**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**

**ІМЕНІ ТАРАСА ШЕВЧЕНКА**

**ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**

**КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ**

**Лабораторна робота №2**

з дисципліни «Високопродуктивні обчислення»

Тема роботи: «Паралельне програмування за допомогою засобів OpenMP»

Виконала студентка

групи КН-31

Промоцька А.А.

Перевірив:

проф. Циганок В.В.

**Київ – 2023**

**Мета:** Освоїти реалізацію паралельних обчислень за допомогою засобів OpenMP та виконати порівняльне оцінювання часу виконання програм при різній реалізації з та без використання OpenMP та програм, реалізованих як результат лабораторної роботи №1.

**Хід роботи**

Для реалізації програми було застосовано принципи ООП, мовою програмування було обрано С++, програму реалізовано на комп’ютері з 4 логічними процесорами.

**Завдання 1**

Для 1 завдання було використано програмний код з минулої лабораторної роботи:

Метод ітеративного наївного алгоритму:

void FirstTask::IterativeNaiveAlgorithm()

{

MatrixMultiplication test(n, m, k, maxValue);

test.fillMatrixRandomly();

fout << "Matrix 1:\n";

test.writeInFile(test.matrix1, fout);

fout << "Matrix 2:\n";

test.writeInFile(test.matrix2, fout);

test.multiply();

fout << "Result matrix:\n";

test.writeInFile(test.resultMatrix, fout);

}

Вимірювання часу виконання алгоритму

int FirstTask::measureTimeForAlgo() {

Stopwatch stopwatch;

stopwatch.start();

IterativeNaiveAlgorithm();

auto elapsedTime = stopwatch.stop();

return elapsedTime.count();

}

**Завдання 2**

Для того, щоб реалізувати 2 завдання, було використано бібліотеку <omp.h> і створено метод для визначення доступної версії стандарту OpenMP, який використовує вбудовані макроси OPENMP:

void SecondTask::checkOpenMPSupport()

{

#ifdef \_OPENMP

std::cout << "OpenMP is supported" << std::endl;

#if \_OPENMP >= 202107 // OpenMP 5.1

std::cout << "OpenMP version 5.1 or higher is available" << std::endl;

#elif \_OPENMP >= 201811 // OpenMP 5.0

std::cout << "OpenMP version 5.0 is available" << std::endl;

#elif \_OPENMP >= 201511 // OpenMP 4.5

std::cout << "OpenMP version 4.5 is available" << std::endl;

#elif \_OPENMP >= 201307 // OpenMP 4.0

std::cout << "OpenMP version 4.0 is available" << std::endl;

#elif \_OPENMP >= 201111 // OpenMP 3.1

std::cout << "OpenMP version 3.1 is available" << std::endl;

#elif \_OPENMP >= 200807 // OpenMP 3.0

std::cout << "OpenMP version 3.0 is available" << std::endl;

#elif \_OPENMP >= 200505 // OpenMP 2.5

std::cout << "OpenMP version 2.5 is available" << std::endl;

#elif \_OPENMP >= 199810 // OpenMP 4.5

std::cout << "OpenMP version 2.0 is available" << std::endl;

#else

std::cout << "OpenMP version is unknown" << std::endl;

#endif

#else

std::cout << "OpenMP is not supported" << std::endl;

#endif

}

За допомогою директиви omp\_get\_wtime() від OpenMP було вирахувано час, затрачений на виконання послідовного алгоритму у модифікованому методі measureTimeForAlgo():

int SecondTask::measureTimeForAlgo() {

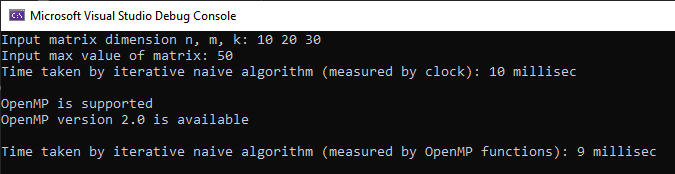
double startTime = omp\_get\_wtime();

IterativeNaiveAlgorithm();

double endTime = omp\_get\_wtime();

double elapsedTimeMilliseconds = (endTime - startTime) \* 1000.0;

return static\_cast<int>(elapsedTimeMilliseconds);

}

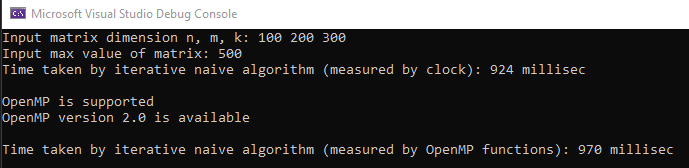
Рисунок 1 – Порівняння часу роботи послідовних алгоритмів

Рисунок 2 – Порівняння результатів 1 і 2 завдання

Можна зробити висновок, що при наявності директив і функцій від OpenMP на великих розмірностях матриці програма виконується повільніше при роботі послідовного алгоритму, тоді як при менших розмірностях навпаки – трохи швидше, але цей виграш у швидкості несе мінімальне значення на практиці.

**Завдання 3**

Аналізуючи створену послідовну програму, можна дійти висновку, що розпалалелюванню доцільно піддати 2 блоки у програмі: зчитування матриць з файлу та сам ітеративний алгоритм множення матриць.

**1 блок:**

Оскільки у файлі міститься 2 вхідні матриці, то доцільно розпаралелити метод зчитування цих матриць на дві секції, які паралельно будуть зчитувати вхідні матриці і записувати їх у поля класу. Це було реалізовано за допомогою конструктів omp parallel sections і omp section:

void ThirdTask::ReadInputMatrices() {

int i, numOfInputMatricies = 2;

omp\_set\_num\_threads(numOfInputMatricies);

#pragma omp parallel sections

{

#pragma omp section

{

GotoLine(fin1, 2);

ReadMatrix(fin1, matrix1);

}

#pragma omp section

{

GotoLine(fin2, n + 3);

ReadMatrix(fin2, matrix2);

}

}

}

**2 блок**

Для того, щоб реалізувати паралельний алгоритм множення, було застосовано директиву omp for із приватними змінними j, l. За замовченням кількість потоків дорівнює кількості рядків першої вхідної матриці:

void ThirdTask::ParallelNaiveAlgorithm()

{

ReadInputMatrices();

int i, j, l;

omp\_set\_num\_threads(n);

#pragma omp parallel

{

#pragma omp for private(j, l)

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

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

for (l = 0; l < m; ++l) {

resultMatrix[i][j] += matrix1[i][l] \* matrix2[l][j];

}

}

}

#pragma omp barrier

}

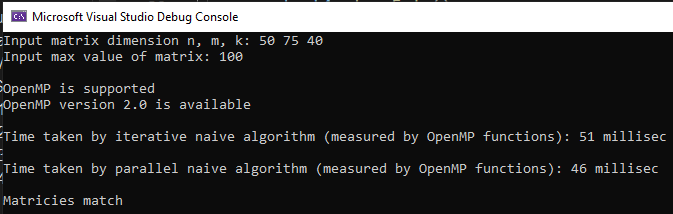
}

Рисунок 3 – Результат роботи програми

Робимо висновок, що для матриць розмірами 50х75 і 75х40 на часі виграє паралельний алгоритм з кількістю створених потоків n = 50.

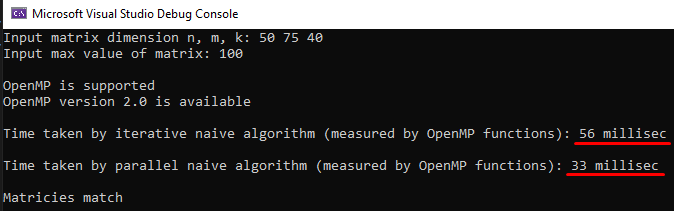
Спробуємо змінити кількість потоків на n/2:

Рисунок 4 – Результат роботи програми

Можна побачити, що зі зменшенням кількості потоків час роботи паралельного алгоритму значно покращився.

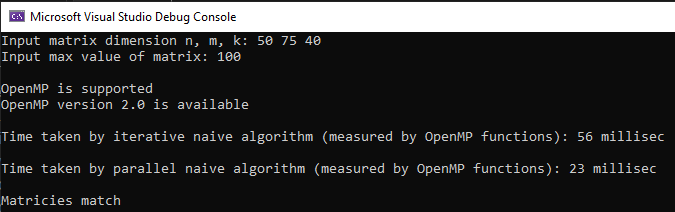
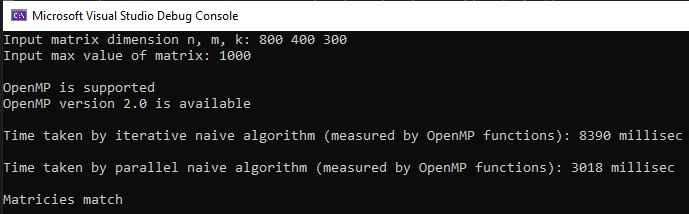
Спробуємо ще зменшити кількість потоків n/5 для того ж розміру вхідних матриць:

Рисунок 5 - Результат роботи програми

Час роботи покращився ще на 10 мілісекунд.

Протестуємо залежність між кількістю потоків потоків на великих розмірах матриць 800х400 і 400х300:

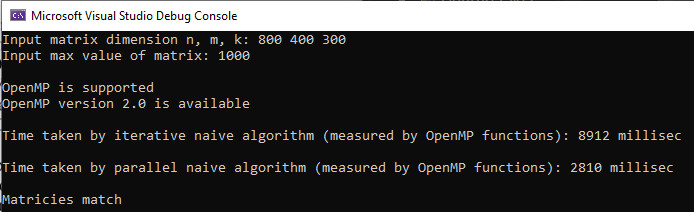
Рисунок 6 – Результат роботи програми при n-потоків

Рисунок 7 - Результат роботи програми при (n/100)-потоків

Загалом можна побачити, що тенденція продовжується: при меншій виділеній кількості потоків паралельний алгоритм працює швидше, але ненабагато. Краща швидкість може бути спричинена меншим часом очікування завершення роботи всіх потоків.

**Завдання 4**

Для того, щоб розділити зовнішній цикл, було застосовано директиви omp parallel sections і omp section:

void FourthTask::multiplyMatrixFromStartPoint(int start, int end)

{

for (int i = start; i < end; ++i) {

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

for (int l = 0; l < m; ++l) {

resultMatrix[i][j] += matrix1[i][l] \* matrix2[l][j];

}

}

}

}

void FourthTask::ParallelNaiveAlgorithm()

{

ReadInputMatrices();

int numOfSections = 2;

int step = n / numOfSections;

vector<int> sectionsStart(numOfSections);

sectionsStart[0] = 0;

for (int i = 1; i < numOfSections; ++i) {

sectionsStart[i] += step;

}

#pragma omp parallel sections

{

#pragma omp section

{

multiplyMatrixFromStartPoint(sectionsStart[0], sectionsStart[1]);

}

#pragma omp section

{

multiplyMatrixFromStartPoint(sectionsStart[1], n);

}

}

}

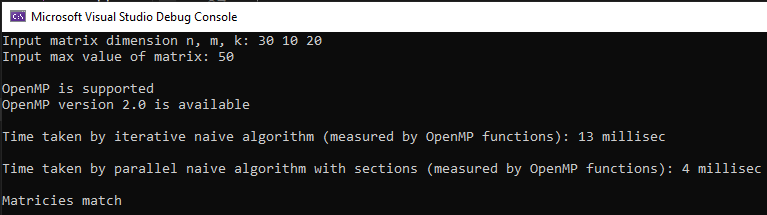
Дослідимо ефективність програми на малих і великих розмірах матриці при двох секціях і двох потоків відповідно:

Рисунок 8 – Результат роботи програми при матрицях 30х10 і 10х20

Бачимо, що паралельний алгоритм з 2 потоками швидше за послідовний приблизно у 3 рази.

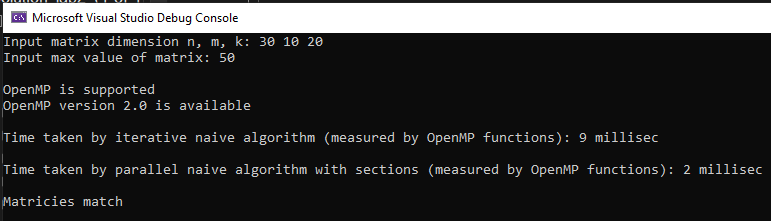
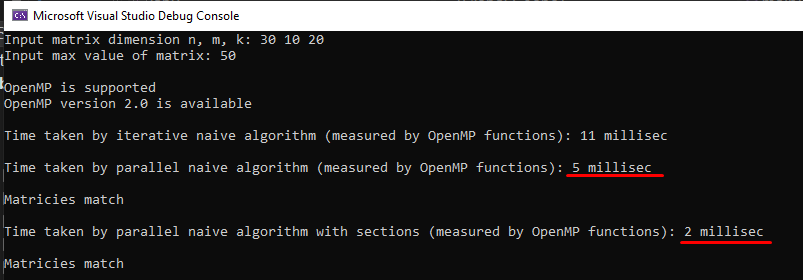
Зробимо 5 секцій:

Рисунок 9 – Результат роботи програми при матрицях 30х10 і 10х20

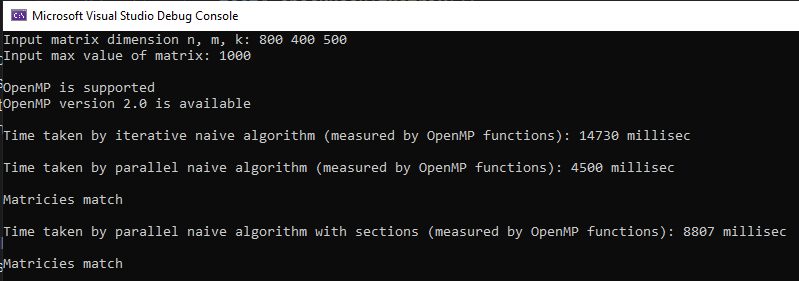
Бачимо, що паралельний алгоритм з 5 потоками швидше за послідовний приблизно у 4 рази.

Порівняємо 5-секційний блок з паралельним блоком, який так само містить 5 потоків:

Рисунок 10 – Результат роботи програми при матрицях 30х10 і 10х20

Робимо висновок, що алгоритм із секціями швидше у 2,5 рази за 5-потоковий.

Протестуємо також наш алгоритм на великих розмірах матриць 800х400 400 х500:

Рисунок 11 – Результат роботи програми

Можна зробити висновок, що алгоритм із секціями працює гірше на великих розмірностях вхідних матриць майже у 2 рази.

На мою думку, це зумовлено тим, що 5 секцій-потоків обчислюють велику кількість ітерацій, коли алгоритм з 5 потоками швидше перемикається між обчисленнями, коли звільняються.

**Висновок**

На цій лабораторній роботі ми освоїли реалізацію паралельних обчислень за допомогою засобів OpenMP та виконали порівняльне оцінювання часу виконання програм при різній реалізації.