

Παράλληλη Επεξεργασία - Προγραμματιστική Εργασία

Ονοματεπώνυμο: Άγγελος Καρδούτσος

A.M. : 1059372

1. Εισαγωγή

Στα πλαίσια της συγκεκριμένης εργασίας, ζητείται να αναπτυχθεί πρόγραμμα σε γλώσσα προγραμματισμού C, το οποίο παραλληλοποιεί τον υπολογισμό της τιμής μιας σταθεράς, η οποία μπορεί να υπολογιστεί με άπειρους όρους ή έναν αλγόριθμο. Για την παραλληλοποίηση του προγράμματος, θα αξιοποιήσουμε το OpenMP API.

2. Περιγραφή Προβλήματος

Ως πρόβλημα ορίστηκε ο υπολογισμός της τιμής της σταθεράς του Πυθαγόρα ή, διαφορετικά, της τετραγωνικής ρίζας του 2 ($\sqrt{2}$). Η σταθερά του Πυθαγόρα αποτελεί ένας αριθμός απείρων ψηφίων, του οποίου η τιμή είναι περίπου ίση με $\sqrt{2} = 1.41421356$. Επιπλέον, μπορούμε να απεικονίσουμε τη σταθερά αυτή με διάφορους τρόπους και σχέσεις, μεταξύ των οποίων και ως γινόμενο απείρων όρων. Η σχέση αυτή είναι:

$$\prod_{k=0}^{\infty} \left(1 + \frac{1}{4k+1}\right) \left(1 - \frac{1}{4k+3}\right) \quad (1)$$

3. Κώδικας

Με βάση την σχέση (1), θα αναπτύξουμε κώδικα, ο οποίος υπολογίζει το αποτέλεσμα του γινομένου n όρων. Αρχικά σχεδιάζουμε το πρόγραμμά μας, ώστε να υπολογίσει την τιμή που επιθυμούμε σειριακά:

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {

    long n=1e8;
    int k;
    long double sum1 = 1.0;

    for (k=0; k < n; k++) {

        sum1 *= (1 + ( 1.0 / (4*k+1) )) * (1 - ( 1.0 / (4*k+3) ));

    }
```

```
printf("Serial Result: %.12Lf\n", sum1);  
  
return 0;  
  
}
```

Ορίζουμε μια επαναληπτική διαδικασία, όπου σε κάθε επανάληψη, πολλαπλασιάζουμε το τρέχον γινόμενο που έχουμε υπολογίσει *sum1* με τον όρο που ορίζει η σχέση (1), όπου *k*: ο αριθμός της τρέχουσας επανάληψης. Το τελικό αποτέλεσμα, που προκύπτει για $n = 10^8$ όρους, έχει ακρίβεια μέχρι το 7^ο δεκαδικό ψηφίο (Εικόνα 1).

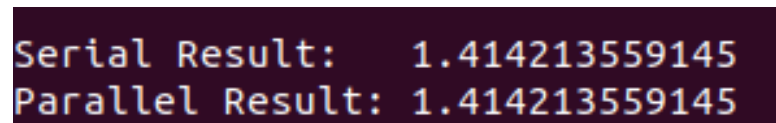
Στη συνέχεια, με βάση τον σειριακό κώδικα, αναπτύσσουμε τον κώδικα, ο οποίος υπολογίζει τη ζητούμενη τιμή παράλληλα:

```
#include <math.h>  
#include <omp.h>  
#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
  
    long n=1e8;  
    int k, thread_count = 4;  
    long double sum2 = 1.0;  
  
    #pragma omp parallel for num_threads(thread_count) \  
    default(none) \  
    shared(n) private(k) \  
    reduction(* : sum2)  
  
    for (k=0; k < n; k++) {  
  
        sum1 *= ( 1 + ( 1.0 / (4*k+1) ) ) * ( 1 - ( 1.0 / (4*k+3) ) );  
  
    }  
  
    printf("Serial Result: %.12Lf\n", sum1);  
  
    return 0;  
  
}
```

Αρχικά, φορτώσαμε το header file *omp.h* και ορίσαμε την τιμή της μεταβλητής *thread_count*, η οποία υποδεικνύει το πλήθος των νημάτων που θα δημιουργηθούν. Επιπλέον, πριν την επαναληπτική διαδικασία ορίζουμε μια οδηγία *pragma*, ειδική εντολή προς τον προεπεξεργαστή, η οποία θα εξασφαλίσει την παράλληλη εκτέλεση του βρόχου. Συγκεκριμένα, χρησιμοποιούμε την οδηγία *omp parallel for*, με την οποία το σύστημα μοιράζει τις επαναλήψεις μεταξύ των νημάτων. Θέτοντας τον όρο *num_threads*, ορίζουμε το πλήθος των νημάτων στα οποία θα εκτελεστεί ο βρόχος επανάληψης. Ο όρος *default(none)* μας επιτρέπει να ορίσουμε την εμβέλεια των μεταβλητών. Ειδικότερα, με τον όρο *shared*, θέτουμε την μεταβλητή *n* κοινή για όλα τα νήματα, ενώ, με τον όρο *private*,

θέτουμε την μεταβλητή k ως ιδιωτική για κάθε νήμα. Τέλος, ο όρος αναγωγής (*reduction*) συμβάλλει στο να μην έχουμε race condition. Χρησιμοποιείται σε περιπτώσεις όπου εφαρμόζεται μια πράξη (εδώ ο πολλαπλασιασμός) επαναληπτικά και για την οποία δεν μας ενδιαφέρει με ποια σειρά θα γίνουν οι πράξεις. Το τελευταίο είναι σημαντικό, καθώς κάθε νήμα που εφαρμόζει, κάθε φορά βρίσκεται σε ένα τυχαίο αριθμό επανάληψης.

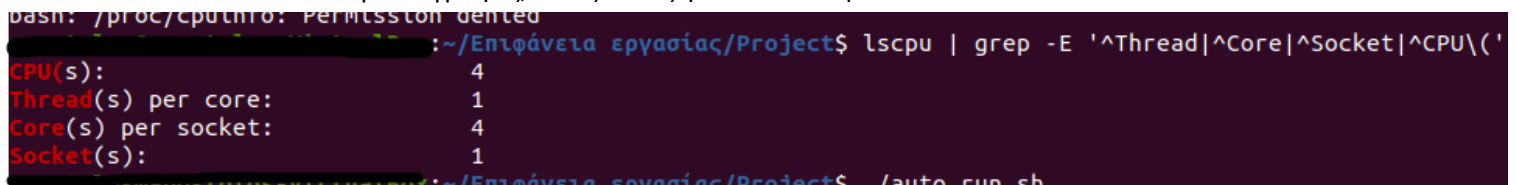
Εκτελώντας το πρόγραμμα, παρατηρούμε ότι, και σε αυτή την περίπτωση, για $n = 10^8$ όρους, το αποτέλεσμα παρουσιάζει ακρίβεια μέχρι το 7^ο δεκαδικό ψηφίο (Εικόνα 1).



```
Serial Result: 1.414213559145
Parallel Result: 1.414213559145
```

Εικόνα 1: Αποτελέσματα εκτέλεσης σειριακού και παράλληλου κώδικα για $n = 1E8$

Σημειώνεται ότι και το πρόγραμμα έτρεξε σε εικονική μηχανή με λειτουργικό σύστημα Ubuntu και προδιαγραφές, όπως αυτές φαίνονται στην εικόνα 2.



```
bash: /proc/cpuinfo: permission denied
~/Επιφάνεια εργασίας/Project$ lscpu | grep -E '^Thread|^Core|^Socket|^CPU\('
CPU(s): 4
Thread(s) per core: 1
Core(s) per socket: 4
Socket(s): 1
~/Επιφάνεια εργασίας/Project$
```

Εικόνα 2: Προδιαγραφές υπολογιστικού συστήματος

4. Αποτελέσματα και Σύγκριση

Όπως αναφέραμε και στην προηγούμενη, τα αποτελέσματα από την εκτέλεση του σειριακού και του παράλληλου προγράμματος παρουσιάζουν ίδια ακρίβεια. Σημαντικό είναι, επίσης, να συγκρίνουμε τους δύο κώδικες όσον αφορά το χρόνο εκτέλεσής τους. Επομένως, χρησιμοποιούμε την εντολή `omp_get_wtime()` για να υπολογίσουμε το χρόνο εκτέλεσης κάθε κώδικα. Τελικά, το πρόγραμμά μας έχει την εξής μορφή:

```

#include <math.h>
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void) {

    long n=1e8;
    int k, thread_count=4;
    long double sum1 = 1.0, sum2 = 1.0;

    double t0 = omp_get_wtime();

    for (k = 0; k < n; k++) {

        sum1 *= (1 + (1.0/(4*k+1)))*(1-(1.0/(4*k+3)));

    }

    double t1 = omp_get_wtime();


    double t2 = omp_get_wtime();

    #pragma omp parallel for num_threads(thread_count) \
    default(none) \
    shared(n) private(k) \
    reduction(* : sum2)

    for (k = 0; k < n; k++) {

        sum2 *= (1 + (1.0/(4*k+1)))*(1-(1.0/(4*k+3)));

    }

    double t3 = omp_get_wtime();


    printf("\nSerial Result: %.12Lf\n", sum1);
    printf("Parallel Result: %.12Lf\n", sum2);
    printf("Real Value:    %.12f\n\n", sqrt(2));
    printf("Serial Time: %.5f\n", t1-t0);
    printf("Parallel Time: %.5f\n\n", t3-t2);

    return 0;

}

```

Εκτελούμε το πρόγραμμα για διαφορετικό πλήθος όρων n , καθώς και για διαφορετικό αριθμό νημάτων $thread_count$, οπότε προκύπτουν τα εξής αποτελέσματα:

Πρόγραμμα	Όροι $n = 10^6$		Όροι $n = 10^7$		Όροι $n = 10^8$		Όροι $n = 10^9$	
	Ακρίβεια (δεκαδικό ψηφίο)	Χρόνος Εκτέλεσης (sec)	Ακρίβεια (δεκαδικό ψηφίο)	Χρόνος Εκτέλεσης (sec)	Ακρίβεια (δεκαδικό ψηφίο)	Χρόνος Εκτέλεσης (sec)	Ακρίβεια (δεκαδικό ψηφίο)	Χρόνος Εκτέλεσης (sec)
Σειριακό	6	0.00638	7	0.06392	7	0.60303	7	6.04897
Παράλληλο (Νήματα = 2)	6	0.00331	7	0.03288	7	0.32757	7	3.32082
Παράλληλο (Νήματα = 4)	6	0.00270	7	0.01649	7	0.16842	7	1.68387
Παράλληλο (Νήματα = 8)	6	0.00180	7	0.01670	7	0.18485	7	1.68511

Παρατηρούμε ότι όσο αυξάνονται το πλήθος των όρων που χρησιμοποιούμε, τόσο πιο εμφανής είναι η διαφορά στους χρόνους εκτέλεσης του σειριακού προγράμματος σε σύγκριση με τα παράλληλα προγράμματα.

Όταν χρησιμοποιούμε 2 και 4 νήματα, ο χρόνος εκτέλεσης είναι ταχύτερος κατά 2 και 4 φορές, αντίστοιχα, σε σχέση με την σειριακή εκτέλεση. Ωστόσο, όταν θέτουμε τον αριθμό των νημάτων ίσο με 8, ο χρόνος εκτέλεσης δεν βελτιώνεται σε σύγκριση με την εκτέλεση του προγράμματος για 4 νήματα. Αυτό οφείλεται στο γεγονός ότι το σύστημά μας διαθέτει 4 πυρήνες με 1 νήμα ο καθένας, όπως είδαμε και στις προδιαγραφές (Εικόνα 1).

Παρακάτω παρατίθενται τα αποτελέσματα από την εκτέλεση του προγράμματος για διαφορετικά n και *thread_count*:

```

-VirtualBox:~/Επιφάνεια εργασίας/Project$ ./a.out 1000000 1
Serial Result:                1.414213473985
Parallel Result (Threads=2):  1.414213473985
Parallel Result (Threads=4):  1.414213473985
Parallel Result (Threads=8):  1.414213473985
Real Value:                   1.414213562373

Serial Time:                   0.00638
Parallel Time (Threads=2):    0.00331
Parallel Time (Threads=4):    0.00270
Parallel Time (Threads=8):    0.00180

```

Εικόνα 3α: Αποτελέσματα για $n=1E6$

```

-VirtualBox:~/Επιφάνεια εργασίας/Project$ ./a.out 10000000 1
Serial Result:          1.414213553538
Parallel Result (Threads=2): 1.414213553538
Parallel Result (Threads=4): 1.414213553538
Parallel Result (Threads=8): 1.414213553538
Real Value:            1.414213562373

Serial Time:            0.06392
Parallel Time (Threads=2): 0.03288
Parallel Time (Threads=4): 0.01649
Parallel Time (Threads=8): 0.01670

```

Εικόνα 36: Αποτελέσματα για $n=1e7$

```

-VirtualBox:~/Επιφάνεια εργασίας/Project$ ./a.out 100000000 1
Serial Result:          1.414213559145
Parallel Result (Threads=2): 1.414213559145
Parallel Result (Threads=4): 1.414213559145
Parallel Result (Threads=8): 1.414213559145
Real Value:            1.414213562373

Serial Time:            0.60303
Parallel Time (Threads=2): 0.32757
Parallel Time (Threads=4): 0.16842
Parallel Time (Threads=8): 0.18485

```

Εικόνα 37: Αποτελέσματα για $n=1e8$

```

-VirtualBox:~/Επιφάνεια εργασίας/Project$ ./a.out 1000000000 1
Serial Result:          1.414213525574
Parallel Result (Threads=2): 1.414213525572
Parallel Result (Threads=4): 1.414213525571
Parallel Result (Threads=8): 1.414213525571
Real Value:            1.414213562373

Serial Time:            6.04897
Parallel Time (Threads=2): 3.32082
Parallel Time (Threads=4): 1.68387
Parallel Time (Threads=8): 1.68511

```

Εικόνα 38: Αποτελέσματα για $n=1e9$

5. Εκτέλεση Κώδικα

Στον ίδιο φάκελο με την παρούσα αναφορά, εμπεριέχονται δύο προγράμματα C (*project.c* και *project_extra.c*) και δύο εκτελέσιμα αρχεία.

Το αρχείο **project.c** διαθέτει το κύριο πρόγραμμα, όπως αυτό περιεγράφηκε στην ενότητα 3. Δηλαδή περιέχει τον σειριακό και παράλληλο κώδικα, τους οποίους και εκτελεί για $n=1e8$ και *thread_count*=4. Κάνοντας compile το αρχείο αυτό με την εντολή:

```
gcc -fopenmp project.c -lm -o project.out
```

προκύπτει το εκτελέσιμο **project.out**.

Το αρχείο **project_extra.c** αποτελεί μια τροποποίηση του αρχικού προγράμματος, καθώς δίνει στον χρήστη να θέσει δικές του τιμές σε συγκεκριμένες μεταβλητές. Κάνοντας το αρχείο αυτό compile με την εντολή:

```
gcc -fopenmp project_extra.c -lm -o project_extra.out
```

προκύπτει το εκτελέσιμο **project_extra.out**.

ο χρήστης μπορεί να εκτελέσει το συγκεκριμένο εκτελέσιμο ως εξής:

```
./project_extra.out [n: (Number || 0)] [mode: (0 || 1)] [thread_count: Number]
```

όπου:

- n: Ο χρήστης θέτει το πλήθος των όρων που θα χρησιμοποιηθούν στο γινόμενο για τον υπολογισμό της τετραγωνικής ρίζας του 2. Αν θέσει $n = 0$, τότε το πρόγραμμα θα εκτελεστεί για $n=1e8$.
- mode (Προαιρετικό): Αν ο χρήστης το θέσει ίσο με 0, τότε μπορεί να θέσει το πλήθος των νημάτων που θα δημιουργηθούν ορίζοντας και το τρίτο όρισμα thread_count. Αν δεν το ορίσει, τότε θα είναι thread_count = 4. Αν ο χρήστης το θέσει ίσο με 1, τότε ο κώδικας θα εκτελεστεί τρεις φορές παράλληλα με 2, 4 και 8 νήματα, αντίστοιχα.
- thread_count: Ο χρήστης θέτει το πλήθος των νημάτων. Ο ορισμός του ορίσματος έχει νόημα αν τεθεί mode = 0, διαφορετικά αγνοείται.

Σημείωση: Αν ο χρήστης επιθυμεί να ορίσει μόνο το mode (και το thread_count, αν το χρειαστεί) χωρίς να αλλάξει το πλήθος των όρων, τότε είναι απαραίτητο να θέσει πρώτα το $n = 0$. Παράδειγμα: ./project_extra.out 0 0 8

Επίσης, ο χρήστης μπορεί να εκτελέσει το πρόγραμμα χωρίς ορίσματα, όπου, σε αυτή την περίπτωση, οι default τιμές είναι όπως και του αρχείου project.c. Δηλαδή $n=1e8$, mode=0 και thread_count=4.