Introduction to programming in MC# language

Version 1.07

September 10, 2009

Contents

1. **Introduction**

2. **Asynchronous methods**

3. **Channels and handlers**

4. **Distributed programming in MC#**

5. **Sample programs**

   5.1 Fibonacci numbers
   5.2 All2all program
   5.3 Conway's "Game Of Life"
   5.4 LINQ based image rendering
   5.5 N-Queens problem

1. Introduction

MC# programming language is an extension of the C# language and is intended for developing of concurrent and distributed programs. A concurrent program is a program which is intended for running on multicore/multiprocessor machines with shared memory. A distributed program is a program for running on the network of (possibly, multicore/multiprocessor) machines with separate memories. The clusters and Grids are the examples of systems for running of distributed programs.

MC# programming language is based on the asynchronous parallel programming model which originally was introduced in the Polyphonic C# language (http://research.microsoft.com/en-us/um/people/nick/polyphony/). The given model proposes the high-level, concurrent constructions which turn the object-oriented C# language to parallel programming language. In particular, they provide all needed features which are required for parallel programming, namely, tools for

1) creating,
2) interaction (message passing) and
3) synchronizing

of concurrent processes.

Such high-level constructs fit well into the object-oriented programming model and, in fact, free the programmer from the need to use the additional libraries (such as System.Threading library from .NET Framework, Microsoft Parallel Extensions for .NET and others). The mentioned above libraries have such shortcomings as, first of all, they introduce additional process-like notions as "thread" or "task", and, secondly, they doesn't support a distributed programming.

In the given manual, novel constructs of MC# language are described and complete examples of their usage to develop concurrent and distributed programs are provided. The compilation and running rules for both types of programs can be found in "User's Guide" which has been included into the install package of MC# programming system.

2. Asynchronous methods

In any traditional object-oriented language, conventional methods are ***synchronous***: the caller always waits until the callee is completed and only then continues its work. The one of the key features of MC# is the introduction of so called ***asynchronous*** methods in addition to conventional synchronous ones. Asynchronous (and also ***movable*** – see below) methods are the only way to create concurrent and distributed processes in MC# programs. (The traditional tools to create concurrent processes and threads are accessible in C# programs through library function calls).

The general syntax of asynchronous method declaration in MC# is the following:

modifiers ***async*** *method_name (arguments)*
{

< method body >

}

Note that the keyword ***async*** defining a method as asynchronous is placed instead the return type of the method. Correspondingly, we have the following rule which defines the syntax of return type in MC# language:

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

So the keyword ***movable*** defining corresponding method is placed also instead of the return type.

Declaring a method with ***async*** keyword means that given method will be running locally in a separate thread. The differences of ***async***-method from conventional synchronous method are the following:

- ***async***-method call completes almost immediately; i.e., after the call of it, a control is passing to the following statement without waiting of the former;
- ***async***-methods never return a result (for interaction and message passing among them see Section 3 "Channels and handlers").

By the rule of correct definition, ***async***-methods in MC#

- may not have a ***static*** modifier,
- never use a ***return*** statement, and
- ***ref***, ***out*** and ***params*** modifiers can't be applied to the formal parameters of such methods.

Example 1.

The example below demonstrates the using of asynchronous methods in the concurrent program for matrix multiplication. The program is intended for running on two processors (its extension to arbitrary number of processors can be a simple exercise for the reader). An object of *ManualResetEvent* type serves as a tool to determine termination of asynchronous methods.

```
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()
 }
}
```

Indeed, MC# language has its own high-level tools to support both an interaction of asynchronous methods (namely, data and signals transferring) and synchronization between them. Channels and handlers are such tools and they are considered in the next Section.

3. Channels and handlers

Channels and channel message handlers (or, simply, handlers) are the tools to support an interaction between concurrent or distributed processes. The second role of them is to serve as a synchronization tool for the processes. Syntactically, channels and handlers are declared using the special constructs – the ***chords***.

For example, the channel *sendInt* for transferring single integers is declared along with corresponding handler *getInt* as

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

In general, the chords (and, correspondingly, channels and handlers) are declared in MC# programs according to the following syntactical rules:

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)

In above rules, the non-terminals *body*, *attributes*, *modifiers*, *return-type* and *formal-parameters* are defined by C# standard syntactical rules. The non-terminals *handler-name* and c*hannel-name* are the simple (non-qualified) identifiers.

Channel and handler declarations are subject to the following restrictions:

1) channels and handlers cannot be defined as ***static***,
2) modifiers ***ref***, ***out*** and ***params*** cannot be applied to formal parameters of channels and handlers,
3) if a handler has defined with *return-type* else than *void* in the chord, then every ***return*** statement in the chord body must return a value of *return-type*;
4) all identifiers for channel and handler parameters in the chord must be unique.

The important key feature of MC# language is that channels and handlers can be passed as arguments to methods (in particular, to ***async***- and ***movable*** methods) ***separately*** from the object to which they belong (in other words, from the object within which they has been defined). In this sense, channels and handlers are similar to the pointers to functions in C/C++, or, in C# terms, to ***delegates***. Accordingly, the type system of MC# language includes the types for channels and handlers:

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

The difference between channels and handlers and other types (both scalar and reference) is in that they can be declared ***only*** within the chords with obligatory defining of the chord's body. As a consequence, channels and handlers cannot be declared similar to the convenient types; for example, a declaration as

```
public channel c1;
```

is not allowed. Correspondingly, there are no ways to declare both channel and handler arrays directly and to use an assignment statement for channels and handlers. But it should be noted that due to that channels and handlers are always parts of objects within which they has been declared , all mentioned above operations can be implemented by using such objects indirectly. For example, to declare an array of channels it's enough to declare an array of objects, which, in turn, contains the corresponding channels (see suitable illustrations in Section 5 "Example programs").

The syntax of statement to send value through the channel in much is similar to invoke an ordinary method:

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

Thus, we may send an integer *x* by the channel *sendInt* as

a.sendInt ! (n);

where *a* is an object for which the channel *sendInt* has been defined.

The syntax of statement to call a handler has the dual form:

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

For handler which returns a value, we need ***to cast*** this value before assigning it to the some variable. For example, to receive an integer value by the handler *getInt* we need to write

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

If, by the time a handler is called, the corresponding channel is empty (i.e. if there have been no calls to this channel at all or all of the values sent through this channel before were absorbed during previous calls to the handler), then the call blocks and the program passes to wait state. If a handler is tied with few channels in the chord, a blocking state comes in the case when there is some empty channel. After receiving a value from the channel (or, in general case, all channels have values), body of the chord executes and returns a result value through the handler.

Conversely, if a value is sent on a channel when there are no pending calls to the handler, the value is simply saved in the internal queue of channel, where all the values coming with multiple sendings over the channel are accumulated. After invoking the handler and under condition that all channels from the chord contain the values, the first values from the channels queues will be selected for handling.

It is worth to note that triggering of the chord consisting from the handler and a few channels is possible principally due to they are called typically from the different threads.

Example 2.

The current example illustrates the running of several ***async***-methods where each of them takes the channel *sendStop* as one of the arguments. After termination, each of the ***async***-methods sends a stop signal to the main program through the call

```
sendStop ! ( );
```

The main program receives a corresponding number of stop signals from the ***async***-methods in the ***for*** loop:

```
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;
 }
}
```

Example 3.

The example below demonstrates the using of chords as a synchronization tool. The body of the chord

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

can be triggered and the handler *Get2* will return a value only both channels *c1* and *c2* have the values. In general, one handler can be joined with an arbitrary number of channels.

A chord of the above mentioned form is used typically

a) to detect a termination of ***async***-methods, and
b) to take the values from them.

In the sample shown below – a program to compute Fibonacci numbers, a computation of n^{th} Fibonacci number is reduced to the recursive computation of $n\text{-}1^{th}$ and $n\text{-}2^{th}$ Fibonacci numbers asynchronously. The corresponding chord allows to detect the termination of recursively called methods and to take the result values from them.

```
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. Distributed programming in MC#

By "distributed programming" we mean a writing of programs which intended to run on 2 or more computers (for example, on computational cluster having one main and many work nodes).

The distinctive feature of MC# language is that it preserves a single programming model both for the concurrent (local) and distributed cases: ***async***-methods are used to create local asynchronous threads, while ***movable*** methods are used to create threads which can be scheduled to execute on remote machines.

The syntax rules to declare the ***movable*** methods are similar to the rules for ***async***-methods with that exception that ***movable*** methods may have only ***public*** modifier:

```
[ public ] movable  method_name   ( arguments )
{
        < method body >
}
```

The distinctions of ***movable*** methods from conventional methods and the rules of correct definition of the former coincide with the ones for ***async***-methods (see Section 2 "Asynchronous methods").

During development of distributed programs in MC# language it is necessarily to take into account some properties of performing of distributed programs. These properties follow from the rules of object passing between the machines which perform a distributed MC# program.

First of all, the objects created during of MC# program execution are ***static*** by their nature: once created, they remain bound to the place (machine) where they were created and don't move further. But when we invoke a ***movable*** method, all necessary data, namely

1) the object itself to which the given ***movable*** method belongs, and
2) arguments of call (both scalar and reference values)

are only ***copied*** (but not moved) to the remote machine. As a consequence, changes made afterwards to the copy at remote machine will not affect the original object.

Example 4.

In the code below, an invoke of ***movable*** method *Compute*, which alters the field *x*, doesn't change that field of object *b* in the main program.

```
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;
 }

}
```

A running of that program gives the output

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

Example 5.

For the program from Example 3 it is easy to make its distributed version by replacing ***async*** keyword by ***movable*** in the declaration of *Compute* method:

```
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?() );
 }
}
```

Recall (see Section 3) that channels and handlers can be passed as arguments to the ***movable*** methods separately from the object to which they belong. One of the ***key feature of execution of distributed MC# programs*** is that if channels and handlers were copied to a remote machine autonomously or as part of some object, then they become *proxy objects*, or intermediaries for the original channels and handlers. This replacement is hidden from the applied programmer – he can use the passed channels and handlers (in fact, their proxy objects) on the remote machine as the original ones: as usual, all actions over the proxy objects are redirected to the original channels and handlers by the Runtime-system. In this sense, channels and handlers are different from convenient objects: modifications of the latter on a remote machine are not passed to the original objects.

Below Fig.1 and Fig.2 demonstrate schematically the passing and use of channels and handlers on a remote machine. The superscripts on the channel and handler names denote the name of the machine where they were created originally.

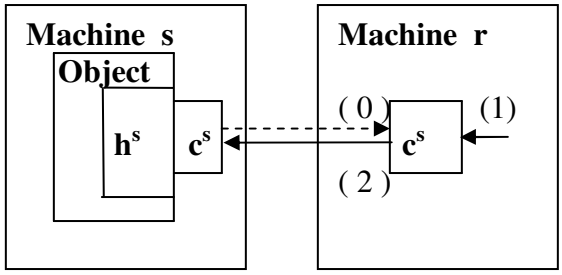


Fig. 1. Message sending through remote channel:
(0) copying of the channel to remote machine,
(1) message sending through (remote) channel,
(2) message redirection to the original machine**.**

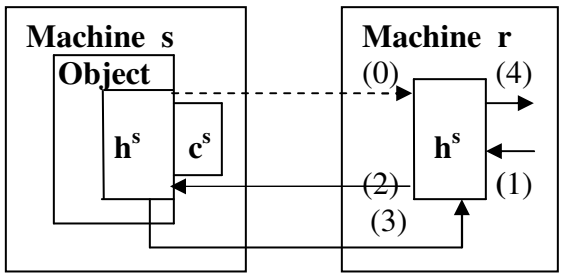


Fig. 2. Message reading from remote handler:
(0) copying of the handler to remote machine,
(1) message reading from (remote) handler,
(2) reading redirection to the original machine,
(3) message return from the original machine,
(4) result message return.

Note.

During development of distributed applications, a programmer must seek to minimize as far as possible a creation of proxy-objects for channels and, especially, for handlers. Sometimes it is possible to build an equivalent variant of the program in which proxy-objects for handlers are absent that allows to avoid a reading from remote machines (see an example of such equivalent variant in Section 5.1 “Fibonacci numbers”).

It turns out that channels, handlers and chords are enough tools to organize an interaction of arbitrary complexity among the concurrent or distributed processes that has been demonstrated in the examples of the next Section.

5. Sample programs

In this section, sample programs are presented which illustrate the use of specific constructs of MC# language – **async**- и **movable** methods, channels, handlers and chords. Initial information about developing effective algorithms in MC# is given. Complete code for sample programs can be found in the installation package of MC# programming system.

5.1 Fibonacci numbers

The concurrent and distributed programs to compute Fibonacci numbers have been presented in Examples 3 and 5 (see Sections 3 and 4, correspondingly). These programs' main goal was to demonstrate specific constructs of MC# language in action.

However these programs are very inefficient from a computational point of view, because each newly created thread performs just a small number of computational operations – it spawns two child threads and sends the result to the channel. For this case, the overhead of creating and running threads by many times exceeds the number of useful computational operations.

One of the methods to improve efficiency of such kind of programs is to introduce a special THRESHOLD for input parameter. If the input parameter is equal to or less than the THRESHOLD, the former is proceeded serially, without thread spawning. Otherwise the input value is proceeded in a parallel mode spawning threads until the input for the next recursive call is equal to or less than the THRESHOLD.

Example 6

The given example demonstrates a modified class *Fib* which uses a THRESHOLD for the input parameter of function *Compute*. Also it applies function *cfib* to compute the n'th Fibonacci number serially.

```
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 );
   }
}
```

The program from Example 6 is more effective than the one from Example 3. Nevertheless each thread with the input parameter greater than the THRESHOLD still performs a small number of useful operations creating two child threads only. But there is a so called a "linear" variant of the parallel program to compute Fibonacci numbers. The essence of it is to compute two recursive calls to *Compute* differently: one is computed serially and the other is computed by a newly created ***async***-method. So to compute (THRESHOLD + N)th Fibonacci number we need N + 1 concurrent threads.

Example 7

Here is a modified class *Fib* which implements a linear variant of the concurrent program to compute Fibonacci numbers:

```
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 ) );
 }
}
```

Earlier we have presented (see Example 5) a distributed program to compute Fibonacci numbers. Note that in it (as and in Example 3) an expression for recursive call of *Compute* method has the form

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

Here, creating a new object of class *Fib* every time is necessary to form a tree-like system of channels and handlers by means of which the recursively called methods return resulting values to the parent methods. (The interested reader may keep on what happens if we will spawn recursive calls to *Compute* method without constructing new objects).

In the distributed case, such version of the program will be ineffective due to creating a great number of proxy-handlers, and, as consequence, creating a great number of reading operations from remote machines. Suppose some current call to *Compute* method is executed on machine *M1*. While this is being done, the recursive computation of the expression

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

creates a new object of class *Fib*, which will have its own channels *c1* and *c2* and its handler *Get2*. Invoking this object's ***movable*** method *Compute* will result in that some machine *M2* will be scheduled to execute the method. The above-mentioned object of class *Fib* will be copied to *M2*, where the object's channels and handlers will become proxy-objects for the original ones located on machine *M1*. Therefore, invoking the handler

Get2 ? ()

on machine *M2* will lead to remote reading from machine *M1*.

Example 8

This example shows how to make the distributed Fibonacci program more efficient by avoiding proxy-handlers. For this purpose it is enough

1) to place a chord containing handler *Get2* and channels *c1* and *c2* into the separate class;
2) to provide for creating an object of that class inside the *Compute* method; this will result in creating channels and handlers exactly on that machine where the *Compute* method is executed.

The additional class *Comm* and modified class *Fib* are shown below.

```
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 All2all program

As was mentioned above, channels and handlers are sufficient tools to arrange arbitrary pattern interactions among the concurrent and distributed processes. One of the wide-spread types of such patterns is “all2all”, in which each process from a group of processes exchanges messages with any other process from the group.

To organize such pattern in MC# program we need to carry out 3 preliminary steps before the basic interaction begins:

1) each process must create a channel (along with the corresponding handler) by which other processes can send messages to it; that chord "channel-handler" is placed into the object *interact_bdc* of class *BDChannel* (bi-directional channel);
2) each process must send its *interact_bdc* object to the main program, which collects all of them in the array *interact_bdchans*;
3) main program must send the *interact_bdchans* array to each process in the group.

After these steps, each process will have possibility to send messages to any other process from the group through the channels collected in the *interact_bdchans* array.

Example 9

The complete code of program *all2all* is shown below. Note that each process creates along with object *interact_bdc* also object *bdc* of class *BDChannel*. This object is intended for receiving object array *interact_bdchans* from the main program. In fact, there are two ways to avoid creating object *bdc*:

1) to use the object *interact_bdc* to transfer the array *interact_bdchans* from the main program; but in this case we need to have more complex messages passed by object *interact_bdc*, because we need now to point out the message source – main program or process from the group;
2) to pass to the process as an additional parameter some special handler from the main program; this handler belongs to the chord which contains the channel to transfer the array *interact_bdchans* from the main program.

An interested reader has chance to implement these alternative variants of program *all2all* as an exercise.

```
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 Conway's "Game Of Life"

The Conway's "Game Of Life" is a simple mathematical model of evolution of living cell community. A field of game is a rectangular area where each cell can be in one of two states – ALIVE or DEAD. A computer modeling of community of living cells consists of an iterative recomputation of each cell's state. In general, the state of a cell depends on the states of neighboring cells. The precise rules of game can be found in the directory *Examples/Async/IntelThreading/Challenge/GameOfLife* of installation package of MC# programming system.

The essence of parallel implementation of the "Game Of Life" is a partitioning of rectangular area into a number of horizontal stripes according to the number of parallel

threads. Each parallel thread interacts with the neighboring threads which handle the stripes located above and below its own stripe. This is necessary to take into account the influence that the border cells of neighboring stripes have on each other.

The interaction pattern implemented in MC# program is based on using a *comms* array containing objects of class *Communicator*. The object *comms[i]* ($0 \le i <$ number of threads – 1) provides interaction between the thread *i* and thread *i+1*. The interaction consists of sending signals: the thread *i* sends signals to thread *i-1* by channel *toUp* and sends signals to thread *i+1* by channel *toDown*. The thread *i-1* receives the signals through the handler *fromDown*, and the thread *i+1* receives the signals through the handler *fromUp.*

On each iteration, each parallel thread performs the following actions:

1) it modifies the states of cells from its own stripe invoking the functions *Vivify* and *Kill;* simultaneously, it prepares information for neighboring threads in *up_deltas* and *down_deltas* arrays;
2) when data in *up_deltas* and *down_deltas* arrays are ready, it sends the signals by the channels *toUp* and *toDown;*
3) it receives the "data are ready" signals from the neighboring threads through the *fromUp* and *fromDown* handlers;
4) it recomputes the states of border cells based on the data received from the neighboring threads;
5) when the handling of *up_deltas* and *down_deltas* arrays from the neighboring threads has been finished, it sends the corresponding signals by channels *toUp* and *ToDown*;
6) it receives the signals reporting the completion of corresponding work from neighboring threads through *fromUp* and *fromDown* handlers.

Shown below is the code for the basic loop of the parallel thread that handles a separate stripe of the game area:

```
//   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

  }
```

The complete code of program "Game Of Life" can be found in the directory *Examples/Async/IntelThreadingChallenge/GameOfLife* of the installation package of MC# programming system.

5.4 LINQ based image rendering

Starting with version 2.1, MC# language supports all C# 2.0 and 3.0 novel constructions such as lambda-functions, LINQ-expressions etc.

The directory *Examples/Async/SimpleLinqRayTracer* of installation package of MC# programming system contains an image rendering program based on ray tracing technique. Basic parts of this ray tracing algorithm are presented as LINQ-expressions. In particular, the function that finds points of intersection of a given ray with all objects of the scene has the form

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

To parallelize the entire image rendering, we divide the image into vertical stripes. The number of stripes equals the number of processors we have. For each stripe we start the asynchronous method *render* to handle that stripe:

```
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;
}
```

The asynchronous method *render* computes the color in each pixel of the stripe. The function *setPixel* writes a computed color to the array *rgb* shared by all parallel threads:

```
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 ! ();
  }
```

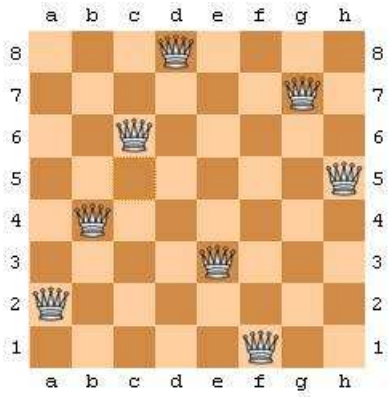
The complete code of the program can be found in the directory *Examples/Async/SimpleLinqRayTracer*.

Concerning the image rendering program, we would like to stress some specific property of developing MC# programs. This property may have a great influence on the efficiency of MC# programs performance.

In general, the MC# compiler translates every class of MC# program into the corresponding C# class extended in a special way to support message passing (particularly, in distributed mode) through the channels that the original class may have. Such extension may have an influence on the entire performance of application, especially in the case when application creates a very large number of objects based on MC# class. But often application has a set of classes that do not use the specific constructs of MC# language – ***async-*** and ***movable*** methods, channels, handlers and chords. Therefore to increase entire efficiency of application it is reasonable to collect all such classes in separate C# modules. Such C# modules (.cs-files) can be used as input to the MC# compiler along with .mcs-files because the MC# compiler leaves .cs-files unchanged. In particular, in the program *SimpleLinqRayTracer* all image rendering specific functions which have no relation with parallel handling are collected in separate file *Classes.cs*.

5.5 N-Queens problem

The N-Queens problem is a problem to place N queens on an N x N chess board such that no queen can attack another. As usual, it is considered that one queen attacks another if they share the same horizontal or vertical or diagonal. Finding the number of all possible arrangements is the goal of the problem. Below we consider a parallel solution of N-Queens problem in MC# language.



The basic algorithm implemented in the MC# program is a bit-vector solution with only .bit operations (see Qiu Zongyan's paper "Bit-Vector Encoding of N-Queen Problem").

Omitting algorithm's details, we present below only a parallelization scheme for it. Note once more that due to universality of programming model exploited in MC# language, the concurrent and distributed variants of the program differ from one another only by the keyword – ***async*** or ***movable*** – used to marked the method running in the parallel thread.

The technique for the parallelization of N-Queens problem solution is the following. At first we determine all possible placements of M queens on the first M rows of the chess board (where M ≤ N and rows are numbered top down). Each such placement is considered as a task representing the placement by the three vectors *left*, *down* and *right*. All generated tasks are sent to the channel *sendTask*. The parallel threads draw out the tasks from the channel through the handler *getTask*.

The parallel thread (*Worker*) extracts the next task, computes the number of all possible placements of N – M queens on the lower N – M rows, with M queens being fixed on the first M rows, and sends the result to the main program simultaneously with the call for a new task. By this way we have a simple technique for workload balancing – those threads that handle tasks faster will extract more tasks from the queue of channel *sendTask*.

Listed below is the complete code of the program which uses an asynchronous method *Worker*. A distributed variant of this program can be obtained by replacing ***async*** keyword to ***movable*** in the declaration of method *Worker*.

```
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                                   );
  }
 }
}
}
```

The sample output of the distributed program (problem size is N = 18) running on the 8 two-processor nodes is given below.

```
$ 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 ---
```