Rubius
Academy

Курс-интенсив

Программирование на С++

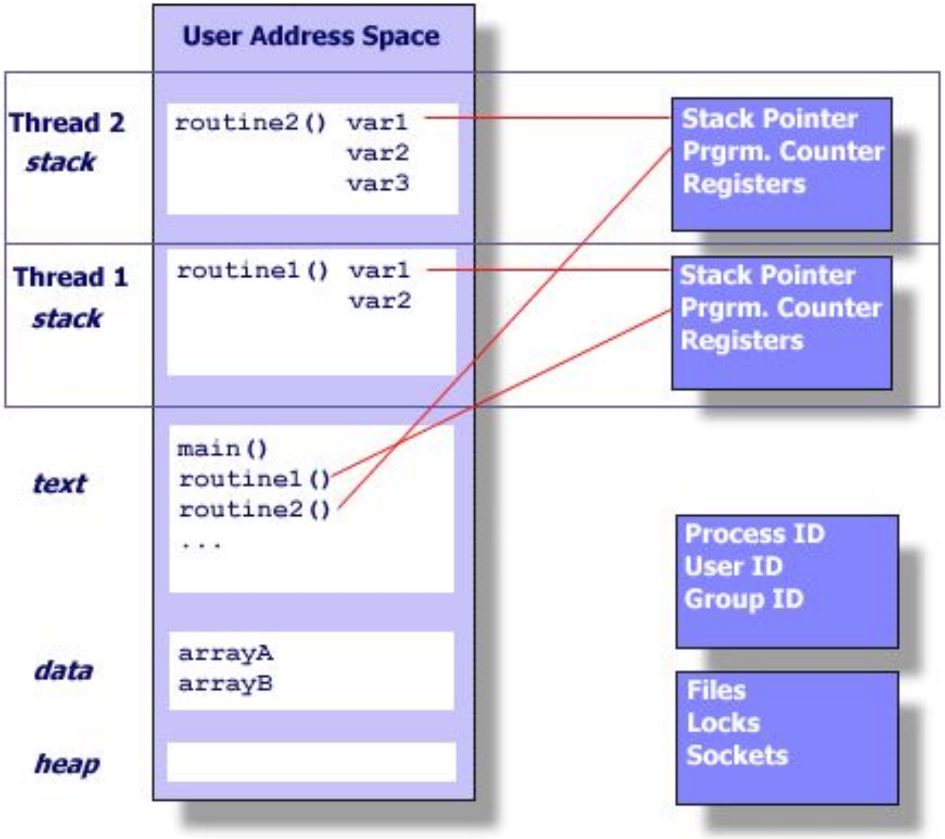
Параллельный С++
Часть 1

academy.rubius.com
sergey@prohanov.com
Сергей Проханов

Потоки (threads)

- Posix threads
- WInAPI threads

- boost
- Qt
- OpenMP



POSIX Threads

```
#include <iostream>
#include <pthread.h>
constexpr int NUM_THREADS=5;

void *PrintHello(void *threadid) {
  long tid;
  tid = static_cast<long>(reinterpret_cast<uintptr_t>(threadid));
  std::cout << "Hello World! Thread ID, " << tid << "\n";
  pthread_exit(nullptr);
  return nullptr;
}

int main () {
  pthread_t threads[NUM_THREADS];
  int rc;
  int i;

  for( i = 0; i < NUM_THREADS; i++ ) {
     std::cout << "main() : creating thread, " << i << "\n";
     rc = pthread_create(&threads[i], nullptr, PrintHello, reinterpret_cast<void *>(static_cast<intptr_t>(i)));

     if (rc) {
        std::cout << "Error:unable to create thread," << rc << "\n";
        exit(-1);
     }
  }
  pthread_exit(nullptr);
}
```

```
main() : creating thread, 0
main() : creating thread, 1
main() : creating thread, 2
main() : creating thread, 3
Hello World! Thread ID, 0
main() : creating thread, 4
Hello World! Thread ID, 2
Hello World! Thread ID, 1
Hello World! Thread ID, 3
Hello World! Thread ID, 4
```

WinAPI Threads

```cpp
#include <Windows.h>
#include <iostream>

constexpr int NUM_THREADS = 10;

DWORD WINAPI mythread(__in LPVOID lpParameter)
{
    std::cout << "Hello World! Thread ID, " << GetCurrentThreadId() << "\n";
    return 0;
}

int main()
{
    HANDLE myhandle[NUM_THREADS];
    DWORD mythreadid[NUM_THREADS];

    for (int i = 0; i < NUM_THREADS; i++) {
        myhandle[i] = CreateThread(0, 0, mythread, 0, 0, &mythreadid[i]);

        if (myhandle[i] == nullptr) {
            std::cout << "Error:unable to create thread," << "\n";
            exit(-1);
        }
    }
    return 0;
}
```

```
Hello World! Thread ID, 7352
Hello World! Thread ID, 19212
Hello World! Thread ID, 18828
Hello World! Thread ID, 13900
Hello World! Thread ID, 1232
```

std::thread

```
#include <iostream>
#include <thread>

constexpr int num_threads = 5;


void call_from_thread(int tid) {
        std::cout << "Hello World! Thread ID, " << tid << "\n";
}

int main()
{
      std::thread t[num_threads];

      for (int i = 0; i < num_threads; ++i) {
            t[i] = std::thread(call_from_thread, i);
      }

      for (int i = 0; i < num_threads; ++i) {
          t[i].join();
      }

      return 0;
}
```

```
Hello World! Thread ID, 0
Hello World! Thread ID, 1
Hello World! Thread ID, 2
Hello World! Thread ID, 4
Hello World! Thread ID, 3
```

std::thread

```cpp
#include <iostream>
#include <thread>

void threadfunc0(int value)
{
	std::cout << "in thread func0: " << value << "\n";
}
void threadfunc(int &value)
{
	std::cout << "in thread func: " << value++ << "\n";
}

int main()
{
	int value = 42;
	std::thread t0 { threadfunc0, 333 };
	std::thread t1{ threadfunc, std::ref(value) };
	t0.join();
	t1.join();
	std::thread t2{ [](int &val) {
	std::cout << "in thread func: " << val++ << "\n";
	}, std::ref(value) };
	t2.join();
	std::thread t3{ [&]() {
	std::cout << "in thread func: " << value++ << "\n";
	}};
	t3.join();
	std::cout << "in main thread: " << value << "\n";
	return 0;
}
```

```
in thread func0: 333
in thread func: 42
in thread func: 43
in thread func: 44
in main thread: 45
```

std::thread

```
#include <iostream>
#include <thread>
#include <chrono>

template<typename T>
void threadfunc()
{
        std::cout << std::this_thread::get_id() << " " << "Type is: " << typeid(T).name() << "\n";
}

int main()
{
        std::thread t1{ threadfunc<int> };
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::thread t2{ threadfunc<double> };
        t1.join();
        t2.join();
        return 0;
}
```

```
9468 Type is: int
18332 Type is: double
```

- `std::this_thread::yield();`
- `std::this_thread::sleep_until();`

Race Condition

```
#include<iostream>
#include<string>
#include<thread>

void threadfunc()
{
      for (int i = 0; i > -10; i--)
            std::cout << "Thread i= " << i << "\n";
}

int main()
{
 std::thread t1{ threadfunc };

 for (int i = 0; i < 10; i++)
       std::cout << "Thread i= " << i << "\n";

 t1.join();

}
```

```
Thread i= Thread i= 0
Thread i= 1
Thread i= 0
Thread i= -1
Thread i= -2
Thread i= -3
Thread i= -4
2
Thread i= 3
Thread i= 4
Thread i= 5
Thread i= 6
Thread i= 7
Thread i= 8
Thread i= 9
Thread i= -5
Thread i= -6
Thread i= -7
Thread i= -8
Thread i= -9
```

Data Race

```cpp
#include <iostream>
#include <vector>
#include <thread>
class Wallet
{
    int money;
public:
    Wallet() : money{0} {};
    int getMoney() { return money; }

    void addMoney(int countmoney)
    {
        for (int i = 0; i < countmoney; i++)
        { money++; }
    }
};

int main()
{
    Wallet wallet;
    std::vector<std::thread> threads;

    for (int i = 0; i < 10; i++) {
        threads.push_back(std::thread(&Wallet::addMoney, &wallet, 100000));
    }

    for (int i = 0; i < threads.size(); i++)
    {
        threads[i].join();
    }

    std::cout << " Money in Wallet = " << wallet.getMoney() << "\n";
}
```

```
Money in Wallet = 963039
Money in Wallet = 876812
Money in Wallet = 1000000
```

mutex

```
#include<iostream>
#include<string>
#include<thread>
#include<mutex>

void threadfunc(std::mutex &mtx)
{
	mtx.lock();
	for (int i = 0; i > -10; i--)
	{
		std::cout << "Thread i= " << i << "\n";
	}
	mtx.unlock();
}

int main()
{
	std::mutex mtx;
	std::thread t1{ threadfunc, ref(mtx) };

	mtx.lock();
	for (int i = 0; i < 10; i++)
	{
		std::cout << "Thread i= " << i << "\n";
	}
	mtx.unlock();
	t1.join();
	return 0;
}
```

```
Thread i= 0
Thread i= -1
Thread i= -2
Thread i= -3
Thread i= -4
Thread i= -5
Thread i= -6
Thread i= -7
Thread i= -8
Thread i= -9
Thread i= 0
Thread i= 1
Thread i= 2
Thread i= 3
Thread i= 4
Thread i= 5
Thread i= 6
Thread i= 7
Thread i= 8
Thread i= 9
```

mutex

```
class Wallet
{
    int money;
    std::mutex mtx;
public:
    Wallet() : money{0} {};
    int getMoney() { return money; }
    void addMoney(int countmoney)
    {
        mtx.lock();
        for (int i = 0; i < countmoney; i++)
        { money++;}
        mtx.unlock();
    }
};
int main()
{
    Wallet wallet;
    std::vector<std::thread> threads;
    for (int i = 0; i < 10; i++) {
        threads.push_back(std::thread(&Wallet::addMoney, &wallet, 100000));
    }
    for (int i = 0; i < threads.size(); i++)    {
        threads[i].join();
    }
    std::cout << " Money in Wallet = " << wallet.getMoney() << std::endl;
}
```

```
Money in Wallet = 1000000
Money in Wallet = 1000000
Money in Wallet = 1000000
```

std::mutex

```
std::chrono::milliseconds interval(100);
std::mutex mutex;
int job_shared = 0;
int job_exclusive = 0;
void job_1()
{
   std::this_thread::sleep_for(interval);
   while (true) {
       if (mutex.try_lock()) {
           std::cout << "job shared (" << job_shared << ")\n";
           mutex.unlock();
           return;
       } else {
           ++job_exclusive;
           std::cout << "job exclusive (" << job_exclusive << ")\n";
           std::this_thread::sleep_for(interval);
       }
   }
}
void job_2()
{
   mutex.lock();
   std::this_thread::sleep_for(5 * interval);
   ++job_shared;
   mutex.unlock();
}
```

```
job exclusive (1)
job exclusive (2)
job exclusive (3)
job exclusive (4)
job shared (1)
```

```
int main()
{
   std::thread thread_1(job_1);
   std::thread thread_2(job_2);
   thread_1.join();
   thread_2.join();
}
```

std::mutex

```cpp
#include <iostream>
#include <thread>
#include <mutex>
#include <chrono>

void threadfunc(std::mutex &mtx)
{
	std::lock_guard<std::mutex> lock(mtx);
	std::cout << "Mutex locked in threadfunc" << "\n";
	std::this_thread::sleep_for(std::chrono::seconds(5));
}

int main()
{
	std::mutex mtx;
	std::thread th { threadfunc, std::ref(mtx)};
	std::this_thread::sleep_for(std::chrono::seconds(2));
	std::unique_lock<std::mutex> lock(mtx);
	std::cout << "Main Thread..." << "\n";
	th.join();
	return 0;
}
```

DeadLock

```
struct Calc {
   std::mutex mutex;
   int i;
   Calc() : i(0) {}
   void mul(int x){
       std::lock_guard<std::mutex> lock(mutex);
       i *= x;
   }
   void div(int x){
       std::lock_guard<std::mutex> lock(mutex);
       i /= x;
   }
   //!!!
   void both(int x, int y){
       std::lock_guard<std::mutex> lock(mutex);
       mul(x);
       div(y);
   }
};

int main()
{
 Calc calc;
 calc.both(42,42);
 return 0;
}
```

std::recursive_mutex

```
struct Calc {
   std::recursive_mutex mutex;
   int i;
   Calc() : i(1) {}
   void mul(int x){
       std::lock_guard<std::recursive_mutex> lock(mutex);
       i *= x;
   }
   void div(int x){
       std::lock_guard<std::recursive_mutex> lock(mutex);
       i /= x;
   }
   void both(int x, int y){
       std::lock_guard<std::recursive_mutex> lock(mutex);
       mul(x);
       div(y);
   }
};

int main()
{
 Calc calc;
 calc.both(42,3);
 return 0;
}
```

std::mutex

```cpp
void work(std::timed_mutex &mutex){
   std::chrono::milliseconds timeout(100);
   while(true){
       if(mutex.try_lock_for(timeout)){
           std::cout << std::this_thread::get_id() << ": do work with the mutex" << "\n";
           std::chrono::milliseconds sleepDuration(250);
           std::this_thread::sleep_for(sleepDuration);
           mutex.unlock();
           std::this_thread::sleep_for(sleepDuration);
       } else {
           std::cout << std::this_thread::get_id() << ": do work without mutex" << "\n";
           std::chrono::milliseconds sleepDuration(100);
           std::this_thread::sleep_for(sleepDuration);
       }
   }
}
int main(){
   std::timed_mutex mtx;
   std::thread t1(work,ref(mtx));
   std::thread t2(work,ref(mtx));

   t1.join();
   t2.join();
   return 0;
}
```

```
2: do work with the mutex
3: do work without mutex
3: do work with the mutex
...
```

std::shared_mutex

```cpp
class MyData {
    std::vector<double> data_;
    mutable shared_mutex mtx;
public:
    void write() {
        unique_lock<shared_mutex> lk(mtx);
        // ... write to data_ ...
    }
    void read() const {
        shared_lock<shared_mutex> lk(mtx);
        // ... read the data ...
    }
};
// --- main program ---
MyData a;
std::thread t_write([&](){
    a.write();
    sleep_for_a_while();
});
std::thread t_read1([&](){
    a.read();
});
std::thread t_read2([&](){
    a.read();
});
```

std::shared_mutex

```
class dns_cache
{
   std::map<std::string, dns_entry> entries;
   mutable std::shared_mutex entry_mutex;

public:

   dns_entry find_entry(std::string const& domain) const
   {
      shared_lock<shared_mutex> lk(entry_mutex);
      std::map<std::string, dns_entry>::const_iterator const it = entries.find(domain);
      return (it == entries.end()) ? dns_entry() : it->second;
   }
   void update_or_add_entry(std::string const& domain,
      dns_entry const& dns_details)
   {
      std::lock_guard<shared_mutex> lk(entry_mutex);
      entries[domain] = dns_details;
   }
};
```

Parallel STL

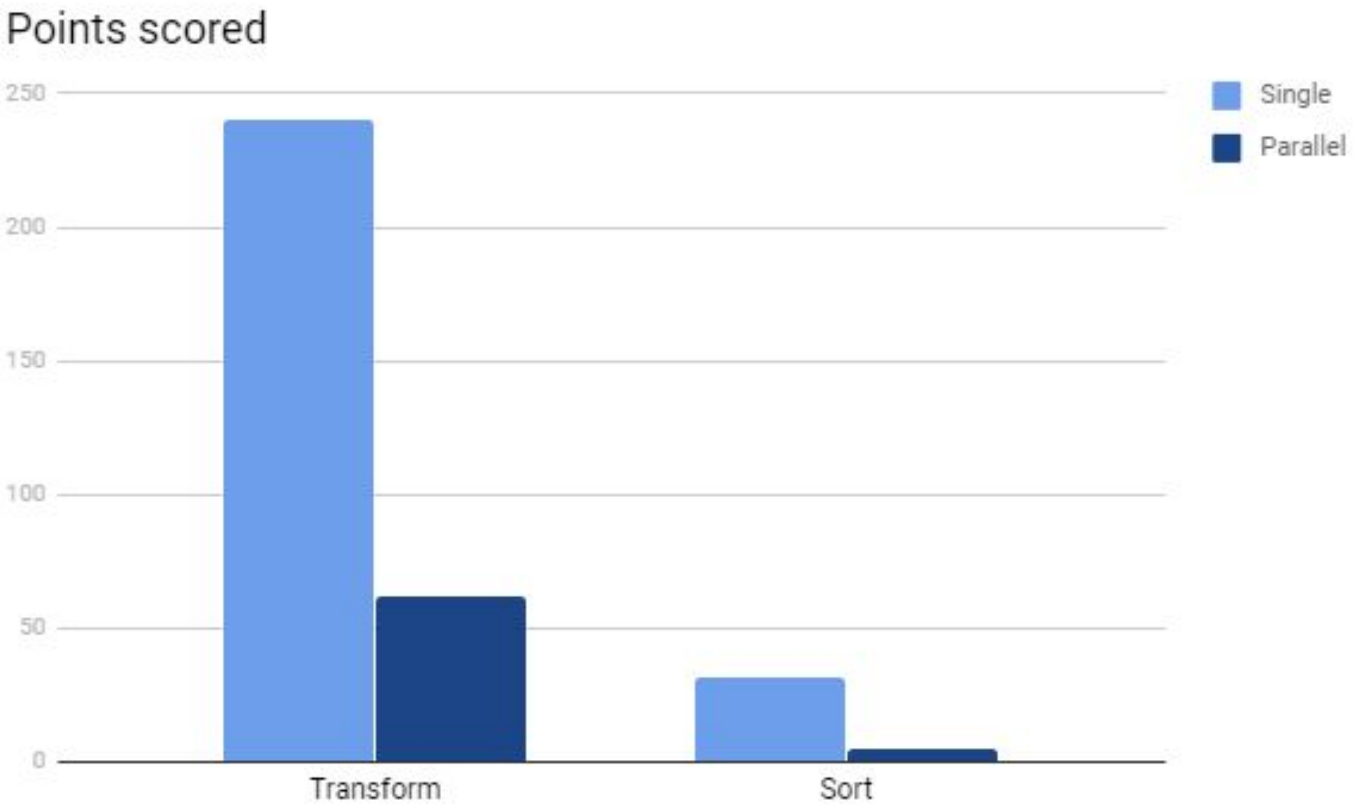
```
sequential_execution_policy  (seq)
parallel_execution_policy (par)
parallel_vector_execution_policy (par_unseq/par_vec)
```

Parallel STL

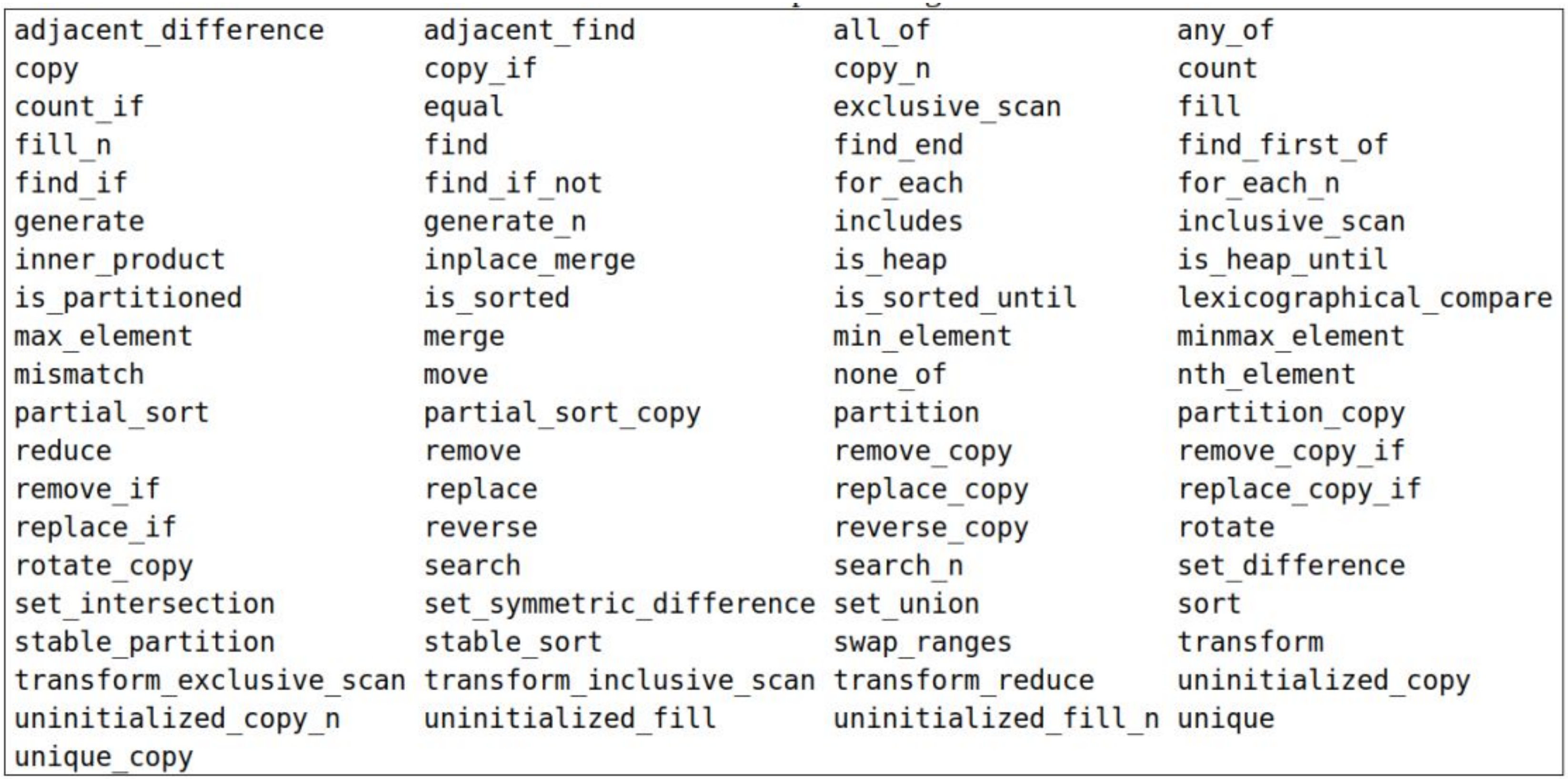
```
#include <iostream>
#include <algorithm>
#include <random>
#include <execution>
#include <chrono>
int main()
{
      constexpr int n = 10000000;
      std::vector<int> a(n);
      std::vector<int> b(n);
      std::default_random_engine generator;
      std::uniform_int_distribution<int> distribution(1, n);
      std::generate(std::execution::seq, a.begin(), a.end(), [&]() { return distribution(generator); });


     {
           auto t1 = std::chrono::high_resolution_clock::now();
           std::transform(std::execution::par,a.begin(), a.end(), b.begin(), [](int f) { return f + 3; });
           auto t2 = std::chrono::high_resolution_clock::now();
           auto duration = std::chrono::duration_cast<std::chrono::milliseconds>(t2 - t1).count();
           std::cout << duration << " " << "\n";
     }

     {
           auto t1 = std::chrono::high_resolution_clock::now();
           std::sort(std::execution::par_unseq,a.begin(), a.end());
           auto t2 = std::chrono::high_resolution_clock::now();
           auto duration = std::chrono::duration_cast<std::chrono::seconds>(t2 - t1).count();
           std::cout << duration << " " << "\n";
     }
```

Parallel STL

Результат работы:



Parallel STL



adjacent_difference	adjacent_find	all_of	any_of
copy	copy_if	copy_n	count
count_if	equal	exclusive_scan	fill
fill_n	find	find_end	find_first_of
find_if	find_if_not	for_each	for_each_n
generate	generate_n	includes	inclusive_scan
inner_product	inplace_merge	is_heap	is_heap_until
is_partitioned	is_sorted	is_sorted_until	lexicographical_compare
max_element	merge	min_element	minmax_element
mismatch	move	none_of	nth_element
partial_sort	partial_sort_copy	partition	partition_copy
reduce	remove	remove_copy	remove_copy_if
remove_if	replace	replace_copy	replace_copy_if
replace_if	reverse	reverse_copy	rotate
rotate_copy	search	search_n	set_difference
set_intersection	set_symmetric_difference	set_union	sort
stable_partition	stable_sort	swap_ranges	transform
transform_exclusive_scan	transform_inclusive_scan	transform_reduce	uninitialized_copy
uninitialized_copy_n	uninitialized_fill	uninitialized_fill_n	unique
unique_copy			

https://en.cppreference.com/w/cpp/experimental/parallelism/existing

Parallel STL

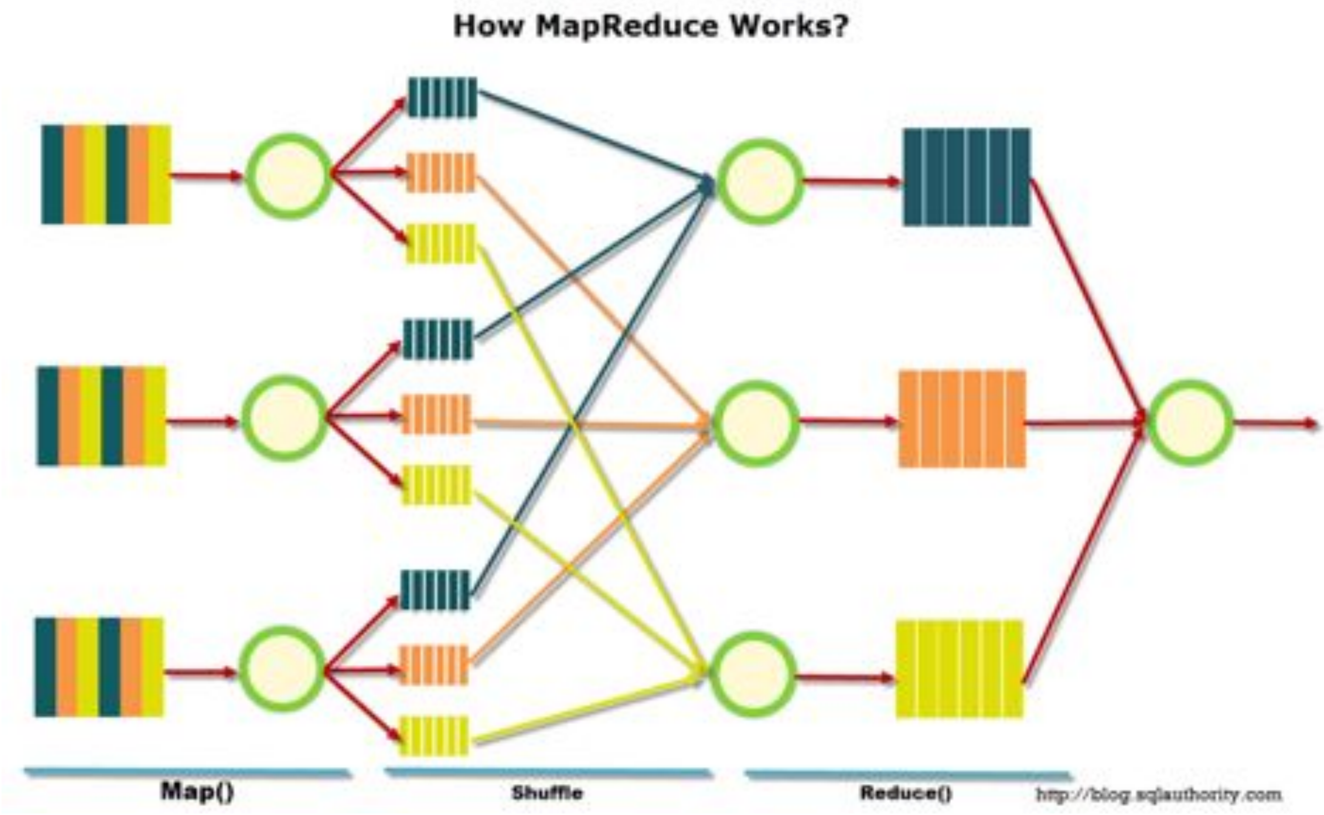
```
before: 135478 835009 968868 221035 308168 547221 ...
after:  0.858683 0.870999 0.999417 0.154167 0.680315 ...
sum: 145.351 sum2= 145.351
```

```
int main()
{
	constexpr int n = 1000000;
	constexpr int print_n = 10;
	double sum = 0;
	double sum2 = 0;
	std::mutex mtx;
	std::vector<double> nums(n);
	std::default_random_engine generator;
	std::uniform_real_distribution<double> distribution(1, n);
	std::generate(std::execution::seq, nums.begin(), nums.end(), [&]() { return distribution(generator); })
	auto print = [](const double& n) { std::cout << " " << n; };
	std::cout << "before:";
	std::for_each_n(nums.begin(), print_n, print);
	std::cout << '\n';
	std::for_each(std::execution::par,nums.begin(), nums.end(), [](double &n) { n=sin(n*n*n); });
	std::for_each(std::execution::par,nums.begin(), nums.end(), [&](double i) {
	   std::lock_guard<std::mutex> guard(mtx); sum += i; });
	std::for_each(nums.begin(), nums.end(), [&](double i) { sum2 += i; });
	std::cout << "after: ";
	std::for_each_n(nums.begin(), print_n, print);
	std::cout << '\n';
	std::cout << "sum: " << sum << " sum2= " << sum2 << '\n';
}
```

MapReduce

```
#include <functional>
```

```
transform_reduce
```



```
template<class ExecutionPolicy,class ForwardIt1, class ForwardIt2, class T>
T transform_reduce(ExecutionPolicy&& policy,ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2, T init);

template<class ExecutionPolicy,class ForwardIt1, class ForwardIt2, class T, class BinaryOp1, class BinaryOp2>
T transform_reduce(ExecutionPolicy&& policy,  ForwardIt1 first1, ForwardIt1 last1, ForwardIt2 first2,
                   T init, BinaryOp1 binary_op1, BinaryOp2 binary_op2);

template<class ExecutionPolicy,class ForwardIt, class T, class BinaryOp, class UnaryOp>
T transform_reduce(ExecutionPolicy&& policy, ForwardIt first, ForwardIt last,
                   T init, BinaryOp binary_op, UnaryOp unary_op);
```

MapReduce

X - вектор
L2norm = sqrt(x[0]*x[0]+x[1]*x[1]+...)

```cpp
#include <iostream>
#include <vector>
#include <cmath>
#include <execution>
#include <functional>
int main()
{
     std::vector<double> x = { 1.0, 2.0, 3.0, 4.4, 5.6, 10.0 };

auto norm = std::sqrt(std::transform_reduce(
        //policy
         std::execution::par_unseq,
         x.begin(), x.end(),
         0.0,
         // Binary reduction op.
         [](double xl, double xr) { return xl + xr; },
         // Unary transform op.
         [](double x) { return x * x; }
     )
);
std::cout << "L2 norma of x = " << norm << "\n";
return 0;
}
```

MapReduce

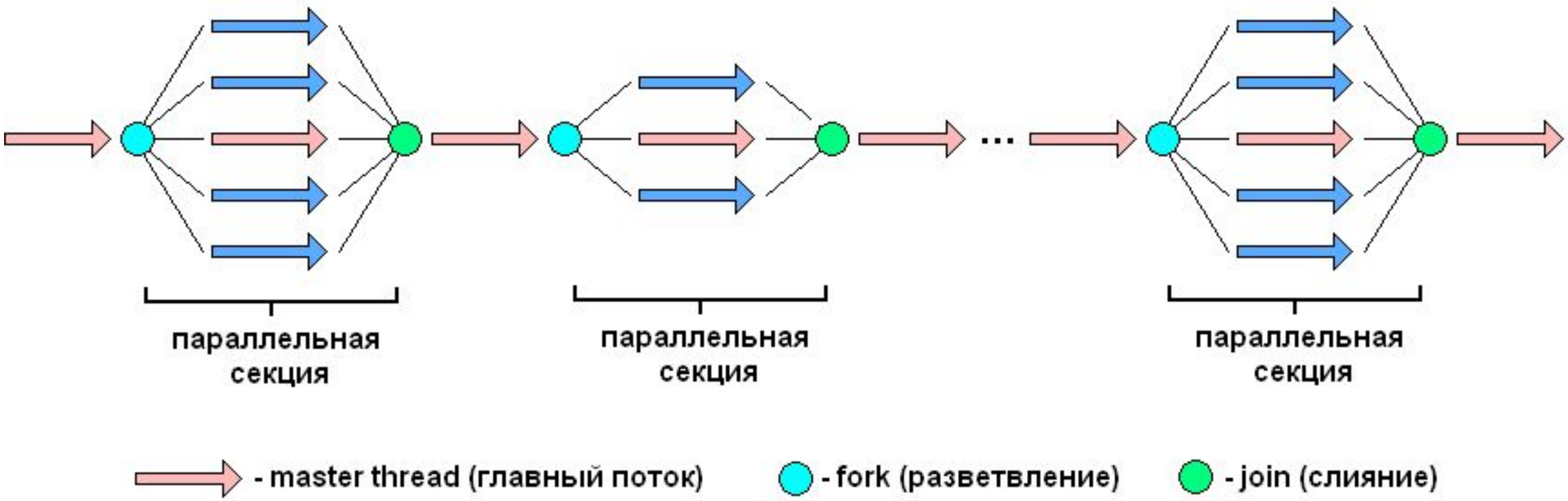
X,Y - векторы
Mult = x[0]*y[0]+x[1]*y[1]+...

```
#include <iostream>
#include <vector>
#include <cmath>
#include <execution>
#include <functional>

int main()
{
     std::vector<double> x = { 1.0, 2.0, 3.0, 4.4, 5.6, 10.0 };
     std::vector<double> y = { 3.3, 42.0, 18.1, 56.2, 2.3, 70.7 };

auto mult = std::transform_reduce         (
         std::execution::par_unseq,
         x.begin(), x.end(),
         y.begin(),
         0.0,
         [](double x, double y) { return x + y; },
         [](double x, double y) { return x * y; }
);
std::cout << "Mult = " << mult << "\n";
return 0;
}
```

OpenMP

```
#include <omp.h>
```

OpenMP
Enabling HPC since 1997



OpenMP

```
#include <iostream>
#include <omp.h>

int main()
{
     omp_set_num_threads(2);

     std::cout << "Sequential area" << "\n";

#pragma omp parallel num_threads(3)
     {
          std::cout << "Parallel area 1" << "\n";
     }
#pragma omp parallel
     {
          std::cout << "Parallel area 2" << "\n";
     }
return 0;
}
```

```
Sequential area
Parallel area 1
Parallel area 1
Parallel area 1
Parallel area 2
Parallel area 2
```

OpenMP

```
#pragma omp parallel [опция[[,] опция]...]

If (условие)
num_threads (целочисленное выражение)
default(private|firstprivate|shared|none)
private(список)
firstprivate(список)
shared(список)
copyin(список)

reduction(оператор:список)
оператор это:  -, +, *, -, &, |, ^, &&, ||
```

OpenMP

```
#pragma omp for [опция [[,] опция]...]
```

- private(список)
- firstprivate(список)
- lastprivate(список)
- reduction(оператор: список)
- schedule(type[, chunk])
- collapse(n)
- ordered
- nowaitz

schedule type:

- static
- dynamic
- guided
- auto
- runtime

OpenMP

```cpp
#include<omp.h>
#include<iostream>
#include<cmath>
#include<chrono>
double f(double x)
{
    return sin(x);
}
int main()
{
    constexpr int n = 10000000;
    constexpr double h = (1.0 - 0.0) / n;
    double t1 = 0, t2 = 0;
    double sum = 0;
    t1 = omp_get_wtime();
    auto beg = std::chrono::high_resolution_clock::now();
#pragma omp parallel for num_threads(4) reduction(+:sum) schedule(dynamic)
    for (int i = 0; i < n; i++)    {
      sum += h * f(i*h);
    }
    auto end = std::chrono::high_resolution_clock::now();
    t2 = omp_get_wtime() - t1;
    auto dur = std::chrono::duration_cast<std::chrono::microseconds>(end - beg);
    std::cout << "Time= " << t2 << " " << dur.count() << " Value= " << sum << "\n";
    return 0;
}
```

Курс-интенсив

Программирование на С++

Параллельный С++
Часть 1

academy.rubius.com

sergey@prohanov.com

Сергей Проханов