.58 sec	6.58 sec	6.580 sec
3	4.50 sec	4.50 sec	4.50 sec	4.50 sec	4.500 sec
4	3.46 sec	3.46 sec	3.46 sec	3.46 sec	3.460 sec



Guided Schedule					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	13.03 sec	13.03 sec	13.03 sec	13.03 sec	13.030 sec
2	6.55 sec	6.55 sec	6.54 sec	6.55 sec	6.547 sec
3	4.48 sec	4.48 sec	4.48 sec	4.48 sec	4.480 sec
4	3.43 sec	3.43 sec	3.43 sec	3.43 sec	3.430 sec





Συμπέρασμα

Μελετώντας το παραπάνω γράφημα το οποίο συσχετίζει των αριθμό των νημάτων σε σχέση με την πολιτική διαμοίρασης αλλά και τον συνολικό χρόνο εκτέλεσης προκύπτουν τα εξής συμπεράσματα.

Όσον αφορά το ποσοστό αλλά και την ποιότητα της παραχθείσας παραλληλοποίησης του κώδικα μπορούμε να ισχυριστούμε ότι έχουμε φτάσει σε ικανοποιητικό επίπεδο δεδομένου ότι σε όλους τους πίνακες ισχύει η σχέση -> Χρόνος_με_N_Νήματα ≈ Χρόνος_Σειριακά/N

Άρα η δουλεία έχει μοιραστεί ισόποσα σε όλα τα νηματα και έχουμε φτάσει την μέγιστη δυνατή παραλληλοποίηση.

—

Όσον αφορά την πολιτική διαμοίρασης είναι εμφανές ότι η **static** διαμοίραση είναι η πιο χρονοβόρα ενώ η **dynamic** και η **guided** είναι ισοδύναμες και αισθητα πιο γρήγορες. Αυτό συμβαίνει λόγω της πολιτικής μη αυτοδρομολόγησης στην **static**.

Στην πολιτική **static** και μη ορίζοντας chunk_size , ορίζεται το προκαθορισμένο το οποίο είναι να χωριστούν οι επαναλήψεις του βρόχου σε ομάδες με ίσο αριθμό επαναλήψεων και να δοθούν στους πυρήνες. Όμως ενώ ό αριθμός των επαναλήψεων που θα δοθεί στον κάθε πυρήνα θα είναι ίδιος, ο φόρτος δεν θα είναι. Διότι όσο αυξάνεται το i στον βρόγχο τόσο πιο πολύ χρόνο εκτέλεσης θα χρειάζεται το while που περιέχει η κάθε επανάληψη (σύμφωνα με τον αλγόριθμο που υλοποιεί ο κώδικας). Αρά ο φόρτος δεν μοιράζεται ισόποσα στα νήματα και με αποτέλεσμα αυτή η μικρή καθυστέρηση.

Με την χρήση της πολιτικής **dynamic** μη έχοντας ορίσει **chunk_size** χρησιμοποιείται το προκαθορισμένο , δηλαδή ανάθεση μιας επανάληψης στο κάθε νήμα και άρα αν καποιο νήμα είναι πιο χρονοβόρο δεν θα εμποδίσει την εκτέλεση των αλλων διότι δεν υπάρχει προκαθορισμένη σειρά εκτέλεσης των νημάτων αλλά ανταγωνισμός μεταξύ τους.

Η πολιτική **guided** στις χρονομετρήσεις μας προκύπτει ισοδύναμη με την **dynamic** λόγω του μικρού αριθμού πυρήνων - νηματων που χρησιμοποιηθηκαν στην χρονομέτρηση. Τέσσερα (4) νήματα δεν είναι αρκετά μεγάλος αριθμός προκειμένου να δημιουργηθούν συμφόρηση και προβλήματα ανταγωνισμού μεταξύ των νημάτων. Μόνο τότε θα υπήρχε κάποιο αισθητό όφελος με τον διαφορετικό τρόπο ανάθεσης των νημάτων που χρησιμοποιεί η πολιτική **guided** σε σχέση με την **dynamic**.

Θόλωση Εικόνων

Δεδομένα - Ζητούμενα

Δίνεται ένα σειριακό πρόγραμμα το οποίο εφαρμόζει Gaussian blur προκειμένου να θολώσει (ή να ομαλοποιήσει) μία εικόνα.

Η συνάρτηση που κάνει τη θόλωση είναι η gaussian_blur_serial(), η οποία παίρνει μία εικόνα imgin και παράγει τη θολωμένη της εκδοχή imgout, βάσει μιας ακτίνας θόλωσης radius (όσο μεγαλύτερη η ακτίνα,τόσο πιο έντονο το θόλωμα).

Ζητείται να γίνεται η θόλωση παράλληλα χρησιμοποιώντας το OpenMP με παραλληλοποίηση των loops αλλά και με την χρήση tasks.

Παραλληλοποίηση Loop - Υλοποίηση

Προκειμένου να παραλληλοποιήσουμε το δοθέν σειριακό πρόγραμμα χρησιμοποιήθηκαν οδηγίες που χρησιμοποιούν το API του OpenMP.

Η συνάρτηση
void gaussian_blur_omp_loops(int radius, img_t *imgin, img_t *imgout)
χρησιμοποιεί ως βάση την σειριακή υλοποίηση της συνάρτησης
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
προκειμένου να παραλλαληλοποίησουμε τους βρόγχους του κώδικα.

Συγκεκριμένα στην συνάρτηση
void gaussian_blur_omp_loops(int radius, img_t *imgin, img_t *imgout)
με την οδηγία
#pragma omp parallel for private(i,j,row,col,redSum, greenSum, blueSum, weightSum) collapse(2) schedule(runtime)
παραλληλοποιούμε τον πρώτο βρόγχο που συναντάμε στον κώδικα μας.

Η οδηγία ***#colapse(2)*** ενωνει τα 2 πρώτα for-loops σε ένα, γεγονός που μπορεί να γίνει επειδή οι δύο πρώτοι βρόγχοι είναι ανεξάρτητοι μεταξύ τους , έχουν κανονική μορφή και δεν συμπεριλαμβάνεται καποιος κώδικας αναμεσα τους. Η οδηγία αυτή αυξάνει την παραλληλοποίηση του προγράμματος μας εξαλείφοντας ουσιαστικά τον 1 απο τους 4 εσωτερικούς βρόγχους.

Η οδηγία ***schedule(runtime)*** μας επιτρέπει να καθορίζουμε μέσω της μεταβλητής περιβάλλοντος του τερματικού την πολιτική διαμοιράσης των νημάτων κάνοντας πιο ευέλικτη την διαδικασία χρονομέτρησης.

Η οδηγία ***private(i,j,row,col,redSum, greenSum, blueSum, weightSum)*** ορίζει τις συγκεκριμένες μεταβλητές τοπικές για το κάθε νήμα , γεγονός αναγκαίο διότι σε κάθε επανάληψη των βροχών του κώδικα οι μεταβλητες αυτές διαβάζονται - τροποποιούνται.

Η οδηγία ***shared(radius,imgin,imgout,width,height)*** ορίζει τις συγκεκριμένες μεταβλητές ώς κοινες. Οι μεταβλητές ***radius,imgin,width,height*** δεν τροποποιούνται κατά την εκτέλεση του κώδικα των βρόγχων παρά μόνο διαβάζονται έχοντας όμως σταθερη τιμή, αρά δεν δημιουργούνται προβλήματα race-conditions και δεν μας δημιουργείται πρόβλημα να είναι κοινές μεταξύ των νημάτων.

Η μεταβλητή ***imgout*** ενημερώνεται κατά την διάρκεια του εκτέλεσης των βρόγχων , όμως σύμφωνα με τον αλγόριθμο που υλοποιείται είναι εξασφαλισμένο ότι το κάθε νήμα θα γράψει σε διαφορετικό σημείο του πίνακα που δείχνει ο δείκτης **imgout**. Αρά δεδομένου αυτου, δεν υπάρχει πρόβλημα race-condition και είναι εφικτό να είναι και αυτή η μεταβλητή κοινή μεταξύ των νημάτων.

Παραλληλοποίηση με χρήση Tasks

Προκειμένου να παραλληλοποιήσουμε το δοθέν σειριακό πρόγραμμα χρησιμοποιήθηκαν οδηγίες που χρησιμοποιούν το API του OpenMP.

Η συνάρτηση
void gaussian_blur_omp_tasks(int radius, img_t *imgin, img_t *imgout)
χρησιμοποιεί ως βάση την σειριακή υλοποίηση της συνάρτησης
void gaussian_blur_serial(int radius, img_t *imgin, img_t *imgout)
προκειμένου να παραλλαληλοποίησουμε τον κώδικα με την χρήση tasks.

Βρισκόμαστε ακριβώς πριν τον πρώτο από τους 4εις βρόγχους του κώδικα.

Η οδηγία ***#pragma omp parallel*** δημιουργεί μια παράλληλη περιοχή και νήματα. Μέσα σε αυτή την παράλληλη περιοχή η οδηγία ***#pragma omp single*** δημιουργεί μια περιοχή στην οποία ο κώδικας που περιέχει θα εκτελεστεί μόνο από ένα νήμα από αυτά που δημιουργήθηκαν στην παράλληλη περιοχή. Στην περίπτωση μας η παραλληλη περιοχή και η περιοχή single ταυτίζονται αλλά θα μπορούσε να ήταν και διαφορετικά.

Η οδηγία
#pragma omp task shared(imgin,imgout,radius,width,height) private(row,col) firstprivate(i,j,weightSum,redSum,greenSum,blueSum)

δημιουργεί τα Tasks. Όλα αυτά τα Tasks που θα δημιουργηθούν, μετά το πέρας της περιοχής single και λόγω του barrier που υπάρχει στο τέλος αυτης , θα αρχίσουν να καταναλώνονται και να εκτελούνται παράλληλα από όλα τα νήματα που δημιουργήθικαν οταν δημιουργήσαμε την παράλληλη περιοχή στην αρχή.

Όσον αφορά την οδηγία που δημιουργεί τα Tasks :

Η οδηγία ***shared(imgin,imgout,radius,width,height)*** ορίζει τις συγκεκριμένες μεταβλητές κοινές μεταξύ όλων των Tasks.

Οι μεταβλητές ***radius,imgin,width,height*** δεν τροποποιούνται κατά την διάρκεια εκτέλεσης του εκάστοτε Task παρα μόνο διαβάζονται έχοντας όμως σταθερη τιμή, αρά δεν δημιουργούνται προβλήματα race-conditions και δεν δημιουργείται πρόβλημα να είναι κοινές μεταξύ των Tasks.

Η μεταβλητή ***imgout*** ενημερώνεται κατά την διάρκεια εκτέλεσης του κώδικα του εκάστοτε Task όμως σύμφωνα με τον αλγόριθμο που υλοποιείται είναι εξασφαλισμένο ότι το κάθε Task θα γράψει σε διαφορετικό σημείο του πίνακα που δείχνει ο δείκτης ***imgout.*** Αρά δεδομένο αυτου δεν υπάρχει πρόβλημα race-condition και είναι εφικτό να είναι και αυτή η μεταβλητή κοινή μεταξύ των Tasks.

Η οδηγία ***private(row,col)*** ορίζει τις συγκεκριμένες μεταβλητές και ιδιωτικές για το εκάστοτε Task. Σύμφωνα με τον κώδικα αυτές οι μεταβλητες αρχικοποιουνται μέσα στο εκάστοτε Task.

Η οδηγία ***firstprivate(i,j,weightSum,redSum,greenSum,blueSum)*** ορίζει τις συγκεκριμένες μεταβλητές ιδιωτικές για το εκάστοτε Task. Η διαφορα με την οδηγία private έγκειται στο γεγονος ότι αυτές οι μεταβλητες αρχικοποιουνται την στιγμή δημιουργίας του Task με τις τιμές που είχαν εκείνη την στιγμή “εξωτερικα “, πράγμα που είναι και το επιθυμητο διότι χρησιμοποιούνται και σε κώδικα εκτός του Task άρα μεταβάλλονται στο χρόνο. Δεν γνωρίζουμε πότε θα εκτελεστεί το εκάστοτε Task άρα έτσι εξασφαλίζουμε ότι όταν εκτέλέστει οι παράμετροι του θα έχουν τις τιμές που είχαν κατά την στιγμή της δημιουργίας του.

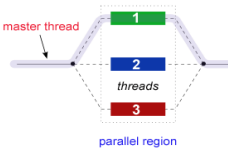
Χρονομέτρηση

Τα προγράμματα εκτελέστηκαν σε υπολογιστή με τα τεχνικά χαρακτηριστικά που αναφέρθηκαν στην εισαγωγή και για την χρονομέτρηση χρησιμοποιήθηκε η συνάρτηση ***gettimeofday()*** απο την βιβλιοθήκη ***<sys/time.h>***

Τονίζεται ότι σε κάθε πολιτική διαμοιράσης δεν έχει καθοριστεί ρητά το chunk και άρα χρησιμοποιείται το προκαθορισμένο από το σύστημα.

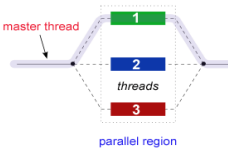
Static Schedule - Parallel Loop					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	18.36 sec	18.31 sec	18.31 sec	18.30 sec	18.320 sec
2	9.21 sec	9.21 sec	9.22 sec	9.21 sec	9.212 sec
3	6.30 sec	6.30 sec	6.30 sec	6.30 sec	6.300 sec
4	4.90 sec	4.86 sec	4.86 sec	4.86 sec	4.870 sec

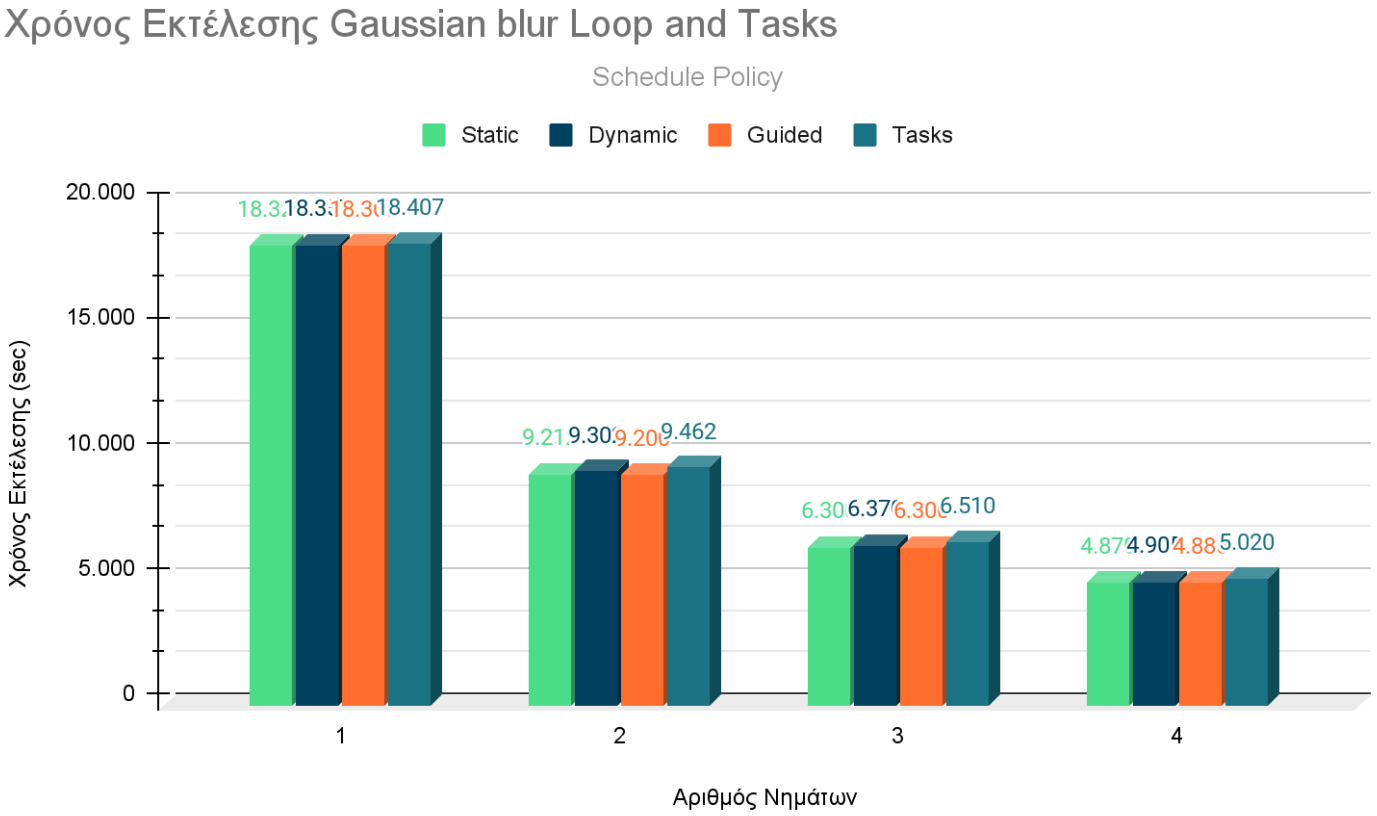
Dynamic Schedule - Parallel Loop					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	18.36 sec	18.35 sec	18.35 sec	18.36 sec	18.357 sec
2	9.31 sec	9.29 sec	9.30 sec	9.31 sec	9.302 sec
3	6.37 sec	6.37 sec	6.37 sec	6.37 sec	6.370 sec
4	4.90 sec	4.91 sec	4.90 sec	4.91 sec	4.905 sec



Guided Schedule - Parallel Loop					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	18.30 sec	18.31 sec	18.31 sec	18.31 sec	18.307 sec
2	9.17 sec	9.21 sec	9.21 sec	9.21 sec	9.200 sec
3	6.30 sec	6.30 sec	6.30 sec	6.30 sec	6.300 sec
4	4.86 sec	4.92 sec	4.88 sec	4.88 sec	4.885 sec

Parallel Tasks					
Νήματα	1η εκτέλεση	2η εκτέλεση	3η εκτέλεση	4η εκτέλεση	Μέσος όρος
1	18.40 sec	18.42 sec	18.41 sec	18.40 sec	18.407 sec
2	9.47 sec	9.46 sec	9.46 sec	9.46 sec	9.462 sec
3	6.51 sec	6.51 sec	6.52 sec	6.52 sec	6.510 sec
4	5.02 sec	5.02 sec	5.02 sec	5.02 sec	5.020 sec





Συμπέρασμα

Μελετώντας το παραπάνω γράφημα το οποίο συσχετίζει των αριθμό των νημάτων σε σχέση με την πολιτική διαμοίρανσης αλλά και τον συνολικό χρόνο εκτέλεσης , προκύπτουν τα εξής συμπεράσματα.

Όσον αφορά το ποσοστό αλλά και την ποιότητα της παραχθείσας παραλληλοποίησης του κώδικα μπορούμε να ισχυριστούμε ότι έχουμε φτάσει σε ικανοποιητικό επίπεδο δεδομένου ότι σε όλους τους πίνακες ισχύει η σχέση -> Χρόνος_με_N_Νήματα ≈ Χρόνος_Σειριακά/N

Άρα η δουλεία έχει μοιραστεί ισόποσα σε όλα τα νηματα και έχουμε φτάσει την μέγιστη δυνατή παραλληλοποίηση.

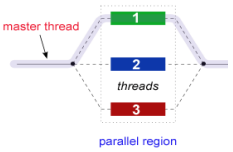
Όσον αφορά την πολιτική διαμοίρανσης είναι εμφανές ότι όλες οι πολιτικές είναι ισοδύναμες χρονικά και δεν υπάρχει κάποιο όφελος για την χρήση κάποιας σε σχέση με κάποια άλλη.

Η χρήση των Tasks θα μπορούσε να θεωρηθεί αφαιρετικά ως μια πολιτική διαμοιράνσης παρόμοια με την χρήση της dynamic με chunk = 1 δεδομένου ότι ο κώδικας μας δεν είναι ακανόνιστος και ουσιαστικά χρησιμοποιεί πάλι τους βρόγχους.

Ολες λοιπόν οι παραπάνω πολιτικές διαμοίρανσης στην περίπτωση της παραλληλοποίησης του βρόχου καταλήγουν ισοδύναμες λόγω του γεγονότος οτι ο φόρτος σε κάθε επανάληψη του (εξωτερικού) βρόγχου είναι συνολικά ισοδύναμος και επίσης οι επαναλήψεις είναι ανεξάρτητες.

Στην περίπτωση των Task ισχύει ακριβώς το ίδιο. Κάθε Task έχει ισοδύναμο φόρτο και είναι ανεξάρτητο το ένα με το άλλο.

Δεδομένου επίσης του σχετικά μεγάλου αριθμού εμφωλευμένων βρόγχων (4) που έχουμε στην εκδοχή της παραλληλοποίησης του βρόχου , αντισταθμίζεται η χρονική επιβάρυνση για την δημιουργία των Tasks στην εκδοχή με τα Tasks και εν τέλει όλοι οι προαναφερθέντες τρόποι εκτέλεσης του κώδικα αλλά και διαμοιράνσης του φόρτου , καταλήγουν ισοδύναμοι.



Task Dependencies

Τα Task Dependencies είναι ουσιαστικά μια δυνατότητα συγχρονισμού των Task η οποία είναι διαθέσιμη από την έκδοση OpenMP 4.0 και έπειτα. Με την χρήση των οδηγιών αυτών μπορούμε να συσχετίσουμε Task μεταξύ τους αυξάνοντας τον παραλληλισμό της υλοποίησης μας διατηρώντας όμως μια μορφή διοχέτευσης προκειμένου να μην χαθεί η συσχέτιση των δεδομένων των Task.

Προκειμένου να γίνει πιο κατανοητη η λειτουργία τους θα υλοποιήσουμε μια τετρημενη περίπτωση ανάγκης συγχρονισμου χρησιμοποιώντας Task Dependencies. Αυτή η περίπτωση είναι το γνωστό σε όλους μας πρόβλημα συγχρονισμού "παραγωγού - καταναλωτή"

Στην υλοποίηση μας διαβάζουμε ένα αρχείο κειμένου txt και θελουμε καθως το διαβάζουμε να τυπώνουμε το περιεχόμενο του στο τερματικό. Θα χρησιμοποιήσουμε Tasks προκειμένου να παραλληλοποιήσουμε την εν λόγω διαδικασία και Task Dependencies για να συγχρονίσουμε τα Tasks και το αποτέλεσμα της εκτύπωσης να είναι με την σωστή σειρά. Δηλαδή τα περιεχόμενα του αρχείου να τυπώνονται με την σειρα που διαβάστηκαν και όχι τυχαία.

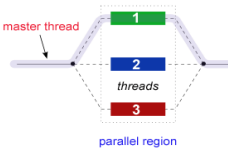
Παρατίθεται ο κωδικας που υλοποιεί το παράδειγμα μας αλλά και το αρχείο εισόδου :

```
1 non-host device has a unique device number that is greater than or equal to zero and less than the
2 device number for the host device. Additionally, the constant omp_initial_device can be
3 used as an alias for the host device and the constant omp_invalid_device can be used to
4 specify an invalid device number. A conforming device number is either a non-negative integer that
5 is less than or equal to omp_get_num_devices() or equal to omp_initial_device or
6 omp_invalid_device.
7 When a target construct is encountered, a new target task is generated. The target task region
8 encloses the target region. The target task is complete after the execution of the target region
9 is complete.
10 When a target task executes, the enclosed target region is executed by an initial thread. The
11 initial thread executes sequentially, as if the target region is part of an initial task region that is
12 generated by an implicit parallel region. The initial thread may execute on the requested target
13 device, if it is available and supported. If the target device does not exist or the implementation
14 does not support it, all target regions associated with that device execute on the host device.
15 The implementation must ensure that the target region executes as if it were executed in the data
16 environment of the target device unless an if clause is present and the if clause expression
17 evaluates to false.
18 The teams construct creates a league of teams, where each team is an initial team that comprises
19 an initial thread that executes the teams region. Each initial thread executes sequentially, as if the
20 code encountered is part of an initial task region that is generated by an implicit parallel region
21 associated with each team. Whether the initial threads concurrently execute the teams region is
22 unspecified, and a program that relies on their concurrent execution for the purposes of
23 synchronization may deadlock.
24 If a construct creates a data environment, the data environment is created at the time the construct is
25 encountered. The description of a construct defines whether it creates a data environment.
26 When any thread encounters a parallel construct, the thread creates a team of itself and zero or
27 more additional threads and becomes the primary thread of the new team. A set of implicit tasks,
28 one per thread, is generated. The code for each task is defined by the code inside the parallel
29 construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is
30 always executed by the thread to which it is initially assigned. The task region of the task being
31 executed by the encountering thread is suspended, and each member of the new team executes its
32 implicit task. An implicit barrier occurs at the end of the parallel region. Only the primary
33 thread resumes execution beyond the end of the parallel construct, resuming the task region
34 that was suspended upon encountering the parallel construct. Any number of parallel
35 constructs can be specified in a single program.
36 parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or
37 is not supported by the OpenMP implementation, then the new team that is created by a thread that
38 encounters a parallel construct inside a parallel region will consist only of the
39 encountering thread. However, if nested parallelism is supported and enabled, then the new team
```

Αρχείο εισόδου :
openMP_execution_model.txt

⇐

*** Στην αρχή κάθε γραμμής υπάρχει για ευκολία ο αριθμός που της αντιστοιχεί προκειμένου η εκτύπωση στην κονσόλα να είναι εμφανές αν έγινε διατηρόντας την ορθή σειρά του αρχείου.



Κώδικας Προγράμματος : task_dependencies.c ⤵

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <omp.h>

#define BUFFER_SIZE  50

char buffer[BUFFER_SIZE];
int fp;
long file_size;

void reader(size_t nbytes){
   read(fp,buffer,nbytes);
}

void writer(size_t nbytes){
   write(STDOUT_FILENO, buffer, nbytes);
}

int main(void) {

   fp = open("openMP_execution_model.txt", O_RDONLY);

   if (fp == -1) {
       perror("Opening file failed. System exit");
       return -1;
   }

   file_size = lseek(fp, 0, SEEK_END);
   lseek(fp, 0, SEEK_SET);

   int iterations_num = file_size / BUFFER_SIZE;
   size_t last_chunk_bytes = file_size % BUFFER_SIZE;

   #pragma omp parallel
   #pragma omp single
   {
       for (int i = 0; i <= iterations_num; i++) {
           #pragma omp task depend(out : buffer)
           writer(BUFFER_SIZE);

           #pragma omp task depend(in : buffer)
           reader(BUFFER_SIZE);
       }
   }

   reader(last_chunk_bytes);
   writer(last_chunk_bytes);

}
```

Επεξήγηση Κώδικα

1. Ανοιγουμε το αρχειο που αναφέραμε πριν προκειμένου να το διαβάσουμε στην συνέχεια.

2. Υπολογίζουμε το μέγεθος του σε bytes και το διαιρούμε με το μέγεθος του buffer που έχουμε προκειμένου να υπολογιστεί πόσα Tasks writer και reader είναι αναγκαίο να φτιαχτούν στην συνέχεια. Αν δεν διαιρείται ακριβώς το μέγεθος του αρχείου txt σε bytes με το μέγεθος του buffer, το υπολοιπο των bytes που απομένουν διαβάζονται και γράφονται σειριακά μετά το πέρας της παράλληλης περιοχής.

3. Δημιουργούμε μια παράλληλη περιοχή. Εδώ δημιουργούνται και τα νήματα.

4. Δημιουργούμε μια περιοχή τύπου **single** της οποίας τον κώδικα θα εκτελέσει μόνο ένα από τα νήματα που δημιουργήθηκαν προηγουμένως.

   Μέσα στην περιοχή **single** δημιουργούμε τα **Tasks** του αλγοριθμου μας. Με την χρήση του βρόχου δημιουργούμε αριθμητικά όσα Task είναι αναγκαία με βάση την λογική που εξηγήθηκε στο βήμα 2.

   Σε κάθε επανάληψη δημιουργείται ένα **Task** για ανάγνωση **buffer_size** bytes του άρχειου αλλά και ένα **Task** για εκτύπωση στο τερματικό **buffer_size** bytes απο τον buffer.

   Tasks που περιέχουν τις συναρτήσεις **writer()** και **reader()** αντίστοιχα δηλαδή.

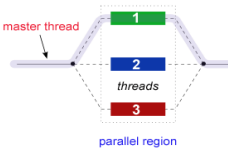
   Αυτά τα **Tasks** δεν εκτελούνται την στιγμή της δημιουργίας τους (διότι βρισκόμαστε μέσα στην περιοχή **single**) αλλά θα αρχίσουν να εκτελούνται μολις βγουμε απο την περιοχή **single** και “πέσουμε” πάνω στο **barrier** που υπάρχει στο τέλος της περιοχής αυτής. (Το οποίο τυχαίνει να ταυτίζεται και με το τέλος της παράλληλης περιοχής συνολικά).

   Άρα ενώ έχουμε δημιουργήσει επιτυχώς τον αναγκαίο αριθμό **Tasks** για ανάγνωση όλων των bytes του αρχείου txt και εκτύπωσης τους στο τερματικό αν δεν χρησιμοποιούσαμε task dependencies θα υπήρχε θέμα συγχρονισμού.

   Έτσι λοιπόν στο σημείο που δημιουργούμε τα task μετά την οδηγία ***#pragma omp task*** η οποία δημιουργεί ένα **Task** σύμφωνα με τον γνωστό τρόπο, προσθέτουμε την οδηγία ***depend(out : buffer)*** στα **Tasks** που υλοποιούν τον **writer()** και ***depend(in : buffer)*** στα **Tasks** που υλοποιούν τον **reader()** .

   Με την οδηγία ***depend(out : buffer)*** δηλώνουμε στο σύστημα ότι το **Task** που την περιέχει θα γράφει στην μεταβλητη **buffer**.

   Με την οδηγία ***depend(in : buffer)*** δηλώνουμε στο σύστημα ότι το **Task** που την περιέχει θα διαβάζει την μεταβλητή **buffer**.



Αρά με βάση αυτή την πληροφορία το σύστημα μπορεί να καταλάβει την συσχέτιση που έχουν τα Tasks μεταξύ τους αλλά και οτι μοιράζονται μια κοινή μεταβλητή.

Κάθε φορά θα συσχετίζονται δύο **Task** , ένας **reader()** και ενας **writer()** και πρώτα θα πρέπει να εκτελεστει το **Task** που γράφει (writer) και μετα το **Task** που διαβάζει (reader) την μεταβλητη **buffer**. Λόγω της επιβολής αυτής στην σειρά εκτέλεσης των Task εξαλείφεται και το προβλημα του race-condition που σε άλλη περίπτωση θα υπήρχε για την μεταβλητη **buffer**.

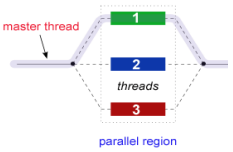
Έτσι λοιπόν δημιουργούμε μια δομή εκτέλεσης παρόμοια με διοχέτευση. Κάθε φορά που κάποιο **Task writer()** διαβάζει την μεταβλητή **buffer** μπορεί ταυτόχρονα ενα **Task reader()** να τυπώνει στο τερματικό την τιμή που είχε διαβάσει το αμέσως προηγούμενο **Task writer()**.

Άρα δημιουργείται πρακτικά παραλληλισμός της εργασίας.

Παράδειγμα Εκτέλεσης

Ορθή εκτέλεση με χρήση της οδηγίας depend(xxx : buffer)

```
xrdim@ubuntu-pc: ~/Desktop/lab_01_OpenMP
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ gcc -o task_dependencies -fopenmp task_dependencies.c
xrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$ task_dependencies
1 non-host device has a unique device number that is greater than or equal to zero and less than the
2 device number for the host device. Additionally, the constant omp_initial_device can be
3 used as an alias for the host device and the constant omp_invalid_device can be used to
4 specify an invalid device number. A conforming device number is either a non-negative integer that
5 is less than or equal to omp_get_num_devices() or equal to omp_initial_device or
6 omp_invalid_device.
7 When a target construct is encountered, a new target task is generated. The target task region
8 encloses the target region. The target task is complete after the execution of the target region
9 is complete.
10 When a target task executes, the enclosed target region is executed by an initial thread. The
11 initial thread executes sequentially, as if the target region is part of an initial task region that is
12 generated by an implicit parallel region. The initial thread may execute on the requested target
13 device, if it is available and supported. If the target device does not exist or the implementation
14 does not support it, all target regions associated with that device execute on the host device.
15 The implementation must ensure that the target region executes as if it were executed in the data
16 environment of the target device unless an if clause is present and the if clause expression
17 evaluates to false.
18 The teams construct creates a league of teams, where each team is an initial team that comprises
19 an initial thread that executes the teams region. Each initial thread executes sequentially, as if the
20 code encountered is part of an initial task region that is generated by an implicit parallel region
21 associated with each team. Whether the initial threads concurrently execute the teams region is
22 unspecified, and a program that relies on their concurrent execution for the purposes of
23 synchronization may deadlock.
24 If a construct creates a data environment, the data environment is created at the time the construct is
25 encountered. The description of a construct defines whether it creates a data environment.
26 When any thread encounters a parallel construct, the thread creates a team of itself and zero or
27 more additional threads and becomes the primary thread of the new team. A set of implicit tasks,
28 one per thread, is generated. The code for each task is defined by the code inside the parallel
29 construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is
30 always executed by the thread to which it is initially assigned. The task region of the task being
31 executed by the encountering thread is suspended, and each member of the new team executes its
32 implicit task. An implicit barrier occurs at the end of the parallel region. Only the primary
33 thread resumes execution beyond the end of the parallel construct, resuming the task region
34 that was suspended upon encountering the parallel construct. Any number of parallel
35 constructs can be specified in a single program.
36 parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or
37 is not supported by the OpenMP implementation, then the new team that is created by a thread that
38 encounters a parallel construct inside a parallel region will consist only of the
39 encountering thread. However, if nested parallelism is supported and enabled, then the new teamxrdim@ubuntu-pc:~/Desktop/lab_01_OpenMP$
```



Λανθασμένη εκτέλεση χωρίς την χρήση της οδηγίας depend(xxx : buffer)

Τυχαίο Παράδειγμα