Министерство образования Республики Беларусь

Учреждение образования

«Белорусский государственный университет

информатики и радиоэлектроники»

Кафедра информатики

Отчёт по лабораторной работе №3

«ПРОГРАММИРОВАНИЕ МНОГОЯДЕРНЫХ АРХИТЕКТУР»

Выполнил:

студент гр. 153503

Вергасов В. М.

Проверила:

Калиновская А. А.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc151498888)

[2 Постановка задачи 4](#_Toc151498889)

[3 Теоретические сведения 5](#_Toc151498890)

[4 Результат выполнения 6](#_Toc151498891)

[5 Вывод 7](#_Toc151498892)

[Приложение А 8](#_Toc151498893)

1 ЦЕЛЬ РАБОТЫ

Изучить принципы и средства реализации программирования многоядерных архитектур. Получить практические навыки по программированию и использованию этих средств.

2 ПОСТАНОВКА ЗАДАЧИ

Написать программу, реализующую алгоритм умножения матрицы на матрицу с использованием интерфейса для разработки на многоядерной архитектуре *OpenMP* с помощью *C++.*

1. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

*OpenMP* – интерфейс прикладного программирования *(API)* для масштабируемых *SMP*-систем (симметричные мультипроцессорные системы) в модели общей памяти. Исполняемый процесс в памяти может состоять из множественных нитей, которые имеют общее адресное пространство, но разные потоки команд и раздельные стэки. В простейшем случае, процесс состоит из одной нити, выполняющую функцию *main*. Нити иногда называют также потоками, легковесными процессами, *LWP (light-weight processes).* *OpenMP* основан на существовании множественных потоков в общедоступной памяти. Схема процесса представлена на рисунке 1.

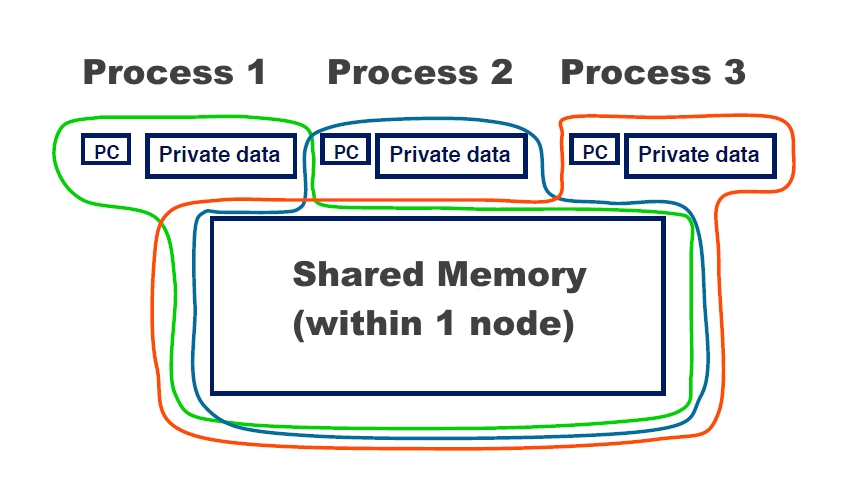


Рисунок 1– Схема процесса

Все программы *OpenMP* начинаются как единственный процесс с главным потоком. Главный поток выполняется последовательно, пока не сталкиваются с первой областью параллельной конструкции. Создание нескольких потоков *(FORK)* и объединение *(JOIN)* проиллюстрировано на рисунке 2.



Рисунок 2 – Создание и объединение нескольких потоков

1. РЕЗУЛЬТАТ ВЫПОЛНЕНИЯ

В качестве средств для написания и запуска программы использовался язык *C++*, *Microsoft Visual Studio* и *OpenMP*.

На рисунке 4.1 представлены результаты сравнения скорости работы умножения при разном размере матрий.

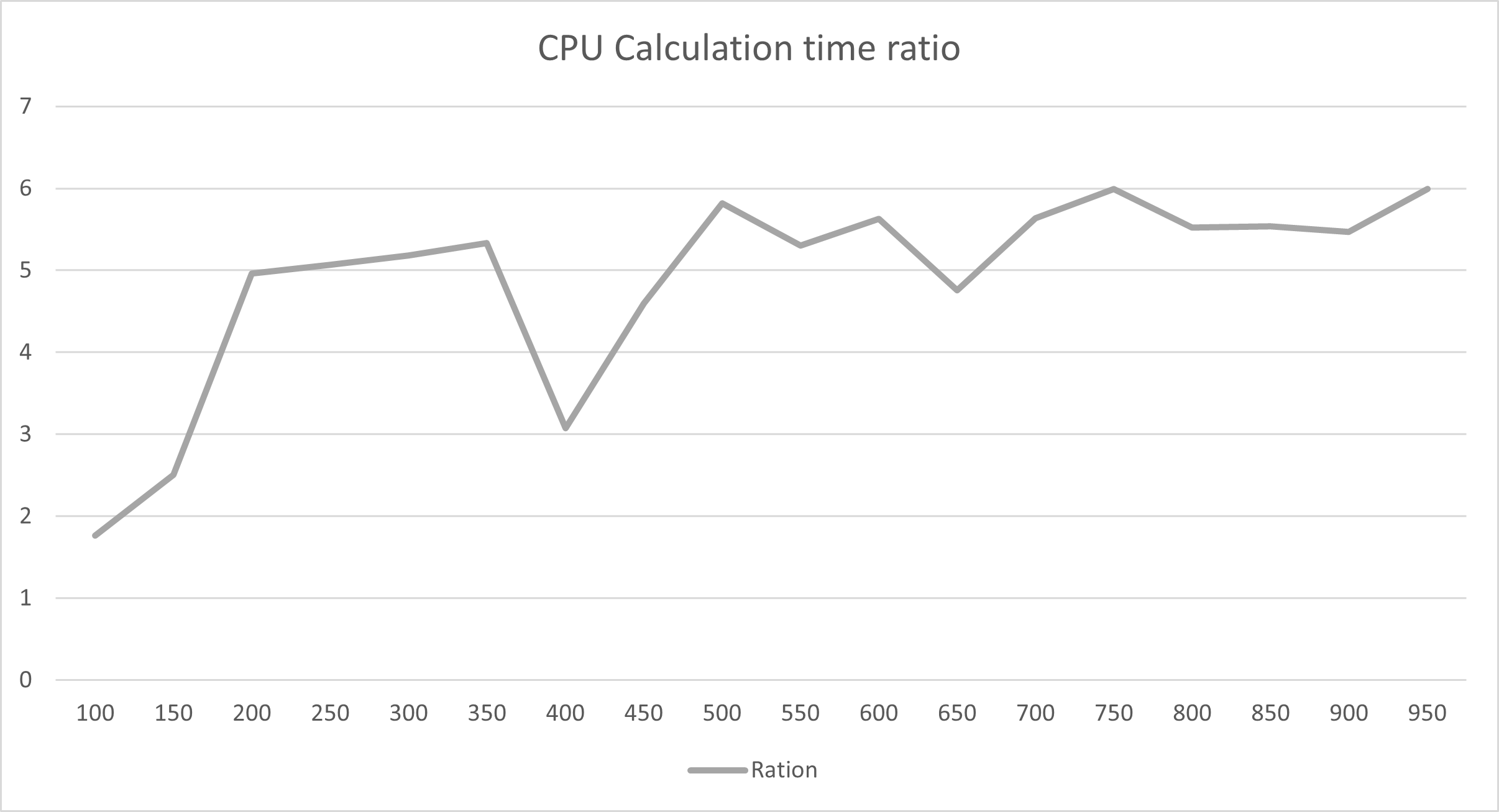


Рис. 4.1 – Результаты сравнения скорости работы умножения без многопоточности и с ней

Так же график скорости работы программы с разбиением на время выделения памяти, её заполнения и вычисления изображены на рисунке 4.2 для многопоточной работы и рисунке 4.3 для однопоточной программы.

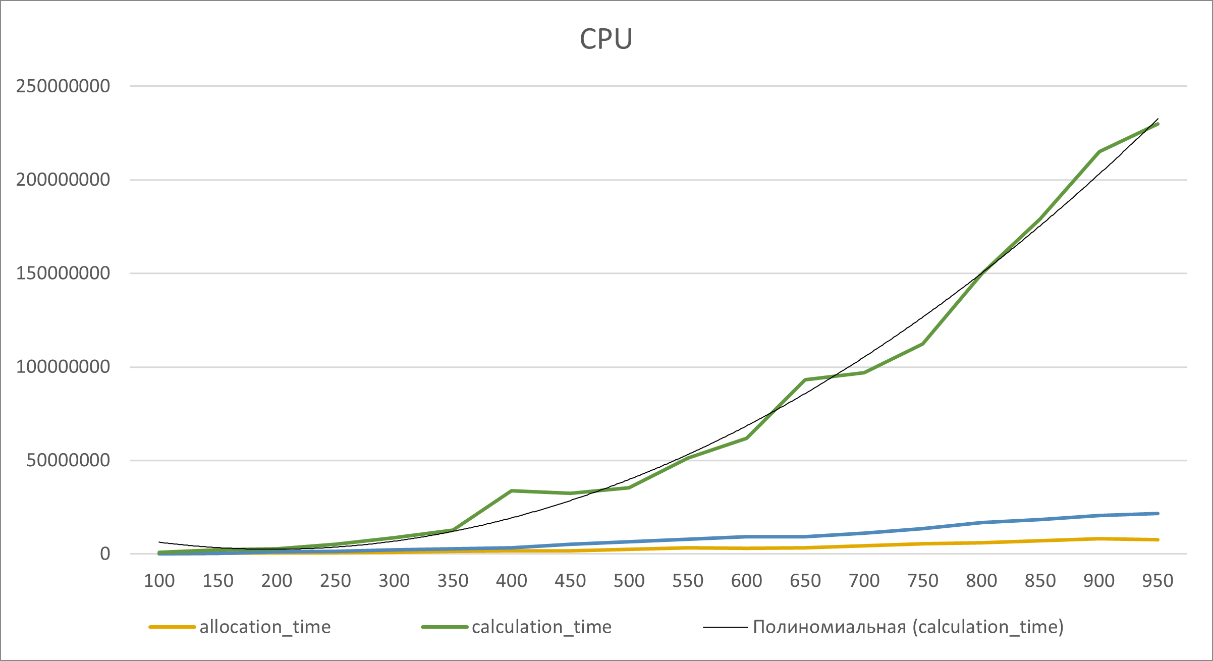


Рисунок 4.2 – время работы многопоточной программы

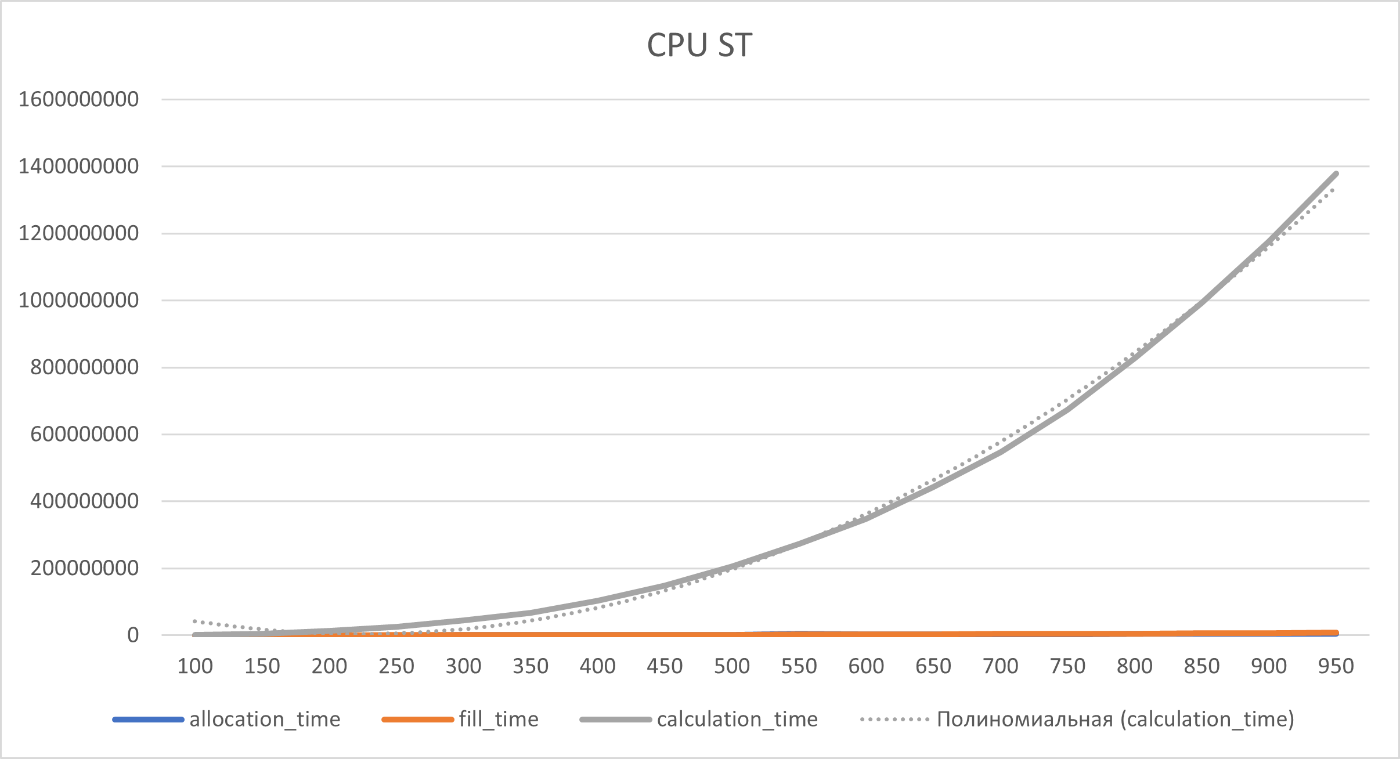


Рисунок 4.3 – время работы однопоточной программы

1. ВЫВОД

В результате выполнения лабораторной работы была разработана программа, реализующая алгоритм умножения матрицы на матрицу с использованием интерфейса для разработки на многоядерной архитектуре *OpenMP* с помощью *C++.*

ПРИЛОЖЕНИЕ А

(обязательное)

Листинг кода

Листинг 1 – Файл main.cpp

#include <iostream>

#include <vector>

#include <chrono>

#include <random>

#include <fstream>

#include <sstream>

#include <omp.h>

void multiplyOMP(const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>& B,

std::vector<std::vector<int>>& C) {

#pragma omp parallel for

for (int i = 0; i < A.size(); i++) {

for (int k = 0; k < B.size(); k++) {

for (int j = 0; j < A.front().size(); j++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

void multiply(const std::vector<std::vector<int>>& A, const std::vector<std::vector<int>>& B,

std::vector<std::vector<int>>& C) {

for (int i = 0; i < A.size(); i++) {

for (int k = 0; k < B.size(); k++) {

for (int j = 0; j < A.front().size(); j++) {

C[i][j] += A[i][k] \* B[k][j];

}

}

}

}

int main(int argc, const char\*\* argv) {

if (argc != 6) {

std::cout << "You should specify start size, end size, step size of matrix and output filename and use "

"parallel or not!"

<< std::endl;

return -1;

}

std::ostringstream output;

output << "count,allocation\_time,fill\_time,calculation\_time\n";

std::mt19937 random(time(0));

for (size\_t N = std::atoi(argv[1]); N < std::atoi(argv[2]); N += std::atoi(argv[3])) {

auto start\_serial = std::chrono::high\_resolution\_clock::now();

std::vector<std::vector<int>> A(N, std::vector<int>(N));

std::vector<std::vector<int>> B(N, std::vector<int>(N));

std::vector<std::vector<int>> C(N, std::vector<int>(N));

auto end\_serial = std::chrono::high\_resolution\_clock::now();

auto elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << N << ',' << elapsed << ',';

start\_serial = std::chrono::high\_resolution\_clock::now();

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

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

A[i][j] = random();

B[i][j] = random();

}

}

end\_serial = std::chrono::high\_resolution\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << elapsed << ',';

bool use\_mp = std::atoi(argv[5]);

start\_serial = std::chrono::high\_resolution\_clock::now();

if (use\_mp) {

multiplyOMP(A, B, C);

} else {

multiply(A, B, C);

}

end\_serial = std::chrono::high\_resolution\_clock::now();

elapsed = std::chrono::duration\_cast<std::chrono::nanoseconds>(end\_serial - start\_serial).count();

output << elapsed << '\n';

}

std::ofstream output\_file(std::string(argv[4]) + ".csv");

output\_file << output.str();

return 0;

}