Яцек Галовиц

Глава 9. Параллелизм и конкурентность

Введение

Автоматическое распараллеливание кода, использующего стандартные алгоритмы

Приостанавливаем программу на конкретный промежуток времени

Запускаем и приостанавливаем потоки

Выполняем устойчивую к исключениям общую блокировку с помощью std::unique_lock и std::shared_lock

Избегаем взаимных блокировок с применением std::scoped_lock

Синхронизация конкурентного использования std::cout

Безопасно откладываем инициализацию с помощью std::call_once

Отправляем выполнение задач в фоновый режим с применением std::async

Реализуем идиому «производитель/потребитель» с использованием std::condition_variable

Реализуем идиому «несколько производителей/потребителей» с помощью std::condition_variable

Распараллеливание отрисовщика множества Мандельброта в ASCII с применением std::async

Небольшая автоматическая библиотека для распараллеливания с использованием std::future

C++17: параллельные алгоритмы

В C++17 появилось одно действительно *крупное* расширение для параллелизма: *политики выполнения* для стандартных алгоритмов. Шестьдесят девять алгоритмов были расширены и теперь принимают политики выполнения, чтобы работать параллельно на нескольких ядрах.

```
template< class ExecutionPolicy, class RandomIt >
void sort( ExecutionPolicy&& policy, RandomIt first,
RandomIt last );
```

```
sort(execution::par, begin(d), end(d));
```

std::execution::seq, std::execution::par

Лекция: Задание 1

Опишите вычислительный эксперимент, цель которого – определить эффективность параллельной реализации алгоритма std::sort()

По шагам!!!
Шаг 1. …

Лекция: Задание 2

C++11: **Автоматическое распараллеливание кода, использующего стандартные алгоритмы**

Ставим эксперименты со стандартными алгоритмами

1. Выбрать 3 алгоритма, которые можно автоматически распараллелить
2. Придумать «большую» задачу, для решения которой можно применить выбранные алгоритмы
3. Выполнить вычислительные эксперименты и определить эффективность параллельной реализации

Futures and asynchronous function calls

https://www.justsoftwaresolutions.co.uk/threading/multithreading-in-c++0x-part-8-futures-and-promises.html

Энтони Уильямс

Параллельное программирование на C++ в действии. Практика разработки многопоточных программ

М., 2012, 672 с.

Futures and asynchronous function calls

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

int calculate_the_answer();
void do_stuff();

int main()
{
    std::future<int> the_answer=std::async(calculate_the_answer);
    do_stuff();
    std::cout << "The answer is “ << the_answer.get();
}
```

https://ru.cppreference.com/w/cpp/thread/async

```
template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
  RAIter::difference_type len = end - beg;
  if (len < 1000)
    return std::accumulate(beg, end, 0);

  RAIter mid = beg + len / 2;
  auto handle = std::async(parallel_sum<RAIter>, mid, end);
  int sum = parallel_sum(beg, mid);
  return sum + handle.get();
}

int main()
{ std::vector<int> v(10000, 1);
  std::cout << "The sum is " << parallel_sum(v.begin(),
       v.end()) << '\n';
}
```

Divide and Conquer

```
template<typename Iterator,typename Func>
void parallel_for_each(Iterator first, Iterator last, Func f)
{
    ptrdiff_t const range_length=last-first;
    if(!range_length)
        return;
    if(range_length==1)
    {
        f(*first);
        return;
    }
```

Divide and Conquer

```
    Iterator const mid=first+(range_length/2);

    std::future<void> bgtask =
        std::async(&parallel_for_each<Iterator,Func>,
                                first,mid,f);
    try
    {
        parallel_for_each(mid,last,f);
    }
    catch(...)
    {
        bgtask.wait();
        throw;
    }
    bgtask.get();
}
```

Неделя 12: Промежуточный контроль

Энтони Уильямс

Глава 3

Разделение данных между потоками