**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**НАЦІОНАЛЬНОМУ УНІВЕРСИТЕТІ “ЛЬВІВСЬКА ПОЛІТЕХНІКА”**



**Лабораторна робота №2**

з дисципліни

**«Технології розподілених систем та паралельних обчислень»**

**Виконав:**

студент групи КН-308

Матвіїв Микола

**Викладач:**

Мочурад Л.І.

Львів – 2020р.

**Лабораторна робота №2**

**Тема:** розподіл роботи між потоками засобами OpenMP

**Мета:** вивчити розподіл роботи між потоками засобами OpenMP інавчитися їх застосовувати.

**Варіант №13**

**Завдання**

****

Створити програму яка повинна реалізувати наступні дії:

1. Створити квадратні матриці А та В розміром n\*n, елементи якихзаповнюютьсядовільними числами, n задає кількість рядків і кількістьпотоків, які виконуватимутьпаралельну область програми. Змінні можутьзадаватися в програмному коді абовводитися з клавіатури.
2. У кожній матриці окремо обчислити паралельним способом завданнявідповідно досвого варіанту. Розподіл ітерацій між потоками виконати задопомогою директиви **for** звикористанням опції **schedule**. Для кожної зматриць використати різні значенняпараметру **type** та розміру блоку –**chunk.** Результати обробки записати в масиви С та D.
3. Вивести результати обробки матриць паралельним способом, вказавшипри виводірозподіл ітерацій по потоках для різних значень опції **schedule**.Порівняти результати,отримані при різних значеннях параметру **type** тарозміру блоку – **chunk.**

**Програмний код**

#include <iostream>

#include <omp.h>

#include <time.h>

#include <math.h>

#include <cmath>

#include <cstdlib>

#define dynamic\_chunk 6

#define guided\_chunk 4

#define n 1000

using namespace std;

double par, seq;

int A[n][n] = { 0 };

int B[n][n] = { 0 };

int C[n] = { 0 };

int Cs[n] = { 0 };

int D[n] = { 0 };

int Ds[n] = { 0 };

void print\_matrix(int arr[n][n]) {

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

printf("%d ", arr[i][j]);

}

cout << "\n";

}

cout << "\n";

}

void print\_vector(int arr[n]) {

for (int i = 0; i < n; i++) {

printf("%d ", arr[i]);

}

cout << "\n\n";

}

void parallel\_method\_guided(int Mx[n][n], int Vc[n], int thr\_num, int chk)

{

int tmp;

double t1, t2;

omp\_set\_dynamic(0);

omp\_set\_num\_threads(thr\_num);

#pragma omp parallel

{

#pragma omp for schedule(guided, chk)

for (int i = 0; i < n; i++) {

tmp = 0;

for (int j = 0; j < n; j++) {

if (Mx[i][j] < 0)

tmp++;

}

}

}

t1 = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp for schedule(guided, chk)

for (int i = 0; i < n; i++) {

tmp = 0;

for (int j = 0; j < n; j++) {

tmp += abs(pow(Mx[i][j],2));

}

Vc[i] = tmp;

}

}

t2 = omp\_get\_wtime();

par = t2 - t1;

printf("Parallel (%dc/guided/%d) time = %.15f\n", thr\_num, chk,par);

printf("Parallel Speedup = %.3f\n", seq / par);

printf("Parallel Efficiency = %.3f\n\n", seq / (par \* thr\_num));

}

void parallel\_method\_dynamic(int Mx[n][n], int Vc[n], int thr\_num, int chk)

{

int tmp;

double t1, t2;

omp\_set\_dynamic(0);

omp\_set\_num\_threads(thr\_num);

#pragma omp parallel

{

#pragma omp for schedule(dynamic, chk)

for (int i = 0; i < n; i++) {

tmp = 0;

for (int j = 0; j < n; j++) {

tmp += abs(pow(Mx[i][j],2));

}

//Vc[i] = tmp;

}

}

t1 = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp for schedule(dynamic, chk)

for (int i = 0; i < n; i++) {

tmp = 0;

for (int j = 0; j < n; j++) {

tmp += abs(pow(Mx[i][j],2));

}

Vc[i] = tmp;

}

}

t2 = omp\_get\_wtime();

par = t2 - t1;

printf("Parallel (%dc/DYNAMIC/%d) time = %.15f\n", thr\_num, chk, par);

printf("Parallel Speedup = %.3f\n", seq / par);

printf("Parallel Efficiency = %.3f\n\n", seq / (par \* thr\_num));

}

void serial\_method(int Mx[n][n], int Vc[n])

{

double ts1, ts2;

int tmp;

ts1 = omp\_get\_wtime();

for (int i = 0; i < n; i++) {

tmp = 0;

for (int j = 0; j < n; j++) {

tmp += abs(pow(Mx[i][j],2));

}

Vc[i] = tmp;

}

ts2 = omp\_get\_wtime();

seq = ts2 - ts1;

printf("serial\_method time = %.15f\n\n", seq);

}

int main()

{

int min = -100, max = 100, output;

srand(time(NULL));

for (int i = 0; i < n; i++) {

for (int j = 0; j < n; j++) {

A[i][j] = min + (rand() % static\_cast<int>(max - min + 1));

B[i][j] = min + (rand() % static\_cast<int>(max - min + 1));

}

}

serial\_method(A, Cs);

parallel\_method\_guided(A, C, 4, guided\_chunk);

parallel\_method\_dynamic(A, C, 4, dynamic\_chunk);

cout << endl;

serial\_method(B, Ds);

parallel\_method\_guided(B, D, 4, guided\_chunk);

parallel\_method\_dynamic(B, D, 4, dynamic\_chunk);

cin.ignore();

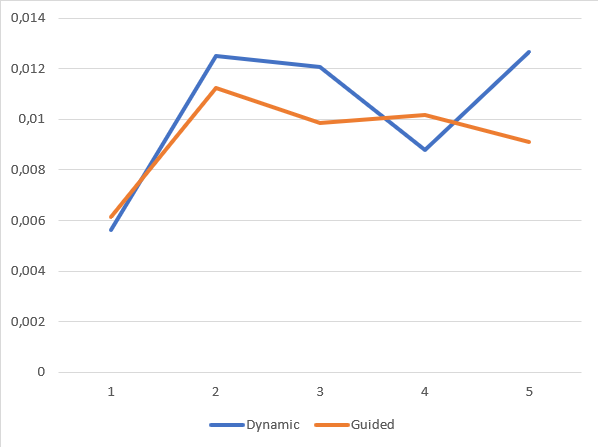
return 0;

}

**Результат**

Результати виконання алгоритмів з різної кількості потоків при n =1000:

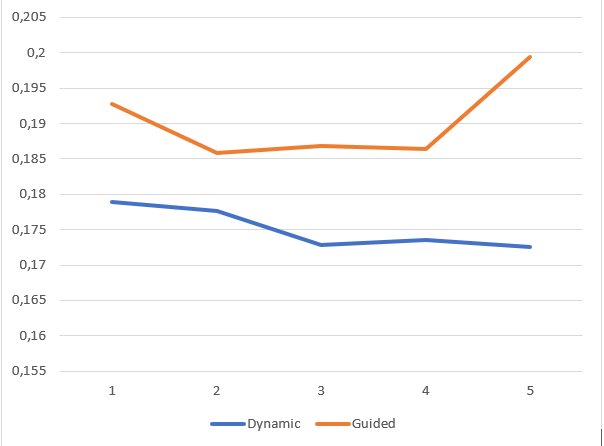
|  |  |  |
| --- | --- | --- |
| Chunk | Dynamic | Guided |
| 1 | 0.00564 | 0.00614 |
| 2 | 0.01251 | 0.01126 |
| 4 | 0.01209 | 0.00987 |
| 6 | 0.00878 | 0.01018 |
| 8 | 0.01268 | 0.00912 |



Графік 1. Залежність часу виконання алгоритмів n =1000 між Dynamic та Guided

Результати виконання алгоритмів з різної кількості потоків при n =5000:

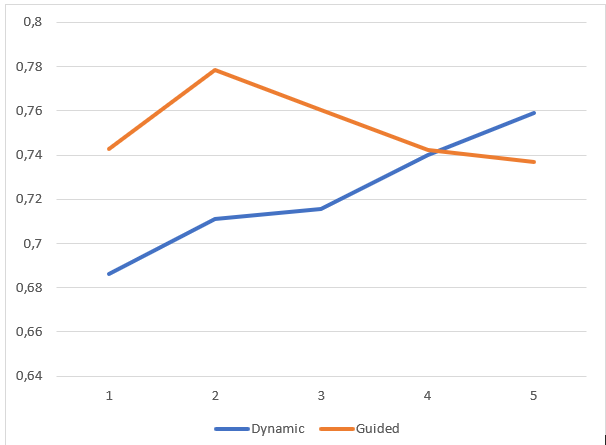
|  |  |  |
| --- | --- | --- |
| Chunk | Dynamic | Guided |
| 1 | 0.17895 | 0.19275 |
| 2 | 0.17763 | 0.18581 |
| 4 | 0.17282 | 0.18678 |
| 6 | 0.17348 | 0.18636 |
| 8 | 0.17256 | 0.19937 |



Графік 2. Залежність часу виконання алгоритмів при n =5000 між Dynamic та Guided

Результати виконання алгоритмів з різної кількості потоків при n =10000:

|  |  |  |
| --- | --- | --- |
| Chunk | Dynamic | Guided |
| 1 | 0.68632 | 0.74264 |
| 2 | 0.71126 | 0.77834 |
| 4 | 0.71573 | 0.76056 |
| 6 | 0.74012 | 0.74235 |
| 8 | 0.75891 | 0.73684 |



Графік 3. Залежність часу виконання алгоритмів при n =10000 між Dynamic та Guided

Дослідивши та проаналізувавши всі отримані результати та графіки, можемо зробити висновок, що чим більша кількість чанків, тим швидшим буде тип dynamic. Ми можемо побачити, що при малій кількості чанків тип guided є швидшим за dynamic, але коли кількість чанків стає більшою, то швидкість dynamic зростає і перевищує швидкість guided.

**Висновок**

Виконуючи дану лабораторну роботу, я ознайомився з технологією OpenMP та набув практичних навичок в її використанні.