Russian Academy Of Sciences

Institute for System Programming

The mpC Programming Language Specification

1994-1997

(last updated November 1997)

Table of Contents

1. Introduction

The mpC language was developed to support efficiently portable modular parallel programming for a wide range of distributed memory machines, especially, for heterogeneous networks of computers. The language is an ANSI C superset based on the notion of network comprising processor nodes of different types and performances, connected with links of different bandwidths. The user can describe a network topology, create and discard networks, distribute data and computations over the networks. The mpC programming environment uses the topological information in run time to ensure the efficient execution of the application on any uderlying hardware.

The mpC language is a superset of the C[] programming language. C[] is an ANSI C superset for vector and superscalar computers. It supports vector computations. While programmins in mpC, the user doesn't need to know C[] in details. To write good mpC programs one should first of all be familiar with operator [] allowing to specify send and receive buffers in communication operations.

It is very useful to learn sample mpC programs available at the mpC homepage. Not all of these programs are good mpC programs (from the point of view of their efficiency/portability/modularity), but all of them are correct.

2. Basic concepts

2.1 Computing space and network objects

When programming in C, the user may imagine that there is the storage accessible to him, and he can manage this storage allocating data objects in there. When programming in mpC, the 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 in there.

In mpC, the notion of *computing space* is defined as a set of typed virtual processors connected with links of different bandwidth accessible to the 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 has an additional attribute characterizing its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from the source processor to the 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 bandwidth. 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 and performances 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, the creation of new network is initiated by a processor of an existing network. This processor is called a *parent* of the created network. The parent belongs to the created network.

The only processor defined from the beginning of program execution till program termination is the pre-defined *host-processor* of the `scalar` type and ordinary performance.

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

```
/* Line 1 */          nettype Rectangle {
/* Line 2 */              coord I=4;
/* Line 3 */              node {
/* Line 4 */                 I<2:  fast scalar;
/* Line 5 */                 I>=2: slow scalar;
/* Line 6 */              };
/* Line 7 */              link {
/* Line 8 */                 I>0:  [I]<->[I-1];
/* Line 9 */                 I==0: [I]<->[3];
/* Line 10 */             };
/* Line 11 */             parent [0];
/* Line 12 */         };
```

introduces the network type named `Rectangle` that corresponds to networks consisting of 4 processors of the `scalar` type and different performances 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 3-6 are a node declaration. It relates processors to the coordinate system and declares their types and performances. Line 4 stands for the predicate ***for all*** `I<4` ***if*** `I<2` ***then*** *fast processor of the* `scalar` *type is related to the point with the coordinate* `[I]`. Line 5 stands for the predicate ***for all*** `I<4` ***if*** `I>=2` ***then*** *slow processor of the* `scalar` *type is related to the point with the coordinate* `[I]`. The performance specifiers `fast` and `slow` specify relative performances of processor nodes of the same type. For any network of this type, this information allows the compiler to associate the weight with each processor of the network, normalizing it in relation to the weight of the parent processor. Note, that the host-processor is always of the `scalar` type and ordinary performance.

Lines 7-10 are a link declaration. It specifies links between processors. 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* `[I-1]`, and line 9 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]`. Note, that if a link between two processors is not specified explicitly, it is meant that there is a link whose length is longest for this network.

Line 11 is a parent declaration. It specifies that the 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;
```

introduces the identifier r1 of network object of the type `Rectangle`.

By definition, data object *distributed* over a region of the computing space comprises a set of components of any one type so that every processor of the region holds one component. For example, the declarations

```
net Rectangle r2;
int [*]de, [r2]da[10];
repl [*]di;
```

declare the integer variable `de` distributed over the entire computing space, the array `da` of 10 `int`s distributed over the network `r2`, and the integer variable `di` *replicated* over the entire computing space. By definition, a distributed object is *replicated* if all its components is equal to each other (see sections 2.3 and 4).

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 */          nettype Ring(n, p[n]) {
/* Line 2 */              coord I=n;
/* Line 3 */              node {
/* Line 4 */                 I>=0: fast*p[I] 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 the 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 parameters of the topology `Ring`, namely, the integer parameter `n` and the vector parameter `p` consisting of `n` integers. 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*** *fast processor of the* *`scalar`* *type, whose relative performance is specified by the value of* *`p[I]`*, *is related to the point with the coordinate* *`[I]`*, and so on. Here, the performance specifier `fast*p[I]` includes the so-called power specifier `*p[I]`. In general, the value of the expression in a power specifier shall be positive integer. Any operand in the expression shall consist only of coordinate variables, constants and generic parameters (if any). If the value of the expression is equal to 1, the power specifier may be omitted. It is meant that in the framework of the same network-type declaration any performance specifier with the `fast` keyword specifies more powerful processor than a performance specifier with the `slow` keyword. It is meant also that the greater value of the expression in a power specifier the more performance is specified. Note, that in this case the following simplified form of line 4

```
I>=0: p[I];
```

may be used (see 3.1.2 for details).

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

```
repl [*]a[4]={10,20,30,40};
net Ring(4,a) r;
```

introduces the integer array `a` replicated over the entire computing space, the network type `Ring(4,a)` as an instance of the topology `Ring` as well as the identifier `r` of the network object of this type.

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

```
repl [*]m, [*]n[100];
/* Computation of m,n[0],...,n[m-1]*/
{
  net Ring(m,n) rr;
  ...
}
```

introduces the identifier `rr` of the network object, the type of which is defined completely only in 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 declared with *static* computing space duration is created only once, conceptually, either on the first entry into the block in which it is defined (for local static networks), or on the first entry into any of basic functions (see sections 2.4, 2.6) being in scope of its identifier (for global static networks). Once created the static network exists till termination of the entire program.

A new instance of a network declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network 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(3,p1) r3;
static net Ring(4,p2) r4;
extern net Ring(5,p3) r5;
int [*]f(repl int k)
{
   net Ring(k,pk) rk;
   static net Ring(k+1,pk1) rk1;
   ...
}
```

specifies that the identifiers `r3` and `r5` of static network objects has file scope and external linkage, the identifier `r4` of the static network object has file scope and internal linkage, the identifier `rk` of the automatic network object and the identifier `rk1` of the static network object have block scope. (Note, that the header of the definition of the function `f` includes the construct `[*]` which specifies the kind of the function (see sections 2.4, 2.6)).

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 `r5`, all the rest declarations are network object definitions. The parent of all these network object is the host-processor. The following example shows how one can specify another parent:

```
net Ring(6,p6) r6;
net Ring (7,p7) [r6:I==3] r7;
```

Here, the network `r6` has the host-processor as its parent, meantime the network `r7` has the processor of the network `r10` with the coordinate `[3]` as its parent. Note, that a processor node of an automatic network cannot be a parent of a static network. For example, if `r6` is an automatic network, then the declaration

```
static net Ring (8,p8) [r6:I==0] r8;
```

is not correct, meantime the equivalent declaration

```
static net Ring (8,p8) r8;
```

is correct.

2.2 Subnetworks

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

Unlike an implicitly defined subnetwork, an explicitly defined subnetwork has the 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. Note, that a static subnetwork cannot be allocated in an automatic region of the computing space The lifetime of an implicitly-defined subnetwork is defined by compiler.

For example, the mpC file fragment

```
/*Line 1 */    nettype Web(m,n) {
/*Line 2 */        coord R=m, Phi=n;
/*Line 3 */        node {
/*Line 4 */            R==0&&Phi>0:  void;
/*Line 5 */            R==0&&Phi==0: fast scalar;
/*Line 6 */            default:      scalar;
/*Line 7 */        };
/*Line 8 */        link {
/*Line 9 */            R==0:       [0,0]<->[1,Phi];
/*Line 10*/            R>0:        [R,Phi]<->[R-1,Phi];
/*Line 11*/            Phi>0&&R>0:  length*1 [R,Phi]<->[R,Phi-1];
/*Line 12*/            Phi==0&&R>0: length*1 [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: Phi%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*/       subnet [seastar10x10: R<5] seastar5x5;
/*Line 23*/       ...
/*Line 24*/    }
```

introduces the topology `Web` and defines the network object `web10x20` of the `Web(10,20)` type as well as its subnetworks `seastar10x10`, `subweb5x20`, `grid5x20` and `seastar5x5`.

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 computation load of this processor will be more intensive than those of the rest of the processors. Radial links between processors are of normal length (the attribute *length* is equal to 0), meantime circular links are longer (their attribute *length* is equal to 1). It means that data exchange through radial links will be more intensive than through the circular ones. In general, the attribute *length* for normal links is equal to 0 and may not be specified explicitly, meantime this attribute for long links is greater than 0 and for short links is less than 0 and must be specified explicitly.

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: Phi%2==0]` specifies the processors of `web10x20` that constitute the subnetwork. Namely, a proces-

sor of `web10x20` with coordinates `[R,Phi]` belongs to the subnetwork `seastar10x10` if and only if `Phi%2==0`.

Similarly, lines 20 and 21 are explicit definitions of the automatic subnetwork `subweb5x20` and the static subnetwork `grid5x20` of the network `web10x20` both named by identifiers with block scope.

Line 22 is an explicit definition of the automatic subnetwork `seastar5x5` whose identifier has block scope. The subnetwork `seastar5x5` is a subnetwork of the subnetwork `seastar10x10` and, hence, of the network `web10x20`.

A subnetwork always inherits the coordinate system of its supernetwork. 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 the correspondence between processors of different subnetworks in distributed operations.

The partial order "to be a subnetwork of" is defined on a set of subnetworks of the same network. A compiler needs the relation for correct translation of many expressions. The partial order should be explicitly specified in mpC program. There are 2 ways to do it. First, the relation can be specified with a subnetwork declaration, similar to the declaration in line 22 which specifies that `seastar5x5` is a subnetwork of `seastar10x10`. Second, one can specify the relation by the special *relation declaration*. For example, although, in fact, `seastar5x5` is a subnetwork of `subweb5x20`, the compiler will treat them as incomparable ones, if there is not the declaration

```
relation seastar5x5<subweb5x20;
```

in the corresponding block.

Finally, there are *hard* and *flexible* subnetworks. Hard subnetworks can be used everywhere, where networks can be used. On the other hand, there are some restrictions in the use of flexible subnetworks. In particular, flexible subnetworks cannot be used to evaluate postfix reduction operators, network functions cannot be called on such subnetworks. Any implicitly-defined subnetwork is flexible. An explicitly-defined subnetwork is flexible only if its declaration includes the keyword `flex`. The only advantage of flexible subnetworks is their less cost of creation than the creation of hard networks.

2.3 Distribution of data

The notion of *distributed data object* is introduced. Namely, data object distributed over a region of the computing space comprises a set of components of any one type so that every processor of the holds one component. So, a distributed data object is characterized by the type and attributes of the region over which it is distributed as well as the type and attributes of components.

In particular, data object can be distributed over the entire computing space. It means that creation of any network includes the creation of the corresponding component of the data object on every processor of the network.

Except implicit specification, 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,p11) Net1;
int [*]Derror, [Net1]Da[10], *[Net1:I<7]Dpi[5];
```

declare `Derror` as an integer data object distributed over the entire computing space, declare `Da` as an array of 10 `int`s distributed over 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 over this subnetwork.

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

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

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

A distributed object all the components of which are equal to each other during all the time of program execution is a *replicated* data object. To specify replicated data objects, the qualifier `repl` is used in the manner similar to the use of the type qualifiers `const` and `volatile`. For example, the declaration

```
int repl [*]n=10;
```

defines the variable `n` replicated over the entire computing space.

The notion of *distributed value* is introduced similarly. A value distributed over a region of the computing space comprises a set of components of any one type so that every processor of the region holds one component. the notions of value distributed over the entire computing space and replicated value are introduced similarly.

2.4 Distribution of computations

An expression can be evaluated by the host-processor, by a single processor of a network, by a network or a subnetwork, by a set of networks/subnetworks, or by the entire computing space. In the latter three 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 over 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, an asynchronous expression doesn't need communications between processors of the evaluating region of the computing space 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 over the same region of the computing space, and the distributed operator is divided into a set of independent undistributed operators each of which operates on corresponding components of the operands. If an expression is built only from such operators, and all they are distributed over the same region of the computing space, then the entire expression will be asynchronous.

A statement of the mpC language can be executed on the host-processor, on a single processor of a network, on a network or subnetwork, on a set of networks/subnetworks, or on 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`. If a statement is executed on 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. An asynchronous statement does not need communications between processors of the executing region of the computing space during its execution.

In particular, if all expressions and substatements of a sequential statement are asynchronous and distributed over the same region of the computing space, then the entire statement is asynchronous. In this case, the distributed statement is divided into a set of independent undistributed statements each of which is executed on the corresponding processor using the corresponding data components.

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

The following simple mpC program computes the sum of two vectors.

```
/*Line 1 */    nettype 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 M 4 /*The number of virtual processors*/
/*Line 12*/    #define N 300
/*Line 13*/    #define NM N*M
/*Line 14*/    void [*]main()
/*Line 15*/    {
/*Line 16*/       double [host]x[NM], [host]y[NM], [host]z[NM];
/*Line 17*/       int [host]i;
/*Line 18*/       void [*]parsum();
/*Line 19*/       int printf();
/*Line 20*/       .../* Input of the arrays x and y */
/*Line 21*/       parsum((void*)x, (void*)y, (void*)z);
/*Line 22*/       for(i=0; i<NM; i++)
/*Line 23*/         ([host]printf)("%f ", z[i]);
/*Line 24*/    }
/*Line 25*/    void [*]parsum(double *[host]x[N],
/*Line 26*/                   double *[host]y[N],
/*Line 27*/                   double *[host]z[N])
/*Line 28*/    {
/*Line 29*/       net Star(M) Sn;
/*Line 30*/       double [Sn]dx[N], [Sn]dy[N], [Sn]dz[N];
/*Line 31*/
/*Line 32*/       dx[]=x[];
/*Line 33*/       dy[]=y[];
/*Line 34*/       dz[]=dx[]+dy[];
/*Line 35*/       z[]=dz[];
/*Line 36*/    }
```

The program includes 3 functions - main and `parsum` defined here and the library function `printf`. Lines 14-21 contain the `main` definition. Line 16 contains the definition of the arrays x, y and z all belonging to the host-processor. Line 17 contains the definition of integer variable `i` belonging to the host-processor. Lines 18-19 contain the declaration of function identifiers `parsum` and

`printf`. In general, mpC allows 3 kinds of functions. Here, functions of two kinds are used: main and `parsum` are *basic* functions, and `printf` is a *nodal* function.

A call to *basic function* is an overall expression. Its arguments (if any) shall either belong to the host-processor or be distributed over the entire computing space, and the returning value (if any) shall be distributed over the entire computing space. In contrast to another kind of functions, it can define network objects. In lines 14, 18 and 25, the construct `[*]`, placed just before the function identifier, specifies that an identifier of basic function is declared.

Nodal function can be executed completely by any one processor. Only local data objects of the executing processor can be created in such a function. In addition, the corresponding component of an externally-defined distributed data object can be used in the function. A declaration of nodal function (e.g., in line 19) does not need any additional specifiers. All pure C functions are nodal from the point of view of mpC.

Line 23 contains an undistributed statement executed on the host-processor. It includes a call to the nodal function `printf` on the host-processor. Line 21 contains a call to the basic function `main` and is executed on the entire computing space.

Lines 25-36 contain the definition of the function `parsum`. Line 29 contains the definition of the automatic network `Sn`. Line 30 contains the definition of automatic arrays `dx`, `dy` and `dz` all distributed over `Sn`.

Line 32 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, 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 `dx[]` designates the distributed array `dx` as a whole. In addition, mpC allows to apply the operator `[]` not only to expressions having the type "array of *type*", but also to expressions having the type "pointer to *type*". The result is treated as an array of *type*s of undefined size. So, the expression `x[]` designates the array of undefined size whose members have the type `double[300]`. The expression `dx[]=x[]` scatters the elements of the array `x[]` to components of `dx`. The number of scattered elements is equal to the number `M` (=4) of components of `dx`.

Similarly, the statement in line 33 scatters the elements of the array `y[]` to components of the distributed `dy`.

The statement in line 34 performs asynchronously the sum of the vector values of distributed arrays `dx` and `dy` and assigns the result to the distributed array `dz`.

Finally, the statement in line 35 gathers components of the distributed vector value of the distributed array `dz` to the host-processor putting then in sequential members of the array `z[]` of undefined size.

2.5 Distributed networks and nested parallelism

To support nested parallelism, mpC allows to define not only a single network but also a set of single networks by means of defining so-called *distributed network object*. A definition of a distributed network object specifies the type of the network object and its parent network. Such a definition may be considered as a distributed over the parent network definition of a single network of the specified type. The parent network of a distributed network can also be distributed. But in any case, a distributed network is a set of single networks of the same type. The number of single networks in this set is equal to the number of processors in the parent network each of processors of the parent network being a parent of a single network of the set.

There are not facilities to specify a single network belonging to a distributed network in mpC. Therefore, whenever one specifies a subnetwork of a distributed network, he means a set of subnetworks of the single networks constituting the distributed network. Similarly, if one specifies a single processor of a distributed network, he means a set of single processors of the single networks constituting the distributed network. Any computation on a distributed network is divided into independent computations on single networks constituting the distributed network.

Therefore, the notion of distributed network implies the notions of *partially asynchronous expression and partially asynchronous statement.* So, if an expression is evaluated by a distributed network, there are no communications between parent processors of the networks constituting the set specified by the distributed network, but such communications are possible inside each of the single networks, then the expression is called a partially asynchronous expression in relation to the parent network of the distributed network. The notion of partially asynchronous statement is defined similarly.

For example, in the fragment

```
/* Line 1  */     nettype 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 */     };
/* Line 12 */     net Ring(5) r5;
/* Line 13 */     void [*]main()
/* Line 14 */     {
/* Line 15 */        net Ring(3) [r5] r3;
/* Line 16 */        int [host]x[5], [r5]dx, [r3]ddx, [r3]ddy;
/* Line 17 */        ([host]Input)(x);
/* Line 18 */        dx=x[];
/* Line 19 */        ddx=dx;
/* Line 20 */        ddy=ddx[+];
/* Line 21 */        dx=[r3:parent]ddy;
/* Line 22 */        x[]=dx;
/* Line 23 */        ...
/* Line 24 */     }
```

line 15 defines the network `r3` distributed over 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 16 defines the array `x` which belongs to the host-processor, the variable `dx` distributed over `r5`, and the variables `ddx` and `ddy` both distributed over `r3`.

Line 17 contains a call to the nodal function `Input` on the host-processor. Since the function `Input` is not declared explicitly, it is considered to be declared implicitly in the least enclosing block as a nodal function returning `int`.

The statement in line 18 scatters the elements of `x` to the corresponding components of `dx`.

The statements in lines 19-20 are partially asynchronous in relation to the network `r5`. The statement in line 19 for each of `5` single networks constituting `r3` broadcasts the relevant component of `dx` to the corresponding components of `ddx` in parallel. The statement in line 20 assigns the sum of the relevant components of `ddx` to the corresponding components of `ddy` in parallel for each single network from r3.

The statement in line 21 is asynchronous in relation to `r5` and assigns the specified components of `ddy` to the components of `dx`. It contains the special subnetwork specifier `[r3:parent]` which specifies the parent of the network `r3`.

Finally, the statement in line 22 gathers the components of the distributed variable `dx` to the array `x`. So, if the input value of the array `x` is equal to {0, 1, 2, 3, 4} then just after line 22 its value will be equal to {0, 3, 6, 9, 12}.

In general, not only a network but a subnetwork may be the parent of a distributed network.

Every time when speaking of an entity (data object, value, or function) distributed over a distributed network, we mean a set of entities each of which is distributed over a single network belonging to the distributed network. Similarly, when speaking about an entity distributed over a subnetwork of the distributed network, we mean a set of entities each of which is distributed over a subnetwork of a single network belonging to the distributed network. Finally, when speaking about an undistributed entity belonging to a single processor of the distributed network, we mean a set of undistributed entities each of which belongs to a processor of a single network belonging to the distributed network.

2.6 Network functions

Like ANSI C, in mpC the minimum translation unit is a source file. A file consists of network type declarations, external network/subnetwork declarations, external relation declaration, external data object declarations, function definitions, and possibly some other external declarations (here, an external declaration is a declaration appearing out of functions).

To support modular parallel programming as well as the writing of libraries of parallel programs, in addition to basic and nodal functions so-called network functions are introduced in mpC.

In general, a *network function* is called and executed on some network or hard subnetwork, and its arguments and value (if any) is also distributed over this region of the computing space. The header of a network function definition either specifies an identifier of static network or subnetwork having file scope, 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 an appropriate type. In any case, no network other than the network specified in the function definition header can be created or used in the function definition body. But it is allowed to create and use its subnetworks. Only data objects belonging to the region of the computing space specified in the header can be defined in the body. In addition, the corresponding components of an externally-defined distributed data object can be used. For example, in the fragment

```
/* Line 1 */   net Ring(5) r5;
/* Line 2 */   int [r5]da, [*]db, [host]a[5], [host]b[5], [host]x[5];
/* Line 3 */   void [*]main()
/* Line 4 */   {
/* Line 5 */     int [r5]dx;
/* Line 6 */     int [r5]f();
/* Line 7 */     ([host]Input)(a,x);
/* Line 8 */     [r5]db=da=a[];
/* Line 9 */     dx=x[];
/* Line 10 */    dx=f(dx);
/* Line 11 */    x[]=dx;
/* Line 12 */    ([host]Output)(x);
/* Line 13 */  }
/* Line 14 */  int [r5]f(int dx)
/* Line 15 */  {
/* Line 16 */    int result;
/* Line 17 */    result=da+db*dx;
/* Line 18 */    return result;
/* Line 19 */  }
```

line 10 contains the call to the network function `f`, and line 6 contains the function declaration being in scope for `f`. The definition of this function is contained in lines 14-19. The function `f` is related with the network `r5`. This is specified by means of the construct `[r5]` both in the declaration of its identifier (line 6) and in its definition (line 14). Note, that it is meant that the formal parameter `dx` declared in line 14 is distributed over the network `r5`. In addition, in line 17, expression `db*dx` is equivalent to expression `[r5]db*dx`, where operator `[r5]` cuts from `db` the components belonging to `r5`.

If a function has the *network formal parameter*, the declaration of this parameter in the function definition header specifies its network type. This network type may be either completely defined or parametrized. For example, in the fragment

```
/* Line 1  */    nettype Grid(n) {
/* Line 2  */       coord I=n, J=n;
/* Line 3  */       node {
/* Line 4  */          default: scalar;
/* Line 5  */       }
/* Line 6  */       link {
/* Line 7  */          default: [I,J]<->[I+1,J], [I,J]<->[I,J+1];
/* Line 8  */       }
/* Line 9  */       parent [0,0];
/* Line 10 */    }
/* Line 11 */    #define N 100
/* Line 12 */    void [*]main()
/* Line 13 */    {
/* Line 14 */       net Grid(N) gN;
/* Line 15 */       int [net Grid(2)] sum(), [host]x[N][N], [gN]dx;
/* Line 16 */       int repl [gN]i;
/* Line 17 */       ([host]Input)(x);
/* Line 18 */       dx=((int*)x)[];
/* Line 19 */       for(i=N-2; i>=0; i--) {
/* Line 20 */          subnet [gN:I>=i&&J>=i&&I<i+2&&J<i+2] g;
/* Line 21 */          [g]dx=[()g]sum([g]dx);
/* Line 22 */       }
/* Line 23 */       ([host]Output)([host]dx);
/* Line 24 */    }
/* Line 25 */    int [net Grid(2) g2] sum(int dx)
/* Line 26 */    {
/* Line 27 */       int [g2:I==0]d0, [g2:I==0&&J==O]d00;
/* Line 28 */       d0=[g2:I==1]dx;
/* Line 29 */       dx+=d0;
/* Line 30 */       d00=[g2:I==0&&J==1]dx;
/* Line 31 */       dx+=d00;
/* Line 32 */       return dx;
/* Line 33 */    }
```

line 21 contains the call to the network function `sum`, and line 15 contains the function declaration being in scope for `sum`. The definition of this function is contained in lines 25-33. The header of this definition (line 25) contains the declaration of the special formal parameter `g2` corresponding to the network on which this function is called.

In general, if a network formal parameter has a completely defined type, the corresponding argument should be either a network or a hard subnetwork conforming to the formal parameter. By definition, the network (or subnetwork) A *conforms* to the network (or subnetwork) B if and only if they have the same number of (non-void) processors.

Line 14 defines the automatic network `gN` representing `NxN` grid of scalar processors. In line 16 the distributed variable `i` is declared with the specifier `repl` meaning that if the value of this variable is defined then all its components are equal to each other. The statement in line 18 sends the value of `x[i][j]` to the component `[gN:I==i&&J==j]dx` for all `i`, `j` from `0` to `N-1`.

The iteration statement in lines 19-22 is performed on the network `gN`. Line 20 contains the definition of the automatic subnetwork `g` of `gN` representing a rectangle on the main diagonal of the grid.

Line 21 contains the call to `sum` on `g`. The value of the function call is distributed over `g`. The component of the value with the coordinates `I==i` and `J==i` is equal to the sum of the components of the argument `[g]dx`. The assignment in line 21 modifies the corresponding components of `dx`. So, the execution of the iteration statement produces the value of component `[gN:I==0&&J==0]dx` (or equally `[host]dx`) equal to the sum of the values of the `dx` components disposed on the three diagonals of the grid `gN`.

Note, that mpC allows one of operands of an asynchronous operator to be distributed over a subregion of the computing space region through which the other operand is distributed. In this case, the operator is performed on this subregion. So, the expression `dx+=d0` in line 29 is equivalent to the expression `[g2:I==0]dx+=d0`, and the expression `dx+=d00` in line 31 is equivalent to the expression `[g2:I==0&&I==0]dx+=d00`.

If a network formal parameter has a parametrized type, the corresponding topological parameters are also declared in the header of the function definition being also special formal parameters. In the function body, each scalar topological parameter is treated as an unmodifiable variable of the type `int` replicated over the network formal parameter, and the vector topological parameter - as an unmodifiable indexed set of integer variables replicated over the network formal parameter. (The only operation is applicable to an indexed set of integer variables, namely, an access to an element via its indices). The number of indices of the latter and their ranges (may be defined dynamically) are detected by the compiler from the declaration of the corresponding topology.

When calling to the function, the corresponding topological arguments specify a network type as an instance of the corresponding topology, and the network argument specifies a region of the computing space treated by the function as a network of this type. An argument corresponding to the scalar topological parameter should be of the type `int` and replicated over the network argument. An argument corresponding to the vector topological parameter should be a distributed pointer (of any type) to the initial member of an integer array replicated over the network argument. For example, in the fragment

```
/* Line 1  */     void [*]main()
/* Line 2  */     {
/* Line 3  */        net Ring(5) r5;
/* Line 4  */        net Rectangle r;
/* Line 5  */        int [r5]dx, [r]dy;
/* Line 6  */        void [net Ring(n)] shift();
/* Line 7  */        ...
/* Line 8  */        [(5)r5]shift(&dx);
/* Line 9  */        [(4)r]shift(&dy);
/* Line 10 */        ...
/* Line 11 */     }
/* Line 12 */     void [net Ring(n) rn] shift(int *da)
/* Line 13 */     {
/* Line 14 */        int [rn]me, [rn]he;
/* Line 15 */        me = I coordof da;
/* Line 16 */        he = (me==n-1)?0:(me+1);
/* Line 17 */        [rn:I==me](*da) = [rn:I==he](*da);
/* Line 18 */     }
```

lines 8-9 contains the calls to the network function `shift`, and line 6 contains the function declaration being in scope for `shift`. The definition of this function is contained in lines 12-18. The header of this definition (line 12) contains the declaration of the network formal parameter `rn,` corresponding to

the network on which this function is called, as well as the topological formal parameter `n` treated in the function body as if it was declared with the declaration

```
int const repl [rn]n;
```

As a result of a call to the function, all the components of `n` should have the same value specifying the type of `rn` as an instance of the topology `Ring`.

So, in line 8 the function `shift` is called on the network `r5` that is just a network argument. This network is of the type `Ring(5)`, therefore the constant `5` is used as a topological argument.

In line 9, the function is called on the network `r` that is a network argument in this case. This network has the type `Rectangle` not being an instance of the topology `Ring`. The topological argument (the constant `4`) specifies that in this case the function called shall treat its network argument (that is, the network `r`) as having the type `Ring(4)`. The call is correct, because a network of the type `Rectangle` conforms to a network of the type `Ring(4)`.

The result of the binary operator `coordof` in line 15 is an integer value distributed over `rn` each component of which is equal to the value of the coordinate `I` of the processor to which the component belongs. The right operand of the operator `coordof` is not evaluated and used only for specification of the region of the computing space. The statements in lines 15-16 are asynchronous. The statement in line 17 shifts clockwise the distributed data object `*da`. Note, that the coordinate variable `I` is treated as an integer variable distributed over `rn`.

2.7 Pointer to function

In C, a function call includes the pointer to the function called. In mpC, a function call on a region of the computing space is treated as a set of undistributed function calls each of which is performed on its single processor of the region. In other words, the distributed function call may be treated as a distributed call to undistributed functions called *functional components* of the distributed call. Therefore, the distributed function call shall include the distributed pointer to the corresponding functional components.

So, the C language notion of function as an entity that may be pointed to is transformed to the mpC language notion of undistributed function. A nodal function as well as a functional component of basic or network function represent undistributed functions.

When declaring an identifier of the pointer to undistributed function, one can describe the function pointed to detailed enough. For example, whether it is a nodal function, or whether it is a functional component of basic function, or whether it is a functional component of network function with special formal parameters (in the latter case, the number of topological parameters as well as the type of the network parameter should be specified). If such a declaration is in scope for the identifier used in a function call, compiler shall check the correctness of the function call. Otherwise, the correctness of the function call is the responsibility of the user.

For example, the declaration

```
int [*](*[net1]pf)();
```

declares the identifier pf as a distributed over the network net1 pointer to functional component of basic function. The declaration

```
int (*[net2]pf)();
```

declares the identifier `pf` as a distributed over the network `net2` pointer to nodal function. The declaration

```
int [net Ring(3)](*[net3]pf)();
```

declares the identifier `pf` as a distributed over the network `net3` pointer to functional component of network function whose network formal parameter has the type `Ring(3)`. The declaration

```
int [net Web(4,n)](*[net4]pf)();
```

declares the identifier `pf` as a distributed over the network `net3` pointer to functional component of network function having two special formal parameters the network formal parameter belonging to the type family `Web(4,n)`.

Except when used as an operand where a function designator is permitted, a basic function identifier is converted to a distributed over the entire computing space pointer to undistributed function; a nodal function identifier is converted to pointer to nodal function distributed over the region of the computing space on which the calling function is called; an identifier of network function without special formal parameters is converted to a distributed over the corresponding region pointer to functional component of network function without special formal parameters; an identifier of network function with special formal parameters is converted to pointer to functional component of network function having the specified special formal parameters distributed over the region of the computing space on which the calling function is called.

3. Managing the computing space

3.1 Network type declaration

Syntax.

```
<network_type_declaration>:
   <network_type_class_specifier>(opt)
      nettype <identifier>
      <generic_parameter_declaration>(opt)
      { <network_declaration_list> } ;

<generic_parameter_declaration>:
   (<generic_parameter_list>)

<generic_parameter_list>:
   <generic_parameter_declarator>
   <generic_parameter_list> , <generic_parameter_declarator>

<generic_parameter_declarator>:
   <identifier>
   <generic_parameter_declarator> [ <expression> ]

<network_declaration_list>:
   <coordinate_declaration>
      <node_declaration>(opt)
      <link_declaration>(opt)
      <parent_node_declaration>(opt)

<network_type_class_specifier>:
   static
   extern
```

Constraints.

A network type declaration shall not appear in a function.

Semantics.

A network-type declaration introduces a network-type identifier and specifies attributes of the network type, such as the number, types and relative performances of processors, links and their lengths, as well as the parent processor. A network type can be either simple or parametrized (generic). A declaration of generic network type called also a topology shall contain a generic parameter declaration. There are scalar and vector generic (or topological) parameters. A scalar generic parameter is treated as an integer. A vector generic parameter is treated as an indexed set of integers. The number of indices and their ranges are specified by the generic parameter declarator. The expression in the generic

parameter declarator can be built only from integer constants and scalar generic parameters. The scope of generic parameters is the corresponding network-type declaration.

A network-type declaration may include the specifier `extern` or `static`.

A network-type declaration that also causes a compiler to generate target program components providing the access to topological information about networks of a relevant type is a network-type definition.

A network-type declaration without the specifier `extern` is a network-type definition. The specifier `static` specifies internal linkage for the network-type identifier declared. The network-type declaration without any specifier specifies external linkage.

A network-type declaration with the specifier `extern` is not a network-type definition and is used by a compiler to access correctly to the corresponding topological information. In this case, somewhere in the set of source files that constitutes the entire program there exists a definition for the given identifier.

3.1.1 Coordinate declaration

Syntax.

```
<coordinate_declaration>:  coord <coordinate_list> ;

<coordinate_list>:
   <coordinate_declarator>
   <coordinate_list> , <coordinate_declarator>

<coordinate_declarator>: <identifier> = <expression>
```

Constraints.

The expression in the coordinate declarator shall be integer. The operands in the expression shall consist only of constants and generic parameters of the generic network type (if any).

Semantics.

A coordinate declaration declares a coordinate system which processor nodes of the network declared are related to. A coordinate declarator introduces an identifier of a coordinate variable and specifies its attributes. Coordinate names belong to the same name space as ordinary identifiers. The scope of an identifier of a coordinate variable extends from the completion of its declarator but is not continuous; it includes the network declaration list that contains the corresponding coordinate declaration, all relevant subnetwork specifiers as well as left operands of relevant `coordof` operators. If a declaration of a lexically identical identifier exists in this scope, it is hidden.

A coordinate variable has the type `int` and is characterized by the number in the list of coordinates and the range of values. Correspondingly, if the coordinate variable occurs in an expression in a link descriptor or in the parent node description, the number of expressions in an expression list shall agree with the number of coordinate variables in the coordinate list. The range of values of the coordinate variable is specified by the expression in the coordinate declarator and includes integers from 0 to N-1, where N is the value of the expression.

Example. The coordinate declaration

```
coord x=100, y=10, z=N;
```

declares the 3-D coordinate system which a network containing up to 100*10*N nodes may be related to.

3.1.2 Node declaration

Syntax.

```
<node_declaration>: node {<node_declarator_list>};

<node_declarator_list>:
   <node_declarator>
   <node_declarator_list> <node_declarator>

<node_declarator>:
   <expression> ':' <performance_specifier>(opt) <node_type>(opt) ;
   default ':' <performance_specifier>(opt) <node_type>(opt) ;

<node_type>:
   void
   memory
   scalar
   vector

<performance_specifier>:
   <expression>
   fast <power_specifier>(opt)
   slow <power_specifier>(opt)

<power_specifier>:
   * <expression>
```

Constraints.

The expression in the node declarator shall be integer. The operands in the expression shall consist only of coordinate variables, constants and generic parameters (if any).

Either the performance specifier or the node type shall appear in the node declarator.

There may exist at most one default node declarator in a node declarator list.

The expression in the performance specifier shall be integer. The operands in the expression shall consist only of coordinate variables, constants and generic parameters (if any).

The expression in the power specifier shall be integer. The operands in the expression shall consist only of coordinate variables, constants and generic parameters (if any).

Semantics.

A node declaration associates processor nodes to the given coordinate system and declares their types and performances.

A processor node of the type `void` has no data and does not take part in computations. The equivalent interpretation is that the type `void` indicates that no processor is related to the positions with the corresponding coordinates. A processor of the type `memory` can rather store data than operate on it. A

processor of the type `vector` can perform vector operations efficiently. Finally, most common processors are of the type `scalar`. If the node type does not appear in the node declarator, it specifies a processor of the `scalar` type.

Performance specifiers specify relative performances of processor nodes of the same type. The value of the expression in the power specifier shall be positive. If it is equal to 1, the power specifier may be omitted. It is meant that any performance specifier with the `fast` keyword specifies more powerful processor than a performance specifier with the `slow` keyword. It is meant also that the greater value of the expression in a power specifier the more performance is specified. For every network of relevant type, this information allows the compiler to associate a weight with each processor of the network normalizing it in relation to the weight of the parent processor. Note, that the host-processor is always of the `scalar` type and the regular performance.

It is meant that a simplified performance specifier having the form of expression is `fast` and of the `scalar` type.

When processing a node declarator, the compiler evaluates the (logical) expression for every permissible set of values of the coordinate variables. If the value is non-zero (that corresponds to the logical value `true`), a processor of the specified type and performance is related to the coordinates. If the same coordinates satisfy more than one logical expressions, it depends in implementation processor of which type and performance will be associated with the coordinates.

The default node declarator declares the type and performance of all the processor nodes whose coordinates don't satisfy any (logical) expression in the rest of the node declarators of the node declaration. If there does not exist a default node declarator, these processor nodes shall have the type `void`.

If a network declaration list does not contain a node declaration, all the processor nodes of the network shall have the type `scalar` and the regular performance.

Example. The declaration

```
net Star(N) {
 coord i=N;
 node {
   default: scalar;
 }
 ...
};
```

declares all the processor nodes to be of the type `scalar`. The declaration

```
net Star2(M,N) {
 coord i=M;
 node {
   !i   : memory;
   i % N  : scalar;
   default : vector;
 }
 ...
};
```

declares a generic network type with different types of nodes whose relation to coordinates depend on the generic parameters.

3.1.3 Link declaration

Syntax.

```
<link_declaration>:
   link { <link_declarator_list> } ;
   link <free_coordinate_list> { <link_declarator_list> } ;

<link_declarator_list>:
   <link_declarator>
   <link_declarator_list> <link_declarator>

<link_declarator>:
    <expression> ':' <single_link_declarator_list> ;
    default ':' <single_link_declarator_list> ;

<single_link_declarator_list>:
   <single_link_declarator>
   <link_length_specifier>(opt) <single_link_declarator>
   <single_link_declarator_list>,<single_link_declarator>

 <single_link_declarator>:
   [<expression_list>] <direction_specifier> [<expression_list>]

 <free_coordinate_list>:  ( <coordinate_list> )

 <link_length_specifier>:
    length * <expression>

 <direction_specifier>:
    ->
    <->
```

Constraints.

An expression in the link declarator shall be integer. The operands in the expression shall consist only of constants, generic parameters (if any), and coordinate variables including free coordinates variables (if any).

The expression in the link-length specifier shall be integer. The operands in the expression shall consist only of constants, generic parameters (if any), and coordinate variables including free coordinates variables (if any).

Semantics.

A link declaration declares links between processor nodes. A link is characterized by the length and the direction.

If a free coordinate list appears in the link declaration, it declares additional coordinate variables (named free coordinate variables) and specifies their ranges of values. The declaration of free coordinate variables does not change the coordinate system that has been declared. The scope of an identifier of a free coordinate includes the link declarator list that follows the corresponding free coordinate list.

Free coordinate variables are used if the network topology can not be specified with only regular coordinate variables.

Link declarators in the link declarator list are processed sequentially. When processing a link declarator, the compiler evaluates the (logical) expression for every permissible set of values of the coordinate variables (including free coordinate variables, if any). If a set of values satisfies the logical expression (makes it non-zero), for every single link declarator in the single link declarator list all the expressions in both expressions lists are evaluated, and the link between the processor node, whose coordinates are determined by the left part of the single link declarator, and the processor node, whose coordinates are determined by the right part of the single link declarator, is established. The direction of the link is specified by the direction specifier.

The length of a link is specified by a link-length specifier. The value of the expression in the link-length specifier characterizes the length of the link. If it is equal to 0, the link-length specifier may be omitted, and the link shall be of the regular length. It is meant that the greater value of the expression in the link-length specifier the longer length is specified. So, negative values correspond to short links, and positive value correspond to long links. For every network of relevant type, this information allows the compiler to associate a weight with each link of the network.

If there exists a default link declarator, it is processed as if it is the last link declarator in the link declarator list, whose logical expression is non-zero for all permissible sets of values of the coordinate variables.

If a network declaration list does not contain a link declaration, there exists a link of the regular length between any two processor nodes.

If the link declaration does not specify a link between some pair of processor nodes, it means existence very long link connecting them rather than absence of any link.

Example. The declaration

```
net Star(N) {
 coord i=N;
 node {
   default:scalar;
 }
 link {
   i>0: [0] -> [i] , [i] -> [0]; }
 ...
};
```

declares a generic type of networks of the star topology. The declaration

```
net Star2(N) {
 coord i=N;
 node {
   default:scalar;
 }
 link {
   i>0 && i%2    :    length*(-1) [0] -> [i+1], [i] -> [0] ;
   i>0 && !(i%2) :    length*1  [0] -> [i-1], [i] -> [0] ;
 }
 ...
};
```

declares a generic type of networks of the star topology with links of different length.

Example. The following declaration illustrates the usage of free coordinate variables:

```
net All_To_All(N) {
 coord i=N;
 node {
   default:scalar;
 }
 link (j=N) {
   i!=j && i%2 && j%2 : length*(-1) [i] -> [j];
   default            : [i] -> [j];
 }
 ...
};
```

3.1.4 Parent node declaration

Syntax.

```
<parent_node_declaration>: parent [<expression_list>];
```

Constraints.

An expression in the expression list shall be integer. The operands in the expression shall consist only of constants and generic parameters of the generic network type (if any). The number of expressions in the expression list shall agree with the dimension of the coordinate system that has been declared.

Semantics.

The parent node declaration specifies the coordinates of the parent processor node in the given coordinate system.

If a network declaration list does not contain a parent node declaration, the parent has zero number in the natural numeration of processor nodes (recall, that it is supposed that all non-void processor nodes are numerated in correspondence with the lexicographic order of their coordinates).

Example. The following complete generic network type declaration

```
net SeaStar(M,N) {
 coord r=M, fi=N;
 node {
   r==0 && fi>0 : void;
   default      : scalar;
 }
 link {
   r==0 : [0,0] -> [1,fi], [1,fi] -> [0,0];
   r>1  : [r-1,fi]->[r,fi], [r,fi]->[r-1,fi];
 }
 parent [0,0];
};
```

introduces the sea-star topology.

3.2 Network declarations.

Syntax.

```
<network_declaration>:
   <computing_space_class_specifier> (opt)
      <network_type_specifier> <network_list> ;

<network_list>:
   <network_declarator>
   <network_list> , <network_declarator>

<computing_space_class_specifier>:
   <storage_class_specifier>
```

Constraints.

Only `extern`, `static`, `auto` or `typedef` may be used as computing-space-class specifiers in a network declaration.

Semantics.

A network declaration introduces a set of identifiers, that are interpreted as names of networks, as well as specifies attributes of the identifiers (such as network type, parent, class of computing space duration). A network declaration that also causes computing space to be reserved for an network named by an identifier is a network definition.

The network declaration may contain specifiers `extern`, `static`, `auto`, or `typedef`.

Like ANSI C, the `typedef` specifier is called a "storage-class specifier" for syntactic convenience only. Within the scope of a declaration whose computing-space-class specifier is `typedef`, each identifier declared therein becomes a synonym for the network type specified by the network type specifier. Such a name shares the same name space as other identifiers declared in ordinary declarators.

A network declaration with the specifier `extern` indicates that somewhere in the set of source files that constitutes the entire program there exists an external definition for the given network identifier. Such a network declaration can not serve as a network definition.

If the network declaration without specifier `extern` occurs outside a function, the network identifier is declared with global static computing space duration, and serves as the definition. The specifier `static` specifies internal linkage for the network identifier declared. The network declaration without any storage-class specifier specifies external linkage.

Within a function, a declaration of a network with specifier `static`, `auto`, or without any computing-space-class specifier also serves as a network definition. The network declaration with specifier `static` declares the network identifier with local static computing space duration. The network declaration with specifier `auto` or without any computing-space-class specifier declares the network identifier with automatic computing space duration.

3.2.1 Network type specifier

Syntax.

```
<network_type_specifier>:
   net <identifier>
```

```
net <identifier> ( <argument_expression_list> )
```

Constraints.

An expression in the argument expression list corresponding to a scalar generic parameter shall be of the type `int`. If the network type specifier is a part of an external network declaration, the expression shall be constant. Otherwise, it shall be replicated over the entire computing space.

An expression in the argument expression list corresponding to a vector generic parameter shall be a distributed pointer (of any type) to the initial member of an integer array replicated over the network argument.

Semantics.

In mpC, one can declare a single network type as well as parametrized (generic) network type. Correspondingly, when declaring a network, one can specify its type either with the identifier of single network type, or by means of generic instantiation of a generic network type. The generic instantiation concludes in replacement generic parameters with values of generic arguments. The number of the generic arguments shall agree with the number of generic parameters. An array corresponding to a vector topological argument should be of the enough size. It is meant that it holds an indexed set of integers in such a way that the right index is faster then the left one.

3.2.2 Network declarator

Syntax.

```
<network_declarator> :
   <network_or_subnetwork_specifier>(opt) <identifier>

<network_or_subnetwork_specifier>:
   [ host ]
   [ <identifier> ]
   <subnetwork_specifier>

<subnetwork_specifier>:
   [ <identifier> ':' <expression> ]
```

Constraints.

A network declarator including a network-or-subnetwork specifier shall not appear in a declaration with the `typedef` specifier.

The identifier in the network-or-subnetwork specifier shall designate a network or an explicitly declared subnetwork.

The expression in the subnetwork specifier shall be asynchronous (in relation to the region R designated by the corresponding identifier) expression without side effects, each subexpression of which that does not include coordinate variables is replicated over the region R or its superregion. If the identifier in the subnetwork specifier designates a network, then the keyword `parent` can be used instead of the expression specifying the parent of the network.

Semantics.

Each network declarator declares one identifier of network object or network type.

If the network declarator appears in a network declaration without the `typedef` specifier and does not include a network-or-subnetwork specifier, a single network, whose parent is the host-processor, is declared. If there exists such a specifier, but it specifies a single processor node, then a single network, whose parent is the processor node specified, is declared. Otherwise, a distributed network, whose parent network is specified by the specifier, is declared.

Neither an automatic network nor its subnetwork (including a one-processor ones) can be a parent of a static network.

3.3 Declaration of subnetworks

Syntax.

```
<subnetwork_declaration>:
   <computing_space_class_specifier>(opt)
      subnet <subnetwork_declarator_list>;

<subnetwork_declarator_list>:
   <subnetwork_declarator_list> , <subnetwork_declarator>
   <subnetwork_declarator>

<subnetwork_declarator>:  <subnetwork_specifier> <identifier>

<relation_declaration>:
   relation <relation_declarator_list>;

<relation_declarator_list>:
   <relation_declarator_list> , <relation_declarator>

<relation_declarator>:
   <identifier> <relational_operator> <identifier>
```

Constraints.

Only `extern`, `static`, `auto` or `flex` may be used as the computing-space-class specifier in the subnetwork declaration.

Semantics.

A subnetwork declaration specifies attributes of a set of subnetwork identifiers. The subnetwork declarator consists of the subnetwork specifier and the subnetwork identifier being declared.

The subnetwork specifier includes an identifier of the region (network or subnetwork), whose subnetwork is specified, and a (logical) expression separating the processor nodes included in the specified subnetwork. The expression shall be asynchronous (in relation to the supernetwork R designated by the corresponding identifier) expression without side effects, each subexpression of which that does not include coordinate variables being replicated over the region R. Each processor of the supernetwork, whose component of the value of this expression is not equal 0, is included in the declared subnetwork.

A subnetwork inherits the coordinate system of its supernetwork. It means that any processor included in the subnetwork has there the same coordinates as in the corresponding supernetwork. At the same time, its natural number in the subnetwork may differ from its natural number in the supernetwork.

A subnetwork declaration that also causes computing space to be reserved for an subnetwork named by an identifier is an explicit subnetwork definition. In any case, the lifetime of a subnetwork does not continue over the lifetime of its supernetwork.

A subnetwork declaration with specifier `extern` indicates that somewhere in the set of source files that constitutes the entire program there exists an external definition for the given subnetwork identifier. Such declaration can not serve as a definition.

If a subnetwork identifier declaration without specifier `extern` occurs outside a function, then it serves as the definition. Conceptually, such a subnetwork is created once, when the program begins execution, but after creation of the corresponding supernetwork, and exists till the end of the execution of the entire program. The specifier `static` specifies internal linkage for the subnetwork identifier. The declaration without any computing-space-class specifier specifies external linkage.

Within a function, a subnetwork identifier declaration with specifier `static` serves as the definition. Conceptually, such subnetwork is created only on first entry into the block, in which it is declared, and exists till the end of its supernet lifetime.

Within a function, a subnetwork identifier declaration without any computing-space-class specifier or with the `auto` specifier serves as the definition. A new instance of the subnetwork is created on each entry into the block in which it is declared. The subnetwork is discarded when execution of the block ends in any way. Note, that a static subnetwork cannot be declared as a subnetwork of an automatic network.

A subnetwork declaration with specifier `flex` is an `auto` declaration with a suggestion that the creation of the subnetwork declared has the less cost. In addition, there are the same constraints in the use of such subnetworks as for implicitly defined subnetworks, namely: they cannot be used in postfix reduction operations, and on such subnetworks cannot be called network functions.

Subnetwork and relation declarations specify completely the partial order "to be a subnetwork of" on the set of defined subnetworks of the same network. This partial order is built as follows. Let `s1` and `s2` be identifiers of subnetworks of the same network. Then:

- if the declaration of `s1` specifies `s2` as a supernetwork, then `s1` "is a subnetwork of" `s2`;
- `s1` "is a subnetwork of" `s2`, if there is a relation declaration including one of the following relation declarators: `s1<s2`, `s1<=s2`, `s1==s2`, `s2>s1`, `s2>=s1`.

The partial order is defined as reflexive and transitive closure of the relation defined above.

4. Declarations of data objects

4.1 Explicit declaration of distributed data objects

Syntax.

```
<distribution_specifier>:
   [*]
   [host]
   [<identifier>]
   <subnetwork_specifier>
```

Constraints.
The identifier in the distribution specifier should be an identifier of network or subnetwork.

Semantics.
In general, to declare an identifier designating a distributed data object, it is necessary to place in the corresponding declaration just before the identifier the distribution specifier specifying the region of the computing space, over which the declared data object is distributed.

Distribution specifier `[*]` specifies the entire computing space, `[host]` specifies the host-processor, an identifier in brackets specifies a network or explicitly defined subnetwork. Finally, in the case of the subnetwork specifier as a distribution specifier, in addition to explicit declaration of the data object distributed over a subnetwork, the subnetwork is declared implicitly. Note, that the expression in the subnetwork specifier should satisfy the same constraints as for explicit subnetwork declaration (see 3.3).

Example. The declaration

```
int [*]Derror, [Net1]Da[10], *[Net1:I==J]Dpi[5];
```

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

4.2 Explicit declaration of undistributed data objects

Except the cases considered below, to declare an identifier designating an undistributed data object, it is necessary to place in the corresponding declarator just before the identifier one of the following language constructs:

- specifier `[host]`;
- a subnetwork specifier with keyword `parent` instead of an expression;
- a subnetwork specifier of the form `[s:c1==e1&&...&&cN==eN]`, where `s` is an identifier of network or subnetwork having `N` coordinate variables `c1,...,cN,` and `e1,...,eN` are asynchronous integer expressions replicated over `s`;
- a specifier of the form `[s]`, where s is an identifier designating a 1-processor network or subnetwork (if it designates a subnetwork, it should be declared with one of above specifiers as a subnetwork specifier, and if it designates a network, the type of the network should be defined completely in compile time).

Example. The declaration

```
double [host]x;
```

declares the undistributed variable `x` belonging to the host.

4.3 Implicit declaration of data object distribution

A declaration of a formal parameter of network or nodal function shall not include a distribution specifier. A formal parameter of nodal function belongs to the processor executing the function. A formal parameter of a network function is distributed over the region executing the function.

A formal parameter of basic function shall either belong to the host or be distributed over the entire computing space. A declaration of a formal parameter of a basic function without a distribution specifier indicates that the formal parameter is distributed over the entire computing space.

If a data object declaration without a distribution specifier appears out of a function or in the body of a basic function, it declares a data object distributed over the entire computing space.

If a declaration of a data object appears in the body of a nodal function, it shall not include a distribution specifier. If such a declaration is a definition, it specifies an undistributed data object belonging to the processor executing the function. Otherwise, it specifies the corresponding component of a distributed data object, whose external definition exists somewhere in the set of source files that constitutes the entire program. So, in the body of a nodal function any identifier of a data object defined out of the function designates the corresponding component of the data object.

If a declaration of data object without a distribution specifier appears in a network function, it declares a data object distributed over the region on which the function is executed. If this declaration is not a definition, it specifies the corresponding components of a distributed data object, whose external definition exists in the set of files constituting the whole program. So, inside a network function, a identifier of the data object defined out of the function designates the corresponding cutting from this data object.

4.4 Declaration of replicated data objects

The qualifier `repl`, specifying that the values of all components of the corresponding data object are equal each other throughout the lifetime of the data object, is introduced. Such a data object is called *replicated*. The compiler shall warn about all changes of the value of a replicated data object that may violate this property.

The attribute "to be replicated" is associated not only with lvalue but with any expression also.

Example. In the fragment

```
/* Line 1 */    int repl n=10;
/* Line 2 */    void [*]main()
/* Line 3 */    {
/* Line 4 */      net Ring(n) rn;
/* Line 5 */      net Ring(n+1) [rn]rn1;
/* Line 6 */      ...
/* Line 7 */    }
```

the variable `n` is replicated over the entire computing space. The expressions `n` and `n+1`, that are used as topological arguments in lines 4-5, are replicated over the entire computing space also.

5. Expressions

Except postfix reduction operators, a simple assignment, and function calls (except calls to nodal function), all the rest operators are asynchronous.

If an expression is evaluated by a distributed network and is not asynchronous, it shall partially asynchronous in relation to the parent of the distributed network.

An expression, all components of the value of which are equal to each other, is called a *replicated* expression.

5.1 Primary expressions

If an identifier is declared as designating a distributed object, it is an asynchronous expression.

It depends on the context, if a constant or a string literal are distributed expressions. If so, they are asynchronous replicated expressions.

5.2 Asynchronous unary operators

Unary `++`, `--`, `&`, `*`, `+`, `-`, `~`, `!`, `sizeof`, `[]`, `[*]`, `[/]`, `[%]`, `[?<]`, `[?>]`, `[+]`, `[&]`, `[^]`, `[|]` operators and scalar cast operators of the C[] language may have operand whose value is distributed over a region of the computing space. In this case, an operator is performed asynchronously on all components of the value of the operand, and its result is distributed over the same region. In addition, if the operand is an asynchronous expression, the whole expression will be also asynchronous.

Note, that in mpC the `sizeof` operator is not a compile-time operator. At the same time, the compile-time operator `MPC_sizeof`, that yields the size of its operand in the translation environment, is introduced.

The `[]` operator of the C[] language is extended allowing a pointer of any type as an operand. So, if `e` is a primary expression having the type of pointer to type `T` with step `N`, then `e[]` designates a blocked array of the undefined size whose members are of type `T` and allocated in the storage with step `N`.

If a type name in a cast operator specifies a type of pointer to function, it may include the corresponding specifiers specifying attributes of function pointed to.

Example. The type name

```
int [host](*)()
```

specifies the type of pointer to functional component of basic function returning `int`. The type name

```
int [*](*)()
```

specifies the type of pointer to nodal function returning `int`. The type name

```
int [net Web(n,4)](*)()
```

specifies the type of pointer to functional component of network function that has two special formal parameters, the network formal parameter having the type belonging to the network type family `Web(n,4)`.

5.3 Asynchronous binary operators

Both operands of binary `*, /, %, ?<, ?>, +, -, <<, >>, <, >, <=, >=, ==, !=, &, ^, |, &&, ||, *=, /=, %=, ?<=, ?>=, +=, -=, <<=, >>=, &=, ^=, |=, []` operators of the C[] language may be expressions whose values are distributed over any one region of the computing space. In this case, an operator is performed asynchronously on components of values of operands, and its result is distributed through the same region. In addition, if both operands are asynchronous expressions, then the entire expression is also asynchronous.

By definition, two operands are distributed over the same region of the computing space if and only if they satisfy one of the following hypothesizes:

- two regions, over which the operands are distributed, are equivalent subnetworks (see sec. 2.2) of the same network (by definition, a network is a subnetwork of itself);

- the first (second) operand is distributed over a network or a explicitly defined subnetwork `s` (in particular, 1-processor one), the distribution region of the second (first) operand is specified by a distribution specifier of the form `[n:parent]`, and `s` is a parent of `n`.

The language permits the value of one of operands to be distributed over a subregion of the region over which the value of the another operand is distributed (for example, the value of one of the operands may belong to a processor belonging to the region over which the value of another operand is distributed). In this case, the operator is performed asynchronously on the subregion, and its result is also distributed over the subregion.

By definition, region `r1` is a subregion of region `r2` if and only if they satisfy one of the following hypothesizes:

- both `r1` and `r2` are subnetwork of the same network, and it is specified that `r1` is a subnetwork of `r2` (see sec. 2.2);

- there exists region `r3` such that `r1` is an explicitly defined network or subnetwork, `r3` is specified with a distribution specifier of the form `[n:parent]`, `r1` is a parent of `n`, and `r3` is a subregion of `r2`.

The operators `.` and `->` may have the left operand whose value is distributed through a region of the computing space. In this case, an operator is performed asynchronously on all components of the value of the operand, and its result is distributed through the same region. In addition, if the operand is an asynchronous expression, then the entire expression is also asynchronous.

5.4 Asynchronous ternary operators

All operands of the ternary `?:` and `[:]` operators of the C[] language may be expressions whose values are distributed through any one region of the computing space. In this case the operator is performed asynchronously on components of values of operands, and its result is distributed through the same region. In addition, if both the operands are asynchronous expressions, then the whole expression is also asynchronous.

5.5 Cutting operator

Syntax.

```
<cutting>:
   <unary_expression>
   <distribution_specifier> <cutting>
```

Constraints.

The expressions (if any) in the distribution specifier shall be asynchronous (in relation to the corresponding supernetwork) expressions without side effects. The distribution specifier should not be `[*]`.

Semantics.

The cutting operator is specified by the distribution specifier specifying the region (say r1) of the computing space, which should be a subregion of region r2 over which the value of the operand is distributed. The result is the corresponding segment of the distributed value of the operand. The operator is executed asynchronously, and if the operand is an lvalue then the whole expression is an lvalue also.

5.6 Simple assignment

Execution of a simple assignment shall not cause sending unions or bit arrays.

In any case, the operator is performed on the smallest of networks or hard subnetworks enclosing the regions over which the values of the operands are distributed.

The following extensions of the simple assignment operator with distributed operands are admissible.

5.6.1 Asynchronous assignment

The values of both operands are distributed over the same region of the computing space (see sec.5.3). In this case, the operator is performed asynchronously on components of the values of the operands, and its result is distributed over this region. In addition, if both operands are asynchronous expressions, then the whole expression is also asynchronous.

5.6.2 Broadcast/scatter assignment

The left operand is distributed over some region (say R) of the computing space, and the value of the right operand belongs to a processor node of some network or hard subnetwork enclosing R.

5.6.2.1. If the value of the right operand may be assigned without a type conversion to a component of the left operand, then the execution of the operator consists in sending the value of the right operand to each processor of R, where the value is assigned to the corresponding component of the left operand.

5.6.2.2. Otherwise, the value of the right operand shall be a vector, whose elements may be assigned without a type conversion to components of the left operand, and the number of elements of the vector is either equal to the number N of components of the left operand or not specified (the latter is permissible, only if the right operand is a blocked array whose size is not specified). In this case, the execution of the operator consists in sending i-th element of the vector to i-th (in the natural numeration) processor of R, where the element is assigned to i-th component of the left operand for all i from 0 to N-1.

5.6.3 Parallel-send assignment

The left operand and the value of the right operand are distributed over different subnetworks of the same network (say S0 and S1 correspondingly). In this case, S0 and S1 shall consist of the same number N (N>0) of processors and be incomparable in relation to the partial order "to be subnetwork of". Types of components of the operands shall be compatible in relation to assignment and not cause type conversation. The execution of the operator consists in sending i-th (in the natural numeration) component of the value of the right operand to the i-th processor of S0, where it is assigned to i-th component of the left operand for all i from 0 to N-1.

5.6.4 Gather assignment

The value of the right operand is distributed over some region R of the computing space, and the left operand belong to some processor P of some network enclosing R. In this case, the left operand shall be an lvector whose length is either equal to the number N of components of the value of the right operand or not specified, and the type of members of the lvector shall be compatible in relation to assignment with the type of components of the value of the right operand and not cause a type conversation. The execution of the operator consists in sending i-th (in natural numeration) component of the value of the right operand to P, where it is assigned to i-th member of the left operand for all i from 0 to N-1.

5.7 The coordof operator

Syntax.

```
<coordinate_expression>:
   <identifier> coordof <unary_expression>
```

Semantics.

The left operand is a coordinate name associated with a region of the computing space over which the value of the right operand is distributed. The result is an integer value distributed over this region each component of which is equal to the value of the specified coordinate of the processor to which the component belongs. The right operand is not evaluated, but only used to specify the region of the computing space.

5.8 Postfix reduction operators

Postfix unary `[*]`, `[+]`, `[?<]`, `[?>]`, `[-]`, `[&]`, `[^]`, `[|]`, `[&&]`, `[||]` operators are introduced. The result of an operator is distributed over the same region of the computing space as the value of the operand. Note, that the region should be either a network or a hard subnetwork.

If the region is a single one (that is, undistributed), all the components of the result are identical and equal to the result of the corresponding prefix reduction operator performed on the vector comprising the components of the value of the operand. If this region is distributed, then execution of the operator is divided into a set of independent operations each of which is executed by its own single region belonging to the set specified by the corresponding distributed region.

In addition, two binary `[<>]` , `[><]` operators are introduced. Let `E` be an expression whose value is distributed through some region R. Let `F` be an expression whose value is a pointer, distributed

through R, to a nodal function implementing a binary operator of the same type as a component of the value of E. Then, under hypothesis that mpC includes the associative and commutative binary operator op implemented by the function which F points to, the E[<F>] expression is equivalent to E[op]. Similarly, under hypothesis that mpC includes the associative but non-commutative binary operator op implemented by the function which F points to, the expression E[>F<] is equivalent to E[op].

5.9 Function call

Syntax.

```
<function_call>:
   <special_argument_expression>(opt)
      <function_designation> ( <ordinary_argument_list>(opt) )
<special_argument_expression>:
   ( [ ( <topological_argument_list>(opt) ) <idendifier> ] )
```

Semantics.

If the function designation has type "pointer to functional component of basic function", its value shall be distributed over the entire computing space. In this case, the special argument expression shall not appear, and the value of an ordinary argument (if any) shall either belong to the host or be replicated over the entire computing space. In this case, the function call shall be an overall expression, and the returning value (if any) shall be distributed over the entire computing space.

If the function designation has type "pointer to function" (without additional attributes) or type "pointer to nodal function", the special argument expression shall not appear. In this case, the value of the function designation and the values of ordinary arguments (if any) shall either belong the same processor (say P) or be distributed over the same region of the computing space (say R). In the first case, the function call shall be performed on processor P, and the returning value (if any) shall belong to P also. In the second case, the function call shall be performed on the region R, and the returning value (if any) shall distributed over P also. In addition, if the ordinary arguments and the function designation are asynchronous expressions and the function designation has type "pointer to nodal function", then the function call is an asynchronous expression also.

If the function designation has type "pointer to functional component of network function", then its value shall be distributed over a region of the computing space enclosing the region R that is specified by the identifier in the special argument expression. The values of all arguments (if any) shall be distributed over R. The value of a scalar topological argument (if any) shall be replicated over R. In this case, the function call shall be performed on R, and the returning value (if any) shall be distributed over R.

6. Statements

A statement may be executed either on a single processor, or on a region of the computing space (a network or a subnetwork), or on a set of regions, or on the entire computing space.

If statement S0 follows statement S1 and the sets of processors executing the statements are disjoint, then it depends on the compiler whether the statements are executed in parallel. Otherwise, they are executed as if all computations specified in statement S0 end before any computation specified in statement S1begins. But in the latter case, the compiler can also overlap executions of these statements, if it does not break functional semantics of their successive execution.

By definition, a set of processors executing a statement, execution of which causes the creation of a network, includes all free (at the moment of execution of the statement) processors of the computing space. Therefore, if execution both S0 and S1 causes creation of networks, then the intersection of the sets of processors executing these statements can not be empty (although the intersection can not be computed in compile time).

6.1 Labeled statements

New kind of labeled statements is introduced in mpC.

Syntax.

```
<labeled_statement>:
   <distribution_specifier> ':' <statement>
```

Constraints.

Only jump statements may be labeled by the specifier `[*]`.

Only a sequential asynchronous statement may be labeled by the distribution specifier. The resulting labeled statement should be asynchronous.

Semantics.

If a statement labeled by a distribution specifier is syntactically built from expressions and substatements, it is equivalent to the statement obtained from the initial statement by both applying the corresponding cutting operator to every identifier appearing in the expressions and labeling the substatements by the distribution specifier. If the result of recursive application of the described procedure is a statement, that can not be detected in compile time as an asynchronous statement, then the initial labeled statement is not correct.

If a jump statement is labeled by the distribution specifier, it is divided into a set of independent jump statements each of which is executed by the corresponding processor of the region specified by the distribution specifier.

6.2 Compound statement (block)

Statements that are grouped into a block may be distributed. If no network or hard subnetwork is defined in a block, and all the statements are asynchronous and distributed over the same region, then the block is also asynchronous.

6.3 Expression statement

The expression in an expression statement may be distributed. If it is asynchronous, the expression statement is also asynchronous.

6.4 Selection statements

Syntax.

```
<selection_statement>:
   if ( <expression> ) <statement>
   if ( <expression> ) <statement> else <statement>
   switch ( <expression> ) <statement>
```

Semantics.

If the value of a controlling expression in a selection statement is undistributed, the selection statement selects among a set of statements depending on this value. Execution of such a selection statement includes evaluation of its controlling expression and sending the value of the controlling expression to all processors of the least set of networks enclosing the set of regions taking part in the execution of the statements among which selection is done.

If the value of the controlling expression of the selection statement is distributed over a region of the computing space (in particular, over the entire computing space), the statements, among which the selection is done, shall be either asynchronous statements distributed over the same region, or partially asynchronous in relation to this region statements. If the controlling expression and these statements are asynchronous, the selection statement is also asynchronous, and it is divided into a set of independent selection statements each of which is executed by the corresponding processor of the region. If the controlling expression is asynchronous, and the statements, among which the selection is done, are partially asynchronous, then the selection statement is partially asynchronous, and its execution is divided into parallel execution of a set of selection statements, the value of controlling expression each of which is undistributed. For example, the `if` statement in the fragment

```
net Ring(5) r5;
net Ring(3) [r5]r3;
double [r5]dx, [r3]ddx;
...
if(dx>0)
  ddx=dx;
else
  ddx=-dx;
...
```

is partially asynchronous in relation to the network `r5` and is divided into parallel execution of five `if` statements, each of which is executed on own component of the distributed network `r3`.

Finally, if the value of the controlling expression and the statements, among which the selection is done, are distributed over the same region and at least one of these statements is neither asynchronous or partially asynchronous in relation to this region, then the controlling expression shall be replicated. Otherwise, the behavior is undefined.

6.5 Iteration statements

Syntax.

```
<iteration_statement>:
   while ( <expression> ) <statement>
   do <statement> while ( <expression> ) ;
   for ( <expression>(opt>;
         <expression>(opt) ;
         <expression>(opt) ) <statement>
```

Semantics.

6.5.1 The `while` and `do` statements

If the value of a controlling expression in a `while` or `do` statement is undistributed, the iteration statement causes the loop body to be executed repeatedly until the controlling expression evaluates to zero. Execution of such an iteration statement includes broadcasting the value of the controlling expression to all processors of the least set of networks enclosing the set of regions taking part in the evaluation of the controlling expression and the execution of the body loop.

If the value of the controlling expression is distributed over a region of the computing space (in particular, over the entire computing space), then the loop body shall be either an asynchronous statement, distributed over the same region, or a partially asynchronous in relation to this region statement. If the controlling expression and the loop body are asynchronous, then the iteration statement is also asynchronous and divided into a set of independent iteration statements each of which is executed by own processor node of the region. If the controlling expression is asynchronous, and the loop body is partially asynchronous, then the iteration statement is partially asynchronous, and its execution is divided into parallel execution of a set of iteration statements, the value of controlling expression each of which is undistributed.

Finally, if the value of the controlling expression and the loop body are distributed over the same region, but the loop body is neither asynchronous or partially asynchronous in relation to this region, then the controlling expression shall be replicated. Otherwise, the behavior is undefined.

6.5.2 The `for` statement

Except for the behavior of the `continue` statement in the loop body, the statement

```
for ( expression-1; expression-2; expression-3 ) statement
```

and the statement

```
{
  expression-1;
  while ( expression-2 ){
    statement
    expression-3;
  }
}
```

are equivalent.

6.6 Jump statements

The mpC language constrains essentially the usage of jump statements.

If a jump statement is labeled (explicitly or implicitly) by a distribution specifier, it is distributed over the region specified by the specifier.

If a jump statement not labeled by a distribution specifier appears in a network function, it is distributed over the region on which the function is called.

If a jump statement not labeled by a distribution specifier appears in a basic function, it is overall (that is, executed on the entire computing space).

A jump statement, that appears in a nodal function, shall not be labeled by a distribution specifier and executed by the processor executing the function.

6.6.1 The `goto` statement

Constraints.

An undistributed `goto` statement and the label used in it shall appear somewhere inside an undistributed statement executed by the same processor as the `goto` statement.

A distributed `goto` statement is considered to be correct only in the following two cases:

- both the `goto` statement and the statement labeled by the label used in the `goto` statement are (high-level) elements of the statement list constituting a block, and both of them are distributed over the region of the computing space executing the block;
- both the `goto` statement and the label used in it appear somewhere inside an asynchronous statement.

Semantics.

A `goto` statement causes an unconditional jump to the named label in the current function.

6.6.2 The `continue` statement

Constraints.

An undistributed `continue` statement shall appear only inside the loop body of an undistributed iteration statement executed by the same processor as the `continue` statement.

A distributed `continue` statement shall appear only inside the loop body of an asynchronous iteration statement.

Semantics.

A `continue` statement causes a jump to the loop-continuation portion of the smallest enclosing iteration statement; that is, to the end of the loop body.

6.6.3 The `break` statement

Constraints.

An undistributed `break` statement shall appear either in the switch body of an undistributed switch statement or in the loop body of an undistributed iteration statement executed by the same processor as the `break` statement.

A distributed `break` statement shall appear either in the switch body of an asynchronous switch statement, or in the loop body of an asynchronous iteration statement, or in the body of a `fan` statement, or in the body of a `pipe` statement.

Semantics.

A `break` statement terminates execution of the smallest enclosing switch or iteration statement, or terminates execution of the body of the smallest enclosing `fan` statement, or terminates execution of the smallest enclosing `pipe` statement. The latter means that the processor executing the `break` statement terminates its execution of the `pipe` statement and sends the signal of preschedule termination to processors taking part in the execution of the `pipe` statement.

6.6.4 The `return` statement

Constraints.

A `return` statement shall not be labelled explicitly by a distribution specifier.

A `return` statement shall not appear in any place of a function body where it may be executed in parallel with another statement of the function body.

A `return` statement with an expression shall not appear in a function returning type `void`.

Semantics.

A `return` statement terminates execution of the current function and returns control to its caller. A function may have any number of `return` statements, with or without expressions. If a return statement with an expression is executed, the value of the expression is returned to the caller. If the expression has a type different from that of the function in which it appears, it is converted as if it were assigned to an object of that type. If a `return` statement without an expression is executed, and the value of the function call is used by the caller, the behavior is undefined. Reaching the } that terminates a function is equivalent to executing a `return` statement without an expression.

7. Library and embedded functions

Library and embedded nodal, basic and network functions support the development of more effective mpC programs. They also make debugging easier. All corresponding declarations are contained in header file <mpc.h>. The library is still under development, so changes in the set of functions are possible.

7.1 Nodal library functions

7.1.1 Function MPC_Printf

Synopsis

```
#include <mpc.h>
int MPC_Printf( const char* format, ... );
```

Description

`MPC_Printf` allows to output formatted strings to `stdout` on the host virtual processor from any virtual processor of the computing space. Syntax strictly follows standard `printf` syntax.

Returned value

The function returns 0 if all is OK, and non-zero otherwise.

7.1.2 Function MPC_Wtime

Synopsis

```
#include <mpc.h>
double MPC_Wtime(void);
```

Description

`MPC_Wtime` returns floating-point number of seconds, representing elapsed wall-clock time since some time in the past. The "time in the past" is guaranteed not to change from the program start to finish, but it may be different on different virtual processors of the computing space.

7.1.3 Function MPC_Total_nodes

Synopsis

```
#include <mpc.h>
int MPC_Total_nodes(void);
```

Description

`MPC_Total_nodes` returns the total number of virtual processors in the computing space.

7.1.4 Function MPC_Processors_static_info

Synopsis

```
#include <mpc.h>
int MPC_Processors_static_info (
    int *num_of_processors, double **relative_performance);
```

Description

After a call to `MPC_Processors_static_info` object `*num_of_processors` will contain the total number N of physical processors of the underlying distributed memory machine. Object `*relative_performance` will contain a pointer to the initial element of N-element double array, containing relative performances of the processors.

Returned value

The function returns 0 if all is OK, and non-zero otherwise.

7.1.5 Function MPC_Abort

Synopsis

```
#include <mpc.h>
int MPC_Abort( repl errcode );
```

Description

`MPC_Abort` tries to abort all processes in the computing space. The value of `errcode` will be returned to command shell.

Return value

Ignored.

7.2 Basic library functions

7.2.1 Function MPC_Exit

Synopsis

```
#include <mpc.h>
int [*]MPC_Exit( repl exitcode );
```

Description

`MPC_Exit` terminates execution of a mpC program. A call to `MPC_Exit` is a point of global synchronization (i.e. all virtual processors from the computing space call it in synchronous manner). The value of `exitcode` will be returned into command shell.

Return value

Ignored.

7.2.2 Function MPC_Global_barrier

Synopsis

```
#include <mpc.h>
int [*]MPC_Global_barrier(void);
```

Description

A call to `MPC_Global_barrier` is a point of global synchronization.

Return value

The function returns 0 if all is OK, and non-zero otherwise.

7.3 Network library functions

There is the following topology declaration

```
nettype SimpleNet(n) { coord I=n; };
```

in <mpc.h>.

7.3.1 Function MPC_Barrier

Synopsis

```
#include <mpc.h>
int [net SimpleNet(n) w] MPC_Barrier( void );
```

Description

A call to `MPC_Barrier` is a point of synchronization of all virtual processors of `w`.

Return value

The function returns 0 if all is OK, and non-zero otherwise.

7.4 Embedded network functions

Embedded network functions look like library network functions, but the compiler knows their semantics and treats them in a special way. They are similar to C++ templates.

7.4.1 Function MPC_Assign

Synopsis

```
#include <mpc.h>
int [net SimpleNet(n) w] MPC_Assign( repl const *source,
                                     <s_type> *s_buffer,
                                     int const s_step,
                                     repl const count,
                                     repl const *destination,
                                     <d_type> *d_buffer,
                                     int const d_step);
```

Description

`MPC_Assign` sends `count` elements of type `<s_type>` from a virtual processor of `w`, the coordinate of which is equal to `*source`, to a virtual processor of `w`, the coordinate of which is equal to `*destination`. Parameters `s_buffer` and `s_step` are significant only at the sender and specify the initial address of the source buffer and the step between elements in the buffer, respectively. Similarly, parameters `d_buffer` and `d_step` are significant only at the receiver and specify initial the address of the receive buffer and the step between elements in the buffer, respectively. For every element to send the matching element to receive must be specified. In other words, types `<s_type>` and `<d_type>` must contain equivalent sequences of basic types. If this condition is not satisfied, the compiler should detect such a situation as erroneous.

The value of parameter `n` is ignored, so the corresponding actual parameter may be arbitrary integer (for example 0).

Return value

The function returns 0 if all is OK, and non-zero otherwise.

7.4.2 Function MPC_Bcast

Synopsys

```
#include <mpc.h>
int [net SimpleNet(n) w] MPC_Bcast(
    repl const *source,
    <s_type> *s_buffer,
    int const s_step,
    repl const count,
    <d_type> *d_buffer,
    int const d_step);
```

Description

`MPC_Bcast` sends `count` elements of the type `<s_type>` from a virtual processor of `w`, the coordinate of which is equal to `*source`, to all virtual processors (including the sender) in `w`. Parameters `s_buffer` and `s_step` are significant only at the sender and specify the initial address of the source buffer and the step between elements in the buffer, respectively. Parameters `d_buffer` and `d_step` specify the initial address of the receive buffer and the step between elements in the buffer, respectively. For every element to send the corresponding element to receive must be specified. In other words, types `<s_type>` and `<d_type>` must contain equivalent sequences of basic types. If this condition is not satisfied, the compiler should detect this situation as erroneous.

The value of parameter `n` is ignored, so the corresponding actual parameter may be arbitrary integer (for example 0).

Return value

The function returns 0 if all is OK, and non-zero otherwise.

7.4.3 Function MPC_Scatter

Synopsis

```
#include <mpc.h>
int [net SimpleNet(n) w] MPC_Scatter(
    repl const *source,
    <s_type> *s_buffer,
    int const *disps,
    int const *lengths,
    const count,
    <d_type> *d_buffer);
```

Description

`MPC_Scatter` scatters the values of a number of elements of type `<s_type>` from a virtual processor of `w`, the coordinate of which is equal to `*source`, over all virtual processors of `w`. Parameter `s_buffer` is significant only at the sender and specifies the initial address of the source buffer. Parameters `disps` and `lengths` are significant only at the sender, and `disps` points to an integer array, the i-th element of which specifies the displacement (relative to `s_buffer`) from which `lengths[i]` elements will be taken to send to the i-th virtual processor of `w`.

Parameter `d_buffer` specifies the initial address of the receive buffer. Parameter `count` specifies the number of elements in the receive buffer. For every element to send the matching element to receive must be specified. In other words, types `<s_type>` and `<d_type>` must contain equivalent sequences of basic types. If this condition is not satisfied, the compiler should detect such a situation as erroneous.

The value of parameter `n` is ignored, so the corresponding actual parameter may be arbitrary integer (for example 0).

Return value

The function returns 0 if all is OK, and non-zero otherwise.

7.4.4 Function MPC_Gather.

Synopsis

```
#include <mpc.h>
int [net SimpleNet(n) w] MPC_Gather(
    repl const *destination,
    <d_type> *d_buffer,
    int const *disps,
    int const *lengths,
    const count,
    <s_type> *s_buffer);
```

Description

`MPC_Gather` gathers on a virtual processor `w`, the coordinate of which is equal to the value of `*destination`, a number of `<s_type>` elements from all virtual processors (including the receiver) of `w`. Parameter `d_buffer` is significant only at the receiver and specifies the initial address of the receive buffer. Parameters `displs` and `lengths` are also significant only at the receiver, and `displs` points to an integer array, i-th element of which specifies the displacement (relative to `d_buffer`) to which `lengths[i]` elements to receive from the i-th virtual processor of `w` will be placed.

Parameter `s_buffer` specifies the initial address of the send buffer. Parameter `count` specifies the number of elements in the send buffer. For every element to receive the matching element to send must be specified. In other words, types `<d_type>` and `<s_type>` must contain equivalent sequences of basic types. If this condition is not satisfied, the compiler should detect such a situation as erroneous.

The value of parameter `n` is ignored, so the corresponding actual parameter may be an arbitrary integer (for example 0).

Return value

The function returns 0 if all is OK, and non-zero otherwise.

8. Implementation restrictions

The current implementation does not support full implementation of parallel statements `fan`, `par` and `pipe`, so their descriptions were omitted.

In addition, the following features are not supported by the current implementation:

- negative steps in arrays;
- user-defined postfix reduction operations;
- vectors as return values;
- `void` as a processor node type;
- any expression, other than an identifier, in the left expression list of a single link declarator, if the expression contains a coordinate variable (including a free coordinate variable);
- 2-operand versions of `?:`;
- `[:]`, `[#]` and vector forming C[] operators;
- compile-time checking the correctness of special labels in the form of distribution specifier;
- compile-time checking the correctness of distributed `goto`.