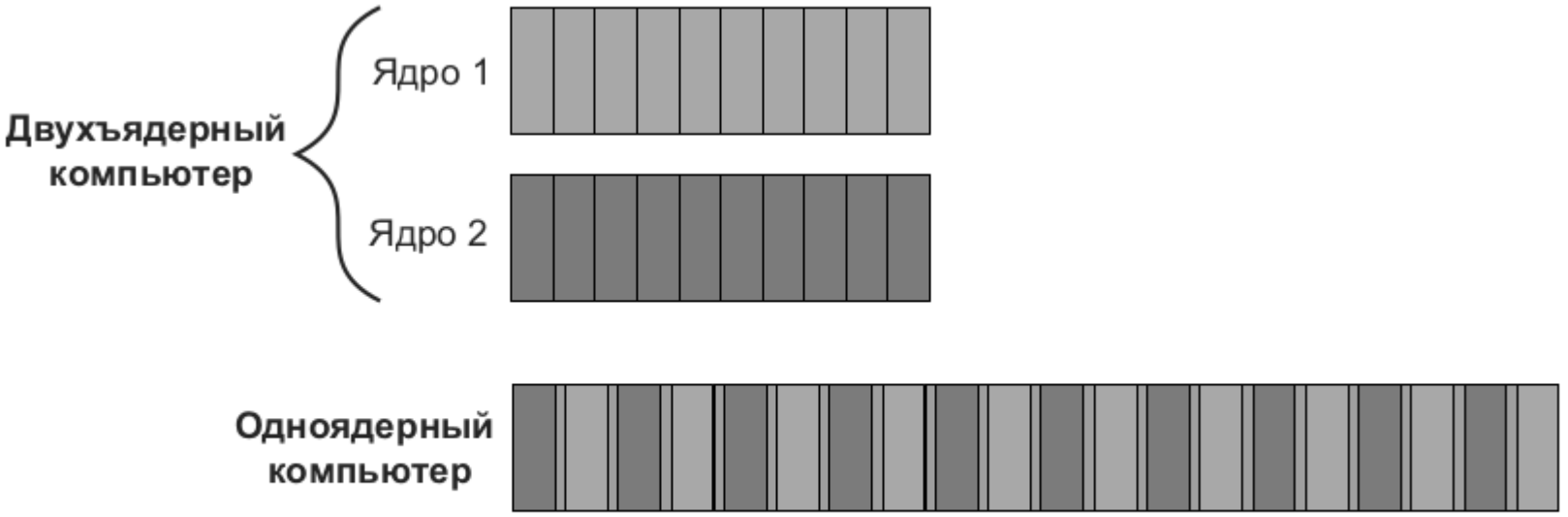
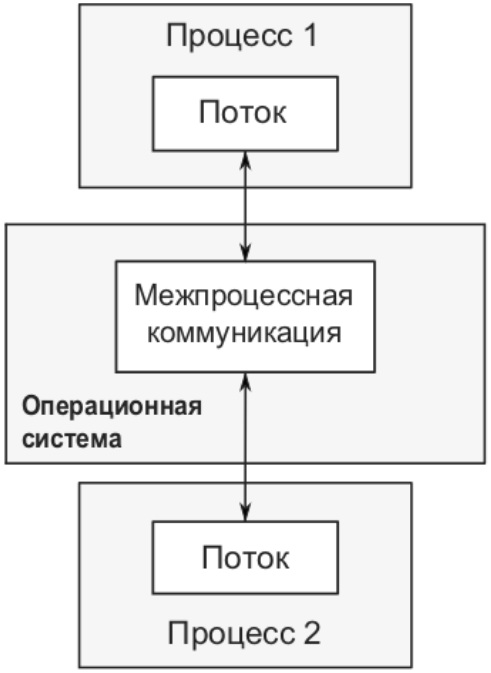
Два подхода к параллелизму

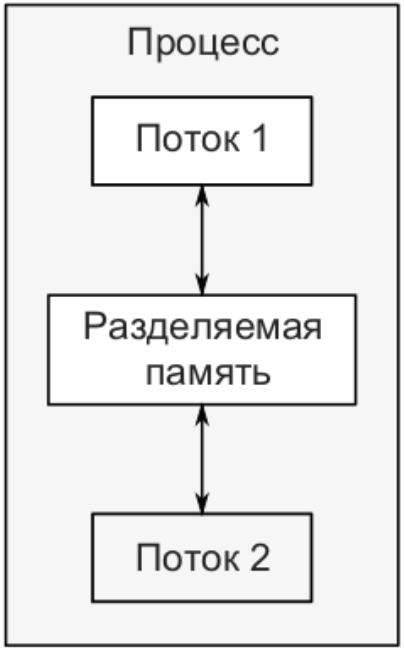


Два подхода к параллелизму: параллельное выполнение на двухъядерном компьютере и переключение задач на одноядерном

Подходы к организации параллелизма

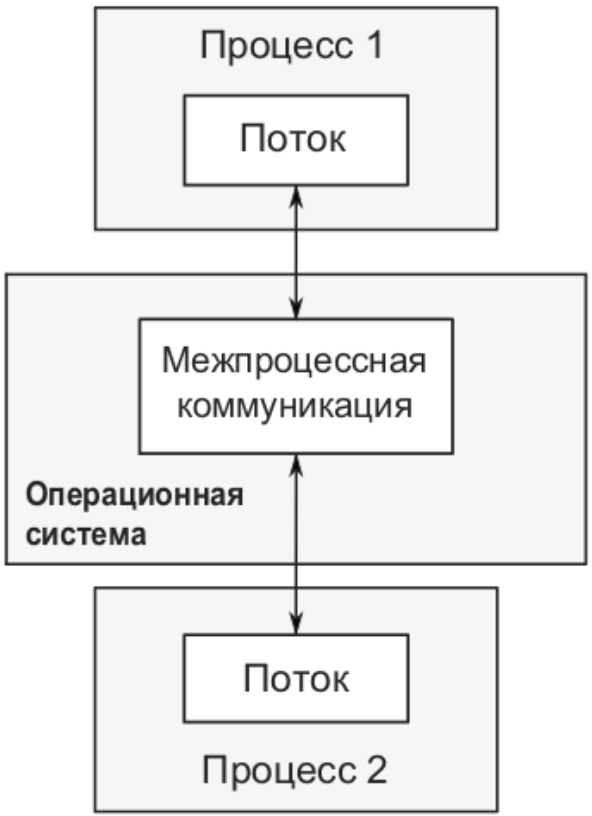


Параллелизм за счет нескольких процессов



Параллелизм за счет нескольких потоков

Параллелизм за счет нескольких процессов

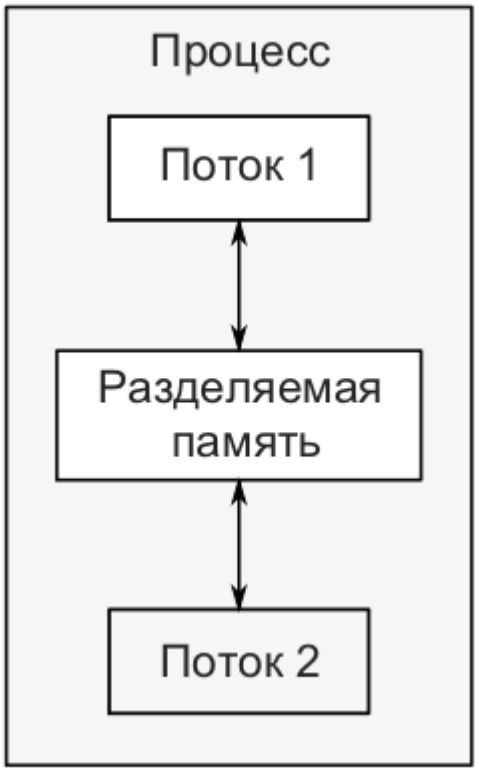


Первый способ распараллелить приложение – разбить его на несколько однопоточных одновременно исполняемых процессов. Именно так вы и поступаете, запуская вместе браузер и текстовый процессор. Затем эти отдельные процессы могут обмениваться сообщениями, применяя стандартные каналы межпроцессной коммуникации (сигналы, сокеты, файлы, конвейеры и т. д.),

Недостаток: низкая скорость, потому что операционная система должна обеспечить защиту процессов, так чтобы ни один не мог случайно изменить данные, принадлежащие другому.

Преимущество: благодаря надежной защите процессов, обеспечиваемой операционной системой, и высокоуровневым механизмам коммуникации написать безопасный параллельный код проще, когда имеешь дело с процессами, а не с потоками.
Процессы можно запускать на разных машинах, объединенных сетью.

Параллелизм за счет нескольких потоков



Альтернативный подход к организации параллелизма – запуск нескольких потоков в одном процессе. Потоки можно считать облегченными процессами – каждый поток работает независимо от всех остальных, и все потоки могут выполнять разные последовательности команд. Однако все принадлежащие процессу потоки разделяют общее адресное пространство и имеют прямой доступ к большей части данных – глобальные переменные остаются глобальными, указатели и ссылки на объекты можно передавать из одного потока в другой. Для процессов тоже можно организовать доступ к разделяемой памяти, но это и сделать сложнее, и управлять не так просто, потому что адреса одного и того же элемента данных в разных процессах могут оказаться разными.

Преимущества: не нужна защита данных в разных потоках, потому что общее адресное пространство

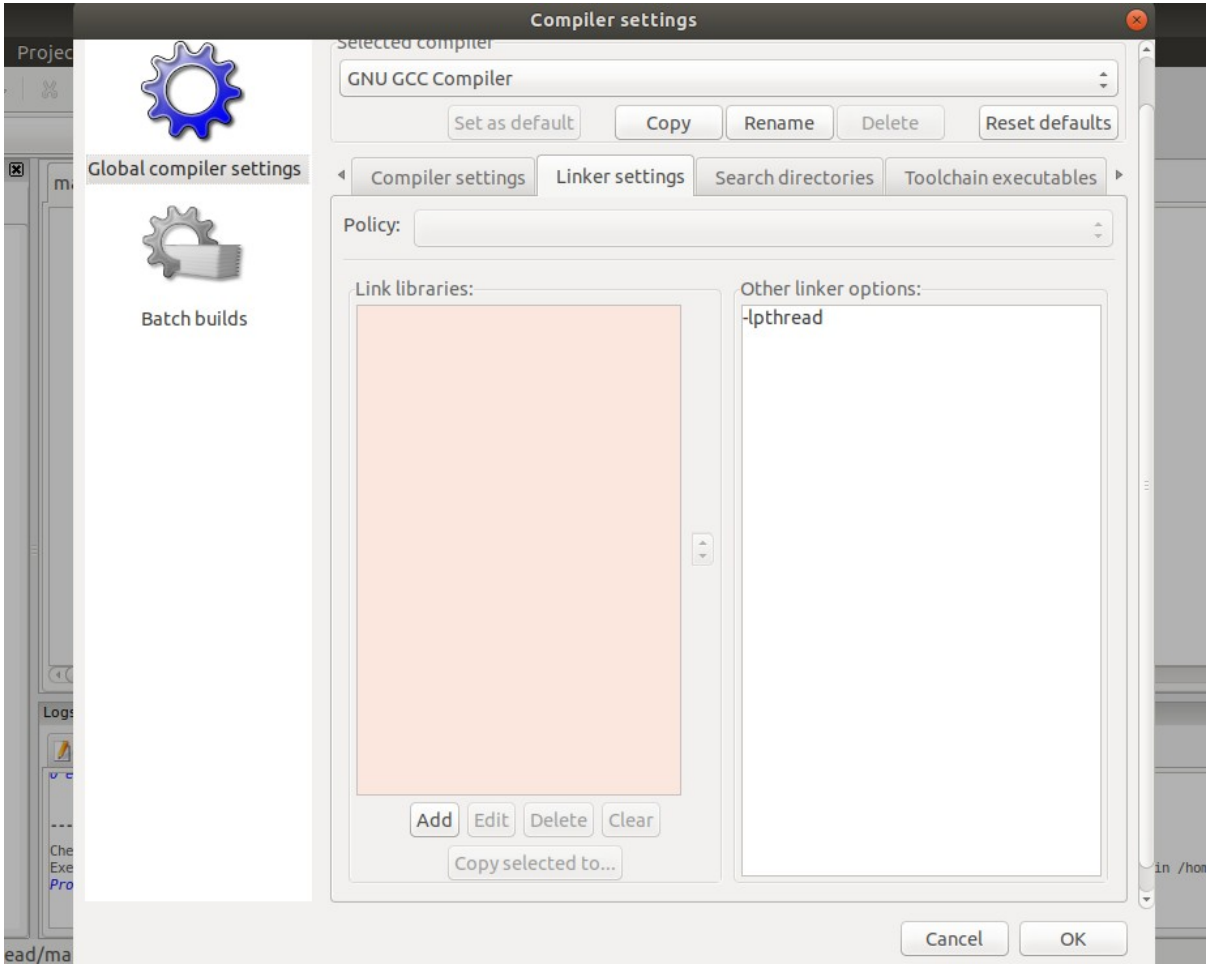
Недостатки: если к некоторому элементу данных обращаются несколько потоков, то нужно обеспечить согласованность представления этого элемента во всех потоках

Два способа применить распараллеливание для повышения производительности

Распараллеливание по задачам: разбить задачу на части и запустить их параллельно (распараллеливание может усложниться из-за наличия зависимостей между различными частями)

Распараллеливание по данным: каждый поток выполняет одну и ту же задачу, но с разными данными

Настроки компилятора в Code::Blocks для работы с потоками



Нужно компилировать с флагом -lpthread

Пример однопоточной программы

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

using namespace std;

void hello()
{
    cout << "Hello, world!" << endl;
}

int main()
{
    thread t(hello);
    t.join();
    return 0;
}
```

В main() запускается новый поток, так что теперь общее число потоков равно двум:
главный, с начальной функцией main(), и дополнительный, начинающий работу в функции hello() .

После запуска нового потока начальный поток продолжает работать. Если бы он не ждал завершения нового потока, то просто дошел бы до конца main(), после чего исполнение программы закончилась бы – быть может, еще до того, как у нового потока появился шанс начать работу.

Чтобы предотвратить такое развитие событие, мы добавили обращение к функции **join()**; это заставляет вызывающий поток (main()) ждать завершения потока, ассоциированного с объектом std::thread, – в данном случае t.

Запуск потоков в фоновом режиме

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

using namespace std;

void hello()
{
    cout << "Hello, world!" << endl;
}

int main()
{
    thread t(hello);
    t.detach();
    return 0;
}
```

Вызов функции-члена **detach()** объекта std::thread оставляет поток работать в фоновом режиме, без прямых способов коммуникации с ним.

Теперь ждать завершения потока не получится – после того как поток отсоединен, уже невозможно получить ссылающийся на него объект std::thread, для которого можно было бы вызвать join().

Передача аргументов функции потока

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

using namespace std;

void f(double a, double b, double &c)
{
    c = a*b;
}

int main()
{
    double c;
    // 1.0, 2.0 и ссылка на переменную c - аргументы функции f
    thread t(f, 1.0, 2.0, ref(c));
    t.join();
    cout << c << endl;
    return 0;
}
```

Передача владения потоком

Предположим, что требуется написать функцию для создания потока, который должен работать в фоновом режиме, но при этом мы не хотим ждать его завершения, а хотим, чтобы владение новым потоком было передано вызывающей функции. Или требуется сделать обратное – создать поток и передать владение им некоторой функции, которая будет ждать его завершения. В обоих случаях требуется передать владение из одного места в другое.

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

using namespace std;

void someFunction() {
    std::cout << "some function\n";
}

void someOtherFunction() {
    std::cout << "some other function\n";
}

int main()
{
    std::thread t1(some_function); // создается новый поток и связывается с объектом t1
    std::thread t2 = std::move(t1); // владение потоком передается объекту t2
    t1 = std::thread(some_other_function); // создается еще один поток, который связывается с объектом t1
    std::thread t3;
    t3 = std::move(t2); // владение потоком, связанным с t2, передается объекту t3
}
```

scoped_thread

Основное назначение класса **scoped_thread** – гарантировать завершение потока до выхода из области видимости.

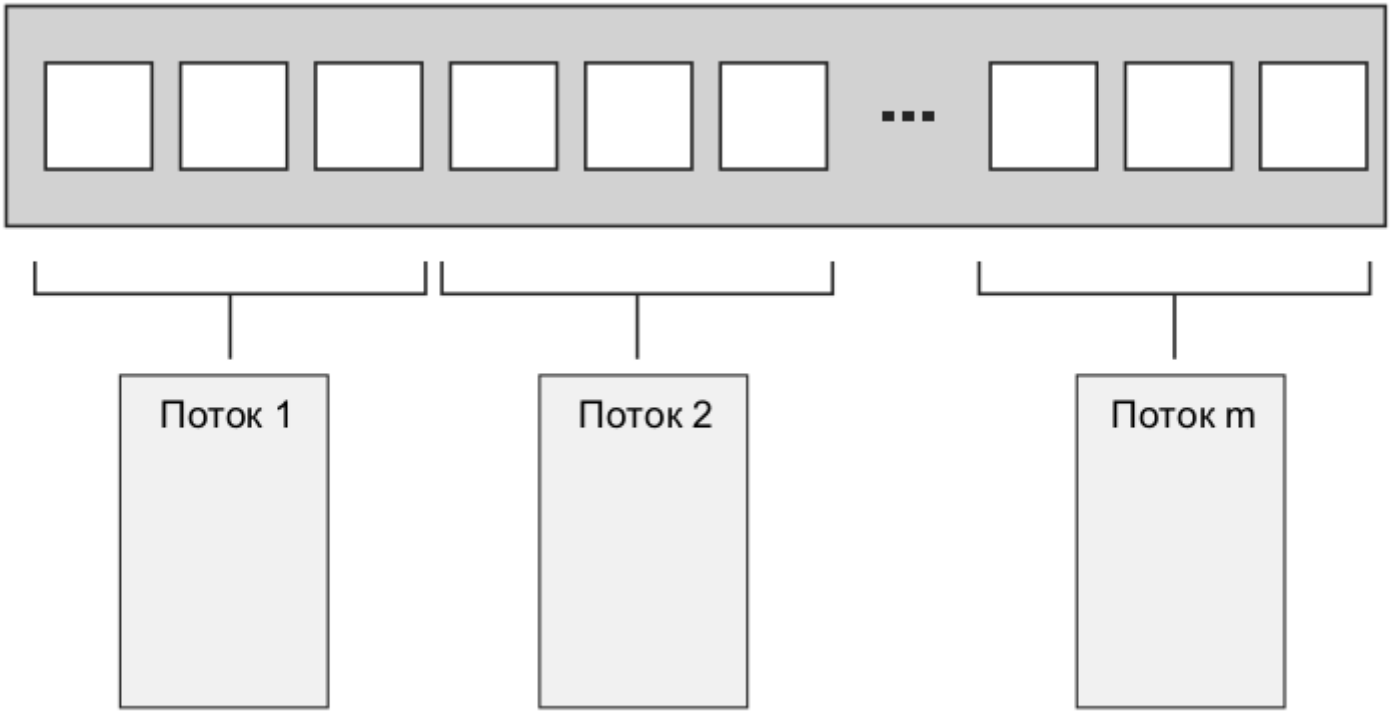
```
class Scoped_Thread
{
public:
    // конструктор
    explicit Scoped_Thread(std::thread thread) : m_thread(std::move(thread))
    {
        if (!m_thread.joinable())
            throw std::logic_error("No thread");
    }
    Scoped_Thread(const Scoped_Thread &) = delete;
    Scoped_Thread & operator=(const Scoped_Thread &) = delete;
    ~Scoped_Thread()
    {
        m_thread.join(); // гарантируется завершение потока до выхода
                         // объекта класса Scoped_Thread из области видимости
    }
private:
    std::thread m_thread;
};
```

Добавление потоков в vector и выполнение с помощью алгоритма for_each

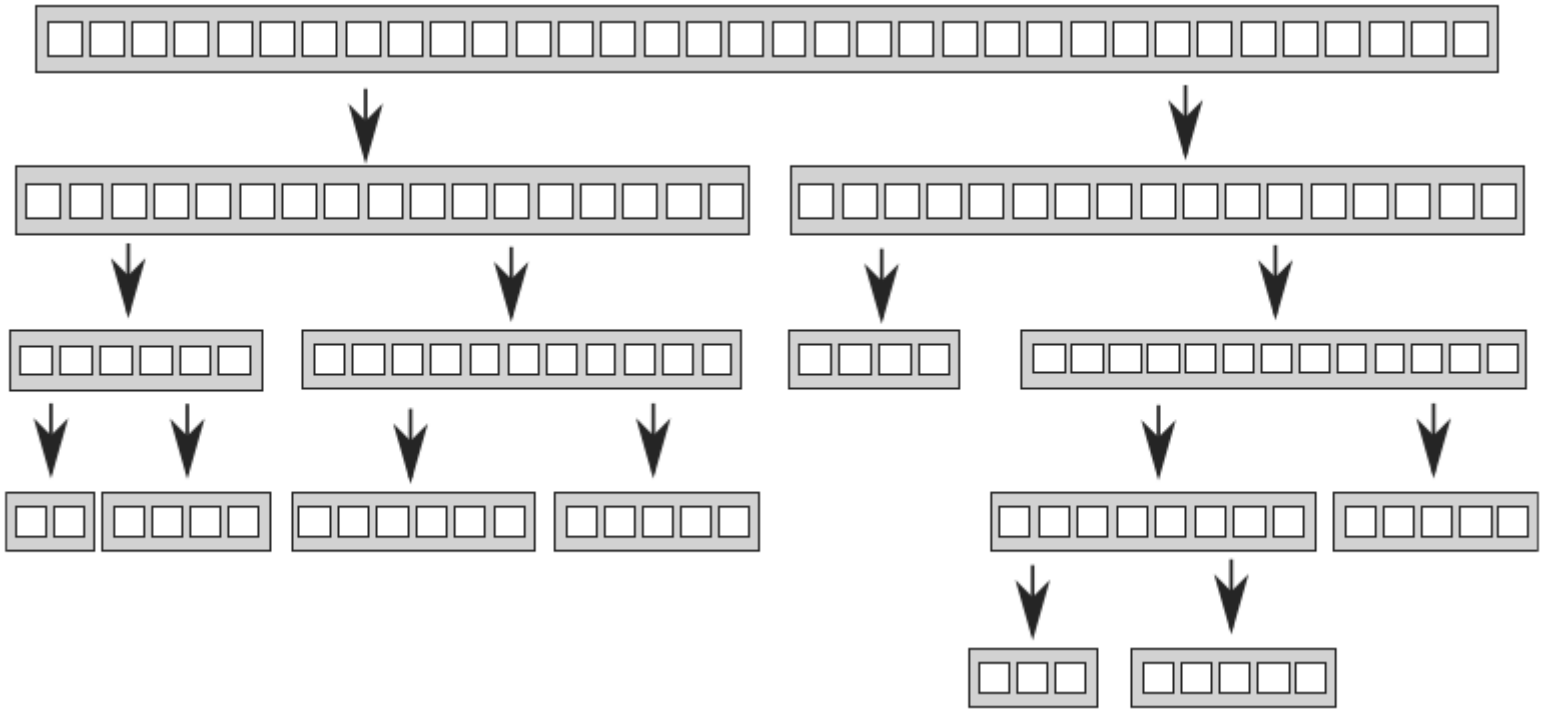
std::thread::hardware_concurrency() - доступное количество аппаратных потоков
Аргументы функции **for_each**: итератор начала контейнера, итератор конца контейнера и функция, которая применяется к каждому элементу контейнера.
В данном примере вызывается функция **join**, тобы **дождаться завершения** всех потоков

```
#include <iostream>
#include <algorithm>
#include <functional>
#include <thread>
#include <vector>

void f(int i) {std::cout << i << '\n';}

int main(int argc, char ** argv)
{
    std::vector <std::thread> threads;
    for (std::size_t i = 0; i < std::thread::hardware_concurrency(); ++i)
    {
        threads.push_back(std::thread(f, i));
    }
    std::for_each(threads.begin(), threads.end(),
        std::mem_fn(&std::thread::join));
}
```

Распределение последовательных блоков данных между потоками



Рекурсивное распределение данных



Реализация рекурсивного распределения данных

Функция **std::accumulate(first, last, init)** вычисляет **сумму элементов** начиная от итератора first до итератора last и прибавляет к начальному значению init

Основная идея алгоритма:

1) делим контейнер на два интервала
2) рекурсивно вызываем функцию от левого интервала и от правого интервала и складываем результаты
3) если на очередном рекурсивном шаге длина интервала (количество элементов) меньше порогового значения, вычисляем сумму всех элементов интервала

Реализация рекурсивного распределения данных

```
template <typename Iterator, typename T>
T parallel_accumulate(Iterator first, Iterator last, T init)
{
    // подсчет количества элементов контейнера
    const std::size_t length = std::distance(first, last);
    const std::size_t max_size = 25;
    // если при очередном делении интервала пополам его длина меньше или равна 25, складываем элементы
    if (length <= max_size)
    {
        return std::accumulate(first, last, init);
    }
    // если длина интервала больше 25, продолжаем делить пополам
    else
    {
        Iterator middle = first;
        std::advance(middle, length / 2);
        // рекурсивный вызов в новом потоке функции parallel_accumulate от левой половины интервала
        std::future <T> first_half_result =
            std::async(parallel_accumulate <Iterator, T>, first, middle, init);
        // рекурсивный вызов в текущем потоке функции parallel_accumulate от правой половины интервала
        T second_half_result = parallel_accumulate(middle, last, T());
        // сумма результатов суммирования в правом и левом интервалах
        return first_half_result.get() + second_half_result;
    }
}
```

Асинхронное программирование

Под асинхронным программированием я понимаю стиль программирования, который можно охарактеризовать следующим выражением: “дал задачу и забыл”. Т.е. это такой стиль при котором все тяжеловесные задачи исполняются в отличном от вызывающего потоке, и результат выполнения которых может быть получен, вызывающим потоком, когда он того пожелает(при условии, что результат доступен).

К примеру, вам необходимо загрузить большой файл с диска, и отобразить его в интерфейсе пользователя. Вы, конечно, можете сделать это из основного потока, заставив пользователя ждать. Если же идти по пути асинхронного программирования. тогда метод, отвечающий за загрузку файла с диска, не будет блокировать поток выполнения, а будет загружать файл в отдельном потоке, и вернёт результат по первому требованию. Т.е. фактически будет отдана задача: “Загрузи файл”, и больше никаких взаимодействий с этой задачей, до её выполнения, из основного потока, не будет.

std::future

```cpp
#include <iostream>
#include <future>

std::future<bool> submitForm(const std::string& form);

int main()
{
    auto check = submitForm("my form");
    if(check.get())
        std::cout << "OK\n";
    else
        std::cout << "Failed\n";
}
```

Класс **std::future** представляет собой обертку, над каким-либо значением или объектом, вычисление или получение которого происходит **отложено**.

Точнее, future предоставляет доступ к некоторому разделяемому состоянию, которое состоит из 2-х частей: **данные** (здесь лежит значение) и **флаг** готовности. future является получателем значения и не может самостоятельно выставлять его.

std::future

```cpp
#include <iostream>
#include <future>

std::future<bool> submitForm(const std::string& form);

int main()
{
    auto check = submitForm("my form");
    if(check.get())
        std::cout << "OK\n";
    else
        std::cout << "Failed\n";
}
```

Для получения значения из **future** предназначен метод **std::future::get**. При этом, поток вызвавший get **блокируется** до вычисления значения. Именно поэтому, мы и говорим об **отложенном** получении значения, ведь поток, получивший объект future, может не сразу блокироваться на нём, для получения значения, а может исполнять всё, что угодно и только **придя к точке**, когда необходимо получить значения future, – **получить** его. Можно, также, просто **подождать** появления значения без его непосредственного получения, для этого предназначен метод **std::future::wait**. В примере выше был использован метод **get**, т.к. нас интересует сам **ответ(значение)**, а не просто его появление. Если бы нам нужно было просто получить уведомления, и мы были бы не заинтересованы в ответе мы могли бы использовать метод wait.

Пакуем задачи

Задача является базовым блоком асинхронного программирования. В C++ роль задачи выполняет объект класса **std::packaged_task**. При выполнении std::packaged_task исполняет код **функции**, который был передан ей при создании и выставляет значение в **future**, которое, в свой черёд, является возвращаемым значением этой функции.

```
std::future<bool> submitForm(const std::string& form)
{
    auto handle = [](const std::string& form) -> bool
    {
        std::cout << "Handle the submitted form: " << form << "\n";
        return true;
    };
    std::packaged_task<bool(const std::string&)> task(handle);
    auto future = task.get_future();
    std::thread thread(std::move(task), form);
    thread.detach();
    return std::move(future);
}
```

В коде выше, мы использовали packaged_task в качестве аргумента thread, для того, чтобы исполнить задачу в отдельном потоке.

std::promise

```cpp
#include <future>
#include <thread>
#include <limits>

int main()
{
    // создание объекта promise
    auto spPromise = std::make_shared<std::promise<void>>();
    // получение future
    std::future<void> waiter = spPromise->get_future();
    // lambda-функция, которая ищет значение value
    auto call = [spPromise](size_t value)
    {
        size_t i = std::numeric_limits<size_t>::max();
        while(i--)
        {
            if(i == value)
                // сохранение значения в разделяемом состоянии
                spPromise->set_value();
        }
    };
    // создание потока для выполнения функции call
    std::thread thread(call, std::numeric_limits<size_t>::max() - 500);
    // выполнение потока в фоновом режиме
    thread.detach();
    // получение значения value из future
    waiter.get();
}
```

std::promise

```cpp
#include <future>
#include <thread>
#include <limits>

int main()
{
    auto spPromise = std::make_shared<std::promise<void>>();
    std::future<void> waiter = spPromise->get_future();
    auto call = [spPromise](size_t value)
    {
        size_t i = std::numeric_limits<size_t>::max();
        while(i--)
        {
            if(i == value)
                spPromise->set_value();
        }
    };
    std::thread thread(call, std::numeric_limits<size_t>::max() - 500);
    thread.detach();
    waiter.get();
}
```

Назначение **promise - поставка значения для future**.
Для получения future, promise содержит специальный метод **std::promise::get_future()**.

set_value – сохраняет значение в разделяемом состоянии и выставляет флаг готовности.
Важно помнить, что значение может быть выставлено только **один раз**. При попытке повторного выставления вы получить исключение **std::future_error**. Поэтому один раз выставленное значение не может быть изменено, если это не ссылка или указатель, конечно.
Как и future, promise является исключительно **перемещаемым типом**, но, в отличие от future не обладает разделяемой версией. Поэтому для копирования promise придется пользоваться сторонними средствами, например **std::shared_ptr**.

Асинхронные вызовы

std::async принимает в качестве аргументов функцию, аргументы функции и, опционально, флаг, который влияет на политику вызова async. **Возвращаемым значением async является future,** значение которого будет выставлено по возвращении функции, и будет иметь значение, ей возвращённое. Поведение async зависит от переданных флагов следующим образом:

launch::async – если передан этот флаг, то поведение async будет следующим: **будет создан объект класса thread, с функцией и её аргументами в качестве аргументов нового потока.** Т.е. async инкапсулирует создание потока, получение future и предоставляет однострочную запись для выполнения такого кода.

launch::deferred – если передан этот флаг, то никакого асинхронного вызова не произойдёт. **Вместо исполнения функции в новом потоке, она вместе с аргументами будет сохранена в future (еще одна особенность future), чтобы быть вызванными позже**. Это позже наступит тогда, когда кто-либо вызовет метод get (или wait, но не wait_for) на future, которое вернул async. При этот вызываемый объект выполнится в потоке, который вызывал get. Это поведение есть ни что иное, как отложенный вызов процедуры.

Асинхронные вызовы

```cpp
#include <iostream>
#include <string>
#include <future>
#include <thread>
#include <chrono>

using namespace std;

void defaultFunc() {
    cout << "async default, " << this_thread::get_id() << endl;
}

void deferredFunc(const string& str) {
    cout << "async deferred, " << str << ' ' << this_thread::get_id() << endl;
}

void asyncFunc() {
    cout << "async " << this_thread::get_id() << endl;
}
```

Асинхронные вызовы

```
int main()
{
    cout << "Main thread id=" << this_thread::get_id() << "\n";
    /* по умолчанию будет выбран один из флагов launch::deferred или launch::async
       в зависимости от реализации */
    auto asyncDefault = async(defaultFunc);
    /* функция deferredFunc с аргументом string("end string") сохраняется в объекте
       asyncDeffered и потом вызывается в момент вызова функции asyncDeffered.get() */
    auto asyncDeffered = async(launch::deferred, deferredFunc, string("end string"));
    // функция asyncFunc будет выполняться в новом потоке
    auto trueAsync = async(launch::async, asyncFunc);
    // блокирует выполнение потока на 5 секунд
    this_thread::sleep_for(chrono::seconds(5));
    cout << "Sleep ended\n";
    // получение значения из future
    asyncDefault.get();
    asyncDeffered.get();
    trueAsync.get();
}
```

Параллельные алгоритмы C++17

C++17 предлагает параметр режима выполнения для большинства алгоритмов:

std::execution::seq — последовательное выполнение

std::execution::par — параллельное выполнение алгоритма

std::execution::par_unseq — параллельное выполнение алгоритма и при этом поддержка инструкций AVX и SSE

Алгоритмы, которые поддерживают параллельное выполнение C++

- adjacent_difference
- adjacent_find
- all_of
- any_of
- count
- count_if
- equal
- exclusive_scan
- find
- find_end
- find_first_of
- find_if
- for_each
- for_each_n
- inclusive_scan
- mismatch
- none_of
- partition
- reduce
- remove
- remove_if
- search
- search_n
- sort
- stable_sort
- transform
- transform_exclusive_scan
- transform_inclusive_scan
- transform_reduce

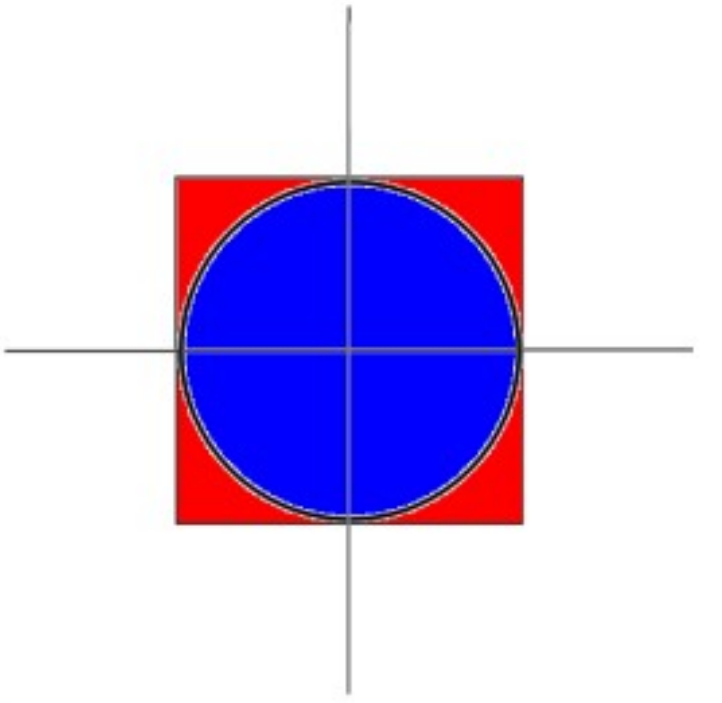
Пример: параллельная сортировка вектора myVec:

std::sort(std::execution::par, myVec.begin(), myVec.end());

Алгоритм Монте-Карло вычисления числа пи

Задаем радиус R.
Генерируем N точек из интервала (-R, R) (функция генерации случайного числа вызывается 2N раз, так как нужно сгенерировать x и y координаты точек).
Если для точки (x, y) $x^2+y^2=R^2$, значит, точка попала в круг.

$\pi = 4*N_{\text{точек в круге}} / N_{\text{всего точек}}$



Предположим, что нам нужно сгенерировать N (например, 10000) точек и число аппаратно доступных потоков равно n (например, 8). Можно в каждом отдельном потоке вызывать функцию, которая генерирует N/n точек (т. е. 1250 в нашем примере) и проверяет их на попадание внутрь круга.