ECE445: ΠΑΡΑΛΛΗΛΟΣ ΚΑΙ ΔΙΚΤΥΑΚΟΣ ΥΠΟΛΟΓΙΣΜΟΣ

Χειμερινό Εξάμηνο 2021-2022

Εργασία 2

Ομάδα φοιτητών # 10

Ψαρού Αναστασία - 2860

Νίκος Λυμπερόπουλος - 2812

**Εισαγωγή**

To software και το hardware που χρησιμοποιήσαμε για την υλοποίηση και των τριών ασκήσεων

παρουσιάζεται παρακάτω:

* Operating System: Ubuntu 20.04.3 LTS
* Compiler: gcc 4:9.3.0-1ubuntu2
* Processor: Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz
* Ram: 7737MB

## Ερώτηση 1

Στην πρώτη άσκηση καλούμαστε να φτιάξουμε ένα πρόγραμμα που θα εκτυπώνει κάποια πράγματα με τη βοήθεια του openmp.

Για αρχή, ζητάμε από το χρήστη να μας πει τον αριθμό των νημάτων που θα χρησιμοποιήσουμε στο πρόγραμμα μας.

Στη συνέχεια, υπολογίζουμε τον αριθμός των επεξεργαστών του υπολογιστή που εκτελεί το πρόγραμμα και το μέγιστο πλήθος νημάτων του υπολογιστή. Αυτό γίνεται με τη βοήθεια των συναρτήσεων **omp\_get\_num\_procs()** και **omp\_get\_max\_threads()**.

Έπειτα, για την εύρεση των νημάτων που συμμετέχουν στην εκτέλεση του προγράμματος γίνεται χρήση της συνάρτησης **omp\_get\_num\_threads()** και για τον υπολογισμό του χρόνου της συνάρτησης **omp\_get\_wtime().**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 5 | 10 | 20 | 50 | 100 |
| 0.000223 | 0.002497 | 0.003369 | 0.019966 | 0.024736 |

## 

## Ερώτηση 2

Στη δεύτερη άσκηση καλούμαστε να υπολογίσουμε την προσέγγιση του π με τη χρήση του κανόνα του Simpson. Αυτό γίνεται στη συνάρτηση void simpson().

Αυτό που κάνουμε είναι ότι δημιουργούμε μια λούπα και σε κάθε μια επανάληψη υπολογίζεται το αντίστοιχο yi με τον παράγοντα που του αντιστοιχεί. Αν είναι το πρώτο ή τελευταίο y της προσέγγισης τότε είναι ίσος με 1. Αν το i από το yi είναι άρτιος αριθμός τότε ο παράγοντας είναι ίσος με 2 αλλιώς αν είναι περιττός ίσος με 4.

Ουσιαστικά κατά τη διάρκεια της λούπας κάθε νήμα εκτελεί μερικές επαναλήψεις, ώστε να φτάσουμε στις Ν. Ο αριθμός των νημάτων που χρησιμοποιούμε κάθε φορά είναι NUM\_OF\_THREADS και είναι #define στην αρχή του κώδικας. Το ίδιο συμβαίνει με το μέγεθος του CHUNK.

Για την μεταβλητή sum βάζουμε reduction(+:sum), διότι θέλουμε να αποφύγουμε το να πάνε να γράψουν 2 νήματα ταυτόχρονα στο sum και έτσι να χαθεί ένα τμήμα της πληροφορίας που χρειαζόμαστε. Με αυτήν την εντολή η πρόσθεση στο sum προστατεύεται, όντας κρίσιμο τμήμα.

Ο χρόνος υπολογίζεται και εδώ, όπως και στην ερώτηση 1, με τη χρήση της συνάρτησης **omp\_get\_wtime()**.

Στη συνέχεια ακολουθούν οι μετρήσεις που κάναμε. Αυτό που παρατηρούμε είναι ότι ο χρόνος για 1 νήμα είναι σχεδόν διπλάσιος από το χρόνο για 2 νήματα και αντίστοιχα ο χρόνος για 2 νήματα είναι σχεδόν διπλάσιος από το χρόνο για 4 νήματα. Μετά από τα 4 νήματα δεν παρατηρούμε τόσο μεγάλες διαφορές χρόνων. Αυτό οφείλεται στο ότι ο μέγιστος αριθμός νημάτων που μπορούμε να έχουμε είναι 4, όπως είδαμε στην άσκηση 4.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Thread=1 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 4.058385 | 4.058199 | 4.056409 | 4.160508 | 4.087582 | 4.053467 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads=2 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 2.283766 | 2.318929 | 2.436573 | 2.563927 | 2.363690 | 2.402638 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads=4 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 1.422180 | 1.398982 | 1.471799 | 1.509070 | 1.306220 | 1.263176 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads=8 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 1.343840 | 1.356544 | 1.436310 | 1.532763 | 1.395771 | 1.327838 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads=16 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 1.414691 | 1.404812 | 1.371367 | 1.493986 | 1.343273 | 1.322221 |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Threads=32 | static | | | dynamic | | |
| Chunks | 10 | 100 | 1000 | 10 | 100 | 1000 |
| time | 1.375902 | 1.371203 | 1.377838 | 1.497893 | 1.342843 | 1.325494 |

## Ερώτηση 3

Στην τρίτη άσκηση καλούμαστε να βρούμε τη λύση ενός συστήματος Ax=b με τη βοήθεια της μεθόδου Jacobi.

Για την υλοποίηση αυτής της άσκησης σπάμε σε δύο κομμάτια και υπολογίζουμε ξεχωριστά το δεξί τμήμα του κανόνα του Jacobi, δηλαδή τα D-1\*b και (L+U)\*xold. To D-1\*b είναι σταθερό οπότε το υπολογιζούμε στην αρχή του προγράμματος μας. Έτσι απλοποιούμε κάπως το πρόβλημα.

Ουσιαστικά σε κάθε επανάληψη της λούπας που γίνεται Μ φορές υπολογίζεται μια καινούρια προσέγγιση του χ.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| threads | 1 | 2 | 4 | 8 | 16 | 32 |
| time | 0.154289 | 0.140257 | 0.211638 | 0.168081 | 0.157259 | 0.169878 |

Γενικά δεν παρατηρούμε ιδιαίτερη διαφορά στους χρόνους. Αυτό μπορεί να συμβαίνει, διότι πιθανώς η καθυστέρηση από τη δημιουργία και ανάθεση “εργασιών” στα νήματα είναι περίπου ίση με τον επιπλέον χρόνο που χρειάζεται 1 ή και 2 νήματα να τρέξουν τον κώδικα, λόγω έλλειψης παραλληλίας.

# ΒΙΒΛΙΟΓΡΑΦΙΑ

1. Γραμματή Πάντζιου, Βασίλειος Μάμαλης, Αλέξανδρος Τομαράς, «Εισαγωγή στον Παράλληλο Υπολογισμό».
2. wikipedia.org

# Παράρτημα

## Κώδικας Ερώτησης 1

Ακολουθεί ο κώδικας για το ερώτημα 1. Όνομα αρχείου “ask1.c”.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ergasia 2 – Askhsh 1 – 13/01/2022

\* Psarou Anastasia - 2860

\* Lymperopoulos Nikos - 2812

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <float.h>

void function(void){

int num\_of\_threads;

double start, end, exec\_time;

printf("Enter number of threads to be used: ");

scanf("%d", &num\_of\_threads);

printf("\n");

start = omp\_get\_wtime();

printf("Number of processors: %d\n", omp\_get\_num\_procs());

printf("Max number of threads: %d\n", omp\_get\_max\_threads());

omp\_set\_num\_threads(num\_of\_threads);

#pragma omp parallel

{

#pragma omp master

{

printf("Num of threads: %d\n", omp\_get\_num\_threads());

printf("\n");

printf("Hello. I am the master thread. \n");

printf("\n");

}

#pragma omp barrier

//edw tha mpoun ola ta nimata ektos apo to master

if(omp\_get\_thread\_num() != 0){

printf("Hi there! I am thread %d\n", omp\_get\_thread\_num());

}

}

end = omp\_get\_wtime();

exec\_time = end - start;

printf("Execution time is: %lf\n", exec\_time);

}

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

function();

return(0);

}

## Κώδικας Ερώτησης 2

Ακολουθεί ο κώδικας για το ερώτημα 2. Όνομα αρχείου “ask2.c”.

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ergasia 2 – Askhsh 1 – 13/01/2022

\* Psarou Anastasia - 2860

\* Lymperopoulos Nikos - 2812

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <float.h>

#include <math.h>

#define NUM\_OF\_THREADS 32

#define N 100000000

#define CHUNK 1000

void simpson(){

double sum = 0;

int i;

double start, end, total\_time;

int num\_flops = 0;

start = omp\_get\_wtime();

#pragma omp parallel for reduction(+:sum) schedule(dynamic, CHUNK)

for(i = 0; i <= N; i++){

if(i != 0 || i != (N-1)){

if(i % 2 == 0){

sum += 4 \* 4 / (1 + pow(i, 2) / pow(N, 2));

num\_flops += 3;

continue;

}

else{

sum += 4 \* 2 / (1 + pow(i, 2) / pow(N, 2));

num\_flops += 3;

continue;

}

}

else{

sum += 4 / (1 + pow(i, 2) / pow(N, 2));

num\_flops += 3;

}

}

sum = sum/(3\*N);

end = omp\_get\_wtime();

total\_time = end - start;

printf("pi is approximately %.20lf , computations time = %lf, number of threads = %d, FLOPS = %lf, chunk = %d , scheduling = static\n", sum, total\_time, NUM\_OF\_THREADS, num\_flops/total\_time, CHUNK);

return;

}

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

omp\_set\_num\_threads(NUM\_OF\_THREADS);

simpson();

return(0);

}

## Κώδικας Ερώτησης 3

Ακολουθεί ο κώδικας για το ερώτημα 3. Όνομα Αρχείου «filename.c»

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Ergasia 2 – Askhsh 3 – October 28, 1940

\* Eponymo Onoma - AEM

\* Eponymo Onoma - AEM

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <float.h>

#include <math.h>

#define N 1000

#define M 20

#define NUM\_OF\_THREADS 32

int main(int argc, char \*argv[])

{

double \*\*A;

double \*\*D\_ant;

double \*\*L\_plus\_U;

double \*b;

int i, j, k, l;

double \*x\_old;

double \*x\_new;

double \*D\_b;

double \*\*D\_L\_plus\_U;

double \*help\_array;

double residual, difference;

double start, end, total\_time;

//memory allocation for the arrays

A = (double \*\*)malloc(N \* sizeof(double \*));

L\_plus\_U = (double \*\*)malloc(N \* sizeof(double \*));

D\_ant = (double \*\*)malloc(N \* sizeof(double \*));

D\_L\_plus\_U = (double \*\*)malloc(N \* sizeof(double \*));

x\_old = (double \*)malloc(N \* sizeof(double));

x\_new = (double \*)malloc(N \* sizeof(double));

b = (double \*)malloc(N \* sizeof(double));

D\_b = (double \*)malloc(N \* sizeof(double));

help\_array = (double \*)malloc(N \* sizeof(double));

for (i = 0; i < N; i++)

{

A[i] = (double \*)malloc(N \* sizeof(double));

L\_plus\_U[i] = (double \*)malloc(N \* sizeof(double));

D\_ant[i] = (double \*)malloc(N \* sizeof(double));

D\_L\_plus\_U[i] = (double \*)malloc(N \* sizeof(double));

}

//initialization of the arrays

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

D\_L\_plus\_U[i][j] = 0;

if (i == j)

{

D\_ant[i][j] = 0.5;

L\_plus\_U[i][j] = 0;

}

else if ((j == i + 1) || (j == i - 1))

{

L\_plus\_U[i][j] = -1;

D\_ant[i][j] = 0;

}

else

{

D\_ant[i][j] = 0;

L\_plus\_U[i][j] = 0;

}

}

}

for (i = 0; i < N; i++)

{

b[i] = 0;

help\_array[i]=0;

x\_old[i] = 0;

x\_new[i] = 0;

}

b[N - 1] = N + 1;

omp\_set\_num\_threads(NUM\_OF\_THREADS);

//calculation of D^(-1)\*D\_b

#pragma omp parallel for

for (j = 0; j < N; j++)

{

D\_b[j] = b[j] \* D\_ant[j][N - 1];

}

//calculation of D^(-1)\*(L+U)

for (i = 0; i < N; i++)

{

for (j = 0; j < N; j++)

{

for (k = 0; k < N; k++)

{

D\_L\_plus\_U[i][j] += ((D\_ant[i][k]) \* L\_plus\_U[k][j]);

}

}

}

double sum = 0;

start = omp\_get\_wtime();

for (int counter = 0; counter < M; counter++)

{

#pragma omp parallel for reduction(+:sum)

for (i = 0; i < N; i++)

{

sum = 0;

for (j = 0; j < N; j++)

{

sum += D\_L\_plus\_U[i][j] \* x\_old[j];

}

x\_new[i] = D\_b[i] - sum;

}

// calculate residual, difference

#pragma omp for private(l)

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

for (l=0; l < N; l++)

help\_array[k] += A[k][l]\*x\_new[l];

help\_array[k] = pow(b[k]- help\_array[k], 2);

#pragma omp critical (residual)

{

residual += help\_array[k];

}

help\_array[k] = pow(x\_old[k]-x\_new[k], 2);

#pragma omp critical (difference)

{

difference += help\_array[k];

}

}

#pragma omp parallel for

for (i = 0; i < N; i++)

{

x\_old[i] = x\_new[i];

x\_new[i] = 0;

}

printf("iter = %d, residual = %lf, difference = %lf\n", counter, residual, difference);

}

end = omp\_get\_wtime();

total\_time = end - start;

printf("Total time is %lf\n", total\_time);

//deallocation of memory used

for (i = 0; i < N; i++)

{

free(A[i]);

free(L\_plus\_U[i]);

free(D\_ant[i]);

free(D\_L\_plus\_U[i]);

}

free(A);

free(D\_L\_plus\_U);

free(L\_plus\_U);

free(D\_ant);

free(x\_old);

free(x\_new);

free(b);

free(D\_b);

return (0);

}