Параллельное программирование для многоядерных процессоров

Материалы для семинарских занятий

Ю. Сердюк (Институт программных систем РАН, г.Переславль-Залесский)
А. Петров (Рыбинская государственная авиационно-технологическая академия)

Май, 2009

Содержание

Семинарское занятие №1

1 Наиболее распространенные причины низкой производительности параллельных программ

Параллельное программирование представляет собой написание программ, которые могут исполняться на нескольких ядрах/процессорах, что, в общем случае, должно вести к сокращению времени решения задач. Однако, на практике, в частности, при использовании библиотеки PFX, возможны ситуации когда с увеличением количества используемых процессоров не удается добиться такого уменьшения − в этом случае говорят о низкой производительности параллельной программы.

Причины таких ситуаций могут быть различны:

а) задача, как таковая, плохо поддается распараллеливанию,

б) неправильное использование библиотеки PFX для распараллеливания приложения,

и др.

Существует ряд общих ситуаций, которые приводят к низкой эффективности параллельных программ, в частности, использующих библиотеку PFX. Ниже приведено краткое описание некоторых из таких ситуаций.

Достаточный объем вычислительной работы

Одним из основных требований, которым должна удовлетворять программа для того, чтобы была возможность её эффективного распараллеливания, является наличие достаточного объема работы, который она должна выполнить. Допустим, что только половина всех вычислений в программе может быть выполнена параллельно, тогда согласно закону Амдала [1] множитель производительности будет не более двух. Данное правило вполне очевидно, если представить себе, что программа 90% времени исполнения находится в ожидании ответа на SQL-запрос – параллельное исполнение оставшихся 10% не принесет существенного эффекта. Вместе с тем стоит отметить, что в этом случае мы можем параллельно посылать SQL-запросы разным серверам и использовать асинхронное взаимодействие с серверами.

Одним словом, для того чтобы получить эффект от параллельного выполнения задач, нужно иметь достаточный объем работы для распараллеливания, как минимум перекрывающий издержки самого процесса распараллеливания.

Гранулярность задач (размер отдельных задач и их общее количество)

Параллельная программа, решающая общую задачу, обычно состоит из отдельных фрагментов – подзадач, которые определяет в коде сам программист. Если количество

подзадач будет слишком велико, то большинство из них будут простаивать в очереди к ядрам процессора, а объем издержек на запуск подзадач будет расти. Если количество подзадач будет слишком малым, то некоторые ядра процессора будут простаивать, снижая общую производительность системы.

Некоторые компоненты Parallel Extensions API, такие как, например, Parallel.For и PLINQ, способны сами определить подходящий для данной системы объем и количество параллельно исполняющихся задач, тогда как при работе с такими компонентами как TPL и Futures ответственность за принятие правильного решения лежит на самом программисте.

Балансировка нагрузки

Даже в том случае, когда гранулярность задач подобрана верно, возникает проблема распределения этих задач по ядрам. Эта проблема связана с тем, что потенциально разные задачи могут исполняться разное (зачастую неопределенное) время и, следовательно, в процессе исполнения программы без балансировки нагрузки ядер возможны простои некоторых ядер и простои задач в очередях к другим ядрам.

Так, например, если Parallel.For будет предполагать, что все итерации распараллеливаемого цикла исполняются одинаковое время, то мы можем просто разбить пространство итераций на блоки по числу доступных ядер и параллельно исполнить эти блоки. Однако в действительности длительность каждой итерации может быть различной, что ведет к снижению производительности в отсутствии балансировки. В действительности, компонент Parallel.For в PFX реализует автоматическую балансировку, которая во многих случаях решает эту проблему.

Выделение памяти и сборка мусора

Некоторые программы характеризуются интенсивным взаимодействием с памятью, что вызывает значительные временные затраты как на выделение памяти, так и на сборку мусора. К примеру, программа, которая обрабатывает текст, будет интенсивно использовать память, в особенности, если программист не обратил должного внимания на косвенные выделения памяти.

К сожалению, выделение памяти на современных вычислительных архитектурах это операция, которая может требовать синхронизации между вычислительными потоками, в которых выполняется эта операция. Как минимум мы должны быть уверены, что области памяти, выделяемые параллельно, не перекрываются.

Более серьезные временные потери возникают при сборке мусора. Если сборщик мусора находится в постоянной активности при исполнении программы, то это может стать узким местом при её распараллеливании. Разумеется, сборщик мусора в .NET может выполнять свою работу параллельно с остальными задачами программы, но для этого могут потребоваться дополнительные усилия со стороны программиста (подробности см. в [2] и [3]).

Кэш-промахи

Данная ситуация связана с принципами кэширования в современных компьютерах. Когда процессор производит выборку значения из основной памяти, копия этого значения попадает в кэш, что ускоряет доступ к этому значению, если, конечно, оно понадобится еще раз в последующих вычислениях. На самом деле, процессор кэширует не только одно выбранное значение, но и некоторую окрестность этого значения в памяти. Значения, перемещаемые из основной памяти в кэш и сохраняемые в кэш построчно, называются строками кэша, и типичный размер составляет 64 или 128 байт.

Одной из проблем, связанных с кэшированием на многоядерных компьютерах, является ситуация при которой одно из ядер записывает значение в определенную область памяти, находящуюся в кэше другого ядра. В этом случае другое ядро при обращении к соответствующей строке кэша получит ситуацию промаха и будет вынуждено обновить эту строку из основной памяти. С некоторой вероятностью может сложиться ситуация при которой одно ядро постоянно пишет в основную память, нарушая тем самым целостность кэша второго ядра, которое будет вынуждено постоянно обновлять свой кэш, что приведет к резкому падению производительности.

Более тонкая проблема возникает, когда ядра записывают значения в разные области памяти, которые, тем не менее, расположены так близко, что находятся в одной строке кэша. Несмотря на кажущуюся малую вероятность такой ситуации, она достаточно часто встречается на практике, поскольку, например, поля объекта или промежуточные результаты обработки массивов объектов зачастую находятся в памяти относительно близко.

Есть несколько способов избежать подобных проблем. Можно, например, проектировать структуры данных специальным образом, снижая вероятность появления проблем кэширования, или выделять память в разных потоках.

Кроме того, проектируя процессы параллельной обработки необходимо уже на алгоритмическом уровне сохранять локальность обрабатываемых данных относительно потока. Так если, например, стоит задача обработать массив чисел, то очевидно, что локальность обработки каждым ядром своей половины массива будет гораздо выше, чем локальность обработки одним ядром элементов массива с четными индексами, а вторым – с нечетными. В последнем случае только половина элементов строки кэша ядра будет содержать нужные данному ядру элементы.

[1] Закон Амдала http://ru.wikipedia.org/wiki/Закон_Амдала
[2] Server, Workstation and Concurrent GC
http://blogs.msdn.com/clyon/archive/2004/09/08/226981.aspx
[3] GC в .NET http://www.rsdn.ru/article/dotnet/GC.xml#ENE

2. Как начать программировать с использованием Parallel Extensions

В состав данных лекций включены исходные коды нескольких примеров (см. приложения в Лекциях 10, 11, 12).

Задачи:

- изучите состав прилагаемых проектов, использующиеся в них сборки;
- попробуйте скомпилировать и запустить эти примеры на своем компьютере;
- поэкспериментируйте с входными параметрами и изучите эффективность данных параллельных приложений путем замера времени их исполнения при запуске на разном числе ядер/процессоров.

Семинарское занятие № 2

1 Вариант Parallel.For с локальными состояниями

Конструкции Parallel.For/ForEach имеют несколько перегруженных (overloaded) вариантов, с помощью которых можно организовать передачу информации между итерациями цикла, исполняющимися в одном потоке. Вспомним, что при выполнении Parallel.For/ForEach каждый рабочий поток выполняет, в общем случае, несколько итераций. Например, при общем количестве итераций 100, на 4-хядерной машине, обычно будет запущено 4 потока, каждый из которых выполнит 25 итераций. Обычно, передача информации (например, подсчет какого-либо значения) между итерациями, исполняющимися в одном потоке, нужна для вычисления некоторого сводного значения по всем итерациям цикла.

Одним из вариантов перегруженной конструкции Parallel.For для поддержки вычислений такого рода выглядит так:

```
public static void For<TLocal>(
    int fromInclusive, int toExclusive,
    Func<TLocal> threadLocalInit,
    Action<int, ParallelState<TLocal>> body,
    Action<TLocal> threadLocalFinally);
```

Здесь тип TLocal задает тип объекта, с которым будут работать итерации в рамках одного потока. Делегат threadLocalInit вызывается один раз для каждого потока, выделенного для исполнения группы итераций цикла, перед тем как этот поток начал исполнять свои итерации. Обычно с помощью этого делегата создается или инициализируется объект типа TLocal с котрыми будут работать итерации.

Результат каждой итерации может быть записан в объект класса ParallelState<TLocal> который является одним из встроенных классов библиотеки PFX), передаваемый в качестве параметра в делегат body. Делегат body может обновлять специальное поле (а точнее, свойство) ThreadLocalState, которое имеет объект ParallelState<TLocal>, и которое имеет тип TLocal, и значение этого поля будет доступно следующим итерациям, исполняющимся в данном потоке. После того, как поток завершил исполнение своих итераций, в рамках его же, будет вызван делегат threadLocalFinally, которому в качестве параметра будет передано значение ThreadLocalState.

Примером использования одного из перегруженных вариантов конструкции Parallel.For может служить следующий код:

```
Parallel.For(0, N, () => new NonThreadSafeData(), (i,loop)=>
{
    UseData(loop.ThreadLocalState);
});
```

(Здесь не использован делегат threadLocalFinally из вышеприведенной сигнатуры).

В данном примере для каждого нового потока в рамках Parallel.For будет создан свой объект класса NonThreadSafeData (т.е., в данном случае TLocal = NonThreadSafeData). Таким образом, объект NonThreadSafeData будет передаваться между итерациями одного потока, что позволит потоку безопасно использовать этот объект. При этом, таких объектов

будет создано ровно столько, сколько потоков будет запущено. Отметим также, что локальная переменная loop в приведенном выше примере имеет тип ParallelState<TLocal>.

Полный пример вычисления некоторого сводного значения по совокупности данных с использованием объектов-состояний приведен ниже. Такое использование локальных по отношению к потокам объектов позволяет избежать применения блокировок при доступе к общему объекту на каждой итерации, вычислить промежуточные значения и затем объединить их с помощью делегата threadLocalFinally, используя блокирующую конструкцию:

```
int total = 0;
Parallel.ForEach(data, ()=>0, (elem,i,loop)=>
{
    loop.ThreadLocalState += Process(elem);
},
partial => Interlocked.Add(ref total, partial));
```

Отметим, что в данном случае локальная переменная partial имеет тип TLocal, который, в данном примере, есть тип int (см. делегат threadLocalInit).

Если требуется передавать между итерациями несколько общих переменных, то, как обычно делают в таких случаях, можно создать собственный класс из этих переменных, и уже его передавать как ThreadLocalState:

```
class MultipleValues { public int Total, Count; }

int total = 0, count = 0;
Parallel.ForEach(data,
    ()=>new MultipleValues { Total=0, Count=0 },
    (elem,i,loop)=>
{
    loop.ThreadLocalState.Total += Process(elem);
    loop.ThreadLocalState.Count++;
},
partial => {
    Interlocked.Add(ref total, partial.Total);
    Interlocked.Add(ref count, partial.Count);
});
```

Задания:

Реализуйте подсчет суммы элементов массива с использованием предлагаемого подхода, кроме того, подсчитайте, сколько потоков было запущено при исполнении Parallel.For/ForEach в вашей программе.

Изучите понятие "анонимный класс" языка C# и выясните можно ли использовать анонимные классы вместо явного определения класса MultipleValues в приведенном выше примере,и если нельзя, то почему.

Семинарское занятие № 3

1 Пользовательские расширения Parallel.For

Конструкция Parallel.For, реализованная в PFX, является, по сути, параллельным аналогом последовательного цикла for перебора в некотором диапазоне чисел. Однако, конструкция цикла for в языке C# не ограничена только перебором чисел из определенного диапазона - она также поддерживает произвольные выражения для инициализации цикла, условия окончания и обновления индекса:

```
for(var i = init(); cond(i); i = upd(i)) { Use(i); }
```

В PFX, однако, отсутствуют аналоги подобного for-цикла. Вместе с тем основной идеей, заложенной в PFX, было создание базовых конструкций, которые могли бы расширяться пользователями под их собственные задачи. Пример с циклом for позволяет продемонстрировать это. Рассмотрим следующий код:

```
public static void ParallelFor<T>(
    Func<T> init, Func<T,bool> cond, Func<T,T> upd, Action<T> body)
{
    Parallel.ForEach(Iterate(init, cond, upd), body);
}

private static IEnumerable<T> Iterate<T>(
    Func<T> init, Func<T,bool> cond, Func<T,T> upd)
{
    for(var i=init(); cond(i); i = upd(i)) yield return i;
}
```

Можно видеть, что здесь создана некоторая версия конструкции Parallel.For, которая поддерживает произвольные выражения для инициализации цикла, условия окончания и обновления индекса. Другими словами, здесь представлена параллельная версия C#-итератора.

Приведем пример использования новой конструкции. Допустим нам дан список узлов, который мы хотим обработать. Последовательная версия обработки может выглядеть следующим образом:

```
for(Node n = list; n != null; n = n.Next)
{
    Process(n);
}
```

Параллельная версия на основе введенной нами выше конструкции выглядит так:

```
ParallelFor(() => list, n => n != null, n => n.Next, n =>
{
    Process(n);
});
```

2 Изучение и анализ примеров

В Лекциях 10 и 12 приведены описания и приложены исходные коды параллельных программ быстрого преобразования Фурье (БПФ) и теста RandomAccess. Рассмотрите применение в них конструкций Paralell.For, а также возможность их замены на более высокоуровневые аналоги по типу тех, которые описаны в разделе 1 данного семинарского занятия. Кроме того, посредством запуска этих примеров на различном количестве процессоров, оцените эффективность данных реализаций БПФ и RandomAccess и выявите возможные проблемы с производительностью этих приложений.

Семинарское занятие № 4

1 Рекурсия и параллелизм (часть 1)

При изучении применения рекурсии в программировании, одним из наиболее типичных примеров является обработка структур данных, представленных в виде дерева. При этом, при обработке деревьев используются различные способы прохождения по их вершинам. Однако, при попытках параллельной (многопоточной) обработки деревьев возникают проблемы, которые отсутствуют при обычной, последовательной обработке.

Рассмотрим простую структуру данных – бинарное дерево (Tree):

Пример 1.

```
class Tree<T>
{
   public Tree<T> Left, Right; // потомки
   public T Data; // данные для этого узла
}
```

Предположим, что нам необходимо обработать каждую вершину дерева с помощью действия Action<T>, не заботясь о порядке в котором обрабатываются вершины. В последовательном, рекурсивном варианте это сделать очень легко:

Пример 2.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Обработать текущую вершину, затем левое поддерево,
    // а потом правое

    action(tree.Data);
    Process(tree.Left, action);
    Process(tree.Right, action);
}
```

Задача 1.

Реализуйте класс Tree, представляющий дерево с произвольной степенью ветвления (т.е., не только со степенью 2). Переделайте соответственно процедуру Process.

Нерекурсивный вариант с явным использованием стека (объекта класса Stack) может выглядеть следующим образом:

Пример 3.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
```

```
    var toExplore = new Stack<Tree<T>>();

    // Начало обработки с корневой вершины
    toExplore.Push(tree);
    while (toExplore.Count > 0)
    {
        // Извлечь очередную вершину, обработать ее, поместить в
        // стек ее потомков

        var current = toExplore.Pop();
        action(current.Data);
        if (current.Left != null)
            toExplore.Push(current.Left);
        if (current.Right != null)
            toExplore.Push(current.Right);
    }
}
```

Задача 2.

Переписать приведенный выше метод Process с использованием класса Queue вместо Stack.

Для перехода к параллельному варианту обработки вершин дерева, предположим, что само действие является независимым, вычислительно сложным (т.е., параллельная обработка вершин дерева имеет смысл) и безопасным для использования в многопоточной среде.

С использованием средств .NET Framework, имеются различные возможности для реализации такого параллельного варианта. Ниже приведена реализация, следующая оригинальному рекурсивному алгоритму и использующая класс ThreadPool:

Пример 4.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Событие mre используется, чтобы реализовать возврат из
    // данного метода только после того, как будет закончена
    // обработка вершин-потомков

    using (var mre = new ManualResetEvent(false))
    {
        // Обработать левого потомка асинхронно

        ThreadPool.QueueUserWorkItem(delegate
        {
            Process(tree.Left, action);
            mre.Set();
        });

        // Обработать текущую вершину и правого потомка синхронно

        action(tree.Data);
```

```
            Process(tree.Right, action);

            // Ожидание окончания обработки левого потомка

            mre.WaitOne();
        }
    }
```

Задача 3.

Показать, как аналогичный параллельный вариант можно реализовать, используя класс Thread.

Недостаток предыдущего варианта заключался в том, что обработка правого потомка задерживалась до тех пор, пока не будут обработаны данные текущей вершины. Чтобы повысить степень параллелизма метода Process, можно его модифицировать следующим образом:

Пример 5.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Необходимо воспользоваться конструкцией события
    // для ожидания завершения обработки потомков

    using (var mre = new ManualResetEvent(false))
    {
        int count = 2;

        // Обработать левого потомка асинхронно

        ThreadPool.QueueUserWorkItem(delegate
        {
            Process(tree.Left, action);
            if (Interlocked.Decrement(ref count) == 0)
                mre.Set();
        });

        // Обработать правого потомка асинхронно

        ThreadPool.QueueUserWorkItem(delegate
        {
            Process(tree.Right, action);
            if (Interlocked.Decrement(ref count) == 0)
                mre.Set();
        });

        // Обработать текущую вершину синхронно

        action(tree.Data);

        // Ожидание завершения обработки потомков

        mre.WaitOne();
    }
```

```
}
```

Задача 4.

Переписать приведенный выше метод Process в предположении, что класс Tree определен так, как в Задаче 1. Подготовьте несколько объектов класса Tree с количеством вершин, соответственно, 100, 200, 500, 1000, и протестируйте метод Process на данных деревьях.

Семинарское занятие № 5

Рекурсия и параллелизм (часть 2)

На предыдущем семинарском занятии было рассмотрено несколько методов рекурсивной параллельной обработки (бинарных) деревьев. Хотя последний из приведенных методов обладает достаточной степенью параллелизма, он имеет ряд серьезных недостатков:

1) вызов метода Process в рамках потока из ThreadPool блокирует этот поток до тех пор, пока не будут обработаны вершины-потомки для данной вершины-родителя; таким образом, данная реализация требует одного потока из ThreadPool на одну вершину обрабатываемого дерева; однако, класс ThreadPool имеет ограниченное количество потоков, а именно 25 потоков на ядро (процессор) для .NET 1.x/2.0, и 250 потоков на процессор для .NET 2.0 SP1; таким образом, если из пула выбраны все потоки, то новые потоки не могут быть созданы, а потому приложение может перейти в состояние дедлока;

2) создание нового потока в ThreadPool, когда количество уже созданных потоков больше или равно количеству доступных ядер (процессоров), занимает около 500 мсек; а потому обработка дерева, например, из 250 вершин займет более 2-х минут, потраченных только на создание потоков;

3) для каждого вновь создаваемого потока в .NET отводится около 1 Мб (виртуальной) памяти;

4) для обработки каждой вершины дерева создается один объект класса ManualResetEvent, операции над которым Set и WaitOne выполняются ядром операционной системы, а потому являются дорогостоящими в смысле используемых ресурсов.

Чтобы избавиться от некоторых из этих недостатков, ниже показана реализация, которая основана на последовательном проходе по дереву и записи в очередь на обработку в пул потоков задания на обработку каждой вершины в дереве.

Пример 6.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Использование события для ожидания завершения
    // обработки всех вершин дерева

    using (var mre = new ManualResetEvent(false))
    {
        int count = 1;
```

```
            // Рекурсивный делегат для прохода по дереву

            Action<Tree<T>> processNode = null;
            processNode = node =>
            {
                if (node == null) return;

                // Асинхронный запуск обработки текущей вершины

                Interlocked.Increment(ref count);
                ThreadPool.QueueUserWorkItem(delegate
                {
                    action(node.Data);
                    if (Interlocked.Decrement(ref count) == 0)
                        mre.Set();
                });

                // Обработка потомков

                processNode(node.Left);
                processNode(node.Right);
            };

            // Запуск обработки, начиная с корневой вершины

            processNode(tree);

            // Сигнал о том, что заданий на обработку больше
            // создаваться не будет

            if (Interlocked.Decrement(ref count) == 0) mre.Set();

            // Ожидание завершения обработки всех вершин

            mre.WaitOne();
        }
    }
```

Задача 5.

1. Объясните почему данная реализация свободна от дедлоков в отличие от предыдущей реализации.

2. Объясните возможен ли вариант данной реализации, в котором исходным значением счетчика является 0, т.е.,

int count = 0;

а в методе Process последний оператор

if (Interlocked.Decrement (ref count) == 0) mre.Set();

опущен.

Рекурсивный подход, приведенный в предыдущем примере, можно реализовать итеративно:

Пример 7.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Использование события для ожидания завершения
    // обработки всех вершин дерева

    using (var mre = new ManualResetEvent(false))
    {
        int count = 1;

        // Запуск обработки, начиная с корневой вершины

        var toExplore = new Stack<Tree<T>>();
        toExplore.Push(tree);

        // Обработка всех вершин

        while (toExplore.Count > 0)
        {
            // Извлечь текущую вершину и поместить в стек
            // её потомков

            var current = toExplore.Pop();
            if (current.Left != null)
                toExplore.Push(current.Left);
            if (current.Right != null)
                toExplore.Push(current.Right);

            // Асинхронная обработка данных

            Interlocked.Increment(ref count);
            ThreadPool.QueueUserWorkItem(delegate
            {
                action(current.Data);
                if (Interlocked.Decrement(ref count) == 0)
                    mre.Set();
            });
        }

        // Сигнал о том, что больше заданий на обработку
        // создаваться не будет

        if (Interlocked.Decrement(ref count) == 0) mre.Set();

        // Ожидание завершения обработки всех вершин

        mre.WaitOne();
    }
}
```

Недостаток предыдущего варианта состоит в том, что для каждой вершины дерева в ThreadPool помещается одно задание для обработки этой вершины. Гораздо экономичнее, а потому, эффективнее будет создать только N заданий на обработку (где N равно числу

доступных процессоров на машине), и где каждое задание будет состоять в обработке примерно 1/N-ой части всех вершин дерева. Такой подход можно реализовать, например, сохранив все вершины дерева в виде списка, и разделив его на N частей, для обработки каждой части в виде параллельного задания:

Пример 8.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Создать список всех вершин дерева

    var nodes = new List<Tree<T>>();
    var toExplore = new Stack<Tree<T>>();
    toExplore.Push(tree);
    while (toExplore.Count > 0)
    {
        var current = toExplore.Pop();
        nodes.Add(current);
        if (current.Left != null)
            toExplore.Push(current.Left);
        if (current.Right != null)
            toExplore.Push(current.Right);
    }

    // Разбиение списка на части

    int workItems = Environment.ProcessorCount;
    int chunkSize = Math.Max( nodes.Count / workItems, 1);
    int count = workItems;

    // Использование события для ожидания завершения
    // обработки всех заданий

    using (var mre = new ManualResetEvent(false))
    {
        // В каждом задании обрабатывается примерно 1/N-я
        // часть вершин

        WaitCallback callback = state =>
        {
            int iteration = (int)state;
            int from = chunkSize * iteration;
            int to = iteration == workItems - 1 ?
                nodes.Count : chunkSize * (iteration + 1);

            while (from < to) action(nodes[from++].Data);

            if (Interlocked.Decrement(ref count) == 0)
                mre.Set();
        };

        // ThreadPool используется для обработки N - 1 части;
        // для обработки последней части используется
```

```
            // текущий поток

            for (int i = 0; i<workItems; i++)
            {
                if (i < workItems-1)
                    ThreadPool.QueueUserWorkItem(callback, i);
                else
                    callback(i);
            }
            // Ожидание завершения обработки всех заданий

            mre.WaitOne();

        }
    }
```

Задача 6.

Проверьте, какой величины получатся задания для каждого процессора, если обрабатываемое дерево имеет 63 вершины, а количество процессоров равно 8.

Придумайте и реализуйте более оптимальный способ распределения вершин по заданиям, такой, чтобы размер любой пары заданий отличался не более, чем на 1.

Статическое разбиение списка на части также может быть не оптимальным способом деления работы между потоками. Альтернативным вариантом является динамическое разбиение, когда потоки конкурируют между собой за получение задания на обработку очередной вершины дерева после окончания обработки предыдущей вершины. Подход на основе динамического разбиения можно реализовать следующим способом:

Пример 9.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;

    // Получение конструкции перечисления для дерева

    IEnumerator<T> enumerator =
        GetNodes(tree).GetEnumerator();
    int workItems = Environment.ProcessorCount;
    int count = workItems;

    // Использование события для ожидания завершения
    // работы всех потоков

    using (var mre = new ManualResetEvent(false))
    {
        // Каждый поток будет получать данные для обработки,
        // используя механизм перечисления до тех пор,
        // пока не будут исчерпаны все вершины дерева

        WaitCallback callback = delegate
        {
            while (true)
            {
                T data;
```

```
                    lock (enumerator)
                    {
                        if (!enumerator.MoveNext()) break;
                        data = enumerator.Current;
                    }
                    action(data);
                }
                if (Interlocked.Decrement(ref count) == 0)
                    mre.Set();
            };

            // Из ThreadPool'а берется всего N - 1 потоков;
            // кроме того, для обработки используется
            // текущий поток

            for (int i = 0; i < workItems; i++)
            {
                if (i < workItems-1)
                    ThreadPool.QueueUserWorkItem(callback, i);
                else
                    callback(i);
            }

            // Ожидание завершения работы всех потоков

            mre.WaitOne();
        }
    }

    // Конструкция перечисления вершин дерева

    public static IEnumerable<T> GetNodes<T>(Tree<T> tree)
    {
        if (tree != null)
        {
            yield return tree.Data;
            foreach (var data in GetNodes(tree.Left))
                yield return data;
            foreach (var data in GetNodes(tree.Right))
                yield return data;
        }
    }
```

Недостаток последнего варианта состоит в том, что если обработка каждой вершины состоит из небольшого количества вычислительных операций, то общая производительность приложения будет низка из-за частого использования конструкции запирания lock. Поэтому, в некоторых случаях более оптимальным может оказаться вариант при котором при каждом применении lock к enumerator потоком извлекается сразу несколько вершин для обработки.

Задача 7.

Пусть для предыдущего примера дополнительным аргументом метода Process является целочисленный параметр $p \geq 1$, задающий количество вершин, извлекаемых из

enumerator'a, при каждом применении к нему оператора lock. Реализуйте метод Process с указанным дополнением.

Возможны и некоторые другие способы распределения вершин (бинарного) дерева между потоками для обработки.

Задача 8.

Пусть P есть количество доступных ядер/процессоров на машине. Предположим, что $P = 2^n$ для некоторого $n \geq 0$. Напишите программу обработки всех вершин бинарного дерева, в которой главный поток сам обрабатывает все вершины дерева, начиная с корня, и до глубины $d = \log_2 P$, и порождает дополнительные потоки для обработки соответствующих поддеревьев на этой глубине.

Пример с обходом вершин дерева был достаточно прост в том смысле, что не требовалось блокировать обработку родительской вершины до тех пор, пока не будут обработаны её вершины-потомки, и, наоборот, обработку вершин-потомков можно было начинать, не дожидаясь окончания обработки родительской вершины.

Однако, для многих рекурсивных задач такая автономная схема обработки невозможна. Рассмотрим, для примера, типичный рекурсивный алгоритм сортировки, а именно, алгоритм быстрой сортировки (дополнительную информацию об этом алгоритме можно найти на странице http://en.wikipedia.org/wiki/Quicksort):

Пример 10.

```
public static void Quicksort<T>(T[] arr, int left, int right)
      where T : IComparable <T>
{
      if (right > left)
      {
       int pivot = Partition ( arr, left, right );
       Quicksort ( arr, left, pivot - 1 );
       Quicksort ( arr, pivot + 1, right );
      }
}
```

Из этого примера видно, что стартовать асинхронное выполнение вызовов Quicksort можно только после завершения шага Partition.

Задача 9.

Реализуйте параллельный вариант рекурсивного алгоритма Quicksort с помощью класса ThreadPool. Оцените количество порождаемых потоков при сортировке массивов большой длины. Придумайте способ увеличения эффективности данного алгоритма и реализуйте его.

Противоположная проблема, по сравнению с алгоритмом Quicksort, имеет место в рекурсивном варианте программы сортировки слиянием:

Пример 11.

```
public static void Mergesort<T>(T[] arr, int left, int right)
      where T : IComparable <T>
{
      if (right > left)
      {
       int mid =  ( right + left ) / 2;
       Mergesort ( arr, left, mid );
       Mergesort ( arr, mid + 1, right );
       Merge ( arr, left, mid + 1, right );
      }
}
```

В этом варианте, мы получили алгоритм, аналогичный рассматривавшимся выше в примерах 3 и 4. А именно - поток, выполняющий функцию Mergesort, блокируется перед выполнением шага Merge до тех пор, пока не будут (асинхронно) выполнены рекурсивные вызовы Mergesort. Другими словами, такой вид обработки очень трудно эффективно реализовать средствами класса ThreadPool.

Семинарское занятие № 6

1 Асинхронная модель программирования и класс Future<T>

В данном занятии будет показано как

1) можно реализовать "асинхронную модель программирования" (АМП) с использованием объектов класса Future<T>, и, наоборот, как
2) можно создавать объекты класса Future<T> с использованием имеющейся реализации АМП.

Таким образом, будет показано как Task Parallel Library интегрирована с асинхронными механизмами, уже имеющимися в .NET Framework.

Базовыми конструкциями, лежащими в основе АМП, являются

(1) метод BeginXx, с помощью которого запускается асинхронная операция, и который возвращает объект типа IAsyncResult, и
(2) метод EndXx, который принимает в качестве входного аргумента IAsyncResult и возвращает вычисленное значение.

Для конкретных задач, для которых асинхронные операции являются вычислительно сложными (т.е., выполнение этих операций требует большого количества вычислительных операций), эти операции могуть быть реализованы с помощью объектов класса

System.Threading.Tasks.Future<T>,

поскольку класс Future<T> наследует класс

System.Threading.Tasks.Task

и реализует интерфейс IAsyncResult (отметим, что интерфейс IAsyncResult является в .NET, в действительности, объектом типа System.Runtime.Remoting.Messaging.AsyncResult).

Предположим, что мы имеем класс, предназначенный для вычисления числа π с произвольной точностью, задаваемой количеством требуемых десятичных знаков после запятой.

Пример 1.

```
public class PI
{
      public string Calculate (int decimalPlaces )
      {
            .   .   .
      }
}
```

Реализовать АМП, в данном случае, означает ввести в класс PI методы BeginCalculate и EndCalculate, позволяющие использовать метод Calculate асинхронным образом. Используя Future<T>, это можно сделать, добавив лишь несколько строк кода:

Пример 2.

```
public class PI
{     . . .
      public IAsyncResult BeginCalculate (int decimalPlaces)
      {
       return Future.Create ( () => Calculate(decimalPlaces) );
      }
      public string EndCalculate ( IAsyncResult ar )
      {
       var f = ar as Future<string>;
       if ( f == null )
            throw new ArgumentException ( "ar" );
       return f.Value;
      }
}
```

На самом деле, в классической асинхронной модели программирования требуется также, чтобы метод BeginXx мог принимать еще 2 аргумента:

1) функцию "обратного вызова" (AsyncCallback), которая автоматически вызывается по завершении асинхронной операции, и

2) объект, задающий состояние программы пользователя в момент асинхронного вызова (user-defined state object).

Добавление функции AsyncCallback очень легко реализовать, используя возможности, предоставляемые классом Future<T>:

Пример 3.

```
public IAsyncResult BeginCalculate (int decimalPlaces,
                                    AsyncCallback ac )
{
 var f = Future.Create ( () => Calculate (decimalPlaces) );
 if ( ac != null )
      f.Completed += delegate { ac ( f ); };
 return f;
}
```

Решение, реализованное в Примере 3, состоит в регистрации для объекта класса Future<T> события Completed, при наступлении которого теперь будет вызываться функция AsyncCallback.

Задача 1.

Реализуйте полностью класс PI с методами BeginCalculate и EndCalculate, добавив к нему соответствующий метод Main для проверки всего решения.

Однако, текущая реализация класса Future<T> не поддерживает, в отличие от класса Task, аргумент, задающий состояние. Тем не менее, эту трудность можно обойти, реализовав свой собственный класс на основе Future<T>, который будет принимать объект, задающий состояние:

Пример 4.

```
private class FutureWithState<T> : IAsyncResult
{
 public IAsyncResult Future;
 public object       State;

 public object AsyncState { get { return State; } }
 public WaitHandle AsyncWaitHandle {
      get { return Future.AsyncWaitHandle; } }
 public bool CompletedSynchronously {
      get { return Future.CompletedSynchronously; } }
 public bool IsCompleted {
      get { return Future.IsCompleted; } }
}
```

Модификации методов BeginCalculate и EndCalculate, поддерживающие задание состояния, очевидны:

Пример 5.

```
public IAsyncResult BeginCalculate
      (int decimalPlaces, AsyncCallback ac, object state )
{
 var f = Future.Create ( () => Calculate(decimalPlaces) );
 if ( ac != null ) f.Completed += delegate { ac ( f ); };
 return new FutureWithState<string> {
      Future = f, State = state };
}
public string EndCalculate ( IAsyncResult ar )
{
 var f = ar as FutureWithState<string>;
 if ( f == null )
    throw new ArgumentException ( "ar" );
 return f.Future.Value;
}
```

Аналогичный подход на основе создания контейнера может быть применен и для других сценариев, в которых требуется сохранение большего количества аргументов в добавок к IAsyncResult (Future<T>).

В качестве примера реализации АМП в конкретном случае, рассмотрим класс System.Net.NetworkInformation.Ping, который был введен в .NET Framework 2.0. Этот класс, на самом деле, выводится из класса Component и соответствует схеме асинхронного программирования, базирующегося на событиях (Event-based Asynchronous Pattern - EAP). А потому, в отличие от АМП, которая предоставляет методы BeginSend и EndSend, данный класс имеет метод SendAsync, который возвращает void, а также содержит событие

PingCompleted, которое происходит по заверешении асинхронной операции посылки сигнала ping. С помощью Future<T> легко реализовать функциональность класса Ping в рамках АМП:

Пример 6.

```
public class MyPing
{
    public IAsyncResult BeginPing(string host)
    {
        return Future.Create(() => new Ping().Send(host));
    }

    public PingReply EndPing(IAsyncResult ar)
    {
        var f = ar as Future<PingReply>;
        if (f == null) throw new ArgumentException("ar");
        return f.Value;
    }
}
```

Задача 2.

Реализуйте метод Main для данного класса, в котором асинхронным образом пингуется несколько хостов, имена которых заданы в виде списка или массива.

Однако, реализация, приведенная в Примере 6, обладает одним недостатком – функция Send, содержащая мало вычислительных операций, является синхронной операцией, а потому будет блокировать поток, в рамках которого она будет выполняться. Чтобы обеспечить реальную асинхронность операции BeginPing, нужно воспользоваться асинхронной операцией SendAsync класса Ping, а результат операции пингования получить через свойство Value объекта класса Future<T>:

Пример 7.

```
public class MyPing
{
    public IAsyncResult BeginPing(string host, AsyncCallback ac)
    {
        var f = Future<PingReply>.Create();
        if (ac != null) f.Completed += delegate { ac(f); };

        Ping p = new Ping();
        p.PingCompleted += (sender, state) =>
        {
            if (state.Error != null) f.Exception = state.Error;
            else f.Value = state.Reply;
        };
        p.SendAsync(host, null);
```

```
            return f;
        }

        public PingReply EndPing(IAsyncResult ar)
        {
            var f = ar as Future<PingReply>;
            if (f == null) throw new ArgumentException("ar");
            return f.Value;
        }
    }
```

В Примере 7, объект класса Future<T> создается без делегата, связанного с ним:

```
var f = Future<PingReply>.Create();
```

Поток, обратившийся за значением Value объекта Future<T>, заблокируется до тех пор, пока не будут установлены свойства Value или Exception этого объекта, которые, в свою очередь, получат значение в обработчике события PingCompleted. Таким образом, мы здесь имеем полностью асинхронную реализацию механизма Ping, соответствующую асинхронной модели программирования (АМП).

Задача 3.

Рассмотрите Пример 3 и выясните может ли произойти такая ситуация, когда к моменту регистрации события Completed, выполнение функции, связанной с Future, уже закончится, а потому не будет произведен необходимый вызов функции AsyncCallback ? [Подсказка: найдите в документации CTP Parallel Extensions правила регистрации делегатов для события Completed].

Теперь решим обратную задачу - покажем, как используя имеющуюся реализацию АМП, создавать объекты класса Future<T>. Суть подхода состоит в том, что объект Future<T> может создаваться без задания делегата Func<T>, представляющего вычислительную функцию, которая должна выполняться в рамках Future<T>. Свойства же Value и Exception объекта Future<T>, в этом случае, устанавливаются явно при завершении асинхронной операции:

Пример 8.

```
static Future<T> Create<T>(
      Action<AsyncCallback> beginFunc,
      Func<IAsyncResult, T> endFunc)
{
    var f = Future<T>.Create();
    beginFunc(iar => {
        try { f.Value = endFunc(iar); }
        catch (Exception e) { f.Exception = e; }
    });
    return f;
}
```

В Примере 8, вначале создается объект класса Future<T> как таковой. Затем происходит вызов делегата beginFunc, запускающий исполнение асинхронной операции. В качестве аргумента этого вызова передается делегат функции, которая будет вызываться по завершении асинхронной операции.

Покажем, как описанный подход работает в случае его применения к (асинхронным) операциям чтения из файла. В частности, класс FileStream имеет следующие асинхронные операции BeginRead и EndRead:

Пример 9.

```
IAsyncResult BeginRead(
      byte[] array, int offset, int numBytes,
      AsyncCallback userCallback, object stateObject);

int EndRead(IAsyncResult asyncResult);
```

Таким образом, если мы хотим создать объект Future<T>, который представляет асинхронную операцию чтения для FileStream, достаточно применить вновь определенный метод Create (Пример 8) следующим образом:

Пример 10.

```
var readFuture = Create<int>(
      ac => fs.BeginRead(buffer, 0, buffer.Length, ac, null),
      fs.EndRead);
```

Задача 4.

Написать метод Main, в котором попеременно асинхронно читаются 2 файла с использованием объектов Future<T>, аналогичных показанному в Пример 10.

Таким образом, если объект класса FileStream был создан с поддержкой асинхронных операций ввода/вывода и версия ОС Windows поддерживает асинхронный ввод/вывод, то при запросе на чтение из файла, будет создан объект Future<T>, но дополнительного потока создано не будет.

Способы реализации метода Create объекта Future<T>, показанного в Примере 8, могут различаться. Например, альтернативная версия может базироваться на использовании ThreadPool:

Пример 11.

```
      static Future<T> Create<T>(
            IAsyncResult iar, Func<IAsyncResult, T> endFunc)
{
    var f = Future<T>.Create();
    ThreadPool.RegisterWaitForSingleObject(
            iar.AsyncWaitHandle, delegate {
```

```
            try { f.Value = endFunc(iar); }
            catch (Exception e) { f.Exception = e; }
        }, null, -1, true);
        return f;
    }
```

В Примере 11, вместо делегата beginFunc, используется обращение к методу RegisterWaitForSingleObject класса ThreadPool. Когда произойдет событие AsyncWaitHandle, которое принадлежит классу IAsyncResult, и причиной которого является завершение асинхронной операции, то произойдет вызов endFunc и будут установлены свойства Value или Exception объекта Future<T> как в предыдущем случае. Данная версия метода Create может быть использована следующим образом:

Пример 12.

```
var readFuture = Create<int>(
      fs.BeginRead(buffer, 0, buffer.Length, null, null),
      fs.EndRead);
```

Задача 5.

Реализуйте метод Main для использования функции Create из Примера 11, и распечатайте ID главного потока и потока в рамках которого будет выполняться присваивание

```
      f.Value = endFunc(iar);
```

Семинарское занятие № 7

1 Параллельные шаблоны агрегирования в PLINQ

Прежде чем перейти к конструкциям параллельных шаблонов агрегирования в PLINQ, рассмотрим само понятие агрегирования в языке запросов LINQ.

1.1 Понятие агрегирования в LINQ

Операция агрегирования - это вычисление некоторого суммарного, сводного значения по некоторому множеству (например, списку или последовательности) элементов. Примерами такого рода операций являются вычисление суммы последовательности чисел, минимального или максимального значения последовательности, и др.

Обычно, эта операция выполняется по некоторому стандартному шаблону − перебираются п порядку элементы исходной последовательности, а текущий результат сохраняется в специальной переменной, называемой аккумулятором. На каждом шаге перебора, к очередному элементу последовательности применяется основная вычислительная функция (например, функция, вычисляющая минимальное значение двух чисел), вторым аргументом которой является текущее значение переменной-аккумулятора. Результат применения этой функции снова записывается в переменную-аккумулятор, которая, в конечном итоге, будет хранить окончательный результат. Иногда основную вычислительную функцию называют функцией редукции, потому что с ее помощью все элементы исходной последовательности редуцируются (сводятся) к некоторому одному значению, объявляемому результатом. Дополнительными примерами операций агрегирования являются операции вычисления суммы квадратов чисел, принадлежащих последовательности, отыскание количества элементов, превышающих некоторое заданное число, и др.

В языке LINQ для вычисления операций агрегирования предусмотрена специальная функция-шаблон − функция Aggregate, реализованная, во-первых, в виде extension-метода (об extension-методах см., например, документ "C# Version 3.0 Specification", http://msdn.microsoft.com/en-us/library/ms364047(VS.80).aspx), а, во-вторых, имеющая несколько перегруженных вариантов для применения в различных ситуациях. Ниже показан пример реализации одного из таких вариантов (код, обрабатывающий ошибки, опущен):

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
        this IEnumerable<TSource> source,
        TAccumulate seed,
        Func<TAccumulate, TSource, TAccumulate> func,
        Func<TAccumulate, TResult> resultSelector
    )
{

        TAccumulate accumulator = seed;
        foreach (TSource elem in source)
        {
            accumulator = func(accumulator, elem);
        }
```

```
    return resultSelector(accumulator);
}
```

Чтобы воспользоваться этой функцией, программист должен указать

- исходную обрабатываемую последовательность (source),
- начальное значение переменной-аккумулятора (seed),
- вычислительную функцию (func) – функцию редукции,
- функцию пост-обработки результата (resultSelector).

В частности, с использованием данной функции сумму квадратов последовательности чисел можно вычислить следующим образом:

```
public static int SumSquares(IEnumerable<int> source)
{
 return source.Aggregate(0,(sum, x) => sum + x * x, (sum) => sum);
}
```

1.2 Параллельное вычисление операций агрегирования

Предположим, что вызов функции

SumSquares (Enumerable.Range (1,4))

происходит на машине с двумя ядрами (процессорами). Если предполагать, что при этом будут запущены два потока, то вполне возможно, что один поток будет вычислять сумму квадратов для {1,4}, а второй поток - для {3,2}. Таким образом, при последовательном и параллельном исполнении, шаги вычислений могут быть следующими:

Последовательное исполнение: ((((0 + 1^2) + 2^2) + 3^2) + 4^2) = 30
Параллельное исполнение: (((0 + 1^2) + 4^2) + ((0 + 3^2) + 2^2)) = 30

1.3 Промежуточные результаты при параллельных операциях агрегирования

Из примеров последовательного и параллельного вычисления суммы квадратов чисел видно, что при параллеьном вычислении операции агрегирования необходимо дополнительно вычислять функцию объединения промежуточных результатов, хранящихся в переменных-аккумуляторах. Заметим, что функция объединения промежуточных результатов, в общем случае, может не совпадать с функцией редукции. Например, в случае SumSquares функция редукции имеет вид

```
(sum, x) => sum + x * x
```

а функция объединения промежуточных результатов есть просто функция сложения

```
( x, y ) => x + y
```

Кроме того, тип (TSource) элементов обрабатываемой последовательности может, в общем случае, отличаться от типа (TAccumulate) промежуточных значений, что еще больше подчеркивает различие двух названных выше функций.

По этой причине, функция Aggregate в PLINQ имеет в качестве дополнительного параметра, по сравнению с аналогичной функцией в LINQ, функцию finalReduceFunc объединения (редукции) значений переменных-аккумуляторов:

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IParallelEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate,TSource,TAccumulate> intermediateReduceFunc,
    Func<TAccumulate, TAccumulate, TAccumulate> finalReduceFunc,
    Func<TAccumulate, TResult> resultSelector
)
```

1.4 Свойства операций параллельного агрегирования

В общем случае, не любая операция агрегирования может быть корректно вычислена с помощью шаблонов (функций Aggregate), предлагаемых в PLINQ.

Для того, чтобы определить возможно ли корректное параллельное вычисление некоторой операции агрегирования, необходимо промоделировать шаги такого вычисления. Для этого нужно предположить, что исходная последовательность элементов произвольным образом переупорядочена и "разрезана" на несколько подпоследовательностей. Для каждой подпоследовательности, переменная-аккумулятор инициализируется значением seed и к элементам подпоследовательности применяется функция редукции. Затем, промежуточные значения, хранящиеся в переменных-аккумуляторах, объединяются с помощью функции finalReduceFunc. Если такой алгоритм всегда дает корректный результат, то это значит, что его можно вычислять с помощью шаблонов агрегирования PLINQ.

Далее мы рассмотрим некоторые свойства операций агрегирования, которые обеспечивают их корректное параллельное вычисление.

Во-первых, поскольку вычисление является параллельным, то делегаты, которые подставляются в шаблон, должны корректно работать с общими данными − в случае необходимости, в них долдны быть предусмотрены необходимые операции блокировки (не забудем, что такие операции будут уменьшать эффективность распараллеливания).

Во-вторых, как было отмечено выше, при параллельном вычислении операции агрегирования порядок применения функции редукции к элементам последовательности необязательно будет совпадать с соответствующим порядком при последовательном вычислении. В примере вычисления суммы квадратов, приведенном выше, несмотря на различный порядок вычислений при последовательном и параллельном исполнении, результат оставался неизменным, поскольку оператор "+" обладает двумя важными свойствами − коммутативностью и ассоциативностью.

Оператор F(x,y) является ассоциативным, если F(F(x,y),z) = F(x,F(y,z)) и коммутативным, если F(x,y) = F(y,x), для всех значений x,y,z. Например, оператор Max обладает обоими этими свойствами, так как: Max(x,y) = Max(y,x) и Max(Max(x,y),z) = Max(x,Max(y,z)). Оператор «-» не является ни ассоциативным, ни коммутативным.

Ниже приведена таблица с примерами различных по категориям операторов:

		Ассоциативный	
		Нет	**Да**

Коммутативный	**Нет**	`(a, b) => a / b`
		`(a, b) => a - b`
		`(a, b) => 2 * a + b`
			`(string a, string b) => a.Concat(b)`
			`(a, b) => a`
			`(a, b) => b`

	Да	`(float a, float b) => a + b`
		`(float a, float b) => a * b`
		`(bool a, bool b) => !(a && b)`
		`(int a, int b) => 2 + a * b`
		`(int a, int b) => (a + b) / 2`
			`(int a, int b) => a + b`
			`(int a, int b) => a * b`
			`(a, b) => Min(a, b)`
			`(a, b) => Max(a, b)`

Таким образом, необходимым условием корректности параллельного вычисления операции агрегирования является коммутативность и ассоциативность функции редукции.

При использовании LINQ, программист может задавать в качестве начального значения (seed) переменной-аккумулятора любое значение. Например, вычислить «сумму квадратов плюс 5» можно с помощью следующей операции агрегирования:

```
public static int SumSquaresPlus5(IEnumerable<int> source)
{
 return source.Aggregate(5, (sum, x) => sum + x * x, (sum) => sum);
}
```

Однако, при выполнении этой операции параллельно, переменная-аккумулятор каждого потока будет приниициализирована числом 5, что даст, в итоге, неверный результат.

Также неверными, в общем случае, начальными значениями переменных-аккумуляторов будут значение, возвращаемое оператором default для некоторого типа T, или первый элемент исходной последовательности.

Правильным начальным значением переменной-аккумулятора при параллельных вычислениях будет "единичный" (с алгебраической точки зрения) элемент, т.е., такой элемент, многократное применение к которому функции редукции не изменяет результат (например, пустой список для функции конкатенации списков).

Пусть F() – функция редукции, G() – функция объединения промежуточных результатов (полученных отдельными потоками), а s есть начальное значение переменных-аккумуляторов. Тогда, чтобы определенная пользователем операция агрегирования могла быть корректно вычислена параллельно с использованием средств PLINQ, необходимо и достаточно чтобы:

- функции редукции корректно работали с общими данными в многопоточной среде
- G(a,b) = G(b,a) для любых a, b
- G(G(a, b), c) = G(a, G(b, c)) для любых a, b, c
- F(a, x) = G(a, F(s, x)), для любых a, x

Задача 1.

Реализуйте с помощью параллельной операции агрегирования вычисление среднеарифметического для последовательности целых чисел.

Задача 2.

Реализуйте с помощью параллельной операции агрегирования подсчет частоты встречаемости символов алфавита a … z в заданной строке в этом алфавите. Все частоты символов должны быть сохранены в специальном массиве.

Задача 3.

В Лекции 11 "Решето Эратосфена для нахождения простых чисел", параллельная реализация построена на основе конструкции Parallel.For. При этом, логика алгоритма допускает использование операций агрегирования. Измените реализацию алгоритма так, чтобы она использовала функцию Aggregate из PLINQ.

Семинарское занятие № 8

1 Модификация concurrent-структур данных во время перечисления их элементов

В состав PFX входят так называемые координационные структуры данных, среди которых имеются структуры для безопасного применения в многопоточной среде такие как, например, ConcurrentQueue<T> и ConcurrentStack<T>.

PFX-аналоги классов, содержащихся в пространстве имен System.Collections, имеют, во многом, сходные с ними свойства. Так, например, для объектов обоих видов классов допускается перечисление элементов этих объектов с помощью цикла foreach. Однако, стоит обратить вниманте, что для объектов обычных классов-коллекций запрещена модификация этих объектов в процессе перечисления. Например, выполнение кода, приведенного ниже, вызовет исключительную ситуацию InvalidOperationException: "Collection was modified after the enumerator was instantiated.":

```
var q = new Queue<int>(new[] { 1, 2, 3, 4, 5 });
foreach (var item in q)
{
    if (item <= 5) q.Enqueue(item * 6);
}
```

Если же изменить тип переменной q с `Queue<int>` на тип concurrent-коллекции ConcurrentQueue<int>:

```
var q = new ConcurrentQueue<int>(new[] { 1, 2, 3, 4, 5 });
foreach (var item in q)
{
    if (item <= 5) q.Enqueue(item * 6);
}
```

то выполнение такого кода уже не вызовет исключительной ситуации, а в после выполнения цикла foreach, согласно правил, зафиксированных в текущей реализации библиотеки PFX, в очереди будет находиться 10 элементов.

Такое изменение свойств объектов-коллекций было специально принято в PFX, поскольку параллельная обработка (в т.ч., параллельное перечисление и изменение) concurrent-структур данных является основным вариантом их использования.

Задача 1.

Перечислите элементы, которые будут содержаться в очереди, после выполнения следующего кода:

```
var q = new ConcurrentQueue<int>(new[] { 1, 2, 3, 4, 5 });
foreach (var item in q)
{
    if (item <= 5) q.Enqueue(item * 6);
}
```

2 Сохранение порядка возвращаемых значений при параллельных вычислениях

При параллельной обработке, сохранение порядка значений, возвращаемых программой, часто требует от программиста написания дополнительного кода, сохраняющего этот порядок. Представим себе последовательное приложение, которое реализует рендеринг (порождение) кадров, допустим, видеофильма и последующую их запись в видеофайл:

```
for (int i = 0; i < numberOfFrames; i++)
{
    var frame = GenerateFrame(i);
    WriteToMovie(frame);
}
```

Этот же алгоритм можно реализовать и с помощью средств LINQ:

```
var frames = from i in Enumerable.Range(0, numberOfFrames)
             select GenerateFrame(i);
foreach (var frame in frames) WriteToMovie(frame);
```

Выполнение метода GenerateFrame может занимать достаточно много времени, поэтому имеет смысл распараллелить генерацию видеокадров. Для примера, предположим, что исполнение метода GenerateFrame в отдельном потоке является безопасным (т.е., использование общих данных, если таковые имеются, одновременно с другими потоками происходит корректно), и что порождение очередного кадра не зависит от предыдущих кадров. При параллельной генерации отдельных кадров, необходимо, тем не менее, предусмотреть средства блокировки при записи полученных кадров из разных потоков в один и тот же видеофайл:

```
using (ManualResetEvent mre = new ManualResetEvent(false))
{
    int count = numberOfFrames;
    object obj = new object();
    for (int i = 0; i < numberOfFrames; i++)
    {
        ThreadPool.QueueUserWorkItem(state =>
        {
            var frame = GenerateFrame((int)state);
            lock(obj) WriteToMovie(frame);

            if (Interlocked.Decrement(ref count) == 0)
                mre.Set();
```

```
        }, i );
    }
    mre.WaitOne();
}
```

Легко видеть, что полученное параллельное решение имеет серьезный недостаток – оно не обеспечивает правильный порядок сгенерированных кадров в результирующем видеофайле. Сохранить порядок возвращаемых значений можно, например, с использованием массива событий ManualResetEvent:

```
var frames = new Bitmap[numberOfFrames];
var events = (from i in Enumerable.Range(0, numberOfFrames)
              select new ManualResetEvent(false)).ToArray();

for (int i = 0; i < numberOfFrames; i++)
{
    ThreadPool.QueueUserWorkItem(state =>
    {
        int frameNum = (int)state;
        frames[frameNum] = GenerateFrame(frameNum);
        events[frameNum].Set();
    }, i);
}

for (int i = 0; i < numberOfFrames; i++)
{
    events[i].WaitOne();
    WriteToMovie(frames[i]);
}
```

Представленный выше код является корректным – он обеспечивает правильный порядок кадров в результирующем видеофайле, но не эффективным. Неэффективность состоит в том, что на каждый генерируемый кадр заводится отдельное событие ManualResetEvent, которое реализуется через исполнение некоторого кода из ядра операционной системы. Замен событий можно воспользоваться классом Future<T> для реализации того же алгоритма:

```
var frames = (from i in Enumerable.Range(0, numberOfFrames)
              select Future.Create(() => GenerateFrame(i))).
             ToArray();
foreach (var frame in frames) WriteToMovie(frame.Value);
```

В приведенном выше фрагменте, вместо массива событий создается массив объектов класса Future<T>. Исполнение соответствующих им делегатов происходит параллельно, а при последовательной записи кадров в видеофайл происходит ожидание завершения работы соответствующего делегата. Код стал более кратким и более эффективным.

Возможен еще один вариант решения этой задачи, в котором не используются средства LINQ, но в добавок к Future<T> задействована очередь:

```
var frames = new Queue<Future<Bitmap>>();
for (int i = 0; i < numberOfFrames; i++)
```

```
{
    var num = i;
    frames.Enqueue(Future.Create(() => GenerateFrame(num)));
}
while (frames.Count > 0) WriteToMovie(frames.Dequeue().Value);
```

Наконец, для решения этой задачи средствами PLINQ, можно применить опцию PreserveOrdering перечисления ParallelQueryOptions:

```
var frames = from i in Enumerable.Range(0, numberOfFrames).
    AsParallel(ParallelQueryOptions.PreserveOrdering)
             select GenerateFrame(i);
foreach (var frame in frames) WriteToMovie(frame);
```

Задача 1.

Пусть дан массив A, содержащий натуральные числа, большие 1. Напишите параллельную программу, которая записывает в другой массив B все простые числа из массива A, сохраняя их относительный порядок в исходном массиве. Т.е., если массив A содержит, например, числа 14,13,21,25,31,20,17, то массив B должен содержать числа 13, 31, 17 в указанном порядке.

Семинарское занятие № 9

Рекурсия и параллелизм (часть 3)

Одной из целей при проектировании библиотеки Parallel FX для .NET Framework было облегчить реализацию рекурсивных параллельных операций и обеспечить для них максимально возможную эффективность.

В частности, используя средства PLINQ, обход и обработка вершин дерева записывается в виде нескольких строк кода. Используя итератор класса Tree<T>, который был реализован в Примере 9, метод Process запишется следующим образом:

Пример 12.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    GetNodes(tree).AsParallel().ForAll(action);
}
```

Внутренние действия, производимые библиотекой PFX при реализации данного PLINQ-фрагмента, очень похожи на действия из Примера 9, в котором создаются несколько потоков, которые выбирают вершины для обработки с помощью конструкции перечисления, используя механизм запирания (lock). Однако, в PLINQ такой подход реализован более эффективно путем увеличения количества вершин, извлекаемых из перечисления за один раз, что минимизирует использование конструкции lock (см. Задачу 7).

На самом деле, как и в ранее приведенных реализациях, в Примере 12 осуществляется

последовательный проход по дереву с запуском действий обработки (actions) в асинхронном режиме. Чтобы реализовать параллельный проход по дереву, когда различные поддеревья обрабатываются различными потоками, можно воспользоваться библиотекой TPL (Task Parallel Library):

Пример 13.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    Parallel.Invoke(
        () => action(tree.Data),
        () => Process(tree.Left, action),
        () => Process(tree.Right, action));
}
```

Оператор Parallel.Invoke потенциально запускает параллельное исполнение всех трех операторов из данного примера, заканчивая свою работу только по завершении их всех. Отличие от аналогичного примера, использующего класс ThreadPool (см. Пример 5), состоит в том, что здесь не может наступить состояние дедлока ввиду нехватки потоков для исполнения, поскольку реализация TPL по максимуму использует текущий (главный) поток для выполнения в нем обработки, которая, при наличии свободных потоков, исполнялась бы ими.

Если необходимо более гибкое управление параллельным исполнением отдельных фрагментов кода, то они могут быть оформлены в виде задач:

Пример 14.

```
public static void Process<T>(Tree<T> tree, Action<T> action)
{
    if (tree == null) return;
    var t1 = Task.Create(
        delegate { Process(tree.Left, action); });
    var t2 = Task.Create(
```

```
            delegate { Process(tree.Right, action); });
        action(tree.Data);
      Task.WaitAll(new Task[] { t1, t2 });
    }
```

Аналогичные методы могут быть применены к реализации алгоритма быстрой сортировки:

Пример 15.

```
public static void Quicksort<T>(T[] arr, int left, int right)
       where T : IComparable<T>
{
    if (right > left)
    {
        int pivot = Partition(arr, left, right);
        Parallel.Invoke(
            () => Quicksort(arr, left, pivot - 1),
            () => Quicksort(arr, pivot + 1, right));
    }
}
```

а также к реализации алгоритма сортировки слиянием:

Пример 16.

```
public static void Mergesort<T>(
      T[] arr, int left, int right) where T : IComparable<T>
{
    if (right > left)
    {
        int mid = (right + left) / 2;
        Parallel.Invoke(
            () => Mergesort(arr, left, mid),
            () => Mergesort(arr, mid + 1, right));
        Merge(arr, left, mid + 1, right);
    }
}
```

Хотя параллельные версии этих алгоритмов мы получили очень просто, однако, этот код, по-прежнему, имеет проблемы. Реализация конструкции Parallel.Invoke построена на создании задач (см. Пример 14) для отдельных операций и ожидании завершения работы этих задач. Если вычислительная сложность отдельных

операций мала, то накладные расходы на создание задач и ожидание завершения их работы будут велики по сравнению с основными вычислительными операциями. Это может приводить к тому, что параллельная реализация будет работать медленнее, чем последовательная.

Для того, чтобы решить эту проблему, необходимо немного усложнить код, применив широко известный механизм на базе использования порога (threshold). Идея применения порога состоит в том, что мы вводим параллелизм в исполнение программы до тех пор, пока не будут загружены достаточной работой все ядра (процессоры), после чего происходит переключение на последовательное исполнение работ в рамках запущенных параллельных потоков, что позволяет избежать дополнительных накладных расходов. Например, в случае обхода дерева, используя понятие глубины вершины дерева в качестве порога, метод Process может выглядеть следующим образом:

Пример 17.

```
private static void Process<T>(
    Tree<T> tree, Action<T> action, int depth)
{
    if (tree == null) return;
    if (depth > 5)
    {
        action(tree.Data);
        Process(tree.Left, action, depth + 1);
        Process(tree.Right, action, depth + 1);
    }
    else
    {
        Parallel.Invoke(
            () => action(tree.Data),
            () => Process(tree.Left, action, depth + 1),
            () => Process(tree.Right, action, depth + 1));
    }
}
```

В Примере 17 присутствуют одновременно последовательная и параллельная реализации обхода дерева, а переключение с одной реализации на другую происходит в зависимости от глубины вершины, которая обрабатывается в текущий момент. Следует, однако, заметить, что сама по себе глубина не всегда может служить эффективным порогом − хорошей здесь иллюстрацией являются несбалансированные деревья. Тем не менее, использование порогов может существенно повысить эффективность реализации параллельных алгоритмов сортировки (см. Задачу 9).

Задача 10.

Реализуйте алгоритмы Quicksort и Mergesort с помощью конструкции Parallel.Invoke, используя механизм порогов.

Семинарское занятие № 10

Параллельный рендеринг изображений

Рендеринг изображений − это построение графических образов на основе их символьных и/или числовых описаний. В силу специфики самой проблемы, эта задача является вычислительно сложной, а потому для ее решения часто применяются параллельные алгоритмы.

В дистрибутиве библиотеки Parallel Extensions to the .NET Framework имеется четыре реализации задачи рендеринга изображений на основе метода трассировки лучей (ray tracing). Описание самого этого метода и, в частности, объяснение его математических основ можно, например, найти на странице http://www.cs.unc.edu/~rademach/xroads-RT/RTarticle.html.

Целью данного семинарского занятия является ознакомление со способами распараллеливания алгоритма рендеринга изображений на основе трассировки лучей и конструкциями из библиотек TPL и PLINQ, используемыми для этого. Отметим также, что само (последовательное) ядро метода трассировки лучей не представляет собой оптимизированную реализацию, и вопросы ускорения его работы здесь не обсуждаются.

Рассматриваемые здесь программы рендеринга изображений базируются на двух основных реализациях метода трассировки лучей, обе написанных на языке C#, но

(а) в первом случае, с использованием стандартных конструкций императивного языка программирования, и

(б) во втором случае, с использованием запросов на языке LINQ (http://blogs.msdn.com/lukeh/archive/2007/04/03/a-ray-tracer-in-c-3-0.aspx).
(В действительности, имеется еще одна реализация с использованием LINQ, в которой весь алгоритм рендеринга выражен в виде одного, но очень большого, LINQ-запрсоа, см. http://blogs.msdn.com/lukeh/archive/2007/10/01/taking-linq-to-objects-to-extremes-a-fully-linqified-raytracer.aspx).

В дистрибутиве библиотеки PFX, эти два базовых примера расширены следующим образом:

1. Исходная программа рендеринга на языке C# распараллелена средствами библиотеки PFX (а именно, с использованием Task Parallel Library и некоторых координирующих структур данных. Для полученной таким образом версии, имеются ее варианты на языках Visual Basic и F#, демонстрирующие возможности использования библиотеки PFX из любых .NET-языков. Эти реализации можно найти в директории …\Samples\RayTracer\.. библиотеки PFX.
2. Программа рендеринга на базе LINQ распараллелена средствами PLINQ. Данная реализация находится в директории …\Samples\LINQRayTracer.

В перечисленных реализациях задачи рендеринга изображений используется довольно большое количество конструкций, предоставляемых библиотекой PFX:

- System.Threading.Parallel
- System.Threading.Task
- System.Threading.TaskManager
- System.Threading.TaskManagerPolicy
- System.Threading.LazyInit<T>
- System.Threading.Collections.IConcurrentCollection<T>
- System.Threading.Collections.ConcurrentQueue<T>
- System.Linq.ParallelQuery

Кроме того, в этих реализациях также задействованы некоторые стандартные .NET-конструкции, связанные с потоками:

- System.Threading.Interlocked
- System.Windows.Forms.Control.BeginInvoke
- System.Threading.Monitor (конструкция lock).

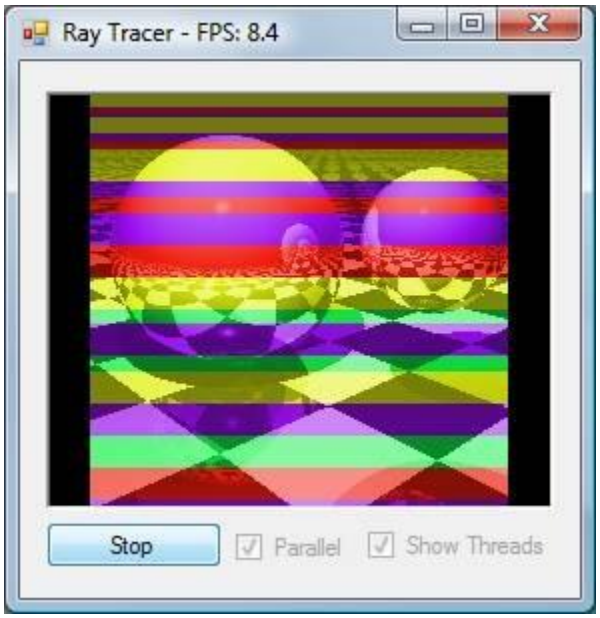
Задача 1.

Скомпилируйте и запустите программу RayTracer на языке C#, воспользовавшись решением Samples\RayTracer\C#\RayTracer.sln. С помощью кнопки Start, запустите процесс анимации в однопоточном режиме:

Задача 2.

Переключив режим на “Parallel”, запустите процесс анимации в параллельном режиме. Сравните скорость анимации (кадров в секунду – frames per second, FPS), отображаемую в титульной строке окна приложения, в последовательном и параллельном режимах.

Воспользовавшись опцией “Show Threads” (доступной только в режиме “Parallel”), запустите процесс рендеринга с цветовой разметкой точек (пикселов) изображения, указывающей распределение воспроизведенных точек по параллельным потокам:



(Приведенное изображение получено на компьютере с 4-мя ядрами).

Из приведенного выше скриншота видно, что пикселы по потокам распределяются построчно, причем это распределение не является равномерным между потоками. Причина этого состоит в том, что распараллеливание основного процесса ренедеринга реализовано с помощью конструкции Parallel.For, исполнение которой основано на динамической балансировке нагрузки между рабочими потоками с целью поддержания полной нагрузки всех потоков (процессоров).

То, что пикселы распределяются по потокам построчно, заложено в самом приложении (см. фрагмент программного кода ниже), и этот прием обеспечивает достаточно небольшие накладные расходы, идущие на распределение работы между потоками. Наоборот, параллелизация на уровне отдельных пикселов привела бы к очень большим накладным расходам и резко бы снизила производительность всего приложения. При запуске на 4-хядерной машине, при отключенной опции “Show Threads”, построчная декомпозиция обеспечивает приблизительно 3-ехкратное ускорение рендеринга. Учитывая, что воспроизведение кадров на экране и обновление экранной формы снижает общую скорость рендеринга, ускорение для только вычислительных операций может быть равным для данного приложения от 3.5 до 4.0.

В программе RayTracer (см. файл Raytracer.cs) ключевыми функциями, в которых реализован рендеринг, являются

a) RenderSequential(),
b) RenderParallel() и
c) RenderParallelShowingThreads().

Единственное различие в реализациях последовательного и параллельного рендеринга заключается в оформлении внешнего цикла рендеринга:

```
for(int y=0; y < screenHeight; y++)
{
 /* funcBody */
}
```

(в RenderSequential()), и

```
Parallel.For(0, screenHeight, y =>
       /* funcBody */
)
```

(в RenderParallel()).

Главная проблема любого параллельного приложения сотоит в обеспечении потокобезопасности тех фрагментов кода, которые выполняются в разных потоках (как, например, funcBody в параллельной версии). Это означает, что TraceRay() и другие действия, производимые в параллельном цикле, не могут изменять общие для нескольких потоков переменные, если не обеспечена надлежащая синхронизация этих потоков, либо эти модификации гарантированно являются безопасными. Например, можно заметить, что funcBody изменяет общую структуру данных rgb[], но это именно тот случай, для которого имеются гарантии безопасности модификаций: каждый элемент этого массива записывается

однократным исполнением funcBody, а потому конфликты между потоками невозможны. Общие правила обеспечения потокобезопасности сводятся к

a) упрощению кода, который исполняется в параллельных потоках,
b) использованию блокировок (конструкции lock) при модификации общих данных,
c) анализу и тестированию кода для обеспечения его корректности при параллельном исполнении.

В программе Raytracer используется явно один объект класса System.Threading.Task для управления процессом анимации, а режимы, касающиеся параллелизма, устанваливаются с помощью объекта класса TaskManager. Следует отметить, что этот TaskManager, связанный с вышеуказанным объектом класса Task, становится TaskManager.Current для каждой задачи, создаваемой внутри основной задачи. Эти неявные новые задачи создаются при исполнении Parallel.For, и на них распространяются конфигурационные параметры менеджера задач TaskManager.Current.

Другое полезное свойство реализации цикла анимации с использованием Task состоит в том, что с помощью класса Task можно обеспечить простые средства управления процессом анимации, в частности, средства снятия этого процесса. А именно, тело функции btnStartStop_Click() в MainForm.cs содержит такой оператор создания объекта класса Task:

```
_renderTask = Task.Create(RenderLoop,
      chkParallel.Checked ? _parallelTm.Value : _sequentialTm.Value);
```

Этот оператор задает, что внутри задачи будет выполняться функция RenderLoop, а управление задачей будет осуществляться с помощью одного из TaskManager'ов − _parallelTm или _sequentialTm. С помощью первого TaskManager'а − _parallelTm, на каждое ядро будет спланировано исполнение одного потока, тогда как с помощью _sequentialTm будет спланирован запуск только одного потока для осуществления последовательного рендеринга.

В действительности, каждая из этих двух переменных имеет тип LazyInit<TaskManager>, который обеспечивает создание основного типа (в данном случае, типа TaskManager) только в случае, когда переменная действительно становится используемой. Хотя использование параметризованного типа LazyInit<T> в данном случае не является показательным, однако этот тип может быть очень полезен для объектов, создание которых является ресурсо емкой операцией. Например, если данное приложение никогда не будет запускаться в параллельном режиме, то для него не будет создан TaskManager, управляющий параллельным исполнением.

Код, отрабатывающий при нажатии кнопки "Stop" и останавливающий анимацию, имеет вид:

```
_renderTask.ContinueWith(delegate
{
      BeginInvoke((Action)delegate
      {
       chkParallel.Enabled = true;
       chkShowThreads.Enabled = chkParallel.Checked;
       btnStartStop.Enabled = true;
       btnStartStop.Text = "Start";
      });
```

```
});
_renderTask.Cancel();
```

Этот код восстанавливает в исходном виде GUI, в частности, состояние его различных элементов. Здесь нужно отметить использование вызова BeginInvoke(), который производится относительно MainForm, чтобы обеспечить выполнение данного кода в потоке, связанном с GUI. Такой вызов необходим, поскольку поток задачи выполняется как фоновый рабочий поток.

Только после того, как с задачей связаны действия постобработки (посредством ContinueWith), происходит непосредственное снятие задачи через вызов Task.Cancel(). Отметим, что внутри RenderLoop() проверяется условие t.IsCanceled для обнаружения снятия задачи и корректного выхода из цикла анимации.

При стандартной перегрузке (overload) функции Task.ContinueWith(), запуск связанного сней кода будет происходить всякий раз, когда задача заканчивает свою работу, независимо от причины завершения: успешное окончание, аварийное завершение или снятие. Однако, существуют другие возможности перегрузить функцию ContinueWith для изменения условий её срабатывания (см. API для Parallel Extensions for .NET Framework).

Одним из важных вопросов в реализации эффективного рендеринга является управление объектами типа Bitmap, представляющие построенные и подготовленные для вывода на экран изображения. Операция создания объекта Bitmap (особенно большого размера) является ресурсозатратной и выполняется за достаточно большое время. Поэтому простой подход, состоящий в ожидании завершения отображения очередного Bitmap-объекта в GUI, а затем в переходе к вычислению следующего, является очень неэффективным. Один из способов решения этой проблемы заключается в использовании классического метода двойной буферизации, когда один Bitmap-объект используется для отрисовки сцены, а второй такой объект предназначен уже для вывода на экран (именно такой подход был реализован в примере RayTracer в одном из ранних версий библиотеки PFX). Более общее и более эффективное решение состоит просто в использовании очереди (вместо буфера размерности 2) изображений для их отображения на экран, которая пополняется с максимальной скоростью изображениями, генерируемыми основным процессом рендеринга. При этом, каждый построенный образ отсылается на отображение путем вызова Form.BeginInvoke() с соответствующим делегатом. Вместо того, чтобы многократно создавать и удалять объекты типа Bitmap, в приложении реализовано повторное использование таких объектов посредством класса ObjectPool<T> (см. моуль ObjectPools.cs), построенного, в свою очередь, на основе класса ConcurrentQueue<T>. С помощью класса ObjectPool<T> можно повторно использовать объекты типа T, прибегая к созданию новых объектов этого типа только в случае, когда повторноиспользуемые объекты на текущий момент отсутствуют. Использование при этом (в качестве базового) класса ConcurrentQueue<T> позволяет обеспечить потокобезопасность класса ObjectPool<T>.

Приложение RayTracer, написанное на C#, портировано также на языки Visual Basic и F#. В этих приложениях реализованы все темеханизмы и свойства, которые были описаны выше, за исключением раскрашивания частей изображения, обработанных различными потоками. В частности, для тог, чтобы построить и запустить версию приложения на языке F#, необходимо загрузить и установить пакет поддержки языка F# со страницы

http://research.microsoft.com/fsharp/fsharp.aspx. (Для компиляции программы рендеринга на F#, в случае использования 64-разрядной машины, необходимо изменить конфигурацию проекта, чтобы подключить соответствующую System.Core.dll, поскольку ее месторасположение отличается от места, где эта библиотека находится на 32-хразрядных машинах).

Программа на языке F# непосредственно использует библиотеку Parallel Extensions for .NET, и для нее также достигается высокая производительность при исполнении на многоядерной машине:

```
member this.RenderToArrayParallel(scene, rgb : int[]) =
Parallel.For(0, screenHeight, fun y ->
    let stride = y * screenWidth
    for x = 0 to screenWidth - 1 do
        let color = TraceRay ({Start = scene.Camera.Pos; Dir =  GetPoint x
       y scene.Camera }, scene, 0)
                let intColor = color.ToInt ()
                rgb.[x + stride] <- intColor)
```

В директории Samples\LINQRayTracer расположена еще одна, совершенно отличная по реализации от рассмотренных, версия программы рендеринга, использующая язык запросов LINQ, который является частью языка C# 3.0. Это приложение не выполняет анимации, однако строит и воспроизводит более сложную сцену, на которой хорошо виден эффект от параллелизации.

Исходный запрос (см. http://blogs.msdn.com/lukeh/archive/2007/04/03/a-ray-tracer-in-c-3-0.aspx) был записан в виде

```
from y in Enumerable.Range(0, screenHeight)
                    …
   select from x in Enumerable.Range(0, screenWidth)
```

который представляет собой 2-D цикл по каждому пикселу в каждой строке. Для того, чтобы запустить этот запрос в режиме параллельного исполнения, разбив изображение на отдельные строки (как в предыдущих версиях программы рендеринга), достаточно указать, чтобы самый внешний цикл исполнялся параллельно:

```
from y in Enumerable.Range(0, screenHeight).AsParallel()
                    …
   select from x in Enumerable.Range(0, screenWidth)
```

Использование конструкции AsParallel() из пакета PLINQ, позволяет распределить исполнение внешнего цикла по всем доступным ядрам. После того, как изображение получено в виде объекта pixelsQuery, для отображения на экране, его необходимо перевести в массив int[] rgb, что также можно выполнить параллельно:

```
pixelsQuery.ForAll(row =>
{
      foreach (var pixel in row)
      {
       rgb[pixel.X + (pixel.Y * screenWidth)] =  pixel.Color.ToInt32();
      }
```

```
            int processed = Interlocked.Increment(ref rowsProcessed);
            if (processed % rowsPerUpdate == 0 ||
                processed >= screenHeight) updateImageHandler(rgb);
        });
```

Здесь, делегат, предназначенный для обработки отдельной строки, осуществляет доступ к общему счетчику rowsProcessed способом, обеспечивающим потокобезопасность при параллельном исполнении.

Задача 3.

Изучите вариант реализации рендеринга, использующий LINQ, представленный на http://blogs.msdn.com/lukeh/archive/2007/10/01/taking-linq-to-objects-to-extremes-a-fully-linqified-raytracer.aspx . Разработайте параллельный вариант этой программы.