Πανεπιστήμιο Ιωαννίνων

Πολυτεχνική Σχολή

Τμήμα Μηχανικών Η/Υ και Πληροφορικής

Προπτυχιακό Μάθημα: «Παράλληλα Συστήματα κ' Προγραμματισμός»

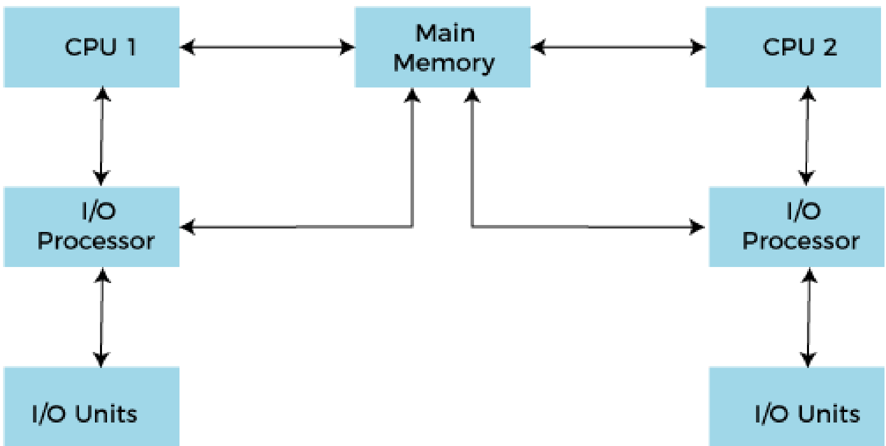
Πρώτο Σετ Προγραμματιστικών Ασκήσεων

<u>Όνομα Φοιτητή – Α.Μ.:</u>

Γεώργιος Κρομμύδας – 3260

<u>E-mail Φοιτητή:</u>

cs03260@uoi.gr



ΙΩΑΝΝΙΝΑ,

2022

Πίνακας περιεχομένων

1. Εισαγωγή:

Το πρώτο σετ προγραμματιστικών ασκήσεων αφορά στον παράλληλο προγραμματισμό με το μοντέλο κοινόχρηστου χώρου διευθύνσεων μέσω του προτύπου ***OpenMP***. Ζητείται η παραλληλοποίηση δύο εφαρμογών οι οποίες αφοράν τον υπολογισμό των πρώτων αριθμών που εμφανίζονται στο σύνολο αριθμών $\{0,..,N\}$, όπου ο N είναι ο ένας ακέραιος αριθμός. Η δεύτερη εφαρμογή αφορά το φιλτράρισμα εικόνων με χρήση του φίλτρου ***Gaussian Blur*** και δεδομένης ακτίνας r. Επιπλέον, από την έκδοση 4.5 του ***OpenMP*** υποστηρίζεται η εντολή ***taskloop*** η οποία επιτρέπει επαναλήψεις ενός βρόχου ***for*** να εκτελεστούν μέσω ***tasks*** και θα μελετηθεί με ένα απλό πρόγραμμα πολλαπλασιασμού πινάκων.

Όλες οι μετρήσεις έγιναν στο παρακάτω σύστημα:

Όνομα Υπολογιστή	opti3060ws09
Επεξεργαστής	Intel i3-8300 3.7Ghz
Πλήθος Πυρήνων	4
Μεταγλωττιστής	gcc v.7.5.0

Πίνακας 1: Λεπτομέρειες Συστήματος

2. Άσκηση-1:

2.1. Το πρόβλημα:

Σε αυτή την άσκηση ζητείται να παραλληλοποιηθεί ο αλγόριθμος υπολογισμού πρώτων αριθμών που είναι μικρότεροι από το $N = 10.000.000$ και να συγκριθούν οι χρόνοι εκτέλεσης του σειριακού προγράμματος με το παράλληλο με διαφορετικό πλήθος νημάτων. Αυτό επιτυγχάνεται παραλληλοποιώντας τον βρόχο του αλγορίθμου που υπολογίζει τους πρώτους αριθμούς. Τέλος, ζητείται να δοκιμαστούν διαφορετικές τρόποι διαμοίρασης των επαναλήψεων (schedules) και να καταλήξουμε στην καλύτερη δυνατή λύση.

2.2. Μέθοδοι Παραλληλοποίησης:

Για την παραλληλοποίηση, χρησιμοποιήθηκε το σειριακό πρόγραμμα που υπήρχε στην ιστοσελίδα του μαθήματος (primes.c). Σε αυτό το πρόγραμμα τροποποιήθηκε η συνάρτηση ***openmp_primes()***, ο οποίος θα έχει τον ίδιο κορμό με αυτό του σειριακού, δηλαδή του ***serial_primes()***. Συγκεκριμένα, το πρώτο βήμα ως προς την παραλληλοποίηση του προγράμματος είναι να προστεθεί η παρακάτω οδηγία

$$\#pragma\ omp\ parallel\ private(num, divisor, quotient, remainder)\ reduction(+: count)$$

και στη συνέχεια προστέθηκε και η οδηγία

$$\#pragma\ omp\ for\ shedule(static \mid dynamic \mid guided\ [, chunk])$$

πριν από το βρόχο του ***for*** του ***i***. Αρχικά, οι μεταβλητές *num, divisor, quotient* και *remainder* πρέπει να είναι ιδιωτικές, έτσι ώστε κάθε νήμα να κάνει τους υπολογισμούς στο δικό του χώρο χωρίς να επηρεάζει τα υπόλοιπα νήματα. Επίσης, η μεταβλητή ***i*** γίνεται και αυτή αυτόματα ιδιωτική χωρίς την δήλωση της στην έναρξη της παράλληλης περιοχής, καθώς την δηλώνει το ***#pragma omp for***. Ενώ η μεταβλητή count που μετρά το σύνολο των πρώτων αριθμών, είναι διαφορετικό για κάθε νήμα. Οπότε, θα πρέπει να προστεθούν τα επιμέρους count των νημάτων έτσι ώστε να πάρουμε το συνολικό πλήθος και να είναι ίδιο με αυτό του σειριακού.

2.3. Πειραματικά Αποτελέσματα – Μετρήσεις:

Το πρόγραμμα εκτελέστηκε στο σύστημα που αναφέρθηκε προηγουμένως στην εισαγωγή και η χρονομέτρηση έγινε με την συνάρτηση ***gettimeofday(struct timeval *, struct tzp *)***. Αρχικοποιήθηκαν δύο μεταβλητές τύπου ***struct timeval start, end*** για την αρχή της χρονομέτρησης και για το τέλος της χρονομέτρησης. Αρχικά καλούνται οι εντολές ***gettimeofday(&start, NULL), serial_primes(UPTO)*** και ***gettimeofday(&end, NULL)***. Αντίστοιχα, υπολογίζεται και ο χρόνος εκτέλεσης και για την συνάρτηση ***openmp_primes(UPTO)*** όπου είναι η συνάρτηση παραλληλοποίησης του αλγορίθμου. Στη συνέχεια υπολογίζονται οι χρόνοι ***exectime exectimepar***. Η μετατροπή των χρόνων από ***struct*** σε ***double*** γίνεται με τον τύπο

$$
\begin{aligned}
exectime, exectimepar \\
&= (double)(end.tv\_usec - start.tv\_usec) * 1\mathrm{E} - 06 \\
&+ (double)(end.tv\_sec - start.tv\_sec);
\end{aligned}
$$

Χρησιμοποιήσαμε από 1 μέχρι 4 νήματα για την παραλληλοποίηση, η οποία γίνεται δυναμικά με την εντολή

$$omp\_set\_num\_threads(NumOfThreds)$$

πριν την έναρξη της παράλληλης περιοχής.

Κάθε πείραμα εκτελέστηκε τέσσερις φορές και υπολογίστηκαν οι μέσοι χρόνοι. Οι χρόνοι αυτοί υπολογίζουν μονό την εκτέλεση του εκάστοτε αλγορίθμου (σειριακός ή παράλληλος). Τα αποτελέσματα δίνονται στον παρακάτω πίνακα (οι χρόνοι είναι σε sec).

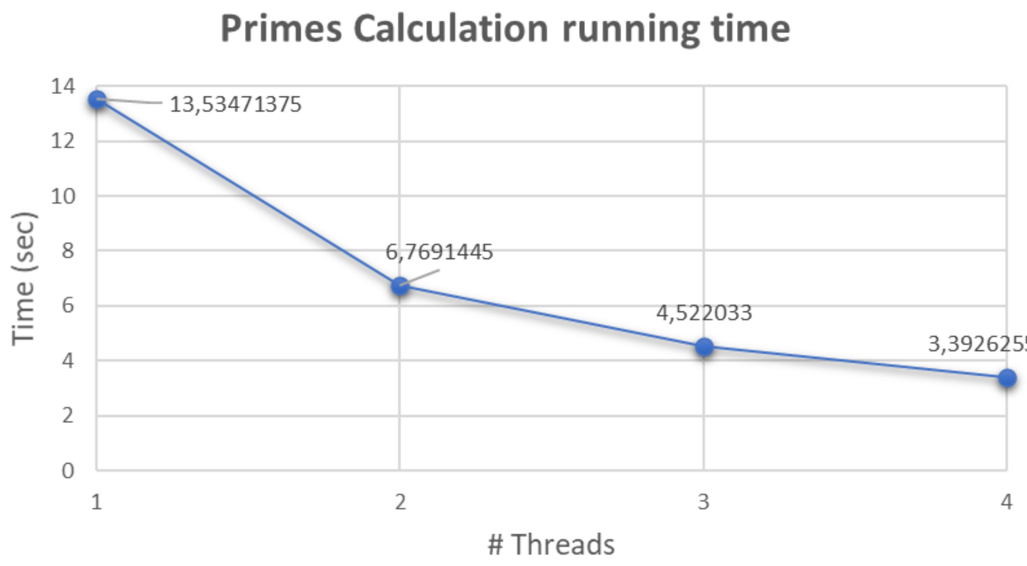
Το πείραμα είναι με παραλληλοποίηση έξω από το for loop. Οι μετρήσεις φαίνονται παρακάτω:

Νήματα	**1η Εκτέλεση**	**2η Εκτέλεση**	**3η Εκτέλεση**	**4η Εκτέλεση**	**Μέσος Χρόνος**
1	13,533682	13,537357	13,534223	13,533593	13,53471375
2	6,769147	6,769081	6,76921	6,76914	6,7691445
3	4,522058	4,522141	4,52164	4,522293	4,522033
4	3,390575	3,391643	3,390634	3,39765	3,3926255

Πίνακας 2: Αποτελέσματα Primes Calculation Algorithm

Σειριακός Χρόνος Προγράμματος = 13.561237 sec

Με βάση τον παραπάνω πίνακα προκύπτει το εξής γράφημα:



Εικόνα 1: Γράφημα Μέσου Χρόνου Primes Calculation

2.4. Σχόλια:

Με βάση τα παραπάνω αποτελέσματα παρατηρούμε πως η αύξηση αριθμού των νημάτων επιφέρει μείωση του χρόνου εκτέλεσης περίπου ιδανικά (δηλαδή με χρήση k νημάτων έχουμε 1/k μείωση του σειριακού χρόνου). Επίσης, η πολιτική schedule έχει επίδραση στην μείωση του χρόνου, καθώς κατανέμονται σωστά οι επαναλήψεις στα νήματα με τον ίδιο φόρτο εργασίας ανά νήμα. Η πολιτική που επιφέρει την μεγαλύτερη μείωση στον χρόνο εκτέλεσης είναι η ***static*** με ***chunk_size = 1000***. Αυτό συμβαίνει καθώς οι επαναλήψεις χωρίζονται σε $((N-1)/2)/1000$ ανά νήμα και το κάθε νήμα θα εκτελεί 1000 συνεχόμενες επαναλήψεις. Επιπλέον, δεν υπάρχει ανταγωνισμός μεταξύ των νημάτων όσον αφορά την επιλογή των τμημάτων σε σχέση με τις υπόλοιπες πολιτικές (***dynamic, guided***).

Όλα τα παραπάνω αποτελέσματα είναι μάλλον αναμενόμενα διότι οι επαναλήψεις κατανέμονται ισόποσα στα νήματα και μάλιστα με βάσει του ***chunk_size***, όπου κάθε μία έχει παρόμοιο φόρτο υπολογισμών με τις άλλες. Συνεπώς, η αύξηση των νημάτων φέρει μείωση του χρόνου εκτέλεσης χωρίς τα νήματα να ανταγωνίζονται μεταξύ τους και μάλιστα η κοινόχρηστη μεταβλητή ***count*** δεν επηρεάζει καθόλου τους υπολογισμούς, παρόλο που γίνεται διαμοίραση της μεταβλητής με τοπικά αντίγραφα στα νήματα.

3. Άσκηση-2:

3.1. Το πρόβλημα:

Σε αυτή την άσκηση μας ζητείται να παραλληλοποιηθεί ο αλγόριθμος φιλτραρίσματος εικόνων με την μέθοδο Gauss. Αυτό θα γίνει με δύο τρόπους. Ο πρώτος τρόπος παραλληλοποίησης θα γίνει χρησιμοποιώντας την μέθοδο παραλληλοποίησης των for loops και ο δεύτερος τρόπος θα γίνει με χρήση των tasks.

3.2. Μέθοδοι Παραλληλοποίησης:

Για την παραλληλοποίηση, χρησιμοποιήθηκε το σειριακό πρόγραμμα που υπήρχε στην ιστοσελίδα του μαθήματος (gaussian-blur.c). Σε αυτό το πρόγραμμα τροποποιήθηκε η σειριακή εκδοχή του αλγορίθμου gaussian_blur_serial() σε δύο περιπτώσεις. Συμπληρώθηκε η συνάρτηση gaussian_blur_omp_loops() η οποία χρησιμοποιεί την μέθοδο των for loops και αντίστοιχα η συνάρτηση gaussian_blur_omp_tasks() η οποία χρησιμοποιεί την μέθοδο των tasks.

Ας δούμε πρώτα την μέθοδο παραλληλοποίησης των loops. Για την παραλληλοποίηση των βρόχων χρησιμοποιήθηκε η εντολή

$$\#pragma\ omp\ parallel\ private(i, j, row, col, weightSum, redSum, greenSum, blueSum)$$

όπου ορίζουμε τις μεταβλητές μας ως ιδιωτικές για κάθε νήμα, έτσι ώστε να μην επηρεάζει το ένα νήμα το άλλο κατά τους υπολογισμούς τους. Έτσι, το κάθε νήμα θα κάνει στον δικό του χώρο τους αντίστοιχους υπολογισμούς, δίχως να επηρεάζονται μεταξύ τους. Στην συνέχεια προστέθηκε και η εντολή

$$\#pragma\ omp\ for\ schedule(static \mid dynamic \mid guided\ [, chunk])$$

πριν από τον πρώτο βρόχο for ώστε να γίνει η παραλληλοποίηση της επεξεργασίας της εικόνας.

Η δεύτερη μέθοδος παραλληλοποίησης είναι με την χρήση των tasks. Για την παραλληλοποίηση των βρόχων χρησιμοποιήθηκε η εντολή

$$\#pragma\ omp\ parallel\ private(i, j, row, col, weightSum, redSum, greenSum, blueSum)$$

όπου ορίζουμε τις μεταβλητές μας ως ιδιωτικές για κάθε νήμα, έτσι ώστε να μην επηρεάζει το ένα νήμα το άλλο κατά τους υπολογισμούς τους. Έτσι, το κάθε νήμα θα κάνει στον δικό του χώρο τους αντίστοιχους υπολογισμούς, δίχως να επηρεάζονται μεταξύ τους. Στην συνέχεια προστέθηκε και η εντολή

$$\#pragma\ omp\ single\ nowait$$

ώστε σε κάθε task να μπαίνει ένα νήμα στην παράλληλη περιοχή χωρίς να επηρεάζονται μεταξύ τους και στο τέλος τα υπόλοιπα να περιμένουν στο τέλος της παράλληλης περιοχής για να ελαχιστοποιηθούν οι καθυστερήσεις. Καθώς κάθε γραμμή της εικόνας αποτελεί και ένα task, τότε η εντολή

$$\#pragma\ omp\ task\ firstprivate(i, weightSum, redSum, greenSum, blueSum)$$

πριν το δεύτερο for loop, ώστε το task να αποτελεί ολόκληρη την γραμμή μιας εικόνας. Οι μεταβλητές ***i***, ***weightSum***, ***redSum***, ***greenSum*** και ***blueSum*** είναι ***firstprivate***, καθώς κάθε task θα πρέπει να έχει τον δικό του χώρο και να μην επηρεάζει τα επόμενα tasks. Επίσης, κάθε task θα πρέπει να γνωρίζει τις τιμές των τοπικών μεταβλητών ώστε να μπορεί να βλέπει αρχικά σε ποια γραμμή βρίσκεται και τις τιμές των βαρών και αθροισμάτων των αντίστοιχων καναλιών της εικόνας.

3.3. Πειραματικά Αποτελέσματα – Μετρήσεις:

Το πρόγραμμα εκτελέστηκε στο σύστημα που αναφέρθηκε προηγουμένως στην εισαγωγή και η χρονομέτρηση έγινε με την συνάρτηση ***gettimeofday(struct timeval *, struct tzp *)***. Αρχικοποιούνται οι μεταβλητές ***start*** και ***end*** στην αρχή της main ως ***struct timeval*** και χρησιμοποιούνται τα πεδία τους για την χρονομέτρηση των συναρτήσεων. Η συνάρτηση αυτή εισάγεται πριν την κληθείσα συνάρτηση (σειριακού, παράλληλου αλγορίθμου) και μετά για να παρθούν οι χρόνοι. Τέλος, για κάθε μέθοδο χρησιμοποιείται η εντολή

$$exec\_time = (double)(end.tv\_sec - start.tv\_sec) + (double)(end.tv\_usec - start.tv\_usec) * 1E - 06$$

αυτή η διαδικασία γίνεται με χρήση της συνάρτησης ***timeit***() που δέχεται ως όρισμα την συνάρτηση επεξεργασίας και τις αντίστοιχες παραμέτρους της, δηλαδή την ακτίνα θόλωσης και την εικόνα εξόδου και επιστρέφεται η θολωμένη εικόνα.

Χρησιμοποιήσαμε από 1 μέχρι 4 νήματα για την παραλληλοποίηση, η οποία γίνεται δυναμικά με την εντολή

$$omp\_set\_num\_threads(NumOfThreds)$$

πριν την έναρξη της παράλληλης περιοχής.

Κάθε πείραμα εκτελέστηκε τέσσερις φορές και υπολογίστηκαν οι μέσοι χρόνοι. Οι χρόνοι αυτοί υπολογίζουν μονό την εκτέλεση του εκάστοτε αλγορίθμου (σειριακός ή παράλληλος). Τα αποτελέσματα δίνονται στους παρακάτω πίνακες (οι χρόνοι είναι σε sec). Επιπλέον, τα παρακάτω πειράματα έγιναν με ακτίνα $r = 8$ και με την εικόνα 1500.bmp.

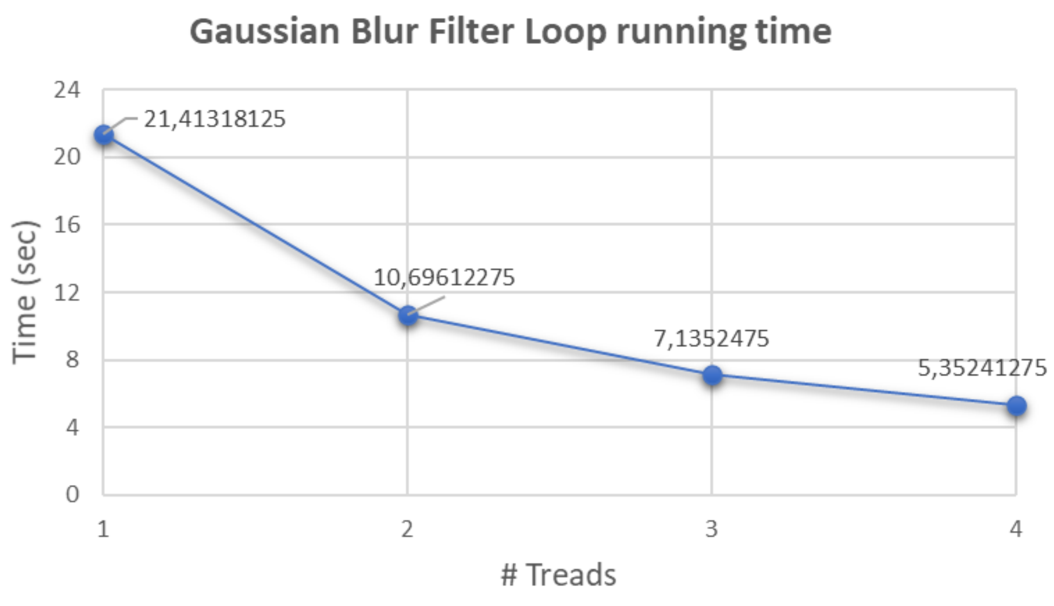
Το πρώτο πείραμα είναι με παραλληλοποίηση έξω από το for loop. Οι μετρήσεις φαίνονται παρακάτω:

Νήματα	**1η Εκτέλεση**	**2η Εκτέλεση**	**3η Εκτέλεση**	**4η Εκτέλεση**	**Μέσος Χρόνος**
1	21,530214	21,473641	21,327606	21,321264	21,41318125
2	10,664858	10,70748	10,755889	10,656264	10,69612275
3	7,179823	7,109340	7,1218470	7,129980	7,1352475
4	5,346917	5,344965	5,339817	5,377952	5,35241275

Πίνακας 3: Αποτελέσματα Gaussian Blur Loops Method

Σειριακός Χρόνος Προγράμματος = 21,46479525 sec

Με βάση τον παραπάνω πίνακα προκύπτει το εξής γράφημα:



Εικόνα 2: Γράφημα Μέσου Χρόνου Gaussian Blur Loops Method

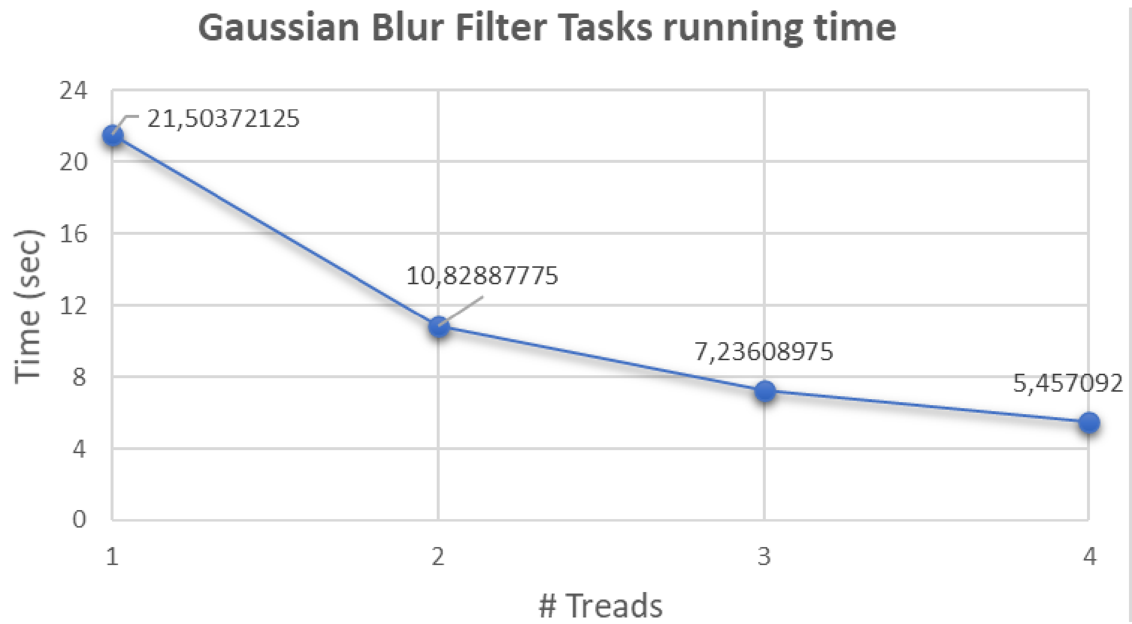
Το δεύτερο πείραμα είναι με παραλληλοποίηση με χρήση των tasks. Οι μετρήσεις φαίνονται παρακάτω:

Νήματα	**1η Εκτέλεση**	**2η Εκτέλεση**	**3η Εκτέλεση**	**4η Εκτέλεση**	**Μέσος Χρόνος**
1	21,519325	21,601328	21,433935	21,460297	21,50372125
2	10,786669	10,818328	10,924426	10,786088	10,82887775
3	7,2299560	7,2265390	7,2707180	7,2171460	7,23608975
4	5,4673560	5,4517920	5,4527220	5,4564980	5,457092

Πίνακας 4: Αποτελέσματα Gaussian Blur Tasks Method

Σειριακός Χρόνος Προγράμματος = 21,46479525 sec

Με βάση τον παραπάνω πίνακα προκύπτει το εξής γράφημα:



Εικόνα 3: Γράφημα Μέσου Χρόνου Gaussian Blur Tasks Method

3.4. Σχόλια:

Με βάση τα αποτελέσματα βλέπουμε ότι η αύξηση του αριθμού των νημάτων μειώνει σημαντικά τους χρόνους εκτέλεσης και μάλιστα περίπου ιδανικά (δηλαδή k νήματα δίνουν περίπου το 1/k του σειριακού χρόνου) και στις δύο μεθόδους. Επιπρόσθετα, όσο αυξάνουμε και την ακτίνα θόλωσης, τόσο πιο θολή θα γίνεται η εικόνα.

Επίσης, η καλύτερη πολιτική schedule που εφαρμόστηκε ήταν η static στην πρώτη μέθοδο (χωρίς chunk) καθώς χωρίζεται ανά νήμα περίπου $\boldsymbol{imgin{-}{>}header.height/NumThread}$ οι επαναλήψεις. Επίσης, στην πρώτη μέθοδο καλύτερα αποτελέσματα βγήκαν κατά την παραλληλοποίηση του πρώτου loop σε σχέση με τα υπόλοιπα loops και αντίστοιχα με τις υπόλοιπες πολιτικές διαμοίρασης. Αυτό σχετίζεται με το τρόπο διαχωρισμού των επαναλήψεων στα νήματα με την βέλτιστη λύση να είναι η στατική δρομολόγηση.

Σχετικά με την δεύτερο μέθοδο, παρατηρούμε πως η χρήση των tasks ανά γραμμή επιφέρει καλύτερα αποτελέσματα, καθώς ο ίδιος ο αλγόριθμος θόλωσης επεξεργάζεται τα pixel του ανά γραμμή. Επιπλέον, αυτό που παρατηρούμε είναι πως κάθε νήμα θα αποτελεί ένα task και με την αύξηση των νημάτων θα μειώνεται σημαντικά ο χρόνος εκτέλεσης.

Τα αποτελέσματα αυτά που παρατηρούμε είναι λογικά, καθώς ο διαμοιρασμός των εργασιών κατανέμονται ισόποσα στα νήματα και έχουν παρόμοιο φόρτο υπολογισμών και στην πρώτη μέθοδο και στην δεύτερη μέθοδο αντίστοιχα. Επομένως, κατά την αύξηση των νημάτων θα έχουμε και περισσότερους υπολογισμούς ταυτόχρονα, το οποίο μας εξασφαλίζει και λιγότερο χρόνο εκτέλεσης.

4. Άσκηση-3:

4.1. Η οδηγία Taskloop:

Από την έκδοση 4.5 του OpenMP και μετά, υποστηρίζεται η εντολή ***taskloop***, η οποία επιτρέπει τις επαναλήψεις ενός βρόχου ***for*** να εκτελούνται μέσω ***tasks***. Η σύνταξη της εντολής είναι η εξής:

$$\#\textbf{pragma omp taskloop}\ [clause[[,]clause]\ldots]\ new - line$$

$$for - loops$$

όπου το ***clause*** είναι ένα από το παρακάτω:

```
if([ taskloop :] scalar-expr)
shared(list)
private(list)
firstprivate(list)
lastprivate(list)
default(shared | none)
grainsize(grain-size)
num_tasks(num-tasks)
collapse(n)
final(scalar-expr)
priority(priority-value)
untied
mergeable
nogroup
```

Εικόνα 4: Taskloop clauses

Η οδηγία ***taskloop*** έχει ως απώτερο σκοπό την χρήση των ***tasks*** σε βρόχους επανάληψης ***for***. Οι επαναλήψεις κατανέμονται σε όλα τα ***tasks*** που δημιουργούνται από την οδηγία, έτσι ώστε να δρομολογηθούν και να εκτελεστούν. Όταν ένα νήμα συναντά την παραπάνω οδηγία, τότε αυτή τμηματοποιεί τους βρόχους και τους εισάγει μέσα σε ***tasks***, έτσι ώστε να έχουμε παράλληλη εκτέλεση των επαναλήψεων του βρόχου. Τα δεδομένα που εκτελούνται μέσα στους βρόχους θα είναι αρχικοποιημένα, ανάλογα με τον ορισμό τους που έχουν στην εντολή ***taskloop***.

Επίσης, η σειρά αρχικοποίησης των ***tasks*** γίνεται τυχαία. Σε περίπτωση που χρειάζεται να οριστεί ο αριθμός των επαναλήψεων που θα έχει κάθε ***task***, τότε χρησιμοποιείται το ***clause grainsize(pos_num)*** με τον ελάχιστο αριθμό επαναλήψεων να είναι το ***pos_num***.

Σε περίπτωση που χρειάζεται κάποιος συγκεκριμένος αριθμός ***tasks***, τότε χρησιμοποιείται το ***clause num_tasks(pos_num)***, για να δημιουργηθούν τόσα ***tasks*** όσο και το ***pos_num***. Έτσι, το κάθε ***task*** θα πρέπει να έχει τουλάχιστον μία επανάληψη του βρόχου που είναι υπό παραλληλοποίηση.

Επιπρόσθετα, μπορεί να οριστεί και ο αριθμός των loops που μπορούν να συσχετιστούν με τα ***taskloops***. Δηλαδή, σε περίπτωση εμφωλιασμένων βρόχων, τότε χρησιμοποιείται το ***clause collapse(pos_num)***, για να μπορέσει να παραλληλοποιήσει με ***tasks*** και τον εσωτερικό βρόχο. Σε περίπτωση που δεν υπάρχει το συγκεκριμένο ***clause***, τότε το ***taskloop*** παραλληλοποιεί μόνο τον βρόχο που βρίσκεται ακριβώς από κάτω του. Στην περίπτωση που έχουμε πολλαπλά loops με σχετίζονται με ένα συγκεκριμένο ***taskloop***, τότε οι επαναλήψεις των εκάστοτε loops καταλαμβάνουν τον ίδιο χώρο υπολογισμών και τμηματοποιούντε ανάλογα με τους αριθμούς των ***grainsize(pos_num)*** και ***num_tasks(pos_num)***.

Εξ ορισμού το ***taskloop*** εκτελείται σαν να υπάρχει μέσα σε ένα ***taskgroup*** χωρίς statements ή οδηγίες εκτός του ***taskloop***. Έτσι, ένα ***taskloop*** στην ουσία δημιουργεί μία implicit ***taskgroup*** περιοχή. Στην περίπτωση που δεν υπάρχει το ***nogroup clause***, τότε δεν δημιουργείται implicit ***taskgroup*** περιοχή.

4.2. Λειτουργία εντολής Taskloop:

Παρακάτω θα δοθεί και ένα παράδειγμα πολλαπλασιασμού πινάκων με χρήση των ***taskloop***. Καθώς έχουμε τριπλό βρόχο, η καλύτερη μέθοδος παραλληλοποίησης είναι στον δεύτερο για να επεξεργάζονται κατά γραμμές. Στην ουσία κάθε γραμμή θα αποτελεί ένα task. Τα ***i***, ***k*** θα πρέπει να είναι private για να μην επηρεάζουν τα υπόλοιπα ***task***. Επίσης, οι πίνακες ***A*** και ***B*** θα πρέπει να είναι shared μεταβλητές, καθώς δεν επηρεάζονται τα ***tasks*** μεταξύ του κατά τον υπολογισμό του γινομένου των δύο πινάκων. Αυτό συμβαίνει διότι τα υποσύνολα των ***tasks*** είναι ανεξάρτητα μεταξύ τους.

Η εκδοχή του προγράμματος φαίνεται παρακάτω:

```c
/* OpenMP matrix multiplication */
void matmul_omp_taskloop()
{
    /* TODO: Implement parallel matrix multiplication with OpenMP taskloop */
    int i, j, k;

    #pragma omp parallel num_threads(4)
    {
        #pragma omp single
        {
            for (i = 0; i < N; ++i)
            {
                #pragma omp taskloop private(i,k) shared(A,B)
                    for (j = 0; j < K; ++j)
                    {
                        for (k = 0; k < M; ++k)
                        {
                            C[i][j] += A[i][k] * B[k][j];
                        }
                    }
            }
        }
    }
}
```

Κώδικας 1: Matrix Multiplication Taskloop Method

Επιπλέον, το πρόγραμμα χρονομετρήθηκε και συγκρίθηκε με την σειριακή του εκδοχή στο σύστημα που αναφέρθηκε στην εισαγωγή. Επίσης, οι εκτελέσεις έγιναν με χρήση νημάτων από 1 έως 4 με την ***num_threads(Num)*** στον ορισμό της παράλληλης περιοχής.

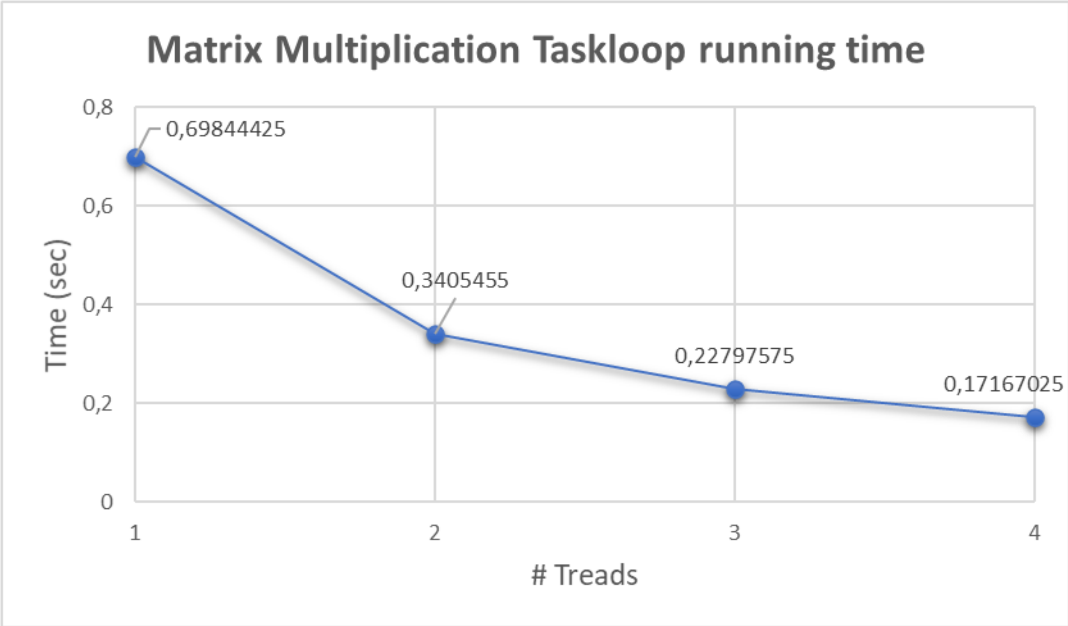
Κάθε πείραμα εκτελέστηκε 4 φορές και υπολογίστηκαν οι μέσοι όροι. Τα αποτελέσματα δίνονται στον παρακάτω πίνακα (οι χρόνοι είναι σε sec):

Νήματα	**1η Εκτέλεση**	**2η Εκτέλεση**	**3η Εκτέλεση**	**4η Εκτέλεση**	**Μέσος Χρόνος**
1	0,707138	0,69524	0,695264	0,696135	0,69844425
2	0,341796	0,338417	0,341819	0,34015	0,3405455
3	0,228545	0,228161	0,227745	0,227452	0,22797575
4	0,171729	0,171659	0,171228	0,172065	0,17167025

Πίνακας 5: Αποτελέσματα Πολ/σμού Πινάκων

Σειριακός Χρόνος Προγράμματος = 0.707138 sec

Και έχουμε και το εξής γράφημα:



Εικόνα 5: Γράφημα Μέσου Χρόνου Matrix Multiplication Taskloop Method

Με βάση τα αποτελέσματα βλέπουμε ότι η αύξηση του αριθμού των νημάτων μειώνει σημαντικά τους χρόνους εκτέλεσης και μάλιστα περίπου ιδανικά (δηλαδή k νήματα δίνουν περίπου το 1/k του σειριακού χρόνου) και στις δύο μεθόδους.

Συμπερασματικά, η εντολή taskloop χρησιμοποιεί tasks για την παραλληλοποίηση των βρόχων. Αποτελεί μία αποδοτική μέθοδος καθώς διαιρεί και τμηματοποιεί τον φόρτο εργασίας που έχει κάθε επανάληψη ισόποσα στα ***tasks***. Έτσι, ο χρόνος υπολογισμών κρίσιμων εφαρμογών μειώνεται σημαντικά.

Βιβλιογραφία

[1] Β. Β. Δημακόπουλος, “*Παράλληλα Συστήματα και Προγραμματισμός*”, (1η Αναθεωρημένη Έκδοση), Εκδόσεις Κάλλιπος, 2017.

[2] *OpenMP API 4.5 Complete Specifications*, November 2015.

[3] *OpenMP API 4.5 Reference Guide – C/C++*, November 2015.

[4] P. Pacheco, “*An Introduction to Parallel Programming*”, Morgan Kaufmann Publishers - Elsevier, 2015.

[5] R. Chandra, L. Dagun, D. Kohr, D. Maydan, J. McDonald, R. Menon, “*Parallel Programming in OpenMP*”, Morgen Kaufmann Publishers, 2001.