**«Распределенные и параллельные системы»**

**Лабораторная работа 7**

**06.05.2024**

**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++**

**Использованные источники**

1. Галовиц, Яцек. С++17 STL. Стандартная библиотека шаблонов. — СПб.: Питер, 2018.
2. Гримм Р. Параллельное программирование на современном языке C++. – М.: ДМК Пресс, 2022.
3. <https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-8-futures-and-promises.html>
4. <https://ru.cppreference.com/w/cpp/thread/async>

**Задания**

***Задание 1.* [1]**

Автоматическое распараллеливание кода, использующего стандартные алгоритмы.

Ставим эксперименты со стандартными алгоритмами:

• Выбрать 3 алгоритма, которые можно автоматически распараллелить;

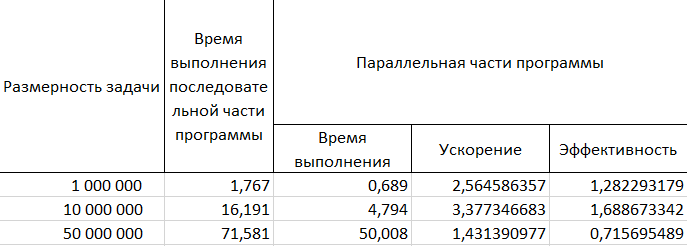
• Придумать «большую» задачу, для решения которой можно применить выбранные алгоритмы;

• Выполнить вычислительные эксперименты и определить эффективность параллельной реализации.

***Подсказка***!!! Взять контейнер большой размерности и обработать его с использованием трех алгоритмов, которые имеют параллельную реализацию.

***Отчет:***

* Текст программы
* #include <iostream>
* #include <vector>
* #include <random>
* #include <execution>
* #include <algorithm>
* #include <thread>
* #include <ctime>
* const int size = 50'000'000;
* bool is\_lucky(int n) {
* return n % 7 == 0;
* }
* int main()
* {
* std::mt19937 gen;
* std::uniform\_int\_distribution<int> dis(0, 100000);
* auto rand\_num([=]() mutable { return dis(gen); });
* std::vector<int> v1(size);
* auto begin = std::chrono::steady\_clock::now();
* std::generate(std::execution::par,v1.begin(), v1.end(), rand\_num);
* std::sort(std::execution::par, v1.begin(), v1.end());
* auto lucky(std::count\_if(std::execution::par, v1.begin(), v1.end(), is\_lucky));
* auto end = std::chrono::steady\_clock::now();
* auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);
* std::cout << "================================\n";
* std::cout << (100.0 \* lucky / v1.size()) << "% of the numbers are lucky.\n";
* std::cout << "The time: " << double(elapsed\_ms.count()) / 1000 << " sec\n";
* std::cout << "================================\n";
* std::vector<int> v2(size);
* auto begin2 = std::chrono::steady\_clock::now();
* std::generate(std::execution::seq, v2.begin(), v2.end(), rand\_num);
* std::sort(std::execution::seq, v2.begin(), v2.end());
* auto lucky2(std::count\_if(std::execution::par, v2.begin(), v2.end(), is\_lucky));
* auto end2 = std::chrono::steady\_clock::now();
* auto elapsed\_ms2 = std::chrono::duration\_cast<std::chrono::milliseconds>(end2 - begin2);
* std::cout << "================================\n";
* std::cout << (100.0 \* lucky2 / v2.size()) << "% of the numbers are lucky.\n";
* std::cout << "The time: " << double(elapsed\_ms2.count()) / 1000 << " sec\n";
* std::cout << "================================\n";
* }
* Таблица с результатами экспериментов



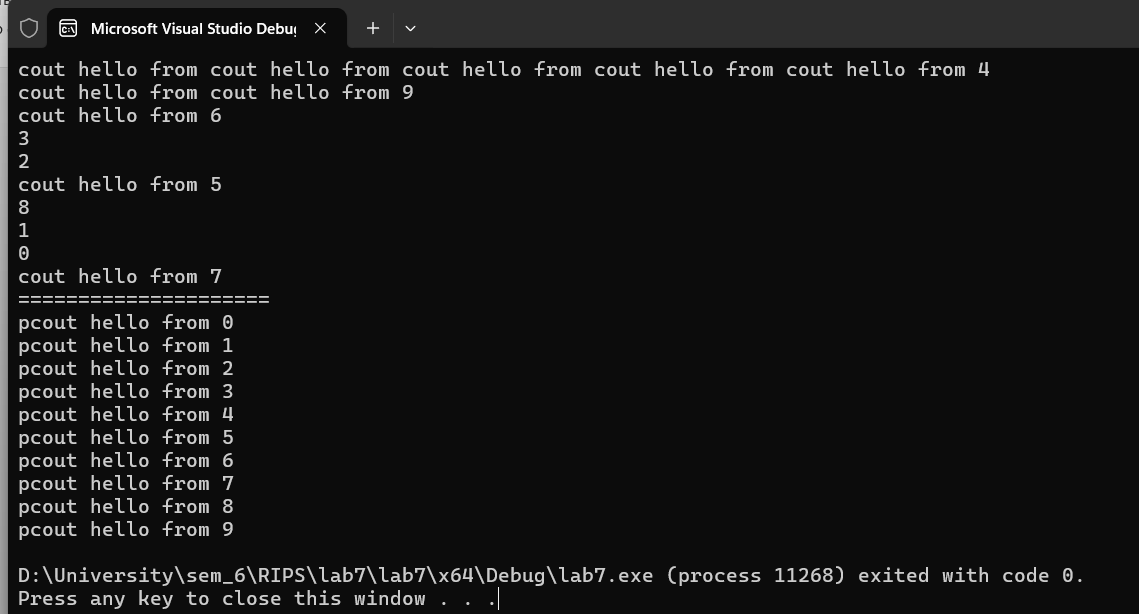
***Задание 2.* [1]**

Синхронизация конкурентного использования std::cout.

• Изучить и подтвердить экспериментами.

***Отчет:***

* Текст программы
* #include <iostream>
* #include <vector>
* #include <thread>
* #include <sstream>
* #include <mutex>
* struct pcout : public std::stringstream {
* static inline std::mutex cout\_mutex;
* ~pcout() {
* std::lock\_guard<std::mutex> l {cout\_mutex};
* std::cout << rdbuf();
* std::cout.flush();
* }
* };
* static void print\_cout(int id)
* {
* std::cout << "cout hello from " << id << '\n';
* }
* static void print\_pcout(int id)
* {
* pcout{} << "pcout hello from " << id << '\n';
* }
* int main()
* {
* std::vector<std::thread> v;
* for (size\_t i{ 0 }; i < 10; ++i) {
* v.emplace\_back(print\_cout, i);
* }
* for (auto& t : v) { t.join(); }
* std::cout << "=====================\n";
* v.clear();
* for (size\_t i{ 0 }; i < 10; ++i) {
* v.emplace\_back(print\_pcout, i);
* }
* for (auto& t : v) { t.join(); }
* }
* Описание экспериментов



**С++ и Java: Асинхронное выполнение задач**

**[2]**

3.9. Асинхронные задания

3.9.1. Отличие заданий от потоков

3.9.2. Функция std::async

3.9.2.1. Политика запуска

3.9.2.2. Запустить и забыть

3.9.2.3. Параллельное вычисление скалярного произведения

**Листинг 4.6. Использование std::future для получения значения, возвращаемого задачей, выполняемой в асинхронном режиме**

#include <future>

#include <iostream>

int find\_the\_answer\_to\_ltuae();

void do\_other\_stuff();

int main()

{

std::future<int> the\_answer=std::async(find\_the\_answer\_to\_ltuae);

do\_other\_stuff();

std::cout<<"The answer is "<<the\_answer.get()<<std::endl;

**}**

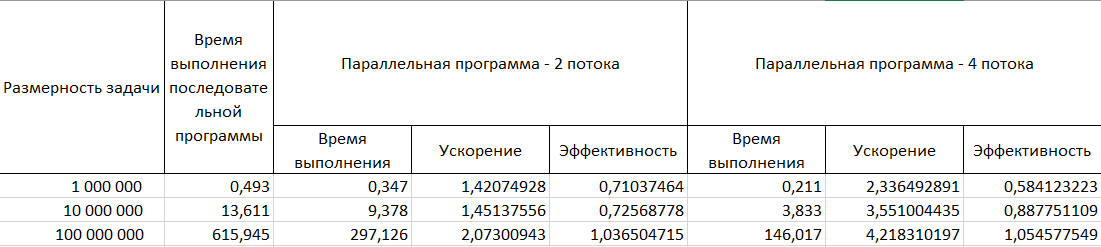
***Задание 3.***

Поиск простых чисел в диапазоне [a, b] (или любая другая задача).

* Использовать std::async и std::future.
* Выполнить вычислительные эксперименты для определения ускорения и эффективности параллельной программы.

***Отчет:***

* Текст программы
* #include <iostream>
* #include <thread>
* #include <future>
* #include <vector>
* const int MAX = 100'000'000;
* bool isPrime(int x) {
* int top = (int)sqrt(x);
* for (int i = 2; i <= top; i++)
* if (x % i == 0)
* return false;
* return true;
* }
* void countNumbers(int min, int max, int& res)
* {
* int count = 0;
* for (int i = min; i <= max; ++i)
* {
* if (isPrime(i))
* ++count;
* }
* res = count;
* }
* int main()
* {
* int numberOfThreads;
* std::cout << "Enter number of threads:";
* std::cin >> numberOfThreads;
* auto begin = std::chrono::steady\_clock::now();
* int delta = (MAX - 2) / numberOfThreads;
* int l = 2;
* int result = 0;
* auto K = [=](int l, int r) {
* int count = 0;
* for (int j = l; j <= r; ++j)
* {
* if (isPrime(j))
* ++count;
* }
* return count;
* };
* std::vector<std::future<int>> VF;
* for (int i = 0; i < numberOfThreads; i ++) {
* int r = l + delta;
* VF.push\_back(std::async(K,l,r));
* l += delta;
* }
* for (int i = 0; i < numberOfThreads; i++)
* result += VF[i].get();
* std::cout << "Parallel program: result = " << 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

****

***Задание 4.***

Изучить способы асинхронного выполнения задач в Java.

Выполнить Задание 3.

***Отчет:***

* Текст программы

**Результат Задания 3 есть выше.**

* Таблица с результатами экспериментов

**С++: Мьютексы**

**[1, стр. 356-366]**

Выполняем устойчивую к исключениям общую блокировку с помощью std::unique\_lock и std::shared\_lock

Избегаем взаимных блокировок с применением std::scoped\_lock

***Задание 5.***

Выполняем все эксперименты.

***Отчет:***

* Выводы по экспериментам

**Вывод**: **в ходе проведенных экспериментов было продемонстрирована техника устойчивой к исключениям общей блокировки с помощью std::unique\_lock и std::shared\_lock. А также возможность с помощью std::scoped\_lock исключить взаимные блокировки.**

***Задание 6.* [3, 4]**

Поиск значения максимального элемента вектора

Многопоточная реализация в двух вариантах:

* Простая реализация параллельной версии std::accumulate [Энтони Уильямс, 2.4.]
* Divide and Conquer

***Отчет:***

* Текст программы

Версия согласно Энтони Уильямсу:

#include <iostream>

#include <thread>

#include <vector>

#include <algorithm>

#include <numeric>

template<typename Iterator, typename T>

struct find\_max\_block

{

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

{

result = \*first;

for (Iterator now = first; now != last; ++now)

if(result < \*now)

result = \*now;

}

};

template<typename Iterator, typename T>

T parallel\_find\_max(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(

find\_max\_block<Iterator, T>(),

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

block\_start = block\_end;

}

find\_max\_block<Iterator, T>()(

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

for (auto& entry : threads)

entry.join();

for (Iterator now = results.begin(); now != results.end(); ++now)

if (init < \*now)

init = \*now;

return init;

}

int main()

{

std::vector<int> v;

for (int i = 0 ; i < 1'000'00; i ++)

v.push\_back(-1000 + (rand() % 2000));

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

int k = parallel\_find\_max(v.begin(), v.end(), 0);

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

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

std::cout << k;

std::cout << "The time: " << double(elapsed\_ms.count()) / 1000 << " sec\n";

}

* Версия согласно Divide and Conquer
* #include <iostream>
* #include <thread>
* #include <vector>
* #include <algorithm>
* #include <numeric>
* int maxsimum(std::vector<int> v, int l, int r) {
* if (l == r)
* return v[l];
* int m = (l + r) / 2;
* int u = maxsimum(v,l, m);
* int c = maxsimum(v, l +1, r);
* return u > c ? u : c;
* }
* template<typename T>
* struct find\_max\_block
* {
* void operator()(std::vector<int> v, int first, int last, T& result)
* {
* result = maxsimum(v,first, last);
* }
* };
* template<typename T>
* T parallel\_find\_max(std::vector<int> v, int first, int last, T init)
* {
* unsigned long const length = last - first;
* if (first == last)
* 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);
* int block\_start = first;
* for (unsigned long i = 0; i < (num\_threads - 1); ++i)
* {
* int block\_end = block\_start;
* block\_end += block\_size;
* threads[i] = std::thread(
* find\_max\_block<T>(),
* v, block\_start, block\_end, std::ref(results[i]));
* block\_start = block\_end;
* }
* find\_max\_block<T>()(v, block\_start, last, results[num\_threads - 1]);
* for (auto& entry : threads)
* entry.join();
* for (int i = 0 ; i < results.size(); i ++)
* if (results[i] > init)
* init = results[i];
* return init;
* }
* int main()
* {
* std::vector<int> v;
* for (int i = 0; i < 1'000'00; i++)
* v.push\_back(-1000 + (rand() % 2000));
* auto begin = std::chrono::steady\_clock::now();
* int k = parallel\_find\_max(v,0, v.size()-1, 0);
* auto end = std::chrono::steady\_clock::now();
* auto elapsed\_ms = std::chrono::duration\_cast<std::chrono::milliseconds>(end - begin);
* std::cout << k;
* std::cout << "The time: " << double(elapsed\_ms.count()) / 1000 << " sec\n";
* }
* Таблица с результатами экспериментов

