**ΜΥΕ023–Παράλληλα Συστήματα και Προγραμματισμός**

**Σετ Ασκήσεων #1**

Γεώργιος Παπαδάτος

ΑΜ: 3306

Email: [cs03306@uoi.gr](mailto:cs03306@uoi.gr)

Το 1ο σετ ασκήσεων αφορά τον παράλληλο προγραμματισμό μέσω του προτύπου OpenMP. Ζητείται η παραλληλοποίηση ενός προγράμματος με 2 τρόπους, καθώς και η μελέτη και περιγραφή των “Doacross loops”.

**Άσκηση 1**

Όλες η μετρήσεις έγιναν στο παρακάτω σύστημα:

|  |  |
| --- | --- |
| **Όνομα υπολογιστή** | opti3060ws08 |
| **Επεξεργαστής** | Intel i3-8300 |
| **Πλήθος πυρήνων** | 4 |
| **Μεταφραστής** | gcc v7.5.0 |

**Το πρόβλημα**

Στην άσκηση αυτή ζητείται να παραλληλοποιηθεί ο πολλαπλασιασμός πινάκων με συνολικά 6 τρόπους. Συγκεκριμένα πρέπει να παραλληλοποιήσουμε κάθε φορά μόνο τον 1 από τους 3 βρόγχους for, θέτοντας τον διαμοιρασμό των επαναλήψεων (scheduling) σε static και dynamic.

**Μέθοδος παραλληλοποίησης**

Χρησιμοποιήθηκε το σειριακό πρόγραμμα από την ιστοσελίδα του μαθήματος. Για την παραλληλοποίηση του 1ου βρόγχου χρησιμοποιήθηκε η εντολή

**#pragma omp parallel private(j,k,sum)**

πριν την αρχή των βρόγχων για να ορίσουμε την παράλληλη περιοχή. Η περιοχή περιείχε και τους 3 βρόγχους. Οι μεταβλητές j, k και sum πρέπει να είναι ιδιωτικές ώστε κάθε νήμα να κάνει τους υπολογισμούς στο δικό του χώρο χωρίς να επηρεάζει τα υπόλοιπα ενώ οι Α , Β και C είναι κοινόχρηστες. Δεν απαιτείται αμοιβαίος αποκλεισμός μιας και κάθε νήμα επηρεάζει διαφορετικά στοιχεία του C. Στην συνέχεια πριν τον 1ο βρόγχο προστέθηκε η εντολή

**#pragma omp for schedule(static** ή **dynamic)**

η οποία είναι υπεύθυνη για την παράλληλη εκτέλεση του βρόγχου. Εδώ έχουμε την δυνατότητα να επιλέξουμε τον τρόπο διαμοιρασμού των πακέτων μέσο του schedule(). Οι εκτελέσεις έχουν γίνει με static και dynamic scheduling.

Για την παραλληλοποίηση του 2ου βρόγχου δεν χρησιμοποιήθηκε η εντολή omp parallel αλλά ενσωματώθηκε απευθείας στην omp for. Επομένως αμέσως πριν τον 2ο βρόγχο προστέθηκε η εντολή

**#pragma omp parallel for private(k,sum) schedule(static** ή **dynamic) firstprivate(i)**

Αυτή την φορά οι μεταβλητές k και sum είναι ιδιωτικές καθώς το κάθε νήμα θα τις χρησιμοποιήσει για τους δικούς του υπολογισμούς. Το i πρέπει να είναι firstprivate έτσι ώστε το κάθε νήμα να έχει το δικό του i αλλά να είναι αρχικοποιημένο με την σωστή τιμή (π.χ. αν το i ήταν κοινόχρηστο κάποιο νήμα μπορεί να μην προλάβει να ολοκληρώσει την δουλειά του και να αλλάξει το i, δημιουργώντας έτσι εσφαλμένο αποτέλεσμα). Επίσης έχουμε την δυνατότητα επιλογής του scheduling, και έχει πραγματοποιηθεί εκτέλεση και με τους 2 τρόπους

Αντίστοιχα, για την παραλληλοποίηση του 3ου βρόγχου η εντολή omp parallel ενσωματώθηκε στην omp for. Όμως υπήρξε μία ακόμα αλλαγή στο πρόγραμμα πέρα από την προσθήκη της εντολής

**#pragma omp parallel for firstprivate(i,j) reduction(+: sum) schedule(static** ή **dynamic)**

Συγκεκριμένα η μεταβλητή sum αρχικοποιήθηκε έξω από την παράλληλη περιοχή γιατί αλλιώς δεν δούλευε σωστά το τελικό άθροισμα (reduction(+: sum)). Οι μεταβλητές i, j πρέπει να είναι ιδιωτικές καθώς χρησιμοποιούνται ξεχωριστά από τα νήματα για την επιλογή των στοιχείων του πολλαπλασιασμού, αλλά πρέπει να είναι αρχικοποιημένες στην τελευταία τιμή τους. Το reduction επιτρέπει την εύκολη παραλληλοποίηση της πρόσθεσης και εξασφαλίζει ότι στο τέλος του for θα έχουμε το τελικό σωστό άθροισμα. Όπως και παραπάνω, έχει πραγματοποιηθεί εκτέλεση και με τους 2 τρόπους scheduling.

**Πειραματικά αποτελέσματα – μετρήσεις**

Τα προγράμματα εκτελέστηκαν στο σύστημα που αναφέραμε στην αρχή και η χρονομέτρηση έγινε με τη συνάρτηση omp\_get\_wtime(). Συγκεκριμένα, αποθηκεύσαμε την αρχική χρονική στιγμή πριν την εκτέλεση των βρόγχων και την αφαιρέσαμε από την τελική χρονική στιγμή μετά την εκτέλεση όλων των βρόγχων.

Στην εκτέλεση των παράλληλων κομματιών χρησιμοποιήθηκαν 4 νήματα. Αυτό επιτεύχθηκε με την απενεργοποίηση του ευέλικτου πλήθους νημάτων με την εντολή

**omp\_set\_dynamic(0);**

και τον ορισμό του αριθμού νημάτων με την εντολή

**omp\_set\_num\_threads(4);**

Κάθε πείραμα εκτελέστηκε 4 φορές και υπολογίστηκαν οι μέσοι όροι. Προσοχή δόθηκε οι χρόνοι να μην συμπεριλαμβάνουν την ανάγνωση και εγγραφή των αρχείων. Τα αποτελέσματα δίνονται στον παρακάτω πίνακα (οι χρόνοι είναι σε δευτερόλεπτα):

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Πρόγραμμα | 1η εκτέλεση | 2η εκτέλεση | 3η εκτέλεση | 4η εκτέλεση | Μέσος όρος |
| Serial | 88.595868 | 95.822262 | 95.581512 | 95.196698 | 93.799085 |
| Parallel for #1 static | 35.078201 | 33.281067 | 33.522918 | 32.544863 | 33.606762 |
| Parallel for #1 dynamic | 36.530240 | 32.134451 | 35.667790 | 35.032520 | 34.841250 |
| Parallel for #2 static | 44.116407 | 43.528094 | 44.096005 | 43.128043 | 43.717137 |
| Parallel for #2 dynamic | 27.364888 | 27.380109 | 27.286072 | 27.450719 | 27.370447 |
| Parallel for #3 static | 46.208635 | 46.226932 | 46.293383 | 46.447735 | 46.294171 |
| Parallel for #3 dynamic | 223.259482 | 222.055907 | 219.940430 | 220.830895 | 221.521679 |

Με βάση τα παραπάνω αποτελέσματα προκύπτει το ακόλουθο γράφημα.

**Σχόλια**

Παραλληλοποίηση 1ου βρόγχου

Με βάση τα αποτελέσματα παρατηρούμε ότι η παραλληλοποίηση του 1ου βρόγχου μειώνει σημαντικά τον χρόνο εκτέλεσης όμως δεν φτάνει την ιδανική μείωση του να είναι το 25% του αρχικού χρόνου. Σε αυτή την περίπτωση η αλλαγή του scheduling δεν είχε μεγάλη επίδραση στο αποτέλεσμα. Η παραλληλοποίηση του 1ου βρόγχου είναι μία κλασική περίπτωση παραλληλοποίησης πολύ μεγάλου φόρτου και έχει τα αναμενόμενα αποτελέσματα.

Παραλληλοποίηση 2ου βρόγχου

Στην περίπτωση της παραλληλοποίησης του 2ου βρόγχου βλέπουμε επίσης σημαντική μείωση του χρόνου εκτέλεσης. Αυτή την φορά όμως παίζει σημαντικό ρόλο η επιλογή του scheduling, καθώς με static scheduling επιτυγχάνουμε περίπου 50% βελτίωση της απόδοσης, ενώ με dynamic scheduling πλησιάζουμε πολύ στην ιδανική βελτίωση του 75%.

Παραλληλοποίηση 3ου βρόγχου

Τέλος, στην περίπτωση του 3ου βρόγχου τα αποτελέσματα δεν είναι μόνο θετικά. Πιο συγκεκριμένα, με το static scheduling καταφέραμε βελτίωση της τάξης του 50%, αλλά με το dynamic scheduling προκαλέσαμε την καθυστέρηση της εκτέλεσης κατά πάνω από 100%! Αυτό πιθανώς προκύπτει από την συνεχή δημιουργία και καταστροφή νημάτων, καθώς το κάθε νήμα με το που δημιουργείται εκτελεί μόνο έναν πολλαπλασιασμό και μία πρόσθεση πριν καταστραφεί. Αυτό έχει σαν αποτέλεσμα την τεράστια κατασπατάληση πόρων, και έτσι καταλήγουμε με μία τόσο μεγάλη καθυστέρηση σε σχέση με το σειριακό πρόγραμμα.

**Άσκηση 2**

Όλες η μετρήσεις έγιναν στο παρακάτω σύστημα:

|  |  |
| --- | --- |
| **Όνομα υπολογιστή** | opti3060ws07 |
| **Επεξεργαστής** | Intel i3-8300 |
| **Πλήθος πυρήνων** | 4 |
| **Μεταφραστής** | gcc v7.5.0 |

Η αλλαγή συστήματος συνέβη γιατί ο υπολογιστής opti3060ws08 είχε ήδη κάποιον χρήστη και έτσι ήταν πιθανό τα νούμερα των 2 ασκήσεων να μην είναι σχετικά μεταξύ τους.

**Το πρόβλημα**

Στην άσκηση αυτή ζητείται να παραλληλοποιηθεί ο πολλαπλασιασμός πινάκων με την χρήση των tasks. Συγκεκριμένα ζητείται το κάθε task να είναι ο υπολογισμός ενός block (υποπίνακα) του αποτελέσματος, μεγέθους S × S, όπου το S θα εισάγεται ως παράμετρος κατά την εκτέλεση του προγράμματος.

**Μέθοδος παραλληλοποίησης**

Χρησιμοποιήθηκε το σειριακό πρόγραμμα από την ιστοσελίδα του μαθήματος. Αρχικά φτιάξαμε μία δομή ελέγχου για την εισαγωγή του S ως παράμετρο, έτσι ώστε αν υπάρξει κάποιο πρόβλημα να πάρουμε το σωστό μήνυμα λάθους. Στην συνέχεια αρχικοποιήσαμε κάποιες βοηθητικές μεταβλητές M και totalSubMat που σχετίζονται με τον αριθμό των υποπινάκων. Αρχικοποιήσαμε την παράλληλη περιοχή χρησιμοποιώντας την εντολή

**#pragma omp parallel**

πριν την έναρξη των βρόγχων. Αμέσως μετά ακολουθεί η εντολή

**#pragma omp single nowait**

καθώς θέλουμε μόνο ένα νήμα να δημιουργήσει τα tasks. Το nowait προστέθηκε γιατί δεν υπάρχει λόγος τα υπόλοιπα νήματα να περιμένουν να ολοκληρωθούν όλα τα tasks για να ξεκινήσουν να τα εκτελούν.

Για την δημιουργία των tasks φτιάχτηκε ένας επιπλέον βρόγχος for με την εντολή

**for (taskNum = 0; taskNum < totalSubMat; taskNum++)**

Το taskNum περιέχει τον αναγνωριστικό αριθμό του block που θα φτιάχνει το task και ο βρόγχος αυτός θα εκτελεστεί τόσες φορές όσοι είναι οι υποπίνακες που πρέπει να δημιουργηθούν.

Μέσα στον βρόγχο δημιουργούμε τα tasks με την εντολή

**#pragma omp task private(i,j,k,sum) firstprivate(taskNum)**

Η οποία περιέχει τους 3 βρόγχους που κάνουν τον πολλαπλασιασμό πινάκων. Τα i, j, k, sum πρέπει να είναι ιδιωτικά καθώς το κάθε task θα εκτελεστεί για διαφορετικά i, j, k και το άθροισμα επίσης δεν θα είναι κοινό. Ο αναγνωριστικός αριθμός taskNum χρησιμοποιείται για να γνωρίζει το κάθε task για ποιο block είναι υπεύθυνο, οπότε πρέπει να είναι ιδιωτικός και να αρχικοποιείται στην κατασκευή του task με τον τελευταία τιμή του. Οι πίνακες A, B και C δεν υπάρχει πρόβλημα να είναι κοινόχρηστοι καθώς το κάθε task θα χρησιμοποιεί διαφορετικό κομμάτι τους.

Πριν τους βρόγχους του πολλαπλασιασμού αρχικοποιήθηκαν 2 μεταβλητές x και y οι οποίες περιέχουν τους αριθμούς γραμμής και στήλης του υποπίνακα

**int x = taskNum / M;**

**int y = taskNum % M;**

όπου M ο αριθμός υποπινάκων που υπάρχουν σε μία γραμμή ή στήλη του πίνακα αποτελέσματος.

Τέλος, οι 2 από τους 3 βρόγχους τροποποιήθηκαν ώστε να υπολογίζουν μόνο τους πολλαπλασιασμούς που αφορούν το συγκεκριμένο block

**for (i = x\*S; i < (x+1)\*S; i++)**

**for (j = y\*S; j < (y+1)\*S; j++)**

Πλέον ο βρόγχος ξεκινάει από την το σημείο έναρξης του block και εκτελεί επαναλήψεις μέχρι πριν φτάσει στην αρχή του επόμενου block.

**Πειραματικά αποτελέσματα – μετρήσεις**

Τα προγράμματα εκτελέστηκαν στο σύστημα που αναφέραμε στην αρχή και η χρονομέτρηση έγινε με τη συνάρτηση omp\_get\_wtime(). Συγκεκριμένα, αποθηκεύσαμε την αρχική χρονική στιγμή πριν την εκτέλεση των βρόγχων και την αφαιρέσαμε από την τελική χρονική στιγμή μετά την εκτέλεση όλων των βρόγχων.

Στην εκτέλεση των παράλληλων κομματιών χρησιμοποιήθηκαν 4 νήματα. Αυτό επιτεύχθηκε με την απενεργοποίηση του ευέλικτου πλήθους νημάτων με την εντολή

**omp\_set\_dynamic(0);**

και τον ορισμό του αριθμού νημάτων με την εντολή

**omp\_set\_num\_threads(4);**

Κάθε πείραμα εκτελέστηκε 4 φορές και υπολογίστηκαν οι μέσοι όροι. Προσοχή δόθηκε οι χρόνοι να μην συμπεριλαμβάνουν την ανάγνωση και εγγραφή των αρχείων. Τα αποτελέσματα δίνονται στον παρακάτω πίνακα (οι χρόνοι είναι σε δευτερόλεπτα):

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Τύπος προγράμματος | Serial | Submatrix size: 16 | Submatrix size: 256 | Submatrix size: 1024 |
| 1η εκτέλεση | 101.551301 | 29.960616 | 45.756525 | 42.014935 |
| 2η εκτέλεση | 101.990635 | 40.803010 | 46.137368 | 42.468503 |
| 3η εκτέλεση | 101.571880 | 41.494928 | 45.617336 | 41.787801 |
| 4η εκτέλεση | 102.046474 | 41.026631 | 47.541253 | 41.902812 |
| Μέσος όρος | 101.790073 | 38.321296 | 46.263121 | 42.043513 |

Με βάση τα παραπάνω αποτελέσματα προκύπτει το ακόλουθο γράφημα.

**Σχόλια**

Με βάση τα αποτελέσματα των δοκιμών παρατηρούμε την βελτίωση της απόδοσης κατά περίπου 50% σε όλα τα μεγέθη υποπινάκων. Η καλύτερη απόδοση επιτεύχθηκε επιλέγοντας μέγεθος υποπίνακα τα 16 στοιχεία. Αυτό που παρατηρήθηκε κατά την εκτέλεση των δοκιμών με το συγκεκριμένο μέγεθος είναι ότι το thread που ασχολούταν με την δημιουργία των tasks δεν πρόλαβε να εκτελέσει το ίδιο πολλά tasks, καθώς τα περισσότερα εκτελέστηκαν παράλληλα με την δημιουργία από τα υπόλοιπα threads. Αυτό μας δείχνει πολύ καλό διαμοιρασμό του φόρτου εργασίας και γι’ αυτό έχει και την καλύτερη απόδοση.

Στην 2η περίπτωση φαίνεται ότι οι υπολογισμοί του κάθε block ήταν αρκετοί έτσι ώστε το thread που έφτιαξε τα tasks να προλάβει να εκτελέσει και εκείνο αρκετά tasks. Από την άλλη όμως το πλήθος των task ήταν αρκετά μεγάλο με αποτέλεσμα να υπάρξει μεγάλο overhead.

Τέλος, στην 3η περίπτωση φτιάχτηκε πολύ μικρός αριθμός tasks, οπότε ο φόρτος δημιουργίας τους ήταν μικρός. Όμως παραμένει πιο αργή από την 1η περίπτωση καθώς οι υπολογισμοί που ζητάει το κάθε task είναι πολλοί και εάν υπάρξουν διακοπές στην εκτέλεση από ένα thread, τα υπόλοιπα δεν μπορούν να καλύψουν το κενό αυτό.