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

Отчёт по лабораторной работе №5  
по дисциплине «Параллельные вычисления»

**Технология OpenMP**

Выполнил: студент группы ФИб-4301-51-00     / К.О. Дёмин /

Проверил: ст.преподаватель каф. ПМИ     / А.В. Торбеева /

Киров 2019

Оглавление

[Цель работы 3](#_Toc25567491)

[Задание 3](#_Toc25567492)

[Листинг 4](#_Toc25567493)

[Полученные результаты 7](#_Toc25567494)

[Выводы по лабораторной работе 10](#_Toc25567495)

# Цель работы

Получить навыки разработки параллельных алгоритмов с использованием технологии OpenMP.

# Задание

1. Написать параллельную программу, вычисляющую число π как ,   
   а) с использованием параметра reduction; б) с использованием директивы critical.

Замерить среднее время выполнения программ (по 2-3 значения, на каждое – не менее 10 запусков) на 1, 2, 4 и 8 потоках. Вычислить среднее ускорения для 2, 4 и 8 потоков. Рассчитать эффективность параллельного алгоритма. Построить графики ускорения и эффективности.

1. Написать параллельную программу, выполняющую умножение двух матриц. При разработке алгоритма учитывать особенности работы с памятью.

Замерить среднее время выполнения программ для *N* = 500, 1000, 1500, 2000 на 1, 2, 4 и 8 потоках. Вычислить среднее ускорения для 2, 4 и 8 потоков. Рассчитать эффективность параллельного алгоритма. Построить графики ускорения и эффективности.

# Листинг

Задание 1

#include <iostream>

#include "omp.h"

#include <chrono>

const int Iterations = 10;  //  число итераций

double criticalCalc(long parts, double step)

{

double result = 0;  //  результат

double x = 0;

long i = 0;

double temp = 0;

//  каждый поток считает сумму отдельно

#pragma omp parallel private(x, i) firstprivate(temp)

{

#pragma omp for

for (i = 0; i < parts; ++i)

{

x = (i + .5) \* step;

temp += 4.0 / (1.0 + x \* x);

}

//  прибавляем локальное значение в критической сессии

#pragma omp critical

result += temp;

}

return result \* step;

}

double reductionCalc(long parts, double step)

{

double result = 0;

double x = 0;

long i = 0;

//  параллельно считаем через редукцию

#pragma omp parallel for private(x, i) reduction(+: result)

for (i = 0; i < parts; ++i)

{

x = (i + .5) \* step;

result += 4.0 / (1 + x \* x);

}

return result \* step;

}

int main()

{

long parts = 0;

std::cout << "Input number of partitions:" << std:: endl;

std::cin >> parts;

int threadCount = 0;

std::cout << "Input thread count:" << std::endl;

std::cin >> threadCount;

omp\_set\_num\_threads(threadCount);  //  устанавливаем число потоков

double step = 1.0 / parts;

double reductionTime = 0, criticalTime = 0;

std::chrono::time\_point<std::chrono::high\_resolution\_clock> start, end;

double res = 0;

for (int i = 0; i < Iterations; ++i)

{

//  считаем время и выводим результат

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

res = reductionCalc(parts, step);

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

printf("%.9f\n", res);

auto diff = end - start;

reductionTime += diff.count();

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

res = criticalCalc(parts, step);

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

printf("%.9f\n", res);

diff = end - start;

criticalTime += diff.count();

}

//  выводим время

std::cout << "Reduction time:\t" << reductionTime / Iterations << std::endl;

std::cout << "Critical time:\t" << criticalTime / Iterations << std::endl;

return 0;

}

Задание 2

#include <iostream>

#include <chrono>

#include <omp.h>

const int Iterations = 10;

//  заполняем матрицу случайными значениями

void generateMatrix(int\*\* arr, int size)

{

for (int i = 0; i < size; ++i)

for (int j = 0; j < size; ++j)

arr[i][j] = std::rand() & 10;

}

//  заполняем матрицу 0

void initMatrix(int\*\* arr, int size)

{

for (int i = 0; i < size; ++i)

for (int j = 0; j < size; ++j)

arr[i][j] = 0;

}

//  параллельное перемножение матриц

void ikj(int\*\* first, int\*\* second, int\*\* res, int size)

{

int i, k, j;

#pragma omp parallel for shared(first, second, res) private(i, k, j)

for (i = 0; i < size; ++i)

for (k = 0; k < size; ++k)

for (j = 0; j < size; ++j)

res[i][j] += (first[i][k] \* second[k][j]);

}

//  создание матрицы

int\*\* createMatrix(int size)

{

int\*\* arr = new int\*[size];

for (int i = 0; i < size; ++i)

arr[i] = new int[size];

return arr;

}

//  удаление матрицы

void deleteMatrix(int\*\* arr, int size)

{

for (int i = 0; i < size; ++i)

delete[] arr[i];

delete[] arr;

}

int main()

{

int size;

std::cout << "Input matrix size" << std::endl;

std::cin >> size;

//  создаём матрицу и заполняем

int\*\* first = createMatrix(size);

generateMatrix(first, size);

int\*\* second = createMatrix(size);

generateMatrix(second, size);

int threads;

std::cout << "Input thread count" << std::endl;

std::cin >> threads;

omp\_set\_num\_threads(threads);  //  устанавливаем число потоков

double time = 0;

std::chrono::time\_point<std::chrono::high\_resolution\_clock> start, end;

int\*\* res = createMatrix(size);

//  считаем затраченное время на итерациях

for (int i = 0; i < Iterations; ++i)

{

initMatrix(res, size);

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

ikj(first, second, res, size);

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

auto diff = end - start;

time += diff.count();

}

//  выводим время и удаляем матрицы

std::cout << "Time: " << time / Iterations << std::endl;

deleteMatrix(res, size);

deleteMatrix(first, size);

deleteMatrix(second, size);

std::cin.get();

return 0;

}

# Полученные результаты

Тестирование проводилось на процессоре Intel Core i7 4770 с частотой 3,7 ГГц, четырьмя ядрами и восьмью потоками, 16 Гб ОЗУ с частотой 1600 МГц.

Задание 1.

В таблице 1 представлено время работы алгоритма с использованием reduction.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Число разбиений | Число потоков | | | |
| 1 | 2 | 4 | 8 |
| 10 7 | 0,070 | 0,035 | 0,019 | 0,011 |
| 10 8 | 0,711 | 0,347 | 0,185 | 0,112 |
| 10 9 | 6,886 | 3,544 | 1,819 | 1,074 |
| Таблица 1. Время на вычисление числа пи через reduction. | | | | |

В таблице 2 представлено время работы алгоритма с использованием критических сессий.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Число разбиений | Число потоков | | | |
| 1 | 2 | 4 | 8 |
| 10 7 | 0,072 | 0,035 | 0,019 | 0,011 |
| 10 8 | 0,705 | 0,347 | 0,185 | 0,105 |
| 10 9 | 6,882 | 3,543 | 1,841 | 1,058 |
| Таблица 2. Время вычисления числа пи через critical. | | | | |

Как видно из таблиц, время работы алгоритмов через разную реализацию примерно совпадает и может отличаться из-за погрешности.

Ниже представлены графики ускорения и эффективности с реализацией через reduction.

Максимальная эффективность приближается к 6,5 на 8 потоках с размером делений интервала .

Посмотрим графики ускорения и эффективности с реализацией через critical.

Реализация через critical отличается от реализации reduction на уровне погрешности. Ускорение и эффективность практически идентичны.

Задание 2.

В таблице 3 представлено время работы алгоритма параллельного перемножения матриц.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Размер матрицы | Число потоков | | | |
| 1 | 2 | 4 | 8 |
| 500 | 0,421 | 0,214 | 0,166 | 0,146 |
| 1000 | 3,368 | 1,689 | 1,136 | 0,889 |
| 1500 | 11,425 | 5,804 | 3,533 | 2,952 |
| 2000 | 26,966 | 13,639 | 8,442 | 6,982 |
| Таблица 3. Время работы алгоритма перемножения матриц. | | | | |

Ниже представлены графики ускорения и эффективности.

Максимальное ускорение приближается к 4 на восьми потоках и размерах матрицы более 1500. При этом, эффективность приближается к 0,5 сверху.

# Выводы по лабораторной работе

При выполнении лабораторной работы были получены навыки разработки с использованием OpenMP. Распараллеливание с использованием данной технологии позволяет получить в некоторых задачах ускорение почти в 6,5 раз. Кроме того, использование reduction может сильно упростить процесс разработки и получить аналогичную производительность.