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 №1  
по дисциплине «Параллельные вычисления»

**Многопоточность в языке C++**

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

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

Киров 2019

Оглавление

[Цель работы 3](#_Toc19438902)

[Задание 3](#_Toc19438903)

[Листинг 4](#_Toc19438904)

[Полученные результаты 7](#_Toc19438905)

[Выводы по лабораторной работе 9](#_Toc19438906)

# Цель работы

Получить навыки работы с потоками в языке C++.

# Задание

1. Запустить на выполнение пример 1. Заменить join на detach. Прокомментировать результат.
2. Создать *N* потоков (число *N* ≤ 10 задаётся при запуске). Каждый поток должен выводить информацию о своем запуске и завершении. Не забудьте имитировать бурную деятельность потока.
3. Написать программу, вычисляющую сумму чисел от 1 до *N*. Число *N* вводится как параметр командной строки. Замерить время работы программы. Результат вывести на консоль.
4. Распараллелить вычисление суммы с использованием std::thread. Результат, полученный каждым потоком, записать в разделяемую переменную. Для синхронизации потоков использовать std::mutex.
5. Замерить время работы программы для *N* = 107, 108 и 109 на 1, 2, 4 и 8 потоках. На каждом примере запустить не менее 10 раз. В таблицу занести среднее время выполнения на одном примере в секундах.

Таблица 1 – Время параллельного суммирования элементов массива в секундах

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размерность** | **Число потоков** | | | |
| **1** | **2** | **4** | **8** |
| **107** |  |  |  |  |
| **108** |  |  |  |  |
| **109** |  |  |  |  |

1. Вычислить ускорения для каждого значения *N* и построить диаграмму зависимости ускорения от числа потоков (три графика на одной диаграмме).

# Листинг

Задание 1

// task 1.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <thread>

#include <iostream>

using namespace std;

// Метод, исполняемый потоком

void foo(int i)

{

std::cout << "thread " << i << " start!\n";

std::this\_thread::sleep\_for(std::chrono::seconds(4));

std::cout << "thread " << i << " finish!\n";

}

int main()

{

std::thread myThread;

std::cout << "before starting, joinable: " << myThread.joinable() << '\n';

std::cout << "Start new thread.\n";

myThread = std::thread(foo, 1); // инициализация потока

std::cout << "after starting, joinable: " << myThread.joinable() << '\n';

myThread.join(); // привязка потока к текущему потоку, т.е. главный поток будет ожидать завершения потока

std::cout << "after finish, joinable: " << myThread.joinable() << '\n';

std::cout << "Bye!\n";

return 0;

}

Задание 2

// task 2.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <thread>

#include <iostream>

#include <mutex>

using namespace std;

std::mutex mutex\_lock; // мьютекс для синхронизации

void foo(int i)

{

mutex\_lock.lock(); // блокируем

std::cout << "thread " << i << " start!\n"; // выводим текст

mutex\_lock.unlock(); // разблокируем

std::this\_thread::sleep\_for(std::chrono::seconds(1)); // имитируем деятельность

mutex\_lock.lock(); // блокируем, выводим текст и разблокируем

std::cout << "thread " << i << " finish!\n";

mutex\_lock.unlock();

}

int main()

{

// читаем число потоков с консоли

int N = 0;

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

do

std::cin >> N;

while (N <= 0 || N > 10);

std::thread\* myThread = new thread[N];

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

{

// создаем и запускаем потоки

myThread[i] = std::thread(foo, i);

}

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

myThread[i].join(); // привязываем потоки к главному и ожидаем их завершения

std::cout << "Bye!\n";

delete[] myThread;

system("pause");

return 0;

}

Задание 3

// task 3.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <iostream>

#include <chrono>

// метод для высчитывания диапазона

void foo(long long & val, int start, int count)

{

int length = start + count; // промежуток, который надо посчитать

for (int j = start; j < length; ++j)

{

val += j;

}

}

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

{

int n = atoi(argv[1]);

long long val = 0;

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

start = std::chrono::high\_resolution\_clock::now(); // записываем время начала рассчета

foo(val, 1, n); // считаем

end = std::chrono::high\_resolution\_clock::now(); // запоминаем время завершения

std::chrono::duration<double> diff = end - start;

std::cout << "time to sum = " << diff.count() << "s\n";

std::cout << "Result: " << val << std::endl;

system("pause");

return 0;

}

Задание 4

// task 4.cpp: определяет точку входа для консольного приложения.

//

#include "stdafx.h"

#include <iostream>

#include <chrono>

#include <thread>

#include <mutex>

#include <cmath>

std::mutex mutex\_lock; // мьютекс для синхронизации

const int Iterations = 10; // число тестов (итераций)

void foo(long long & val, int start, int count)

{

// считаем значение в промежутке

int length = start + count;

long long res = 0;

for (int j = start; j < length; ++j)

{

res += j;

}

mutex\_lock.lock(); // блокируем мьютекс

val += res; // прибавляем значение к

mutex\_lock.unlock(); // разблокируем

}

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

{

int threads = 0;

int range = atoi(argv[1]); // считываем число элементов из параметров командной строки

do

{

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

std::cin >> threads;

}

while (threads <= 0 || threads > 10); // считываем количество потоков

std::thread\* myThread = new std::thread[threads];

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

double timeval = 0;

int step = (int)ceil(range / (double)threads); // интервал для одного потока

if (step == 0 || range < threads) { // если число меньше числа потоков, то сокращаем кол-во потоков

step = 1;

threads = range;

}

for (int j = 0; j < Iterations; ++j)

{

long long val = 0;

int first = 1;

int remainder = range; // остаток

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

for (int i = 0; i < threads; ++i) // зваускаем потоки

{

myThread[i] = std::thread(foo, std::ref(val), first, remainder < step ? remainder : step);

first += step; // увеличиваем первый элемент последовательности

remainder -= step; // уменьшаем остаток

}

for (int i = 0; i < threads; ++i)

myThread[i].join(); // ожидаем потоки

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

std::chrono::duration<double> diff = end - start; // считаем разницу и выводим

std::cout << "time to sum = " << diff.count() << "s\n";

std::cout << "Result: " << val << std::endl;

timeval += diff.count();

}

std::cout << "Time: " << timeval / Iterations << std::endl; // выводим среднее время

delete[] myThread;

system("pause");

return 0;

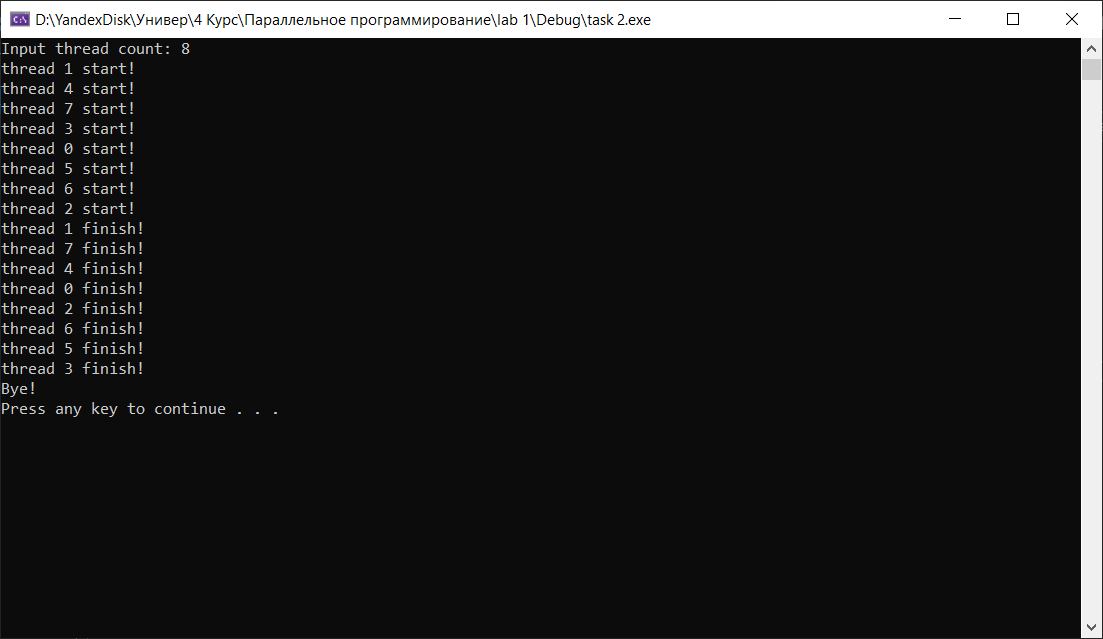
}

# Полученные результаты

Задание 1.

Join привязывает созданный поток к родительскому потоку и ожидает его завершения. Detach отвязывает поток от родительского, что позволяет главному потоку завершиться раньше и закончить выполнение программы.

Задание 2.



Потоки запускаются и завершаются в разное время.

Задание 3-6.

Тестирование проводилось на 4х ядерном и 8 поточном процессоре Intel Core i5 8250U и 8 гб оперативной памяти.

В таблице 1 представлено время выполнения суммы для 1-8 потоков в зависимости от размерности.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размерность** | **Число потоков** | | | |
| **1** | **2** | **4** | **8** |
| **10^7** | 0,0193656 | 0,0127917 | 0,0110878 | 0,0108195 |
| **10^8** | 0,157256 | 0,0938316 | 0,0632353 | 0,0593847 |
| **10^9** | 1,53318 | 0,810256 | 0,519082 | 0,474171 |
| Таблица 1. Время вычисления суммы с определенным числом потоков. | | | | |

В таблице 2 представлено ускорение относительно последовательного выполнения.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Размерность** | **Число потоков** | | | |
| 1 | 2 | 4 | 8 |
| 10^7 | 1,00 | 1,51 | 1,75 | 1,79 |
| 10^8 | 1,00 | 1,68 | 2,49 | 2,65 |
| 10^9 | 1,00 | 1,89 | 2,95 | 3,23 |
| Таблица 2. Ускорение для суммы чисел размерности в зависимости от числа потоков. | | | | |

На Рис. 1 изображен график ускорения.

|  |
| --- |
|  |
| Рис. 1. График ускорения относительно числа потоков. |

Таким образом, распараллеливание дало ускорение более чем в 1.5 раза для двух потоков, 1.75 для четырех потоков и 1.79 для восьми потоков. Наибольший прирост производительности наблюдается при больших размерностях, получая ускорение более чем в 3 раза при использовании всех потоков системы.

# Выводы по лабораторной работе

Распараллеливание позволяет значительно ускорить время работы. В то же время, необходимо учитывать количество физических ядер и потоков процессора. Наиболее заметный прирост производительности наблюдается при использовании всех физических ядер процессора. Гипертрейдинг также увеличивает ускорение, однако прирост не такой большой, как при использовании физических ядер. При разработке приложения необходимо учитывать число имеющихся в системе потоков, чтобы не создавать конкуренции за ресурсы железа. Кроме того, стоит правильно использовать механизмы синхронизации потоков, чтобы достичь наибольшей производительности и не создавать ситуаций взаимных блокировок и простаивания. Так, при распараллеливании суммы интервала стоит ввести отдельную переменную и прибавить значение после выхода из цикла фор, используя мьютекс.