Введение в параллельное программирование на языке МС#

Сердюк Ю. П. (Yury@serdyuk.botik.ru)
Институт программных систем РАН,
г. Переславль-Залесский

Переславль-Залесский
2007

Содержание

1. Введение

Широкое распространение вычислительных систем с массовым параллелизмом, таких как многоядерные процессоры, кластеры и GRID-архитектуры, с особой остротой поставило вопрос о разработке высокоуровневых, мощных и удобных в использовании языков программирования, позволивших бы создавать сложное, но, одновременно и надежное программное обеспечение, эффективно использующее возможности параллельных, распределенных вычислений и легко масштабируемое на заданное число процессоров (ядер), узлов или машин.

Доступные на настоящий момент программные интерфейсы и библиотеки для организации параллельных вычислений, такие как OpenMP (www.openmp.org) (для систем с общей (shared) памятью) и MPI (Message Passing Interface, www.mcs.anl.gov/mpi) (для систем на основе передачи сообщений), ориентировны, в основном, на языки C и Fortran, а потому являются очень низкоуровневыми и неадекватными современным языкам объектно-ориентированного программирования, таким как C++, C# и Java. В частности, одной из причин низкоуровневости этих средств является то, что они опираются либо на вызовы библиотечных функций, либо на аналоги таких функций – препроцессорные директивы, а не на соответствующие, “родные” конструкции языков программирования.

В общем случае, современный высокоуровневый язык программирования состоит из двух частей:

1) базовых конструкций языка как таковых, и

2) совокупности специализированных библиотек, доступных через соответствующие API (Application Programming Interfaces).

Современные требования к увеличению количества программистов, владеющих основами параллельного программирования, и повышению продуктивности их работы (в частности, через более высокий уровень абстракции предоставляемых им языковых средств), а также повышению надежности и безопасности создаваемых ими программ, обусловили тенденцию перетекагия ключевых понятий наиболее важных API в соответствующие базовые конструкции языков программирования.

Одним из последних достижений в этом направлении является введение модели асинхронного параллельного программирования в рамках языка Polyphonic C# на основе платформы .NET фирмы Microsoft **[L1]**. В свою очередь, эта модель базируется на так называемом join-исчислении **[L2]** – математическом исчислении процессов с высокоуровневым механизмом обработки сообщений, адекватно абстрагирующим соответствующий низкоуровневый механизм (на основе IP-адресов, портов и сокетов), который используется в современных компьютерных системах.

Введение новых конструкций для параллельного программирования – асинхронных методов и связок (chords) в язык Polyphonic C#, который является расширением языка C#, позволяет обойтись без библиотеки System.Threading, которая обычно требуется для реализации мультипоточных приложений в рамках .NET. С другой стороны, введение новых конструкторов типов данных (потоков (streams), анонимных структур, разделенных объединений (discriminated unions) и др.) вместе с соответствующими средствами определения запросов в язык Cω **[L3]** (которые, вначале, вошли в проект Linq, http://msdn2.microsoft.com/en-us/netframework/aa904594.aspx, а затем стали частью спецификации языка C# 3.0, http://msdn2.microsoft.com/en-us/vcsharp/aa336745.aspx) делает ненужной подсистему для работы с данными ADO.NET (а именно, традиционные библиотеки System.Data и System.XML, предназначенные для работы с реляционными и слабоструктурированными данными).

Мы предлагаем сделать следующий шаг в этом направлении – ввести в объектно-ориентированный язык высокоуровневые конструкции для создания параллельных и распределенных программ, и, тем самым, освободить программиста от необходимости использования библиотек System.Threading и System.Remoting, которые требуются для разработки параллельных и распределенных приложений на базе .NET. С практической точки зрения, цель, которая преследовалась разработчиками языка MC#, заключалась в разработке языка для промышленного параллельного и распределенного программирования, которое, в настоящее время, вовлекает в себя все больше и больше человеческих ресурсов в связи с наступлением эры многоядерных процессоров. Этот язык призван заменить языки C и Фортран в разработке приложений указанного типа. В этом отношении, подход, принятый в рамках проекта MC#, совпадает с подходом, принятым при разработке языка X10 **[L4]**, который ориентирован как на многоядерные, так и на "неоднородные кластерные вычисления". Разработка языка X10 является частью совместного проекта PERCS (Productive Easy-to-use Reliable Computer Systems) фирмы IBM с несколькими академическими партнерами. Этот проект нацелен на создание к 2010г. новых масштабируемых систем, которые должны обеспечить десятикратное увеличение продуктивности работы программистов при разработке параллельных приложений **[L5]**.

В данном учебном введении рассматриваются основы программирования на языке MC#, предназначенном для написания программ, работающих на всём спектре параллельных архитектур – от многоядерных процессоров до Grid-сетей. Единственное требование к таким системам со стороны MC# - на них должна быть установлена среда исполнения CLR (Common Language Runtime) с соответствующим набором библиотек. На машинах с операционной системой Windows реализацией такой среды является Microsoft .NET Framework, а на машинах с операционной системой Linux – система Mono (www.mono-project.com), которая является свободной реализацией платформы .NET для Unix-подобных систем.

Язык MC# является адаптацией и развитием базовых идей языка Polyphonic C# на случай параллельных и распределенных вычислений. Язык Polyphonic C# был разработан в 2002г. в Microsoft Research Laboratory (г. Кембридж, Великобритания) Н. Бентоном (N. Benton), Л. Карделли (L. Cardelli) и Ц. Фурнье (C. Fournet). Целью его создания было добавление высокоуровневых средств асинхронного параллельного программирования в язык C# для использования в серверных и клиент-серверных приложениях на базе Microsoft .NET Framework.

Ключевая особенность языка Polyphonic C# заключается в добавлении к обычным, синхронным методам, так называемых "асинхронных" методов, которые предназначены играть в (многопоточных) программах две основные роли:

1) автономных методов, предназначенных для выполнения базовой вычислительной работы, и исполняемых в отдельных потоках, и

2) методов, предназначенных для доставки данных (сигналов) обычным, синхронным методам.

Для синхронизации нескольких асинхронных методов, а также асинхронных и синхронных методов, в язык C#, кроме того, были введены новые конструкции, получившие название связок (chords).

При этом исполнение Polyphonic C#-программ, по замыслу авторов этого языка, по-прежнему, предполагалось либо на одной машине, либо на нескольких машинах, с зафиксированными на них асинхронными методами, взаимодействующими между собой с

использованием средств удаленного вызова методов (RMI – Remote Method Invocation), предоставляемых библиотекой System.Runtime.Remoting платформы .NET.

В случае языка MC#, программист может предусмотреть исполнение автономных асинхронных методов либо локально, либо удаленно. В последнем случае, метод может быть спланирован для исполнения на другой машине, выбираемой двумя способами: либо согласно явному указанию программиста (что не является типичным случаем), либо автоматически (обычно, на наименее загруженном узле кластера или машине Grid-сети). Взаимодействие асинхронных методов, в рамках языка MC#, реализуется посредством передачи сообщений с использованием каналов и обработчиков канальных сообщений. Эти каналы и обработчики определяются в MC#-программах с помощью связок в стиле языка Polyphonic C#.

Таким образом, написание параллельной, распределенной программы на языке MC# сводится к выделению с помощью специального ключевого слова ***async*** методов, которые должны быть исполнены асинхронно локально (в виде отдельных потоков), а также с помощью ключевого слова ***movable*** тех методов, которые могут быть перенесены для исполнения на другие машины.

Выбор языка C# в качестве базового, дает возможность использовать современный язык объектно-ориентированного программирования, обладающий богатым множеством библиотек (для создания Web-приложений и работы с Web-сервисами, разработки графических приложений, обработки реляционных и слабоструктурированных (XML-) документов, реализации систем с повышенными средствами безопасности, и т.д.), быстро развивающийся (см. спецификации языков C# 2.0 и C# 3.0), и, одновременно, исключающий низкоуровневые и небезопасные средства, такие, как указатели и команды резервирования и освобождения областей памяти, которые значительно снижают производительность труда программистов и надежность создаваемых ими систем.

По сравнению с применением интерфейса MPI, в программах на языке MC# нет необходимости явно управлять распределением вычислительных процессов по процессорам (ядрам, узламкластера, машинам Grid-сети), хотя необходимые для этого средства в языке также предоставляются. Чтобы написать параллельную программу на языке MC#, достаточно только указать с помощью специальных ключевых слов (***async*** или ***movable***) какие функции (методы классов) могут быть выполнены параллельно (распределенно). Кроме того, новые вычислительные процессы могут создаваться и распределяться по доступным узлам (в распределенном режиме) динамически во время исполнения MC#-программ, что невозможно для MPI-программ. Средства динамического создания так называемых “активностей” также предусмотрены в языке X10, но с явным указанием места исполнения новой активности. Таким образом, в системе исполнения X10-программ не предусмотрены средства автоматического распределения активностей по доступным физическим процессорам. Более точно, асинхронная активность в языке X10 создается с помощью оператора ***async*** *(P) S*, где *P* есть выражение, задающее место исполнения активности, а *S* есть оператор, представляющий вычислительное тело активности. В языке X10, внутри одного метода класса возможно создание нескольких активностей, тогда как в языке MC# параллельность возможна только на уровне методов класса.

В сравнении с MPI, в программах на MC# (в распределенном режиме) нет необходимости в ручном программировании сериализации объектов (данных) для их пересылке на другой узел – Runtime-система языка MC# производит сериализацию/десериализацию пересылаемых объектов автоматически.

Структура данного учебного пособия построена следующим образом. В Разделе 2 дано описание новых средств языка MC# - ***async-*** и ***`movable`***-методов, а также каналов и

обработчиков канальных сообщений. Приведены примеры определения связок каналов и обработчиков, а также отмечены особенности программирования для локального (многоядерного) и распределенного (кластерного) режимов.

Раздел 3 посвящен некоторым типичным задачам, имеющим параллельные алгоритмы решения, на которых демонстрируется использование специфических конструкций языка MC# и общие методы построения параллельных программ на этом языке.

В Приложении даны полные тексты программ из Раздела 3.

2. Новые средства языка MC#: async- и movable-методы, каналы и обработчики

В любом традиционном языке объектно-ориентированного программирования, таком, как например, C#, обычные методы являются синхронными - вызывающая программа всегда ожидает завершения вычислений вызванного метода, и только затем продолжает свою работу.

При исполнении программы на параллельной архитектуре, сокращение времени её работы может быть достигнуто путем распределения множества исполняемых методов на несколько ядер одного процессора, и, возможно, отправкой части из них на другие процессоры (машины) при распределенных вычислениях (Рис.1):

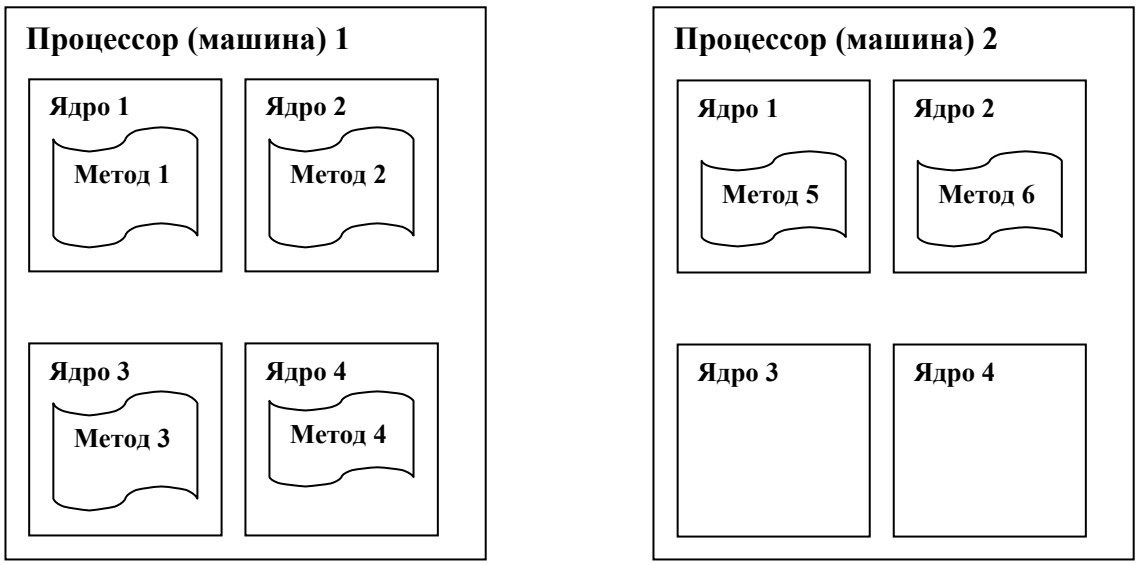

Рис. 1. Распределение исполняемых методов по ядрам и процессорам (машинам)

Разделение всех методов в программе на обычные (синхронные) и асинхронные (в том числе, на те, которые могут быть перенесены для исполнения на другие машины) производится программистом с использованием специальных ключевых слов ***async*** и ***movable***. (В языке MC#, семантика и использование ключевого слова ***async*** полностью совпадает с использованием этого слова в языке Polyphonic C# за тем исключением, что в MC# async-методы не могут встречаться в связках – см. об этом ниже).

Async- и movable-методы являются единственным средством создания параллельных процессов (потоков) в языке MC#.

Кроме средств создания параллельных процессов, любой язык параллельного программирования должен содержать конструкции

а) для обеспечения взаимодействия параллельных процессов между собой,

б) для их синхронизации.

Основой взаимодействия параллельных процессов в языке MC# является передача сообщений (в отличие от другой альтернативы – использования общей (разделяемой) памяти). Продолжая сравнение языка MC# с языком X10, следует отметить, что в языке X10 реализованы обе парадигмы взаимодействия – как на основе передачи сообщений, так и на

основе общей (разделяемой) памяти. В языке MC#, средства взаимодействия между процессами оформлены в виде специальных синтаксических категорий – каналов и обработчиков канальных сообщений. При этом, синтаксически посылка сообщения по каналу или прием из него с помощью обработчика выглядят в языке как вызовы обычных методов. В языке X10, пересылка значений с одного "места" (place – в терминологии X10) в другое, требует явного порождения (асинхронной) активности, осуществляющей транспортировку этого сообщения.

Для синхронизации параллельных процессов в MC# используются связки (chords), определяемые в стиле языка Polyphonic C#. В языке X10, для синхронизации используются более сложные конструкции под названием clocks.

1.1. Async- и movable-методы

Общий синтаксис определения async- и movable-методов в языке MC# следующий:

```
модификаторы { async | movable } имя_метода ( аргументы )
{
        < тело метода >
}
```

Ключевые слова ***async*** и ***movable*** располагаются на месте типа возвращаемого значения, поэтому синтаксическое правило его задания при объявлении метода в языке MC# имеет вид:

```
return-type ::= type | void | async | movable
```

Задание ключевого слова ***async*** означает, что при вызове данного метода он будет запущен в виде отдельного потока *локально*, т.е., на данной машине (возможно, на отдельном ядре процессора), но без перемещения на другую машину. Ключевое слово ***movable*** означает, что данный метод при его вызове может быть спланирован для исполнения на другой машине.

Отличия async- и movable-методов от обычных методов состоят в следующем:

- ✓ вызов async- и movable-методов заканчивается, по существу, мгновенно (для последних из названных методов, время затрачивается только на передачу необходимых для вызова этого метода данных на удаленную машину),
- ✓ эти методы никогда не возвращают результаты (о взаимодействии movable-методов между собой и с другими частями программы, см. Раздел 2.2 "Каналы и обработчики").

Соответственно, согласно правилам корректного определения async- и movable-методов:

- ✓ они не могут объявляться статическими,
- ✓ в их теле не может использоваться оператор ***return*** .

Вызов movable-метода имеет две синтаксические формы:

1) *имя_объекта.имя_метода (аргументы)*
(место исполнения метода выбирается Runtime-системой автоматически),

2) *имя_машины@имя_объекта.имя_метода (аргументы)*

(*имя_машины* задает явным образом место исполнения данного метода).

При разработке распределенной программы на языке MC# (т.е., при использовании в ней movable-методов и исполнении её на кластере или в Grid-сети), необходимо учитывать следующие особенности системы исполнения (Runtime-системы) MC#-программ.

Во-первых, объекты, создаваемые во время исполнения MC#-программы, являются, по своей природе *статическими*: после своего создания, они не перемещаются и остаются привязанными к тому месту (машине), где они были созданы. В частности, именно в этом месте (на этой машине) они регистрируются Runtime-системой, что необходимо для доставки канальных сообщений этим объектам и чтения сообщений с помощью обработчиков, связанных с ними (этими объектами).

Поэтому, ***первой ключевой особенностью языка MC#*** (а точнее, его семантики) является то, что, в общем случае, во время вызова movable-метода, все необходимые данные, а именно:

1) сам объект, которому принадлежит данный movable-метод, и
2) его аргументы (как ссылочные, так и скалярные значения)

только ***копируются*** (но не перемещаются) на удаленную машину. Следствием этого является то, что все изменения, которые осуществляет (прямо или косвенно) movable-метод с внутренними полями объекта, проводятся с полями объекта-копии на удаленной машине, и никак не влияют на значение полей исходного объекта.

Эта ситуация проиллюстрирована на Рис. 2.

Эта ситуация проиллюстрирована ниже:

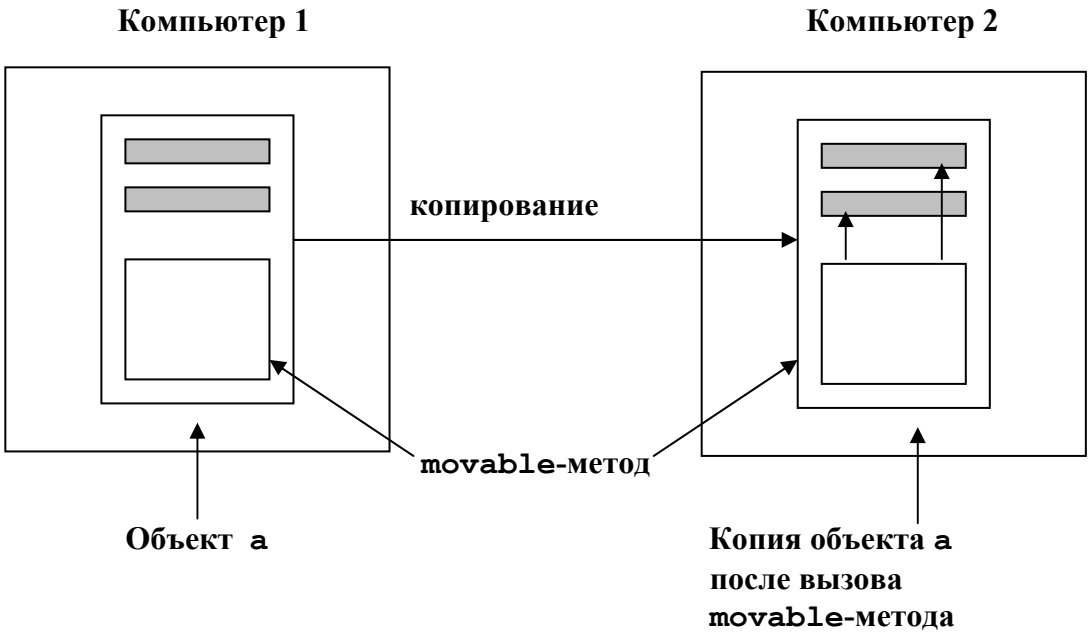

Рис.2. Вызов и исполнение movable-метода

Поэтому выполнение программы

```
class B {
  public int x;
  public B() { }
  movable Compute() {
    x = 2;
  }
```

```
}

class A {
  public static void Main( String[] args ) {
    B b = new B();
    b.x = 1;
    Console.WriteLine( "Before: x = " + b.x );
    b.Compute();
    Console.WriteLine( "After: x = " + b.x );
  }
}
```

даст

```
Before: x = 1
After: x = 1
```

Если копируемый (при вызове его movable-метода) объект обладает каналами или обработчиками (или же просто, они являются аргументами этого movable-метода), то они также копируются на удаленную машину. Однако, в этом случае, они становятся "прокси"-объектами для исходных каналов и обработчиков (см. об этом подробнее в Разделе 2.2).

В распределенном режиме, имеется два режима параллелизации MC#-программ: "функциональный" и "нефункциональный" (или объектный), и от выбора того или другого будет, в конечном счете, зависеть эффективность исполнения программы. Эти режимы задаются модификаторами ***functional*** и ***nonfunctional*** при объявлении movable-метода (для async-методов эти модификаторы игнорируются). Значением по умолчанию является режим ***functional***.

В функциональном режиме, объект, для которого вызывается movable-метод, не передается на удаленную машину. То есть, все данные, необходимые movable-методу, должны передаваться через его аргументы. Наоборот, путем задания модификатора ***nonfunctional***, обеспечивается передача объекта на удаленную машину.

При использовании MC#-программы на кластерных архитектурах, которые обычно состоят из главной машины (фронтенда) и подчиненных ей узлов, имеется специфика в вызове movable-метода с явным заданием места его исполнения – в этом случае, должны указываться как имя главной машины, так и имя узла в формате

имя_машины:имя_узла@имя_объекта.имя_метода (аргументы)

1.2. Каналы и обработчики

Каналы и обработчики канальных сообщений являются средствами для организации взаимодействия параллельных распределенных процессов между собой. Синтаксически, каналы и обработчики обычно объявляются в программе с помощью специальных конструкций – *связок* (chords). Впервые, в императивных языках, конструкция связок появилась в языке Polyphonic C#. Помимо объявления каналов и обработчиков, связки также играют в языке роль средств синхронизации процессов: как распределенных (использующих movable-методы), так и нераспределенных (использующих async-методы).

Например, объявление канала *sendInt* для передачи одиночных целочисленных значений вместе с соответствующим обработчиком *getInt* для получения значений из этого канала, выглядит следующим образом:

```
handler getInt int() & channel sendInt( int x ) {
  return x;
}
```

В общем случае, синтаксические правила определения связок в языке MC# имеют вид:

```
chord-declaration ::= [handler-header] [ & channel-header ]* body
handler-header ::= attributes modifiers handler handler-name
                    return-type ( formal-parameters )
channel-header ::= attributes modifiers channel channel-name
                   ( formal-parameters )
```

Связки определяются в виде членов класса. По правилам корректного определения, каналы и обработчики не могут иметь модификатора ***static***, а потому они всегда привязаны к некоторому объекту класса, в рамках которого они объявлены:

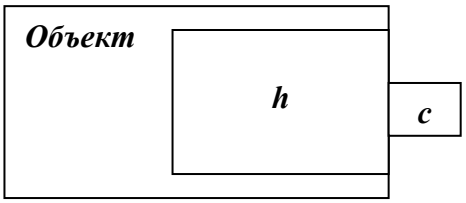

Рис. 3. Объект с каналом *c* и обработчиком *h*

Таким образом, мы можем послать целое число *n* по каналу *sendInt*, записав выражение

```
a.sendInt( n );
```

где *a* есть объект для которого определен канал *sendInt.*

Если имя канала использовано без уточняющих префиксов, например, как

```
c( n );
```

то подразумевается, как обычно, использование текущего объекта:

```
this.c ( n );
```

Обработчик используется для приема значений (возможно, предобработанного с помощью кода, являющегося телом связки) из канала (или группы каналов), совместно определенных с этим обработчиком. Например, для приема значения из канала *sendInt* можно записать

```
int m = a.getInt();
```

Если, к моменту вызова обработчика, связанный с ним канал пуст (т.е., по этому каналу значений не поступало или они все были выбраны посредством предыдущих обращений к обработчику), то этот вызов блокируется. Когда по каналу приходит очередное значение, то

происходит исполнение тела связки (которое может состоять из произвольных вычислений) и по оператору *return* происходит возврат результирующего значения обработчику.

Наоборот, если к моменту прихода значения по каналу, нет вызовов обработчика, то это значение просто сохраняется во внутренней очереди канала, где, в общем случае, накапливаются все сообщения, посылаемые по данному каналу. При вызове обработчика и при наличии значений во всех каналах соответствующей связки, для обработки в теле этой связки будут выбраны первые по порядку значения из очередей каналов.

Следует отметить, что, принципиально, срабатывание связки, состоящей из обработчика и одного или нескольких каналов, возможно в силу того, что они вызываются, в типичном случае, из различных потоков.

Аналогично языку Polyphonic C#, в одной связке можно определить несколько каналов. Такого вида связки являются главным средством синхронизации параллельных (в том числе, распределенных) потоков в языке MC#:

```
handler equals bool() & channel c1( int x )
                      & channel c2( int y ) {
   if  ( x == y )
      return ( true );
   else
      return ( false );
}
```

Таким образом, общее правило срабатывания связки состоит в следующем: тело связки исполняется только после того, как вызваны **все** методы из заголовка этой связки.

Приведенный выше пример иллюстрирует случай, когда один обработчик объявлен для нескольких каналов:

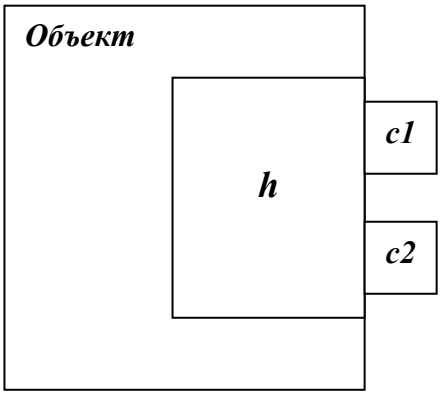

Рис. 4. Объект с одним обработчиком для нескольких каналов

Также, возможно объявление канала, разделяемого между несколькими обработчиками:

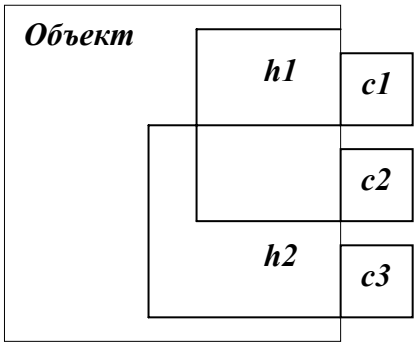

Рис. 5. Объект с разделяемым каналом

Синтаксически, такое объявление возможно путем применения двух связок, как показано в примере ниже:

```
handler h1 int() & channel c1( int x )
                 & channel c2( int y ) {
   return ( x + y );
}
handler h2 int() & channel c2( int y )
                 & channel c3( int z ) {
   return ( y + z );

}
```

Таким образом, в данном примере, если имеются значения в каналах *c1* и *c2*, то возможно срабатывание обработчика *h1*. Аналогичная ситуация имеет место для каналов *c2*, *c3* и обработчика *h2*. В общем случае, это может привести к нетерминизму в поведении программы.

Наконец, возможен случай, когда один и тот же обработчик объявлен в разных связках и с различными каналами:

```
handler h String() & channel c1( int x )
{
   return ( x.ToString() );
}

handler h String() & channel c2( String s )
{
   return ( s );
}
```

Схематически, совокупность таких определений может быть представлена как на Рис. 6:

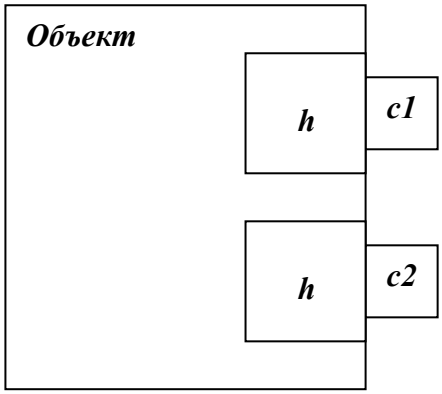

Рис. 6. Объект с одним обработчиком для нескольких несвязанных между собой каналов

В этом случае, вызов обработчика при наличии значения хотя бы в одном из каналов, приведет к срабатыванию соответствующей связки. Легко заметить, что и здесь наличие значения более, чем в одном канале, может стать источником недетерминизма в поведении программы.

При использовании связок в языке MC# нужно руководствоваться следующими правилами их корректного определения:

1. Формальные параметры каналов и обработчиков не могут содержать модификаторов `ref` или `out`.
2. Если в связке объявлен обработчик с типом возвращаемого значения `return-type`, то в теле связки должны использоваться операторы ***return*** только с выражениями, имеющими тип `return-type`.
3. Все формальные параметры каналов и обработчика в связке должны иметь различные идентификаторы.
4. Каналы и обработчики в связке не могут быть объявлены как `static`.

Вторая ключевая особенность языка MC# состоит в том, что каналы и обработчики могут передаваться в качестве аргументов методам (в том числе, *async*- и *movable*-методам) отдельно от объектов, которым они принадлежат (в этом смысле, они похожи на указатели на функции в языке C, или, в терминах языка C#, на *делегатов* (***delegates***)).

Третья ключевая особенность языка MC# состоит в том, что, в распределенном режиме, при копировании каналов и обработчиков на удаленную машину (под которой понимается узел кластера или некоторая машина в Grid-сети) автономно или в составе некоторого объекта, они становятся прокси-объектами, или посредниками для оригинальных каналов и обработчиков. Такая подмена скрыта от программиста – он может использовать переданные каналы и обработчики (а, в действительности, их прокси-объекты) на удаленной машине (т.е., внутри movable-методов) также, как и оригинальные: как обычно, все действия с прокси-объектами перенаправляются Runtime-системой на исходные каналы и обработчики. В этом отношении, каналы и обработчики отличаются от обычных объектов: манипуляции над последними на удаленной машине не переносятся на исходные объекты (см. первую ключевую особенность языка MC#).

На Рис. 7 и 8 схематически демонстрируются передача и использование каналов и обработчиков на удаленной машине. Верхние индексы *y* у имен каналов и обработчиков обозначают исходную машину, где они были созданы.

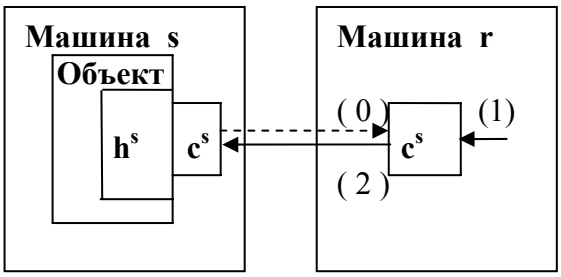

Рис. 7. Посылка сообщения по удаленному каналу:
(0) копирование канала на удаленную машину,
(1) посылка сообщения по (удаленному) каналу,
(2) перенаправление сообщения на исходную машину**.**

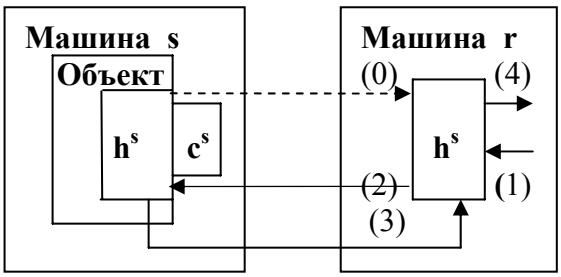

Рис. 8. Чтение сообщения из удаленного обработчика:
(0) копирование обработчика на удаленную машину,
(1) чтение сообщения из (удаленного) обработчика,
(2) перенаправление операции чтения на исходную машину,
(3) получение прочитанного сообщения с исходной машины,
(4) возврат полученного сообщения.

Этих средств и механизмов оказывается достаточно для организации взаимодействия произвольной сложности между параллельными, распределенными процессами, что демонстрируется в примерах следующего раздела.

3. Программирование на языке MC#

В этом разделе, использование специфических конструкций языка MC# будет проиллюстрировано на ряде параллельных и распределенных программ. Также излагаются и иллюстрируются общие принципы построения MC#-программ для нескольких типичных задач параллельного программирования. Подчеркиваются различия при разработке параллельных программ, предназначенных для исполнения на системах с общей памятью (например, мультиядерных процессорах), и распределенных программ, исполняющихся на сети машин (вычислительном кластере).

3.1 Вычисление чисел Фибоначчи

Последовательность чисел Фибоначчи есть бесконечный ряд из натуральных чисел

$$a_0,\ a_1,\ a_2,\ a_3,\ \ldots$$

таких, что

$$a_0 = 1,$$
$$a_1 = 1, \text{ и}$$
$$a_i = a_{i-1} + a_{i-2}, \quad \text{для } i \geq 2.$$

Построим параллельную программу, находящую n-ое ($n \geq 0$) число в ряду Фибоначчи, т.е., элемент a_n последовательности.

Первый вариант такой программы будет иметь рекурсивную структуру, соответствующую формуле определения чисел Фибоначчи. Основной вычислительный метод этой программы будет объявлен асинхронным, и будет возвращать вычисленный результат по каналу, переданному ему в качестве одного из входных аргументов. С другой стороны, в рекурсивных вызовах внутри этого метода будут использоваться каналы для получения значений от методов, вызванных рекурсивно.

Класс `Fib`, содержащий основной вычислительный метод Compute, может иметь следующий вид:

```
class Fib {

 public async Compute( int x, channel (int) sendResult ) {
   if ( n <= 1 )
      sendResult ( 1 );
   else {
      new Fib().Compute( n - 1, ic1 );
      new Fib().Compute( n - 2, ic2 );
      sendResult ( Get() );
   }
 }

 handler Get int() & channel ic1( int x )
                   & channel ic2( int y )
 {
```

```
    return x + y;
  }
}
```

Главный класс для этой программы может выглядеть следующим образом:

```
public class MainFib {
  public static void Main( String[] args ) {
    int n = System.Convert.ToInt32( args [0] ); // n есть номер
                          // искомого числа Фибоначчи ( n >= 1 )
    MainFib mfib = new MainFib();  // Создание объекта
                         // необходимо для создания его каналов
                         // и обработчиков
    Fib fib = new Fib();
    fib.Compute( n, mfib.senResult );
    Console.WriteLine( "For n = " + n + " value is " +
                       mfib.Get() );
  }

  handler Get int() & channel sendResult( int x )
  {
   return x;
  }

}
```

Упражнение 1. *Показать, почему при рекурсивных вызовах функции Compute необходимо создание новых объектов класса Fib.*
(Подсказка: рассмотреть как будут использоваться каналы ic1 и ic2 в программе, в которой опущено создание таких объектов, например, при вызове функции Compute с n = 3).

Легко видеть, что приведенный вариант параллельной программы является очень неэффективным, поскольку в нем порождается 2^n асинхронных вызовов метода Compute, каждый из которых выполняет очень мало вычислительных операций: фактически, порождает два дополнительных вызова и передает результат по каналу. Очевидно, что в этом случае эффект от параллельного исполнения методов будет перекрыт накладными расходами на их порождение.

Избежать порождения асинхронных вызовов функции Compute для малых по величине аргументов *n* можно путем введения "локального" вычисления соответствующей функции для малых n:

```
class Fib {

  public static int threshold = 30;

  public async Compute( int n, channel (int) sendResult ) {
    if ( n < threshold )
       sendResult ( cfib( n ) );
    else {
       new Fib().Compute( n - 1, ic1 );
       new Fib().Compute( n - 2, ic2 );
```

```
        sendResult( Get() );
      }
    }

    handler Get int() & channel ic1( int x )
                      & channel ic2( int y )
    {
     return ( x + y );
    }

    private int cfib( int n ) {
      if ( n <= 1 )
        return ( 1 );
      else
        return ( cfib( n - 1 ) + cfib( n - 2 ) );
    }
  }
```

Приведенный выше вариант является более эффективным, чем первый, но и он обладает существенным недостатком: теперь `async`-методы для больших *n* выполняют очень мало вычислительных операций.

Сократить общее количество порождаемых рекурсивно `async`-методов и более равномерно распределить по ним вычислительную нагрузку позволяет следующий, “линейный” вариант программы `Fib`:

```
  class Fib {
    public static int threshold = 30;
    public async Compute( int n, channel (int) sendResult ) {
      if ( n < threshold )
         sendResult( cfib( n ) );
      else {
         new Fib().Compute( n - 1, c );
         sendResult( cfib( n - 2 ) + Get() );
      }
    }

    handler Get int () & channel c( int x )
    {
     return ( x );
    }

    private int cfib( int n ) { ... }
  }
```

Приведенные выше рассуждения и варианты программы `Fib` переносятся и на варианты этой же программы, но для распределенного исполнения (т.е., с заменой `async` на `movable`). При этом, для получения максимального ускорения программист должен подобрать оптимальное значение константы threshold.

Ниже приведены графики времени выполнения последовательной программы и распределенного, “линейного” варианта программы `Fib` с threshold = 36. Причем количество используемых процессоров в распределенном варианте определялось как n – 34

(для n >= 35). Тестовые замеры проводились на кластере с процессорами AMD Athlon(TM) MP 1800+.

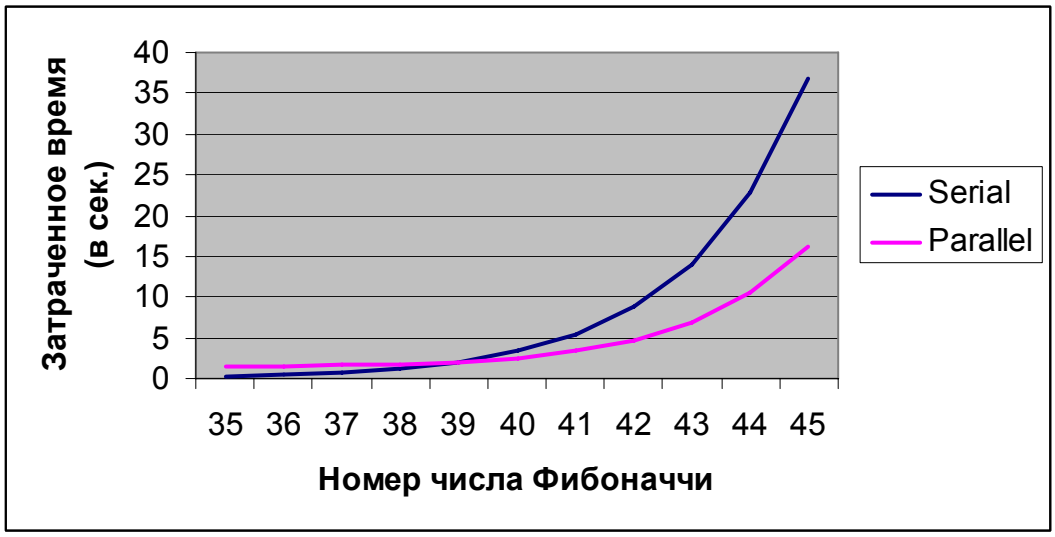

Рис. 9. Время вычисления N-го числа Фибоначчи “линейным” алгоритмом.

Полные тексты вариантов программы `Fib` приведёны в приложении **[A1].**

3.2 Обход бинарного дерева

Если структура данных задачи организована в виде дерева, то его обработку легко распараллелить путем обработки каждого поддерева отдельном `async-` (`movable-`) методом.

Предположим, что мы имеем следующее определение (в действительности, сбалансированного) бинарного дерева в виде класса `BinTree`:

```
class BinTree {

  public BinTree left;
  public BinTree right;

  public int value;

  public BinTree( int depth ) {
    value = 1;
    if ( depth <= 1 ) {
     left = null;
     right = null;
    }
    else {
      left = new BinTree( depth - 1 );
      right = new BinTree( depth - 1 );
    }
  }
```

```
}
```

Тогда просуммировать значения, находящиеся в узлах такого дерева (и, в общем случае, произвести более сложную обработку) можно с помощью следующей программы, структура которой, по существу, совпадает со структурой предыдущей программы `Fib`:

```
public class SumBinTree {
  public static void Main( String[] args ) {

    int depth = System.Convert.ToInt32( args [0] );

    SumBinTree sbt = new SumBinTree();
    BinTree btree = new BinTree( depth );

    sbt.Sum( btree, sbt.c );

    Console.WriteLine( "Sum = " + sbt.Get() );
  }

  // Определение канала и обработчика

  handler Get int () & channel c( int x )
  {
   return ( x );
  }

  // Определение async-метода

  public async Sum( BinTree btree, channel (int) c ) {

    if ( btree.left == null )  // Дерево есть лист
      c ( btree.value );
    else {
      new SumBinTree().Sum( btree.left,  c1 );
      new SumBinTree().Sum( btree.right, c2 );
      c( Get2() );
    }
  }

  // Определение связки  из двух каналов и обработчика

  handler Get2 int() & channel c1( int x )
                     & channel c2( int y )
  {
    return ( x + y );
  }
}
```

Естественно, что для повышения эффективности этой программы, а именно, для получения ускорения при исполнении ее на параллельной архитектуре, необходимо внести в нее усовершенствования, аналогичные тем, которые были сделаны для программы `Fib`.

Следует также отметить, что в случае распределенного варианта этой программы, при вызове `movable`-метода `Sum`, к объекту класса `BinTree`, являющемуся аргументом этого

метода, будут применяться процедуры сериализации/десериализации при переносе вычислений на другой компьютер. (В действительности, с точки зрения Runtime-языка MC#, поддерживающей распределенное исполнение программ, канал также является обычным объектом, к которому будут применяться процедуры сериализации/десериализации).

Полный текст данной программы приведён в приложении **[A2]**.

Упражнение 2. *Предположим, что имеется класс Tree, внутренними полями которого являются поле value, хранящее значение, связанное с корнем данного дерева, и поле subtrees, являющееся массивом объектов класса Tree.*

Написать параллельную программу на MC#, суммирующую значения из всех вершин заданного дерева.

(Подсказка: для получения значений от рекурсивно порождаемых методов, воспользуйтесь одним каналом).

3.3 Быстрое преобразование Фурье

Напомним коротко некоторые понятия и определения, относящиеся к дискретному преобразованию Фурье и быстрым алгоритмам его вычисления. Более подробно об этом см., например, в главе 32.2 “Дискретное преобразование Фурье. Быстрый алгоритм” книги [**L6**].

Комплексное число

$$\omega_n = e^{2\pi i / n}$$

называется *главным значением* корня степени *n* из единицы.

Вектор $y=(y_0, y_1, \dots, y_{n-1})$ называется *дискретным преобразованием Фурье* вектора $a=(a_0, a_1, \dots, a_{n-1})$, где a_i и y_j $(0 \le i, j \le n)$ есть комплексные числа, если

$$y_k = \sum_{0 \le j \le n-1} a_j \omega_n^{kj}$$

для $k = 0, 1, \dots, n-1$.

Быстрое преобразование Фурье (БПФ) представляет собой метод быстрого вычисления дискретного преобразования Фурье, использующий свойства комплексных корней из единицы и требующий времени $O(n \log n)$, в отличие от времени $O(n^2)$ при прямом вычислении по формуле.

В случае, когда n есть степень двойки, имеется следующий алгоритм быстрого преобразования Фурье вектора $a=(a_0, a_1, \dots, a_{n-1})$:

```
Recursive-FFT ( a )

  n = length [ a ];

  if ( n == 1 )
   then return ( a );

  ω_n = e^(2πi / n)
  ω = 1;
```

```
    a[0] = (a0, a2, … , an-2);
    a[1] = (a1, a3, … , an-1);

    y[0] = Recursive-FFT ( a[0] );
    y[1] = Recursive-FFT ( a[1] );

    for ( k = 0; k < n/2; k++ ) {
     yk = yk[0] + ω*yk[1];
     yk+(n/2) = yk[0] – ω*yk[1];
     ω = ω * ωn;
    }

    return ( y );
```

Легко видеть, что используемый в данном алгоритме двоичный принцип “разделяй и властвуй” аналогичен принципам, реализованным в предыдущих программах вычисления чисел Фибоначчи и обхода двоичного дерева.

Ниже приведен текст на языке MC# асинхронного метода, реализующего приведенный выше рекурсивный алгоритм. Полный текст этой программы можно найти в приложении **[A3]**.

```
public async FFT_Async ( int N, Complex[] a, Complex[] y, Channel
() sendStop )   {

  Recursive_FFT ( N, a, y );

  sendStop();

}

public void Recursive_FFT ( int N, Complex[] a, Complex[] y )   {

  int   i;

  if ( N == 1 )   {
   y [ 0 ] = a [ 0 ];
   return;
  }

  int   N2 = N / 2;

  Complex[] a0 = new Complex [ N2 ];
  Complex[] a1 = new Complex [ N2 ];
  Complex[] y0 = new Complex [ N2 ];
  Complex[] y1 = new Complex [ N2 ];

  for ( i = 0; i < N; i += 2 )   {

   a0 [ i / 2 ] = a [ i ];
   a1 [ i / 2 ] = a [ i + 1 ];
   y0 [ i / 2 ] = new Complex();
   y1 [ i / 2 ] = new Complex();
```

```
  }

  Recursive_FFT ( N2, a0, y0 );
  Recursive_FFT ( N2, a1, y1 );

  merge ( N, y0, y1, y );

 }

public void merge (int N, Complex[]y0, Complex[] y1, Complex[] y )
{

  double   wy_Re, wy_Im, tmp;
  double   w_Re, w_Im, wn_Re, wn_Im;

  int      i;

  wn_Re    = Math.Cos ( 2.0 * Math.PI / N );
  wn_Im    = Math.Sin ( 2.0 * Math.PI / N );

  w_Re     = 1.0;
  w_Im     = 0.0;

  int      N2 = N / 2;

  for ( i = 0; i < N2; i++ )   {

   wy_Re = w_Re * y1 [ i ].Re - w_Im * y1 [ i ].Im;
   wy_Im = w_Re * y1 [ i ].Im + w_Im * y1 [ i ].Re;

   y [ i ].Re = y0 [ i ].Re + wy_Re;
   y [ i ].Im = y0 [ i ].Im + wy_Im;

   y [ i + N2 ].Re = y0 [ i ].Re - wy_Re;
   y [ i + N2 ].Im = y0 [ i ].Im - wy_Im;

   tmp  = w_Re * wn_Re - w_Im * wn_Im;
   w_Im = w_Re * wn_Im + w_Im * wn_Re;
   w_Re = tmp;

  }

}
```

Однако, данная программа, даже с оптимизациями, аналогичными рассмотренным для программ `Fib` и `SumBinTree`, не дает ускорения при исполнении на параллельной архитектуре из-за больших объемов требуемой памяти (в 2 раза бóльших, чем для последовательного варианта), необходимой для размещения массивов $a^{[0]}$, $a^{[1]}$, $y^{[0]}$, $y^{[1]}$ при рекурсивных вызовах.

Существуют более эффективные алгоритмы БПФ, например, итеративный алгоритм из главы 32.3 “Эффективные реализации быстрого преобразования Фурье” из книги [**L6**]. Однако, данный алгоритм естественным образом параллелится только на 2 процессора. Реализацию такого алгоритма на языке MC# см. в приложении **[A3]**.

Ниже представлены графики времени исполнения последовательного и итеративного параллельного (на 2 процессора) алгоритмов БПФ, реализованных на языке MC#.

Тестовые замеры проводились на машине с двухядерным процессором Intel Core 2 Duo 2.4 GHz и оперативной памятью 1 Гб.

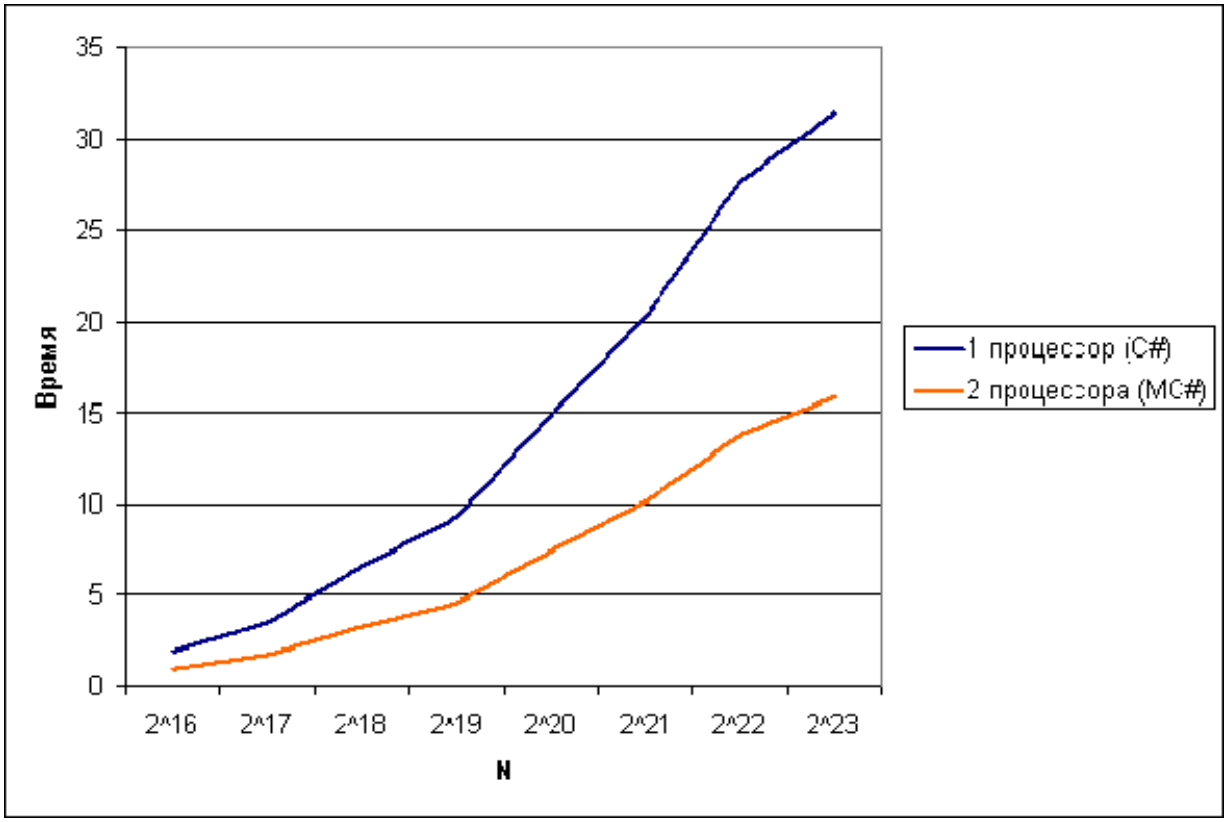

Рис. 10. Время исполнения последовательного и параллельного алгоритмов БПФ.

3.4 Построения списка простых чисел методом “решета Эратосфена”

В данном разделе будет представлена параллельная программа построения списка простых чисел методом просеивания (другое название этого метода – “решето Эратосфена”).

3.4.1 Наивный алгоритм

По условию задачи, по заданному натуральному числу $N \geq 2$, необходимо найти все простые числа в интервале от 2 до N.

Метод просеивания состоит из следующих шагов:

1) из исходного списка l_0 всех натуральных чисел от 2 до N

$$l_0 = [2, \dots, N]$$

выбирается первое число этого списка, а именно 2, и выдается в качестве первого выходного значения;

2) затем строится новый список l_1 , который получается из списка l_0 вычеркиванием из него всех чисел, кратных очередному выбранному простому числу – на первом шаге, числу 2:

$$l_1 = [3, 5, 7, \dots , N] \quad \text{(в предположении, что N нечетно)}$$

3) затем данная процедура повторяется рекурсивно для вновь построенного списка.

Алгоритм заканчивает свою работу, когда очередной список оказывается пустым.

В соответствии со сказанным выше, основная вычислительная процедура программы (метод `Sieve`), тем самым, должна производить следующие действия:

1) переслать первый элемент из входного потока натуральных чисел (если он не пуст) в выходной поток;
2) отфильтровать все оставшиеся числа их входного потока по модулю первого элемента и направить этот отфильтрованный поток на вход новой рекурсивно вызванной процедуре `Sieve`; выходным потоком для рекурсивно вызванной процедуры становится исходный выходной поток.

Выходным же каналом для этой новой процедуры будет исходный выходной канал.

Графически, один шаг рекурсивного развертывания программы может быть представлен следующим образом :

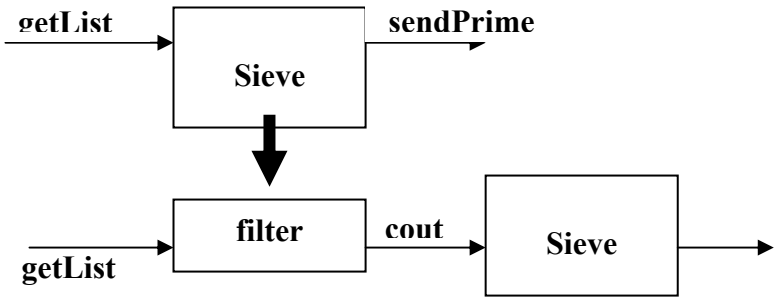

Рис. 12. Шаг рекурсивного развертывания программы.

Рекурсивные вызовы метода `Sieve` сопровождаются созданием соответствующих объектов (класса `CSieve`), имеющих каналы и обработчики. Эти каналы и обработчики используются для создания цепочки обрабатывающих элементов, с помощью которых производится просеивание потока натуральных чисел.

Программа состоит из следующих методов:

`async Sieve(handler int() getList, channel (int) sendPrime)` — async-метод класса `CSieve`, который реализует собственно алгоритм просеивания: из обработчика `getList` предыдущего объекта цепочки читается поток чисел, фильтруется по модулю первого элемента из потока, и выбранные простые числа посылаются в канал `sendPrime` (через дальнейшие рекурсивные вызовы метода `Sieve`);

`void filter (int p, handler int()getList, channel (int) cfiltered)` — функция класса `CSieve`, реализующая фильтрацию потока натуральных чисел: из потока чисел, получаемых путем вызова обработчика `getList`, удаляются все элементы, которые делятся на число `p`, и посылаются по каналу `cfiltered`;

`public static void Main( String[] args )`— главная функция из класса `Eratosthenes`, которая создает поток натуральных чисел `Nats`, и, по мере получения простых чисел из обработчика `getPrime`, выводит их на консоль.

Полный текст программы приведен ниже, а также в приложении **[A4]**.

```
class Eratosthenes   {
 public static void Main(String[] args) {
  int N = System.Convert.ToInt32 (args[0] );
  Eratosthenes E = new  Eratosthenes();
  new CSieve().Sieve ( E.getNat, E.sendPrime );
  for ( int n=2; n <= N; n++ )
    E.Nats ( n );
  E.Nats ( -1 );
  while  (  ( int p = E.getPrime() ) != -1 )
   Console.WriteLine ( p );
 }
 handler getNat  int()  & channel Nats ( int n ) {
    return ( n );
 }
  handler getPrime int()& channel sendPrime( int p ) {
     return ( n );
  }
}

class  CSieve  {
 async Sieve ( handler int() getList, channel (int) sendPrime ) {
  int  p = getList();
  sendPrime ( p );
  if  ( p != -1 )   {
   new CSieve().Sieve ( hin, sendPrime );
   filter ( p, getList, cout );
  }
 }
 handler hin int()  &  channel cout ( int x ) {
  return  ( x );
 }
void filter (int p, handler int() getList,
                    channel (int) cfiltered )   {
  while ( ( int n = getList() ) != -1 )
    if ( n % p != 0 ) cfiltered ( n );
  cfiltered ( -1 );
}
}
```

Как обычно, распределенный вариант программы, который может исполняться на сети машин, получается заменой ключевого слова `async` на `movable`.

В принципе, при исполнении программы на одной машине, элементы цепочки, с помощью которой производится просеивание, могут быть построены из стандартных объектов .NET, например, из очередей (объектов класса `Queue`). Однако, в этом случае, программист должен позаботиться об их блокировке, что необходимо в случае

одновременной работы множества синхронных методов (потоков). Использование каналов и обработчиков языка MC# освобождает программиста от этой обязанности.

Аналогично ранее рассмотренным примерам, метод `Sieve` выполняет слишком мало вычислительных операций, чтобы обеспечить эффективность всей параллельной (распределенной) программы в целом.

3.4.2 Пакетный алгоритм

В данном разделе описывается модификация наивного алгоритма “решето Эратосфена”, существенно улучшающая эффективность параллельной (распределенной) программы, и которая дает существенное ускорение при поиске простых чисел в длинных интервалах (например, для $N \geq 10^6$).

Основная идея предлагаемого алгоритма заключается в переходе от использования потоков одиночных данных (потоков отдельных натуральных чисел) к потокам пакетов натуральных чисел.

Пакет – это массив натуральных чисел фиксированного размера (задаваемого в программе значением PACKAGE_SIZE), пустые хвостовые элементы которого заполняются нулями.

При этом, функции наивного алгоритма обобщаются в данном варианте естественным образом:

`async Sieve(handler int[]() getNatPack, channel (int[]) sendPrimesPack)` — с помощью обработчика `getNatPack` получает первый пакет из потока, обрабатывает его функцией `SynchronousSieve`, получая пакет простых чмсел head, и отправляя его по каналу `sendPrimesPack`; остальные пакеты из входного потока фильтруются с помощью функции `filter` по модулю пакета `head` и направляются на вход следующей в цепочке рекурсивной функции `Sieve`;

`void filter(int[] head, handler int[] () getNatPack, channel (int[]) cfiltered)` — функция, которая фильтрует пакеты из входного потока, получаемые с помощью обработчика `getNatPack`, по модулю пакета простых чисел `head`, отправляя результирующие пакеты в канал `cfiltered`; при этом, все результирующие пакеты (кроме, может быть, последнего) имеют длину PACKAGE_SIZE (строго говоря, и последний пакет имеет длину PACKAGE_SIZE, но его хвостовая часть может быть заполнена нулями).

Полный текст программы Eratosthenes2 приведен ниже, а также в приложении **[A4]**.

```
using System;
using System.Text;

public class Config {

   public static int N = 1000000;
   public static int MAX_LEN = 50000;

   public static void print(int[] a) {
    StringBuilder sb = new StringBuilder();
    sb.Append("-----------------------------------------\n");
    for(int i = 0; i<a.Length && a[i]!=0; i++) sb.Append(a[i]+" ");
    Console.WriteLine(sb.ToString());
   }
```

```
   }
public class CSieve
{
    //   Функция,   реализующая   стандартный   алгоритм   "Решето
//Эратосфена"
    private int[] SynchronousSieve(int[] ar)
    {
        if (ar == null || ar.Length == 0) return new int [ 0 ];
        int[]  primes  =  (int[])  Array.CreateInstance(typeof(int),
Config.MAX_LEN);

        int ind = 0;

        primes[0] = ar[0];

        for(int i = 1; i < ar.Length; i++)
        {
            if(isPrime(ar[i],primes)) primes[++ind] = ar[i];
        }

        return primes;
    }

    // Функция,проверяющая число на простоту
    private bool isPrime(int n, int[]primes)
    {
        bool isPrime = true;
        for(int   j   =   0;   isPrime   &&   j   <   primes.Length   &&
primes[j]!=0 && primes[j]*primes[j] <= n; j++)
isPrime = (n % primes[j] != 0);

        return isPrime;
    }

    async   Sieve(handler   int[]()   getNatPack,   channel   (int[])
sendPrimesPack)
    {
        // Получаем первый пакет из входного потока и
        // извлекаем из него подпакет простых чисел
        int[] head = SynchronousSieve((int[])getNatPack());

        // Посылаем пакет простых чисел в выходной поток
        sendPrimesPack( head );

        // Фильтруем оставшиеся пакеты входного потока
        // относительно пакета head
        if ( head.Length != 0 )
        {
            new CSieve().Sieve( inter, sendPrimesPack);
            filter ( head, getNatPack, cout );
        }
```

```
    }
    handler inter int[]()  &  channel cout ( int[] p) {
        return  ( x );
    }
    // Фильтрация потоков
    void filter(int[] head, handler int[] () getNatPack, channel
(int[]) cfiltered)
    {
     int[]                              al                              =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

        int ind = 0;

        // Для каждого пакета из входного потока
        for( int[] p; (p = (int[])getNatPack() ).Length != 0;)
        {
            // Выбираем простые числа, формируя новый пакет
            for(int i = 0; i < p.Length && p[i]!=0; i++)
            {
                if(isPrime(p[i],head))
                {
                    al[ind++] = p[i];

                    // Если пакет заполнен, то посылаем его
                    // в поток отфильтрованных пакетов
                    if(ind == Config.MAX_LEN)
                    {
                        ind = 0;
                        cfiltered (al);
                        al                                              =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

                    }
                }
            }
        }

        // Посылаем последний пакет в поток
        // отфильтрованных пакетов
        if(al[0] != 0) cfiltered (al);

        // Посылаем маркер конца потока пакетов
        cfiltered( new int [ 0 ] );
    }
class Eratosthenes2   {
 public static void Main(String[] args) {

   Eratosthenes2 er2 = new  Eratosthenes2();
   CSieve csieve = new CSieve();

   // Запускаем метод Sieve

   csieve.Sieve( er2.getNatPack, er2.sendPrimePack);
```

```
   int[]                              al                                  =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

        // Создаем поток пакетов натуральных чисел
        for (int i = 2; i <= Config.N; i++)
        {
            int ind = (i-2)%Config.MAX_LEN;

            al[ind] = i;
            if(ind == Config.MAX_LEN - 1)
            {
                er2.Nats ( al );

                al                                                          =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

            }
        }

        if(al[0] != 0)
         er2.Nats(al);
        er2.Nats ( new int [ 0 ] );

        int[] p;
        // Распечатываем результирующие пакеты простых чисел
        while (( p = (int[])er2.getPrimePack()).Length != 0)
         Config.print(p);
 }

 handler getNatPack  int[]()  & channel Nats ( int[] p ) {
  return ( p );
 }
 handler getPrimePack int[]() & channel sendPrimePack( int[] p ) {
  return ( p );
 }

}
```

3.5 Программа *all2all*

Программа `all2all` предназначена для демонстрации способа, с помощью которого можно обеспечить взаимодействие внутри множества асинхронных (распределенных) процессов в соответствии с принципом “все со всеми”. В определенном смысле, эта программа показывает, как можно реализовать на языке MC# глобальные (в терминах MPI, “широковещательные”(broadcast)) операции передачи данных. Данный подход часто используется в программах, реализующих параллелизм по данным, когда отдельные процессы (процессоры) должны обмениваться сообщениями как со своими соседями, так и со всеми процессами, участвующими в вычислениях.

Ниже будет представлен вариант программы all2all с распределенными (movable-) процессами.

В этой программе, каждый распределенный процесс представляется методом Start объекта класса DistribProcess. Для взаимодействия между собой, каждый распределенный процесс создает объект класса BDChannel (Bidirectional channel), содержащий канал Send и обработчик Receive:

```
class   BDChannel   {
 handler  Receive object()
             & channel Send ( object obj )  {
   return  ( obj );
 }
}
```

Обменявшись между собой такими объектами, распределенные процессы могут посылать и принимать сообщения друг от друга независимо от их физического расположения. Обмен объектами класса BDChannel реализуется через главный процесс, который исполняется на машине, где исходно было запущено распределенное приложение:

```
class  All2all   {
 public static void Main (String[] args)   {

  int  i;

  // N есть число распределенных процессов
  int  N = System.Convert.ToInt32 ( args [ 0 ] );

 All2all a2a = new All2all();
  DistribProcess dproc = new DistribProcess();

  //   Запуск распределенных процессов
  for ( i = 0; i < N; i++ )
    dproc.Start ( i, a2a.sendBDC, a2a,sendStop );

  //   Получение объектов класса BDChannel от процессов
  BDChannel[] bdchans = new BDChannel [ N ];
  for ( i = 0; i < N; i++ )
    a2a.getBDC ( bdchans );

  //   Рассылка массива объектов класса BDChannel
  //   каждому из процессов
  for ( i = 0; i < N; i++ )
     bdchans [ i ].Send ( bdchans );

  //   Получение сигналов обокончании работы процессов
  for ( i = 0; i < N; i++ )
    a2a.getStop();
 }
 handler getBDC void( BDChannel[] bdchans)  &
       channel sendBDC ( int i, BDChannel bdc )      {
    bdchans [ i ] = bdc;
```

```
 }
 handler  getStop void() & channel  sendStop() {
   return;
 }
}
```

Каждый распределенный процесс (объект класса `DistribProcess`) выполняет в данной программе следующую последовательность действий:

1. Создает свой собственный объект класса `BDChannel` и отсылает его главному процессу.
2. Принимает от главного процесса массив объектов класса `BDChannel` всех остальных процессов (включая свой собственный).
3. Пользуясь этим массивом, посылает сообщение всем остальным процессам в группе.
4. Принимает сообщения от всех процессов в группе.
5. Посылает сигнал об окончании своей работы главному процессу.

```
class DistribProcess   {
 movable Start ( int  myNumber, channel  (int, BDChannel)
                              sendBDC, channel () sendStop ) {
  //  i есть собственный номер процесса
  int   i;
  BDChannel  bdc = new BDChannel();
  sendBDC ( myNumber, bdc );
  BDChannel[]   bdchans  =  (BDChannel[]) bdc.Receive();

  //   Посылка сообщений другим процессам
  for  ( i = 0; i < bdchans.Length; i++ )
   if  ( i != myNumber )
     bdchans[j].Send ("Message from process " + myNumber +
                                  " to process " + i       );

  //   Прием сообщений от других процессов
  //   (здесь можно было бы воспользоваться и каналом
  //    bdchans [ myNumber ])
  for  ( i = 0; i < bdchans.Length - 1; i++ )
    Console.WriteLine ( "Process " + myNumber + " : " +
                        (String) bdc.Receive() );

  //   Посылка сигнала об окнчании работы
  sendStop();
 }
}
```

Полный текст программы `All2all` приведен в приложении **[A5]**.

Литература

[L1]	N. Benton, L. Cardelli, C. Fournet “Modern concurrency abstractions for C#”, - ACM Transactions on Programming Languages and Systems, vol. 26, N 5, 2004, pp.769-804.
[L2]	Fournet, C., Gonthier, G. “The join calculus: a language for distributed mobile programming”, - in Proceedings of Applied Semantics Summer School, 2000. Lecture Notes in Computer Science, Vol.2395, Springer, pp. 268-332.
[L3]	Bierman, G., Meijer, E., Schulte, W. “The essence of data access in Cω”, - ECOOP 2005, Lecture Notes in Computer Science, Vol. 3586, Springer, 2005. pp. 287-311.
[L4]	Saraswat, V., Jagadeesan, R. “Concurrent Clustered Programming”, - CONCUR 2005, Lecture Notes in Computer Science, Vol. 3653. Springer-Verlag, Berlin Heidelberg New York, 2005, pp. 353-367
[L5]	Charles, P., Grothoff, C., Donawa, C., Ebciouglu, K., Kielstra, A., von Praun, C., Saraswat, V., Sarkar, V. “X10: An Object-oriented Approach to Non-uniform Cluster Computing”, - Proc. ACM 2005 OOPSLA Conf., Onward! Track, October 2005, pp. 519-538
[L6]	Кормен Т., Лейзерсон Ч., Ривест Р. “Алгоритмы: построение и анализ”, - Москва, Московский Центр Непрерывного Математического Образования, 1999.

Приложение

[A1]	ComputeFib	Программы вычисления чисел Фибоначчи: а) базовый алгоритм, б) алгоритм с локальной функцией cfib, в) “линейный” алгоритм.
[A2]	SumBinTree	Программа обхода (суммирования значений в узлах) бинарного дерева
[A3]	FFT	Программа быстрого преобразования Фурье: а) рекурсивный алгоритм, б) итеративный алгоритм.
[A4]	Eratosthenes	Вычисление простых чисел методом просеивания (решето Эратосфена): а) наивный алгоритм, б) пакетный алгоритм.
[A5]	All2all	Программа, демонстрирующая взаимодействие группы параллельных, распределенных процессов в соответствии с принципом “каждый с каждым”

[A1] ComputeFib

а) Базовый алгоритм

```
   class Fib {

    public async Compute( int x, channel (int) sendResult ) {
      if ( n <= 1 )
         sendResult ( 1 );
      else {
         new Fib().Compute( n - 1, ic1 );
         new Fib().Compute( n - 2, ic2 );
         sendResult ( Get() );
      }
    }

    handler Get int() & channel ic1( int x )
                      & channel ic2( int y )
    {

      return x + y;
    }
}

    public class MainFib {
     public static void Main( String[] args ) {
       int n = System.Convert.ToInt32( args [0] ); // n есть номер
                              // искомого числа Фибоначчи ( n >= 1 )
       MainFib mfib = new MainFib();  // Создание объекта
                             // необходимо для создания его каналов
                             // и обработчиков
       Fib fib = new Fib();
       fib.Compute( n, mfib.senResult );
       Console.WriteLine( "For n = " + n + " value is " +
                           mfib.Get() );
    }

    handler Get int() & channel sendResult( int x )
    {
     return x;
    }

}
```

б) Алгоритм с локальной функцией cfib

```
  class Fib {

   public static int threshold = 30;

   public async Compute( int n, channel (int) sendResult ) {
     if ( n < threshold )
        sendResult ( cfib( n ) );
     else {
        new Fib().Compute( n - 1, ic1 );
        new Fib().Compute( n - 2, ic2 );
        sendResult( Get() );
     }
   }

   handler Get int() & channel ic1( int x )
                     & channel ic2( int y )
   {
    return ( x + y );
   }

   private int cfib( int n ) {
     if ( n <= 1 )
       return ( 1 );
     else
       return ( cfib( n - 1 ) + cfib( n - 2 ) );
   }
}

public class MainFib {
 public static void Main( String[] args ) {
   int n = System.Convert.ToInt32( args [0] ); // n есть номер
                          // искомого числа Фибоначчи ( n >= 1 )
   MainFib mfib = new MainFib();  // Создание объекта
                         // необходимо для создания его каналов
                         // и обработчиков
   Fib fib = new Fib();
   fib.Compute( n, mfib.senResult );
   Console.WriteLine( "For n = " + n + " value is " +
                      mfib.Get() );
 }
 handler Get int() & channel sendResult( int x )
 {
  return x;
 }
}
```

в) “Линейный алгоритм”

```
class Fib {

   public static int threshold = 30;

   public async Compute( int n, channel (int) sendResult ) {
      if ( n < threshold )
         sendResult( cfib( n ) );
      else {
         new Fib().Compute( n - 1, c );
         sendResult( cfib( n - 2 ) + Get() );
      }
   }

    handler Get int () & channel c( int x )
    {
     return ( x );
    }
  private int cfib( int n ) {
   if ( n <= 1 )
     return ( 1 );
   else
     return ( cfib( n - 1 ) + cfib( n - 2 ) );
  }
}

public class MainFib {

  public static void Main( String[] args ) {

    int n = System.Convert.ToInt32( args [0] ); // n есть номер
                          // искомого числа Фибоначчи ( n >= 1 )
    MainFib mfib = new MainFib();  // Создание объекта
                         // необходимо для создания его каналов
                         // и обработчиков
    Fib fib = new Fib();
    fib.Compute( n, mfib.senResult );
    Console.WriteLine( "For n = " + n + " value is " +
                       mfib.Get() );
 }
 handler Get int() & channel sendResult( int x )
 {
  return x;
 }
}
```

[A2] SumBinTree

```
class BinTree {

    public BinTree left;
```

```
    public BinTree right;

    public int value;

    public BinTree( int depth ) {
      value = 1;
      if ( depth <= 1 ) {
       left = null;
       right = null;
      }
      else {
        left = new BinTree( depth - 1 );
        right = new BinTree( depth - 1 );
      }
    }
}

public class SumBinTree {

   public static void Main( String[] args ) {

      int depth = System.Convert.ToInt32( args [0] );

      SumBinTree sbt = new SumBinTree();
      BinTree btree = new BinTree( depth );

      sbt.Sum( btree, sbt.c );

      Console.WriteLine( "Sum = " + sbt.Get() );
   }

    // Определение канала и обработчика

    handler Get int () & channel c( int x )
    {
     return ( x );
    }

    // Определение async-метода

    public async Sum( BinTree btree, channel (int) c ) {

      if ( btree.left == null )  // Дерево есть лист
        c ( btree.value );
      else {
        new SumBinTree().Sum( btree.left,  c1 );
        new SumBinTree().Sum( btree.right, c2 );
        c( Get2() );
      }
    }

    // Определение связки  из двух каналов и обработчика
```

```
    handler Get2 int() & channel c1( int x )
                      & channel c2( int y )
    {
      return ( x + y );
    }
}
```

[A3] FFT

а) Рекурсивный алгоритм

```
using System;

public class Complex {

 public double Re = 0.0;
 public double Im = 0.0;

 public Complex () {}

 public Complex ( double re, double im )  {
  Re = re;
  Im = im;
 }

}

public class FFT   {

 public static void Main ( String[] args )   {

  int     i;

  int N = System.Convert.ToInt32 ( args [0] ); //  N is power of 2

  Random r = new Random();

  Complex[]  a = new Complex [ N ];
  Complex[]  y = new Complex [ N ];

  for ( i = 0; i < N; i++ )   {

    a [ i ] = new Complex ( r.NextDouble(), r.NextDouble() );
    y [ i ] = new Complex ();

  }

  FFT  fft = new FFT();

  DateTime dt1 = DateTime.Now;

  int   N2 = N / 2;

  Complex[] a0 = new Complex [ N2 ];
  Complex[] a1 = new Complex [ N2 ];
  Complex[] y0 = new Complex [ N2 ];
  Complex[] y1 = new Complex [ N2 ];

  for ( i = 0; i < N; i += 2 )   {
```

```
   a0 [ i / 2 ] = a [ i ];
   a1 [ i / 2 ] = a [ i + 1 ];
   y0 [ i / 2 ] = new Complex();
   y1 [ i / 2 ] = new Complex();

  }

  fft.FFT_Async ( N2, a0, y0, fft.sendStop );
  fft.FFT_Async ( N2, a1, y1, fft.sendStop );

  for ( i = 0; i < 2; i++ )
   fft.getStop();

  fft.merge ( N, y0, y1, y );

  DateTime dt2 = DateTime.Now;

  Console.WriteLine ( " N = " + N + "   Elapsed time is " + (dt2-
dt1).TotalSeconds );

 }

 handler getStop void()   &   channel sendStop()  {
  return;
 }

public async FFT_Async ( int N, Complex[] a, Complex[] y, channel
() sendStop )   {

  Recursive_FFT ( N, a, y );

  sendStop();

}

public void Recursive_FFT ( int N, Complex[] a, Complex[] y )   {

  int   i;

  if ( N == 1 )   {
   y [ 0 ] = a [ 0 ];
   return;
  }

  int   N2 = N / 2;

  Complex[] a0 = new Complex [ N2 ];
  Complex[] a1 = new Complex [ N2 ];
  Complex[] y0 = new Complex [ N2 ];
  Complex[] y1 = new Complex [ N2 ];

  for ( i = 0; i < N; i += 2 )   {
```

```
    a0 [ i / 2 ] = a [ i ];
    a1 [ i / 2 ] = a [ i + 1 ];
    y0 [ i / 2 ] = new Complex();
    y1 [ i / 2 ] = new Complex();

   }

   Recursive_FFT ( N2, a0, y0 );
   Recursive_FFT ( N2, a1, y1 );

   merge ( N, y0, y1, y );

  }

public void merge (int N, Complex[]y0, Complex[] y1, Complex[] y )
{

   double    wy_Re, wy_Im, tmp;
   double    w_Re, w_Im, wn_Re, wn_Im;

   int       i;

   wn_Re     = Math.Cos ( 2.0 * Math.PI / N );
   wn_Im     = Math.Sin ( 2.0 * Math.PI / N );

   w_Re      = 1.0;
   w_Im      = 0.0;

   int       N2 = N / 2;

   for ( i = 0; i < N2; i++ )    {

    wy_Re = w_Re * y1 [ i ].Re - w_Im * y1 [ i ].Im;
    wy_Im = w_Re * y1 [ i ].Im + w_Im * y1 [ i ].Re;

    y [ i ].Re = y0 [ i ].Re + wy_Re;
    y [ i ].Im = y0 [ i ].Im + wy_Im;

    y [ i + N2 ].Re = y0 [ i ].Re - wy_Re;
    y [ i + N2 ].Im = y0 [ i ].Im - wy_Im;

    tmp  = w_Re * wn_Re - w_Im * wn_Im;
    w_Im = w_Re * wn_Im + w_Im * wn_Re;
    w_Re = tmp;

   }

}
```

б) Итеративный алгоритм

```
using System;
public class Complex {
 public double Re = 0.0;
 public double Im = 0.0;
 public Complex () {}
 public Complex ( double re, double im )  {
  Re = re;
  Im = im;
 }
}
public class IFFT   {

 public static void Main ( String[] args )   {

  int N = System.Convert.ToInt32 ( args [ 0 ] ); //  N is power of
2

  Random r = new Random();

  Complex[]  a = new Complex [ N ];
  Complex[]  A = new Complex [ N ];

  for ( int i = 0; i < N; i++ )
   a [ i ] = new Complex ( r.NextDouble(), r.NextDouble() );

  DateTime dt1 = DateTime.Now;

  IFFT ifft = new IFFT();

  int logN = 0;
  int m    = N;

  while ( m > 1 )
  {
   m = m / 2;
   logN++;
  }

  for ( int k = 0; k < N; k++ )
   A [ ifft.bit_reverse ( k, logN ) ] = a [ k ];

  int N2 = N / 2;

  ifft.Iterative_FFT ( a, A, N2, logN, 0,  ifft.sendStop );
  ifft.Iterative_FFT ( a, A, N2, logN, N2, ifft.sendStop );

  for ( m = 0; m < 2; m++ )
   ifft.getStop();

  //   Final iteration

  double wn_Re, wn_Im, w_Re, w_Im;
```

```
  double arg, t_Re, t_Im;

  double u_Re, u_Im, tmp;

  int    jN2;

  arg = 2.0 * Math.PI / N;

  wn_Re = Math.Cos ( arg );
  wn_Im = Math.Sin ( arg );

  w_Re  = 1.0;
  w_Im  = 0.0;

  for ( int j = 0; j < N2; j++ )
  {
   jN2  = j + N2;

   t_Re = w_Re * A [ jN2 ].Re - w_Im * A [ jN2 ].Im;
   t_Im = w_Re * A [ jN2 ].Im + w_Im * A [ jN2 ].Re;

   u_Re = A [ j ].Re;
   u_Im = A [ j ].Im;

   A [ j ].Re = u_Re + t_Re;
   A [ j ].Im = u_Im + t_Im;

   A [ jN2 ].Re = u_Re - t_Re;
   A [ jN2 ].Im = u_Im - t_Im;

   tmp  = w_Re * wn_Re - w_Im * wn_Im;
   w_Im = w_Re * wn_Im + w_Im * wn_Re;
   w_Re = tmp;
  }

  DateTime dt2 = DateTime.Now;

  Console.WriteLine ( "N = " + N + "   Elapsed time is " + (dt2-
dt1).TotalSeconds );

 }

 handler getStop void()   &   channel sendStop()  {
  return;
 }

 public int bit_reverse ( int k, int size )   {

  int   right_unit = 1;
  int   left_unit  = 1 << ( size - 1 );

  int   result     = 0;
  int   bit;
```

```
  for ( int i = 0; i < size; i++ )
  {
   bit = k & right_unit;

   if ( bit != 0 )
    result = result | left_unit;

   right_unit = right_unit << 1;
   left_unit  = left_unit  >> 1;
  }

  return ( result );

 }

 public async Iterative_FFT ( Complex[] a, Complex[] A, int N2,
int logN, int shift, channel() sendStop                          )
 {

  int    j, k, m, m2, km2, s;

  double wn_Re, wn_Im, w_Re, w_Im;
  double arg, t_Re, t_Im;

  double u_Re, u_Im, tmp;

  for ( s = 1; s < logN; s++ )
  {
   m = 1 << s;

   arg = 2.0 * Math.PI / m;

   wn_Re = Math.Cos ( arg );
   wn_Im = Math.Sin ( arg );

   w_Re  = 1.0;
   w_Im  = 0.0;

   m2    = m >> 1;

   for ( j = 0; j < m2; j++ )

    for ( k = j + shift; k < N2; k += m )
    {
     km2  = k + m2;

     t_Re = w_Re * A [ km2 ].Re - w_Im * A [ km2 ].Im;
     t_Im = w_Re * A [ km2 ].Im + w_Im * A [ km2 ].Re;

     u_Re = A [ k ].Re;
     u_Im = A [ k ].Im;
```

```
      A [ k ].Re = u_Re + t_Re;
      A [ k ].Im = u_Im + t_Im;

      A [ km2 ].Re = u_Re - t_Re;
      A [ km2 ].Im = u_Im - t_Im;

      tmp  = w_Re * wn_Re - w_Im * wn_Im;
      w_Im = w_Re * wn_Im + w_Im * wn_Re;
      w_Re = tmp;
     }

   }

   sendStop ();

  }

}
```

[A4] Eratosthenes

а) Наивный алгоритм

```
class Eratosthenes   {

 public static void Main(String[] args) {
  int N = System.Convert.ToInt32 (args[0] );
  Eratosthenes E = new  Eratosthenes();
  new CSieve().Sieve ( E.getNat, E.sendPrime );
  for ( int n=2; n <= N; n++ )
    E.Nats ( n );
  E.Nats ( -1 );
  while  (  ( int p = E.getPrime() ) != -1 )
   Console.WriteLine ( p );
 }

 handler getNat  int()  & channel Nats ( int n ) {
    return ( n );
 }

  handler getPrime int()& channel sendPrime( int p ) {
     return ( n );
  }

}

class  CSieve  {

 async Sieve ( handler int() getList, channel (int) sendPrime ) {
  int  p = getList();
  sendPrime ( p );
  if  ( p != -1 )   {
   new CSieve().Sieve ( hin, sendPrime );
   filter ( p, getList, cout );
  }
 }

 handler hin int()  &  channel cout ( int x ) {
  return  ( x );
 }

void filter (int p, handler int() getList,
                    channel (int) cfiltered )   {
  while ( ( int n = getList() ) != -1 )
    if ( n % p != 0 ) cfiltered ( n );
  cfiltered ( -1 );
}

}
```

б) Пакетный алгоритм

```
using System;
using System.Text;
public class Config {

   public static int N = 1000000;
   public static int MAX_LEN = 50000;

   public static void print(int[] a) {
    StringBuilder sb = new StringBuilder();
    sb.Append("-----------------------------------------\n");
    for(int i = 0; i<a.Length && a[i]!=0; i++) sb.Append(a[i]+" ");
    Console.WriteLine(sb.ToString());
   }

   }
public class CSieve
{
    //   Функция,   реализующая   стандартный   алгоритм   "Решето
//Эратосфена"
    private int[] SynchronousSieve(int[] ar)
    {
        if (ar == null || ar.Length == 0) return new int [ 0 ];
        int[]  primes  =  (int[])  Array.CreateInstance(typeof(int),
Config.MAX_LEN);

        int ind = 0;

        primes[0] = ar[0];

        for(int i = 1; i < ar.Length; i++)
        {
            if(isPrime(ar[i],primes)) primes[++ind] = ar[i];
        }

        return primes;
    }

    // Функция,проверяющая число на простоту
    private bool isPrime(int n, int[]primes)
    {
        bool isPrime = true;
        for(int   j   =   0;   isPrime   &&   j   <   primes.Length   &&
primes[j]!=0 && primes[j]*primes[j] <= n; j++)
isPrime = (n % primes[j] != 0);

        return isPrime;
    }

    async   Sieve(handler   int[]()   getNatPack,   channel   (int[])
sendPrimesPack)
```

```
    {
        // Получаем первый пакет из входного потока и
        // извлекаем из него подпакет простых чисел
        int[] head = SynchronousSieve((int[])getNatPack());

        // Посылаем пакет простых чисел в выходной поток
        sendPrimesPack( head );

        // Фильтруем оставшиеся пакеты входного потока
        // относительно пакета head
        if ( head.Length != 0 )
        {
            new CSieve().Sieve( inter, sendPrimesPack);
            filter ( head, getNatPack, cout );
        }

    }
    handler inter int[]()  &  channel cout ( int[] p) {
        return  ( x );
    }
    // Фильтрация потоков
    void filter(int[] head, handler int[] () getNatPack, channel
(int[]) cfiltered)
    {
     int[]                              al                              =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

        int ind = 0;

        // Для каждого пакета из входного потока
        for( int[] p; (p = (int[])getNatPack() ).Length != 0;)
        {
            // Выбираем простые числа, формируя новый пакет
            for(int i = 0; i < p.Length && p[i]!=0; i++)
            {
                if(isPrime(p[i],head))
                {
                    al[ind++] = p[i];

                    // Если пакет заполнен, то посылаем его
                    // в поток отфильтрованных пакетов
                    if(ind == Config.MAX_LEN)
                    {
                        ind = 0;
                        cfiltered (al);
                        al                                              =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

                    }
                }
            }
        }
```

```
        // Посылаем последний пакет в поток
        // отфильтрованных пакетов
        if(al[0] != 0) cfiltered (al);

        // Посылаем маркер конца потока пакетов
        cfiltered( new int [ 0 ] );
    }
class Eratosthenes2   {
 public static void Main(String[] args) {

   Eratosthenes2 er2 = new  Eratosthenes2();
   CSieve csieve = new CSieve();

   // Запускаем метод Sieve

   csieve.Sieve( er2.getNatPack, er2.sendPrimePack);

   int[]                              al                                =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

        // Создаем поток пакетов натуральных чисел
        for (int i = 2; i <= Config.N; i++)
        {
            int ind = (i-2)%Config.MAX_LEN;

            al[ind] = i;
            if(ind == Config.MAX_LEN - 1)
            {
                er2.Nats ( al );

                al                                                    =
(int[])Array.CreateInstance(typeof(int),Config.MAX_LEN);

            }
        }

        if(al[0] != 0)
         er2.Nats(al);
        er2.Nats ( new int [ 0 ] );

        int[] p;
        // Распечатываем результирующие пакеты простых чисел
        while (( p = (int[])er2.getPrimePack()).Length != 0)
         Config.print(p);
 }

 handler getNatPack  int[]()  & channel Nats ( int[] p ) {
  return ( p );
 }
 handler getPrimePack int[]() & channel sendPrimePack( int[] p ) {
  return ( p );
 }
}
```

[A5] All2all

```
using System;

class   BDChannel   {
 handler  Receive object()
             & channel Send ( object obj )  {
   return  ( obj );
 }
}

class  All2all   {

 public static void Main (String[] args)   {

  int  i;

  // N есть число распределенных процессов
  int  N = System.Convert.ToInt32 ( args [ 0 ] );

  All2all a2a = new All2all();
  DistribProcess dproc = new DistribProcess();

  //   Запуск распределенных процессов
  for ( i = 0; i < N; i++ )
    dproc.Start ( i, a2a.sendBDC, a2a,sendStop );

  //   Получение объектов класса BDChannel от процессов
  BDChannel[] bdchans = new BDChannel [ N ];
  for ( i = 0; i < N; i++ )
    a2a.getBDC ( bdchans );

  //   Рассылка массива объектов класса BDChannel
  //   каждому из процессов
  for ( i = 0; i < N; i++ )
     bdchans [ i ].Send ( bdchans );

  //   Получение сигналов обокончании работы процессов
  for ( i = 0; i < N; i++ )
    a2a.getStop();
 }
 handler getBDC void( BDChannel[] bdchans)  &
       channel sendBDC ( int i, BDChannel bdc )      {
    bdchans [ i ] = bdc;
 }
 handler  getStop void() & channel  sendStop() {
   return;
 }
}

class DistribProcess   {
```

```
 movable Start ( int  myNumber, channel  (int, BDChannel)
                              sendBDC, channel () sendStop ) {
  //  i есть собственный номер процесса
  int   i;
  BDChannel  bdc = new BDChannel();
  sendBDC ( myNumber, bdc );
  BDChannel[]   bdchans  =  (BDChannel[]) bdc.Receive();

  //   Посылка сообщений другим процессам
  for  ( i = 0; i < bdchans.Length; i++ )
   if  ( i != myNumber )
     bdchans[j].Send ("Message from process " + myNumber +
                                    " to process " + i       );

  //   Прием сообщений от других процессов
  //   (здесь можно было бы воспользоваться и каналом
  //    bdchans [ myNumber ])
  for  ( i = 0; i < bdchans.Length - 1; i++ )
    Console.WriteLine ( "Process " + myNumber + " : " +
                        (String) bdc.Receive() );

  //   Посылка сигнала об окнчании работы
  sendStop();
 }

}
```