ΠΑΝΕΠΙΣΤΗΜΙΟ
ΔΥΤΙΚΗΣ ΜΑΚΕΔΟΝΙΑΣ

Συστήματα Παράλληλης και Κατανεμημένης Επεξεργασίας

Ενότητα: ΕΡΓΑΣΤΗΡΙΑΚΗ ΑΣΚΗΣΗ No:12

Δρ. Μηνάς Δασυγένης

mdasyg@ieee.org

Τμήμα Μηχανικών Πληροφορικής και Τηλεπικοινωνιών

Εργαστήριο Ψηφιακών Συστημάτων και Αρχιτεκτονικής Υπολογιστών

http://arch.icte.uowm.gr/mdasyg

Άδειες Χρήσης

- Το παρόν εκπαιδευτικό υλικό υπόκειται σε άδειες χρήσης Creative Commons.
- Για εκπαιδευτικό υλικό, όπως εικόνες, που υπόκειται σε άλλου τύπου άδειας χρήσης, η άδεια χρήσης αναφέρεται ρητώς.

Χρηματοδότηση

- Το παρόν εκπαιδευτικό υλικό έχει αναπτυχθεί στα πλαίσια του εκπαιδευτικού έργου του διδάσκοντα.
- Το έργο «**Ανοικτά Ψηφιακά Μαθήματα του Πανεπιστημίου Δυτικής Μακεδονίας**» έχει χρηματοδοτήσει μόνο τη αναδιαμόρφωση του εκπαιδευτικού υλικού.
- Το έργο υλοποιείται στο πλαίσιο του Επιχειρησιακού Προγράμματος «Εκπαίδευση και Δια Βίου Μάθηση» και συγχρηματοδοτείται από την Ευρωπαϊκή Ένωση (Ευρωπαϊκό Κοινωνικό Ταμείο) και από εθνικούς πόρους.

Με τη συγχρηματοδότηση της Ελλάδας και της Ευρωπαϊκής Ένωσης

Περιεχόμενα

1. Σκοπός της άσκησης

- Προγραμματισμός συστήματος με διαμοιραζόμενη μνήμη με τη χρήση του OpenMP #1.

2. Παραδοτέα

(A) 7 ερωτήσεις

(C) 6 ασκήσεις

3. Τι είναι το OpenMP;

Το OpenMP[1] *(Open Multi-Processing)* είναι ένα API για την δημιουργία πολυνηματικών προγραμμάτων κοινόχρηστης μνήμης σε C, C++ και Fortran. Υποστηρίζει τις περισσότερες αρχιτεκτονικές επεξεργαστών και λειτουργικά συστήματα (http://en.wikipedia.org/wiki/OpenMP).

Έχει παρόμοια χρησιμότητα με το POSIX Threads API[2], αλλά διαφορετική υλοποίηση.

Χρησιμοποιεί κι αυτό το μοντέλο **fork-join**. Το master thread δημιουργεί *(fork)* μία ομάδα από threads για να εκτελέσουν το κομμάτι του παραλληλοποιήσιμου κώδικα. Το κάθε thread παίρνει ένα ID, με το master να έχει πάντα το ID=0. Τα threads τερματίζουν *(join)* όταν τελειώσουν με την εκτέλεση του παραλληλοποιήσιμου κώδικα.

Τα πλεονεκτήματα του OpenMP είναι η ευκολία στην χρήση. Ο προγραμματιστής δεν χρειάζεται να δημιουργεί και να τερματίζει τα threads *(όπως συμβαίνει στο POSIX Threads API)*, παρά μόνο να ορίζει τα παραλληλοποιήσιμα κομμάτια και αν χρειάζεται τις παραμέτρους της παραλληλοποίησης. Ο διαμοιρασμός των δεδομένων στα threads γίνεται αυτόματα.

4. Πώς χρησιμοποιείται;

Για να χρησιμοποιήσουμε το OpenMP, πρέπει να το υλοποιεί ο compiler που χρησιμοποιούμε.

Ο GCC compiler από την έκδοση 4.2 και μετά υλοποιεί το OpenMP.

Για να ενεργοποιήσουμε την χρήση του OpenMP πρέπει κατά το compile του προγράμματός μας, να χρησιμοποιηθεί το switch `-fopenmp`, π.χ.

[1] http://openmp.org/wp/
[2] http://en.wikipedia.org/wiki/POSIX_Threads

```
gcc -fopenmp example.c -o example.out
```

Αν χρησιμοποιείτε άλλον compiler μπορείτε να ελέγξετε την σχετική λίστα[3] για το αν υποστηρίζεται.

Επίσης πρέπει να κάνουμε include στο πρόγραμμα το header file omp.h

```
#include <omp.h>
```

Η παραλληλοποίηση μέσα στο πρόγραμμα ελέγχεται με directives *(οδηγίες)* προς τον compiler, που ορίζονται με δηλώσεις της μορφής **#pragma**.

5. Directives

Ακολουθούν μερικές βασικές οδηγίες που μπορούμε να δώσουμε στον compiler. Οι υπόλοιπες μπορούν να βρεθούν στο επίσημο tutorial[4]. Επισκεφτείτε τη σελίδα για να πάρετε μια ιδέα του τι μπορείτε να κάνετε με το openmp.

5.1 Δημιουργία threads

```
#pragma omp parallel
// προαιρετικές επιλογές στην ίδια γραμμή μετά το
parallel:

// εκτελείται παράλληλα μόνο αν ικανοποιείται η συνθήκη
if (scalar_expression)

// λίστα ιδιωτικών μεταβλητών
private (list)

// λίστα κοινόχρηστων μεταβλητών
shared (list)
// ένας τρόπος για να ενώσουμε τα επιμέρους αποτελέσματα
// που έχουν υπολογίσει τα threads
// π.χ. reduction(+:sum)
reduction (operator: list)

// αριθμός threads που θα χρησιμοποιηθούν
num_threads (integer)
// ορισμούς του block που θα γίνει η παραλληλοποίηση
{
code
}
```

[3] http://openmp.org/wp/openmp-compilers/
[4] https://computing.llnl.gov/tutorials/openMP/

5.2 Δομές για καταμερισμό των εργασιών

for – οι επαναλήψεις μίας δομής επανάληψης for κατανέμονται στα threads.

```
#pragma omp for
// προαιρετικές επιλογές μετά τη λέξη for:
// πώς θα γίνει ο καταμερισμός των επαναλήψεων
schedule (type [,chunk])
...<for_loop>...
```

sections – "σπάμε" το πρόγραμμα σε κομμάτια, καθένα από τα οποία εκτελείται από ένα thread.

single – το συγκεκριμένο κομμάτι κώδικα θα εκτελεστεί σειριακά.

master – το συγκεκριμένο κομμάτι κώδικα θα εκτελεστεί μόνο από το master.

5.3 Δομές συγχρονισμού

critical – το συγκεκριμένο κομμάτι κώδικα θα εκτελείται μόνο από ένα thread την φορά

barrier – το κάθε thread περιμένει τα υπόλοιπα να φτάσουν σε αυτό το σημείο

atomic – επόμενη εντολή (κατά την οποία έχουμε εγγραφή στην μνήμη) ανανεώνεται ατομικά

flush – χρησιμοποιείται για να έχουμε μία συνεπή εικόνα της μνήμης

5.4 Συναρτήσεις

Ακολουθούν μερικές βασικές συναρτήσεις που ορίζονται στο OpenMP. Οι υπόλοιπες μπορούν να βρεθούν στο επίσημο tutorial[5].

```
void omp_set_num_threads(int num_threads)
```

ορίζει τον αριθμό των threads που θα χρησιμοποιηθούν

[5] https://computing.llnl.gov/tutorials/openMP/

```
int omp_get_num_threads(void)
```

επιστρέφει τον αριθμό των threads που έχουν δημιουργηθεί

```
int omp_get_thread_num(void)
```

επιστρέφει το ID του thread

5.5 Κοινόχρηστες και ιδιωτικές μεταβλητές

Εξ' ορισμού οι μεταβλητές ενός παράλληλου τμήματος κώδικα στο OpenMP, είναι κοινόχρηστες. Αυτό σημαίνει ότι όλα τα threads έχουν ταυτόχρονη πρόσβαση στην ίδια μεταβλητή.

Αν θέλουμε μπορούμε να ορίσουμε κάποιες μεταβλητές ως ιδιωτικές τροποποιώντας κατάλληλα το directive. Για παράδειγμα με το

```
#pragma omp parallel private(var1, var2)
```

ορίζουμε τις μεταβλητές **var1** και **var2** ως ιδιωτικές.

Έτσι το κάθε thread θα έχει ένα τοπικό αντίγραφο της μεταβλητής, το οποίο δεν θα έχει αρχική τιμή, και η τιμή που θα πάρει στο τέλος της εκτέλεσης του thread δεν αποθηκεύεται.

6. Hello world

Δημιουργήστε μέσα στον κατάλογο εργασίας σας έναν υποκατάλογο openmp στον οποίο θα τοποθετήσετε όλα τα προγράμματα που θα κατασκευάσετε στο εργαστήριο OpenMP.

Μπορείτε να τροποποιήσετε τον αριθμό των threads που δημιουργούνται αυτόματα κάθε φορά με το να θέσετε μια συγκεκριμένη μεταβλητή περιβάλλοντος, πριν την εκτέλεση του προγράμματος ως εξής:

```
setenv OMP_NUM_THREADS 4        (για το φλοιό CSH)

export OMP_NUM_THREADS=4        (για το φλοιό BASH)
```

Θα δημιουργήσουμε ένα πρόγραμμα στο οποίο κάθε thread θα εκτυπώνει το μήνυμα Hello world μαζί με το ID του. Το master thread θα εκτυπώνει το πλήθος των threads.

i. Δημιουργήστε το αρχείο **lab-openmp-hello.c**

ii. Επικολλήστε τον παρακάτω κώδικα.

```
#include <omp.h>
#include <stdio.h>

int main (int argc, char *argv[]) {
```

```
int tid, nthreads;

#pragma omp parallel
{
tid = omp_get_thread_num();
printf("Hello world from thread %d\n", tid);

if ( tid == 0 ) {
nthreads = omp_get_num_threads();
printf("There are %d threads\n",nthreads);
}
}

return 0;
}
```

iii. Αφού το κάνετε compile τρέξτε μερικές φορές το πρόγραμμα με ποικίλο αριθμό από threads. Θα παρατηρήσετε ότι υπάρχει κάποιο πρόβλημα.

iv. **(A1)** Γιατί δεν παίρνουμε τα αναμενόμενα αποτελέσματα;

Μήπως χρειάζεται κάποιες μεταβλητές να οριστούν ως ιδιωτικές;

v. **(A2)** Τροποποιήστε κατάλληλα το πρόγραμμα και επιβεβαιώστε το αποτέλεσμα. Ποιες αλλαγές κάνατε;

vi. Τροποποιήστε το πρόγραμμα (**lab-openmp-hello2.c**), ώστε να δημιουργηθούν 8 threads *(μην τροποποιήσετε τη μεταβλητή περιβάλλοντος)* και επιβεβαιώστε το αποτέλεσμα. **(A3)** Ποιες αλλαγές κάνατε;

7. Υπολογισμός π

Παρακάτω θα σας δοθεί ένα σειριακό πρόγραμμα υπολογισμού του π το οποίο καλείστε να κάνετε παράλληλο.

i. Δημιουργήστε το αρχείο **lab-openmp-pi_serial.c**

ii. Επικολλήστε τον παρακάτω κώδικα

```
#include <stdio.h>
#define STEPS 1000000

int main (int argc, char *argv[]) {
int i;
double step, x, sum, pi = 0;

step = 1.0 / STEPS;

for (i=0; i<STEPS; i++){
```

```
x = (i+0.5)*step;
sum = sum + 4.0/(1.0+x*x);
}

pi = step * sum;

printf("steps = %d\n pi = %f\n", STEPS, pi);
return 0;
}
```

vii. **(A4)** Κάντε compile και επιβεβαιώστε το αποτέλεσμα. Δώστε το screenshot εκτέλεσης.

iii. Δημιουργήστε αντίγραφο του αρχείου με όνομα **lab-openmpi-pi_parallel.c**

iv. **(C1)** Τοποθετήστε τα κατάλληλα `#pragma directives` ώστε να παραλληλοποιηθεί ο αλγόριθμος. Προσέξτε ποιο κομμάτι θα παραλληλοποιήσετε και τις τυχόν ιδιωτικές μεταβλητές που πρέπει να οριστούν, έτσι ώστε το αποτέλεσμα να είναι ίδιο με το αρχικό. Εκτυπώστε τον αριθμό των threads που χρησιμοποιήθηκαν.

8. Μέτρηση επιδόσεων

i. Εισάγετε στο σειριακό πρόγραμμα υπολογισμού του π τις απαραίτητες μεταβλητές και συναρτήσεις για την μέτρηση του χρόνου εκτέλεσης (**lab-openmp-pi_serialTime.c**). Για παράδειγμα:

```
double start, total;

start = omp_get_wtime();
//do something
total = omp_get_wtime() - start_time;
```

ii. Εισάγετε κατάλληλο μήνυμα για την εμφάνιση του χρόνου εκτέλεσης.

iii. **(C2)** Επαναλάβετε την ίδια διαδικασία για το παράλληλο πρόγραμμα (**lab-openmpi-pi_parallelTime.c**), προσέχοντας να βάλετε τις εντολές για την καταγραφή του χρόνου στα ίδια σημεία του κώδικα. Να παραδώσετε τα παραπάνω αρχεία.

iv. **(A5)** Συγκρίνετε τους χρόνους εκτέλεσης και υπολογίστε το speedup S. *(S=χρόνος σειριακού/χρόνος παράλληλου)*

Σημείωση: Χρησιμοποιήστε την συνάρτηση `double omp_get_wtime()` του OpenMP, τόσο στο σειριακό όσο και στο παράλληλο πρόγραμμα *(θα χρειαστεί να βάλετε το header omp.h και να κάνετε compile με το -fopenmp και στο σειριακό πρόγραμμα)*.

9. Υπολογισμός αθροίσματος πίνακα

i. **(A6)** Δημιουργήστε το σειριακό πρόγραμμα **lab-openmp-array_serial.c**, το οποίο υπολογίζει το άθροισμα των στοιχείων ενός πίνακα 100 θέσεων, για τον οποίον ισχύει `a[0]=0`, `a[1]=1`, κτλ.

ii. **(C3)** Δημιουργήστε το παράλληλο πρόγραμμα **lab-openmp-array_parallel1.c** που θα εκτελεί την ίδια λειτουργία, χρησιμοποιώντας μία global(καθολική) μεταβλητή για τον υπολογισμό του αθροίσματος. Θα πρέπει κάθε φορά να προσέχετε την πρόσβαση στην καθολική μεταβλητή για να μην δημιουργηθεί συνθήκη συναγωνισμού. Συγκεκριμένα, θα χρησιμοποιήσετε τη δομή συγχρονισμού `critical` για την εντολή που τροποποιείται η καθολική μεταβλητή.

iii. **(C4)** Δημιουργήστε το παράλληλο πρόγραμμα **lab-openmp-array_parallel2.c** που θα εκτελεί την ίδια λειτουργία, χρησιμοποιώντας την επιλογή reduction του OpenMP *(και δε θα χρειάζεται η δομή* `critical`*)*.

10. Πολλαπλασιασμός πινάκων

i. **(A7)** Δημιουργήστε το σειριακό πρόγραμμα **lab-openmp-mul_matrix_serial.c**, το οποίο υπολογίζει τον πίνακα που προκύπτει αν πολλαπλασιάσουμε δύο πίνακες 100x100, για τους οποίους ισχύει `a[i][j]=i*j`.

ii. **(C5)** Δημιουργήστε το παράλληλο πρόγραμμα **lab-openmp-mul_matrix_parallel.c** που θα εκτελεί την ίδια λειτουργία.

11. Ολοκλήρωση με τη μέθοδο του τραπεζίου

Σε προηγούμενο εργαστήριο μελετήσαμε τον υπολογισμό ολοκληρώματος με τη μέθοδο του τραπεζίου. Συγκεκριμένα, είχαμε δώσει τους τύπους:

$$\int_{x1}^{x2} f(x) = \sum_{i=\tau\rho\alpha\pi\acute{\varepsilon}\zeta\iota o}^{\acute{o}\lambda\alpha\ \tau\alpha\ \tau\rho\alpha\pi\acute{\varepsilon}\zeta\iota\alpha} \textbf{Εμβαδό τραπεζίου } i$$

$$A_i = \left(\frac{f(x_\alpha)+f(x_\beta)}{2}\right) \times \Delta x = \left(\frac{f(x_\alpha)+f(x_\beta)}{2}\right) \times (x_\beta - x_\alpha)$$

(C6) Να δημιουργήσετε το αρχείο **lab-openmp-integral_serial.c**, το οποίο θα υπολογίζει το ολοκλήρωμα από 1 έως 4 *(σωστό αποτέλεσμα:66)* για $f(x) = \int 3x^2 + 1$

Αυτό θα είναι σειριακό *(χωρίς threads/παραλληλία)*. Μόνο αν λειτουργεί αυτό σωστά, μπορείτε να συνεχίσετε στην τροποποίηση του για την παράλληλη υλοποίηση. Στη συνέχεια θα δημιουργήσετε την παράλληλη υλοποίηση *(με openmp)* lab-openmp-integral_parallel.c . Το παράλληλο πρόγραμμα να δέχεται ως παράμετρο της γραμμής εντολής τον αριθμό τραπεζίων *(να γίνονται έλεγχοι ορθότητας). Επίσης, να χρησιμοποιηθεί η συνάρτηση μέτρησης του χρόνου (`omp_get_wtime()`)* πριν και μετά το κυρίως μέρος, ώστε στο τέλος να εκτυπώνεται το μήνυμα σε μια γραμμή.

`Το αποτέλεσμα είναι XXXX για YYYY τραπέζια με ZZZZ νήματα σε BBBBB χρόνο.`

Να δημιουργήσετε στη συνέχεια ένα σενάριο φλοιού SH το οποίο θα εκτελεί το παραπάνω πρόγραμμα με παράμετρο 10, 100, 1000, 10000 τραπέζια και 2, 4, 8, 16, 32, 64, 128 νήματα *(όλοι οι πιθανοί συνδυασμοί)*.