MC#: asynchronous parallel programming language for cluster- and GRID-architectures

V.Guzev
Peoples' Friendship University of Russia,
Moscow, Russia, e-mail: vadim@joker.botik.ru

Y.Serdyuk
Program Systems Institute of the Russian
Academy of Sciences,Pereslavl-Zalessky,Russia
e-mail: Yury@serdyuk.botik.ru

A.Chudinov
Strategy LLC,Pereslavl-Zalessky,Russia
e-mail: chudinov@strategypartner.com

Abstract

MC# is a programming language for cluster- and GRID-architectures based on asynchronous parallel programming model accepted in Polyphonic C# language (N.Benton, L.Cardelli, C.Fournet; Microsoft Research, Cambridge, UK). Asynchronous methods of Polyphonic C# play two major roles in MC#:
1) as autonomous methods executed on remote machines, and
2) as methods used for delivering messages.
The former are identified in MC# as the "movable methods", and the latter form a special syntactic class with the elements named "channels". Similar to Polyphonic C#, chords are used for defining the channels and as a synchronization mechanism. The MC# channels are generalised naturally to "bidirectional channels", which may be used both for sending and receiving messages in the movable methods.

The runtime-system of MC# has as the basic operation a copying operation for the object which is scheduled for execution on remote machine. This copy is "dead" after the movable method has finished its work, and all changes of this remote copy are not transferred to the original object. Arguments of the movable method are copied together with an original object, but the passing of bidirectional channels is realised through transferring the proxies for such channels.

By way of experiments in MC#, we have written a series of parallel programs such as a computation of Fibonacci numbers, walking through binary tree, computation of primes by Eratosthenes sieve, calculation of Mandelbrot set, modeling the Conway's game "Life", etc. In all these cases, we got the easy readable and compact code.

Also we have an experimental implementation in which the compiler is written in SML.NET, and the execution of movable methods on remote machines is based on the Reflection library of .NET platform.

1 Int.Workshop on C# and .NET Technologies on Algorithms, Computer Graphics, Visualization, Computer Vision and Distributed Computig

February 6-8,2003, Plzen, Czech Republic.

ISBN 80-903100-3-6

Keywords

Polyphonic C#, asynchronous parallel programming, movable method, channel, bidirectional channel

1 Introduction

At present time, spread use of computer systems with cluster- and GRID-architectures posed a problem of developing high-level, powerful and flexible programming languages which allow one to create complex, but at the same time, robust applications that effectively use the possibilities of concurrent computations.

The program interfaces and libraries, which we have now, such as MPI (Message Passing Interface), that are realised for C and Fortran languages, are very low-level and not suited for the modern object-oriented languages, such as C++, C# and Java.

One of the recent seminal achievment in this area is introduction of an asynchronous parallel programming model within the Polyphonic C# programming language in the context of the Microsoft .NET platform [Benton et al. 2002].In turn, this model is based on the join-calculus [Fournet and Fessant 2002] - a process calculus with the high-level message handling mechanism adequately abstracting the low-level mechanism which exists in the current computer systems.

The essence of the new model or, in other words, the key feature of the Polyphonic C# language is the use of so called "asynchronous" methods in addition to conventional synchronous methods of a class. Such asynchronous methods can be declared both autonomously, and in this case they are scheduled for execution in a different thread (either a new one or a working thread from some thread pool), and within a bundle (or a chord, in terminology of Polyphonic C#) of other methods (synchronous and asynchronous). In the latter case, calling an asynchronous method, which

was declared in the chord, corresponds to sending a message or posting an event.

Such parallel programming style in Polyphonic C# as before is considered as a programming technique either for a single computer or for many machines interacting through the remote methods calls using .NET Remoting library.

Specific features of the proposed MC# language consist in the transferring of asynchronous parallel programming model of Polyphonic C# to distributed case, where an autonomous asynchronous method can be scheduled for execution in a different machine.With that, the asynchronous methods which are declared by chords and are used to deliver values to synchronous methods, form a special syntactic class with the elements named "channels". Therefore, a parallel program writing in MC# language is reduced to label by the special **movable** keyword the methods which may be transferred for execution to the different processors and arranging their interactions by the channels.

Earlier, an analogous approach in which a programmer has been partitioning all functions in the program into "movable" and "unmovable', used in the T-system [Abramov and Adamovich 1999].This system is intended for the dynamic scheduling of execution of parallel programs written in an extension of C.

Though the channels in MC# are "one-directional" in their nature (same as in the join-calculus), nevertheless they are generalised naturally to "bidirectional" channels which may be used by movable methods both for sending and receiving messages.

An implementation of the MC# language consists of a compiler for translating from the input language of a system to C#, and a runtime-system to execute a translated program. A compiler replaces the movable methods calls in the source program to queries to manager of computational resources that schedules the execution of parallel fragments of program in computer system. Having received a query, a manager selects the most suitable node of multiprocessor and copies an object, of which the movable method is scheduled for remote execution, to the selected node together with the arguments of this method. This copy is "dead" after the movable method has finished its work, and all changes that occurred to it are not transferred to the original object. Passing of bidirectional channels as arguments of methods is realised through the transferring the proxies for such channels.Thus, in MC# language, both the channels and the bidirectional channels are the local entities bounded to the place of their declaration. In particular, this means that the programmer is responsible for effective arrangement of communication by the channels.

As an initial stage of our work for the MC# language, we have written in it a series of parallel algorithms such as a computation of Fibonacci numbers, walking through the binary (balanced) tree, computation of primes by Eratosthenes sieve, calculation of Mandelbrot set, modeling the Conway's game "Life", etc. In all these cases, we got the easy readable and compact code for the corresponding problems due to the possibility to write parallel programs in MC# without taking care of their actual distribution over machines during execution. Similarly, there is no need for the manual programming in MC# of object (data) serialization in order to transfer these objects to remote processors (in contrast to MPI, where a special code is needed for a given problem) - the runtime-system of MC# performs an object serialization/deserialization automatically.

2 Asynchronous model of Polyphonic C# and its distributed variant

In C#, conventional methods are synchronous: the caller waits until the method called is completed, and then continues its work. In the world of parallel computations, reduction of execution time of a program is achieved by transferring some methods for execution in different processors, after that a program which transferred these methods immediately proceeds to the next instructions.

In Polyphonic C#, methods that commonly are scheduled for execution in the different threads within single computer are called asynchronous and they are declared by using the **async** keyword :

```
async Compute ( int n ) {
    // method body
}
```

The specifics of these methods is that their call completes essentially immediately; they never return the result; autonomous asynchronous methods always are scheduled for execution in a different thread (either a new one spawned to execute this call, or a working thread from some pool). In general case, asynchronous methods are defined using chords. A chord consists of a header and a body, where the header is a set of method declarations separated by the "&" symbol :

```
int Get() & async c (int x ) { return ( x ); }
```

The body of a chord is only executed once all the methods from chord header have been called. The single method calls are queued up until they aren't matched with the header of some chord.

In any chord, at most one method may be synchronous. Just in the thread associated with this method, the body of the chord is executed, and the returned value of it becomes a returned value of synchronous method.

In MC#, autonomous asynchronous methods always are scheduled for execution in a different processor, and they are declared by using the **movable** keyword. The main peculiarity of movable method call for some object consists in that the object itself is only copied (but not moved) to remote processor jointly with the movable method and input data for the latter. As a consequence, all changes of internal variables of the object are performed over variables of the copy and have no influence on the original object. So, the execution of the program

```
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);
```

```
  }
}
```
gives
Before:x=1
After:x=1

In MC#, asynchronous methods, which are defined in the chords, are marked by using the **Channel** keyword. And the only synchronous method from the chord plays the role of the method that receives values from the channel:

```
int Get () & Channel c ( int x ) { return ( x ); }
```

By the rules of correct definition, channels may not have a **static** modifier, and so they always are bounded to some object. Thus, we we may send a value by a.c (10), where *a* is an object of some class in which the channel *c* is defined. Also, as any object in a program, a channel may be passed as argument to some method :

```
  movable Compute ( Channel c ) {
  // method body
}
```

Thus, a **Channel** type plays the role of additional type for the type system of C#.

As in Polyphonic C#, it is also possible to declare a few channels in the single chord with the aim of their synchronization:

```
  int Get() & Channel c1(int x) & Channel c2(int y){return ( x + y ); }
```

The calling of *Get* method will return a sum only after receiving both arguments by the channels *c1* and *c2*.

3 Examples of programming in MC#

Let's consider a simple problem of computing the n-th (n≥0) Fibonacci number. The main computational procedure *Compute* of our program should compute the n-th Fibonacci number and return it by the given channel. With the assumption that the above procedure must be executed on a remote processor, we define it as a movable method:

```
class Fib {
  public movable Compute (int n,Channel c) {
    if ( n < 2 ) c ( 1 );
    else {
      Compute ( n - 1, c1 );
      Compute ( n - 2, c2 );
      c ( Get2 () );
    }
  }
  int Get2() & Channel c1 ( int x )
            & Channel c2 ( int y ) {
    return ( x + y );
  }
}
```

The main program may be the following :

```
class ComputeFib {
  public static void Main(String[] args){
    int n=System.Convert.ToInt32 (args[0]);
    ComputeFib cf = new ComputeFib();
    Fib fib = new Fib();
    fib.Compute ( n, cf.c );
  Console.WriteLine("n="+n+"result="+cf.Get());
  }
  public int Get() & Channel c ( int x ){
    return ( x );
  }
}
```

The above program has an essential shortcoming - an execution of any call of movable method comprises very few operations. And the effect of parallel execution will be decreased by the overhead charges to transport it in a different processor. A more effective variant for parallel execution is given below:

```
class Fib {
  public movable Compute(int n,Channel c){
    if ( n < 20 ) c ( cfib ( n ) );
    else {
      Compute ( n - 1, c1 );
      c ( cfib ( n - 2 ) + Get() );
    }
  }
  int Get() & Channel c1 (int x){
    return ( x );
  }
  int cfib ( int n ) {
    if ( n < 2 ) return ( 1 );
    else return (cfib(n-1) + cfib (n-2));
  }
}
```

Bidirectional channels

If some method got the channel as the argument, then it may send some values by the channel.And then how can we receive messages from this channel - if the corresponding method is "left" in the object where the channel was defined ? We may overcome this difficulty as proposed in [Fournet and Fessant 2002]. A programmer must "wrap up" a chord, in which the channel is defined, by the class with the name *BD-Channel* (Bi-Directional Channel) fixed in MC#.

For convenience, public methods for sending and receiving messages for a given channel may be defined in this class.If it is intended to use a few bidirectional channels with the different types, then all of them must be defined in one BDChannel class. This is an example of a simple BDChannel class:

```
public class BDChannel {
  public BDChannel () {}
  private int Get() & private Channel c (int x){
    return ( x );
  }
  public void send (int x) { c ( x ); }
  public int receive () { Get(); }
}
```

Now, having such a class, we can create the corresponding objects and pass them as arguments to other methods, in particular, to movable methods.

Bidirectional channels turn out a convenient feature in the parallel program for constructing primes by the Eratosthenes sieve. Given a natural number N, we need to enumerate all primes from 2 to N. Main computational procedure *Sieve* have two arguments: input channel *in* for receiving integers, and output channel *out* for producing primes extracted from the input stream. The end marker in both streams is -1. A part of the main method for given program is:

```
Main ( String [] args ) {
  int N=System.Convert.ToInt32 (args[0]);
  BDChannel nats = new BDChannel();
  BDChannel primes = new BDChannel();
```

```
  Sieve ( nats, primes );
  for (int i=2; i <= N; i++ ) nats.send ( i );
  nats.send ( -1 );
  while ((int p=primes.receive() ) != -1)
    Console.WriteLine ( p );
}
```

The *Sieve* method uses a function *filter* (**int** x, BDChannel in, BDChannel out), that sends integers not divisible by x , from *in* to *out* :

```
mobable Sieve (BDChannel in, BDChannel out){
  int head = in.receive();
  if ( head == -1 ) out.send ( -1 );
  else {
    out.send ( head );
    BDChannel inter = new BDChannel();
    Sieve ( inter, out );
    filter ( head, in, inter );
  }
}
```

It is possible to write a more effective variant for parallel execution, where a function *filter* handles an input stream *in* not by single prime x, but by each of the primes $x_1, ..., x_n$ from the package. In this case, bidirectional channels transfer packages of integers, where the package size is regulated by the programmer.

4 Implementation

As usual, for any parallel programming language, an implementation of MC# consists of a compiler and a runtime-system. The main functional parts of the runtime-system are:

1)**Manager** - a process running on the central node and distributing movable methods over the nodes.

2)**WorkNode** - a process running on each working node and controlling execution of movable methods transferred to given node.

3)**Communicator** - a process running on each node and responsible for receiving the channel messages for objects located on given node.

A compiler translates a program from MC# to C#, and its main purpose is to create code realising: 1)execution of movable methods in other processors, 2)transferring the channel messages and 3)synchronization defined in the chords. These functions are provided by the corresponding methods of classes of the runtime-system. Among these classes are:

1)**Session** class - provides for computational session;

2)**TCP** class - provides for sending both queries for movable methods execution and channel messages;

3)**Serialization** class - provides for serialization/ deserialization of objects that are transferred to the remote machines;

4)**Channel** class - contains information about the channel;

5)**LocalHost** class - contains information about the local node.

The main functions of MC#-compiler:

1. Adds the calls to functions *Init()* and *Finalize()* of the class **Session** to the main method of the program. Function *Init()* distributes the executable module to the remote machines, starts a **Manager** process, creates a **LocalNode** object and others. Function *Finalize()* stops the running threads and completes a computational session.

2. Adds an extra parameter *LocalHost* to each constructor of each object; it contains an information needed to create channels defined for a given object.

3. Adds the statements for creation of **Channel** objects for all channels defined in the program.

4. Replaces the calls to movable methods by the queries to **Manager** of computational resources.

5. Replaces the calls to channels by sending corresponding messages by TCP- connection.

Translating of chords, containing channel definitions, is conducted in the same way as in Polyphonic C#.

Passing of bidirectional channels as arguments of movable methods is implemented via creation and passing of proxies for these channels. To send a message from a remote machine, proxy sends this message over the TCP-connection to the node to which the original bidirectional channel is bounded. To receive a message on remote machine, the corresponding query is forwarded to the machine with the original channel, and the thread which issued this command is blocked until a reply message is received. A blocking mechanism is similar to one in Polyphonic C# that is used to handle the thread queues there.

The above implementation is a prototype one, so we use a simple decentralized approach to distribute computational resources amongst movable methods.

5 Conclusion

A distributed variant of an asynchronous parallel programming model of Polyphonic C# was demonstrated in the given work. The key notions of our approach are the movable methods and channels. One-directionality of the channels is overcome by explicit introduction of "bidirectional" channels.

Experiments with the prototype implementation demonstrate the easy readability, compactness and satisfactory effectiveness of program code in MC#.

Further lines of our work are to refine the type system for common and bidirectional channels and to test a decentralized distribution of computational resources in order to increase effectiveness of the whole system.

Acknowledgements

The authors wish to thank S. Abramov (PSI RAS, Russia) for his support of current work.

References

Abramov, S., and Adamovich, A. 1999. T-system: a programming environment with support of automatic dynamic parallelizing of programs (in russian). In *Program systems: Theor. found. and appl., Ed. A.C.Ailamazyan, Moscow, 1999*, 201–213.

Benton, N., Cardelli, L., and Fournet, C. 2002. Modern concurrency abstractions for c#. *To appear in ACM Trans. on Prog. Lang. and Systems*.

Fournet, C., and Fessant, F. L. 2002. Jocaml, a language for concurrent, distributed and mobile programming. In *Proc. of the 4th Summer School on Adv. Funct. Progr., Oxford, 19-24 August 2002.*