# MC# 2.0: a language for concurrent distributed programming based on .NET

Yury Serdyuk
Program Systems Institute of
Russian Academy of Sciences
Russia (152020), Pereslavl-
Zalessky
Yury@serdyuk.botik.ru

## ABSTRACT

In this paper, we introduce a new version of MC# — a language for .NET-based concurrent distributed programming. This language is an adaptation of the basic idea of the Polyphonic C# language (Benton N., Cardelli L., Fournet C., Microsoft Research Laboratory, Cambridge, UK) for the case of distributed computations.

We present the background and goals of developing the language and introduce its novel constructs : movable methods, channels and handlers. We describe the specific features of MC# and formulate differences between its current and previous versions. Examples of programming in MC# are given: a program for finding prime numbers by Eratosthenes sieve, and a program named *all2all* which demonstrates interaction between distributed processes. In conclusion, we give a brief description of the current implementation along with the list of applications that have been developed, and identify directions for future work.

### Keywords

Concurrent distributed programming, MC#, movable methods, channels, handlers, Runtime-system, .NET.

## 1. INTRODUCTION

The wide use of computer systems with massive parallelism, such as multicore processors, clusters and Grid-architectures, posed again the problem for developing high-level, powerful and convenient programming languages that would allow one to create complex and at the same time reliable software systems that efficiently use the possibilities of concurrent distributed computations and are easily scalable to a given number of processors, nodes or computers.

Currently available program interfaces and libraries for organizing parallel computations, such as OpenMP [OpenMP] ( for systems with shared memory) and MPI (Message Passing Interface) [MPI] (for systems with message passing), have been implemented for C and Fortran languages, and hence are very low-level and inadequate for modern object-oriented programming languages like C++, C# and Java. Additionally, such interfaces rely on the use of libraries rather than on appropriate programming language constructs.

In general, a modern high-level programming language consists of two parts:

1) basic constructs of the language itself, and

2) a collection of specialized libraries accessible through appropriate APIs (Application Programming Interfaces).

New demands on increased programmers productivity (achieved through a higher abstraction level of language constructs, among other things), as well as on reliability and security of programs they develop, account for a tendency to transfer key concepts of most important APIs into the corresponding native constructs of programming languages.



***.NET Technologies 2006***
*FULL papers conference proceedings*
*ISBN 80-86943-06-2*

For example, the embedding of asynchronous methods and chords into Polyphonic C# [BCF04], which is an extension of the C# language, allows one to use it without the System.Threading library, which is normally required to implement multithreaded applications on top of .NET. On the other hand, the introduction of new data type constructors (for streams, anonymous structures, discriminated unions and others) along with appropriate query definition tools into Cω language [BMS05] renders obsolete the ADO.NET data subsystem (specifically, the traditional System.Data and System.XML libraries intended to handle relational and semistructured data).

We suggest that the next step in this direction be to introduce high-level constructs for creating concurrent distributed programs into the object-oriented language, and thus to free the programmer from the need to use the System.Remoting library (and, in many cases, also the System.Threading library), which is required to develop conventional distributed applications using C#.

From the practical point of view, the goal pursued by the developers of MC# was to design a language for industrial concurrent distributed programming which is going to involve more and more human resources, with the oncoming age of multicore computations. This language aims to replace C and Fortran languages in this area. It allows to create complex software systems that have satisfactory effectiveness when executed on parallel architectures. The choice C# as a basic language gives the possibility of using a modern object-oriented programming language equipped with rich libraries (like libraries for Web-application development, specifically, for dealing with Web-services, designing graphical applications, implementing systems with a high degree of security etc.), and, at the same time, to eliminate such low-level and unsafe features as C pointers which dramatically decrease programmer's productivity and the reliability of software systems. In this regard, our approach coincides with that taken in the development of the X10 language [SJ05], which is oriented towards "non-uniform cluster computing".

In MC# language, in contrast to using MPI interface, there is no need to distribute computational processes over cluster nodes explicitly (though such possibility also is provided by the language) – it is enough only to identify which functions (methods) can be executed concurrently. Moreover, in MC# language the new computational processes can be created and distributed over accessible nodes during program execution dynamically (X10 language also provides for that possibility for "activities"), that is impossible for MPI-programs. Similarly, there is no necessity to code by hand an object (data) serialization preparing moving them to remote node or machine — the Runtime-system performs an object serialization/deserialization automatically.

In fact, MC# language is an adaptation of the basic idea of the Polyphonic C# language (more precisely, of the basic idea of the join calculus [FG02]) for the case of concurrent **distributed** computations. As a matter of fact, the authors of the Polyphonic C# language presumed that asynchronous methods would be used either on a single computer or on a set of machines where they have been fixed and interact through the remote method call tools provided by the .NET Remoting library. In the case of MC#, the execution of an autonomous asynchronous method can be scheduled on a different machine selected either of two ways: by explicit indication by the programmer (which is not a typical case) or automatically (in this case, usually a cluster node or machine in the Grid network with the least workload is selected). Interaction of asynchronous methods that are executed on different machines is implemented through message passing using channels and channel message handlers. In MC#, channels and handlers are defined using chords in the Polyphonic C# style.

Channel message handlers are a new feature of MC# 2.0 as compared to the previous version of the language [GS03]. The second significant distinction consists in a different semantical treatment of channels and handlers (see the third key feature of MC# language in Section 2.1 and a forthcoming paper [S06]).

The paper is organized as follows. Section 2 describes the novel constructs of the MC# language — movable methods, channels and channel message handlers. In Section 3, we demonstrate how MC# constructs can be applied to develop two concurrent distributed programs — finding prime numbers by Eratosthenes sieve and *all2all* program demonstrating interaction of distributed processes. In Section 4, we give details about the current MC# implementation, which consists of a compiler and a Runtime-system. We provide conclusions and directions for the future work in Section 6.

## 2. NOVEL CONSTRUCTS OF MC#: MOVABLE METHODS, CHANNELS AND HANDLERS

In any sequential object-oriented language, conventional methods are synchronous: the caller always waits until the method called is completed, and only then continues its work.

The key feature of Polyphonic C# (which, in fact, became a proper part of the Cω language — and from

now on we will refer only to the latter) is the introduction of so called "asynchronous" methods in addition to conventional synchronous methods. Indeed, such asynchronous methods are intended for playing two major roles in programs:

1) the role of autonomous methods implementing the concurrent parts of the basic algorithm and executed in separate threads, and

2) that of the methods intended for delivering data (possibly, with preliminary processing of it) to conventional, synchronous methods.

In the MC# language, these two kinds of methods form two special syntactic categories of:

1) movable methods and

2) channels

respectively.

In Cω, auxiliary asynchronous methods used for data delivery are usually declared together with synchronous methods. In MC#, the latter are represented as another special syntactic category that includes **channel message handlers** (**channel handlers** or even **handlers** for short).

## 2.1 Movable methods

Writing a parallel program in MC# language reduces to labeling with the special keyword **movable** the methods that may be transferred to other machines for execution:

modifiers **movable** method_name ( arguments ) {

< method body>

}

In MC#, movable methods are the only way to create and run the concurrent distributed processes. A consequence of the mentioned above properties of the movable methods is that

1) method call completes almost immediately (time is spent only on transferring the needed data to the remote machine),

2) movable methods never return a result (for interaction of movable methods among them and with other parts of the program, see Section 2.2 "Channels and handlers").

Correspondingly, by the rules of correct definition, movable methods:

- may not have a **static** modifier, and

- never use a **return** statement.

The movable method call has two syntactical forms:

1) object_name.method_name ( arguments )

- in this case, the Runtime-system selects the execution location for a given movable method automatically, and

2) machine_name@object_name.method_name

( arguments )

- in this case, the execution location is indicated by the programmer explicitly.

Worth to note is that the objects created during an MC# program execution are **static** by their nature: once created, they don't move and remain bound to the place (machine) where they were created. In particular, it is on this machine that they are registered by the Runtime-system, which is necessary for delivering channel messages to those objects.

**The first key feature of MC# language** (or, more precisely, of its semantics) is that, in general, during a movable method call, all necessary data, namely

1) the object itself to which the given movable method belongs, and

2) arguments (both objects and scalar values) for the latter

are only **copied** (but not moved) to the remote machine (in **nonfunctional** mode – see below). As a consequence, changes made afterwards to the copy will not affect the original object.

In particular, if a copied object has channels or handlers, they also are copied to the remote machine — they become "proxy" tools for the original objects (see Section 2.2 for details).

There are two modes of parallelizing MC# programs: "functional" and "nonfunctional" (or objective), and the choice will, in the end, affect the efficiency of program execution. These modes are defined by the modifiers **functional** and **nonfunctional** in the movable method declaration (the default value is **functional**).

In the functional mode, an object for which a movable method is called, is not transferred to a remote machine (i.e., all needed data are passed to the movable method through its arguments). Conversely, by specifying the **nonfunctional** modifier, we force the object to be moved to the remote machine.

The use of MC# on cluster architectures, which typically consist of the frontend machine and the subordinate nodes, is specific in that the names for both the frontend and the node are to be specified if a movable method is being called under explicit indication of execution location:

machine_name : node_name@o.m ( args )

Movable methods in MC# are similar to "activities" in X10. In the latter, asynchronous activities are

created by a statement ***async ( P ) S***, where *P* is a place expression and *S* is a statement. In contrast to MC# language with a "method level" concurrency, it is possible for multiple activities to be created in-line in a single method in X10.

## 2.1 Channels and handlers

Channels and channel message handlers are the tools to support the interaction of distributed objects.

Syntactically, channels and handlers are declared using chords in the Cω style. In the following example, the channel *sendInt* for transferring single integers is defined along with the corresponding handler *getInt*:

**CHandler** getInt **int** () & **Channel** sendInt ( **int** x )

{ **return** ( x );}

In such declarations, handlers have the following general format:

modifiers **CHandler** handler_name

return_type (args)

We can also declare a channel or a group of channels without a handler. In this case, we can use values being received by the channel through the global variables.

By the rules of correct definition, channels cannot have a **static** modifier, and so they are always bound to some object much in the same way as ordinary methods:

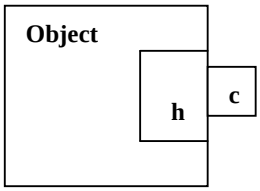


**Figure 1. An object with channel *c* and handler *h***

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

a.sendInt ( x ),

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

A handler is used to receive values from its jointly defined channel (or group of channels). For example, to receive a value from the channel *sendInt* we need to write

**int** m = a.getInt ( )

If, by the time a handler is called, the 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 selected during previous calls to the handler), then the call blocks. After receiving a value from the corresponding channel, the body of the chord (which may consist of arbitrary computations) runs and returns the result value to 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 channel queue, where all the values coming with multiple sendings to this channel are accumulated.

It is worth to note that separate methods (handler or channels) from the chord are typically called from different threads of which the entire concurrent distributed program consists.

Similarly to Cω, it is possible to define several channels in a single chord. This is a major tool for synchronizing the concurrent processes in MC:

**CHandler** equals **bool** () & **Channel** c1 ( **int** x )
& **Channel** c2 ( **int** y ) {
**if** ( x == y ) **return** ( **true** );
**else** **return** ( **false** );
}

Thus, a general rule for chord triggering is the following: the body of a chord is executed only after **all** methods declared in the chord header have been called.

The above example illustrates the case of a single handler for multiple channels:

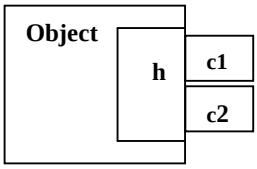


**Figure 2. An object with a single handler for multiple channels**

It is also possible to declare a channel shared by several handlers:

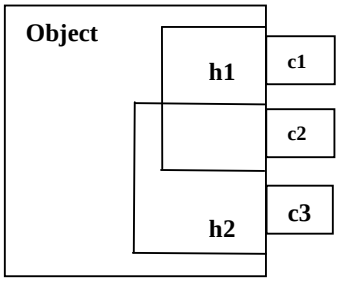


**Figure 3. An object with a "shared" channel**

So, once we have the values in both channels *c1* and *c2*, handler *h1* can be triggered. Similar is the case for channels *c2* and *c3* and handler *h2*. In general, all this together leads to non-determinism in program behaviour.

**The second key feature of MC# language** is that the channels and handlers can be passed as arguments to the methods (in particular, to the movable methods) **separately** from the object to which they belong (in this sense, they are similar to the pointers to methods or, in C# terms, to the **delegates**).

**The third key feature of MC# language** is that if channels or handlers were copied to a remote site (by which we mean a cluster node or a computer in the Grid-network) autonomously or as part of some object, then they become proxy objects, or intermediaries for the original channels and handlers. And the point here is that this replacement is hidden from the applied programmer — he can use the passed channels and handlers (in fact, their proxy objects) on the remote site as the original ones: as usual, all actions over the proxy objects are transferred to the original channels and handlers by the Runtime-system. In this sense, channels and handlers are different from ordinary objects: manipulations over the latter on a remote site are not transferred to the original objects (see the first key feature of MC# language).

Fig. 4 and Fig. 5 schematically demonstrate the passing and use of channels and handlers on a remote site. The subscripts in the channel and handler names denote the original site where they were created.

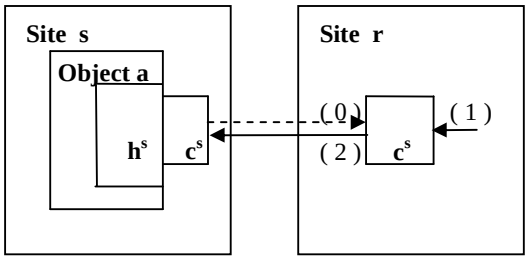


**Figure 4. Message sending by remote channel:**

(0) copying of the channel to remote site,

(1) message sending by (remote) channel,

(2) message redirection to the original site.

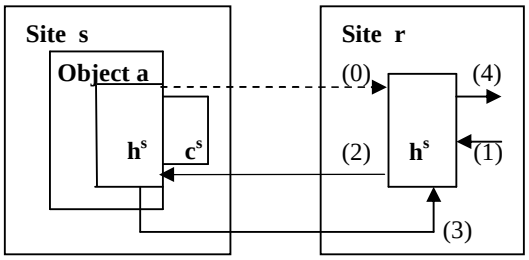


**Figure 5. Message reading from remote handler:**

(0) copying of the handler to remote site,

(1) message reading from (remote) handler,

(2) reading redirection to the original site,

(3) message return from the original site,

(4) result message return.

It turns out that these tools are enough to organize interaction of arbitrary complexity between the concurrent distributed processes.

In MC#, distributed processes can interchange arbitrary objects using channels and handlers. In X10, data interchange between places is realized through explicit spawning of asynchronous activities. So, if some thread wants to get a remote value *v*, it must create two activities:

```
final place origin = here;
finish async ( v ) = {
    final int x = v;
    async ( origin ) y = x;
}
```

In contrast to this, MC# Runtime-system hides from the programmer the spawning of auxiliary threads during message passing (see the example programs in the next Section).

## 3. PROGRAMMING IN MC#

In this Section, we will demonstrate the specific constructs of MC# language — movable methods, channels and handlers — and their semantic properties, on the example of two concurrent distributed programs.

First, we will build a parallel distributed program for finding prime numbers by the sieve method (also known as “Eratosthenes sieve”).

Given a natural number N, we need to enumerate all primes in the interval from 2 to N.

The sieving method is the following recursive procedure applied to the original list [2, … , N]:

1) select the head of the given list and output it to the resulting list of primes;

2) construct a new list by deleting from the given list all integers that are multiples of the head of this list;

3) apply the given procedure to the newly constructed list.

The main computational subroutine, which we called *Sieve* and the recursive calls to which will be distributed over a computer network, has two arguments: the handler *getList* to read the given list of numbers it will search for primes and the channel *sendPrime* to write the resulting list of primes. The end marker in both lists is -1.

An elementary step of unfolding the distributed computations (which consists of producing the next unit of the "conveyor" which sieves the integer stream) is sketched on Fig. 6.

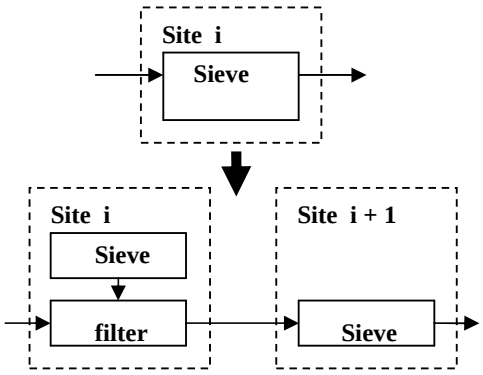


**Figure 6. Unfolding step in the distributed sieve method**

The full program text in MC# is given below. The original integer list [2, … , N] is sent on the channel *Nats* and the resulting list of primes is received from the channel *sendPrime* by the handler *getPrime*:

```
class Eratosthenes  {
 public static void Main (string[] args) {
  int N = System.ConvertToInt32 (args[0] );
  Eratosthenes E = new  Eratosthenes();
  new CSieve().Sieve ( E.getNat, E.sendPrime );
  for ( int n=2; n <= N; n++ )
    E.Nats ( n );
  E.Nats ( -1 );
  while  (  ( int p = E.getPrime() ) != -1 )
   Console.WriteLine ( p );
}
 CHandler getNat  int()  & Channel Nats ( int n )
{   return  ( n );  }
 CHandler getPrime  int()  & Channel sendPrime
   ( int  p )  {  return  ( n );  }
}
class  CSieve  {
 movable Sieve ( CHandler int() getList,
                 Channel (int) sendPrime ) {
  int  p = getList();
  sendPrime ( p );
  if  ( p != -1 )  {
   new CSieve().Sieve ( hin, sendPrime );
   filter ( p, getList, cout );
  }
}
 CHandler hin int()  &  Channel cout ( int x )
{ return  ( x ); }
 void filter (int p, CHandler int() getList,
                 Channel ( int ) cfiltered )  {
  while ( ( int n = getList() ) != -1 )
    if ( n % p != 0 ) cfiltered ( n );
  cfiltered ( -1 );
 }
}
```

The second program, called *all2all*, demonstrates how we can provide for interaction inside a set of distributed processes in accordance with the "all to all" principle.

Below, each distributed process is an object of the *DistribProcess* class. It starts on a remote site selected by the Runtime-system, by calling the *Start* movable method of the mentioned class.

In turn, each distributed process creates *BDChannel* (Bidirectional channel) object containing the channel *Send* and the handler *Receive*, on its own site. By interchanging *BDChannel* objects, distributed processes can send or receive messages to and from one another regardless of their physical location. *BDChannel* object interchange is realized through the main process which is executed on the machine where the application was started.

Below we present the full program text in MC# where the number N of distributed processes is given as the input parameter.

```
class  All2all  {
 public static void Main (string[] args)  {
  int  i;
  int  N = System.Convert.ToInt32 ( args [ 0 ] );
```

```
        // N is number of distributed processes
 All2all a2a = new All2all();
 DistribProcess dproc = new DistribProcess();
 //  Launch distributed processes
 for ( i = 0; I < N; i++ )
   dproc.Start ( i, a2a.sendBDC, a2a,sendStop );
 //  Receive BDChannel objects from processes
 BDChannel[] bdchans = new BDChannel [ N ];
 for ( i = 0; I < N; i++ )
   a2a.getBDC ( bdchans );
 //  Send BDChannel array to each process
 for ( i = 0; i < N; i++ )
    bdchans [ i ].Send ( bdchans );
 //  Receive stop signals from processes
 for ( i = 0; i < N; i++ )
   a2a.getStop();
}
 CHandler getBDC void(BDChannel[] bdchans)  &
     Channel sendBDC ( int i, BDChannel bdc )     {
   bdchans [ i ] = bdc;
}
 CHandler  getStop void() & Channel  sendStop() {
  return;
}
}
class  BDChannel  {
 CHandler  Receive object()
         & Channel Send (object obj )  {
          return  ( obj );
}
}
class DistribProcess  {
 movable Start ( int  i, Channel  (int, BDChannel)
                    sendBDC, Channel () sendStop ) {
 //  i is a process  proper number
 int  j;
 BDChannel  bdc = new BDChannel();
 sendBDC ( i, bdc );
 BDChannel[]  bdchans  =
    (BDChannel[]) bdc.Receive();
 //  Send messages to other processes
 for  ( j = 0; j < bdchans.Size; j++ )
  if  ( j != i )
   bdchans[j].Send ("Message from process " + i +
                    " to process " + j              );
 //  Receive messages from other processes
 for  ( j = 0; j < bdchans.Size; j++ )
  if  ( j != i )
   Console.WriteLine ( "Process " + i + " : " +
            (string) bdchans [ j ].Receive() );
 //  Send stop signal to the main program
 sendStop();
}
}
```

## 4. IMPLEMENTATION

All described above is the development and improvement of the ideas from [GS03]. Therein, the functions of the channel message handlers were shared by the synchronous methods in the chords and the special built-in objects, called "bidirectional channels". Below, we describe the current implementation based on bidirectional channels.

The implementation of MC# language consists of

1) a compiler from MC# to C#, and
2) a Runtime-system.

The compiler's main function is to replace movable methods calls by queries to the Runtime-system which schedules (selects a location of) execution for the methods. Translating the chords is conducted mainly in the same way as in Polyphonic C#, using bitmasks to mark the presence of received channel messages. Once a bitmask is filled up, received message content is extracted and the chord body execution starts. In this part of the compiler, the mechanism of monitors implemented in the .NET class *Monitor* is relied on heavily.

The MC# compiler performs two passes: at the first pass, it gathers information about channels declared by the chords and at the second pass, it emits C# code including, in particular, the needed objects and methods to deal with the channels. Specifically, the compiler is implemented using the ANTLR parser generation framework (http://www.antlr.org).

The main components of the Runtime-system are:

1) *Resource Manager* — a process implementing (currently, the simplest) centralized scheduling of resources (mainly, the cluster nodes) and running on the cluster frontend, and
2) *WorkNode* — a process running on each cluster work node.

Besides, there are *mcsboot* and *mcshalt* utilities to start and terminate the Runtime-system, correspondingly.

The main purpose of the *WorkNode* process is to accept the movable methods scheduled for execution on the given work node and to run them in separate threads. Before running, it deserializes the object associated with a movable method and the method's arguments. The *WorkNode* process has, as a component part, a *Communicator* process running in its own thread. *Communicator* is responsible for receiving and delivering the channel messages intended for objects located on the given node. For this purpose, all objects having channels (and handlers) are registered in a special table located on the node. Thus, a channel message has the following format to ensure proper message delivery:

< (IP-)address, *Communicator* port, object number,

channel name, message content >

The compiler and the Runtime-system run under both Windows and Linux. For the latter we use the Mono system (http://www.mono-project.com) — a free implementation of .NET framework for Unix-like systems.

By way of experiments, we have written a large series of parallel programs in MC#, such as calculation of Mandelbrot set (fractals), 3D rendering, Web search through the Google Web-service, radar-tracking signals processing, solving computational molecular dynamics tasks, etc. Running these tasks on the cluster, we used up to 96 processors. For all mentioned applications, we got an easy to read and compact code and satisfactory results in terms of the efficiency of parallelizing. The graph on Fig. 8 shows the relationship between the processing time (in sec.) for a 40 Mb input file and the number of processors in the radar-tracking signal processing task. The tests were conducted on the "SKIF K-1000" cluster (98$^{th}$ in Top500, November 2004) of the United Institute of Informatics Problems, National Academy of Sciences of Belarus.

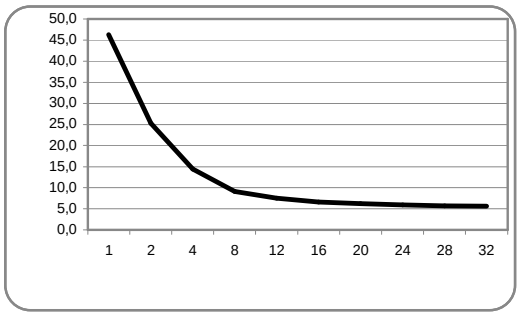


**Figure 8. Processing time for 40 Mb radiohologram**

## 5. CONCLUSIONS

This work presents an extension of C# language with the high-level features for concurrent, distributed programming based on the asynchronous programming model of Polyphonic C#. It can be considered as a general-purpose language for practical industrial programming, which oriented towards creating complex parallel software systems intended to run on cluster architectures.

We built a prototype implementation of MC# language for Linux cluster and a network of Windows machines. (The MC# project site is at: http://u.pereslavl.ru/~vadim/MCSharp)

Our future work will focus on implementing the MC# language in full accordance with the ideas put forward in the paper. Along with that, we are going to develop a more efficient Runtime-system by implementing a decentralized scheduling of movable methods calls and providing support for modern fast interconnects (Infiniband, QsNet II). A version of MC# programming system for metacluster computations is under development.

## ACKNOWLEDGMENTS

The author wishes to thank Vadim Guzev and Alexei Molodchenkov for participating in the implementation of MC# programming system and applications for it.

## REFERENCES

[BCF04] Benton, N., Cardelli L., Fournet C. Modern Concurrency Abstractions for C#. ACM Transactions on Programming Languages and Systems, Vol.26, No.5, 2004, pp. 769-804.

[BMS05] Bierman, G., Meijer, E., Schulte, W. The essence of data access in Cω. ECOOP 2005, LNCS 3586, Springer, 2005. pp. 287-311.

[FG02] Fournet, C., Gonthier, G. The join calculus: a language for distributed mobile programming. In Proc. Applied Semantics Summer School, 2000. LNCS, Vol.2395, Springer, pp. 268-332.

[GS03] Guzev, V., Serdyuk, Y. Asynchronous parallel programming language based on the Microsoft .NET platform. PaCT-2003, LNCS, 2763, Springer, pp. 236-243.

[S06] Serdyuk, Y. A formal basis for the MC# programming language (to appear).

[MPI] Message Passing Interface: http://www-unix.mcs.anl.gov/mpi/

[OpenMP] OpenMP specifications: http://www.openmp.org/specs.

[SJ05] Saraswat, V.A., Jagadeesan R. Concurrent Clustered Programming, CONCUR 2005, LNCS 3653, Springer, 2005, pp.353-367.