



**Нижегородский государственный университет
им. Н.И.Лобачевского**

Факультет Вычислительной математики и кибернетики

***Инструменты параллельного программирования для
систем с общей памятью***

**Библиотека Intel Threading Building Blocks –
краткое описание**

Мееров И.Б., Сысоев А.В., Сиднев А.А.
Кафедра математического обеспечения ЭВМ

Содержание...

- Введение
 - Назначение
 - Характеристики
- Инициализация и завершение библиотеки
- Распараллеливание циклов с известным числом повторений
 - Итерационное пространство
 - Реализация функтора
- Распараллеливание циклов с редукцией
 - Редукция
 - Реализация функтора



Содержание

- ❑ Распараллеливание сложных конструкций
 - Сортировка
 - Циклы с условием
 - Конвейерные вычисления
- ❑ Ядро библиотеки
 - Логические задачи
 - Алгоритм работы
 - Управление логическими задачами
- ❑ Примитивы синхронизации
- ❑ Потокобезопасные контейнеры
- ❑ Приложение
- ❑ Литература



Введение

- Рассматривается один из инструментов параллельного программирования, предназначенный для распараллеливания решения задач в системах с общей памятью, – библиотека **Intel Threading Building Blocks (ТВВ)**.
- **Основная идея ТВВ:** использование С++ для быстрой разработки *кросс-платформенных, хорошо масштабируемых параллельных приложений*.
- ТВВ предоставляет **механизмы абстрагирования** от парадигм многопоточного программирования, позволяя сосредоточиться непосредственно на решении *прикладной задачи*.



Назначение библиотеки

- ❑ Библиотека Intel® Threading Building Blocks (TBB) предназначена для разработки параллельных программ для систем с общей памятью.
- ❑ Использование библиотеки предполагает и дает возможность разработки параллельной программы в объектах.
- ❑ Библиотека скрывает низкоуровневую работу с потоками, упрощая тем самым процесс создания параллельной программы.



Возможности библиотеки...

- В состав ТВВ входит набор классов и функций, позволяющих решать следующие типичные для разработки параллельных программ задачи:
 - распараллеливание циклов с известным числом повторений;
 - распараллеливание циклов с известным числом повторений с редукцией;
 - распараллеливание циклов с условием;
 - распараллеливание рекурсии.



Возможности библиотеки

- Библиотека содержит:
 - потокобезопасные контейнеры (аналогичны контейнерам STL, за исключением того, что накладных расходов при работе с ними в параллельных приложениях меньше, чем при использовании стандартных контейнеров STL);
 - операторы выделения динамической памяти (*аллокаторы*);
 - примитивы синхронизации.



Описание библиотеки

- Межплатформенная библиотека:
 - Windows;
 - Linux;
 - Mac OS.
- Не требует поддержки со стороны компилятора (для сборки приложения необходима установленная версия ТВВ, а для запуска под операционной системой семейства Microsoft Windows достаточно иметь динамическую библиотеку **tbb.dll**).



Инициализация и завершение библиотеки



Инициализация и завершение библиотеки...

- Для использования возможностей TBB по распараллеливанию вычислений необходимо иметь хотя бы один активный (инициализированный) экземпляр класса **tbb::task_scheduler_init**. Этот класс предназначен для создания потоков и внутренних структур, необходимых планировщику потоков для работы.
- Объект класса **tbb::task_scheduler_init** может находиться в одном из двух состояний: активном или неактивном.



Инициализация и завершение библиотеки...

- Активировать экземпляр класса **tbb::task_scheduler_init** можно двумя способами:
 - непосредственно при создании объекта **tbb::task_scheduler_init**. При этом число создаваемых потоков может определяться автоматически библиотекой или задаваться вручную пользователем;
 - отложенной инициализацией при помощи вызова метода **task_scheduler_init::initialize**.



Инициализация и завершение библиотеки...

- Прототип конструктора класса **tbb::task_scheduler_init**.

```
task_scheduler_init(int number_of_threads = automatic);
```

- Доступны следующие варианты значений параметра **number_of_threads**:

- **task_scheduler_init::automatic;**
- целое положительное число;
- **task_scheduler_init::deferred.**



Инициализация и завершение библиотеки.

Автоматическое определение числа потоков

- **task_scheduler_init::automatic** (значение по умолчанию). Библиотека автоматически определяет и создает оптимальное количество потоков для данной вычислительной системы. В приложениях с большим числом компонент определить оптимальное число потоков непросто, в этом случае можно положиться на планировщик потоков библиотеки, который определит их оптимальное число автоматически, поэтому значение **task_scheduler_init::automatic** рекомендуется использовать в release-версиях приложений.

```
task_scheduler_init init; // Инициализация объекта
                        // класса tbb::task_scheduler_init
                        // по умолчанию при создании объекта
```



Инициализация и завершение библиотеки.

Ручное задание числа потоков

- Целое положительное число – число потоков, которое будет создано библиотекой. Потоки создаются сразу после вызова конструктора.

```
task_scheduler_init init(3); // Инициализация объекта класса
                             // tbb::task_scheduler_init с тремя
                             // потоками при создании объекта
```



Инициализация и завершение библиотеки.

Отложенная инициализация

- **task_scheduler_init::deferred** – отложенная инициализация объекта класса **tbb::task_scheduler_init**. Инициализация происходит только после вызова метода **task_scheduler_init::initialize**.

- Прототип метода **task_scheduler_init::initialize**.

```
void initialize(int number_of_threads = automatic);
```

- Аргумент метода (**number_of_threads**) имеет те же варианты, что и аргумент конструктора класса **tbb::task_scheduler_init**.

```
task_scheduler_init init(task_scheduler_init::deferred);  
init.initialize(3);           // Инициализация объекта класса  
                               // tbb::task_scheduler_init с  
                               // тремя потоками
```



Инициализация и завершение библиотеки.

Завершение работы...

- Перед завершением работы приложения необходимо перевести объект класса **tbb::task_scheduler_init** в неактивное состояние (завершить работу всех созданных потоков, уничтожить все созданные объекты). Это происходит автоматически в деструкторе класса **task_scheduler_init**.



Инициализация и завершение библиотеки.

Завершение работы

- Можно деактивировать объект класса **tbb::task_scheduler_init** вручную в любой требуемый момент, чтобы освободить ресурсы системы под другие нужды. Для этих целей существует метод **task_scheduler_init::terminate**.

```
void terminate();
```

- После вызова метода **task_scheduler_init::terminate** можно повторно активировать объект класса **tbb::task_scheduler_init**, вызвав метод **task_scheduler_init::initialize**.



Инициализация и завершение библиотеки

- Для изменения числа потоков библиотеки нужно перевести текущий экземпляр класса **task_scheduler_init** в неактивное состояние, вызвав метод **task_scheduler_init::terminate** или уничтожить объект, вызвав его деструктор. После этого можно вызывать метод **task_scheduler_init::initialize** для нового объекта или создать объект класса **tbb::task_scheduler_init**, указав необходимое число потоков.



Инициализация и завершение библиотеки.

Типичная схема

- Для того, чтобы создать экземпляр класса **tbb::task_scheduler_init** необходимо подключить заголовочный файл **task_scheduler_init.h**.
- Т.к. все функции/классы библиотеки расположены в пространстве имён **tbb**, то удобно объявить эту область видимости в начале программы.

```
#include "tbb/task_scheduler_init.h"

using namespace tbb;

int main()
{
    task_scheduler_init init;
    // Вычисления
    return 0;
}
```



Инициализация и завершение библиотеки. Усложнённая схема

- Динамическое управление количеством потоков создаваемых библиотекой.

```
include "tbb/task_scheduler_init.h"
using namespace tbb;

int main()
{
    task_scheduler_init init; // Инициализация по умолчанию
    //Вычисления 1
    init.terminate();        // Деинициализация
    init.initialize(4);      // Инициализация с 4-мя потоками
    //Вычисления 2
    return 0;
} // Деинициализация при уничтожении
// (вызов деструктора) объекта init
```



Распараллеливание циклов с известным числом повторений



Распараллеливание циклов с известным числом повторений

- Реализация вычислений с заранее определенным числом итераций обычно происходит с использованием цикла **for**. Библиотека ТВВ дает возможность реализовать параллельную версию таких вычислений. Для этого библиотека предоставляет шаблонную функцию **tbb::parallel_for**.

```
template<typename Range, typename Body>  
void parallel_for(const Range& range, const Body& body);
```



Распараллеливание циклов с известным числом повторений

```
template<typename Range, typename Body>
void parallel_for(const Range& range, const Body& body);
```

- Первый параметр функции **parallel_for** представляет итерационное пространство – класс специального вида, задающий количество итераций цикла.
- Второй параметр – функтор, класс, реализующий вычисления цикла через метод **body::operator()**. В C++ *функторами* или *функциональными классами* называют классы специального вида, основная функциональность которых сосредоточена в методе **operator()**.



Пример. Матрично-векторное умножение

$$\begin{array}{c} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \mathbf{a}_3 \end{array} \begin{array}{|c|c|c|c|} \hline a_{11} & a_{12} & a_{13} & a_{14} \\ \hline a_{21} & a_{22} & a_{23} & a_{24} \\ \hline a_{31} & a_{32} & a_{33} & a_{34} \\ \hline \end{array} \times \begin{array}{|c|} \hline b_1 \\ \hline b_2 \\ \hline b_3 \\ \hline b_4 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline c_1 & c_2 & c_3 \\ \hline \end{array}$$

\mathbf{b}

$$c_i = a_{i1}b_1 + a_{i2}b_2 + a_{i3}b_3 + a_{i4}b_4 = (\mathbf{a}_i, \mathbf{b}), \quad i = \overline{1,3}$$



Пример. OpenMP реализация

□ Скалярное умножение векторов

```
double VectorsMultiplication(double *v1, double *v2, int size)
{
    double result=0;
    for(int i=0; i<size; i++)
        result += v1[i] * v2[i];
    return result;
}
```

□ OpenMP реализация

```
void OpenMPMatrixVectorMultiplication(double* matrix,
double* vector, double* resultVector, int rows, int columns)
{
    #pragma omp parallel for
    for(int i=0; i<rows; i++)
        resultVector[i] =
        VectorsMultiplication(&(matrix[i*columns]), vector, columns);
}
```



Пример. ТВВ реализация...

□ OpenMP реализация

```
void OpenMPMatrixVectorMultiplication(double* matrix,
double* vector, double* resultVector, int rows, int columns)
{
    #pragma omp parallel for
    for(int i=0; i<rows; i++)
        resultVector[i] =
        VectorsMultiplication(&(matrix[i*columns]), vector, columns);
}
```

□ ТВВ реализация

```
void TBMatrixVectorMultiplication(double* matrix,
double* vector, double* resultVector, int rows, int columns,
int grainSize)
{
    parallel_for(blocked_range<int>(0, rows, grainSize),
    VectorsMultiplier(matrix, vector, resultVector, columns) );
}
```



Пример. ТВВ реализация...

□ ТВВ реализация

```
class VectorsMultiplier
{
    const double *matrix, *vector;
    double *const resultVector;
    int const columns;

public:
    VectorsMultiplier(double *tmatrix, double *tvector,
        double *tresultVector, int tcolumns) : matrix(tmatrix),
        vector(tvector), resultVector(tresultVector),
        columns(tcolumns) {}

    void operator()( const blocked_range<int>& r ) const
    {
        int begin=r.begin(), end=r.end();

        for( int i=begin; i!=end; i++ )
            resultVector[i] =
                VectorsMultiplication(&(matrix[i*columns]), vector, columns);
    }
};
```



Итерационное пространство

- Библиотека TBB содержит два специально реализованных итерационных пространства:
 - одномерное итерационное пространство **tbb::blocked_range**;
 - двумерное итерационное пространство **tbb::blocked_range2d**.
- Пользователь библиотеки может реализовать свои итерационные пространства.



Одномерное итерационное пространство

- Одномерное итерационное пространство **tbb::blocked_range** задает диапазон в виде полуинтервала [**begin**, **end**), где тип элементов **begin** и **end** задается через шаблон. В качестве параметра шаблона могут быть использованы: тип **int**, указатели, STL-итераторы прямого доступа и др.



Одномерное итерационное пространство

- ❑ Класс **tbb::blocked_range** имеет три основных поля:
 - **my_begin**;
 - **my_end**;
 - **my_grainsize**.
- ❑ Эти поля расположены в секции **private**, получить их значения можно только с помощью методов:
 - **begin**;
 - **end**;
 - **grainsize**.
- ❑ Поля **my_begin** и **my_end** задают левую и правую границу полуинтервала [**my_begin**, **my_end**).
- ❑ Поле **my_grainsize** имеет целый тип и задает размер порции вычислений.



Одномерное итерационное пространство

- Задать значение полей класса **tbb::blocked_range** можно с помощью конструктора.

```
blocked_range::blocked_range(Value begin, Value end,  
                             size_t grainsize = 1);
```

- **tbb::blocked_range:Value** – это тип, задаваемый через шаблон.



Одномерное итерационное пространство.

Расщепление

- ❑ Расщепление – операция разделения итерационного пространства на два подмножества.
- ❑ Для одномерного итерационного пространства разделение итерационного пространства выполняется на два подмножества содержащих (с точностью до округления) одинаковое количество элементов.
- ❑ Расщепление одномерного итерационного пространства происходит с помощью конструктора расщепления.

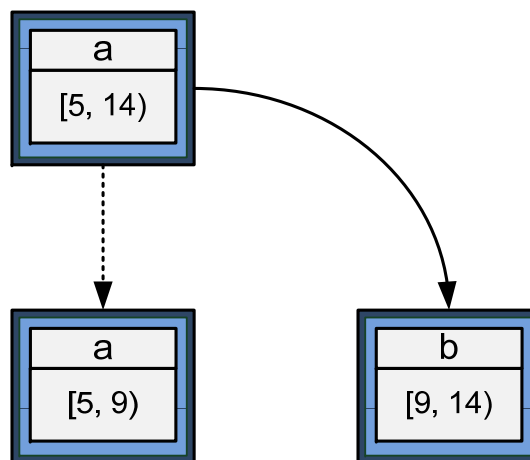


Одномерное итерационное пространство. Расщепление. Пример

- Пусть итерационное пространство **a** задает полуинтервал $[5, 14)$ и размер порции вычислений равный 2. Итерационное пространство **b** создается с помощью конструктора расщепления на основе итерационного пространства **a**.

```
blocked_range<int> a(5, 14, 2);  
blocked_range<int> b(a, split() );
```

- После вызова конструктора расщепления будет создан еще один объект того же типа и будут пересчитаны диапазоны как в новом, так и в старом объекте.



Одномерное итерационное пространство.

Пример

- Использование одномерного итерационного пространства для задания итераций цикла **for**.

```
blocked_range<int> range(0, 100);  
for (int i = range.begin(); i != range.end(); i++)  
{  
    //Вычисления  
}
```

- Аналог без использования одномерного итерационного пространства.

```
for (int i = 0; i != 100; i++)  
{  
    //Вычисления  
}
```



Пользовательское итерационное пространство...

- Разработчик может реализовать своё итерационное пространство (**Range**). Для этого необходимо определить класс, в котором следует реализовать методы:
 - **Range(const R&)** – конструктор копирования;
 - **~Range()** – деструктор;
 - **bool empty()** – метод проверки итерационного пространства на пустоту. Если оно пусто, то функция должна вернуть **true**, иначе **false**;
 - **bool is_divisible()** – метод проверки на возможность разделения итерационного пространства. Если разделение возможно, то функция должна вернуть **true**, иначе **false**;



Пользовательское итерационное пространство

- **Range(R& r, split)** – конструктор расщепления, создает копию итерационного пространства и разделяет диапазон, задаваемый итерационным пространством, на две части (изменяется диапазон итерационного пространства как вновь созданного объекта, так и объекта его породившего).
- Параметр **split** (служебный класс без полей и методов) предназначен для того, чтобы отличить конструктор копирования от конструктора расщепления.

```
namespace tbb
{
    class split
    { };
}
```



Пользовательское итерационное пространство. Пример...

- Аналог одномерного итерационного пространства типа `int`

```
class SimpleRange
{
private:
    int my_begin;
    int my_end;
public:
    int begin() const { return my_begin; }
    int end() const { return my_end; }
    bool empty() const { return my_begin == my_end; }
    bool is_divisible() const { return my_end > my_begin + 1; }

    SimpleRange(int begin, int end): my_begin(begin), my_end(end)
    {}
    SimpleRange(SimpleRange& r, split )
    {
        int medium = (r.my_begin + r.my_end) / 2;
        my_begin = medium;
        my_end = r.my_end;
        r.my_end = medium;
    }
};
```



Пользовательское итерационное пространство.

Пример

- ❑ Конструктор копирования и деструктор явно не реализованы, т.к. их реализация по умолчанию является корректной.
- ❑ В классе **SimpleRange** не объявлено поле **my_grainsize**, которое задавало размер порции вычислений в классе **tbb::blocked_range**. Его наличие в общем случае не является обязательным, т.к. размер порции вычислений на самом деле определяется реализацией метода **is_divisible**. В указанной реализации этот размер равен 1.
- ❑ Поля **my_begin**, **my_end** и методы **begin**, **end** в общем случае не обязательны. В данном примере наглядно демонстрируется как может быть реализовано простейшее одномерное итерационное пространство.



Функтор

```
template<typename Range, typename Body>  
void parallel_for(const Range& range, const Body& body);
```

- ❑ Второй параметр функции **parallel_for** – функтор, класс, реализующий вычисления цикла через метод **body::operator()**.
- ❑ В первом приближении можно считать, что функтор получается в результате трансформации тела цикла в класс.
- ❑ Является функциональным классом и не должен содержать в себе ни обрабатываемые данные, ни получаемый результат.



Функтор

- Функтор для функции **tbb::parallel_for** должен содержать следующие методы:
 - Конструктор копирования. Необходим для корректной работы функции **tbb::parallel_for**, которая создает копии функтора в соответствии с принятым разработчиками библиотеки алгоритмом реализации параллелизма;
 - Деструктор. **~Body()**;
 - Метод **operator()**, выполняющий вычисления. Его аргументом является итерационное пространство.

```
void operator() (Range& range) const
```



Функтор

- Метод **operator()** является основным в функторе. Метод объявлен константным, поскольку не нуждается в изменении значений полей функтора, если таковые в нем имеются.



Функтор. Пример...

- ❑ В качестве примера рассмотрим задачу умножения матрицы на вектор.
- ❑ Ниже представлена реализация вспомогательной функции скалярного умножения векторов.

```
//Скалярное умножение векторов
double VectorsMultiplication(const double *a, const double *b,
                             int size)
{
    double result = 0.0;
    for(int i = 0; i < size; i++)
        result += a[i] * b[i];
    return result;
}
```



Функтор. Пример

```
class VectorsMultiplier //Функтор
{
    const double *matrix, *vector; // Исходные данные для умножения
    double *const resultVector; // Вектор результатов
    int const numOfColumns; // Количество столбцов матрицы
public:
    VectorsMultiplier(double *tmatrix, double *tvector,
        double *tresultVector, int tnumOfColumns) : matrix(tmatrix),
        vector(tvector), resultVector(tresultVector),
        numOfColumns(tnumOfColumns)
    {}

    void operator()(const blocked_range<int>& r) const
    {
        int begin = r.begin(), end = r.end();

        for (int i = begin; i != end; i++)
            resultVector[i] = VectorsMultiplication(&(matrix[i * numOfColumns]),
                vector, numOfColumns);
    }
};
```



Планирование вычислений

- ❑ Алгоритм работы функции **tbb::parallel_for** устроен таким образом, что планирование вычислений осуществляется динамически, то есть на этапе выполнения.
- ❑ Определяющим моментом планирования является то, как реализовано итерационное пространство



Планирование вычислений при использовании одномерного итерационного пространства

- ❑ Одним из полей одномерного итерационного пространства является размер порции вычислений – **grainsize**.
- ❑ Его значение является определяющим при планировании вычислений.



Планирование вычислений при использовании одномерного итерационного пространства

- ❑ Функция **tbb::parallel_for** распределяет между всеми потоками на выполнение части итерационного пространства, размером **grainsize**.
- ❑ Если **grainsize** равно размеру итерационного пространства (общему числу итераций), то все итерации будут выполнены на одном потоке.
- ❑ Если **grainsize** равно $\frac{\text{общее число итерации}}{\text{число потоков}}$, то каждый поток, скорее всего (т.к. планирование осуществляется динамически, то точно сказать нельзя), выполнит одинаковое число итераций, равное **grainsize**.
- ❑ Если **grainSize** меньше, чем $\frac{\text{общее число итерации}}{\text{число потоков}}$, то планировщик потоков распределит итерации между потоками динамически.



Выбор размера порции вычислений...

- ❑ Малое значение **grainsize** способствует увеличению масштабируемости приложения (запуск на системе с большим количеством процессоров/ядер приведет к большему ускорению).
- ❑ Например, если значение **grainsize** равно половине итерационного пространства, то при запуске на машине с 4-мя процессорами работа будет выполняться только двумя из них, т.к. остальным ее просто не достанется из-за большого значения **grainsize**.
- ❑ Необходимо устанавливать маленькие значения **grainsize** для большей масштабируемости приложения.



Выбор размера порции вычислений...

- ❑ Работа планировщика потоков занимает определенное время, поэтому, чем меньше значение **grainsize**, тем больше времени потребуется функции **tbb::parallel_for** на распределение заданий.
- ❑ При очень малых значениях **grainsize** приложение будет обладать очень хорошей масштабируемостью, но при этом будет работать очень не эффективно из-за больших накладных расходов на работу планировщика.
- ❑ При очень больших значениях **grainsize** приложение будет работать максимально эффективно, но будет очень плохая его масштабируемость.



Выбор размера порции вычислений

- ❑ Параметр **grainsize** не должен быть слишком маленьким, т.к. это может негативно отразиться на времени работы приложения (большие накладные расходы на работу функции **tbb::parallel_for**).
- ❑ Параметр **grainsize** не должен быть слишком большим, т.к. это может негативно отразиться на масштабируемости приложения.



Алгоритм экспериментального подбора размера порции вычислений...

- ❑ Установите значение **grainsize** достаточно большим, например равным размеру итерационного пространства в случае использования класса **blocked_range**.
- ❑ Запустите приложение в один поток, замерьте время его выполнения.
- ❑ Установите значение **grainsize** в 2 раза меньше, запустите приложение по-прежнему в один поток и оцените замедление по отношению к шагу 2. Если приложение замедлилось на 5-10%, это хороший результат. Продолжайте уменьшение **grainsize** до тех пор, пока замедление не превысит 5-10%.



Алгоритм экспериментального подбора размера порции вычислений

- ❑ С помощью указанного алгоритма происходит определение накладных расходов, расходующихся на планирование вычислений.
- ❑ В алгоритме устанавливается величина этих накладных расходов, равная 5-10% от времени работы всего алгоритма.
- ❑ Максимальное ускорение параллельной версии над последовательной при таком подходе не будет превышать 10-20 раз.



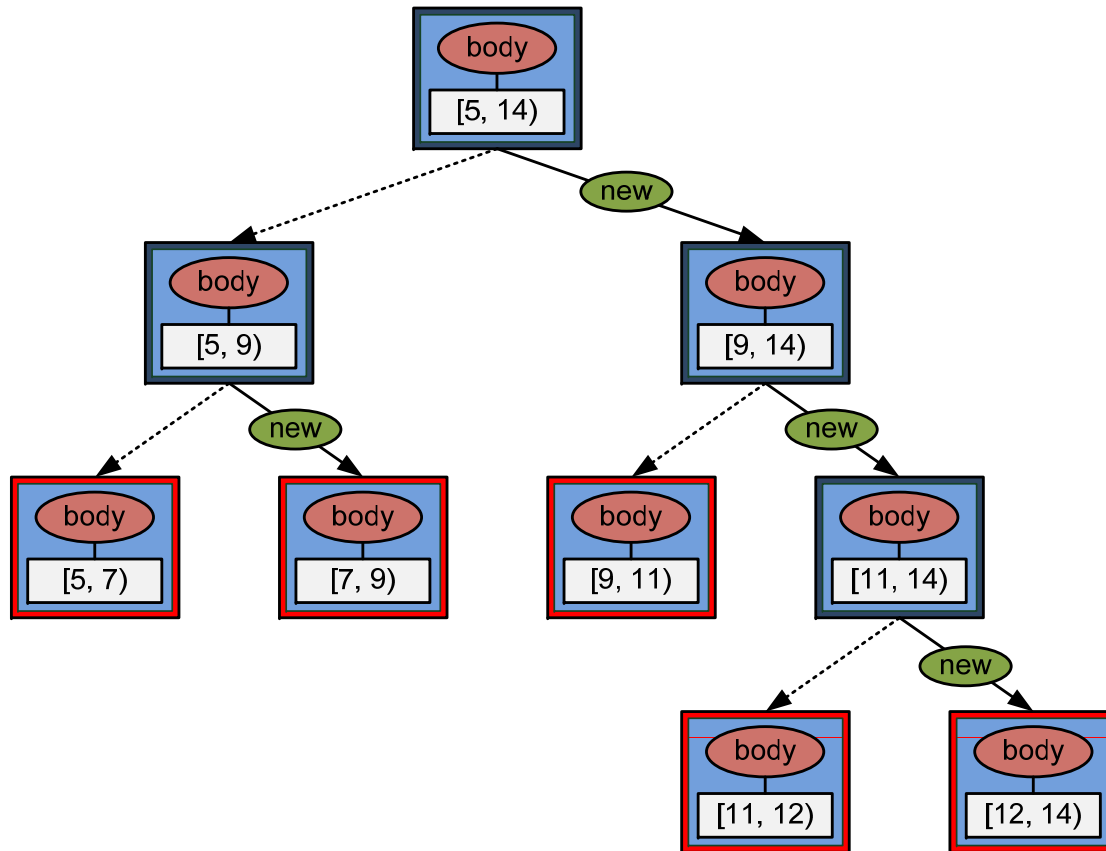
Пример работы функции `tbb::parallel_for`

```
parallel_for(blocked_range<int>(5, 14, 2), body);
```

- ❑ В начале имеется функтор **body** и одномерное итерационное пространство, размер которого равен $14 - 5 = 9$.
- ❑ Размер итерационного пространства больше, чем размер порции вычислений ($9 > 2$), поэтому функция **tbb::parallel_for** расщепляет итерационное пространство на два, одновременно создавая для нового итерационного пространства собственный функтор (через конструктор копирования) и меняя, размер итерационного пространства для старого функтора.
- ❑ Данный процесс будет происходить рекурсивно, до тех пор, пока размер очередного итерационного пространства будет не больше 2.
- ❑ После этого для каждого созданного функтора будет вызван метод **body::operator()** с сопоставленным с этим функтором итерационным пространством в качестве параметра.



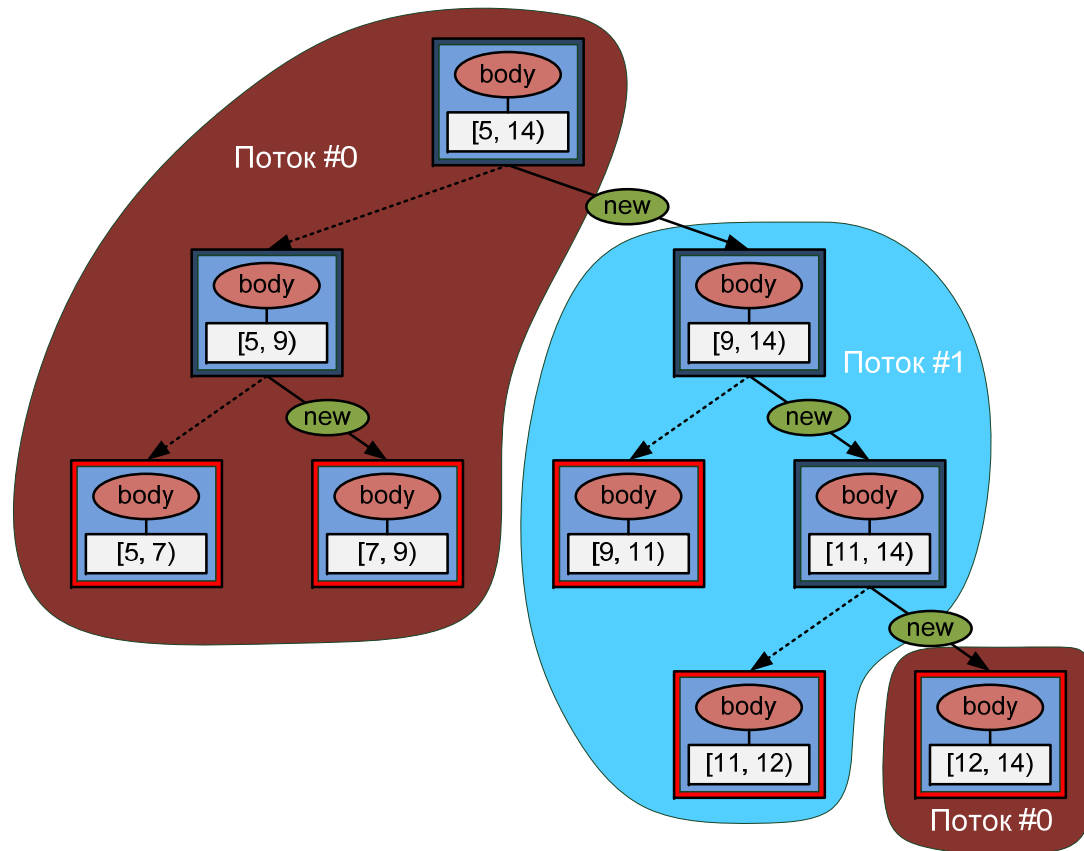
Пример работы функции `tbb::parallel_for`



- ❑ Надпись **new** над стрелкой означает, что создается новый экземпляр итерационного пространства и функтора.
- ❑ Пунктирная стрелка означает, что изменяется диапазон, задаваемый итерационным пространством, при этом создание новых экземпляров функтора и итерационного пространства не происходит.



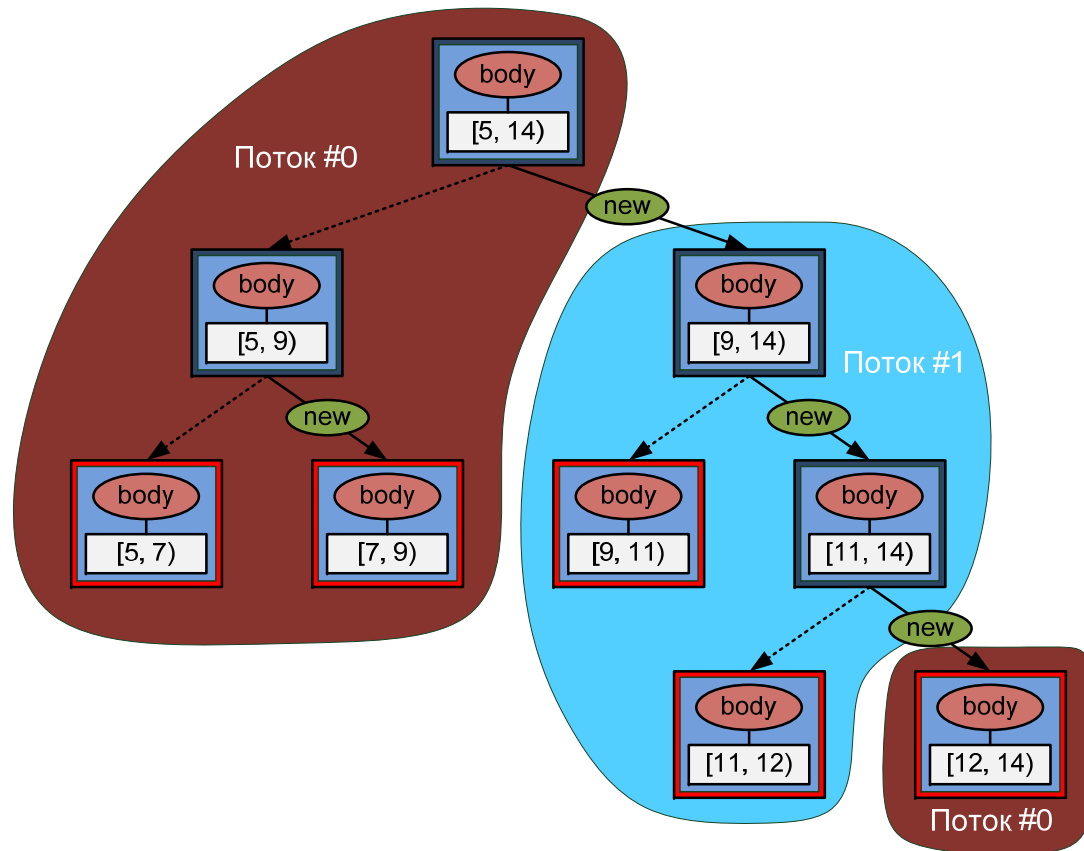
Пример параллельной обработки...



- ❑ После первого расщепления итерационного пространства будут существовать два не пересекающихся итерационных пространства.
- ❑ Если в библиотеке создано больше одного потока, то обработка полученных поддеревьев будет происходить параллельно.



Пример параллельной обработки



- ❑ Создание «порции» вычислений в виде итерационного пространства и связанного с ним функтора может выполняться и часто выполняется отдельно от дальнейшей обработки этого «порции».



Пример использования функции `tbb::parallel_for`

- Пример использования `tbb::parallel_for` в задаче умножения матрицы на вектор (**`numOfRows`** – количество строк матрицы, **`numOfColumns`** – количество столбцов матрицы).

```
parallel_for(blocked_range<int>(0, numOfRows, grainSize),  
            VectorsMultiplier(matrix, vector,  
                               resultVector, numOfColumns));
```

- OpenMP аналог.

```
#pragma omp parallel for schedule(dynamic, grainsize)  
for(int i = 0; i < numOfRows; i++)  
    resultVector[i] =  
        VectorsMultiplication(&(matrix[i*columns]),  
                              vector, numOfColumns);
```



Распараллеливание циклов с редукцией



Редукция

- Одна из типичных задач параллельных вычислений.
- При обработке данных параллельно каждый из потоков выполняет обработку части данных, в итоге конечного (суммарного) результата нет ни у одного потока.
- Необходимо «собрать» данные со всех потоков:
 - это может быть обычное суммирование значений, полученных каждым потоком;
 - это может быть выполнение логических операций (например логическое «и») над значениями, полученными каждым потоком;
 - ...



Распараллеливание циклов с редукцией

- Для распараллеливания циклов в которых необходимо выполнять редукцию, в библиотеке ТВВ используется шаблонная функция **tbb::parallel_reduce**.

```
template<typename Range, typename Body>  
void parallel_reduce(const Range& range, Body& body);
```



Распараллеливание циклов с редукцией

```
template<typename Range, typename Body>
void parallel_reduce(const Range& range, Body& body);
```

- Первый параметр, так же как и в функции **parallel_for**, – итерационное пространство.
- Вторым параметром – функтор, класс, реализующий вычисления цикла через метод **body::operator()** и выполняющий редукцию.
 - Методы, которые необходимо реализовать в функторе для функции **parallel_reduce**, отличаются от тех, которые реализуются для функтора **parallel_for**.



Функтор для функции `parallel_reduce`

- Функтор для функции `tbb::parallel_reduce` должен содержать следующие методы:

- Конструктор расщепления.

```
Body(body&, split)
```

- Деструктор.

```
~Body()
```

- Метод, выполняющий вычисления.

```
void operator()(Range& range)
```

- Метод, выполняющий редукцию.

```
void join(Body& rhs)
```



Функтор для функции `parallel_reduce`. Метод, выполняющий вычисления

```
void operator()(Range& range)
```

- ❑ Аргументом этого метода является итерационное пространство.
- ❑ Заметим, что в отличие от функтора функции `tbb::parallel_for` у функтора функции `tbb::parallel_reduce` вычислительный метод не является константным, а это означает, что в методе можно изменять поля класса. Данное требование связано с необходимостью сохранения промежуточных результатов, которые будут использоваться при выполнении операции редукции для получения окончательного результата.



Функтор для функции `parallel_reduce`.

Метод, выполняющий редукцию

```
void join(Body& rhs)
```

- ❑ В качестве параметра принимает ссылку на функтор, который выполнил часть вычислений.
- ❑ Посчитанные им данные должны быть учтены текущим функтором (**this**), для получения окончательного результата.
- ❑ Функтор, переданный по ссылке, автоматически уничтожается после завершения редукции (вызова функции **join**).



Алгоритм работы `parallel_reduce`

- ❑ Алгоритм работы `tbb::parallel_reduce` похож на `tbb::parallel_for`.
- ❑ Функции `tbb::parallel_reduce` не создает копии исходного функтора при каждом расщеплении (кроме отдельного случая), а оперирует ссылками на функторы.
- ❑ Функция `tbb::parallel_reduce` выполняет дополнительный этап вычислений – редукцию и, в зависимости от того, как происходит распределение вычислений по потокам, реализует одну из двух схем:
 - если очередная «порция» вычислений обрабатывается на том же потоке, что и предыдущая, то реализуется первая схема;
 - если очередная «порция» вычислений обрабатывается на потоке отличном от потока-создателя «порции», то реализуется вторая схема.



Алгоритм работы `parallel_reduce`. Пример...

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

- ❑ Функция **`tbb::parallel_reduce`**, так же как и **`tbb::parallel_for`** расщепляет (в данном примере) одномерные итерационные пространства до тех пор пока их размеры больше, чем **`grainsize`**.
- ❑ Отличие в работе функции **`tbb::parallel_reduce`** состоит в том, что она не создает копии исходного функтора при каждом расщеплении (кроме отдельного случая).



Алгоритм работы `parallel_reduce`. Пример...

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

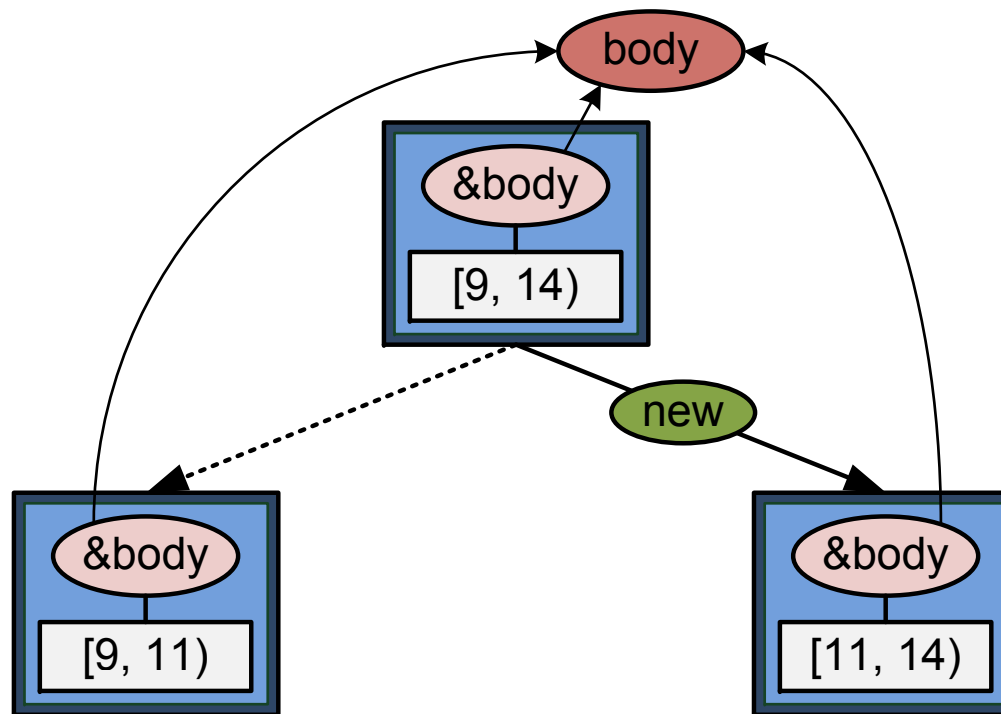
- ❑ Пусть очередная «порция» вычислений обрабатывается на том же потоке, что и предыдущая.
- ❑ В данной схеме требуется и существует только один экземпляр функтора.
- ❑ Данная схема реализуется всегда при вычислении в один поток.



Алгоритм работы `parallel_reduce`. Пример...

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

- Пусть на очередной итерации потоком была создана «порция» вычислений размером $[9, 14)$ и этот же поток обрабатывает эту «порцию».



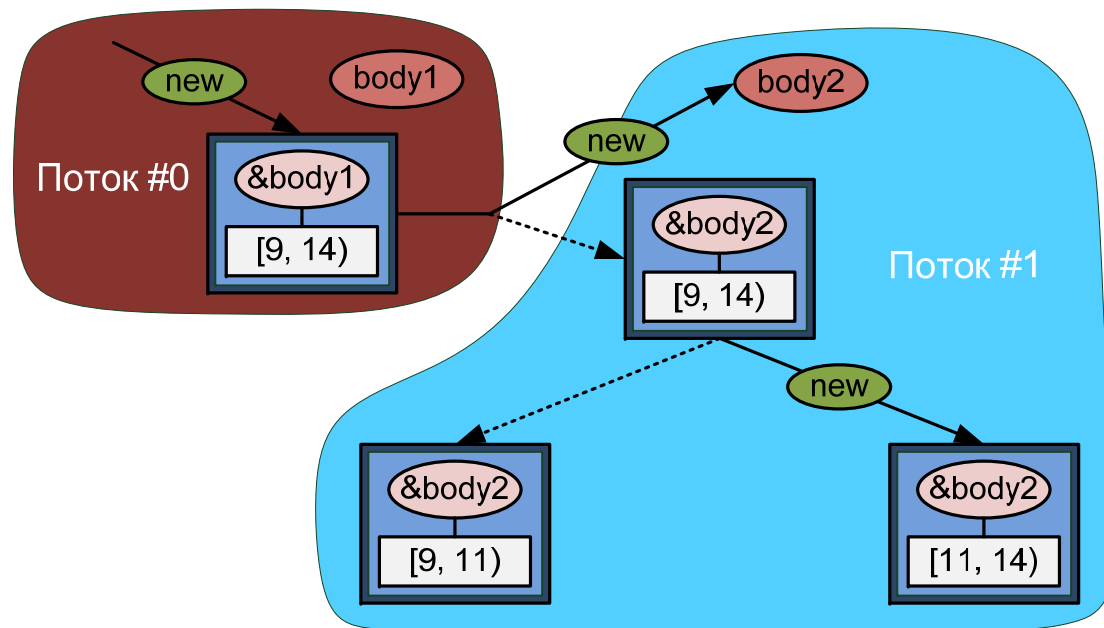
Алгоритм работы `parallel_reduce`. Пример

```
parallel_reduce(blocked_range<int>(5, 14, 2), body);
```

- ❑ Пусть очередная «порция» вычислений выполняется на потоке отличном от потока-создателя «порции».
- ❑ Происходит создание нового функтора с помощью конструктора расщепления.

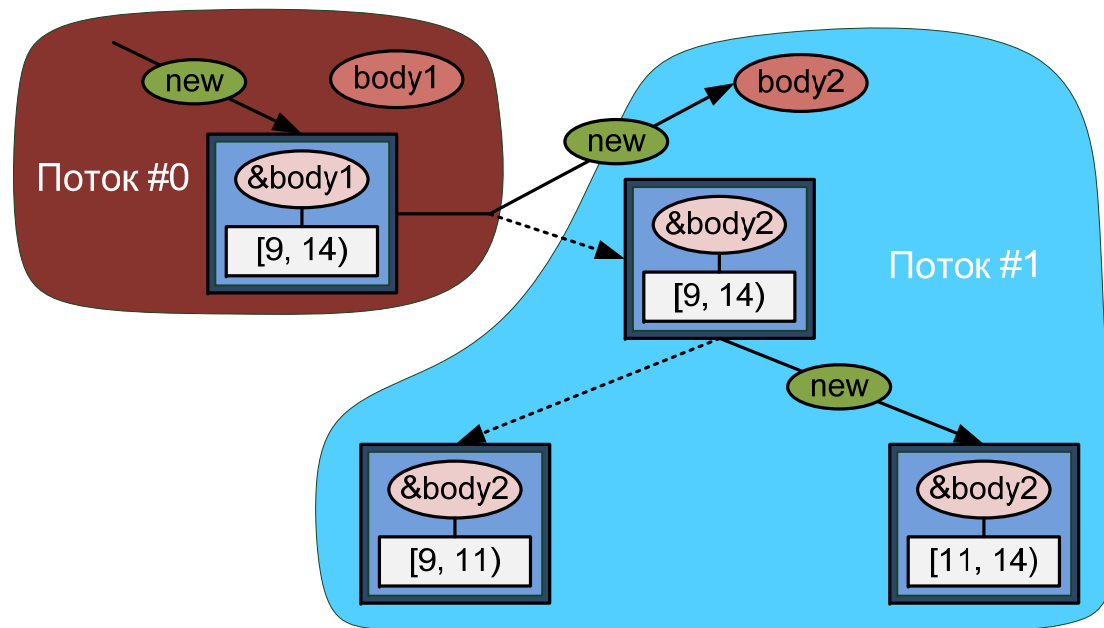


Пример параллельной обработки...



- ❑ Пусть на очередной итерации потоком 0 были созданы две «порции» вычислений [5, 9) и [9, 14).
- ❑ Поток 0 продолжил выполнять вычисления с «порцией» [5, 9), а поток 1 взял на выполнение «порцию» [9, 14).

Пример параллельной обработки



- ❑ Во избежание возможных гонок данных поток 1 не использует ссылку на существующий функтор **body1**, а создает новый функтор **body2** с помощью конструктора расщепления и продолжает работать с ним, подставляя ссылку на него в «порцию» вместо исходного функтора **body1**.
- ❑ Далее все происходит так же как в первой схеме.



Алгоритм работы `parallel_reduce`.

Расщепление функтора

- ❑ Конструктор расщепления функтора реально никакого «разделения» функтора или его полей на части не производит.
- ❑ В большинстве случаев его работа совпадает с работой конструктора копирования.
- ❑ Использование конструктора расщепления в данном случае лишь дает возможность реализовать разработчикам функтора более сложное поведение в момент «переноса» функтора на другой поток.

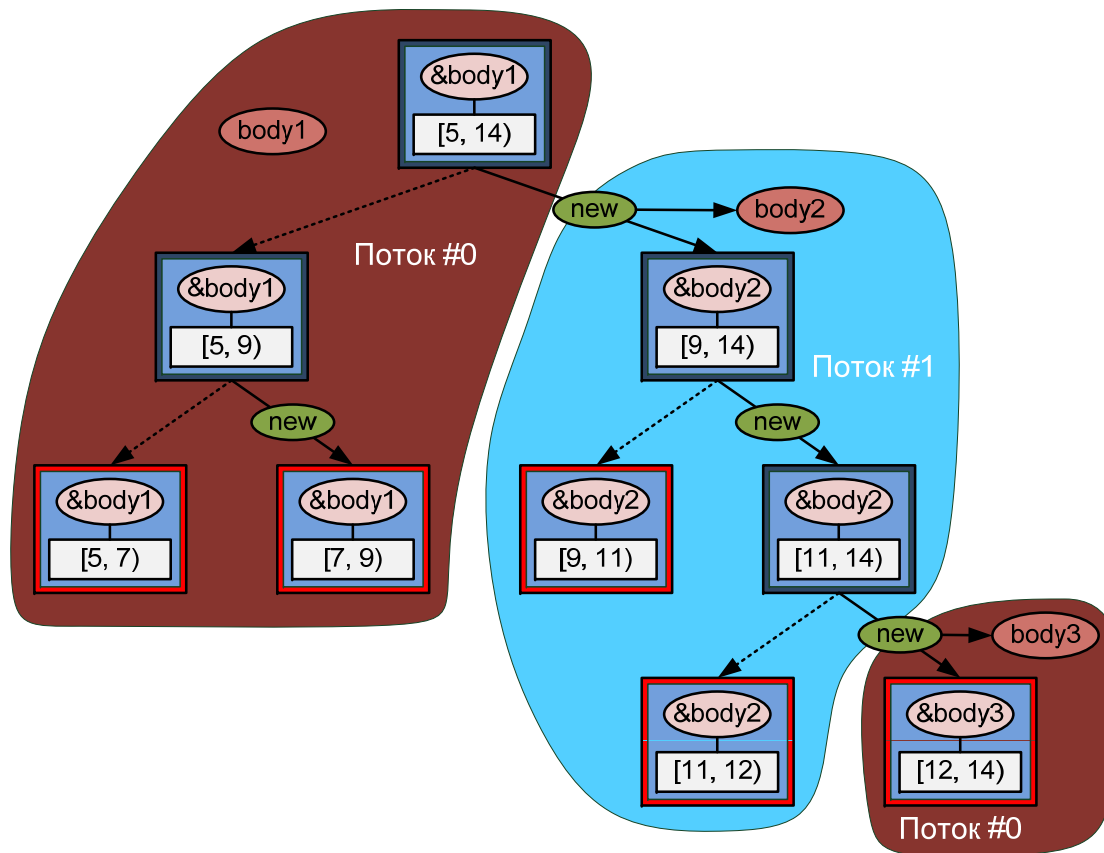


Алгоритм работы `parallel_reduce`. Редукция

- ❑ Если все вычисления происходили в один поток, то операция редукции не требуется, т.к. функтор существует в одном экземпляре, и он один выполнил все вычисления.
- ❑ Если в некоторый момент очередная «порция» вычислений была создана одним потоком, а сами вычисления производились другим, то это приводит к созданию нового функтора. Это значит, что необходимо выполнить редукцию нового функтора на старый.

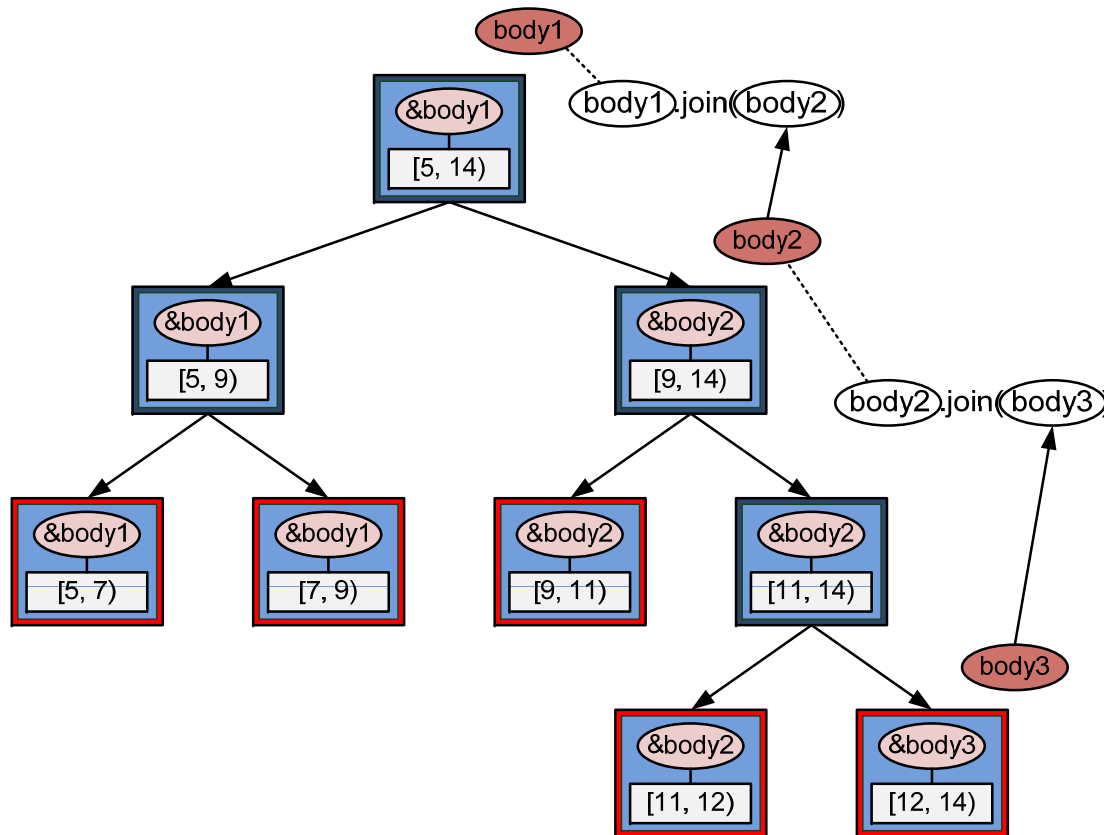


Пример параллельной обработки...



- Процесс вычислений является не детерминированным (какой поток выполнит конкретную часть вычислений определяется только на этапе выполнения).

Пример параллельной обработки...



- В тех узлах дерева, где происходило расщепление функтора, по завершении вычислений будут выполнены операции редукции.



Пример использования функции `tbb::parallel_reduce...`

- ❑ Пример функтора функции `tbb::parallel_reduce` для решения задачи скалярного умножения векторов.
- ❑ Последовательная версия скалярного умножения векторов.

```
//Скалярное умножение векторов
double VectorsMultiplication(double *v1, double *v2, int size)
{
    double result = 0;
    for (int i = 0; i < size; i++)
        result += v1[i] * v2[i];
    return result;
}
```



Пример использования функции `tbb::parallel_reduce`. Реализация функтора...

```
❑ class ScalarMultiplier //Функтор
{
private:
    const double *a, *b;
    double c;

public:
    explicit ScalarMultiplier(double *ta, double *tb):
        a(ta), b(tb), c(0)
    {}

    ScalarMultiplier(const ScalarMultiplier& m, split):
        a(m.a), b(m.b), c(0)
    {}

    //...
};
```



Пример использования функции `tbb::parallel_reduce`. Реализация функтора

```
class ScalarMultiplier //Функтор
{
    //...
public:
    void operator()(const blocked_range<int>& r)
    {
        int begin = r.begin(),
            end = r.end();
        c += VectorsMultiplication(&(a[begin]), &(b[begin]), end - begin);
    }

    void join(const ScalarMultiplier& multiplicator)
    {
        c += multiplicator.c;
    }

    double Result()
    {
        return c;
    }
};
```



Пример использования функции `tbb::parallel_reduce`

- Пример использования `tbb::parallel_reduce` в задаче скалярного умножения векторов (**size** – размер векторов; **a**, **b** – исходные вектора)

```
ScalarMultiplier s(a, b);  
parallel_reduce(blocked_range<int>(0, size, grainSize), s);
```

- OpenMP аналог

```
#pragma omp parallel for schedule(dynamic, grainsize) \  
reduce(+: c)  
for (int i = 0; i < size / grainsize; i++)  
    c += VectorsMultiplication(&(a[i * grainsize]),  
                               &(b[i * grainsize]), grainsize);
```



Распараллеливание СЛОЖНЫХ КОНСТРУКЦИЙ



Параллельная сортировка

- ТВВ содержит шаблонную функцию **tbb::parallel_sort**, предназначенную для сортировки последовательности.
- С помощью **tbb::parallel_sort** можно выполнять параллельную сортировку встроенных типов языка C++ и всех классов, у которых реализованы методы **swap** и **operator()**:
 - **operator()** - выполняет сравнение двух элементов;
 - **swap()** - выполняет перестановку двух элементов.



Параллельная сортировка. Пример

□ Пример использования параллельной сортировки

```
#include "tbb/parallel_sort.h"
#include <math.h>
using namespace tbb;
const int N = 100000;
float a[N];
float b[N];

void SortExample()
{
    for ( int i = 0; i < N; i++ )
    {
        a[i] = sin((double)i);
        b[i] = cos((double)i);
    }
    parallel_sort(a, a + N);
    parallel_sort(b, b + N, std::greater<float>());
}
```



Распараллеливание циклов с условием

- Библиотека TBB содержит шаблонный класс **tbb::parallel_while**, с помощью которого можно выполнить параллельную обработку элементов, размещенных в некотором «входном» потоке данных.
- Элементы могут быть добавлены в поток данных во время вычислений.



Класс обрабатываемых элементов

- Класс обрабатываемых элементов должен содержать следующие обязательные методы:
 - конструктор по умолчанию `element_type()`;
 - конструктор копирования `element_type(const element_type&)`;
 - деструктор `~element_type()`.



Поток данных

- Поток данных – класс, который должен содержать метод, возвращающий очередной элемент. Данный метод будет выполняться только одним потоком в каждый момент времени.

```
bool S::pop_if_present( B::arguement_type& item )
```



Класс обрабатывающий элементы

- ❑ Класс, который будет заниматься обработкой элементов должен содержать метод обработки очередного элемента.

```
B::operator() (B::element_type& item) const
```

- ❑ Этот метод может вызываться несколькими потоками одновременно и выполняться параллельно для разных элементов из потока данных.
- ❑ Класс обрабатывающий элементы должен содержать следующее переопределение типа.

```
typedef element_type argument_type;
```



Конвейерные вычисления

- ❑ Библиотека TBB содержит класс **tbb::pipeline**, с помощью которого можно выполнять конвейерные вычисления.
- ❑ Для таких вычислений характерно выполнения нескольких стадий вычислений над одним и тем же элементом.
- ❑ Если хотя бы на одной из стадий работа над разными элементами может быть выполнена параллельно, то с помощью данного класса можно организовать такие вычисления.



Класс `tbb::pipeline`

- ❑ Класс `tbb::pipeline` выполняет обработку элементов, заданных с помощью потока данных.
- ❑ Обработка осуществляется с помощью набора фильтров, которые необходимо применить к каждому элементу.
- ❑ Фильтры могут быть последовательными и параллельными.
- ❑ Для добавления фильтра в объект класс `tbb::pipeline` используется метод `add_filter`.
- ❑ Для запуска вычислений используется метод `run`.



Фильтр

- Класс фильтра **tbb::filter** является абстрактным, должен быть унаследован всеми фильтрами, реализованными пользователем.
- Класс фильтра должен содержать следующие методы:
 - **bool filter::is_serial() const** – возвращает тип фильтра: **true** – последовательный, **false** – параллельный;
 - **virtual void* filter::operator()(void* item)** – метод обработки элементов. Должен вернуть указатель на элемент, который будет обрабатываться следующим фильтром. Фильтр, использующийся в объекте класса **tbb::pipeline** должен вернуть NULL, если больше нет элементов для обработки.
 - **virtual filter::~~filter()** – деструктор .



Ядро библиотеки



Ядро библиотеки

- Кроме набора высокоуровневых алгоритмов, которые предназначены для упрощения разработки параллельных программ, библиотека ТВВ предоставляет возможность писать параллельные программы на низком уровне – уровне «логических задач», работа с которыми, тем не менее, более удобна, чем напрямую с потоками.



Логическая задача...

- *Логическая задача* в библиотеке ТВВ представлена в виде класса **tbb::task**. Этот класс является базовым при реализации задач, т.е. должен быть унаследован всеми пользовательскими логическими задачами.
- В дальнейшем под логической задачей будем понимать любой класс, который является потомком класса **tbb::task**.



Логическая задача

- ❑ Класс **tbb::task** содержит виртуальный метод **task::execute**, в котором выполняются вычисления.

```
task* task::execute()
```

- ❑ В этом методе производятся необходимые вычисления, после чего возвращается указатель на следующую задачу, которую необходимо выполнить. Если возвращается **NULL**, то из пула готовых к выполнению задач выбирается новая.



Пустая задача

- ❑ Библиотека TBB содержит специально реализованную «пустую» задачу (**tbb::empty_task**), которая часто оказывается полезной.
- ❑ Метод **task::execute** этой задачи не выполняет никаких вычислений.

```
class empty_task: public task
{
    task* execute()
    {
        return NULL;
    }
};
```



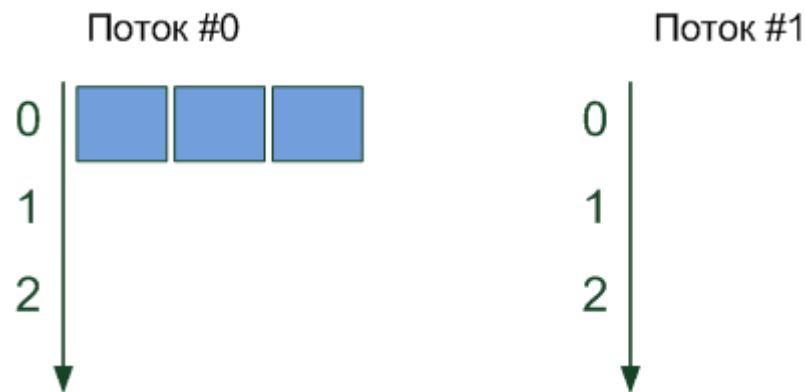
Алгоритм работы

- ❑ Каждый поток, созданный библиотекой, имеет свое множество (пул) готовых к выполнению задач.
- ❑ Это множество представляет собой динамический массив СПИСКОВ.



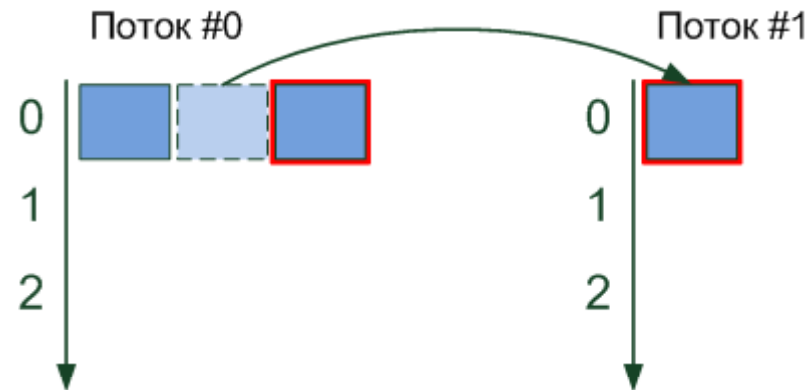
Алгоритм работы. Пример...

- Пусть в библиотеке создано два потока.
- Пусть в вычислительный алгоритм создаёт три задачи, в начальный момент времени они расположены на 0-ом потоке, пул задач потока #1 пуст.



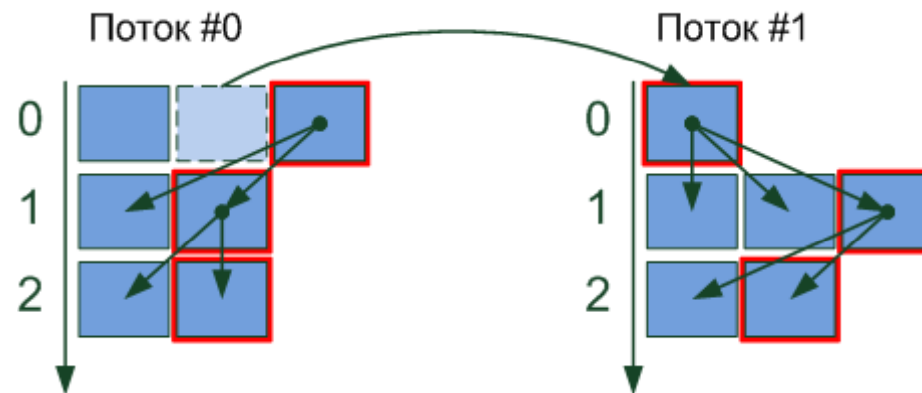
Алгоритм работы. Пример...

- Поток #0 начинает выполнять одну из задач.
- Т.к. пул задач потока #1 пуст, то он «изымает» одну из задач 0-го потока на выполнение.



Алгоритм работы. Пример...

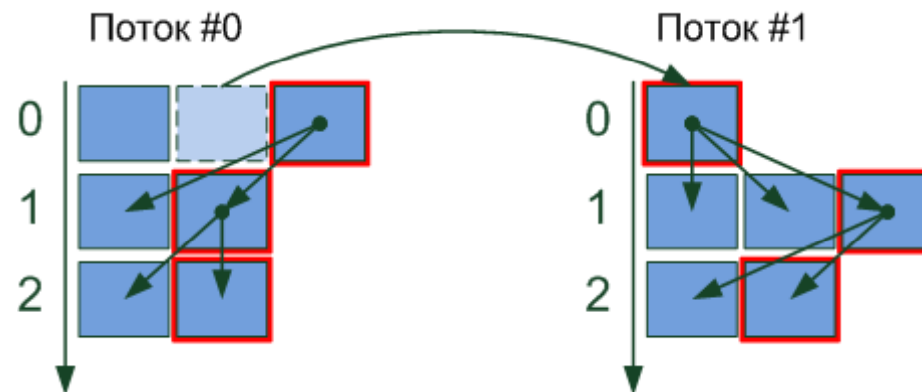
- ❑ Каждая из задач порождает 2-3 подзадачи (в данном примере).
- ❑ После завершения текущей задачи происходит выполнение одной из задач на самом высоком (по номеру) уровне в пуле потока.
- ❑ Потоки выполняют задачи параллельно.



Алгоритм работы. Пример

□ Поток #1

- Задача нулевого уровня порождает три подзадачи, которые размещаются на уровне 1, выполнение задачи на нулевом уровне завершается.
- Выполняется одна из задач 1-го уровня, она порождает две подзадачи 2-го уровня, выполнение задачи на 1-ом уровне завершается.
- Выполняются одна из задач 2-го уровня, подзадачи не создаются, выполнение задачи на 2-ом уровне завершается.
- Выполняется оставшаяся задача на 2-ом уровне.
- и т.д.



Атрибуты задачи

- Каждая задача имеет следующий набор связанных с ней атрибутов:
 - **owner** – поток, которому принадлежит задача.
 - **parent** – равен либо **NULL**, либо указателю на другую задачу, у которой поле **refcount** будет уменьшено на единицу после завершения текущей задачи. Для получения значения этого атрибута предназначен метод **parent**.
 - **depth** – глубина задачи в дереве задач. Получить значение этого атрибута можно с помощью метода **depth**, а установить с помощью **set_depth**.
 - **refcount** – число задач, у которых текущая задача указана в поле **parent**. Получить значение поля **refcount** можно с помощью метода **ref_count**, а установить с помощью **set_ref_count**. Здесь и далее обозначения **parent**, **depth** используются и как имена полей и как имена методов, с помощью которых можно получить их значения.



Алгоритм выполнения логической задачи

- После того как планировщик потоков назначает каждому потоку задачу на выполнение, происходит следующее:
 - Выполнение метода **task::execute** и ожидание его завершения.
 - Если для задачи не был вызван один из методов вида **task::recycle_* (recycle_as_child_of, recycle_to_reexecute, recycle_as_continuation, recycle_as_safe_continuation)**, то:
 - Если поле **parent** не **NULL**, то поле **parent->refcount** уменьшается на единицу. Если поле **parent->refcount** становится равным 0, то задача **parent** помещается в пул готовых к выполнению.
 - Вызов деструктора задачи.
 - Освобождение памяти, занимаемой задачей.
- Если для задачи был вызван один из методов **task::recycle_***, то задача повторно добавляется в пул готовых к выполнению.



Создание и уничтожение логических задач...

- Создание задачи должно осуществляться только с помощью оператора **new**, перегруженного в библиотеке ТВВ:
 - **new(task::allocate_root()) T** – выполняет создание «главной» задачи типа **T** со следующими атрибутами (**NULL, depth, 0**). Для запуска этого типа задач необходимо использовать метод **task::spawn_root_and_wait**.
 - **new(this.allocate_child()) T** – выполняет создание подчиненной задачи типа **T** для задачи **this** со следующими атрибутами (**this, depth + 1, 0**). Атрибуты задачи **this** (**parent, depth, refcount**) автоматически изменяются на (**parent, depth, refcount + 1**).



Создание и уничтожение логических задач

- Создание задачи должно осуществляться только с помощью оператора **new**, перегруженного в библиотеке ТВВ:
 - **new(this.allocate_continuation()) T** – выполняет создание задачи того же уровня, что и задача **this**. Атрибуты задачи **this (parent, depth, refcount)** автоматически изменяются на **(NULL, depth, refcount)**, новая задача создается со следующими атрибутами **(parent, depth, 0)**.
 - **new(this.task::allocate_additional_child_of(parent))** – выполняет создание подчиненной задачи для произвольной задачи, указанной в качестве параметра. Атрибуты задачи **parent (grandparent, depth, refcount)** автоматически изменяются на **(grandparent, depth, refcount + 1)**, новая задача создается со следующими атрибутами **(parent, depth + 1, 0)**.



Создание и уничтожение логических задач.

Пример

- Пример создания задачи

```
task* MyTask::execute()  
{  
    // ...  
    MyTask &t = *new (allocate_child()) MyTask();  
    // ...  
}
```

- Уничтожение задачи осуществляется автоматически с помощью виртуального деструктора.
- Можно уничтожить задачу вручную с помощью метода **task::destroy**. При этом поле **refcount** уничтожаемой задачи должно быть равно 0.

```
void task::destroy(task& victim)
```



Состояние логической задачи...

- ❑ В каждый момент времени задача может находиться в одном из 5 состояний.
- ❑ Состояние задачи изменяется при вызове методов библиотеки или в результате выполнения определенных действий (например, завершение выполнения метода **task::execute**).
- ❑ В библиотеке ТВВ реализован метод **task::state**, который возвращает текущее состояние задачи, для которой он был вызван. Данная информация может оказаться полезной при отладке приложений.

```
state_type task::state()
```



Состояние логической задачи

- Перечислимый тип **task::state_type** может принимать одно из следующих значений, которые отражают текущее состояние выполнения задачи:
 - **allocated** – задача только что была создана или был вызван один из методов вида **task::recycle_* (recycle_as_child_of, recycle_to_reexecute, recycle_as_continuation, recycle_as_safe_continuation)**;
 - **ready** – задача находится в пуле готовых к выполнению задач или в процессе перемещения в/из него;
 - **executing** – задача выполняется и будет уничтожена после завершения метода **task::execute**;
 - **freed** – задача находится во внутреннем списке библиотеки свободных задач или в процессе перемещения в/из него;
 - **reexecute** – задача выполняется и будет повторно запущена после завершения метода **task::execute**.



Контейнер задач

- Для удобства работы с набором задач библиотекой поддерживается класс **tbb::task_list**. Этот класс фактически представляет собой контейнер задач. Класс **tbb::task_list** содержит два основных метода:
 - **task_list::push_back(task& task)** – добавляет задачу в конец списка;
 - **task& task_list::pop_front()** – извлекает задачу из начала списка.



Планирование выполнения задач..

- Методы управления планированием и синхронизации задач:
 - **void task::set_ref_count(int count)** – устанавливает значение поля **refcount** равным **count**.
 - **void task::spawn(task& child)** – добавляет задачу в очередь готовых к выполнению и возвращает управление программному коду, который вызвал этот метод. Задачи **this** и **child** должны принадлежать потоку, который вызывает метод **spawn**. Поле **child.refcount** должно быть больше нуля. Перед вызовом метода **spawn** необходимо с помощью метода **task::set_ref_count** установить число подчиненных задач у задачи **parent**.



Планирование выполнения задач..

- Методы управления планированием и синхронизации задач:
 - **void task::wait_for_all()** – ожидает завершения всех подчиненных задач. Поле **refcount** должно быть равно числу подчиненных задач + 1.
- Типичный пример использования рассмотренных методов.

```
task* MyTask::execute()  
{  
    // ...  
    MyTask &t = *new (allocate_child()) MyTask();  
    set_ref_count(ref_count() + 2);  
    spawn(t);  
    wait_for_all();  
    // ...  
}
```



Планирование выполнения задач..

- Методы управления планированием и синхронизации задач:
 - **void task::spawn (task_list& list)** – добавляет список задач **list** в пул готовых к выполнению и возвращает управление программному коду, который вызвал этот метод. Алгоритм работы данного метода совпадает с последовательным вызовом метода **spawn** для каждой задачи из списка, но имеет более эффективную реализацию. Все задачи из списка **list** и задача **this** должны принадлежать потоку, который вызывает метод **task::spawn**. Поле **child.refcount** должно быть больше нуля для всех задач из списка. Значение поля **depth** у всех задач из списка должно быть одинаковым.



Планирование выполнения задач..

- Методы управления планированием и синхронизации задач:
 - **void task::spawn_and_wait_for_all(task& child)** – добавляет задачу в очередь готовых к выполнению и ожидает завершения всех подчиненных задач. Является аналогом последовательного вызова методов **spawn** и **wait_for_all**, но имеет более эффективную реализацию.
 - **void task::spawn_and_wait_for_all(task_list& list)** – добавляет список задач **list** в очередь готовых к выполнению и ожидает завершения всех подчиненных задач. Является аналогом последовательного вызова методов **spawn** и **wait_for_all**, но имеет более эффективную реализацию.



Планирование выполнения задач..

- Методы управления планированием и синхронизации задач:
 - **static void task::spawn_root_and_wait(task& root)** – выполняет запуск задачи **root**. Память для задачи должна быть выделена с помощью **task::allocate_root()**.
 - **static void task::spawn_root_and_wait(task_list& root_list)** – выполняет параллельный (если возможно) запуск каждой задачи из списка **root_list**, с помощью метода **spawn_root_and_wait**.
- Типичный пример создания и запуска “главной” задачи.

```
int main {  
    //...  
    MyTask &t = *new (task::allocate_root()) MyTask();  
    task::spawn_root_and_wait(t);  
    //...  
}
```



Повторное использование задач...

- Библиотека предоставляет набор методов, которые позволяют повторно использовать задачи для вычислений, что способствует многократному использованию, выделенных ресурсов и уменьшению накладных расходов.
- Методы повторного использования задач:
 - **void task::recycle_as_continuation()** – изменяет состояние задачи на **allocated**, таким образом после завершения метода **task::execute** задача не уничтожается, а остается в пуле готовых к выполнению. Метод должен быть вызван в теле метода **task::execute**. Значение поля **refcount** должно быть равно числу подчиненных задач и после того как метод **task::execute** закончит выполнение должно быть больше нуля (все потомки не должны закончить выполнение). Если это обеспечить нельзя, то необходимо использовать метод **task::recycle_as_safe_continuation**.



Повторное использование задач...

- Методы повторного использования задач:
 - `void task::recycle_as_safe_continuation()` – аналогичен по функциональности методу `task::recycle_as_continuation`. Значение поля `refcount` должно быть равно числу подчиненных задач + 1. Метод должен быть вызван в теле метода `task::execute`.
 - `void task::recycle_to_reexecute()` – запускает текущую задачу на повторное выполнение после завершения выполнения метода `task::execute`. Метод должен быть вызван в теле метода `task::execute`. Метод `task::execute` должен вернуть указатель на другую (не равную `this`) задачу.



Повторное использование задач...

- Методы повторного использования задач:
 - **void task::recycle_as_child_of(task& parent)** – устанавливает текущую задачу подчиненной для **parent**. После завершения метода **task::execute** задача не уничтожается, а остается в пуле готовых к выполнению. Этот метод должен быть вызван в теле метода **task::execute**.
- Пример использования метода **recycle_as_child_of**.

```
task* MyTask::execute()  
{  
    //...  
    empty_task& t = *new( allocate_continuation()) empty_task;  
    recycle_as_child_of(t);  
    t.set_ref_count(1);  
    //...  
}
```



Планирование использования задач

- В тех случаях, когда необходимо изменить алгоритм планирования может оказаться полезным изменять значение поля **depth**. Для этого используются следующие методы:
 - **depth_type task::depth()** – возвращает текущее значение поля **depth**;
 - **void task::set_depth(depth_type new_depth)** – устанавливает значение поля **depth** равным **new_depth**. Значение **new_depth** должно быть неотрицательным;
 - **void task::add_to_depth(int delta)** – устанавливает значение поля **depth** равным **depth+delta**. Значение **depth+delta** должно быть неотрицательным.



Соответствие потоков и логических задач

- Методы класса **tbb::task**, которые обеспечивают связь задач и потоков, на которых они выполняются:
 - **static task& task::self()** – возвращает задачу, принадлежащую текущему потоку;
 - **task* task::parent()** – возвращает значение поля **parent**. Для задач, созданных с помощью **task::allocate_root()**, значение поля **parent** не определено;
 - **bool task::is_stolen_task()** – возвращает **true**, если у задач **this** и **parent** значение полей **owner** не совпадают.



Распараллеливание рекурсии

- Одним из достоинств задач является, то, что с их помощью можно достаточно легко реализовывать параллельные версии рекурсивных вычислений.
- Продемонстрируем это на примере «задачи о ферзях».
 - Пусть дана шахматная доска размером n на n .
 - Каждый ферзь «бьет» все фигуры, расположенные по горизонтали, вертикали и обеим диагоналям.
 - Необходимо подсчитать число возможных вариантов размещения n ферзей на этой доске так, чтобы они не «били» друг друга.



Задача о ферзях...

□ Поля класса логической задачи

```
class Backtracker: public task
{
private:
    concurrent_vector<int> placement;
                                // Размещение ферзей.
                                // Ферзь placement[i] расположен
                                // на i-ой вертикале

    int position;                // Позиция ферзя для проверки
    const int size;              // Размер поля
    static spin_mutex myMutex;   // Мьютекс для блокировки доступа к
                                // переменной count

public:
    static int count;            // Число вариантов размещения
                                // ферзей на доске

};
```



Задача о ферзях...

□ Конструктор и метод `execute`

```
class Backtracker: public task
{
public:
    Backtracker(concurrent_vector<int> &t_placement, int t_position, int t_size):
        placement(t_placement), position(t_position), size(t_size)
    {}

    task* execute()
    {
        for (int i = 0; i < placement.size(); i++)
            // Проверка горизонтальных и диагональных траекторий на пересечение
            // нового ферзя с уже стоящими
            if ((placement[i] == position) ||
                (position - placement.size() == (placement[i] - i) ||
                 position + placement.size() == (placement[i] + i)))
                return NULL;

        placement.push_back(position); // Позицию можно добавить
        //...
    }
};
```



Задача о ферзях

□ Метод `execute` (продолжение)

```
//...
if(placement.size() == size) // Расстановка ферзей на всем поле получена
{
    spin_mutex::scoped_lock lock(myMutex);
    count++;
}
else
{
    empty_task& c = *new(allocate_continuation()) empty_task;
    recycle_as_child_of(c);    c.set_ref_count(size);

    for (int i = 0; i < size - 1; i++)
    {
        Backtracker& bt =
            *new (c.allocate_child()) Backtracker(placement, i, size);
        c.spawn(bt);
    }
    position = size - 1; // Используем текущую "задачу" в очередной итерации
    return this;
}
return NULL;
```



Примитивы синхронизации



Синхронизация

- ❑ Одной из основных задач при написании параллельных программ является задача синхронизации.
- ❑ При работе приложения в несколько потоков могут возникать ситуации, при которых один поток «ожидает» данных (результатов вычислений) от другого.
- ❑ В этом случае появляется потребность в синхронизации выполнения потоков.



“Гонки данных”

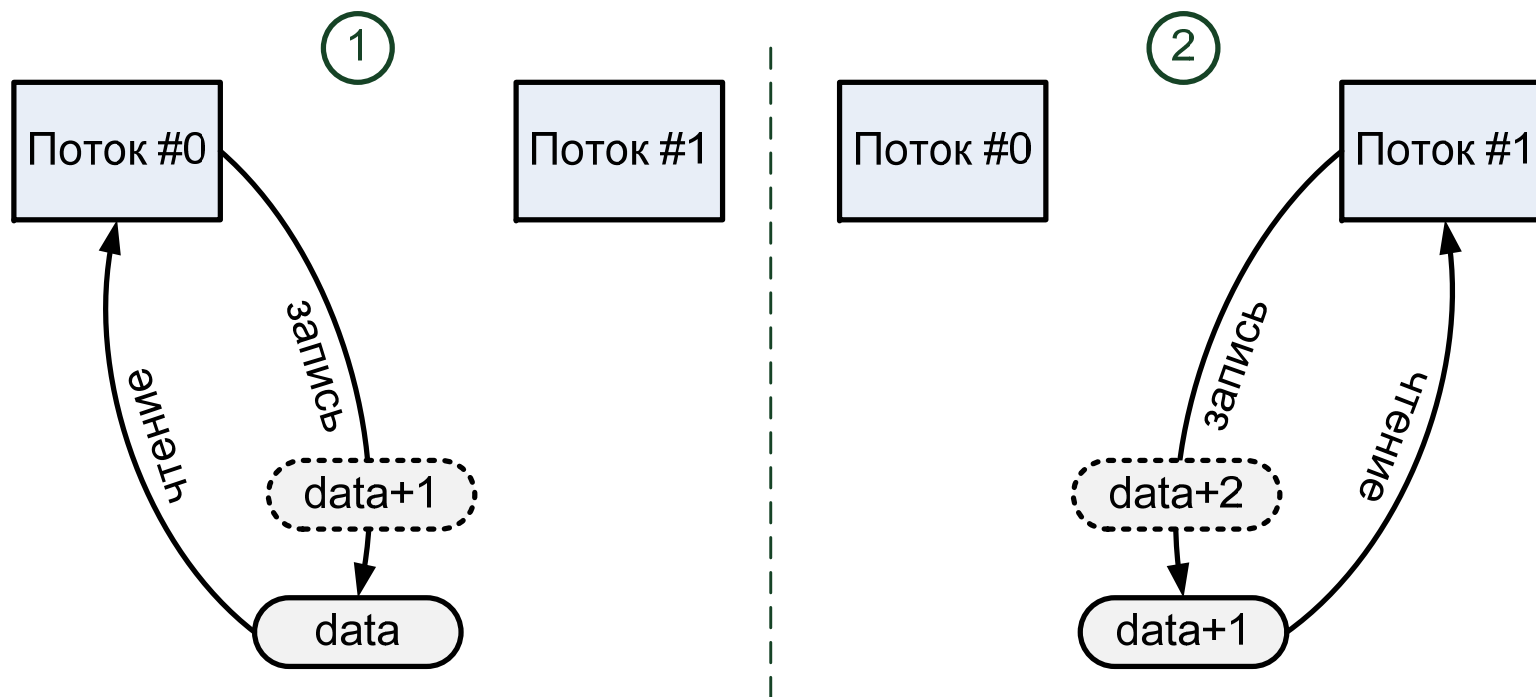
- ❑ Рассмотрим типичную ситуацию, в которой необходима синхронизация, называемую «гонкой данных».
- ❑ Пусть есть общая переменная **data**, доступная нескольким потокам для чтения и записи.
- ❑ Каждый поток должен выполнить инкремент этой переменной (то есть выполнить код **data++**).
- ❑ Для этого процессору необходимо выполнить три операции: чтение значения переменной из оперативной памяти в регистр процессора, инкремент регистра, запись посчитанного значения в переменную (оперативную память).



“Гонки данных”.

Ожидаемая реализация

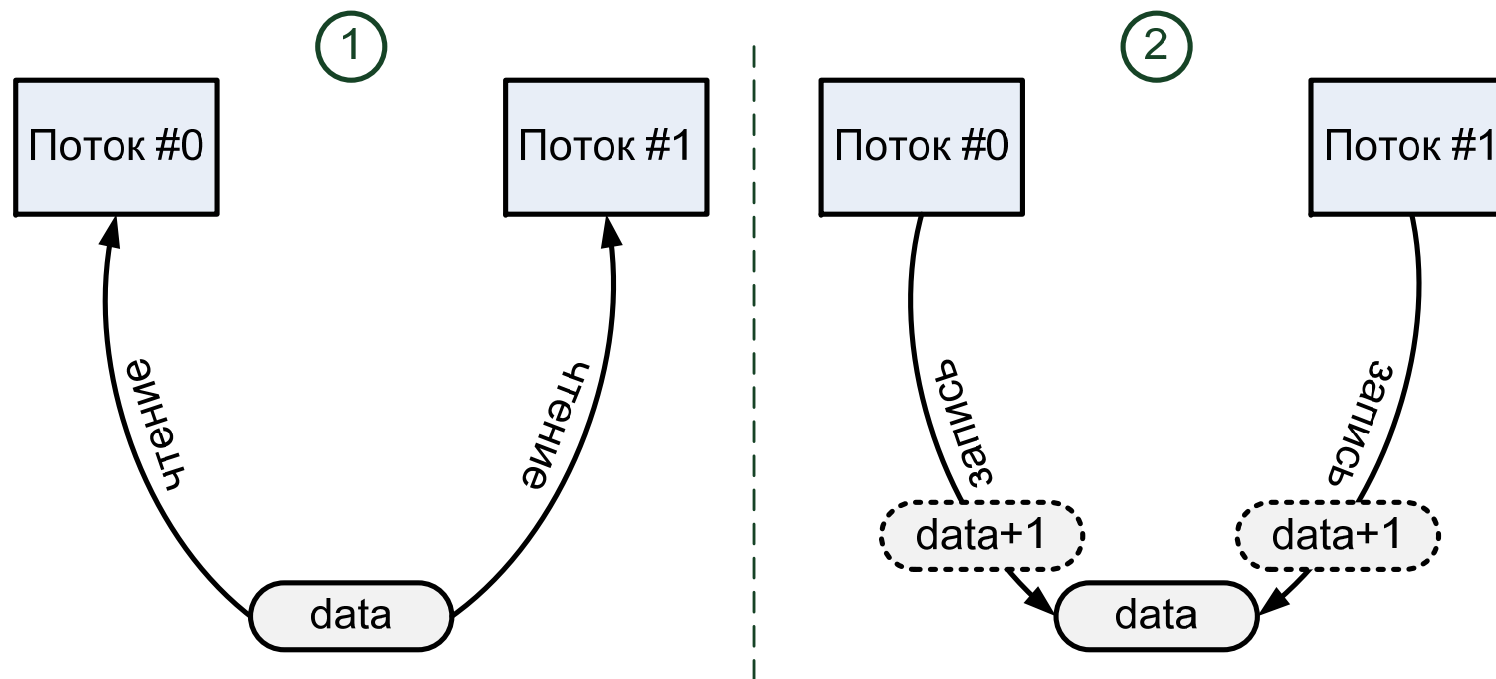
- ❑ Сначала поток 0 выполняет чтение переменной, инкремент регистра и запись его значения в переменную, а потом поток 1 выполнит ту же последовательность действий.
- ❑ Таким образом, после завершения работы приложения значение общей переменной будет равно **data+2**.



“Гонки данных”.

Менее ожидаемая реализация...

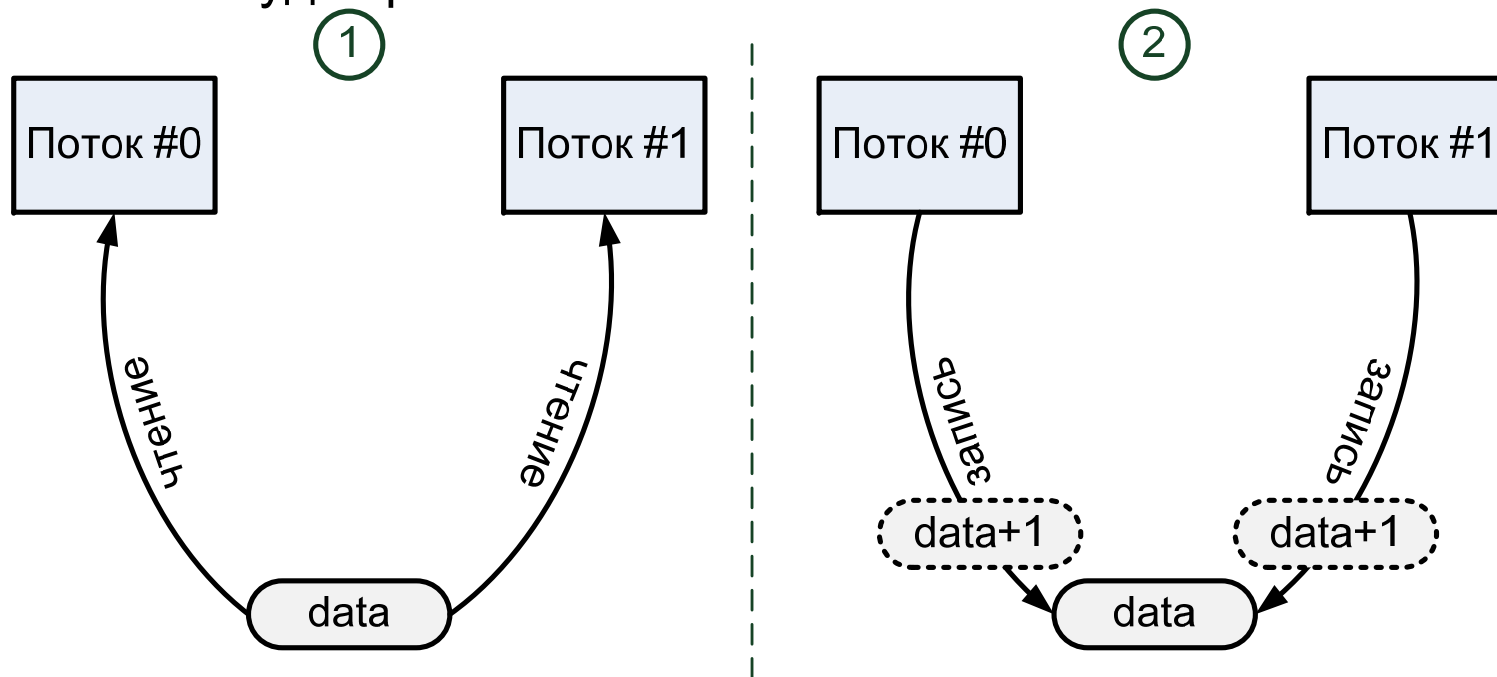
- ❑ Поток 0 выполняет чтение значения переменной в регистр и инкремент этого регистра, и в этот же момент времени поток 1 выполняет чтение переменной **data**.
- ❑ Т.к. для каждого потока имеется свой набор регистров, то поток 0 продолжит выполнение и запишет в переменную значение **data+1**.



“Гонки данных”.

Менее ожидаемая реализация

- ❑ Поток 1 также выполнит инкремент регистра (значение переменной **data** было прочитано из оперативной памяти ранее до записи потоком 0 значения **data+1**) и сохранит значение **data+1** (свое) в общую переменную.
- ❑ Таким образом, после завершения работы приложения значение переменной будет равно **data+1**.



“Гонки данных”

- ❑ Обе реализации могут наблюдаться как на многоядерной (многопроцессорной) системе, так и на однопроцессорной, одноядерной.
- ❑ Таким образом, в зависимости от порядка выполнения команд результат работы приложения может меняться.
- ❑ Возникает необходимость в механизме взаимного исключения, обеспечивающем синхронизацию выполнения потоков, с помощью которого можно было бы обеспечить выполнение части кода не более, чем одним потоком в каждый момент времени.



Примитивы синхронизации

- Основным способом решения задачи взаимного исключения является использование примитивов синхронизации:
 - *мьютекс* (*mutex*);
 - шаблонный класс **tbb::atomic**.



Мьютекс

- ❑ Мьютекс предназначен для того, чтобы критичный (требующий синхронизации) участок программного кода выполнялся ровно одним потоком.
- ❑ Мьютекс может находиться в одном из двух состояний: «свободен» и «захвачен».
- ❑ Любой поток может захватить мьютекс, переведя его из состояния «свободен» в состояние «захвачен».
- ❑ Если какой либо поток пытается захватить мьютекс, находящийся в состоянии «захвачен», то выполнение программного кода приостанавливается до тех пор, пока мьютекс не будет переведен в состояние «свободен».



Типы мьютексов

- Библиотека содержит три типа мьютексов:
 - **mutex** – мьютекс операционной системы;
 - **spin_mutex** – мьютекс, выполняющий активное ожидание;
 - **queuing_mutex**.



Мьютекс операционной системы

- ❑ **mutex** – мьютекс операционной системы, представляет собой обертку над примитивами синхронизации операционной системы (в операционных системах семейства **Microsoft Windows** в качестве основы используются критические секции).
- ❑ Т.к. данный тип мьютекса реализуется с помощью объектов операционной системы, то поток, пытающийся захватить мьютекс, который находится в состоянии «захвачен», переходит в состояние ожидания, а операционная система передает управление другим потокам.
- ❑ После освобождения мьютекса операционная система переводит поток в состояния готового к выполнению.
- ❑ Таким образом, время прошедшее после освобождения мьютекса и того момента когда поток получит управление может оказаться достаточно большим.



Мьютекс активного ожидания

- ❑ **spin_mutex** – мьютекс, выполняющий активное ожидание.
- ❑ Поток, который пытается захватить этот тип мьютекса, остается активным.
- ❑ Если поток пытается захватить мьютекс, который находится в состоянии «захвачен», то поток продолжает пытаться захватить мьютекс до тех пор, пока мьютекс не освободится.
- ❑ Таким образом, сразу после освобождения мьютекса один из ожидающих потоков начнет выполнение критического кода.
- ❑ Этот тип мьютекса желательно использовать, когда время выполнения критического участка кода мало.



Мьютекс `tbb::queuing_mutex`

- ❑ **`queuing_mutex`** – ЭТОТ ТИП МЬЮТЕКСА ВЫПОЛНЯЕТ АКТИВНОЕ ОЖИДАНИЕ ЗАХВАТА МЬЮТЕКСА С СОХРАНЕНИЕМ ОЧЕРЕДНОСТИ ПОТОКОВ, Т.Е. ВЫПОЛНЕНИЕ ПОТОКОВ ПРОДОЛЖАЕТСЯ В ТОМ ПОРЯДКЕ, В КОТОРОМ ОНИ ПЫТАЛИСЬ ЗАХВАТИТЬ МЬЮТЕКС.
- ❑ ЭТОТ ТИП МЬЮТЕКСОВ ОБЫЧНО ЯВЛЯЕТСЯ НАИБОЛЕЕ МЕДЛЕННЫМ, Т.К. ЕГО РАБОТА НЕСЕТ ДОПОЛНИТЕЛЬНЫЕ НАКЛАДНЫЕ РАСХОДЫ.



Класс мьютекса

- Каждый тип мьютексов реализован в виде класса. Обозначим через **M** класс, реализующий мьютекс.
- Класс **M** содержит всего два метода:
 - **M::M()** – конструктор. Создает мьютекс, находящийся в «свободном» состоянии;
 - **M::~~M()** – деструктор. Уничтожает мьютекс, находящийся в «свободном» состоянии.



Захват/освобождение мьютекса...

- Для захвата/освобождения мьютекса предназначен класс **scoped_lock**, реализованный в теле каждого класса мьютекса.
- Класс **scoped_lock** содержит следующие методы:
 - **M::scoped_lock::scoped_lock()** – конструктор. Создает объект класса **scoped_lock** без захвата мьютекса.
 - **M::scoped_lock::scoped_lock(M&)** – конструктор с параметром. Создает объект класса **scoped_lock** и переводит мьютекс в состояние «захвачен».
 - **M::scoped_lock::~~scoped_lock()** – деструктор. Переводит мьютекс в состояние «свободен», если он был захвачен.



Захват/освобождение мьютекса

- Класс `scoped_lock` содержит следующие методы:
 - `M::scoped_lock::acquire(M&)` – метод захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен». Если мьютекс уже находится в состоянии «захвачен», то поток блокируется.
 - `bool M::scoped_lock::try_acquire(M&)` – метод неблокирующего захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен», метод возвращает `true`. Если мьютекс уже находится в состоянии «захвачен», то ничего не происходит, метод возвращает `false`.
 - `M::scoped_lock::release()` – метод освобождения мьютекса. Переводит мьютекс в состояние «свободен», если он был захвачен.



Мьютекс. Пример

- Пример, выполняющий подсчет количества вызовов метода **operator()** с использованием мьютексов.

```
int Functor::count = 0;
mutex Functor::myMutex;

class Functor
{
private:
    static int count;
    static mutex myMutex;
public:
    // Методы функтора
    //...
    void operator()(const blocked_range<int>& Range) const
    {
        mutex::scoped_lock lock;
        lock.acquire(myMutex);
        count++;
        lock.release();
    }
};
```



Мьютексы читателей-писателей...

- ❑ Помимо обычных мьютексов в библиотеке TBB реализованы мьютексы читателей-писателей.
- ❑ Эти мьютексы содержат дополнительный флаг **writer**. С помощью этого флага можно указать какой тип блокировки мьютекса нужно захватить: читателя (**writer=false**) или писателя (**writer=true**).
- ❑ Несколько «читателей» (потоки которые захватили блокировку читателя) при отсутствии блокировки писателя могут выполнять критичный код одновременно.
- ❑ Если поток захватил мьютекс писателя, то все остальные потоки блокируется при попытке захвата мьютекса.



Мьютексы читателей-писателей...

- ❑ Библиотека содержит два типа мьютексов читателей-писателей: **spin_rw_mutex** и **queuing_rw_mutex**. Эти мьютексы по своим характеристикам полностью совпадают с мьютексами: **spin_mutex** и **queuing_mutex**.
- ❑ Отличия мьютексов читателей-писателей от остальных заключаются только в методах класса **scoped_lock**.



Мьютексы читателей-писателей...

- Класс **scoped_lock** для мьютексов читателей-писателей содержит следующие методы:
 - **M::scoped_lock::scoped_lock()** – конструктор. Создает объект класса **scoped_lock** без захвата мьютекса.
 - **M::scoped_lock::scoped_lock(M&, bool write=true)** – конструктор с параметром. Создает объект класса **scoped_lock** и переводит мьютекс в состояние «захвачен». Второй параметр указывает тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**).
 - **M::scoped_lock::~~scoped_lock()** – деструктор. Переводит мьютекс в состояние «свободен», если он был захвачен.



Мьютексы читателей-писателей...

- Класс **scoped_lock** для мьютексов читателей-писателей содержит следующие методы:
 - **M::scoped_lock::acquire(M&, bool write=true)** – метод захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен». Вторым параметром указывается тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**). Если мьютекс уже находится в состоянии «захвачен», то поток блокируется.
 - **bool M::scoped_lock::try_acquire(M&, bool write=true)** – метод не блокирующего захвата мьютекса. Вторым параметром указывается тип блокировки, которую необходимо захватить: читателя (**writer=false**) или писателя (**writer=true**). Переводит мьютекс из состояния «свободен» в состояние «захвачен», метод возвращает **true**. Если мьютекс уже находится в состоянии «захвачен», то ничего не происходит, метод возвращает **false**.



Мьютексы читателей-писателей

- Класс **scoped_lock** для мьютексов читателей-писателей содержит следующие методы:
 - **M::scoped_lock::release()** – метод освобождения мьютекса. Переводит мьютекс в состояние «свободен», если он был захвачен.
 - **M::scoped_lock::upgrade_to_writer()** – изменяет тип блокировки на блокировку писателя.
 - **M::scoped_lock::downgrade_to_reader()** – изменяет тип блокировки на блокировку читателя.



Шаблонный класс `tbb::atomic`

- ❑ Методы этого класса являются атомарными, т.е. несколько потоков не могут одновременно выполнять методы этого класса.
- ❑ Если потоки пытаются одновременно выполнить методы класса `tbb::atomic`, то один из потоков блокируется и ожидает завершения выполнения метода другим.



Шаблонный класс `tbb::atomic`. Пример

- Пример, выполняющий подсчет количества вызовов метода **`operator()`** с использованием класса **`tbb::atomic`**.

```
atomic<int> Functor::count;

class Functor
{
private:
    static atomic<int> count;

public:
    // Методы функтора
    //...
    void operator()(const blocked_range<int>& Range) const
    {
        count++;
    }
};
```



Неявная синхронизация

- ❑ Многие функции и классы библиотеки ТВВ выполняют синхронизацию неявно.
- ❑ Например, функция **parallel_for** «ждет» завершения всех потоков, занятых вычислениями прежде, чем вернуть управление потоку, с которого она была вызвана.



Потокобезопасные контейнеры



Потокобезопасные контейнеры

- Библиотека TBB содержит реализацию трех контейнеров, которые являются аналогами контейнеров STL:
 - **tbb::concurrent_queue** – очередь;
 - **tbb::concurrent_hash_map** – хеш-таблица;
 - **tbb::concurrent_vector** – вектор.



Очередь...

- ❑ Т.к. контейнеры библиотеки ТВВ предназначены для параллельной работы, то их интерфейс отличается от интерфейса STL контейнеров.
- ❑ Отличия между **std::queue** и **tbb::concurrent_queue** заключаются в следующем:
 - Методы **queue::front** и **queue::back** не реализованы в **tbb::concurrent_queue**, т.к. их параллельное использование может быть небезопасно.
 - В отличие от STL тип **size_type** является знаковым.
 - Метод **concurrent_queue::size** возвращает разницу между числом операций добавления элементов и операций извлечения элементов. Если в момент вызова этого метода выполняются методы добавления/извлечения элементов, то они тоже учитываются.



Очередь

- Отличия между **std::queue** и **tbb::concurrent_queue** заключаются в следующем:
 - Для **tbb::concurrent_queue** разрешен вызов метода **pop** для пустой очереди. После его вызова поток блокируется до тех пор пока в очередь не положат элемент.
 - **tbb::concurrent_queue** содержит метод **pop_if_present**, который извлекает элемент из очереди, если в очереди есть элементы.



Приложение



Настройка среды разработки

- ❑ В меню **Tools** выберите пункт **Options....**
- ❑ В открывшемся окне выберите **Projects and Solutions\VC++ Directories.**
- ❑ В выпадающем списке **Show directories for** выберите пункт **Library files.**
- ❑ Нажмите левой кнопкой мыши на изображении папки и укажите путь к папке **lib** библиотеки **Intel Threading Building Blocks.** При установке по умолчанию этот путь будет **C:\Program Files\Intel\TBB\2.0\\vc8\lib** (где, **<arch>** должно быть **ia32** или **em64t** в зависимости от режима работы процессора).
- ❑ Нажмите **ОК.**



Сборка и настройка проекта...

- В настройках проекта необходимо указать библиотеку, с которой будет линковаться приложение: **tbb_debug.lib** или **tbb.lib**
 - **tbb_debug.lib** выполняет проверки корректности во время выполнения приложения и полностью поддерживается профилировщиком Intel® Thread Profiler.
 - **tbb.lib** имеет гораздо более эффективную реализацию функций и методов, чем **tbb_debug.lib**.
- При отладке приложения рекомендуется использовать библиотеку **tbb_debug.lib**, при сборке рабочей версии – **tbb.lib**.



Сборка и настройка проекта

- При использовании в приложении операторов выделения динамической памяти (аллокаторов), в настройках проекта необходимо указать библиотеку **tbbmalloc.lib** (или **tbbmalloc_debug.lib**). При отладке приложения рекомендуется использовать библиотеку **tbbmalloc_debug.lib**, при сборке рабочей версии – **tbbmalloc.lib**.



Подключение библиотек

- ❑ В окне **Solution Explorer** нажмите правой кнопкой мыши на названии проекта и выберите пункт **Properties**.
- ❑ Выберите пункт **Linker\Input** и в поле **Additional Dependencies** введите название библиотеки: **tbb_debug.lib** для **Debug** сборки или **tbb.lib** для **Release** сборки.



Запуск приложения

- Для работы приложения, использующего библиотеку TBB, необходимо иметь одну динамическую библиотеку: **tbb.dll** (при использовании **tbb.lib**) или **tbb_debug.dll** (при использовании **tbb_debug.lib**).
- При использовании в приложении операторов выделения динамической памяти (аллокаторов) необходимо дополнительно иметь динамическую библиотеку **tbbmalloc.dll** (при использовании **tbbmalloc.lib**) или **tbbmalloc_debug.dll** (при использовании **tbbmalloc_debug.lib**).



Заголовочные файлы...

Название функции/класса	Заголовочный файл
<code>aligned_space</code>	<code>aligned_space.h</code>
<code>atomic</code>	<code>atomic.h</code>
<code>blocked_range</code>	<code>blocked_range.h</code>
<code>blocked_range2d</code>	<code>blocked_range2d.h</code>
<code>cache_aligned_allocator</code>	<code>cache_aligned_allocator.h</code>
<code>concurrent_hash_map</code>	<code>concurrent_hash_map.h</code>
<code>concurrent_queue</code>	<code>concurrent_queue.h</code>
<code>concurrent_vector</code>	<code>concurrent_vector.h</code>
<code>tick_count</code>	<code>tick_count.h</code>
<code>mutex</code>	<code>mutex.h</code>
<code>parallel_for</code>	<code>parallel_for.h</code>
<code>parallel_reduce</code>	<code>parallel_reduce.h</code>
<code>parallel_scan</code>	<code>parallel_scan.h</code>



Заголовочные файлы

Название функции/класса	Заголовочный файл
parallel_sort	parallel_sort.h
parallel_while	parallel_while.h
partitioner	partitioner.h
pipeline	pipeline.h
queuing_mutex	queuing_mutex.h
queuing_rw_mutex	queuing_rw_mutex.h
scalable_allocator	scalable_allocator.h
spin_mutex	spin_mutex.h
spin_rw_mutex	spin_rw_mutex.h
split	tbb_stddef.h
task	task.h
task_scheduler_init	task_scheduler_init.h



Совместное использование с OpenMP

- Для использования ТВВ совместно с OpenMP на каждом потоке, созданном с помощью OpenMP (внутри параллельной секции), необходимо запустить планировщик потоков ТВВ.

```
int main()
{
    #pragma omp parallel
    {
        task_scheduler_init init;
        #pragma omp for
        for( int i=0; i<n; i++ )
        {
            // Можно использовать функции и классы библиотеки ТВВ
        }
    }
}
```



Оценка эффективности приложений

- ❑ Основным показателем эффективности приложения является время выполнения вычислительно трудоемких операций
- ❑ Для измерения времени в библиотеке TBB используется класс **tbb::tick_count**.
- ❑ Независимо от аппаратной конфигурации (многопроцессорности, многоядерности), измеряемое время является синхронным между потоками.
- ❑ В операционных системах семейства Microsoft Windows класс **tbb::tick_count** реализован с использованием функции **QueryPerformanceCounter**.



Измерение времени...

- Основной метод класса **tbb::tick_count** – метод **now**:
 - измеряет текущее значение времени;
 - статический метод;
 - возвращает экземпляр класса **tbb::tick_count**.

```
static tick_count tick_count::now()
```



Измерение времени

- ❑ Выполнив два замера времени (в начале и в конце измеряемого участка программы), необходимо вычесть два экземпляра класса **tbb::tick_count** и перевести интервал времени в секунды.
- ❑ Результатом вычитания двух экземпляров класса **tbb::tick_count** является экземпляр вспомогательного класса **tick_count::interval_t**, который представляет интервал времени.
- ❑ Для перевода интервала времени в секунды класс **tick_count::interval_t** содержит метод **seconds**.

```
double interval_t::seconds()
```



Измерение времени. Пример

- Типичная схема измерения времени выполнения вычислений.

```
void SomeFunction()
{
    tick_count start = tick_count::now();
    // Вычисления
    tick_count finish = tick_count::now();
    printf("Время вычислений = %f seconds\n",
          (finish - start).seconds());
}
```



Динамическое выделение памяти

- Обычные операторы выделения динамической памяти работают с общей кучей для всех потоков, что требует наличия синхронизации.
- Библиотека ТВВ содержит масштабируемые операторы выделения динамической памяти:

```
//Выделение памяти и заполнение нулевыми элементами
void* scalable_calloc(size_t nobj, size_t size);
//Освобождение памяти
void scalable_free(void* ptr);
//Выделение памяти
void* scalable_malloc(size_t size);
//Перевыделение памяти и заполнение нулевыми элементами
void* scalable_realloc(void* ptr, size_t size);
```



Использованные источники информации

- ❑ Официальный сайт Intel® Threading Building Blocks. – [<http://www.intel.com/software/products/tbb/>]
- ❑ Intel® Threading Building Blocks. Reference Manual. Version 1.6. – Intel Corporation, 2007.
- ❑ Intel® Threading Building Blocks. Tutorial. Version 1.6. – Intel Corporation, 2007.



Рекомендуемая литература

- ❑ Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley (русский перевод Эндриус Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
- ❑ Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
- ❑ Barbara Chapman, Gabriele Jost, Ruud van der Pas (2007). Using OpenMP: Portable Shared Memory Parallel Programming (Scientific Computation and Engineering).
- ❑ Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. – СПб: Питер, 2006.
- ❑ Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2006.
- ❑ Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2003.
- ❑ Рихтер Дж. Windows для профессионалов: создание эффективных Win32 приложений с учетом специфики 64-разрядной версии Windows/Пер. с англ. – 4-е изд. – СПб: Питер; М.: Издательско-торговый дом «Русская редакция», 2001.



Информационные ресурсы сети Интернет

- ❑ Страница библиотеки TBB на сайте корпорации Intel – [<http://www.intel.com/cd/software/products/asm-na/eng/294797.htm>].
- ❑ Сайт сообщества пользователей TBB – [<http://threadingbuildingblocks.org>].
- ❑ Сайт Лаборатории Параллельных информационных технологий НИВЦ МГУ – [<http://www.parallel.ru>].
- ❑ Официальный сайт OpenMP – [www.openmp.org].



Авторский коллектив

- ❑ Мееров Иосиф Борисович,
к.т.н., доцент кафедры Математического обеспечения ЭВМ
факультета ВМК ННГУ.
- ❑ Сысоев Александр Владимирович,
ассистент кафедры Математического обеспечения ЭВМ
факультета ВМК ННГУ.
- ❑ Сиднев Алексей Александрович,
ассистент кафедры Математического обеспечения ЭВМ
факультета ВМК ННГУ.

