«Нижегородский государственный университет им. Н.И. Лобачевского»
Факультет вычислительной математики и кибернетики
Кафедра математического обеспечения ЭВМ

Приоритетное направление «Стратегические информационные технологии»
Федеральная целевая программа развития образования на 2011-2015 годы
Проект: Подготовка и переподготовка профильных специалистов на базе центров образования и разработок в сфере информационных технологий в ПФО

Учебный курс «Технологии параллельного программирования»

Лекционные материалы

Сиднев А.А., Сысоев А.В., Мееров И.Б.

Нижний Новгород
2011 г.

Библиотека Intel Threading Building Blocks – краткое описание

Содержание

Разработка программного обеспечения, всегда бывшая делом непростым, проходит в настоящий момент очередной виток повышения сложности, вызванный повсеместным распространением многоядерных процессоров, использовать все возможности которых можно, лишь создавая многопоточные программы. С одной стороны, все необходимые для этого средства уже весьма давно существуют в распространенных операционных системах семейств Microsoft Windows и Unix/Linux, с другой – для программистов-прикладников использование этих механизмов по уровню удобства и объему необходимых знаний немногим легче, чем было когда-то программирование на ассемблере. Остро необходимы инструменты, берущие на себя по возможности большую часть задач, связанных с обслуживанием «параллельности» в многопоточных программах и дающих разработчику возможность сосредоточиться на решении конкретных прикладных задач. В настоящем разделе представлено краткое описание одного из инструментов, пытающегося «играть на указанном поле», – библиотеки Intel® Threading Building Blocks [1].

Возможности библиотеки, рассмотренные в настоящем описании, подробно изучаются в лабораторных работах «Распараллеливание циклов с использованием библиотеки Intel Threading Building Blocks на примере задачи матрично-векторного умножения» и «Использование механизма логических задач библиотеки Intel Threading Building Blocks на примере вычисления быстрого преобразования Фурье».

Назначение библиотеки Intel Threading Building Blocks

Intel® Threading Building Blocks (TBB) – библиотека, предназначенная для разработки параллельных программ для систем с общей памятью. В отличие от других известных подходов и инструментов: программирования непосредственно в потоках, использования OpenMP, – TBB и сама написана на языке C++ (в классах и шаблонах) и ее использование предполагает и дает возможность разработки параллельной программы в объектах. Кроме того, библиотека скрывает низкоуровневую работу с потоками, упрощая тем самым процесс создания параллельной программы.

Возможности библиотеки Intel Threading Building Blocks

В состав TBB 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.

Также библиотека содержит:

- потокобезопасные контейнеры (аналогичны контейнерам STL, за исключением того, что они оптимизированы для параллельных программ);
- операторы выделения динамической памяти (*аллокаторы*);
- примитивы синхронизации.

Полную информацию о составе и возможностях библиотеки TBB можно найти в [2, 3].

Описание библиотеки Intel Threading Building Blocks

TBB является межплатформенной библиотекой – на момент написания данного пособия существуют реализации под операционные системы Microsoft Windows, Linux, Mac OS. Кроме того, библиотека свободно распространяется в некоммерческих целях [12].

В отличие от OpenMP библиотека Intel® Threading Building Blocks не требует поддержки со стороны компилятора. Для сборки приложения необходима установленная версия TBB, а для запуска под операционной системой семейства Microsoft Windows достаточно иметь требуемые динамические библиотеки. Подробные сведения о сборке и запуске программы, использующей TBB, см. в приложении 2 «Сборка и настройка проекта».

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. 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.**

Прототип конструктора класса **tbb::task_scheduler_init** представлен ниже:

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

Доступны следующие варианты значений параметра **number_of_threads**:

- **task_scheduler_init::automatic** (как видно из прототипа, это значение по умолчанию). Библиотека автоматически определяет и создает оптимальное количество потоков для данной вычислительной системы. В приложениях с большим числом компонентов 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. Пример инициализации объекта класса **tbb::task_scheduler_init** данным способом:

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

- Положительное число – число потоков, которое будет создано библиотекой. Потоки создаются сразу после вызова конструктора. Пример инициализации экземпляра класса **tbb::task_scheduler_init** данным способом:

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

- **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 этого метода имеет те же варианты, что и аргумент конструктора класса **tbb::task_scheduler_init.** Пример инициализации объекта класса **tbb::task_scheduler_init** данным способом:

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

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**. Также можно деактивировать объект класса **tbb::task_scheduler_init** в любой требуемый момент, чтобы освободить ресурсы системы под другие нужды. Для этих целей существует метод **task_scheduler_init::terminate**. Его прототип представлен ниже:

```
void terminate();
```

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

Если в приложении уже активен один экземпляр класса **tbb::task_scheduler_init,** то при создании нового объекта его параметры (число потоков) будут проигнорированы. Поэтому 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.

Рассмотренные операции требуют значительных временных ресурсов, поэтому типичная схема работы с объектом класса **tbb::task_scheduler_init** – инициализация в начале работы, деинициализация в конце работы приложения. Ниже приведен пример типичной структуры TBB-программы.

```
#include "tbb/task_scheduler_init.h"
// Подключение необходимых заголовочных файлов
using namespace tbb;

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

В примере для создания экземпляра класса **tbb::task_scheduler_init** был подключен заголовочный файл **task_scheduler_init.h.** Аналогичное происходит со всеми остальными функциями/классами библиотеки – воспользоваться ими можно, подключив соответствующий их названию заголовочный файл. Полный список функций/классов и соответствующих им заголовочных файлов представлен в приложении 1 «Заголовочные файлы библиотеки TBB».

Более сложный пример работы с 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
```

Библиотека TBB может использоваться совместно с библиотекой OpenMP. Для этого на каждом потоке, созданном с помощью OpenMP (внутри параллельной секции), необходимо создать активный объект класса **tbb::task_scheduler_init** библиотеки TBB. Более подробная информация об этом представлена в приложении 3 «Совместное использование с OpenMP».

Распараллеливание простых циклов

Циклы с известным числом повторений

Вычисления с заранее определенным числом итераций обычно происходят с использованием цикла **for**. Библиотека TBB дает возможность реализовать параллельную версию таких вычислений. Для этого библиотека предоставляет шаблонную функцию **tbb::parallel_for.** Функция **tbb::parallel_for** имеет несколько прототипов. Ниже будет разобран наиболее часто использующийся вариант:

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

Как видно из прототипа, функция **tbb::parallel_for** имеет два обязательных шаблонных параметра. Первый параметр представляет итерационное пространство – класс специального вида, задающий количество итераций цикла. Второй параметр – функтор[1], класс, реализующий вычисления цикла через метод **body::operator().** Последний параметр является необязательным и определяет стратегию планирования вычислений. Более подробная информация об этом параметре представлена в разделе «Планирование вычислений».

1.1.1.1. Итерационное пространство

Первый аргумент функции **tbb::parallel_for** – итерационное пространство.

Библиотека TBB содержит несколько реализованных итерационных пространств: одномерное итерационное пространство **tbb::blocked_range**, двумерное итерационное пространство **tbb::blocked_range2d** и трёхмерное итерационное пространство **tbb::blocked_range3d.** Пользователь библиотеки может реализовать и свои итерационные пространства.

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. (список требований, предъявляемых к шаблону, представлен в [3]).

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 границы полуинтервала **[my_begin,**

[1] В C++ *функторами*, или *функциональными классами*, называют классы специального вида, основная функциональность которых сосредоточена в методе operator(). Функторы активно используются во многих библиотеках классов, и TBB не является исключением.

my_end). Поле **my_grainsize** имеет целый тип и задает размер порции вычислений. Прототип основного конструктора класса **blocked_range**:

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

Основной операцией любого итерационного пространства является его *расщепление*. Операция расщепления выполняется с использованием *конструктора расщепления*, задача которого разделить итерационное пространство на два подмножества. Для **tbb::blocked_range** разделение итерационного пространства выполняется на два подмножества равного (с точностью до округления) размера. Например, итерационное пространство **a**, созданное представленным ниже способом, 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 (рис. 1). Значение поля **my_grainsize** не изменится.

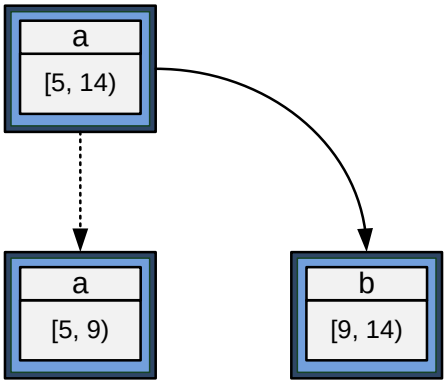

Рис. 1. Расщепление итерационного пространства blocked_range(5,14)

Рассмотрим пример использования одномерного итерационного пространства:

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

Представленный пример является полным аналогом следующего:

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

Двумерное итерационное пространство **tbb::blocked_range2d** является полным аналогом одномерного за исключением того, что оно задает двумерный полуинтервал вида **[x1, x2)×[y1, y2).**

Пользователю библиотеки TBB предоставляется возможность создать свое итерационное пространство. 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**.
- **bool is_divisible()** – метод проверки на возможность разделения итерационного пространства; если разделение возможно, то функция должна вернуть **true**;
- **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. Реализация этого класса в библиотеке TBB:

```
namespace tbb
{
```

```
  class split
  {
  };
}
```

Пример реализации простого одномерного итерационного пространства представлен ниже; конструктор копирования и деструктор явно не реализованы, т.к. их реализация по умолчанию является корректной:

```
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;
  }
};
```

Заметим, что в классе **SimpleRange** мы не объявили 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. Более подробно о порядке обработки итерационного пространства см. «Планирование вычислений».

1.1.1.2. Функтор

Второй аргумент функции **tbb::parallel_for** – функтор. Функтор – это класс специального вида, который выполняет необходимые вычисления с помощью метода **operator().** В первом приближении можно считать, что функтор получается в результате трансформации тела цикла в класс.

Функтор для функции **tbb::parallel_for** должен содержать следующие методы:

- конструктор копирования, необходимый 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.

Аргументом последнего метода является итерационное пространство. Прототип метода:

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

Метод **operator()** является основным в функторе. Метод объявлен константным, поскольку не нуждается в изменении значений полей функтора, если таковые в нем имеются. Ниже мы убедимся в этом факте на конкретном примере.

Одним из примеров «хорошего» параллелизма является задача, в которой итерации цикла могут выполняться без взаимной синхронизации. В качестве таковой рассмотрим задачу умножения матрицы на вектор. В данной задаче:

- операции скалярного умножения векторов, на которых основаны вычисления, выполняются без синхронизаций;
- запись результатов умножения происходит ровно один раз для каждого скалярного умножения векторов, в дальнейших вычислениях уже посчитанные результаты не используются;
- данные для вычислений (элементы массива и вектора) не изменяются во время вычислений.

Функтор является функциональным классом и не должен содержать в себе ни обрабатываемые данные, ни получаемый результат. Именно поэтому все поля представленного функтора, умножающего матрицу на вектор, являются указателями на внешние данные, кроме числа столбцов матрицы. Для последнего поля сделано исключение только потому, что на его хранение в виде копии требуется столько же памяти (размер типа **int**), сколько и на хранение указателя. Инициализация полей осуществляется с помощью конструктора. Как мы увидим далее в процессе работы функции **tbb::parallel_for** на основе первоначально переданного в нее функтора создается некоторое количество копий. Естественно все они благодаря тому, что поля-указатели адресуют одни и те же данные, будут «разделять» и исходные данные и результат, что, собственно говоря, нам и нужно.

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

Заметим, что в классе **VectorsMultiplicator** не реализован ни конструктор копирования, ни деструктор, несмотря на наличие полей-указателей. Над объяснением этого факта предлагаем читателю подумать самостоятельно.

1.1.1.3. Планирование вычислений

Алгоритм работы функции **tbb::parallel_for** устроен таким образом, что планирование вычислений осуществляется динамически, т.е. на этапе выполнения. Определяющим моментом планирования является то, как реализовано итерационное пространство, и какая стратегия планирования используется. Стратегия планирования задаётся через третий параметр функции **tbb::parallel_for**:

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

В библиотеке TBB реализовано три класса, которые определяют стратегию планирования: **simple_partitioner**, **auto_partitioner**, **affinity_partitioner**. По умолчанию используется **auto_partitioner**.

Рассмотрим алгоритм работы функции **tbb::parallel_for** при использовании одномерного итерационного пространства и стратегии **simple_partitioner**.

Одним из полей одномерного итерационного пространства является размер порции вычислений – **grainsize**. Его значение является определяющим при планировании вычислений. Функция **tbb::parallel_for** распределяет на выполнение между всеми потоками 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 <общее число итераций>/<число потоков>, то каждый поток, скорее всего (так как планирование осуществляется динамически, то точно сказать нельзя), выполнит одинаковое число итераций, равное **grainsize**. Если

grainsize меньше, чем <общее число итераций>/<число потоков>, то планировщик потоков распределит итерации между потоками по специальному алгоритму.

Значение **grainsize** выбирается разработчиком приложения. При этом 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, так как остальным ее просто не достанется из-за большого значения **grainsize.** Поэтому необходимо устанавливать маленькие значения **grainsize** для большей масштабируемости приложения. Работа планировщика потоков занимает определенное время, поэтому, чем меньше значение **grainsize,** тем больше времени потребуется функции **tbb::parallel_for** на распределение заданий. Таким образом, при очень малых значениях **grainsize** приложение будет обладать очень хорошей масштабируемостью, но при этом будет работать очень неэффективно из-за больших накладных расходов на работу планировщика. При очень больших значениях **grainsize** приложение будет работать максимально эффективно, но его масштабируемость будет очень плоха.

Итог:

- параметр **grainsize** не должен быть слишком маленьким, так как это может негативно отразиться на времени работы приложения (большие накладные расходы на работу функции **tbb::parallel_for**);
- параметр **grainsize** не должен быть слишком большим, так как это может негативно отразиться на масштабируемости приложения.

Большинство вычислительных систем очень сложны, чтобы можно было теоретически подобрать оптимальное значение **grainsize**, поэтому рекомендуется подбирать его экспериментально.

Алгоритм экспериментального подбора значения **grainsize** следующий [3]:

1. Установите значение **grainsize** достаточно большим, например равным размеру итерационного пространства в случае использования класса **blocked_range.**
2. Запустите приложение в один поток, измерьте время его выполнения.
3. 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%.

Рассмотрим алгоритм работы **tbb::parallel_for** с точки зрения распределения вычислений на следующем примере:

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

На первом шаге имеется функтор **body** и одномерное итерационное пространство, размер которого равен 14 - 5 = 9. Это значение больше, чем размер порции, равный 2, поэтому функция **tbb::parallel_for** расщепляет итерационное пространство на два, одновременно создавая для нового итерационного пространства собственный функтор (через конструктор копирования) и меняя, как мы уже отмечали выше, 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. Алгоритм работы **tbb::parallel_for** представлен на рис. 2. 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.

Как видно из рис. 2, после первого шага будут существовать два непересекающихся итерационных пространства. Если в библиотеке создано больше одного потока, то дальнейшая обработка полученных поддеревьев будет происходить параллельно. При этом процесс вычисления является недетерминированным и может выполняться так, как показано на рис. 3.

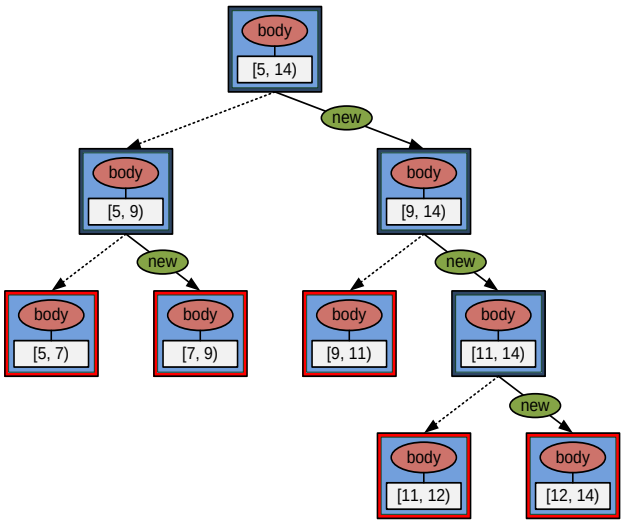

Рис. 2. Алгоритм работы функции **`tbb::parallel_for`**

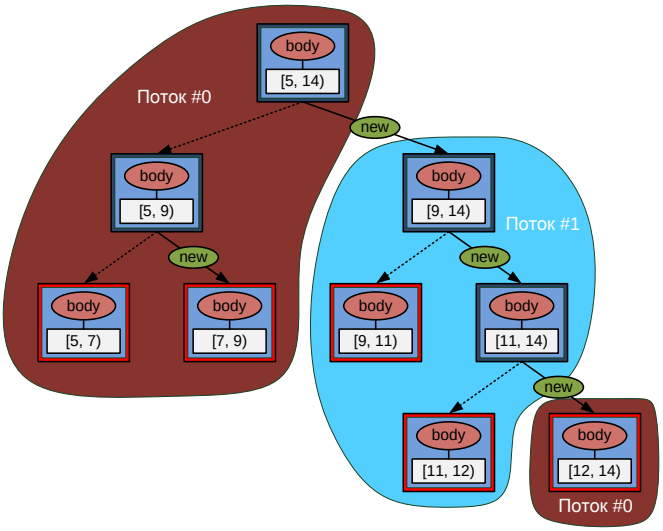

Рис. 3. Пример работы функции **`tbb::parallel_for`** при использовании двух потоков

Итак, при использовании стратегии **`simple_partitioner`** размер порции вычислений, которую каждый поток получит на обработку, не будет превосходить величины **`grainsize`**. Стратегию **`simple_partitioner`** есть смысл использовать в том случае, когда объём вычислений для каждого потока не должен превышать определённой величины.

В большинстве случаев будет эффективнее использовать стратегию автоматического выбора порции вычислений. Стратегия **`auto_partitioner`** будет выбирать размер порции вычислений автоматически, снижая накладные расходы на организацию параллелизма. При этом гарантируется, что размер порции вычислений, которую каждый поток получит на обработку, не будет меньше величины **`[grainsize/2]`**. Стратегия **`affinity_partitioner`** очень похожа на **`auto_partitioner`**, за исключением того, что выбор размера порции вычислений направлен на оптимальное использование кеш-памяти процессора.

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

```
parallel_for(blocked_range<int>(0, numOfRows, grainsize),
  VectorsMultiplicator(matrix, vector, resultVector, numOfColumns),
  simple_partitioner());
```

Для лучшего понимания функции **`tbb::parallel_for`** приведем практически аналогичный по устройству вариант распараллеливания задачи умножения матрицы на вектор на OpenMP:

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

Представленное сравнение явно выглядит не в пользу TBB, во всяком случае по затратам на разработку. Однако задача, которую мы взяли здесь в качестве примера, довольно проста. В более сложных ситуациях возможности TBB явным образом задавать алгоритм планирования вычислений, инкапсулировать связанные с вычислительным потоком код и данные способствуют упрощению разработки и программного кода приложения.

Компилятор Intel C++ Compiler 11.0 поддерживает лямбда-выражения стандарта C++0x. Это позволяет значительно сократить объем программного кода, не объявляя класс функтора. Ниже приведён программный код матрично-векторного умножения с использованием лямбда-выражений:

```
parallel_for(blocked_range<size_t>(0,numOfRows, grainsize),
  [=](const blocked_range<size_t>& r)
  {
    int begin = r.begin(),  end = r.end();

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

  },
  simple_partitioner()
);
```

Циклы с известным числом повторений с редукцией

Функция **tbb::parallel_reduce** предназначена для распараллеливания вычислений, представленных в виде цикла **for** с редукцией (типичным примером задачи, в параллельной реализации которой выгодно использовать редукцию, является скалярное умножение векторов).

Прототип функции **tbb::parallel_reduce**:

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

Нетрудно видеть, что прототип **tbb::parallel_reduce** полностью совпадает с прототипом функции **tbb::parallel_for**, однако методы функторов для этих функций различны. Функтор **body** для **tbb::parallel_reduce** должен содержать следующие методы:

- Конструктор расщепления **Body(body&, split).**
- Деструктор **~Body()**.
- Метод, выполняющий вычисления **void operator()(Range& range).** Аргументом 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.
- 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 **tbb::parallel_reduce** похож на **tbb::parallel_for**. В отличие от **tbb::parallel_for** 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. Алгоритм работы **tbb::parallel_reduce** с точки зрения распределения вычислений рассмотрим на примере:

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

Прежде всего отметим, что **tbb::parallel_reduce**, также как и **tbb::parallel_for** расщепляет итерационные пространства до тех пор, пока их размеры больше, чем **grainsize.** Однако отличие в работе функции **tbb::parallel_reduce** состоит в том, что она не создает копии исходного функтора при каждом расщеплении (кроме отдельного описанного ниже случая), а оперирует ссылками на функторы. Наконец, еще раз подчеркнем, что создание «порции» вычислений в виде итерационного пространства и связанного с ним функтора может выполняться и часто выполняется отдельно от дальнейшей обработки этого «порции».

Итак, первая схема работы **tbb::parallel_reduce** реализуется в том случае, если очередная «порция» вычислений обрабатывается на том же потоке, что и предыдущая. В данной схеме требуется и существует только один экземпляр функтора. Пусть, например, на очередной итерации потоком 0 была

создана «порция» вычислений размером [9, 14). Пусть этот же поток обрабатывает эту «порцию» (рис. 4). Так как размер итерационного пространства больше 2, то происходит его расщепление и копирование указателя на функтор. Расщепление итерационного пространства осуществляется так же, как и при использовании **`tbb::parallel_for`**. Данная схема реализуется всегда при вычислении в один поток.

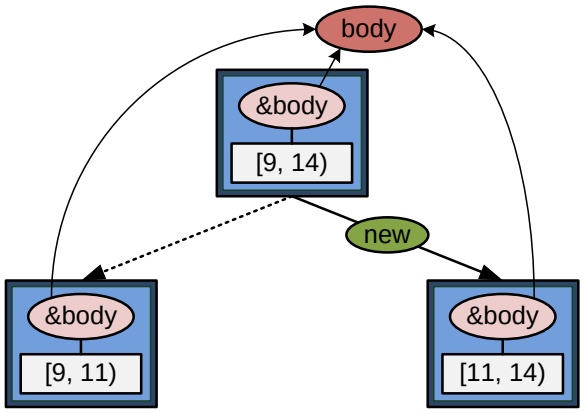

Рис. 4. Выполнение вычислений **`tbb::parallel_reduce`** на одном потоке

Если очередная «порция» вычислений выполняется на потоке отличном от потока-создателя «порции», то реализуется вторая схема путем создания нового функтора с помощью конструктора расщепления, как показано на рис. 5. Отметим, что в отличие от итерационных пространств, 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 разработчикам функтора реализовать более сложное поведение в момент «переноса» функтора на другой поток.

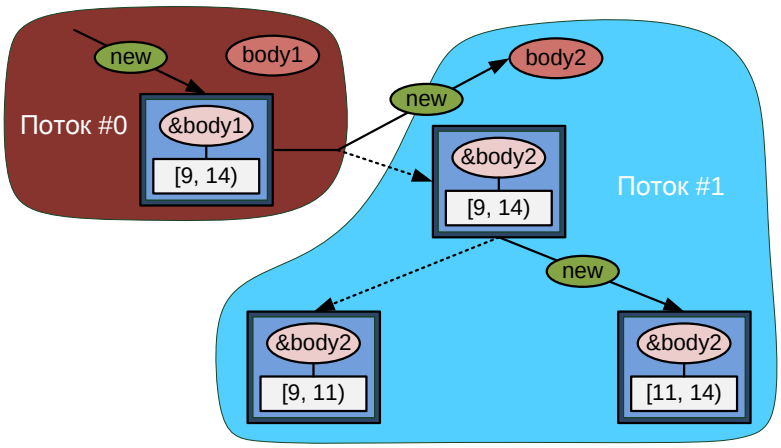

Рис. 5. Выполнение вычислений **`tbb::parallel_reduce`** при «смене» потока

Например, пусть на очередной итерации потоком 0 были созданы две «порции» вычислений [5, 9) и [9, 14). Далее поток 0 продолжил выполнять вычисления с «порцией» [5, 9) (на рис. 5 эта часть не показана), а поток 1 взял на выполнение «порцию» [9, 14). 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.

Мы рассмотрели порядок выполнения «прямого хода» вычислений в цикле с редукцией. Теперь рассмотрим само выполнение операции редукции. Если все вычисления происходили в один поток, то операция редукции не требуется, так как функтор существует в одном экземпляре и он один выполнил все вычисления. Если же в некоторый момент очередная «порция» вычислений была создана одним потоком, а сами вычисления производились другим, то был создан новый функтор, а это значит, что необходимо выполнить редукцию нового функтора на старый.

Рассмотрим алгоритм редукции в функции **`tbb::parallel_reduce`** с точки зрения выполнения вычислений на примере:

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

Как и для **tbb::parallel_for**, процесс вычислений является недетерминированным (какой поток выполнит конкретную часть вычислений, определяется только на этапе выполнения). Пусть реализовалась ситуация, представленная на рис. 6.

В тех узлах дерева, где происходило расщепление функтора, по завершении вычислений будут выполнены операции редукции, как показано на рис. 7.

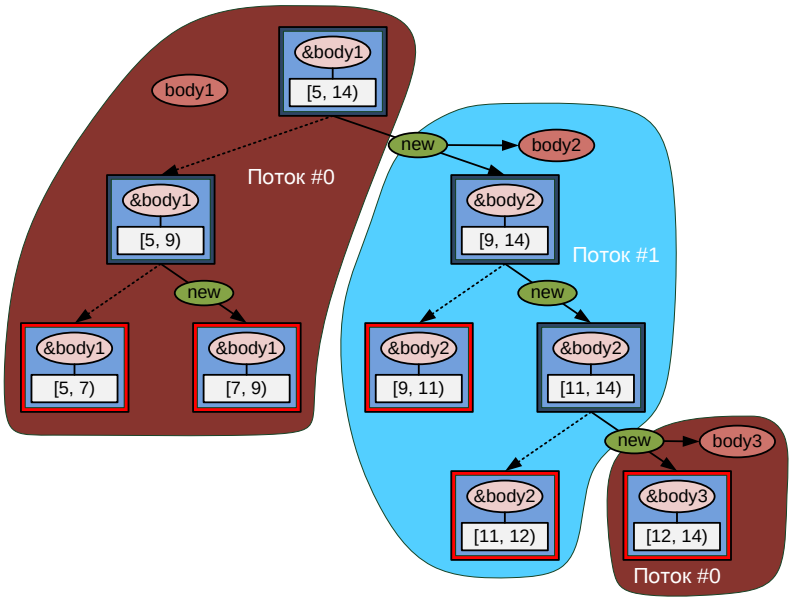

Рис. 6. Алгоритм работы **tbb::parallel_reduce**

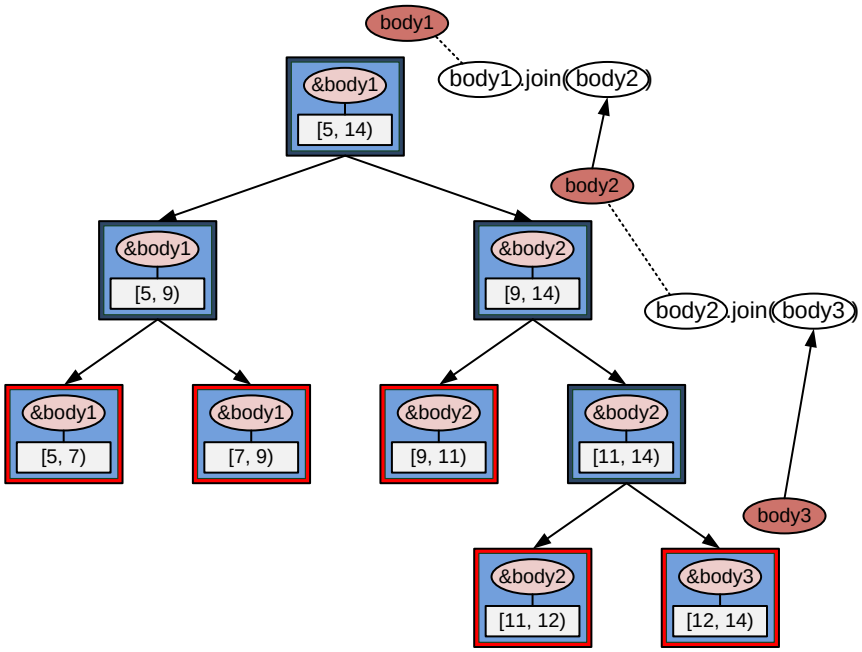

Рис. 7. Выполнение редукции при использовании **tbb::parallel_reduce**

Пример функтора функции **tbb::parallel_reduce** для решения задачи 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;
}

//Функтор
```

```
class ScalarMultiplicator
{
private:
  const double *a, *b;
  double c;

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

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

  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 ScalarMultiplicator& multiplicator)
  {
    c += multiplicator.c;
  }

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

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

```
  ScalarMultiplicator s(a, b);
  parallel_reduce(blocked_range<int>(0, size, grainsize), s);
```

Для лучшего понимания функции **tbb::parallel_reduce** приведем практически аналогичный по устройству вариант распараллеливания задачи скалярного умножения векторов на OpenMP:

```
#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

Библиотека TBB содержит достаточно много шаблонных функций и классов, позволяющих реализовать типовые параллельные алгоритмы. Мы рассмотрим следующие функции и классы:

tbb::parallel_sort;

tbb::parallel_do;

tbb::pipeline.

Сортировка

Библиотека TBB содержит шаблонную функцию **tbb::parallel_sort**, предназначенную для сортировки последовательности. С помощью этой функции можно выполнять параллельную сортировку встроенных типов языка C++ и всех классов, у которых реализованы методы **swap** и **operator().** Последний должен выполнять сравнение двух элементов. Более подробная информация об этой функции представлена в [2, 3] .

Пример использования функции **tbb::parallel_sort** :

```
#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>());
}
```

Циклы с условием

Библиотека TBB содержит шаблонную функцию **tbb::parallel_do**, с помощью которой можно выполнить параллельную обработку элементов, размещенных в некотором «входном» потоке данных. Элементы могут быть добавлены в поток данных во время вычислений. Прототип функции

```
template<typename InputIterator, typename Body>
void parallel_do( InputIterator first, InputIterator last, Body body);
```

Первые два параметра задают стартовый диапазон вычислений и являются итераторами. Последний параметр задаёт функтор, который будет выполнять обработку элементов. При этом **operator()** может принимать второй параметр **parallel_do_feeder<item_t>&** с помощью которого можно добавлять новые элементы на обработку. Вычисления закачиваются, если обработаны все элементы.

Пример использования функции **tbb::parallel_do**:

```
void ParallelDo( const std::vector<int>& root_set )
{
    tbb::parallel_do(root_set.begin(), root_set.end(),Body());
}
```

Конвейерные вычисления

Библиотека TBB содержит класс **tbb::pipeline,** с помощью которого можно выполнять конвейерные вычисления. Для таких вычислений характерно выполнение 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** 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 (тип **parallel**). Последовательные фильтры бывают двух типов:

serial_out_of_order – обрабатывают элементы в произвольном порядке;

serial_in_order – обрабатывают элементы в одном и том же порядке для всех фильтров такого типа.

Тип фильтра указывается в конструкторе при его создании. Для добавления фильтра в объект класса **tbb::pipeline** используется метод **add_filter**. Для запуска вычислений используется метод **run.**

Класс фильтра **tbb::filter** является абстрактным, должен быть унаследован всеми фильтрами, реализованными пользователем. Класс фильтра должен содержать реализацию метода обработки элементов:

```
virtual void* filter::operator()(void* item)
```

Метод обработки элемента должен вернуть указатель на элемент, который будет обрабатываться следующим фильтром. Первый фильтр, использующийся в объекте класса **tbb::pipeline** должен вернуть NULL, если больше нет элементов для обработки.

Фильтры можно создавать на основе функторов с помощью функции **tbb::make_filter.**

Приведенное описание классов **tbb::pipeline** и **tbb::filter** является обзорным. Более подробная информация о реализации конвейерных вычислений представлена в [2].

Ядро библиотеки

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

Общая характеристика логических задач

Логическая задача в библиотеке TBB 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.**

Класс **tbb::task** содержит виртуальный метод **task::execute,** в котором выполняются вычисления. Прототип этого метода:

```
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.

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

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

Алгоритм работы

Рассмотрим общую идею работы планировщика потоков библиотеки TBB. Каждый поток, созданный библиотекой, имеет свое множество (пул) готовых к выполнению задач. Это множество представляет собой динамический массив списков. Списки обрабатываются в порядке LIFO (last-in first-out). Задачи, расположенные на ***i***-м уровне (в списке ***i***-го элемента массива), порождают подзадачи уровня ***i*+1**. Поток выполняет задачи из самого нижнего непустого списка из массива списков. Если все списки потока пусты, то поток случайным образом «забирает» задачи, расположенные в наиболее низком списке у других потоков. Пример работы планировщика потоков библиотеки TBB при запуске в 2 потока представлен на рис. 8.

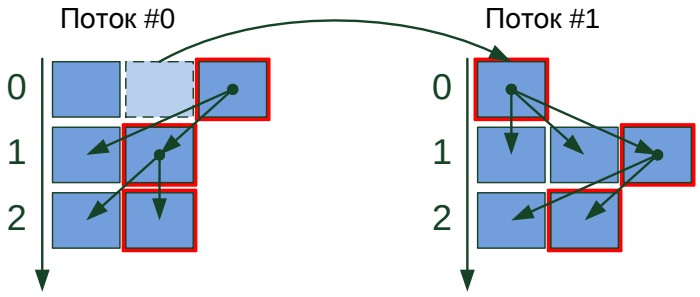

Рис. 8. Алгоритм работы планировщика потоков библиотеки TBB

Каждая задача имеет следующий набор связанных с ней атрибутов:

- **owner** – поток, которому принадлежит задача.
- **parent**[2] – 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.**

В дальнейшем каждую задачу будем характеризовать следующей тройкой: (**parent, depth, refcount**).

[2] Здесь и далее обозначения parent, depth используются и как имена полей и как имена методов, с помощью которых можно получить их значения.

После того как планировщик потоков назначает каждому потоку задачу на выполнение, происходит следующее:

1. Выполнение метода **task::execute** и ожидание его завершения.
2. 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.
3. Если для задачи был вызван один из методов **task::recycle_***, то задача повторно добавляется в пул готовых к выполнению (отличия между этими методами будут представлены ниже).

Создание и уничтожение логических задач

Создание задачи должно осуществляться только с помощью оператора **new**, перегруженного в библиотеке TBB. Виды оператора **new**:

- **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)**;
- **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).**

Пример создания задачи представлен ниже:

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

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

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

Планирование выполнения логических задач

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 TBB реализован метод **task::state**, который возвращает текущее состояние задачи, для которой он был вызван. Данная информация может оказаться полезной при отладке приложений. Прототип метода **task::state**:

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

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 **task::recycle_***;

- **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**.

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.

Ниже представлены основные методы управления планированием и синхронизации задач:

- **void task::set_ref_count(int count)** – устанавливает значение поля **refcount** равным **count.**
- **void task::wait_for_all()** – ожидает завершения всех подчиненных задач. Поле **refcount** должно быть равно числу подчиненных задач + 1.
- **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.**

  Пример использования рассмотренных методов:

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

Продолжаем перечисление методов:

- **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.
- **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.
- **static void task::spawn_root_and_wait(task& root)** – выполняет запуск задачи **root.** Память для задачи должна быть выделена с помощью **task::allocate_root().** Пример использования этого метода:

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

- **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**.

Библиотека предоставляет набор методов, которые позволяют повторно использовать задачи для вычислений, что способствует многократному использованию, выделенных ресурсов и уменьшению накладных расходов:

- **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**.
- **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_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 этого метода:

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

- **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.

В тех случаях, когда необходимо изменить алгоритм планирования может оказаться полезным изменение 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.

Ниже представлены 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** значения 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». Постановка задачи состоит в следующем. 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. Программный код решающий эту задачу обычным перебором с возвратом:

```
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

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
    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;
  }
};

spin_mutex Backtracker::myMutex;
int Backtracker::count;

void Solve(int size)
{
  concurrent_vector<int> placement;
  task_list tasks;

  Backtracker::count=0;
  for (int i = 0; i < size; i++)
  {
    Backtracker& bt =
      *new (task::allocate_root()) Backtracker(placement, i, size);
    tasks.push_back(bt);
  }
  task::spawn_root_and_wait(tasks);
}
```

Использованные в коде для синхронизации мьютексы описаны в следующем разделе.

Примитивы синхронизации

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.

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 (т.е. 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).

Возможны две реализации (с точностью до перестановки потоков) одновременного выполнения такого кода двумя потоками. Наиболее ожидаемое поведение приложения представлено на рис. 8. 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`**.

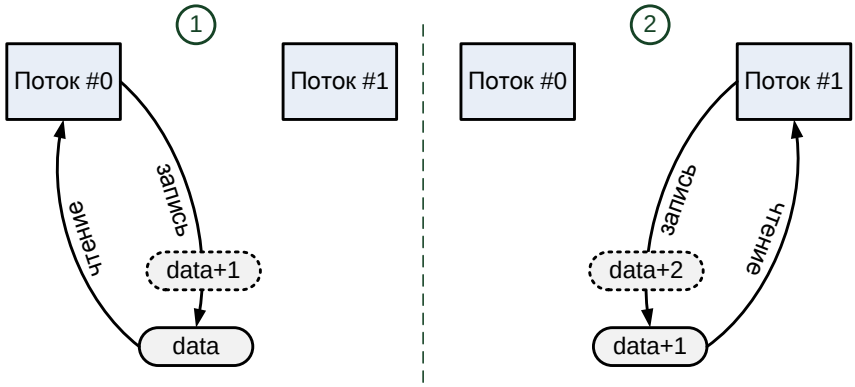

Рис. 9. Вариант реализации «гонки данных»

Другое возможное поведение представлено на рис. 9. Поток 0 выполняет чтение значения переменной в регистр и инкремент этого регистра, и в этот же момент времени поток 1 выполняет чтение переменной **`data`**. Так как для каждого потока имеется свой набор регистров, то поток 0 продолжит выполнение и запишет в переменную значение **`data+1`**. Поток 1 также выполнит инкремент регистра (значение переменной **`data`** было прочитано из оперативной памяти 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`**. Обе реализации могут наблюдаться как на многоядерной (многопроцессорной) системе, так и на однопроцессорной, одноядерной.

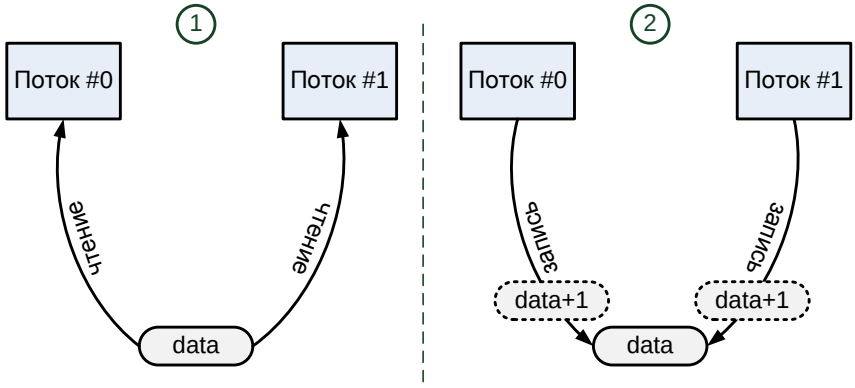

Рис. 10. Вариант реализации «гонки данных»

Таким образом, в зависимости от порядка выполнения команд результат работы приложения может меняться. Возникает необходимость в механизме синхронизации выполнения потоков, с помощью которого можно было бы обеспечить выполнение части кода не более, чем одним потоком в каждый момент времени.

Основным способом решения задачи синхронизации является использование примитивов синхронизации. Одним из таких примитивов является *мьютекс* (*mutex*), который реализован в библиотеке TBB.

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».

Библиотека TBB содержит реализацию трех типов мьютексов:

- **`mutex`** – мьютекс операционной системы. Представляет собой обертку над примитивами синхронизации операционной системы (в операционных системах семейства **Microsoft Windows** в качестве основы используются критические секции). Так как 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 состояние готового к выполнению. Таким образом, время прошедшее после освобождения мьютекса до того момента когда поток получит управление может оказаться достаточно большим;
- **`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 он делает это 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;
- **`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, так как его работа несет дополнительные накладные расходы.

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.

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;
- **`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.

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 SumMutex;

public:
  // Методы функтора
```

```
  //...
  void operator()(const blocked_range<int>& Range) const
  {
    mutex::scoped_lock lock;
    lock.acquire(myMutex);
    count++;
    lock.release();
  }
};
```

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 блокируются при попытке захвата мьютекса.

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.**

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;
- **M::scoped_lock::acquire(M&, bool write=true)** – метод захвата мьютекса. Переводит мьютекс из состояния «свободен» в состояние «захвачен». Второй параметр указывает 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 неблокирующего захвата мьютекса. Второй параметр указывает 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**;
- **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.

Многие функции и классы библиотеки TBB выполняют синхронизацию неявно (без использования мьютексов). Например, функция **parallel_for** «ждет» завершения всех потоков, занятых вычислениями прежде, чем вернуть управление потоку, с которого она была вызвана.

Помимо мьютексов библиотека TBB содержит шаблонный класс **tbb::atomic**, который также может использоваться для синхронизации. 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. 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++;
  }
};
```

Потокобезопасные контейнеры

Библиотека TBB содержит реализацию трех контейнеров, которые являются аналогами контейнеров STL:

- **tbb::concurrent_queue** – очередь;
- **tbb::concurrent_hash_map** – ассоциативный контейнер с поддержкой конкурентного доступа к элементам;
- **tbb::concurrent_unordered_map** – ассоциативный контейнер с поддержкой конкурентного выполнения операций вставки и обхода;
- **tbb::concurrent_priority_queue** – приоритетная очередь;
- **tbb::concurrent_unordered_set** – множество;
- **tbb::concurrent_vector** – вектор.

В качестве примера рассмотрим **tbb::concurrent_queue**. Т.к. контейнеры библиотеки TBB 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**, так как 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 методов, то они тоже учитываются.
- 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.

Более подробная информация о потокобезопасных контейнерах библиотеки TBB представлена в [3].

Литература

Использованные источники информации

1. Официальный сайт Intel® Threading Building Blocks. – [http://www.intel.com/software/products/tbb/]
2. Intel® Threading Building Blocks. Reference Manual. Revision 1.26. – Intel Corporation, 2011.
3. Intel® Threading Building Blocks. Tutorial. Revision 1.20. – Intel Corporation, 2011.

Рекомендуемая литература

4. Andrews, G. R. (2000). Foundations of Multithreaded, Parallel, and Distributed Programming. – Reading, MA: Addison-Wesley (русский перевод Эндрюс Г. Р. Основы многопоточного, параллельного и распределенного программирования. – М.: Издательский дом «Вильямс», 2003).
5. Quinn, M. J. (2004). Parallel Programming in C with MPI and OpenMP. – New York, NY: McGraw-Hill.
6. Barbara Chapman, Gabriele Jost, Ruud van der Pas (2007). Using OpenMP: Portable Shared Memory Parallel Programming (Scientific Computation and Engineering).
7. Майерс С. Эффективное использование C++. 35 новых способов улучшить стиль программирования. – СПб: Питер, 2006.
8. Майерс С. Эффективное использование C++. 55 верных способов улучшить структуру и код ваших программ. – М.: ДМК Пресс, 2006.
9. Павловская Т.А. C/C++. Программирование на языке высокого уровня. – СПб: Питер, 2003.
10. 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

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

Приложения

Заголовочные файлы библиотеки TBB

Для того чтобы воспользоваться классами и функциями библиотеки TBB необходимо подключать соответствующие заголовочные файлы. Например, для работы с функцией **`parallel_for`** необходимо подключить файл **`parallel_for.h`**, для работы с **`concurrent_hash_map`** – **`concurrent_hash_map.h`**. Таблица функций/классов и необходимых для них заголовочных файлов представлена ниже:

Название функции/класса	Заголовочный файл
aligned_space	aligned_space.h
atomic	atomic.h
blocked_range	blocked_range.h
blocked_range2d	blocked_range2d.h
blocked_range3d	blocked_range3d.h
cache_aligned_allocator	cache_aligned_allocator.h
concurrent_hash_map	concurrent_hash_map.h
concurrent_priority_queue	concurrent_priority_queue.h
concurrent_queue	concurrent_queue.h
concurrent_unordered_map	concurrent_unordered_map.h
concurrent_unordered_set	concurrent_unordered_set.h
concurrent_vector	concurrent_vector.h
mutex	mutex.h
parallel_do	parallel_do.h
parallel_for	parallel_for.h
parallel_for_each	parallel_for_each.h
parallel_invoke	parallel_invoke.h
parallel_reduce	parallel_reduce.h
parallel_scan	parallel_scan.h
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
recursive_mutex	recursive_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
tick_count	tick_count.h

Сборка и настройка проекта

Для сборки приложения, использующего библиотеку TBB**,** необходимо указать библиотеку, с которой будет линковаться приложение: **`tbb_debug.lib`** или **`tbb.lib`**. Библиотека **`tbb_debug.lib`** выполняет проверки корректности во время выполнения приложения и полностью поддерживается профилировщиком Intel® Thread Profiler. Библиотека **`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`**. При использовании в приложении операторов, аллокаторов выделения динамической памяти, необходимо указать библиотеку **`tbbmalloc.lib`** (или **`tbbmalloc_debug.lib`**). Для автоматической замены стандартных функций выделения/освобождения динамической памяти необходимо указать библиотеку **`tbbmalloc_proxy.lib`** (или **`tbbmalloc_proxy_debug.lib`**). Для подключения библиотеки в **Microsoft Visual Studio 2005** необходимо выполнить следующую последовательность действий:

1. 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\<arch>\vc8\lib** (где, **<arch>** должно быть **ia32** или **em64t** в зависимости от режима работы процессора). Нажмите **OK**.
2. В окне **Solution Explorer** нажмите правой кнопкой мыши на названии проекта и выберите пункт **Properties**.
3. Выберите пункт **Linker\Input** и в поле **Additional Dependencies** введите название библиотеки: **`tbb_debug.lib`** для **Debug** сборки или **`tbb.lib`** для **Release** сборки.

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 иметь динамическую библиотеку **`tbbmalloc.dll`** (при использовании **`tbbmalloc.lib`**) или **`tbbmalloc_debug.dll`** (при использовании **`tbbmalloc_debug.lib`**). При использовании автоматической замены стандартных функций выделения/освобождения динамической памяти необходимо иметь библиотеку **`tbbmalloc_proxy.dll`** (при использовании **`tbbmalloc_proxy.lib`**) или **`tbbmalloc_proxy_debug.dll`** (при использовании **`tbbmalloc_proxy_debug.lib`**).

Совместное использование с OpenMP

Библиотеку TBB можно использовать одновременно с OpenMP. Для этого на каждом потоке, созданном с помощью OpenMP (внутри параллельной секции), необходимо запустить планировщик потоков TBB.

```
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 TBB
    }
  }
}
```

Оценка эффективности приложений

Основным способом оценки эффективности приложения является измерение времени выполнения приложением вычислительно трудоемких операций. Библиотека TBB содержит класс **tbb::tick_count**, с помощью которого можно выполнять измерения времени. Этот класс реализован таким образом, что 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.**

Основной метод класса **tbb::tick_count** – метод **now**, измеряющий текущее значение времени. Этот метод является статическим и возвращает экземпляр класса **tbb::tick_count**. Его прототип представлен ниже:

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

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**. Его прототип представлен ниже:

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

Типичная схема измерения времени выполнения вычислений представлена ниже:

```
void SomeFunction()
{
  tick_count t0 = tick_count::now();
  // Вычисления
  tick_count t1 = tick_count::now();
  printf("Время вычислений = %f seconds\n", (t1 - t0).seconds());
}
```

Динамическое выделение памяти

Обычные операторы выделения динамической памяти работают с общей кучей для всех потоков, что требует наличия синхронизации. Библиотека TBB содержит масштабируемые операторы выделения динамической памяти (аллокаторы) на каждый поток. Прототипы функций выделения динамической памяти для C-программ представлены ниже:

```
void* scalable_calloc(size_t nobj, size_t size);
void  scalable_free(void* ptr);
void* scalable_malloc(size_t size);
void* scalable_realloc(void* ptr, size_t size);
```

Более подробная информация о динамическом выделении памяти представлена в [2].