**«Распределенные и параллельные системы»**

**Лабораторная работа 6**

**25.04.2024**

**Уважаемые студенты!**

**Отчет по этому занятию следует представить в двух вариантах:**

**1) в конце занятия, чтобы подтвердить свое присутствие и активность на занятии;**

**2) к следующему занятию в завершенном виде.**

**Параллельные вычисления в С++**

**Независимые подзадачи**

**Использованные источники**

1. Страуструп, Бьярне. Язык программирования С++. Краткий курс, 2-е изд. - СПб.: ООО "Диалектика", 2019.
2. Уильямс Энтони. C++. Практика многопоточного программирования. — СПб.: Питер, 2020.
3. <https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-1-starting-threads.html>

#include <thread>

#include <iostream>

void my\_thread\_func(int& i)

{

std::cout << "I am thread\n";

i++;

}

int main()

{

int i = 5;

my\_thread\_func(i);

std::cout << "i = " << i << std::endl;

auto begin = std::chrono::steady\_clock::now(); // устойчивые часы

std::thread t(my\_thread\_func, std::ref(i));

t.join();

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

auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);

std::cout << "The time: " << elapsed\_ms.count() << " ms\n";

std::cout << "i = " << i << std::endl;

std::cout << "Main thread finishes\n";

}

**Задания**

***Задание 1.***

Вычислить количество простых чисел.

***Отчет:***

- Текст программы:

#include <thread>

#include <iostream>

#include <vector>

Using namespace std;

bool isPrime(int number)

{

if (number < 2)

return false;

for (int i = 2; i\*i <= number; i++)

{

if (number % i == 0)

return false;

}

return true;

}

void countPrimesInRange(int start, int end, int& count)

{

for (int i = start; i <= end; i++)

{

if (isPrime(i))

count++;

}

}

int main()

{

int count = 0;

auto begin = chrono::steady\_clock::now();

thread t1(countPrimesInRange, 1, 100000, ref(count));

thread t2(countPrimesInRange, 100001, 200000, ref(count));

t1.join();

t2.join();

auto end = chrono::steady\_clock::now();

auto elapsed\_ms = chrono::duration\_cast<chrono::milliseconds>(end - begin);

cout << "Number of primes: " << count << endl;

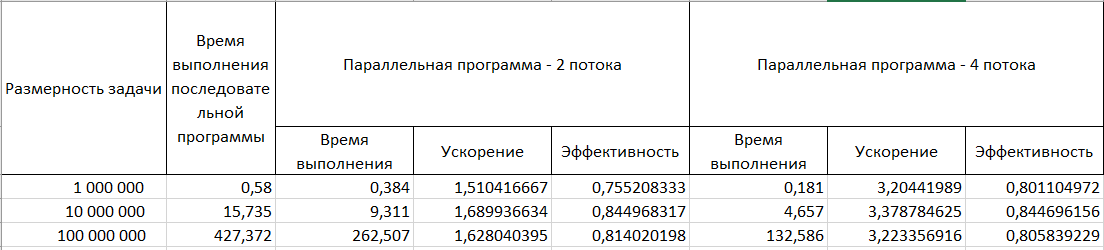
cout << "Time taken: " << elapsed\_ms.count() << " ms" << endl;

return 0;

}

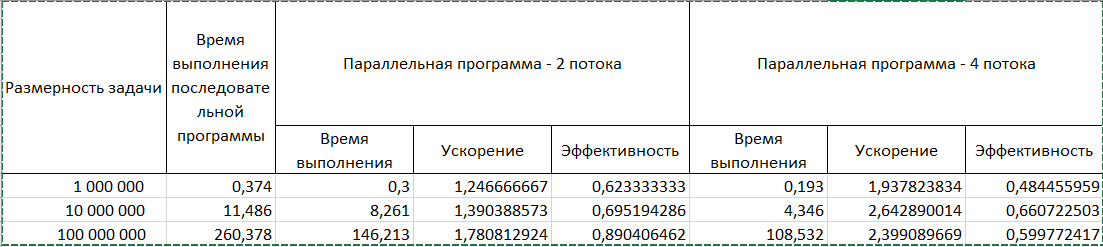
- Таблица с результатами экспериментов

* Таблица с результатами экспериментов



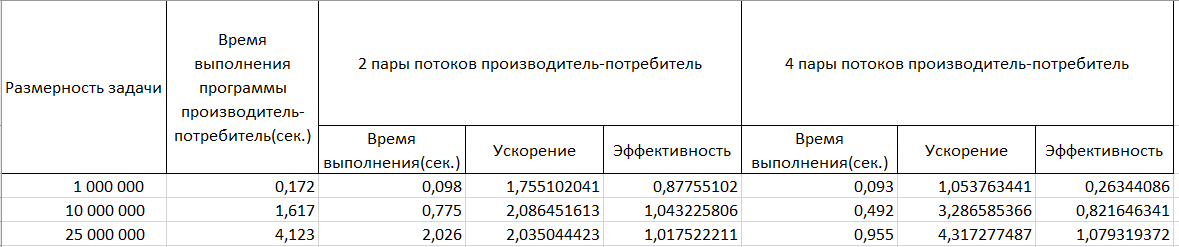
* Сравнение с результатами, полученными для Java

**Результаты на java:**



- С++ считает быстрее

***Задание 2.  
Отчет:***

* Текст программы
* #include <iostream>
* #include <thread>
* #include <ctime>
* const int MAX = 1'000'000;
* void countNumbers(int iter, int& res)
* {
* srand(time(0));
* int in\_circle\_count = 0;
* for (int i = 0; i < iter; ++i)
* {
* double x = ((double)rand() / (double)RAND\_MAX);
* double y = ((double)rand() / (double)RAND\_MAX);
* if (x \* x + y \* y < 1)
* ++in\_circle\_count;
* }
* res = in\_circle\_count;
* }
* class CountPrimeNumbersTask
* {
* public:
* explicit CountPrimeNumbersTask(int \_id, int end\_, int\* res\_) : id(\_id), end(end\_), res(res\_)
* {}
* CountPrimeNumbersTask()
* {}
* void operator()() //const
* {
* countNumbers(end, \*res);
* }
* private:
* int id, end;
* int\* res;
* };
* int main()
* {
* int numberOfThreads;
* std::cout << "Enter number of threads:";
* std::cin >> numberOfThreads;
* auto begin = std::chrono::steady\_clock::now();
* if (numberOfThreads > 1) {
* std::thread\* workers = new std::thread[numberOfThreads];
* int\* results = new int[numberOfThreads];
* int delta = MAX / numberOfThreads;
* for (int i = 0; i < numberOfThreads; i++) {
* workers[i] = std::thread{ CountPrimeNumbersTask(i + 1, delta, &results[i]) };
* }
* for (int i = 0; i < numberOfThreads; i++) {
* workers[i].join();
* }
* int result = 0;
* for (int i = 0; i < numberOfThreads; i++) {
* result += results[i];
* }
* std::cout << "Parallel program: pi = " << double(result) / MAX \* 4 << '\t';
* }
* else {
* int result;
* countNumbers(MAX, result);
* std::cout << "Sequantial program: pi = " << double(result) / MAX \* 4 << '\t';
* }
* auto end = std::chrono::steady\_clock::now();
* auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);
* std::cout << "The time: " << double(elapsed\_ms.count()) / 1000 << " sec\n";
* return 0;
* }
* Таблица с результатами экспериментов
* Сравнение с результатами, полученными для Java

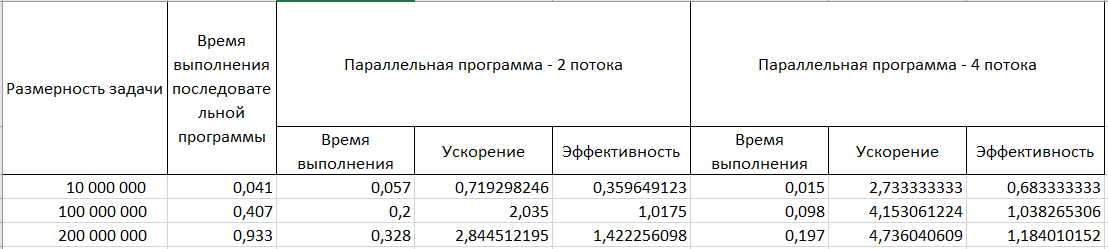
**Результаты на Java:**

**В c++ лучше. При большой размерности задачи это касатеся также и показателей эффективности и усокрения.**

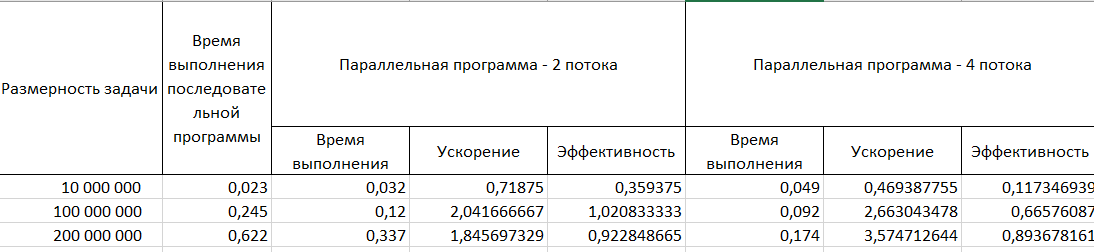
***Задание 3.***

Суммировать элементы одномерного массива.

***Отчет:***

* Текст программы
* #include <iostream>
* #include <thread>
* #include <ctime>
* const int MAX = 200'000'000;
* int array[MAX];
* void SumElements(int begin, int end, int& res)
* {
* int sum = 0;
* for (int i = begin; i < end; ++i)
* {
* sum += array[i];
* }
* res = sum;
* }
* class SumArrayElementsTask
* {
* public:
* explicit SumArrayElementsTask(int \_id,int begin\_, int end\_, int\* res\_) : id(\_id), begin(begin\_), end(end\_), res(res\_)
* {}
* SumArrayElementsTask()
* {}
* void operator()() //const
* {
* SumElements(begin, end, \*res);
* }
* private:
* int id, begin, end;
* int\* res;
* };
* int main()
* {
* for (int i = 0; i < MAX; i++) {
* array[i] = -1000 + (rand() % 2000);
* }
* int numberOfThreads;
* std::cout << "Enter number of threads:";
* std::cin >> numberOfThreads;
* auto begin = std::chrono::steady\_clock::now();
* if (numberOfThreads > 1) {
* std::thread\* workers = new std::thread[numberOfThreads];
* int\* results = new int[numberOfThreads];
* int delta = MAX / numberOfThreads;
* int l = 0;
* for (int i = 0; i < numberOfThreads; i++) {
* workers[i] = std::thread{ SumArrayElementsTask(i + 1, l , l + delta, &results[i]) };
* l += delta;
* }
* for (int i = 0; i < numberOfThreads; i++) {
* workers[i].join();
* }
* int result = 0;
* for (int i = 0; i < numberOfThreads; i++) {
* result += results[i];
* }
* std::cout << "Parallel program: sum of elements of array = " << result<< '\t';
* }
* else {
* int result;
* SumElements(0, MAX, result);
* std::cout << "Sequantial program: sum of elements of array = " << result << '\t';
* }
* auto end = std::chrono::steady\_clock::now();
* auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);
* std::cout << "The time: " << double(elapsed\_ms.count()) / 1000 << " sec\n";
* return 0;
* }
* Таблица с результатами экспериментов
* Сравнение с результатами, полученными для Java

**Результаты на Java:**



***Задание 4.***

Суммировать элементы последовательного контейнера по Энтони Уильямсу [2, Листинг 2.9. Простейшая параллельная версия std::accumulate, стр. 60].

***Отчет:***Текст программы:

template<typename Iterator,typename T>

struct accumulate\_block

{

void operator()(Iterator first,Iterator last,T& result)

{

result=std::accumulate(first,last,result);

}

};

template<typename Iterator,typename T>

T parallel\_accumulate(Iterator first,Iterator last,T init)

{

unsigned long const length=std::distance(first,last);

if(!length)

return init;

unsigned long const min\_per\_thread=25;

unsigned long const max\_threads=

(length+min\_per\_thread-1)/min\_per\_thread;

unsigned long const hardware\_threads=

std::thread::hardware\_concurrency();

unsigned long const num\_threads=

std::min(hardware\_threads!=0?hardware\_threads:2,max\_threads);

unsigned long const block\_size=length/num\_threads;

std::vector<T> results(num\_threads);

std::vector<std::thread> threads(num\_threads-1);

Iterator block\_start=first;

for(unsigned long i=0;i<(num\_threads-1);++i)

{

Iterator block\_end=block\_start;

std::advance(block\_end,block\_size);

threads[i]=std::thread(accumulate\_block<Iterator,T>(),

block\_start,block\_end,std::ref(results[i]));

block\_start=block\_end;

}

accumulate\_block<Iterator,T>()(

block\_start,last,results[num\_threads-1]);

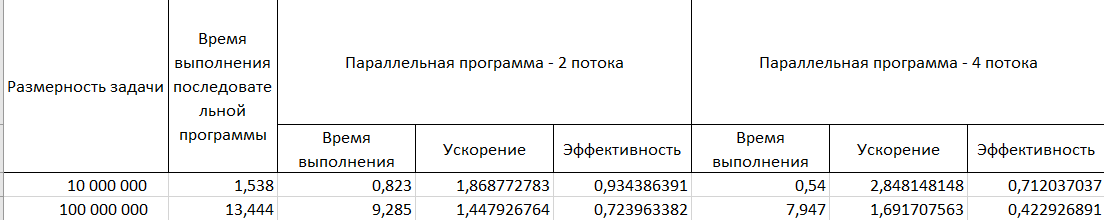
for(auto& entry: threads)

entry.join();

return std::accumulate(results.begin(),results.end(),init);

}

Таблица с результатами экспериментов

* Сравнение с результатами, полученными ранее
* Таблица с результатами экспериментов
* Сравнение с результатами, полученными ранее

**результаты гораздо хуже, чем суммирование элементов одномерного массива.**

***Задание 5.***

Параллельный алгоритм для вычисления определенного интеграла.

• Построить параллельный вычислитель определенного интеграла методом левых прямоугольников. Входные данные (функция и отрезок интегрирования) заданы корректно.

• Провести вычислительные экспериментов для различных функций (3 функции), размерностей задачи (4 размерности), количества потоков (2 потока и 4 потока).

**Требование:** В стиле Параллельного суммирования по Энтони Уильямсу.

***Отчет:***

* Текст программы

#define \_USE\_MATH\_DEFINES

#define NOMINMAX

#include <windows.h>

#include <process.h>

#include <stdio.h>

#include <iostream>

#include <functional>

#include <cmath>

#include <thread>

#include <vector>

#include <limits>

#include <numeric>

double PCFreq = 0.0;

\_\_int64 CounterStart = 0;

void StartCounter()

{

LARGE\_INTEGER li;

if (!QueryPerformanceFrequency(&li))

std::cout << "QueryPerformanceFrequency failed!\n";

PCFreq = double(li.QuadPart) / 1000.0;

QueryPerformanceCounter(&li);

CounterStart = li.QuadPart;

}

double GetCounter()

{

LARGE\_INTEGER li;

QueryPerformanceCounter(&li);

return double(li.QuadPart - CounterStart) / PCFreq;

}

const int g\_n = 100'000;

const double g\_a = 0;

const double g\_b = 7;

double func2(double x)

{

return cos(x);

}

double func1(double x)

{

return x \* x;

}

double func3(double x)

{

return sqrt(x);

}

template<typename T>

struct accumulate\_block

{

void operator()(const size\_t first, const size\_t last, T& result)

{

double h = (g\_b - g\_a) / g\_n;

double sum = 0.0;

double x;

for (size\_t i = first; i < last; ++i) {

x = h \* i;

sum += func1(x);

}

result = h \* sum;

}

};

template<typename T>

T parallel\_accumulate(const size\_t n)

{

unsigned long const length = n;

unsigned long const min\_per\_thread = 25;

unsigned long const max\_threads = (length + min\_per\_thread - 1) / min\_per\_thread;

unsigned long const hardware\_threads = std::thread::hardware\_concurrency();

unsigned long const num\_threads = std::min(max\_threads, (hardware\_threads != 0 ? hardware\_threads : 2));

unsigned long const block\_size = length / num\_threads;

std::cout << "number of threads: " << num\_threads << "\n";

std::vector<T> results(num\_threads);

std::vector<std::thread> threads(num\_threads - 1);

size\_t block\_start = 0;

for (unsigned long i = 0; i < (num\_threads - 1); ++i) {

size\_t block\_end = block\_start;

block\_end += block\_size;

threads[i] = std::thread(

accumulate\_block<T>(),

block\_start, block\_end, std::ref(results[i]));

block\_start = block\_end;

}

accumulate\_block<T>()(block\_start, n, results[num\_threads - 1]);

for (auto& entry : threads) {

entry.join();

}

return std::accumulate(results.begin(), results.end(), 0.0);

}

int main()

{

StartCounter();

double sum = parallel\_accumulate<double>(g\_n);

printf("result = %.16f\n", sum);

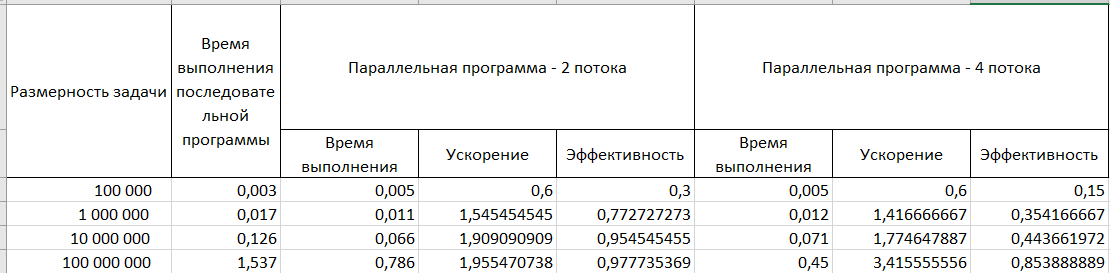
printf("\nTime elapsed: %.16f", GetCounter());

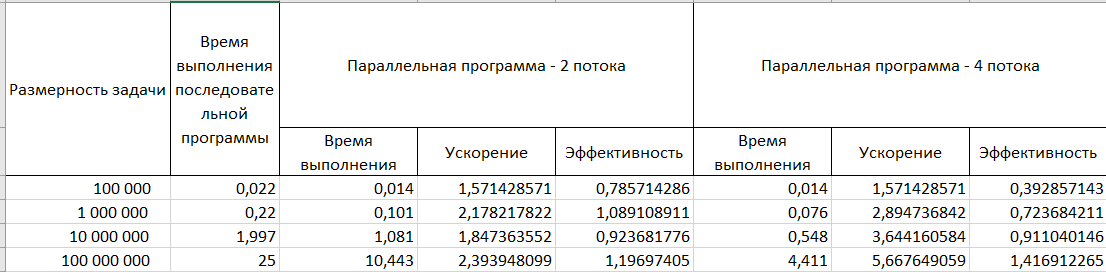
return 0;

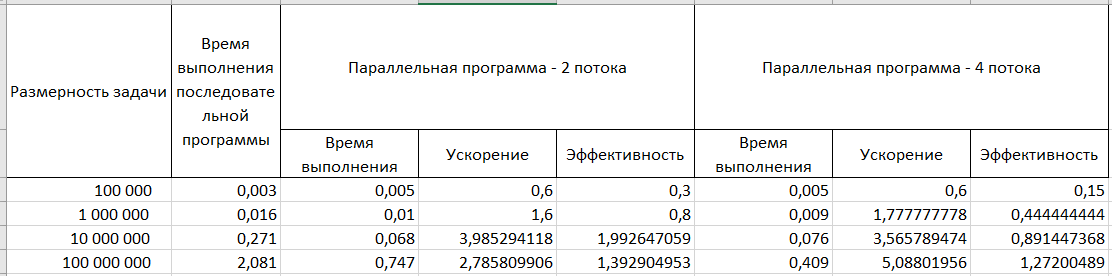
}

* Таблица с результатами экспериментов
* Сравнение с результатами, полученными ранее

Размерность задачи – число прямоугольников

Функция 2x2  на промежутке [1,2]:

Функция 24cos8(x/2) на промежутке [1, π]:

Функция 1/x на промежутке [1,e]:

* Сравнение с результатами, полученными ранее

Результаты показывают, что при увеличении размерности задачи, распараллеливание вычисления определенного интеграла становится все более эффективным. Это сравнение невозможно сделать с предыдущими лабораторными работами, так как вычисления интегралов там не производились