PARALLEL C#: THE USAGE OF CHORDS AND HIGHER-ORDER FUNCTIONS IN THE DESIGN OF PARALLEL PROGRAMMING LANGUAGES

Vadim B. Guzev
Peoples' Friendship University of Russia, Moscow, Russia

Abstract - *In this paper we introduce new parallel programming language Parallel C#, the main feature of which is the combination of chords and higher-order functions in one language. This language extends the standard syntax of C# language for the parallel programming needs and simplifies the task of writing complex multithreaded and distributed applications.*

We describe the design of the language and give examples of its use in addressing a range of concurrent programming problems. Also we introduce new Distributed Runtime Systems for this language both for Windows and Linux machines.

Keywords: parallel computing, distributed computing, concurrent, HPC, Parallel C#.

1 Introduction

In order to solve bigger scientific problems we need bigger computers capable of performing a large number of operations per second. The era of increasing performance by simply increasing clocking frequency now seems to be behind us and nowadays computer designers are starting to adopt the scale-out systems. In such systems computational capacity can be easily increased by adding new nodes to existing ones and connecting them via high-speed communication network. However this approach also posed a problem of developing complex and at the same time reliable software systems and programming languages that could effectively exploit the potential of concurrent distributed computing and easily expand to any given number of processors or computers.

The purpose of this paper is to present the current results of development of new parallel programming language Parallel C# [1]. This language can be considered as a new branch of MC# language [2, 3] which in turn is the parallel extension of wide-spread C# programming language. Unlike MC# language where channels and channel handlers are widely used, Parallel C# combines both chords and higher-order functions, which makes it possible to get rid of such complicated notations as "*channel*" and "*channel handler*". In this document we review the syntax of the language and provide some program examples written in Parallel C#.

2 Language description

Parallel C# is based on the idea according to which programmer should concentrate on correct structuring of the program, using the parallelism and locality properties of the functions. In Parallel C#, Distributed Runtime System (which can be optimized for different types of platforms) takes care about effective resource planning, communication protocols, objects serialization/deserialization and etc.

In addition to standard C# language grammar four syntactic constructs were added in Parallel C#:

1) Spawning new thread on local application domain (via asynchronous functions);

2) Spawning new thread on remote application domain (via movable functions);

3) Higher-order functions and generation of proxy functions;

4) Synchronization of asynchronous and synchronous methods via chords.

It turned out that presence of such constructs is more than enough for convenient compilation of parallel distributed applications of any complexity. Note that abovementioned syntax enhancements can be easily built into the syntax of numerous other languages such as Java, Nemerle and others.

2.1 Asynchronous functions

Conventional methods which are commonly used in such languages as C# are synchronous by their nature. The caller is waiting until the execution of the called method is completed, and then resuming its execution. In the world of parallel computing execution time decrease can be achieved by simultaneous execution of several code segments on different processors. I.e. any parallel programming language has to have syntactic primitives which spawn method in new thread and proceed to the next instructions. These are known as "asynchronous" functions.

In Parallel C# language async keyword is used for the purposes of asynchronous functions identification. To keep it simple you can consider async as the analogue of void keyword, which additionally applies the parallelism property to the function. Here you can see the example of asynchronous function declaration (with one parameter) in C# and Parallel C#:

C# language:

```
class ObjectWrapper {
 int x;
 public ObjectWrapper( int y ) { this.x = y; }
 public void afun() { Console.WriteLine( x ); }
}
…
ObjectWrapper ow = new ObjectWrapper( 5 );
Thread t = new Thread(new ThreadStart(ow.afun));
t.Start();
```

Parallel C# language:

```
async afun( int x ) {
 Console.WriteLine( x );
}
…
afun( 5 );
```

2.2 Movable functions

Asynchronous methods described above allow programmers spawn new threads in the same application domain where the caller function resides. However, Parallel C# was designed as a language, which should allow programmer disengage oneself from the computing platform. Programs written in Parallel C# language can be executed in local mode, as well as in distributed mode on cluster, several clusters (metacluster) or GRID networks. It worth noting that in Parallel C#, Distributed Runtime System (DRS) determines which node of the computing system is better suited for the execution of newly spawned movable function. Depending on the physical structure of distributed computing system different types of DRS can be used. In each DRS implementation the node selection algorithms can be optimized for the particular physical structure of the system.

To make the Runtime System understand that some particular function should be spawned on different node of the cluster/metacluster/GRID network all you need to do is to add modifier movable to this function. To keep it simple you can consider this modifier as analogue of async keyword, which additionally applies distributed property to the function. Thus movable functions cannot return any results directly - all interaction between movable functions is organized via creating "proxies" of the functions passed as parameters. Here is an example of movable function declaration:

```
movable mfun( int x ) {
 // mfun's business logic
}
mfun( 7 );
```

When movable function is called all objects passed as its parameters are automatically serialized and sent to the recipient computer. And what is more, if movable function is not marked as static then original object which this movable function belongs to is copied as well. In this case all changes made to the copy of the object by movable function have no influence on the original object.

2.3 Higher-order functions

Unlike C#, in Parallel C# all functions are higher-order functions, i.e. they can be used like any other object (for example passed as parameters to other functions). The following example demonstrates this feature:

```
class A {
 // This field is a pointer to a function which accepts
 // int and returns long
 public (int) => long fun1;
}

class B {
 public long func( int x ) { return x * x; }
 public fun2() {
  A a = new A();
  // Object initialization of "function" type
  a.fun1 = this.func;
  Console.WriteLine( a.fun1( 2 ) ); // Writes "4"
  () => long function = this.func; // Local function pointer
  Console.WriteLine( function( 2 ) ); // Writes "4"
 }
}
```

This example contains the declarations of two classes - *A* and *B*. Class *A* contains only one field - function object which accepts integer value as its only parameter and returns value of type *long*. Class *B* has declaration of function *func*, which should contain some business logic (in our case it is quite simple - it just returns squared value of its parameter). In method *fun2*, right after object *a* instantiation, property *a.fun1* is initialized by function *func*. Then this function is called as *a.fun1(2)*.

In Parallel C# language you can pass function descriptions (both synchronous and asynchronous) as parameters to other synchronous and asynchronous functions:

```
async fun( int x, (int) => async func ) { func( x ); }

async fun2( int x ) {
   Console.WriteLine( x );
}
```

Calling method fun:

```
void main() {
   fun( 7, fun2 );
   // business logic of main function
}
```

In this example we define function *fun*, which accepts two parameters:

1) Integer number *x*;
2) Asynchronous function *func*, which accepts as its only parameter integer value

All that does this function *fun* is applying value passed as first parameter to a function passed as second parameter. As a result this particular example should print number *7*.

2.4 Threads synchronization

Often in order to continue calculations you need to wait for results from several functions (including movable functions) to get ready. Parallel C# is using "*chords*" (bounds, joins) for such synchronization of different threads. This notion was taken from Polyphonic C# [4] language. In this language, as well as in Parallel C#, chords are used as synchronization patterns to synchronize different threads.

The basic rule of correct chord definition: chord always consists of only one body, not more than one synchronous method and at least one asynchronous method. This rule has one consequence: synchronous method can participate in chords only with asynchronous methods.

The body of the chord is executed iff all methods declared in the chord were called. The body of the chord must return the result of the same type as the return type of synchronous method in this particular chord. Consider an example:

```
class Buffer {
   string Get() & async Put( string s ) {
      return s;
   }
}
```

The code above defines a class *Buffer* declaring two instance methods which are defined together in a single chord. The first method *string Get()* is a synchronous method taking no arguments and returning a string. The second method *async Put(string s)* is asynchronous (so returns no result) and takes a string argument.

If *buff* is an instance of *Buffer* and one calls the synchronous method *buff.Get()* then there are two possibilities:

- If there has previously been an unmatched call to *Result(x)* (for some string *s*) then there is now a match, so the pending *Result(x)* is de-queued and the body of the chord runs, returning *x* to the caller of *Get()*.

- If there are no previous unmatched calls to *buff.Put(.)* then the call to *buff.Get()* blocks until another thread supplies a matching *Put(.)*.

Conversely, on a call to the asynchronous method *buff.Put(s)*, the caller will never wait but there are two possible behaviors with regard to other threads:

- If there has previously been an unmatched call to *buff.Get()* then there is now a match, so the pending call is de-queued and its associated blocked thread is awakened to run the body of the chord, which will return *s*.

- If there are no pending calls to *buff.Get()* then the call to *buff.Put(s)* is simply queued up until one arrives.

As opposed to MC#, in Parallel C# it is possible to declare static chords - this makes additional "*functional*" and "*nonfunctional*" keywords used in MC# unnecessary in Parallel C#. For example:

```
static string Get() & static async Put( string s ) { return s;}
```

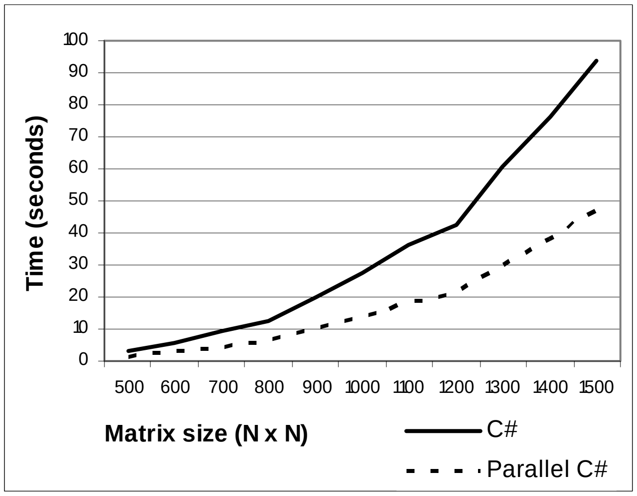
3 Examples of Parallel C# programs

Let's consider a task about multiplication of matrixes *A* and *B*, both of given dimensions *N* x *N*. Let's name the resulting matrix as *C*. Because different elements of matrix *C* can be calculated separately, in our program we'll calculate parts of resulting matrix *C* concurrently in different threads. We'll implement the algorithm which finds the part of matrix *C* in function *Multiply*. We'll also add special asynchronous method *stop()* which will be bounded with synchronous method getStop() - this chord will be used by the main program to wait for the parts of matrix *C* to get ready. If we suppose that this function should be executed on different processors of the given computer we can describe it as asynchronous method:

```
public async Multiply( int N, double[,] A,
                              double[,] B, double[,] C,
 int from, int to, () => async stop ) {
 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];
 stop();
}
```

And here is the main program:

```
public class Program  {
 public static void Main( string[] args )  {
   int N = System.Convert.ToInt32( args [0] );
   double[,] A = new double [N, N], B = new double [N, N],
              C = new double [N, N];
   ReadMatrix( A, B );
   Program p = new Program();
   int  N2 = N / 2;
   p.Multiply( N, A, B, C, 0,  N2, p.stop );
   p.Multiply( N, A, B, C, N2, N, p.stop );
   p.getStop();  p.getStop();
   WriteMatrix(C);
 }
 public void getStop() & async stop() { return; }
}
```



Pic 1 Matrix multiplication using one and two cores of the Intel Core 2 CPU 6400 @2.13GHz

On picture 1 you can see the relationship between the times needed to multiply matrixes and the dimensions of these matrixes for both C# and Parallel C# versions of the program described above.

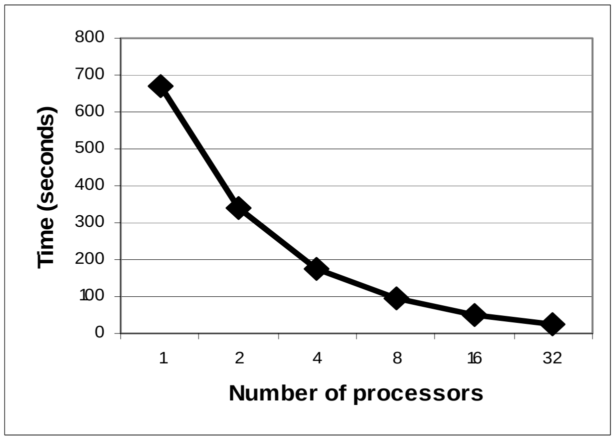
Now let's consider a task of finding the number of words occurrences in the given text. The search of each group of words will be realized in parallel on cluster. As a basis of the solution we'll use Map-Reduce paralleled/distributed model [5]. Let's describe movable method *Map*, which will read the text and search the given word and method *Reduce* which in turn will receive the results from running *Map* methods.

```
public class Program {
 static void Main( string[] args ) {
  Program p = new Program();
  // the number of processors in the cluster
  int np = CommWorld.Size;
  string[] words = ReadTheWords( args [0] );
  int n = Math.Min( words.Length, np );
  int portion = words.Length / n;
  for ( int i = 0; i < n; i++ ) { // sending a portion of words
    string[] part = new string [portion];
    Array.Copy( words, i * portion, part, 0, portion);
    p.Map( args [1], part, p.Reduce );
  }
  for ( int i = 0; i < n; i++ ) p.Pulse();
  PrintTheResult( p.dic );
 }

 movable Map(string fileName, string[] words,
                  (string[], int[]) => async Reduce) {
  int[] counts = new int [words.Length];
  using ( FileStream file = new FileStream(
   fileName, FileMode.Open, FileAccess.Read ) )  {
   TextReader tr = (TextReader) new StreamReader( file );
   string line = tr.ReadLine();
   while ( line != null ) {
    for ( int j = 0; j < words.Length; j++ )
     counts[j] += Regex.Matches( line, words[j] ).Count;
    line = tr.ReadLine();
   }
  }
  Reduce( words, counts );
 }

 void Pulse() & async Reduce( string[] words, int[] counts )
 {
  for ( int i = 0; i < words.Length; i++ )
   dic.Add( words [i] , counts [i] );
 }
 Hashtable dic = new Hashtable();
}
```

Please note that in this particular example to simplify the code we suppose that the number of used processors and the number of words are equal to powers of two. Also it worth noting that MapReduce model relies on distributed file system, which makes the source text available on all nodes of the cluster and thus eliminates a need of its manual sending from main node of the cluster to the others.



Pic 2 Search for 32 words in text (64 Mb).

On picture 2 you can see the graph which shows the relation between the times needed to count the number of occurrences of 32 words in 64Mb text and the number of processors used. The measurements were made on SKIF cluster with the following configuration: 16 nodes, where each node has 2 processors AMD Athlon(TM) MP 1800+ and 896 Mb memory.

4 Conclusions

Parallel programming is becoming more important and wide-spread nowadays, but it is still extremely hard. One of the seminal achievements in this area is the introduction of an asynchronous parallel programming model within the Polyphonic C# programming language in the context of Microsoft .NET platform [4]. The programming model we proposed in this document extends asynchronous programming model accepted in Polyphonic C# for the needs of distributed parallel computing and we believe that it is much clearer than the one used in MC# language.

Currently we've implemented two types of Runtime Systems which are available on the project site [1]:

a) **Parallel C# Many Core Edition** - programming system that allows compile and execute Parallel C# programs in local mode on Windows machines. This version was created to simplify the debugging process of parallel programs on local machines prior to deploying to real clusters.

b) **Parallel C# Cluster Edition** - programming system that allows compile and run Parallel C# programs on Linux-based clusters.

Now we're working on new MetaCluster Edition which will allow us execute Parallel C# programs on several clusters at the same time. Also we're planning to add Visual Studio IDE support for Parallel C# in the future.

5 References

[1] Parallel C# project site: http://www.parallelcsharp.com

[2] MC# project site: http://u.pereslavl.ru/~vadim/MCSharp

[3] Yury Serdyuk. "MC# 2.0: a language for concurrent distributed programming on .NET", .NET Technologies 2006, Plzen, Czech Republic

[4] Benton N., Cardelli L., Fournet C. "Modern Concurrency Abstractions for C#", ACM Transactions on Programming Languages and Systems, v.26, №.5, 2004, pp. 769-804.

[5] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters"

[6] "Report on the Experimental Language X10", http://labs.google.com/papers/mapreduce-osdi04.pdf

[7] John Gough. "Compiling for the .NET Common Language Runtime (CLR)", Prentice Hall PTR, 2002