МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«ВЯТСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

ФАКУЛЬТЕТ КОМПЬЮТЕРНЫХ И ФИЗИКО-МАТЕМАТИЧЕСКИХ НАУК

КАФЕДРА ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №5

по дисциплине

**“Параллельное программирование”**

**Технология OpenMP**

Выполнил: студент гр. ФИб-3301-51-00 Ощепков Д. О. \_\_\_\_\_\_\_\_\_\_\_

Проверил: доцент кафедры ПМИ Чупраков П. Г. \_\_\_\_\_\_\_\_\_\_\_

Киров

2024

**Параметры тестовой машины**

Running Microwave

Run on (4 X 2600 MHz CPU s)

CPU Caches:

L1 Data 32 KiB (x2)

L1 Instruction 64 KiB (x2)

L2 Unified 512 KiB (x2)

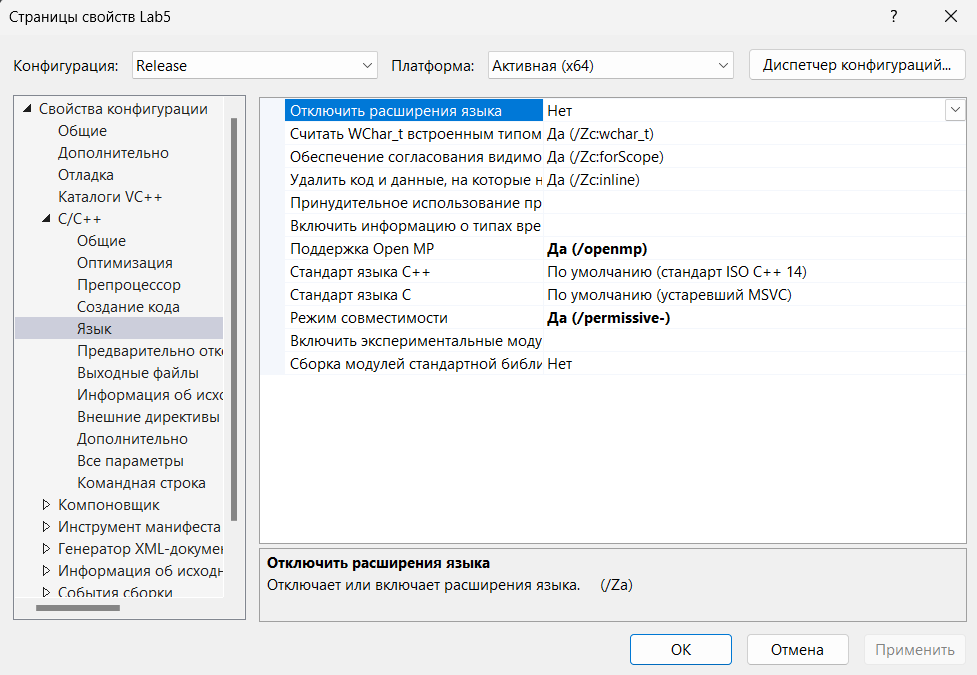
L3 Unified 4096 KiB (x1)

**Цель лабораторной работы**

Получить навыки разработки параллельных алгоритмов с использованием технологии OpenMP.

**Задание 0**

Проверить, поддерживается ли OpenMP. Если нет, то включить поддержку в параметрах проекта.



Код программы:

#include <iostream>

using namespace std;

int main() {

system("chcp 1251>nul");

#ifdef \_OPENMP

printf("OpenMP is supported! %d\n", \_OPENMP);

#else

printf("OpenMP is not supported!\n");

#endif

}

Результат работы программы:



**Задание 1**

Скопировать фрагмент кода в программу. Запустить на выполнение.

Код программы:

#include <omp.h>

#include <iostream>

using namespace std;

int main() {

int nthreads, tid;

// Создание параллельной области

#pragma omp parallel private(tid)

{

// печать номера потока

tid = omp\_get\_thread\_num();

printf("Hello World from thread = %d\n", tid);

// Печать количества потоков – только master

if (tid == 0) {

nthreads = omp\_get\_num\_threads();

printf("Number of threads = %d\n", nthreads);

}

} // Завершение параллельной области

}

Результат работы программы:



Ответить на вопросы:

1. Зачем нужна директива parallel? – *для создания параллельной области, где указанный код выполняется одновременно несколькими потоками.*
2. Сколько потоков было запущено? *Почему? – 16, зависит от числа ядер процессора.*
3. Сколько потоков одновременно работают с переменной tid? Почему? – *только один поток за раз, так как она объявлена как private в директиве #pragma omp parallel – у каждого потока своя копия переменной tid.*
4. Поток с каким tid останется после завершения параллельной области? – *останется только основной поток master thread с tid = 0*

Запросить требуемое количество потоков у пользователя. Задать количество потоков для параллельной области.

Код программы:

#include <omp.h>

#include <iostream>

using namespace std;

int main() {

system("chcp 1251>nul");

int nthreads, tid, requested\_threads;

// Запрос количества потоков у пользователя

cout << "Введите количество потоков: ";

cin >> requested\_threads;

// Установка количества потоков для параллельной области

omp\_set\_num\_threads(requested\_threads);

// Создание параллельной области

#pragma omp parallel private(tid)

{

// печать номера потока

tid = omp\_get\_thread\_num();

printf("Hello World from thread = %d\n", tid);

// Печать количества потоков – только master

if (tid == 0) {

nthreads = omp\_get\_num\_threads();

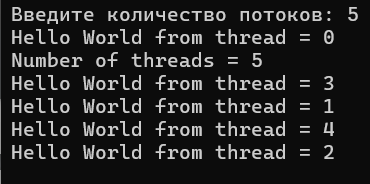
printf("Number of threads = %d\n", nthreads);

}

} // Завершение параллельной области

}

Результат работы программы:



**Задание 2**

Написать программу, задающую работу двух потоков. Первый поток в цикле выводит последовательно числа от 1 до N, а второй – N раз выводит слово «HELLO». Число N задаётся пользователем.

Код программы:

#include <omp.h>

#include <iostream>

using namespace std;

int main() {

system("chcp 1251>nul");

int N;

// Запрос количества итераций у пользователя

cout << "Введите N: ";

cin >> N;

// Установка количества потоков

omp\_set\_num\_threads(2);

// Создание параллельной области

#pragma omp parallel

{

int tid = omp\_get\_thread\_num();

if (tid == 0) {

// Первый поток: вывод чисел от 1 до N

for (int i = 1; i <= N; i++) {

cout << "Поток 1: " << i << endl;

}

}

else if (tid == 1) {

// Второй поток: вывод слова "HELLO" N раз

for (int i = 1; i <= N; i++) {

cout << "Поток 2: HELLO" << endl;

}

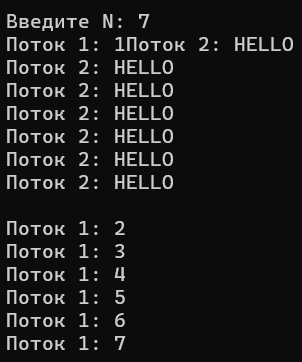
}

} // Конец параллельной области

return 0;

}

Результат работы программы:



**Задание 3**

Написать параллельную программу, находящую поэлементное произведение двух массивов размера N. Задать параметр schedule, попробовать разные аргументы.

Код программы:

#include <omp.h>

#include <iostream>

#include <vector>

int main() {

system("chcp 1251>nul");

int N;

std::cout << "Введите размер массивов N: ";

std::cin >> N;

// Инициализация массивов

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

// Заполнение массивов данными

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

A[i] = i + 1; // Пример: элементы от 1 до N

B[i] = i + 2; // Пример: элементы от 2 до N+1

}

// Параллельное вычисление поэлементного произведения

#pragma omp parallel for schedule(static, 4)

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

C[i] = A[i] \* B[i];

}

// Вывод результата

std::cout << "Произведение массивов:" << std::endl;

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

std::cout << C[i] << " ";

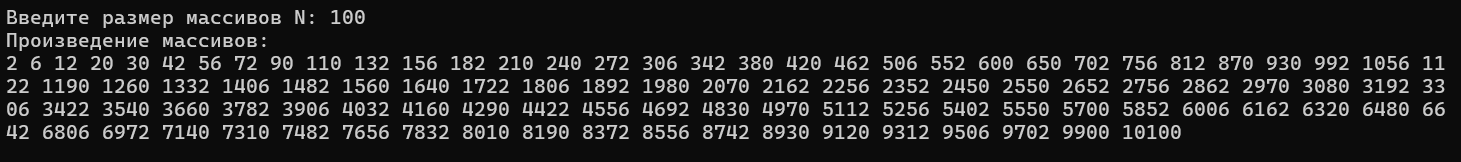
}

std::cout << std::endl;

return 0;

}

Результат работы программы:



**Задание 4**

Написать параллельную программу, вычисляющую максимальное значение среди элементов вектора

1. с использованием параметра reduction;

#include <omp.h>

#include <iostream>

#include <vector>

#include <limits>

int main() {

system("chcp 1251>nul");

int N = 100;

std::vector<int> vec(N, 0);

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

vec[i] = i;

}

int max\_val = std::numeric\_limits<int>::min();

#pragma omp parallel for reduction(max:max\_val)

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

if (vec[i] > max\_val) {

max\_val = vec[i];

}

}

std::cout << "Максимальное значение: " << max\_val << std::endl;

return 0;

}

1. с использованием директивы critical;

#include <omp.h>

#include <iostream>

#include <vector>

#include <limits>

int main() {

system("chcp 1251>nul");

int N = 100;

std::vector<int> vec(N, 0);

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

vec[i] = i;

}

int max\_val = std::numeric\_limits<int>::min();

#pragma omp parallel for

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

#pragma omp critical

{

if (vec[i] > max\_val) {

max\_val = vec[i];

}

}

}

std::cout << "Максимальное значение: " << max\_val << std::endl;

return 0;

}

1. с использованием атомарных операций.

#include <omp.h>

#include <iostream>

#include <vector>

#include <limits>

int main() {

system("chcp 1251>nul");

int N = 100;

std::vector<int> vec(N, 0);

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

vec[i] = i;

}

int max\_val = std::numeric\_limits<int>::min();

#pragma omp parallel for

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

#pragma omp atomic write

max\_val = std::max(max\_val, vec[i]);

}

std::cout << "Максимальное значение: " << max\_val << std::endl;

return 0;

}

Результаты работы программ:



В чем отличия между этими способами?

Reduction: эффективен для операций, которые могут быть разделены на части и выполнены параллельно, с последующим объединением результатов. Это наиболее эффективный вариант из предложенных, так как избегает блокировок и использует локальные копии переменной max\_val для каждого потока.

Critical: обеспечивает взаимное исключение, гарантируя, что только один поток может выполнить блок кода в данный момент времени. Это может стать узким местом в производительности из-за сериализации доступа к общему ресурсу.

Atomic: похож на critical, но ограничен одной атомарной операцией (в данном случае присваиванием). Более эффективен, чем critical, так как затрагивает меньшую часть кода.

**Задание 5**

Написать программу, задающую работу потоков. Первые потоков вычисляют сумму от 1 до 𝑁, а оставшиеся 𝐾 потоков вычисляют длину 𝑁-мерного вектора. Число 𝑁 задаётся пользователем.

*Указание*. Использовать секции и вложенный параллелизм

#include <iostream>

#include <omp.h>

#include <cmath>

int task5() {

system("chcp 1251>nul");

int N, M, K;

std::cout << "Введите число N: ";

std::cin >> N;

std::cout << "Введите число потоков для вычисления суммы (M): ";

std::cin >> M;

std::cout << "Введите число потоков для вычисления длины вектора (K): ";

std::cin >> K;

// Создаем вектор с N элементами после ввода N

std::vector<double> vector(N);

// Заполняем вектор случайными числами

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_real\_distribution<> dis(-10.0, 10.0);

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

vector[i] = dis(gen);

}

std::cout << "Вектор ";

printArray(vector);

// Создаем две параллельные секции для одновременного выполнения задач

#pragma omp parallel sections

{

// Первая секция для вычисления суммы

#pragma omp section

{

double sum = 0;

#pragma omp parallel for num\_threads(M) reduction(+:sum)

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

sum += vector[i];

}

#pragma omp critical

std::cout << "Сумма чисел вектора " << sum << std::endl;

}

// Вторая секция для вычисления длины вектора

#pragma omp section

{

double vector\_length = 0.0;

#pragma omp parallel for num\_threads(K) reduction(+:vector\_length)

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

vector\_length += vector[i] \* vector[i];

}

vector\_length = std::sqrt(vector\_length);

#pragma omp critical

std::cout << "Длина вектора: " << vector\_length << std::endl;

}

}

return 0;

}

**Задание 6**

Написать параллельную программу, которая каждый элемент вектора размера N заменяет на его наибольший простой делитель. Число N задается пользователем. Элементы вектора – случайные натуральные числа из диапазона .

Замерить среднее время выполнения программы для N = 2·107 , 5·107 и 108 на 1, 2, 4 и 8 потоках. Вычислить среднее ускорение для 2, 4 и 8 потоков. Построить 16 диаграмму зависимости ускорения от числа потоков для каждого размера вектора (3 графика на одной диаграмме)

#include <iostream>

#include <vector>

#include <thread>

#include <mutex>

#include <chrono>

#include <numeric>

int task6() {

int N;

std::cout << "Введите размер вектора (N): ";

std::cin >> N;

// Создаем вектор случайных чисел

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dist(100000, 1'000'000);

std::vector<int> numbers(N);

std::generate(numbers.begin(), numbers.end(), [&]() { return dist(gen); });

// Заменяем каждый элемент на его наибольший простой делитель

tr(numbers);

// Выводим результаты (для проверки)

std::cout << "Результат:" << std::endl;

for (int num : numbers) {

std::cout << num << " ";

}

std::cout << std::endl;

return 0;

}

static void BM\_tr(benchmark::State& state, int size, int numThreads) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dist(100000, 1000000);

std::vector<int> numbers(size);

std::generate(numbers.begin(), numbers.end(), [&]() { return dist(gen); });

for (auto \_ : state) {

auto res = tr(numbers, numThreads);

benchmark::DoNotOptimize(res);

}

}

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

for (int N : {1e6, 2e6, 1e7}) {

for (int numThread : {1, 2, 4, 8}) {

benchmark::RegisterBenchmark(std::format("BM\_tr/size{}/threads{}", N, numThread),

BM\_tr,

N,

numThread)

->Unit(benchmark::kMillisecond);

}

}

benchmark::Initialize(&argc, argv);

benchmark::RunSpecifiedBenchmarks();

return 0;

}

Таблица 1 – Время выполнения алгоритма в секундах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Потоки | | | |
| 1 | 2 | 4 | 8 |
|  | 0.004 | 0.002 | 0.002 | 0.003 |
|  | 0.02 | 0.01 | 0.009 | 0.008 |
|  | 0.25 | 0.1 | 0.06 | 0.03 |

Таблица 2 – Ускорение в разах

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность | Потоки | | |
| 2 | 4 | 8 |
|  | 2 | 2 | 1.3 |
|  | 2 | 2.2 | 2.5 |
|  | 2.5 | 4.1 | 8.3 |

**Задание 7**

Написать параллельную программу, выполняющую умножение двух матриц размера . Разработать программы с использованием распараллеливания циклов разного уровня вложенности. Замерить среднее время выполнения программ для на 1, 2, 4 и 8 потоках. Сравнить полученные результаты. Оцените величину накладных расходов на создание и завершение потоков. Для оптимального варианта вычислить среднее ускорение на 2, 4 и 8 потоках. Построить диаграмму зависимости ускорения от числа потоков для каждого размера матриц (3 графика на одной диаграмме).

#include <iostream>

#include <vector>

#include <random>

#include <chrono>

#include <omp.h>

using Matrix = std::vector<std::vector<int>>;

// Функция для инициализации матрицы размером N x N

void initializeMatrix(Matrix& matrix, int N) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 10);

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

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

matrix[i][j] = dis(gen);

}

}

}

// Функция умножения матриц

void multiplyMatrices(const Matrix& A, const Matrix& B, Matrix& C, int numThreads) {

#pragma omp parallel for collapse(2) num\_threads(numThreads)

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

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

C[i][j] = 0;

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

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

}

}

}

}

static void BM\_mult(benchmark::State& state, int size, int numThreads) {

Matrix A(size, std::vector<int>(size));

Matrix B(size, std::vector<int>(size));

Matrix C(size, std::vector<int>(size));

initializeMatrix(A, size);

initializeMatrix(B, size);

for (auto \_ : state) {

multiplyMatrices(A, B, C, numThreads);

}

}

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

for (int N : {500, 1000, 2000}) {

for (int numThread : {1, 2, 4, 8}) {

benchmark::RegisterBenchmark(std::format("BM\_tr/size{}/threads{}", N, numThread),

BM\_mult,

N,

numThread)

->Unit(benchmark::kMillisecond);

}

}

benchmark::Initialize(&argc, argv);

benchmark::RunSpecifiedBenchmarks();

return 0;

}

Результаты работы программы

Таблица 3 – Время выполнения алгоритма в секундах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Потоки | | | |
| 1 | 2 | 4 | 8 |
| 500 | 0.2 | 0.1 | 0.07 | 0.04 |
| 1000 | 1.8 | 0.9 | 0.5 | 0.3 |
| 2000 | 27.4 | 13.2 | 7.3 | 4.7 |

Таблица 4 – Ускорение в разах

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность | Потоки | | |
| 2 | 4 | 8 |
| 500 | 2 | 2.8 | 4.5 |
| 1000 | 2 | 3.3 | 5.1 |
| 2000 | 2 | 3.7 | 5.8 |

**Задание 8**

Написать параллельную программу, выполняющую поиск максимального значения среди минимальных элементов строк матрицы размера N × N. Обосновать выбор средств и методов для распараллеливания. Замерить среднее время выполнения программ для N = 500, 1000, 2000 на 1, 2, 4 и 8 потоках. Вычислить среднее ускорение на 2, 4 и 8 потоках.

#include <iostream>

#include <vector>

#include <limits>

#include <random>

#include <chrono>

#include <omp.h>

using Matrix = std::vector<std::vector<int>>;

// Функция для инициализации матрицы размером N x N

void initializeMatrix(Matrix& matrix, int N) {

std::random\_device rd;

std::mt19937 gen(rd());

std::uniform\_int\_distribution<> dis(1, 10);

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

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

matrix[i][j] = dis(gen);

}

}

}

static void BM\_aaaa(benchmark::State& state, int size, int numThreads) {

std::vector<std::vector<int>> matrix(size, std::vector<int>(size));

initializeMatrix(matrix, size);

for (auto \_ : state) {

int maxMin = std::numeric\_limits<int>::min();

std::vector<int> mins(size);

#pragma omp parallel for num\_threads(numThreads)

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

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

mins[i] = std::min(mins[i], matrix[i][j]);

}

}

int maxOfAllMins = mins[0];

for (int i = 1; i < size; ++i) {

maxOfAllMins = std::max(maxOfAllMins, mins[i]);

}

benchmark::DoNotOptimize(maxOfAllMins);

}

}

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

for (int N : {5000, 10000, 20000}) {

for (int numThread : {1, 2, 4, 8}) {

benchmark::RegisterBenchmark(std::format("BM\_tr/size{}/threads{}", N, numThread),

BM\_aaaa,

N,

numThread)

->Unit(benchmark::kMillisecond);

}

}

benchmark::Initialize(&argc, argv);

benchmark::RunSpecifiedBenchmarks();

return 0;

}

Таблица 5 – Время выполнения алгоритма в секундах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размерность | Потоки | | | |
| 1 | 2 | 4 | 8 |
| 5000 | 0.005 | 0.003 | 0.003 | 0.004 |
| 10000 | 0.01 | 0.01 | 0.01 | 0.01 |
| 20000 | 0.06 | 0.04 | 0.03 | 0.04 |

Таблица 6 – Ускорение в разах

|  |  |  |  |
| --- | --- | --- | --- |
| Размерность | Потоки | | |
| 2 | 4 | 8 |
| 5000 | 1.4 | 1.6 | 1.2 |
| 10000 | 1.5 | 1.4 | 1.3 |
| 20000 | 1.5 | 1.6 | 1.3 |