

**КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ**  
**ІМЕНІ ТАРАСА ШЕВЧЕНКА**  
**ФАКУЛЬТЕТ ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ**  
**КАФЕДРА ІНТЕЛЕКТУАЛЬНИХ ТЕХНОЛОГІЙ**

**Лабораторна робота №1**

з дисципліни «Високопродуктивні обчислення»

Тема роботи: «Паралельне програмування за допомогою стандартних засобів  
сучасних операційних систем»

Виконала студентка

групи КН-31

Примоцька А.А.

Перевірів:

проф. Циганок В.В.

**Київ – 2023**

**Мета:** Освоїти реалізацію паралельних обчислень за допомогою стандартних засобів багатопроесорності та багатопоточності сучасних ОС та виконати порівняльне оцінювання часу виконання програми з послідовним обчисленням.

### Хід роботи

Для реалізації програми було застосовано принципи ООП, мовою програмування було обрано C++.

#### Завдання 1

Генерація вхідних елементів матриць здійснювалася випадково за допомогою бібліотеки <random> відповідно до введеної межі користувачем:

```
void MatrixMultiplication::fillMatrixRandomly()
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            matrix1[i][j] = rand() % max;
        }
    }

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            matrix2[i][j] = rand() % max;
        }
    }
}
```

Час роботи алгоритму вимірювався за допомогою бібліотеки <chrono> і включав у себе заповнення матриць, запис згенерованих матриць у файл, множення матриць і запис результуючої матриці у файл.

Методи класу Stopwatch:

```
void Stopwatch::start() {
    time = high_resolution_clock().now();
    isRunning = true;
}

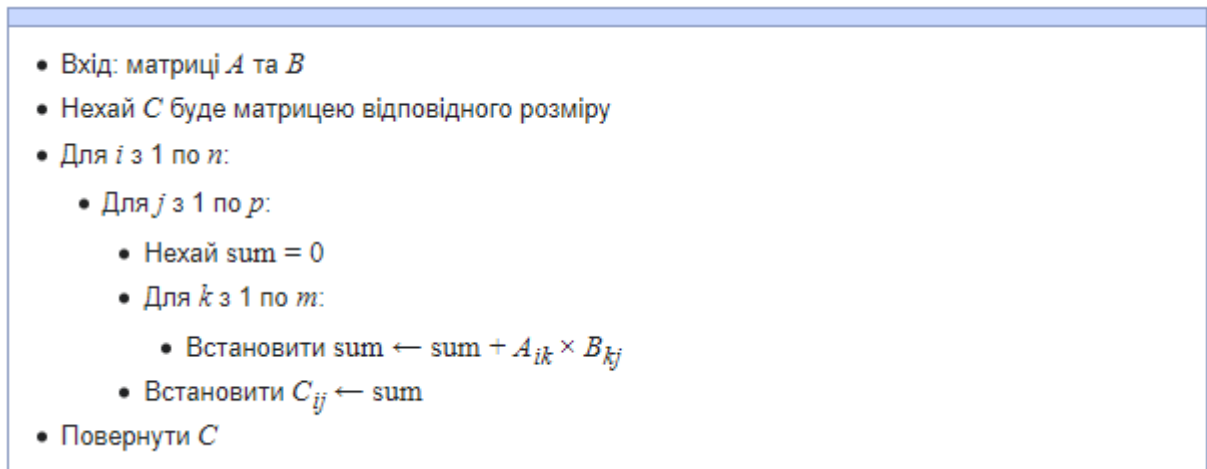
milliseconds Stopwatch::stop() {
    if (isRunning) {
```

```

        isRunning = false;
        return duration_cast<milliseconds>(high_resolution_clock().now() -
time);
    }
    else {
        return duration_cast<milliseconds>(time.time_since_epoch());
    }
}

```

Для виконання операції множення двох матриць було обрано найвний ітеративний алгоритм множення матриць:



```

• Вхід: матриці  $A$  та  $B$ 
• Нехай  $C$  буде матрицею відповідного розміру
• Для  $i$  з 1 по  $n$ :
    • Для  $j$  з 1 по  $p$ :
        • Нехай  $sum = 0$ 
        • Для  $k$  з 1 по  $m$ :
            • Встановити  $sum \leftarrow sum + A_{ik} \times B_{kj}$ 
            • Встановити  $C_{ij} \leftarrow sum$ 
        • Повернути  $C$ 

```

Рисунок 1 – Псевдокод алгоритму

Реалізація алгоритму в класі MatrixMultiplication:

```

void MatrixMultiplication::multiply() {
    for (int i = 0; i < this->n; ++i) {
        for (int j = 0; j < this->k; ++j) {
            for (int l = 0; l < this->m; ++l) {
                resultMatrix[i][j] += matrix1[i][l] * matrix2[l][j];
            }
        }
    }
}

```

Методи основного класу для множення матриць:

```

MatrixMultiplication::MatrixMultiplication(int n, int m, int k, int max)
{
    this->n = n;
    this->m = m;
    this->k = k;
    this->max = max;
}

```

```

matrix1.resize(n);
for (int i = 0; i < n; ++i) {
    matrix1[i].resize(m);
}

matrix2.resize(m);
for (int i = 0; i < m; ++i) {
    matrix2[i].resize(k);
}

resultMatrix.resize(n);
for (int i = 0; i < n; ++i)
{
    resultMatrix[i].resize(k);
}

}

void MatrixMultiplication::fillMatrixRandomly()
{
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < m; ++j) {
            matrix1[i][j] = rand() % max;
        }
    }

    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < k; ++j) {
            matrix2[i][j] = rand() % max;
        }
    }
}

void MatrixMultiplication::multiply() {
    for (int i = 0; i < this->n; ++i) {
        for (int j = 0; j < this->k; ++j) {
            for (int l = 0; l < this->m; ++l) {
                resultMatrix[i][j] += matrix1[i][l] * matrix2[l][j];
            }
        }
    }
}

void MatrixMultiplication::print(vector <vector<int>> matrix) {
    for (int i = 0; i < matrix.size(); ++i) {

```

```

        for (int j = 0; j < matrix[i].size(); ++j) {
            cout << matrix[i][j] << " ";
        }
        cout << endl;
    }
}

```

```

long MatrixMultiplication::measureTimeForMultiplication() {
    Stopwatch stopwatch;

    stopwatch.start();

    this->multiply();

    auto elapsedTime = stopwatch.stop();

    return elapsedTime.count();
}

```

```

void MatrixMultiplication::writeInFile(vector <vector<int>> matrix, ofstream&
fout) {
    for (int i = 0; i < matrix.size(); ++i) {
        for (int j = 0; j < matrix[i].size(); ++j) {
            fout << matrix[i][j] << " ";
        }
        fout << "\n";
    }
}

```

Методи класу для першого завдання:

```

FirstTask::FirstTask(int n, int m, int k, int maxValue)
{
    this->n = n;
    this->m = m;
    this->k = k;
    this->maxValue = maxValue;
}

```

```

bool FirstTask::OpenFileForWriting()
{
    fout.open(path);

    if (!fout.is_open())
    {
        cout << "Cannot open the file for writing" << endl;
    }
}

```

```

        fout.close();
        return false;
    }
}

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);
}

int FirstTask::measureTimeForAlgo() {
    Stopwatch stopwatch;

    stopwatch.start();

    IterativeNaiveAlgorithm();

    auto elapsedTime = stopwatch.stop();

    return elapsedTime.count();
}

void FirstTask::closeFile()
{
    fout.close();
}

```

## Завдання 2

Для реалізації паралельного обчислювання був створений інший клас SecondTask.

Оскільки в завдання входить зчитування матриць вхідних з файлу, то їх було прийнято рішення так само зчитати паралельно.

Для цього застосовувався метод переходу на потрібний рядок у файлі:

```
ifstream& SecondTask::GotoLine(ifstream& file, unsigned int num) {  
    file.seekg(ios::beg);  
    for (int i = 0; i < (num - 1); ++i) {  
        file.ignore(numeric_limits<streamsize>::max(), '\n');  
    }  
  
    return file;  
}
```

Також метод зчитування матриці:

```
void SecondTask::ReadMatrix(ifstream& file, vector<vector<int>>& matrix)  
{  
    int element = 0;  
  
    for (int i = 0; i < matrix.size(); ++i) {  
        for (int j = 0; j < matrix[0].size(); ++j) {  
            file >> element;  
            matrix[i][j] = element;  
        }  
    }  
}
```

І метод, в якому створювалися додаткові два потоки, один для зчитування першої матриці, другий – для зчитування другої:

```
void SecondTask::ReadInputMatrices() {  
    vector<thread> threads(2);  
  
    for (int i = 0; i < threads.size(); ++i) {  
        switch (i) {  
            case 0:  
                GotoLine(fin1, 2);  
                threads[i] = thread(&SecondTask::ReadMatrix, this, ref(fin1),  
ref(matrix1));  
                break;  
            case 1:  
                GotoLine(fin2, n + 3);  
                threads[i] = thread(&SecondTask::ReadMatrix, this, ref(fin2),  
ref(matrix2));  
        }  
    }  
}
```

```

    }
}

for (auto& thread : threads) {
    thread.join();
}
}

```

Для реалізації алгоритму паралельного обчислення множення двох матриць застосовувалося два методи. Перший метод створював n-кількість потоків, що дорівнює n-кількості рядків першої матриці:

```

void SecondTask::ParallelNaiveAlgorithm()
{
    ReadInputMatrices();

    vector<thread> threads(n);

    for (int i = 0; i < n; ++i) {
        threads[i] = thread([this, i]()
        {
            resultMatrix[i] = multiplyRowToMatrix(matrix1[i]);
        });
    }

    for (auto& thread : threads) {
        thread.join();
    }
}

```

Кожен потік викликає другий метод `multiplyRowToMatrix`, який множить вхідний рядок на другу матрицю. Результат роботи методу присвоюється відповідному рядку результуючої матриці:

```

vector<int> SecondTask::multiplyRowToMatrix(vector<int> &row) {
    vector<int> result(k);

    for (int i = 0; i < k; ++i) {
        for (int j = 0; j < m; ++j) {
            result[i] += row[j] * matrix2[j][i];
        }
    }
}

```



```

    return result;
}

```

Для перевірки коректності поелементно отриманої матриці з записаною матрицею у файлі був реалізований метод:

```

bool SecondTask::checkResultMatrix() {
    int matrixElementFromFile;

    GotoLine(fin1, (n + m + 4));

    for (int i = 0; i < resultMatrix.size(); ++i) {
        for (int j = 0; j < resultMatrix[0].size(); ++j) {
            fin1 >> matrixElementFromFile;
            if (resultMatrix[i][j] != matrixElementFromFile)
            {
                cout << "Dot product doesn't match at a[" << i << "][" << j << "];
                return false;
            }
        }
    }

    return true;
}

```

До обчислення швидкості роботи алгоритму входить читання матриць файлу та робота паралельного алгоритму множення матриць, перевірка коректності обчислень – ні.

Порівняння часу обчислень при різній розмірності вхідних матриць наведено у таблиці:

Розмірність матриці			Ітеративний алгоритм (мілісек)	Паралельний алгоритм (мілісек)
n	m	k		
2	3	4	2	7
3	3	3	0	7
10	15	20	3	15
20	22	35	12	29
50	75	100	107	95
100	125	125	286	162
100	200	350	1012	447
274	112	382	1558	606
412	365	721	9558	3079

846	734	987	45860	11910
1000	1500	2000	200821	48716

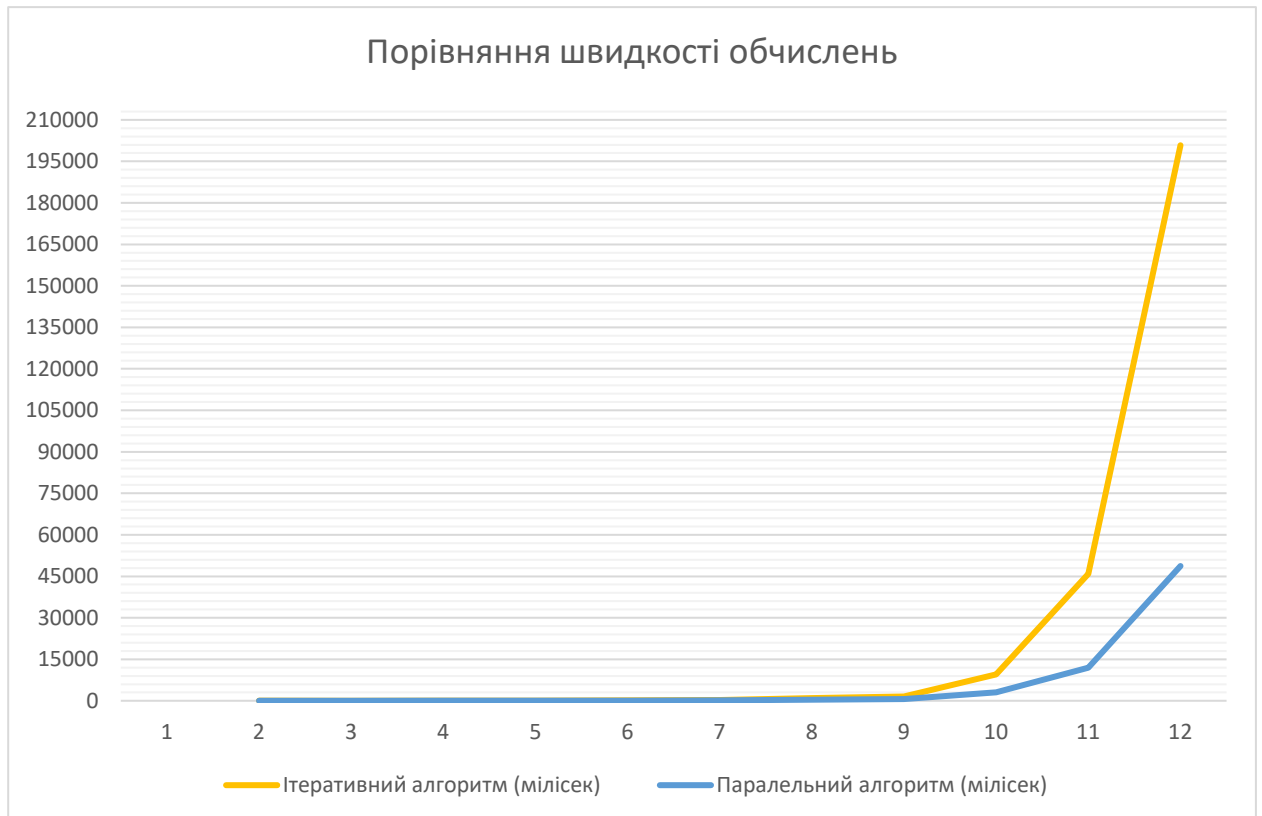


Рисунок 2 – Графік порівняння швидкості роботи алгоритмів

Можна зробити висновок, що при невеликих розмірностях вхідних матриць на часі виграв ітеративний алгоритм обчислень, але при великих вхідних розмірах матриць навпаки – кращий результат показує паралельний наївний алгоритм.

```

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: 894 millisec
Time taken by parallel naive algorithm: 419 millisec
Result matrices are identical

```

Рисунок 3 – Тестування програми

## Висновок

На цій лабораторній роботі ми освоїли реалізації паралельних обчислень на основі ітеративного алгоритму множення матриць і провели порівняльне дослідження і аналіз алгоритму паралельних обчислень з алгоритмом послідовних обчислень.