Введение в программирование на языке MC#

Версия 1.07

Сентябрь 10, 2009г.

Содержание.

1. **Введение**

2. **Асинхронные методы**

3. **Каналы и обработчики**

4. **Распределенное программирование на языке MC#**

5. **Примеры программирования на языке MC#**

   5.1 Числа Фибоначчи
   5.2 Программа all2all
   5.3 Игра Конвэя “Жизнь”
   5.4 Программа рендеринга изображений на базе LINQ
   5.5 Задача N-Queens

1. Введение

Язык программирования MC# является расширением языка C# и предназначен для написания параллельных и распределенных программ. Параллельная программа – это программа, которая предназначена для исполнения на машинах, имеющих несколько ядер/процессоров и общую (разделяемую) память для них. Распределенная программа – это программа, которая предназначена для исполнения на нескольких машинах (возможно, многоядерных), каждая из которых имеет свою собственную память. Примерами систем для исполнения распределенных программ являются кластеры и Grid-сети.

Язык программирования MC# основан на модели асинхронного параллельного программирования, впервые введенной в языке Polyphonic C# (http://research.microsoft.com/en-us/um/people/nick/polyphony.htm). Особенность этой модели состоит в том, что в ней предложены высокоуровневые параллельные конструкции, которые превращают объектно-ориентированный язык C# в язык параллельного программирования. В частности, они обеспечивают все необходимые средства, которые требуются при параллельном программировании:

1) средства порождения параллельных процессов (потоков),
2) средства взаимодействия (передачи сообщений) между параллельными процессами;
3) средства синхронизации работы параллельных процессов.

Эти высокоуровневые конструкции естественным образом входят в объектно-ориентированную модель программирования и, практически, исключают необходимость применения дополнительных библиотек (таких как библиотека System.Threading платформы .NET, а также библиотек Microsoft Parallel Extensions for .NET и Intel Threading Building Blocks). Недостатком последних из вышеназванных библиотек является то, что, во-первых, в них вводится семейство дополнительных параллельных конструкций, таких как потоки (threads) и задачи (tasks), и, во-вторых, они, как таковые, не поддерживают распределенное программирование.

В данном документе описываются новые конструкции языка программирования MC# и приводятся законченные примеры их использования для написания параллельных и распределенных программ. Правила компиляции и запуска программ обоих видов приведены в “Руководстве пользователя”, входящем в установочный пакет системы программирования MC#.

2. Асинхронные методы.

В любом из традиционных объектно-ориентированных языков, обычные методы являются **синхронными**: вызывающая программа всегда ожидает завершения исполнения вызванного метода, и только затем продолжает свою работу. Ключевая особенность языка MC# состоит в добавлении к обычным, синхронным методам так называемых **асинхронных** методов. Асинхронные методы (а также рассматриваемые далее перемещаемые (***movable***) методы) – это единственный способ порождения параллельных процессов в языке MC# (традиционные средства порождения параллельных процессов, такие как потоки, являются библиотечными функциями, доступными из программ на языке C#).

Общий синтаксис определения асинхронных методов в языке MC# имеет вид:

модификаторы ***async*** *имя_метода (аргументы)*
{
  < тело метода >
}

Заметим, что ключевое слово ***async***, определяющее метод как асинхронный, располагается на месте типа возвращаемого значения. Соответственно, синтаксическое правило, задающее тип возвращаемого значения в объявлении метода в языке MC#, имеет вид:

return_type ::= type | ***void*** | ***async*** | ***movable***

(Таким образом, ключевое слово ***movable***, задающее перемещаемый метод, также должно располагаться на месте типа возвращаемого значения).

Задание ключевого слова ***async*** при объявлении некоторого метода означает, что при вызове данного метода он будет запущен в виде отдельного потока на данной машине. Отличия ***async***-метода от обычного, синхронного метода состоят в следующем:

- вызов ***async***-метода заканчивается, по существу, мгновенно; т.е., после вызова такого метода, не дожидаясь завершения его работы, управление передается на оператор, следующий за оператором вызова;
- ***async***-методы не возвращают результатов.

Правила корректного определения ***async-***методов в языке MC# включают в себя также следующие требования:

- ***async***-методы не могут объявляться статическими;
- в их теле не может использоваться оператор ***return***;
- к формальным параметрам ***async-***методов не могут применяться модификаторы ***ref***, ***out*** и ***params***.

Пример 1.

В данном примере иллюстрируется применение асинхронных методов в параллельной программе перемножения матриц. Программа рассчитана на исполнение на двух процессорах (обобщение программы на произвольное число процессоров может быть несложным упражнением для читателя), а в качестве средства определения завершения работы асинхронных методов используется объект класса *ManualResetEvent*.

```
using System;
public class MatrixMultiplier {

public static int N = 1000;
public static int count = 2;
public static void Main ( String[] args )  {
 double[]   a, b, c;
 a = new double [ N, N ];
 b = new double [ N, N ];
 c = new double [ N, N ];
 Random r = new Random();
 for ( int i = 0; i < N; i++ )
  for ( int j = 0; j < N; j++ ) {
   a [ i, j ] = r.NextDouble();
   b [ i, j ] = r.NextDouble();
   c [ i, j ] = 0.0;
 }

    MatrixMultiplier  mm = new MatrixMultiplier();
   using ( ManualResetEvent mre = new ManualResetEvent ( false ) )
  {
   mm.multiply ( 0, N/2, a, b, c, mre );
   mm.multiply ( N/2, N, a, b, c, mre );
   mre.WaitOne();
  }
}
public async multiply ( int from, int to, double[,] a, double[,] b, double[,] c,
                                   ManualResetEvent mre                                           )
{
 for ( int i = from; i < to; i++ )
  for ( int j = 0; j < N; j++ )
   for ( int k = 0; k < N; k++ )
     c [ i, j ] += a [ i, k ] * b [ k, j ];
 if ( Interlocked.Decrement ( ref count ) == 0 )
  mre.Set()
 }
}
```

В действительности, в языке MC# имеются собственные высокоуровневые средства, обеспечивающие как взаимодействие асинхронных методов (передачу данных и сигналов между ними), так и синхронизацию их работы. Этими средствами являются **каналы** и **обработчики**, рассматриваемые в следующем разделе.

3. Каналы и обработчики

Каналы и **обработчики канальных сообщений** (или, просто, **обработчики**) представляют собой средства для организации взаимодействия параллельных и распределенных процессов между собой. Второе их назначение в языке MC# – служить средством синхронизации работы вышеуказанных процессов. Синтаксически, каналы и обработчики обычно объявляются в программе с помощью специальных конструкций – **связок** (chords).

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 (и, соответственно, каналов и обработчиков) в языке MC# имеют вид:

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

В приведенных правилах, нетерминалы *body, attributes, modifiers, return-type* и *formal-parameters* определяются согласно стандартным синтаксическим правилам языка C#. Нетерминалы *handler-name* и *channel-name* являются простыми (т.е., не составными) идентификаторами.

При определении каналов и обработчиков действуют следующие ограничения:

1) каналы и обработчики не могут объявляться статическими;
2) к формальным параметрам каналов и обработчиков не могут применяться модификаторы ***ref***, ***out*** и ***params***;
3) 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;*
4) все идентификаторы формальных параметров каналов и обработчиков из связки должны быть различными.

Одна из важных ключевых особенностей 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 (т.е., в рамках которых они объявлены). В этом смысле, каналы и обработчики похожи на указатели на функции в языке C или, в терминах языка C#, на делегатов (***delegates***).

В соответствии с этим, система типов языка MC# включает в себя типы для каналов и обработчиков:

type ::= chanel-type | *handler-type* | …
channel-type ::= ***channel*** *(type-list)*
handler-type ::= ***handler*** *retur-type (type-list)*
type-list ::= *// empty list*
| *type [, type]**

Отличие каналов и обработчиков от остальных типов (как скалярных, так и ссылочных) состоит в том, что они могут объявляться **только** в составе связок, т.е., с обязательным указанием тела связки. Следствием этого является то, что каналы и обработчики не могут объявляться аналогично другим типам – например, объявление

public channel *c1;*

не является допустимым; соответственно, невозможно (прямое) определение массивов каналов или обработчиков, а также выполнение операций присваивания для них. Отметим, что, так как каналы и обработчики всегда привязаны к некоторому объекту, в рамках которого они определены, то все указанные выше операции могут быть косвенно осуществлены с использованием этих объектов. Так, например, чтобы определить массив каналов, достаточно определить массив объектов, содержащих определенные внутри них каналы. Иллюстрации таких определений будут приведены в Разделе 5 “Примеры программирования на языке MC#”.

Синтаксис оператора посылки значений по каналу во многом совпадает с синтаксисом вызова обычного метода и имеет вид:

[qualified-object-name.] channel-name ! (argument-list);

Например, для посылки по каналу *sendInt*, объявленного в рамках объекта *a*, целого числа *n*, необходимо записать выражение

a.sendInt ! (n);

Синтаксис оператора вызова обработчика имеет двойственный вид:

[qualified-object-name.] handler-name ? (argument-list);

Если обработчик возвращает значение, то при присваивании этого значения некоторой переменной, это значение должно быть **явно протипизировано**. Например, для получения значения с помощью обработчика *getInt*, возвращающего целочисленные значения, необходимо записать выражение вида

int *x = (**int**) a.getInt ? ();*

Если к моменту вызова обработчика, связанный с ним канал пуст (т.е., по этому каналу значений не поступало или все значения были выбраны посредством предыдущих обращений к обработчику), то этот вызов блокируется – программа переходит в состояние ожидания. В случае когда обработчик связан с несколькими каналами, блокировка наступает, если не во всех каналах есть соответствующие значения. Когда по каналу приходит очередное значение (или, в общем случае, во всех каналах связки появляются недостающие значения), то происходит исполнение тела связки, и по оператору ***return*** происходит возврат обработчиком результирующего значения.

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

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

Пример 2

В данном примере иллюстрируется запуск нескольких асинхронных методов, каждому из которых в качестве одного из аргументов передается канал *sendStop*. По завершении своей работы, каждый ***async***-метод посылает сигнал об этом главной программе посредством вызова

sendStop ! ( );

Главная программа, используя цикл ***for***, принимает соответствующее количество сигналов об окончании работы ***async***-методов:

```
for  ( i = 0; i < N; i++ )
  atc.getStop ? ();
```

```
using System;
public class AsyncTerminationClass {
 public static int N = 10;
 public static void Main ( String[] args )   {
   int  i;
   AsyncTerminationClass atc = new AsyncTerminationClass();
   for ( i = 0; i < N; i++ )
    atc.a_method ( i, atc.sendStop );
   for ( i = 0; i < N; i++ )
    atc.getStop ?();
  }
  public async a_method ( int myNumber, channel () sendStop )
  {
   Console.WriteLine ( "Process " + myNumber );
   sendStop ! ();
  }
  public handler getStop void()   &   public channel sendStop () {
   return;
  }
}
```

Пример 3

Данный пример иллюстрирует использование связок в качестве средства синхронизации. Например, для связки вида

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

тело связки сработает и обработчик *Get2* возвратит значение только в случае когда по обоим каналам *c1* и *c2* посланы значения. В общем случае, один обработчик может быть связан с произвольным числом каналов.

Связка, приведенная выше, обычно используется для

а) обнаружения завершения работы ***async***-методов, и
б) получения значений от этих методов.

В примере, показанном ниже – программе вычисления чисел Фибоначчи, вычисление *n*-го числа Фибоначчи сводится к рекурсивному вычислению *n-1*-го и *n-2*-го чисел Фибоначчи асинхронно. Соответствующая связка позволяет обнаружить завершение вычисления рекурсивно вызванных методов и получить от них результирующие значения.

```
using System;
public class Fib
{
 public handler Get2 long() & channel c1( long x ) & channel c2( long y ) {
  return x + y;
 }
 public async Compute( long n, channel( long ) c )
 {
  Console.WriteLine( "Compute: n=" + n );
  if ( n <= 1 )
   c ! ( 1 );
  else
  {
   new Fib().Compute( n-1, c1 );
   new Fib().Compute( n-2, c2 );
   c ! ( (long)Get2 ? () );
  }
 }
}

public class ComputeFib
{
 handler Get long() & channel c( long x ) {
  return x;
 }
 public static void Main( string[] args )
 {
  if ( args.Length < 1 )
  {
   Console.WriteLine( "Usage: Fib.exe <number>" );
   return;
  }
  int n = System.Convert.ToInt32( args [ 0 ] );
  ComputeFib cf = new ComputeFib();
  Fib fib = new Fib();
  fib.Compute( n, cf.c );
  Console.WriteLine( "For n = " + n + " value is " + cf.Get?() );
 }
}
```

4. Распределенное программирование на языке MC#

Под “распределенным программированием” понимается написание программ, предназначенных для исполнения на сети из двух и более компьютеров (например, на вычислительном кластере, состоящем, как правило, из головного узла и множества рабочих узлов).

Особенность языка MC# заключается в том, что в нем сохраняется единая модель программирования как для локального (в рамках одной многоядерной/многопроцессорной машины), так и для распределенного случаев. А именно, для порождения локальных асинхронных потоков используются ***async***-методы, а для порождения асинхронных потоков, которые

могут быть спланированы для исполнения на другой машине, используются, так называемые, “перемещаемые” или ***movable***-методы.

Синтаксис определения ***movable***-методов аналогичен синтаксису ***async***-методов, за исключением того, с первыми из названных методов может использоваться только модификатор ***public***:

> *[* ***public*** *]* ***movable*** *имя_метода (аргументы)*
> *{*
> < *тело метода* >
> *}*

Отличия ***movable***-методов от обычных методов и правила их корректного определения в языке MC# совпадают с аналогичными отличиями и правилами для ***async***-методов (см. Раздел 2 “Асинхронные методы”).

При разработке распределенной программы на языке MC# необходимо учитывать следующие особенности её выполнения, связанные с передачей объектов и значений между различными машинами на которых выполняется программа.

Во-первых, объекты, создаваемые во время исполнения MC#-программы, являются, по своей природе *статическими*: после своего создания они остаются привязанными к тому месту (машине), где они были созданы, и в дальнейшем не перемещаются.

Однако, при вызове ***movable***-метода, все необходимые данные для этого, а именно:

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

только **копируются** (но не перемещаются) на удаленную машину. Следствием этого является то, что все изменения, которые осуществляются с этим скопированным объектом на удаленной машине, не переносятся на оригинальный объект.

Пример 4

В приведенном ниже примере, вызов ***movable***-метода *Compute*, в котором производится изменение поля *x*, никак не влияет на значение этого поля объекта *b* в главной программе.

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

class B  {

  public int x;
 public B ()  { }

 movable Compute ()   {
   x = 2;
 }

}
```

Исполнение этой программы приведет к печати на консоли сообщений:

```
Before movable method call: x = 1
After movable method call: x = 1
```

Пример 5

Программу из Примера 3 легко переделать в распределенную программу, заменив в определении метода *Compute* ключевое слово ***async*** на слово ***movable***:

```
using System;
public class Fib
{
 public handler Get2 long() & channel c1( long x ) & channel c2( long y ) {
  return x + y;
 }
movable Compute( long n, channel( long ) c )
 {
  Console.WriteLine( "Compute: n=" + n );
  if ( n <= 1 )
   c ! ( 1 );
  else
  {
   new Fib().Compute( n-1, c1 );
   new Fib().Compute( n-2, c2 );
   c ! ( (long)Get2 ? () );
  }
 }
}
public class ComputeFib
{
 handler Get long() & channel c( long x ) {
  return x;
 }
 public static void Main( string[] args )
 {
  if ( args.Length < 1 )
  {
   Console.WriteLine( "Usage: Fib.exe <number>" );
   return;
  }
  int n = System.Convert.ToInt32( args [ 0 ] );
  ComputeFib cf = new ComputeFib();
  Fib fib = new Fib();
  fib.Compute( n, cf.c );
  Console.WriteLine( "For n = " + n + " value is " + cf.Get?() );
 }
}
```

Напомним (см. Раздел 3), что каналы и обработчики могут передаваться в качестве аргументов ***movable***-методам отдельно от объектов, которым они принадлежат. Одна из **ключевых особенностей исполнения распределенных MC#-программ** состоит в том, что при копировании каналов и обработчиков на удаленную машину автономно или в составе некоторого объекта, они становятся *прокси-объектами*, или посредниками для оригинальных каналов и обработчиков. Такая подмена скрыта для прикладного программиста – он может использовать переданные каналы и обработчики на удаленной машине (а, в действительности, их прокси-объекты) 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: модификации последних на удаленной машине не переносятся на исходные объекты.

Ниже, на Рис.1 и 2 схематически показаны передача и использование каналов и обработчиков на удаленной машине.

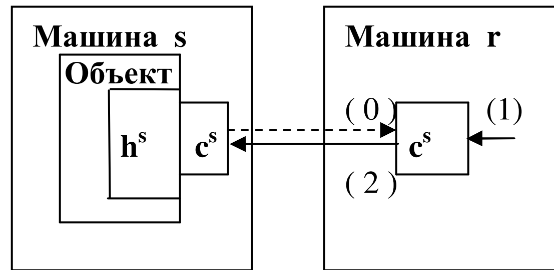


Рис. 1. Посылка сообщения по удаленному каналу:
(0) копирование канала на удаленную машину,
(1) посылка сообщения по (удаленному) каналу,
(2) перенаправление сообщения на исходную машину**.**

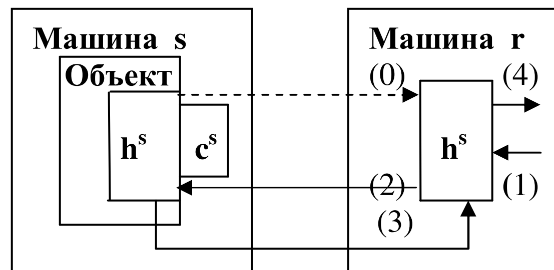


Рис. 2. Чтение сообщения из удаленного обработчика:
(0) копирование обработчика на удаленную машину,
(1) чтение сообщения из (удаленного) обработчика,
(2) перенаправление операции чтения на исходную машину,
(3) получение прочитанного сообщения с исходной машины,
(4) возврат полученного сообщения.

Замечание.

При разработке распределенного приложения, программист должен стремиться, по возможности, минимизировать порождение прокси-объектов для каналов и, особенно, для обработчиков – иногда удается построить эквивалентный вариант программы в котором отсутствуют, например, прокси-объекты для обработчиков, что позволяет избежать операций чтения с удаленной машины. Пример построения такого эквивалентного варианта см. в Разделе 5.1 “Числа Фибоначчи”.

Конструкций каналов и обработчиков оказывается достаточно для организации взаимодействия произвольной сложности между параллельными или распределенными процессами, что иллюстрируется в следующем разделе.

5. Примеры программирования на языке MC#

В данном разделе приводятся примеры программ на языке MC#, которые иллюстрируют использование специфических конструкций языка: ***async***- и ***movable***-методов, каналов, обработчиков и связок. Даются первоначальные сведения о разработке эффективных алгоритмов на языке MC#. Полные тексты приведенных примеров можно найти в установочном пакете системы программирования MC#.

5.1 Числа Фибоначчи

В Примерах 3 и 5 были приведены параллельная и распределенная программы вычисления чисел Фибоначчи, главная цель которых была продемонстрировать использование специфических конструкций языка MC# в действии.

Однако, эти варианты программы являются очень неэффективными с вычислительной точки зрения, поскольку в них каждый вновь порождаемый поток выполняет лишь небольшое количество вычислительных операций, а именно, только порождает два новых дочерних потока и отправляет результат по каналу. В этом случае, накладные расходы операционной системы на порождение новых потоков во много раз превышают количество полезных вычислительных операций.

Один из способов повышения эффективности такого рода программ состоит во введении специального порога (THRESHOLD) для входного аргумента: если входное значение не превышает заданный порог, то это значение обрабатывается последовательно, т.е., без порождения параллельных потоков. Если входное значение превышает порог, то его обработка производится с порождением параллельных потоков вплоть до того уровня, когда входное значение очередного рекурсивного вызова не станет меньше порога.

Пример 6

В данном примере показан модифицированный класс *Fib*, который использует порог для входного аргумента функции *Compute* и функцию *cfib* для последовательного рекурсивного вычисления *n*-го числа Фибоначчи.

```
public class Fib
{
  public static int THRESHOLD = 35;
  public handler Get2 long() & channel c1( long x ) & channel c2( long y ) {
    return x + y;
  }
  public async Compute( long n, channel( long ) c )
  {
    Console.WriteLine( "Compute: n=" + n );
    if ( n <= THRESHOLD )
     c ! ( cfib( n ) );
    else
   {
     new Fib().Compute( n-1, c1 );
     new Fib().Compute( n-2, c2 );
     c ! ( (long)Get2 ? () );
    }
   }
   private long cfib( long n )
  {
    if ( n <= 1 )
     return n;
    else
    return cfib( n - 1 ) + cfib( n - 2 );
  }
}
```

Программа из Примера 6 является более эффективной, чем программа из Примера 3, тем не менее, каждый поток, входное значение для которого превышает значение порога, по-прежнему, выполняет очень небольшое число полезных операций, а именно, порождая два дочерних потока. Существует так называемый “линейный” вариант параллельной программы вычисления чисел Фибоначчи суть которого состоит в том, что каждый поток, порождаемый при вызове ***async***-метода *Compute*, вычисляет один рекурсивный вызов этого метода последовательно, а для другого такого вызова порождает параллельный поток. Таким образом, для получения числа Фибоначчи с номером THRESHOLD + N будет порожден N + 1 параллельный поток.

Пример 7

Здесь показан модифицированный класс *Fib*, реализующий линейный вариант параллельного вычисления чисел Фибоначчи.

```
public class Fib
{
  public static int THRESHOLD = 35;

 public handler Get int()  & channel c1( int x ) { return x ; }

 public async Compute( int n, channel( int ) c )
 {
  if ( n <= THRESHOLD )
   c ! ( cfib( n ) );
  else {
   new Fib().Compute( n - 1, c1 );
   c ! ( cfib( n - 2 ) + (int) Get ? () );
  }
 }
 private int cfib( int n ) {
  if ( n <= 1 )
   return ( n );
  else
   return ( cfib( n - 1 ) + cfib( n - 2 ) );
 }
}
```

В Примере 5 был представлен распределенный вариант программы вычисления чисел Фибоначчи. Заметим, что в нем (как и в Примере 3) выражение для рекурсивного вызова функции *Compute* имело вид

***new** Fib().Compute (value, channel_name);*

Порождение каждый раз нового объекта класса *Fib* здесь необходимо для формирования древовидной системы каналов и обработчиков, с помощью которой рекурсивно вызванные методы возвращают вычисленные ими значения. (Заинтересованный читатель может проследить что произойдет, если не порождать новые объекты при каждом рекурсивном вызове метода *Compute).*

Распределенный вариант программы, приведенный в Примере 5, является недостаточно эффективным из-за порождения большого количества прокси-обработчиков и, как следствие, большого количества операций чтения с удаленных машин. А именно, предположим, что исполнение некоторого очередного вызова метода *Compute* происходит на машине *M1*. При этом, при рекурсивном вычислении выражения

***new** Fib().Compute (n - 1, c1);*

породится новый объект класса *Fib*, который будет обладать своими собственными каналами *c1* и *c2* и обработчиком *Get2*. Вызов ***movable***-метода *Compute* этого объекта приведет к тому, что для исполнения этого метода будет выбрана некоторая машина *M2*, куда будет скопирован упомянутый выше объект класса *Fib,* и на которой его каналы и обработчики станут прокси-

объектами для исходных каналов и обработчиков, расположенных на машине *M1*. Таким образом, обращение к обработчику

Get2 ? ()

на машине *M2* будет приводить к операции удаленного чтения с машины *M1*.

Пример 8

В данном примере показано, как можно повысить эффективность работы распределенной программы вычисления чисел Фибоначчи, исключив использование прокси-обработчиков. Для этого достаточно

1) вынести связку, состоящую из обработчика *Get2* и каналов *c1* и *c2* в отдельный класс;
2) предусмотреть создание объекта указанного класса внутри метода *Compute*, что будет приводить к созданию обработчиков и каналов именно на той машине, где происходит исполнение метода *Compute*.

Вспомогательный класс *Comm* и модифицированный класс *Fib* для данного варианта программы показаны ниже:

```
public class Comm
{
  public handler Get2 long() & channel c1( long x ) & channel c2( long y ) {
    return x + y;
  }
}

public class Fib {
 movable Compute( long n, channel( long ) c )
 {
  Comm comm = new Comm();
  if ( n <= 1 )
   c ! ( n );
  else
  {
   new Fib().Compute( n-1, comm.c1 );
   new Fib().Compute( n-2, comm.c2 );
   c ! ( (long)comm.Get2 ? () );
  }
 }
}
```

5.2 Программа *all2all*

С помощью каналов и обработчиков можно организовывать произвольные схемы взаимодействия между параллельными и распределенными процессами. Одним из распространенных типов таких схем является схема взаимодействия

вида “*all2all”*, когда каждый процесс из группы процессов должен иметь возможность обмениваться сообщениями с любым другим процессом из этой группы.

Для организации такого взаимодействия в программе на языке MC# необходимо выполнить три предварительных шага до начала основного взаимодействия:

1) каждый процесс должен создать канал, по которому другие процессы смогут посылать ему сообщения, с соответствующим обработчиком; эта связка канал-обработчик оформляется в виде объекта *interact_bdc* класса *BDChannel* (bi-directional channel);
2) каждый процесс должен переслать свой объект *interact_bdc* главной программе, которая сохраняет все такие объекты от всех процессов в массиве *interact_bdchans*;
3) главная программа должна переслать массив *interact_bdchans* каждому из процессов.

После выполнения этих трех шагов, каждый процесс сможет посылать сообщения любому другому процессу, используя каналы из массива *interact_bdchans*.

Пример 9

Полный текст программы *all2all* представлен ниже. Следует отметить, что каждый процесс, кроме объекта *interact_bdc*, создает еще объект *bdc* класса *BDChannel*, который предназначен для принятия массива объектов *interact_bdchans* от главной программы. В действительности, создания объекта *bdc* можно избежать одним из двух возможных способов:

1) использовать для пересылки массива *interact_bdchans* от главной программы объект interact_bdc; однако, в этом случае пришлось бы усложнить структуру сообщений, передаваемых с помощью объекта *interact_bdc*, чтобы обеспечить распознавание источника сообщения, которых теперь имеется два – главная программа и процесс из группы;
2) передать параллельному процессу в качестве дополнительного параметра специальный обработчик главной программы, связанный с каналом, по которому главная программа будет пересылать массив *interact_bdchans*.

Заинтересованному читателю предоставляется возможность реализовать эти варианты программы *all2all* в качестве упражнения.

```
using System;
class  BDChannel  {
 public handler  Receive object() & channel Send ( object obj )  {
```

```
  return  ( obj );
 }
}
class  All2all   {
 public static void Main (String[] args)   {
  int  i;
  // N is a number of processes
  int  N = System.Convert.ToInt32 ( args [ 0 ] );
  All2all a2a = new All2all();
  DistribProcess dproc = new DistribProcess();
  //   Run the processes
  for ( i = 0; i < N; i++ )
    dproc.Start ( i, a2a.sendBDC, a2a.sendStop );
  //   Receive the (BD)channels from the processes
  BDChannel[] bdchans          = new BDChannel [ N ];
  BDChannel[] interact_bdchans = new BDChannel [ N ];
  for ( i = 0; i < N; i++ )
    a2a.getBDC ? ( bdchans, interact_bdchans );
  //   Send a (BD)channel array to every process
  for ( i = 0; i < N; i++ )
     bdchans [ i ].Send ! ( interact_bdchans );
  //   Receive the stop signals from the processes
  for ( i = 0; i < N; i++ )
    a2a.getStop ? ();
 }
 public handler getBDC void( BDChannel[] bdchans, BDChannel[] interact_bdchans )  &
      channel sendBDC ( int i, BDChannel bdc, BDChannel interact_bdc )      {
    bdchans [ i ]          = bdc;
    interact_bdchans [ i ] = interact_bdc;
 }
 public handler  getStop void() & channel  sendStop() {
   return;
 }
}
class DistribProcess   {
 movable Start ( int  myNumber, channel  (int, BDChannel, BDChannel ) sendBDC,
                                  channel () sendStop ) {
  int   i;
  BDChannel  bdc          = new BDChannel();
  BDChannel  interact_bdc = new BDChannel();
  sendBDC ! ( myNumber, bdc, interact_bdc );
  BDChannel[]   interact_bdchans  =  (BDChannel[]) bdc.Receive ? ();
  //   Send message to other processes
  for  ( i = 0; i < interact_bdchans.Length; i++ )
   if  ( i != myNumber )
     interact_bdchans[i].Send ! ( myNumber );
  //   Receive messages from other processes
  //   (here it is possible to use  interact_bdchans [ myNumber ] channel )
  //
  for  ( i = 0; i < interact_bdchans.Length - 1; i++ )
    Console.WriteLine ( myNumber + " <- " + interact_bdc.Receive ? () );
  //   Send stop signal
  sendStop ! ();
 }
}
```

5.3 Игра Конвэя "Жизнь"

Игра Конвэя "Жизнь" ("Game of Life") есть простая математическая модель эволюции сообщества живых клеток. Игра проводится на прямоугольной области, каждая клетка которой может находиться в одном из двух состояний – ALIVE ("живая") или DEAD ("мертвая"). Компьютерное моделирование сообщества таких клеток состоит в многократном вычислении состояния каждой клетки, которое зависит от состояния соседних клеток. Точные правила игры можно найти в директории *Examples/Async/IntelThreadingChallenge/GameOfLife* установочного пакета системы программирования MC#.

Особенность реализации этой игры состоит в том, что прямоугольная область разбивается на горизонтальные полосы по количеству параллельных потоков для их обработки. При этом, параллельный поток должен взаимодействовать на каждой итерации с соседними потоками, обрабатывающими полосы выше и ниже полосы, которые обрабатывает данный поток. Это необходимо для учета влияния пограничных клеток соседних полос друг на друга.

Схема взаимодействия соседних потоков, реализованная в программе на MC#, строится на использовании массива *comms* объектов класса *Communicator*. Объект *comms[i]* ($0 \le i <$ количество потоков - 1) обеспечивает взаимодействие потоков с номерами *i* и *i+1*. Само взаимодействие состоит в посылке сигналов о готовности данных – поток *i* посылает сигналы потоку *i-1* по каналу *toUp* и потоку *i+1* по каналу toDown. Соответственно, поток *i-1* принимает эти сигналы с помощью обработчика *fromDown*, а поток с номером *i*+1принимает эти сигналы с помощью обработчика *fromUp*.

На каждой итерации, каждый параллельный поток выполняет следующие действия:

1) модифицирует состояние клеток собственной полосы с помощью функций *Vivify* и *Kill*; одновременно, в массивах *up_deltas* и *down_deltas* готовится информация для соседних потоков;

2) по каналам *toUp* и *toDown* посылаются сигналы о готовности данных в массивах *up_deltas* и *down_deltas*;

3) посредством обработчиков *fromUp* и *fromDown* принимаются сигналы о готовности соответствующих данных от соседних потоков;

4) производится перевычисление состояния пограничных клеток в соответствии с данными, полученными от соседних потоков;

5) по каналам *toUp* и *toDown* посылаются сигналы об окончании обработки данных из массивов *up_deltas* и *down_deltas*, принадлежащих соседним потокам;

6) посредством обработчиков *fromUp* и *fromDown* принимаются сигналы от соседних потоков о завершении обработки информации от данного процесса.

Ниже показан код основного цикла работы параллельного потока, обрабатывающего отдельную полосу области:

```
//    Iterations ( building the generations of cells )
  while ( gencount < gens )   {

   gencount++;

   Vivify ( ownNumber, first, last, P, maylive, newlive ); // maylive |--> newlive
   maylive.Clear();
   Kill   ( ownNumber, first, last, P, maydie, newdie   ); // maydie  |--> newdie
   maydie.Clear();

   if ( ownNumber != 0 )                          //
    comms [ ownNumber - 1 ].toUp ! ();       //       Deltas
   if ( ownNumber != P - 1 )                      //         are
    comms [ ownNumber ].toDown ! ();        //       ready

   if ( ownNumber != 0 )                            //
    comms [ ownNumber - 1 ].fromUp ? ();     //    Wait the deltas
   if ( ownNumber != P - 1 )                        //        from the
    comms [ ownNumber ].fromDown ? ();      //     neighbor threads

   AddNeighbors ( first, last, newlive, maylive, maydie ); //    newlive |--> maylive, maydie
   newlive.Clear();
   SubtractNeighbors ( first, last, newdie, maylive, maydie ); // newdie |--> maylive, maydie
   newdie.Clear();

   if ( ownNumber != 0 )
    HandleDeltas ( ownNumber - 1, down_deltas, first, maylive, maydie );
   if ( ownNumber != P - 1 )
    HandleDeltas ( ownNumber + 1, up_deltas, last, maylive, maydie    );

   if ( ownNumber != 0 )
    comms [ ownNumber - 1 ].toUp ! ();
   if ( ownNumber != P - 1 )
    comms [ ownNumber ].toDown ! ();

   if ( ownNumber != 0 )                           //       Wait
    comms [ ownNumber - 1 ].fromUp ? ();    //    the finishing
   if ( ownNumber != P - 1 )                      //    of deltas handling
    comms [ ownNumber ].fromDown ? ();    //    by the neighbor threads

  }
```

Полный текст программы “Game Of Life” можно найти в директории *Examples/Async/IntelThreadingChallenge/GameOfLife* установочного пакета системы программирования MC#.

5.4 Программа рендеринга изображений на базе LINQ

Начиная с версии 2.1, в языке MC# поддерживаются все конструкции языка C# 2.0 и 3.0, в частности, лямбда-функции и LINQ-выражения.

В директории *Examples/Async/SimpleLinqRayTracer* установочного пакета системы программирования MC# находится программа рендеринга изображений на основе метода трассировки лучей (ray tracing), базовые функции которой выражены с помощью LINQ-выражений. В частности, функция, находящая точки пересечения заданного луча со всеми объектами сцены, имеет вид

```
private IEnumerable<ISect> Intersections(Ray ray, Scene scene)
{
  return scene.Things
             .Select(obj => obj.Intersect(ray))
             .Where(inter => inter != null)
             .OrderBy(inter => inter.Dist);
 }
```

Распараллеливание рендеринга всего изображения производится путем разбиения этого изображения на вертикальные полосы по числу заданных процессоров. Для обработки каждой полосы запускается асинхронный метод *render*:

```
int q = screenWidth / P,
      r = screenWidth % P;

int from = 0,
     to;

for ( i = 0; i < P; i++ ) {
 to = from + q + ( i < r ? 1 : 0 );
 this.render ( i, from, to, scene, rgb, this.sendStop );
 from = to;
}
```

Асинхронный метод *render* вычисляет цвета каждой точки обрабатываемой им полосы. Значение цвета записывается с помощью функции *setPixel* в общий для всех параллельных потоков массив *rgb*:

```
internal async render ( int myNumber, int from, int to, Scene scene, int[] rgb, channel () sendStop ) {

  for (int y = 0; y < screenHeight; y++)
  {
     for (int x = from; x < to; x++)
     {
      Color color = TraceRay(new Ray() { Start = scene.Camera.Pos, Dir = GetPoint(x, y,
scene.Camera) }, scene, 0);
       setPixel(x, y, color );
     }
     sendStop ! ();
  }
```

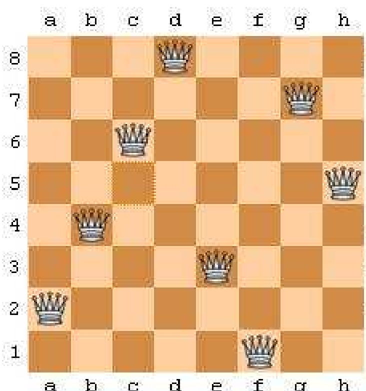
Полный код данной программы можно найти в директории *Examples/Async/SimpleLinqRayTracer*.

В связи с данной программой рендеринга изображений, отметим одну особенность разработки программ на языке MC#, которая может сильно влиять на эффективность их исполнения.

В общем случае, каждый класс MC#-программы транслируется компилятором MC# в класс на языке C#, оформленный специальным образом для поддержки передачи сообщений по каналам (в частности, в распределенном режиме), которые может иметь данный класс. Такое специальное оформление может влиять на общую производительность приложения, особенно в случаях, когда создается очень большое количество объектов на основе этого класса. Однако, часто в приложении имеется набор классов, не использующих специфические средства языка MC# - ***async***- и ***movable*** методы, каналы, обработчики и связки. Поэтому, для обеспечения большей эффективности всего приложения рекомендуется все такие классы собирать в отдельные C#-модули (.cs-файлы), которые могут подаваться на вход MC#-компилятору вместе с .mcs-файлами. MC#-компилятор оставляет .cs-файлы в неизменном виде. В частности, в программе *SimpleLinqRayTracer* специфические функции рендеринга изображений, не связанные с параллельной обработкой, собраны в отдельном классе *Classes.cs*.

5.5 Задача N-Queens

В данном разделе рассматривается реализация на языке MC# задачи N-Queens – подсчитать сколькими различными способами могут быть расставлены ферзи на шахматной доске размером N x N так, чтобы они попарно не конфликтовали. Как обычно, считается, что два ферзя конфликтуют (“бьют” друг друга), если они расположены на одной и той же горизонтали, вертикали или диагонали.



Базовым алгоритмом программы на языке MC# является алгоритм на основе битовых векторов (см. статью Qiu Zougyan “Bit-Vector Encoding of N-Queen Problem”).

Опуская подробности этого алгоритма, мы ниже представим тольку схему его распараллеливания. Отметим здесь еще раз, что в силу универсальности программной модели, заложенной в основу языка MC#, параллельный и распределенный варианты программы отличаются только использованием ключевых слов ***async*** и ***movable***, с помощью которых помечается метод, предназначенный для исполнения в параллельном потоке.

Суть схемы распараллеливания состоит в том, что мы определяем все возможные допустимые расстановки M ферзей на первых M горизонталях доски (где M ≤ N, и горизонтали нумеруются сверху-вниз). Каждая такая расстановка оформляется в виде задачи (task), которая представляет данную расстановку в виде трех битовых векторов *left*, *down* и *right*. Все сгенерированные задачи посылаются в канал *sendTask*, откуда они извлекаются параллельными потоками с помощью обработчика *getTask*.

Параллельный рабочий поток (*Worker*) извлекает очередную задачу, подсчитывает количество всех возможных допустимых расстановок N – M ферзей на нижних N – M горизонталях при фиксированных ферзях на первых M горизонталях, и отправляет полученное число главной программе при обращении за очередной задачей.

Данная схема работы параллельных потоков демонстрирует простейший способ балансировки нагрузки между потоками – те потоки, которые справляются со своей задачей более быстро, извлекут больше задач из очереди канала *sendTask*.

Ниже приведен полный текст программы, в которой используется асинхронный метод *Worker*. С помощью замены ***async*** на ***movable*** в объявлении этого метода, можно получить распределенный вариант данной программы.

```
using System;
public class Task   {
 public int left, down, right;
 public Task ( int l, int d, int r ) {
  left  = l;
  down  = d;
  right = r;
 }
}
//************************************************//
public class NQueens   {
 public static long totalCount = 0;
 public static void Main ( String[] args ) {
  int   N = System.Convert.ToInt32 ( args [ 0 ] );   //  Board size
  int   M = System.Convert.ToInt32 ( args [ 1 ] );   //  Number of fixed queens
  int   P = System.Convert.ToInt32 ( args [ 2 ] );   //  Number of workers
  NQueens nqueens = new NQueens();
  nqueens.launchWorkers ( N, M, P, nqueens.getTask, nqueens.sendStop, nqueens );
  nqueens.generateTasks ( N, M, P, nqueens.sendTask );
  for ( int i = 0; i < P; i++ )
   nqueens.getStop ? ();
  Console.Write     ( "Task challenge : " + N + "    " );
  Console.WriteLine ( "Solutions = " + totalCount );
 }
 //****************************************************************************//
 public handler getTask Task(int count ) & channel sendTask ( Task task ) {
  totalCount += count;
  return ( task );
 }
 //****************************************************************************//
 public handler getStop void() & channel sendStop () {
  return;
 }
 //****************************************************************************//
 public async launchWorkers ( int N, int M, int P, handler Task(int) getTask,
                              channel () sendStop, NQueens nqueens            ) {
  for ( int i = 0; i < P; i++ )
   nqueens.Worker ( i, N, M, getTask, sendStop );
 }
 //****************************************************************************//
 public void generateTasks ( int N, int M, int P, channel (Task) sendTask ) {
  int   y     = 0;
  int   left  = 0;
  int   down  = 0;
  int   right = 0;
  int   MASK  = ( 1 << N ) - 1;
  MainBacktrack ( y, left, down, right, MASK, M, sendTask );
  Task finish_marker = new Task ( -1, -1, -1 );
  for ( int i = 0; i < P; i++ )
   sendTask ! ( finish_marker );
 }
 //******************************************************************************//
 public void MainBacktrack ( int y, int left, int down, int right, int MASK,
                             int M, channel (Task) sendTask                 ) {
  int   bitmap, bit;
  if ( y == M )
```

```
  sendTask ! ( new Task ( left, down, right ) );
 else   {
  bitmap = MASK & ~ ( left | down | right );
  while ( bitmap != 0 )   {
   bit   = -bitmap & bitmap;
   bitmap = bitmap ^ bit;
   MainBacktrack ( y + 1, ( left | bit ) << 1, down | bit, ( right | bit ) >> 1,
                 MASK, M, sendTask                                       );
  }
 }
}
//*****************************************************************************//
public async Worker ( int myNumber, int N, int M, handler Task(int) getTask,
                channel () sendStop                               ) {
 int    MASK  = ( 1 << N ) - 1;
 int    count = 0;
 Task   task  = (Task) getTask ? ( count );
 while ( task.left != -1 )   {
  WorkerBacktrack ( M, task.left, task.down, task.right, MASK, N, ref count );
  task  = (Task) getTask ? ( count );
  count = 0;
 }
 sendStop ! ();
}
//*****************************************************************************//
public void WorkerBacktrack ( int y, int left, int down, int right, int MASK,
                          int N, ref int count                        ) {
 int   bitmap, bit;
 if ( y == N )
  count++;
 else   {
  bitmap = MASK & ~ ( left | down | right );
  while ( bitmap != 0 )   {
   bit   = -bitmap & bitmap;
   bitmap = bitmap ^ bit;
   WorkerBacktrack ( y + 1, ( left | bit ) << 1, down | bit, ( right | bit ) >> 1,
                 MASK, N, ref count                                      );
  }
 }
}
}
```

Пример вывода распределенной программы для N = 18 при исполнении ее на 8 двухпроцессорных узлах приведен ниже:

```
$ mono Distributed_NQueens.exe 18 3 16 /withlog /showstats /completeboot /np 8
MC#.Runtime, v. 2.2.0.1176
Application Guid: 0
Task challenge : 18   Solutions = 666090624
_________________________________________________
==MC# Statistics================================
Number of movable calls: 16
Number of channel messages: 3452
Number of movable calls (across network): 16
Number of channel messages (across network): 16
Total size of movable calls (across network): 10944 bytes
Total size of channel messages (across network): 2016 bytes
Total time of movable calls serialization: 00:00:00.0849960
Total time of channel messages serialization: 00:00:00.0035450
Total size of transported messages: 852504 bytes
Total time of transporting messages: 00:00:00.6773350
Session initialization time: 00:00:01.2010060 / 1.201006 sec. / 1201.006 msec.
Total time: 00:01:45.4398850 / 105.439885 sec. / 105439.885 msec.
Number of movable calls (mc) per node:
  1. 1 x node-41 | ********************    2 mc 12.50%
  2. 1 x node-31 | ********************    2 mc 12.50%
  3. 1 x node-32 | ********************    2 mc 12.50%
  4. 1 x node-21 | ********************    2 mc 12.50%
  5. 1 x node-22 | ********************    2 mc 12.50%
  6. 1 x node-33 | ********************    2 mc 12.50%
  7. 1 x node-24 | ********************    2 mc 12.50%
  8. 1 x node-44 | ********************    2 mc 12.50%
--- 8 of 8 nodes were used ---
```