mpC: a Multi-Paradigm Programming Language for Massively Parallel Computers

Alexey L. Lastovetsky
Institute for System Programming, Russian Academy of Sciences,
25, Bolshaya Kommunisticheskaya str., Moscow 109004, Russia
e-mail: lastov@ivann.delta.msk.su

Abstract

Currently, programming systems for distributed memory machines are limited to either task parallelism or data parallelism. The mpC programming language and its programming system support both task and data parallelism, allows both static and dynamic process and communication structures, enables optimizations aimed at both communication and computation, and supports modular parallel programming and the development of a library of parallel programs. The mpC language is an ANSI C superset. It is based on the notion of network *comprising processor nodes of different types connected with links of different lengths. The mpC language has facilities for description of network topology, for creating and discarding networks, for distribution data and computations over networks, for writing of functions that can be called on networks of the proper topology and so on. The paper presents basic principles underlying the mpC language and outlines its programming system.*

Introduction

To reveal the huge potential of performance that is hidden in distributed memory machines, programming systems supporting the development of efficient parallel programs in machine-independent manner are needed. Currently, basic tools for parallel programming for such machines are message-passing function extensions of C or Fortran, between which the PVM [1] and MPI [2] function extensions of these languages are the most popular now. In fact, for distributed memory machines a message-passing language is a programming language of assembler level, therefore it does not limit a style of parallel programming as well as the class of expressible parallel algorithms.

Although it is possible to write high efficient parallel programs in message-passing languages, most users find such programming to be incredible tedious. Therefore, a variety of programming systems facilitating the development of parallel programs are developed. But as the payment for the facilitation, such a programming system limits the class of message-passing programs it can yield.

The maximum facilitation could be provided by a programming system in which user writes a pure serial program, and it is responsibility of the compiler to synthesize automatically the good parallel program as a result of deep analysis of the serial program. Automatic program synthesis, especially synthesis of parallel programs, is a very difficult problem. Therefore, although very interesting results were obtained in this direction [3-8], they all are far from a practical use yet.

Currently, any programming system for distributed memory machines, where user specifies parallelism somehow, can be put in one of two classes depending on the parallel programming paradigm underlying it. Programming systems based on task parallelism and programming systems based on data parallelism cover the development of different universes of parallel programs.

There are a few task parallel interactive graphical programming systems such as HeNCE [9], CODE [10], or Meander [11], that allow the user to specify synchronization and communication between subroutines in a Fortran or C program. So, in the HeNCE graph, a node represents a process, which is an instantiation of some procedure, and each directed edge represents a communication or synchronization constraints. The HeNCE system then generates a C program with calls to PVM.

Another approach to task parallel programming systems is based on extended programming languages such as Fortran M [12] or Compositional C++ (CC++) [13].

Fortran M is a small set of extensions to Fortran that supports a modular approach to parallel programming, permits writing of provably deterministic parallel programs, and allows specification of dynamic process and communication structures. Fortran M provides constructs for creating tasks and channels, for sending messages on channels, and for mapping tasks to processors.

CC++ is a parallel programming language based on C++. CC++ introduces parallel block and parallel loop constructs for explicitly expressing structured parallel execution and the `spawn` construct for expressing unstructured parallel execution. CC++ has synchronization variables and atomic functions for synchronization. CC++ introduces the keyword `global` to specify processor objects corresponding to CC++ executables as well as global pointers for references between processor objects.

Data parallel programming systems are based on extended programming languages such as Fortran D [14-17], Vienna Fortran [18], High Performance Fortran (HPF) [19], CM Fortran [20], C* [21], Dataparallel C [22], Modula-2* [23], and so on.

Fortran D is a typical representative of Fortran based data parallel languages. In fact, Fortran D extends serial Fortran by three new statements for the specification of data distribution. The DECOMPOSITION statement specifies a virtual processor array. The ALIGN statement specifies the distribution of data array elements over virtual processors. The DISTRIBUTE statement specifies how virtual processors should be mapped to the actual processors. Sizes of virtual processor arrays as well as the number of actual processors are variables of compile time. So, the compiler has full information to generate the Fortran program with calls to message-passing primitives that is the definition of the process running on every actual processor. In contrast to the task parallel Fortran M,

Fortran D does not support dynamic process and communication structures as well as modular parallel programming. It supports only static creating tasks and does not support the separated compilation of modules. It is an obstacle to use Fortran D for writing libraries of parallel programs. The class of parallel programs which one can develop in the Fortran D programming system consists of parallelized sequential programs.

Dataparallel C is a C based counterpart of Fortran D. It does the same things but in the spirit of C. It introduces the `domain` statement which:

- specifies a virtual processor array;
- declares data objects distributed over the virtual processor array;
- specifies how the virtual processors should be mapped to actual processors.

Following the spirit of C, Dataparallel C does not introduce a global pointer. So, a distributed array is not a set of elements that may be located on different virtual processors, but the distributed object each component of which is an array.

Thus, currently there exist two classes of programming systems for distributed memory machines, each of which allows to implement its class of parallel algorithms. On the other hand, efficient implementation of many problems needs parallel algorithms that can not be implemented in pure data parallel or task parallel styles. Few examples of such algorithms can be found in [24]. Therefore, to use distributed memory machines efficiently, programming systems supporting both task parallelism and data parallelism are needed. Various approaches to this problem are possible. For example, Fortran M team declared that their goal is to integrate HPF and Fortran M in a single programming system. Another approach proposed in [24] consists in step-by-step adding data parallel constructs, such as the FORALL statement, to task parallel languages and task parallel facilities, such as nodal functions, to data parallel languages. But it seems that more drastic innovations are needed to decide this problem totally and to integrate adventages of task parallel and data parallel languages in a single parallel programming language.

The mpC programming language [25] was developed to support both task and data parallelism, to allow both static and dynamic process and communication structures, to enable optimizations aimed at both communication and computation, and to support modular parallel programming and the development of a library of parallel programs. The mpC language is an ANSI C superset. It is based on the notion of network comprising processor nodes connected with links. It has facilities for description of network topology, for creating and discarding networks, for distribution data and computations over networks, for writing of functions that can be called on a network of the proper topology and so on. This paper presents basic principles underlying the mpC language and outlines its programming system.

Computing space and network objects

When programming in C, user may imagine that there is the storage accessible to him, and he can manage this storage allocating data objects there. When programming in mpC, user may also imagine that there is some accessible set of virtual processors connected with links, and he can manage this resource allocating network objects there.

In mpC, the notion of *computing space* is defined as a set of typed virtual processors connected with links of different length accessible to user for management. There are three processor types: `memory`, `scalar`, and `vector`. A processor of the `memory` type can rather store data than operate on it. A processor of the `vector` type can perform vector operations efficiently. Finally, most common processors are of the `scalar` type. Besides, a processor of the `scalar` type has additional attribute characterizing its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from source processor to processor of destination. There exists not more than one directed link from source to destination. A link has an attribute named *length* which characterizes its relative speed of data transfer. A pair of opposite directed links between two processors may be considered a single undirected link.

The basic notion of the mpC language is *network object* or simply *network*. Network comprises processor nodes of different types connected with links of different lengths. Network is a region of the computing space which can be used to compute expressions and to execute statements.

Allocating network objects in the computing space and discarding them is performed in similar fashion as allocating data objects in the storage and discarding them. Conceptually, new network is created by a processor of an existing network when the corresponding network definition is interpreted. This processor is called a *parent* of the created network. The parent belongs to the created network. The only processor that exists from the beginning of program execution till program termination is a pre-defined *host-processor* of the `scalar` type.

Every network object declared in mpC program has a type. The type specifies the number and types of processors, links between the processors and their lengths, as well as separates the parent. For example, the type declaration

```
/* Line 1 */          net Rectangle {
/* Line 2 */              coord I=4;
/* Line 3 */              node {
/* Line 4 */                 I>=0: scalar;
/* Line 5 */              }
/* Line 6 */              link {
/* Line 7 */                 I>0:   [I]<->[I-1];
/* Line 8 */                 I==0:  [I]<->[3];
```

```
/* Line 9 */                }
/* Line 10 */               parent [0];
/* Line 11 */           }
```

introduces the network type named `Rectangle` that corresponds to networks consisting of four processors of the `scalar` type interconnected with undirected links of normal length in rectangular structure.

In this example, line 1 is a header of the network type declaration. It introduces the name of the network type.

Line 2 is a coordinate declaration declaring the coordinate system to which processors are related. It introduces the integer coordinate variable `I` ranging from 0 to 3.

Lines 3-5 are a node declaration. It relates processors to the coordinate system and declares their types. The line 4 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>=0`* ***then*** *processor of the `scalar` type is related to the point with the coordinate `[I]`.*

Lines 6-9 are a link declaration. It specifies links between processors. Line 7 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>0`* ***then*** *there exists undirected link of normal length connecting processors with coordinates `[I]` and `[I-1]`,* and line 8 stands for the predicate ***for all*** *`I<4`* ***if*** *`I==0`* ***then*** *there exists undirected link of normal length connecting processors with coordinates `[I]` and `[3]`.*

Line 10 is a parent declaration. It specifies that parent processor has the coordinate [0].

With the network type declaration, one can declare a network object identifier of this type. For example, the declaration

```
net Rectangle r1, r2;
```

introduces two identifiers `r1`, `r2` of network objects. Here, `net Rectangle` is a full name of the corresponding type.

Besides the network type, one can declare a parametrized family of network types called *topology* or *generic network type*. For example, the declaration

```
/* Line 1 */          net Ring(n) {
/* Line 2 */              coord I=n;
/* Line 3 */              node {
/* Line 4 */                 I>=0: scalar;
/* Line 5 */              }
/* Line 6 */              link {
/* Line 7 */                 I>0:  [I]<->[I-1];
/* Line 8 */                 I==0: [I]<->[n-1];
/* Line 9 */              }
/* Line 10 */             parent [0];
/* Line 11 */         }
```

introduces a topology named `Ring` that corresponds to networks consisting of `n` processors of the `scalar` type interconnected with undirected links of normal length in a ring structure. The header (line 1) introduces the integer parameter `n` of the topology `Ring`. Correspondingly, the coordinate variable `I` ranges from `0` to `n-1`, line 4 stands for the predicate ***for all*** *`I<n`* ***if*** *`I>=0`* ***then*** *processor of the `scalar` type is related to the point with the coordinate `[I]`,* and so on.

With the topology declaration, one can declare a network object identifier of the proper type. For example, the declaration

```
net Ring(4) r;
```

introduces the network type `Ring(4)` as an instance of the topology `Ring` as well as the identifier `r` of the network object of this type. Note, that the network types `Rectangle` and `Ring(4)` are essentially equivalent.

An instance of topology can be obtained not only statically but dynamically also. For example, the fragment

```
int i;
i=f();
{
  net Ring(i+1) rr;
  ... }
```

introduces the identifier `rr` of the network object the type of which is defined completely only during run-time.

A network object has a computing space duration that determines its lifetime. There are two computing space durations: static, and automatic. A network object declared with *static* computing space duration is created only once, conceptually just before its first usage. It exists till termination of the entire program. A new instance of a network object declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network object is discarded when execution of the block ends.

A declaration of a network object identifier specifies the scope and the linkage of the identifier and the computing space duration of the network object almost under the same rules that are used for specification of storage duration of data objects and scopes and linkages of their identifiers. For example, the following fragment of mpC file

```
net Ring(4) r4;
static net Ring(5) r5;
extern net Ring(6) r6;
int f(int k)
{
   net Ring(k) rk;
   static net Ring(k+1) rk1;
   ...
```

```
}
```

specifies that the identifier `r4` of the network object with static computing space duration has file scope and external linkage, the identifier `r5` of the network object with static computing space duration has file scope and internal linkage, the identifier `r6` of the static network object has file scope and external linkage, the identifier `rk` of the automatic network object has block scope, and the identifier `rk1` of the static network object has block scope.

Network object declaration that also causes computing space to be reserved for the network object named by an identifier is a *network object definition*. In the above fragment, except the declaration of `r6`, all the rest declarations are network object definitions. The parent of all these network object is the host-processor. The following example shows how you can specify another parent:

```
net Ring(10) r10;
net Ring (11) [r10:I==3] r11;
```

Here, the network `r10` has the host-processor as its parent, meantime the network `r11` has the processor of the network `r10` with the coordinate `[3]` as its parent.

Subnetworks

A new network object can be allocated not only in an unused computing space but also in the region of the computing space that already holds another network object. It can be done by explicit or implicit definition of a *subnetwork* of existing network object.

In contrast to an implicitly defined subnetwork, an explicitly defined subnetwork has a name introduced by the subnetwork declaration. A subnetwork declaration that causes computing space to be reserved for the subnetwork named by an identifier is just an explicit subnetwork definition. Computing space duration of explicitly defined subnetwork as well as scope and linkage of its identifier are specified in the same way as those for network objects, but the lifetime of the subnetwork should not exceed the lifetime of its supernetwork. The lifetime of an implicitly defined subnetwork is defined by compiler.

For example, the mpC file fragment

```
/*Line 1 */    net Web(m,n) {
/*Line 2 */        coord R=m, Fi=n;
/*Line 3 */        node {
/*Line 4 */            R==0&&Fi>0:  void;
/*Line 5 */            R==0&&Fi==0: fast scalar;
/*Line 6 */            default:     scalar;
/*Line 7 */        }
/*Line 8 */        link {
/*Line 9 */            R==0:        [0,0]<->[1,Fi];
/*Line 10*/            R>0:         [R,Fi]<->[R-1,Fi];
/*Line 11*/            Fi>0&&R>0:   long [R,Fi]<->[R,Fi-1];
/*Line 12*/            Fi==0&&R>0:  long [R,0]<->[R,n-1];
/*Line 13*/        }
/*Line 14*/        parent [0,0];
/*Line 15*/    };
/*Line 16*/    net Web(10,20) web10x20;
/*Line 17*/    subnet [web10x20: Fi%2==0] seastar10x10;
/*Line 18*/    int f(void)
/*Line 19*/    {
/*Line 20*/       subnet [web10x20: R<5] subweb5x20;
/*Line 21*/       static subnet [web10x20: R>=5] grid5x20;
/*Line 22*/       ...
/*Line 23*/    }
```

introduces the topology `Web` and defines the network object `web10x20` of the `Web(10,20)` type as well as its subnetworks `seastar10x10`, `subweb5x20` and `grid5x20`.

Here, in line 4, the keyword `void` in the position of the processor type indicates that no processors are related to the points with corresponding coordinates. The equivalent interpretation is that a processor of the `void` type has no memory and can execute no operations. The topology `Web` corresponds to web structure networks with n radial threads, each of them stringed with `m-1`normal speed scalar processors . In the center of the web a fast scalar processor is placed. It means that computational loading of this processor will be more intensive than those of the rest processors. Radial links between processors are of normal length, but circular links are longer. It means that data exchange through radial links will be more intensive than through the circular ones.

Line 17 is an explicit definition of the static subnetwork of the network object `web10x20` named by the identifier `seastar10x10` having file scope and external linkage. The construct `[web10x20: Fi%2==0]` specifies the processors of `web10x20` that constitute the subnetwork. Namely, a processor of `web10x20` with coordinates `[R, Fi]` belongs to the subnetwork `seastar10x10` if and only if `Fi%2==0`.

Similarly, line 20 is an explicit definition of the automatic subnetwork of the network object `web10x20` named by the identifier `subweb5x20` with block scope, and line 21 is an explicit definition of the static subnetwork of the same network object named by the

identifier `grid5x20` with block scope.

A subnetwork always inherits its supernetwork coordinate system. So, in the subnetwork any processor has the same coordinates as in its supernetwork. In addition, for any network or subnetwork so-called *natural numeration* of processors from `0` to `n-1`, where `n` is the number of processors, can be defined. The numeration is determined by lexicographic ordering on the set of coordinates of (non-void) processors. Evidently, a processor may have different natural numbers in the network and its subnetwork. The notion of natural number is used to set up a correspondence between processors of different subnetworks in distributed operations.

Distribution of data and computations

The notion of *distributed data object* is introduced in the spirit of C* and Dataparallel C. Namely, data object distributed through a region of the computing space comprises a set of components of any one type of undistributed data object allocated to processors so that every processor holds one component. So, a distributed data object is characterized by the type and attributes of the region through which it is distributed as well as the type and attributes of components. The lifetime of the distributed data object is limited by the lifetime of the region of the computing space.

A special type of a distributed data object is a *data object distributed through the entire computing space*. It means that creation of any network object includes creation of corresponding component of the data object on every processor of the network object.

Except some special cases, to declare an identifier designating a distributed object, it is necessary to place a specifier of the corresponding region of the computing space in the corresponding declaration just before the identifier. For example, the declarations

```
net Ring(11) Net1;
int [*]Derror, [Net1]Da[10], *[Net1:I<7]Dpi[5];
```

declare `Derror` as an integer data object distributed through the entire computing space, declare `Da` as an array of 10 `int`s distributed through the network object `Net1`, declare implicitly a subnetwork of the network object `Net1`, and declare `Dpi` as an array of 5 pointers to `int` distributed through this subnetwork.

In general, in mpC one can declare both distributed and undistributed data objects specifying precisely their locations - network object, subnetwork, or a single processor (for undistributed ones). For example, the declaration

```
int [web10x20:R==2&&Fi==3] x;
```

declares the undistributed data object `x` located on the processor of the network object `web10x20` with the coordinates `[2,3]`.

The notion of *distributed value* is introduced similarly. A value distributed through a region of the computing space comprises a set of components of any one type of undistributed value belonging to processors so that every processor holds one component.

An expression can be evaluated by the host-processor, by a single processor of a network, by a network or a subnetwork, or by the entire computing space. In the latter two cases, the expression is called a *distributed expression*. The value of a distributed expression may be also distributed. If so, the latter should be distributed through a subregion of the region of the computing space evaluating the expression. If an expression is evaluated by the entire computing space, it is called an *overall expression*. No other computations can be performed in parallel with evaluation of the overall expression.

A special type of a distributed expression called an *asynchronous expression* is introduced. Substantially, evaluation of an asynchronous expression needs neither data exchange between processors of the evaluating region of the computing space nor synchronization of the processors during its evaluation. The property of asynchronity of an expression is determined by the property of asynchronity of operators forming the expression. Most of operators of the mpC language are asynchronous in the sense that either both operands and the result belong to the same processor, or they both are distributed through the same region of the computing space, and the distributed operator is divided into a set of independent undistributed operators each of which is performed on corresponding components of the operands. If an expression is built only from such operators, and all they are distributed through the same region of the computing space, then the entire expression will be asynchronous.

A statement of the mpC language can be executed by the host-processor, by a single processor of a network, by a network or subnetwork, by a set of networks, or by the entire computing space. In the latter three cases, the statement is called a *distributed statement*. A set of distributed statements includes the sequential C statements extended with distributed data as well as the special parallel statements `fan`, `par` and `pipe` borrowed from [26]. If a statement is executed by the entire computing space, it is called an *overall statement*. No other computations can be performed in parallel with execution of the overall statement.

The notion of *asynchronous statement* is introduced. Execution of an asynchronous statement needs neither a data exchange between processors of the executing region of the computing space nor synchronization of the processors during its execution. In particular, if all expressions and substatements of a sequential statement are asynchronous and distributed through the same region of the computing space, then the statement is asynchronous. In this case, the distributed statement is divided into a set of independent undistributed statements each of which is executed by the corresponding processor using the corresponding data components.

Execution of an mpC program begins from a call of the function named `main` on the entire computing space.

The following very simple mpC program implements a parallel sum of two vectors:

```
/*Line 1 */    net Star(n) {
/*Line 2 */        coord I=n;
/*Line 3 */        node {
/*Line 4 */            default: scalar;
/*Line 5 */        }
```

```
/*Line 6 */          link {
/*Line 7 */              I>0: [0]<->[i];
/*Line 8 */          }
/*Line 9 */          parent [0];
/*Line 10*/      };
/*Line 11*/      #define N 100
/*Line 12*/      main()
/*Line 13*/      {
/*Line 14*/         int x[N], y[N], z[N];
/*Line 15*/         Input(x, y);
/*Line 16*/         {
/*Line 17*/            net Star(N) s;
/*Line 18*/            int [s]dx, [s]dy, [s]dz;
/*Line 19*/            dx=x[];
/*Line 20*/            dy=y[];
/*Line 21*/            dz=dx+dy;
/*Line 22*/            z[]=dz;
/*Line 23*/         }
/*Line 24*/         Output(z);
/*Line 25*/      }
```

Here, the undistributed arrays `x`, `y`, and `z` are allocated in the host-processor storage. Line 17 defines the automatic network `s` with the host-processor as a parent. Line 18 defines three automatic variables `dx`, `dy`, and `dz` all distributed through the network `s`. All the statements in lines 19-22 are distributed through the network `s`, but only the statement in line 21 is asynchronous.

Line 19 contains the unusual unary postfix operator `[]`. The point is that mpC is a superset of the vector extension of ANSI C named the C[] language [27], where the notion of *vector* defined as an ordered sequence of values of any one type is introduced. In contrast to an array, a vector is not a data object but just a new kind of value. In particular, the value of an array is a vector. The operator `[]` was introduced to support access to arrays as a whole. It has operand of the type "array of *type*" and blocks (forbids) conversion of the operand to pointer. So, the expression `x[]` designates the array `x` as a whole, and the expression `dx=x[]` scatters the elements of the array `x` to the distributed variable `dx` components.

Similarly, the statement in line 20 scatters the elements of the array `y` to the distributed variable `dy` components.

The statement in line 21 performs asynchronously addition of the distributed variables `dx` an `dy` and assigns the result to the distributed variable `dz`. Finally, the statement in line 22 gathers the distributed variable `dz` components to the array `z`.

Outline of the rest mpC facilities

When programming in mpC, networks are used first of all for adequate decomposition of user's problem. In fact, every virtual processor corresponds to a task that may run in parallel. Static and automatic networks allow to specify static and dynamic structures of interacting tasks. Networks also allow a user to specify the tasks in arbitrary manner mixing data and task parallelism in a most appropriate to his problem way. Besides, automatic networks allow to specify tasks living in disjoint intervals of time that allows compiler to place different tasks on the same actual processor without loss of efficiency. Finally, detailed network topology declarations are the very machine-independent specifications that allow compiler to map virtual processors optimally to actual processors.

Now, let us outline the mpC facilities that have remained out of scope of this paper. Structural task parallelism is supported by the `par` and `pipe` statements. The former specifies pure parallel execution of statements, and the latter is a kind of loop different iterations of which may be overlapped, being executed in parallel. These two together with the data parallel `fan` statement support structural parallel programming.

The notion of *distributed network object* was introduced to support nested parallelism. A definition of a distributed network object specifies the type of the network object and its parent network. Such definition may be considered as distributed through the parent network definition of a single network of the specified type. This notion implies the notions of *partially asynchronous expression and statement*. So, if an expression is evaluated by a distributed network, then although there is no data exchange and mutual synchronization between networks belonging to the set of networks specified by the distributed network, such exchanges and synchronizations may occur inside each of the networks. In this case, the expression is called a partially asynchronous expression in relation to the parent network of the distributed network. Similarly, if a distributed statement is executed by a set of pairwise disjoint networks, and execution of the statement needs nor a data exchange between these networks nor their mutual synchronization, and execution of the statement is both initiated and terminated by the network consisting of parent processors of these pairwise disjoint networks, then the entire statement is partially asynchronous (in relation to the network initiating and terminating its execution). For example, in the fragment

```
/* Line 1  */     net Ring(5) r5;
/* Line 2  */     main()
/* Line 3  */     {
/* Line 4  */        net Ring(3) [r5] r3;
```

```
/* Line 5  */        int x[5], [r5] dx, [r3] ddx;
/* Line 6  */        Input(x);
/* Line 7  */        dx=x[];
/* Line 8  */        ddx=dx;
/* Line 9  */        ...
/* Line 10 */    }
```

line 4 defines the network `r3` distributed through the network `r5`. In fact, `r3` is a set of `5` networks each of which contains `3` processors connected in ring,. Five parent processors of these 5 networks are also connected in a ring constituting the network `r5`. Line 5 defines the undistributed array x, the distributed through r5 variable dx, and the distributed through r3 variable ddx. The statement in line 7 scatters the elements of `x` to the components of `dx`. The statement in line 8 is partially asynchronous in relation to the network `r5` and for each of `5` single networks constituting `r3` broadcasts the corresponding component of `dx` to all components of `ddx` of the corresponding single network in parallel.

To support modular parallel programming as well as writing of a library of parallel programs, a few kinds of functions are introduced. A call of *basic function* is always an overall expression. Its arguments and value (if any) shall belong to the host-processor. In contrast to another kind of functions, it can define network objects.

Nodal function looks like nodal function in terminology of [24]. It can be executed completely by any one processor. Only local data objects of the executing processor can be created. In addition, the corresponding component of an externally-defined distributed data object can be used in the function.

Network function is called and executed on some network or subnetwork, and its value (if it returns a value) is also distributed through this region of the computing space. The header of the definition of the network function either specifies an identifier of static network or subnetwork, or declares an identifier of network being a special formal parameter of the function. In the first case, the function can be called only on the region of the computing space specified. In the second case, it can be called on any network or subnetwork of the suitable type. In any case, only the network specified in the header of the function definition can be used in the function definition body. No network can be declared in the body, but subnetworks of the network specified in the header can be declared. Only data objects belonging to the network specified in the header can be defined in the body. In addition, corresponding components of an externally-defined distributed data object can be used.

Outline of the compiler

The main function of the compiler is to transform an mpC program into a set of programs, each of which runs on its (virtual) processor, and which implement the computations specified by the initial mpC program interacting by means of message passing. Current implementation uses SPMD model of target program when programs executed by all processors have the same code. The portion of the code being executed by a particular processor is determined during run-time. The target language is the C[] language together with the library of the run-time support system. The compilation unit is a mpC file.

The compiler consists of a run-time support system (RTSS), a front-end, a transformer, a generator, and a tuner.

RTSS provides the operations on networks and subnetworks as well as message passing between virtual processors. It has a precisely specified interface and covers a particular communicating package (currently, MPI). It ensures platform-independence of the rest compiler components.

Front-end translates source mpC code into internal representation (a kind of attributed tree) and performs static semantic checking. Besides, for every network type or topology declaration it generates internal representation of functions used for computation of topological information.

Transformer has the internal representation as input and builds internal representation of the relevant C[] code that includes necessary calls of RTSS functions.

Generator has the internal representation of the C[] code as input and builds the corresponding C[] file.

Tuner rebuilds internal representation of mpC file to enable more efficient execution of the mpC program on a target parallel machine. The point is that if the tuner is off, the compiler generates the target program assuming that the number of processes constituting the target program is equal to the number of the virtual processors specified by the mpC program. This will cause execution of several processes by some processors if the target computer lacks them. The goal of the tuner is to reduce the number of processes constituting the program. It has the internal representation of the mpC file and the specification of merging virtual processors as input data. It transforms all distributed computations, executed by a set of virtual processors merged into a single processor, into equivalent undistributed computations (as a rule, into loops).

Currently, the compiler has been developed and is under implementation.

Acknowledgments

I am grateful to Dmitry Arapov (Keldysh Institute of Applied Mathematics, Russian Academy of Sciences), Alexey Kalinov, Ilya Ledovskih (Institute for System Programming, Russian Academy of Sciences), Ted Lewis (Naval Postgraduate School, USA), Hesham

El-Rewini (Nebraska University at Omaha, USA), and Christian Scheidler (Daimler Benz, Germany) for useful discussions. The work was supported by Russian Basic Research Foundation and Office of Naval Research (USA).

References.

1. V. Sunderam. PVM: A framework for parallel distributed computing. *Concurrency: Practice and Experience,* 2(4):315-339, 1990.
2. Message Passing Interface Forum. MPI: A Message-passing Interface Standard. *International Journal of Supercomputer Applications*, 8(3/4), 1994.
3. F. E. Allen, M.Burke, P. Charles, and R. Cytron. An overview of the PTRAN analysis system for multiprocessing. *Parallel and Distributed Computing*, 5:617-640, 1987.
4. F. Allen, M.Burke, R. Cytron, J. Ferrante, V. Sarkar, and W. Hseih. A framework for determing useful parallelism. *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.
5. M. Gircar, and C. Polychronopoulos. Automatic extraction of functional parallelism from ordinary programs. *IEEE Trans. on Parallel and Distributed Systems*, 3(2):166-178, March 1992.
6. C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B. Leung, and D. Schouten. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pp. 114-125, MIT Press, Cambridge Mass., 1990.
7. P. Banerjee, J. A. Chandy, M. Gupta, J. G. Holm, A. Lain, D. J. Palermo, S. Ramaswamy, and E. Su. The PARADIGM Compiler for Distributed-Memory Message Passing Multicomputers. In *Proceedings of the First International Workshop on Parallel Processing*, pp. 322-330, Bagalore, India, December 1994.
8. D. J. Palermo, and P. Banerjee. Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. Technical Report CRHC-95-09, University of Illinois, April 1995.
9. A. Beguelin, J. Dongarra, G. Geist, R. Manchek, and V. Sunderam. Graphical development tools for network-based concurrent supercomputing. *Proceedings of Supercomputing 91*, pp. 435-444, Albuquerque, New Mexico, November 1991.
10. J. C. Browne. Software engineering of parallel programs in a computationally oriented display environment. In D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pp. 75-94, MIT Press, Cambridge Mass., 1990.
11. G. Wirtz. The Meander Language and Programming Environment. *Proceedings of Second International Conference on Software for Multiprocessors and Supercomputers: Theory, Practice, Experience*, pp. 57-66, Moscow, Russia, September 1994.
12. I. Foster, and K. M. Chandy. Fortran M: a language for modular parallel programming. Preprint MCS-P327-0992, Argonne National Lab, 1992.
13. K. M. Chandy, and C. Kesselman. CC++: A Declarative Concurrent Object Oriented Programming Language. Technical Report CS-TR-92-01, California Institute of Technology, Pasadena, California, 1992.
14. G. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D Language Specification. Center for Research on Parallel Computation, Rice University, Houston, TX, October 1993.
15. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Evaluation of compiler optimizations for Fortran D on MIMD distributed-memory machines. *Proceedings of the ACM 1992 International Conference on Supercomputing*, pp. 1-14, Washington, DC, July 1992.
16. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 30(8):66-80, August 1992.
17. S. Hiranandani, K. Kennedy, and C.-W. Tseng. Preliminary Experiences with the Fortran D Compiler. *Proceedings of Supercomputing'93*, pp. 338-350, Portland, Oregon, November 1993.
18. B. Chapman, P. Mehrotra, and H. Zima. Programming in Vienna Fortran. *Scientific Programming*, 1(1):31-50, 1992.
19. High Performance Fortran Forum. *High Performance Fortran language specification, version 1.0*. Rice University, Houston, TX, May 1993.
20. The CM Fortran Programming Language. *CM-5 Technical Summary*, pp. 61-67, Thinking Machines Corporation, Nov. 1992.
21. The C* Programming Language. *CM-5 Technical Summary*, pp. 69-75, Thinking Machines Corporation, November 1992.
22. P. J. Hatcher, and M. J. Quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, Cambridge, MA, 1991.
23. M. Philippsen, and W. Tichy. Modula-2* and its compilation. *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, 1991.
24. P. J. Hatcher, and M. J. Quinn. Supporting Data-Level and Processor-Level Parallelism in Data-Parallel Programming Languages. *Proceedings of the 26th Annual Hawaii International Conference on System Sciences, Volume II, Software Technology*, pp. 14-22, IEEE Computer Society Press, Los Alamitos, CA, 1993.
25. A. L. Lastovetsky. The mpC Programming Language Specification. Technical Report, Institute for System Programming, Russian Academy of Sciences, Moscow, Russia, 1994.
26. T. G. Lewis. *Foundations of Parallel Programming: A Machine-Independent Approach.* IEEE Computer Society Press, 1994
27. S. Gaissaryan, and A. Lastovetsky. ANSI C Superset for Vector and Superscalar Computers and Its Retargetable Compiler. *Journal of C Language Translation*, 5(3):183-198, 1994.