

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.

Перевірів:

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 C++, 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:

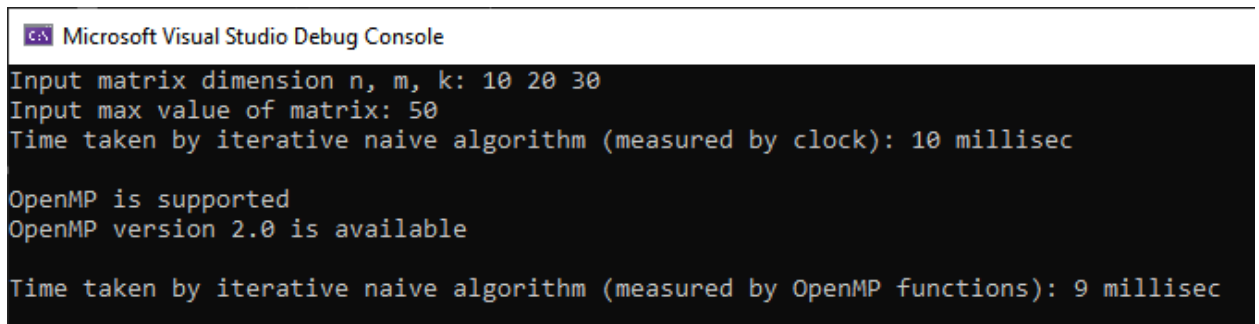
```

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
}

```

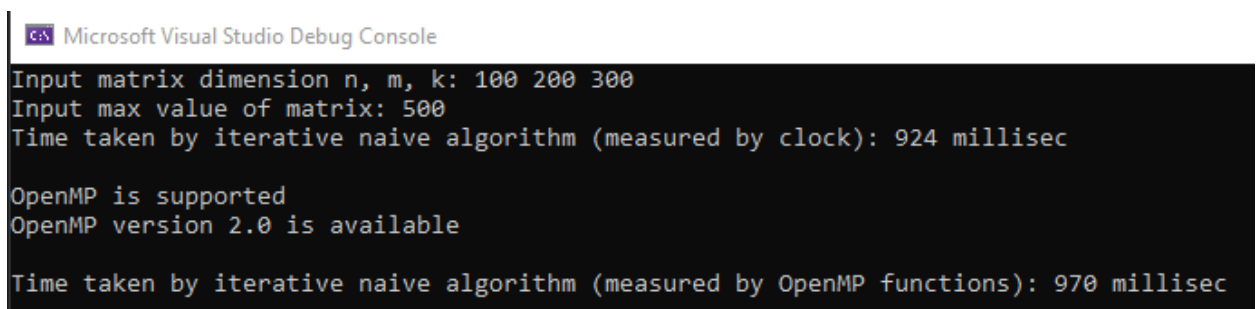
За допомогою директиви `omp_get_wtime()` від OpenMP було вираховано час, затрачений на виконання послідовного алгоритму у модифікованому методі `measureTimeForAlgo()`:

```
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);  
}
```



```
Microsoft Visual Studio Debug Console  
Input matrix dimension n, m, k: 10 20 30  
Input max value of matrix: 50  
Time taken by iterative naive algorithm (measured by clock): 10 millisec  
  
OpenMP is supported  
OpenMP version 2.0 is available  
  
Time taken by iterative naive algorithm (measured by OpenMP functions): 9 millisec
```

Рисунок 1 – Порівняння часу роботи послідовних алгоритмів



```
Microsoft Visual Studio Debug Console  
Input matrix dimension n, m, k: 100 200 300  
Input max value of matrix: 500  
Time taken by iterative naive algorithm (measured by clock): 924 millisec  
  
OpenMP is supported  
OpenMP version 2.0 is available  
  
Time taken by iterative naive algorithm (measured by OpenMP functions): 970 millisec
```

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 вииграш у швидкості несе мінімальне значення на практиці.

Завдання 3

Аналізуючи створену послідовну програму, можна дійти висновку, що розпаралелюванню 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, numOfInputMatrices = 2;
    omp_set_num_threads(numOfInputMatrices);

#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:

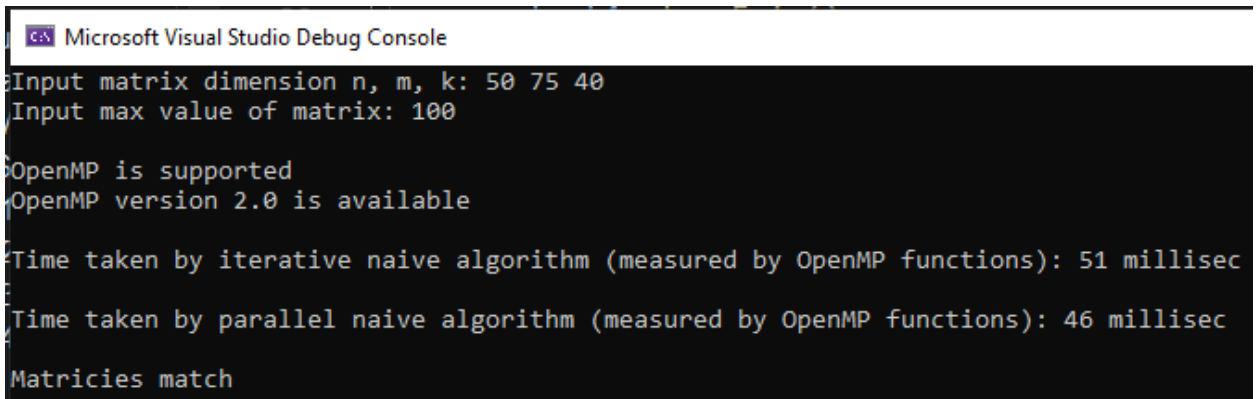
```
void ThirdTask::ParallelNaiveAlgorithm()
{
    ReadInputMatrices();
```

```

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
}
}

```



```

Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 50 75 40
Input max value of matrix: 100

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 51 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 46 millisec

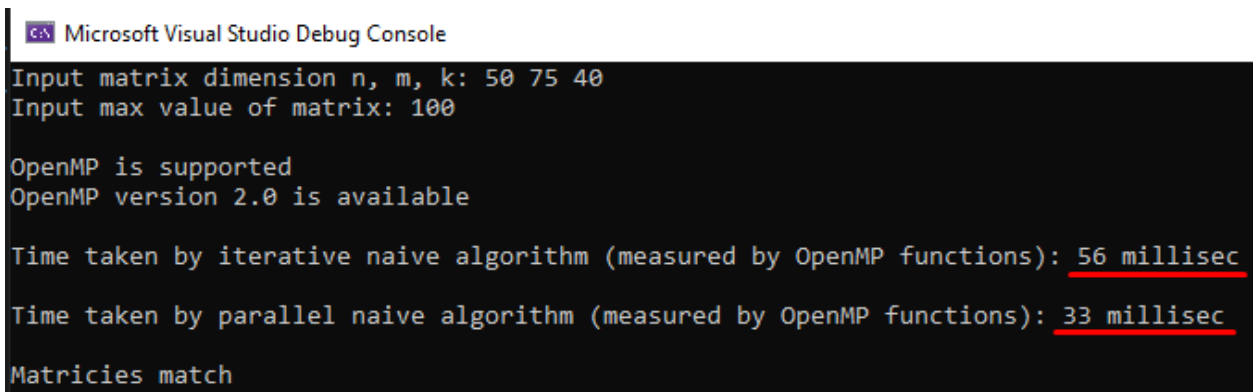
Matrices match

```

Рисунок 3 – Результат роботи програми

Робимо висновок, що для матриць розмірами 50x75 і 75x40 на часі виграв паралельний алгоритм з кількістю створених потоків $n = 50$.

Спробуємо змінити кількість потоків на $n/2$:



```

Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 50 75 40
Input max value of matrix: 100

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 56 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 33 millisec

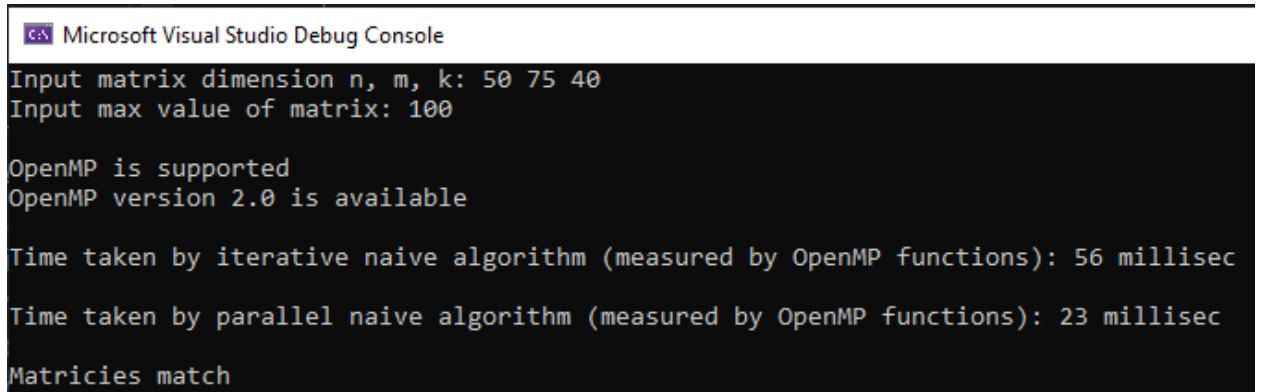
Matrices match

```

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:



```
Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 50 75 40
Input max value of matrix: 100

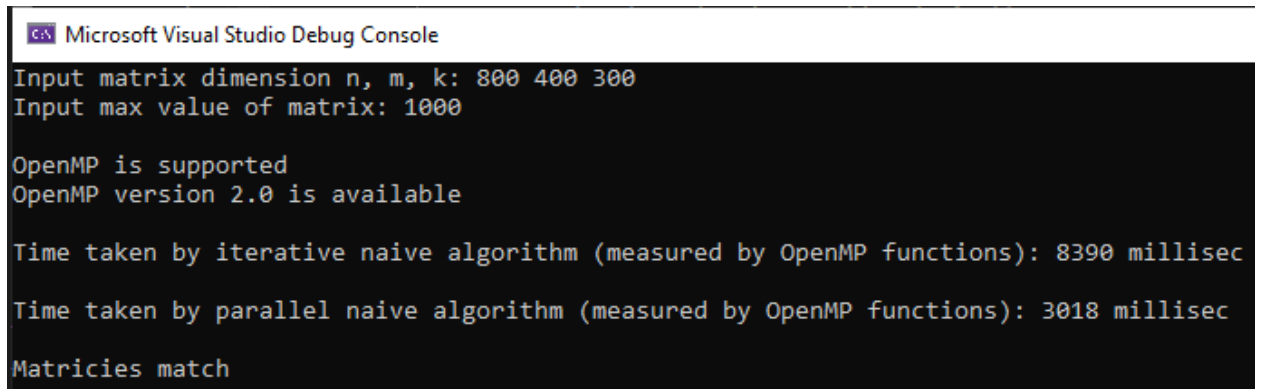
OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 56 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 23 millisec
Matricies match
```

Рисунок 5 - Результат роботи програми

Час роботи покращився ще на 10 мілісекунд.

Протестуємо залежність між кількістю потоків потоків на великих розмірах матриць 800x400 і 400x300:

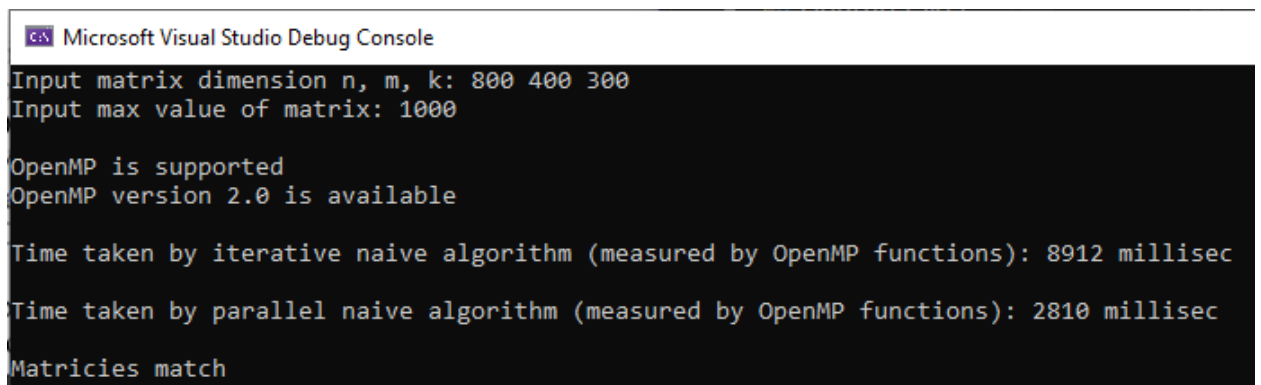


```
Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 800 400 300
Input max value of matrix: 1000

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 8390 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 3018 millisec
Matricies match
```

Рисунок 6 – Результат роботи програми при n-потоків



```
Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 800 400 300
Input max value of matrix: 1000

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 8912 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 2810 millisec
Matricies match
```

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);

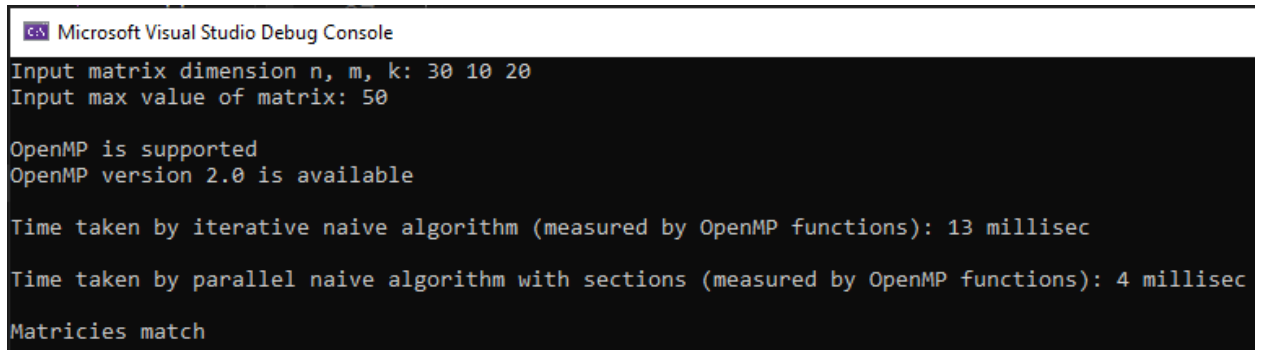
    sectionsStart[0] = 0;
    for (int i = 1; i < numOfSections; ++i) {
        sectionsStart[i] += step;
    }

    #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:

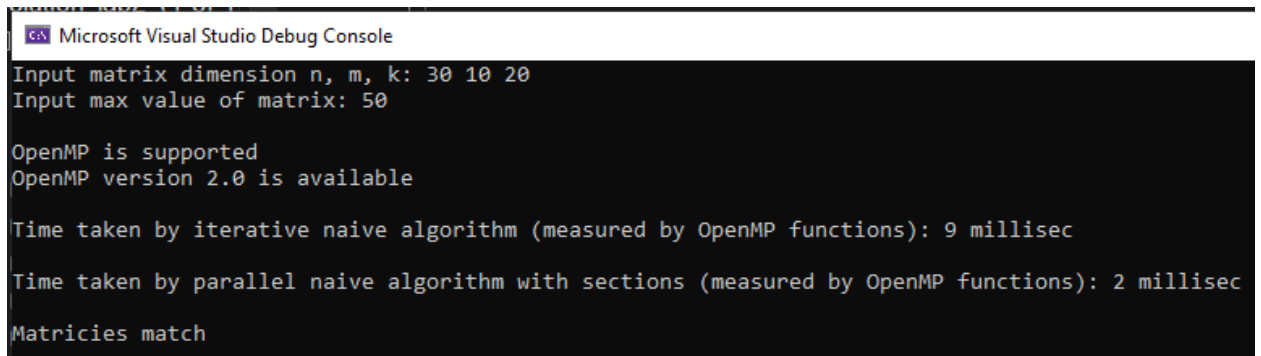


```
Microsoft Visual Studio Debug Console  
Input matrix dimension n, m, k: 30 10 20  
Input max value of matrix: 50  
  
OpenMP is supported  
OpenMP version 2.0 is available  
  
Time taken by iterative naive algorithm (measured by OpenMP functions): 13 millisec  
Time taken by parallel naive algorithm with sections (measured by OpenMP functions): 4 millisec  
Matrices match
```

Рисунок 8 – Результат роботи програми при матрицях 30x10 і 10x20

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

Зробимо 5 секцій:



```
Microsoft Visual Studio Debug Console  
Input matrix dimension n, m, k: 30 10 20  
Input max value of matrix: 50  
  
OpenMP is supported  
OpenMP version 2.0 is available  
  
Time taken by iterative naive algorithm (measured by OpenMP functions): 9 millisec  
Time taken by parallel naive algorithm with sections (measured by OpenMP functions): 2 millisec  
Matrices match
```

Рисунок 9 – Результат роботи програми при матрицях 30x10 і 10x20

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:

```
Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 30 10 20
Input max value of matrix: 50

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 11 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 5 millisec
Matrices match

Time taken by parallel naive algorithm with sections (measured by OpenMP functions): 2 millisec
Matrices match
```

Рисунок 10 – Результат роботи програми при матрицях 30x10 і 10x20

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

Протестуємо також наш алгоритм на великих розмірах матриць 800x400 400x500:

```
Microsoft Visual Studio Debug Console
Input matrix dimension n, m, k: 800 400 500
Input max value of matrix: 1000

OpenMP is supported
OpenMP version 2.0 is available

Time taken by iterative naive algorithm (measured by OpenMP functions): 14730 millisec
Time taken by parallel naive algorithm (measured by OpenMP functions): 4500 millisec
Matrices match

Time taken by parallel naive algorithm with sections (measured by OpenMP functions): 8807 millisec
Matrices match
```

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.