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. 153501

Власенко Т. П.

Проверила:

Калиновская А. А.

Минск 2023

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc151322837)

[2 Постановка задачи](#_Toc151322838) 4

[3 Теоретические сведения 5](#_Toc151322839)

[4 Результат выполнения 6](#_Toc151322840)

[5 Вывод 7](#_Toc151322841)

[Приложение А 8](#_Toc151322842)

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 вектор 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.

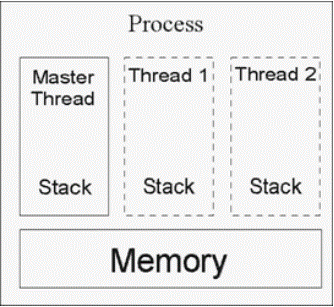


Рисунок 1– Схема процессора

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.

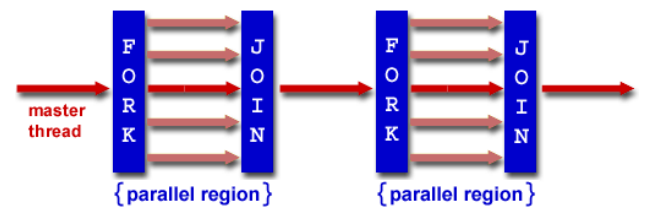


Рисунок 2 – Создание и объединение нескольких потоков

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 работы программы.



Рис. 4.1 – Результаты работы 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

#include <iostream>

#include <omp.h>

#include <vector>

#include <fstream>

#include <chrono>

#include <cmath>

#include <functional>

#include <thread>

#include <atomic>

#include <exception>

template <typename T>

void readFromBinary(std::vector<T>& v, const std::string& filename) {

std::ifstream file(filename, std::ios::binary);

if(!file.good()) throw std::invalid\_argument("Invalid file!");

size\_t size = 0;

file.read(reinterpret\_cast<char\*>(&size), sizeof(size\_t));

v.resize(size);

file.read(reinterpret\_cast<char\*>(v.data()), size \* sizeof(T));

file.close();

}

template <typename T>

void matrixMul(const std::vector<T>& a, const std::vector<T>& b, std::vector<T>& result) {

size\_t size = sqrt(a.size());

int j, k;

#pragma omp parallel for private(j, k)

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

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

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

result[i \* size + j] += a[i \* size + k] \* b[k \* size + j];

}

}

}

}

template <typename T>

void matrixAdd(const std::vector<T>& a, const std::vector<T>& b, std::vector<T>& result) {

size\_t size = a.size();

int i;

#pragma omp parallel for private(i)

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

result[i] = a[i] + b[i];

}

}

void printProgressIndicator(std::atomic<bool>& isCalculating) {

const std::string indicators = "-\\|/";

int index = 0;

while(isCalculating) {

std::cout << "\rCalculating... " << indicators[index++];

index %= indicators.size();

std::cout.flush();

std::this\_thread::sleep\_for(std::chrono::milliseconds(200));

}

std::cout << '\n';

}

template <typename T>

auto measureTime(const std::vector<T>& a, const std::vector<T>& b, std::vector<T>& result,

std::function<void(const std::vector<T>& a, const std::vector<T>& b, std::vector<T>& result)> matrix\_func) {

std::atomic<bool> isCalculating;

isCalculating.store(true);

std::thread progressThread(printProgressIndicator, std::ref(isCalculating));

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

matrix\_func(a, b, result);

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

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

isCalculating.store(false);

progressThread.join();

return elapsed\_time.count();

}

template <typename T>

void calculate(std::string op, const std::string& fileName) {

std::vector<T> a;

try {

readFromBinary(a, fileName);

}

catch(const std::exception& e) {

std::cout << e.what();

return;

}

std::vector<T> b = a;

std::vector<T> result(a.size(), 0);

std::function<void(const std::vector<T>& a, const std::vector<T>& b, std::vector<T>& result)> matrix\_func;

matrix\_func = op == "m" ? matrixMul<T> : matrixAdd<T>;

auto time = measureTime<T>(a, b, result, matrix\_func);

std::cout << time;

}

int main(int argc, char\* argv[]) {

if(argc != 4) {

std::cout << "Requires 3 args\n";

return 0;

}

else {

std::string op = argv[2];

if(op != "a" && op != "m") {

std::cout << "Usage: <file\_path> <function (m or a)>\n";

return 0;

}

std::string dataType = argv[3];

if(dataType != "i" && dataType != "f") {

std::cout << "Usage: <file\_path> <function (m or a)> <dataType (i or f)\n";

return 0;

}

}

omp\_set\_num\_threads(omp\_get\_max\_threads());

std::string op = argv[2];

std::string dataType = argv[3];

std::string fileName = argv[1];

if(dataType == "i") {

calculate<int>(op, fileName);

}

else if(dataType == "f") {

calculate<float>(op, fileName);

}

}