Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Лабораторна робота №1

з дисципліни: «Технології паралельного програмування в умовах великих даних»

з теми: «Паралельні обчислення в моделі зі спільною пам’яттю»

Перевірив:
доцент
Жереб К.А.

Виконав:
студент групи ІТ-01мн
Корзун І.М.

Київ – 2021

Завдання

- Обрати задачу та реалізувати для неї послідовну, в одному потоці, реалізацію та мультипоточну версію зі спільною пам’яттю.
- Забезпечити можливість змінювати кількість потоків, що використовуються для обчислень.
- Порівняти швидкодію послідовної та паралельної реалізації

Хід роботи

В якості задачі обрано матричне множення. В умовах послідовного виконання і великих розмірів вхідних матриць, від 100 елементів у рядку/стовпчику, час обчислення добутку триває недовго, але починає вимірюватися в секундах та хвилинах при багаторазовому відпрацюванні.

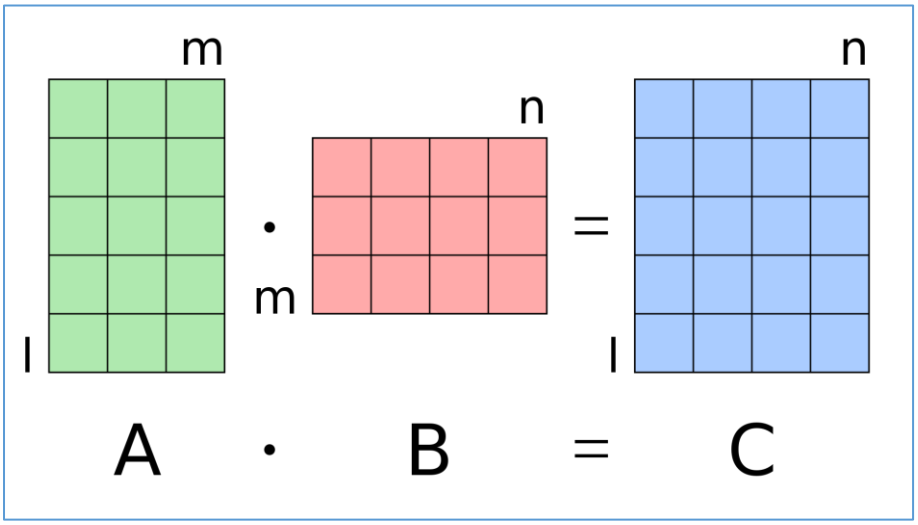

Рисунок 1. Матричне множення на прикладі

При виконанні поставленого завдання було вирішено наступні задачі:

- створено модель двомірної матриці Mat<Scalar> із перевантаженням оператору множення
- підготовлено два варіанти реалізації множення:
  - послідовне, в одному потоці
  - паралельне, у декількох потоках – за допомогою бібліотеки OpenMP
- проаналізовано час відпрацювання множення в кожній із запропонованих реалізацій

Лістинг 1. Інтерфейс для роботи з динамічним масивом як матрицею

```
class Mat
{
    std::unique_ptr<Scalar[]> data_;
    Index rows_, cols_;

public:

    static bool PAR_MUL;

    Mat() = default;
    Mat(Mat const &);
    Mat(Mat &&) = default;

    Mat(Index size);
    Mat(Index rows, Index cols);

    static Mat Random(Scalar max, Index size);
    static Mat Random(Scalar max, Index rows, Index cols);

    Index rows() const;
    Index cols() const;
    Index size() const;

    Scalar &operator()(Index row, Index col);
    Scalar const &operator()(Index row, Index col) const;
    Scalar &operator()(Index idx);
    Scalar const &operator()(Index idx) const;
    Scalar *data();
    Scalar const *data() const;

    Mat &operator=(Mat const &);
    Mat &operator=(Mat &&) = default;

    friend std::ostream &operator<<(std::ostream &out, const Mat &mat);
    friend Mat operator*(Mat const &lhs, Mat const &rhs);

    bool operator==(Mat const &oth);
};
```

Лістинг 2. Реалізація множення послідовно та паралельно із використанням OpenMP

```
Mat operator*(Mat const & lhs, Mat const & rhs)
{
    _ASSERT(lhs.cols_ == rhs.rows_);
    auto res = Mat(lhs.rows_, rhs.cols_);
    if (Mat::PAR_MUL)
    {
        auto ri = 0, ci = 0, el = 0;
#pragma omp parallel shared(lhs, rhs, res) private(ri, ci, el)
        {
#pragma omp for
            for (ri = 0; ri < lhs.rows_; ++ri)
                for (ci = 0; ci < rhs.cols_; ++ci)
                    for (el = 0; el < lhs.cols_; ++el)
                        res(ri, ci) += lhs(ri, el) * rhs(el, ci);
        }
    }
    else
    {
        for (auto ri = 0; ri < lhs.rows_; ++ri)
            for (auto ci = 0; ci < rhs.cols_; ++ci)
                for (auto el = 0; el < lhs.cols_; ++el)
                    res(ri, ci) += lhs(ri, el) * rhs(el, ci);
    }
    return res;
}
```

Лістинг 3. Реалізація вимірювання часу відпрацювання частин програми

```
#include <chrono>
#include <iostream>

#define CONCAT(a, b) CONCAT_INNER(a, b)
#define CONCAT_INNER(a, b) a ## b
#define UNIQUE_NAME(base) CONCAT(base, __LINE__)

#define tic(t) auto t(new Timer());
#define toc(t) delete t;
#define time(block_to_measure) tic(UNIQUE_NAME(t)) block_to_measure toc(UNIQUE_NAME(t))

class Timer
{
        std::chrono::steady_clock::time_point begin;

        Timer(): begin(std::chrono::steady_clock::now())
        {}
        ~Timer()
        {
                auto end = std::chrono::steady_clock::now();
                std::cout
                        << "elapsed time is "
                        << std::chrono::duration_cast<std::chrono::microseconds>
                                (end - begin).count() / 1e+6
                        << " seconds"
                        << '\n';
        }
};
```

Лістинг 4. Виконання програми у двох варіантах реалізації обраної задачі

```
#include <iostream>

#include "matrix.hpp"
#include "timer.hpp"

int main(int, char **)
{
    auto n = 1000, m = 700, k = 900;

    auto mat1 = Mat::Random(100, n, m);
    auto mat2 = Mat::Random(100, m, k);

    Mat::PAR_MUL = false;                   // 1. multiply sequentially
    time(auto prod_seq = mat1 * mat2;)      // elapsed time is 10.3598 seconds
    Mat::PAR_MUL = true;                    // 2. multiply in parallel
    time(auto prod_par = mat1 * mat2;)      // elapsed time is 2.77139 seconds

    std::cout << "seq == par: "             // calculations are equally correct
              << (prod_seq == prod_par)     // seq == par: 1
              << '\n';
}
```

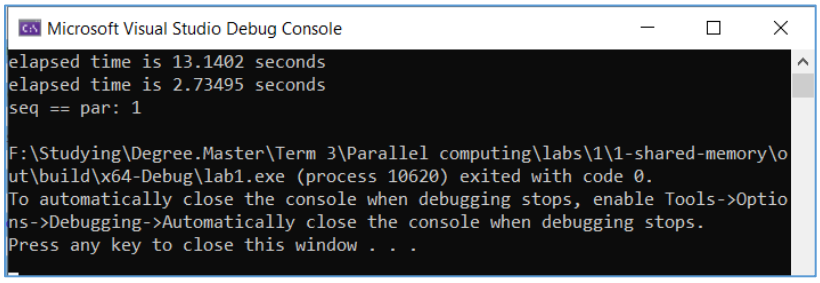

Рисунок 2. Результати порівняння послідовної та паралельної реалізації задачі зі спільною пам’яттю.

Висновок

У результаті виконання лабораторної роботи реалізовано дві версії матричного множення: послідовну та паралельну. Зі створеними програмами проведено аналіз тривалості виконання. Час роботи послідовної версії достатній для одноразового застосування на малих даних – при залученні паралельної версії в даних умовах програма деградує через необхідність синхронізації потоків. Але, коли розміри матрицю достатньо великі, паралельна версія показує помітно кращі результаті і їй слід надавати перевагу при вирішення даного типу задач.