Institute for System Programming

mpC

language specification

December 2002

Table of Contents

mpC

Introduction

The mpC language was developed to support efficiently portable modular parallel programming for 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 networks. The mpC programming environment uses the topological information in run time to ensure the efficient execution of the application on the underlying 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. When programming 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 sending and receiving 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.

Basic Concepts

In order to understand how to implement parallel applications in mpC, it is helpful to understand some of the key concepts of the language. In this chapter the following basic concepts of mpC are described:

- Computing space and network objects. When programming in mpC, user could imagine that there is some accessible set of virtual processors connected with links, and he can manage this resource allocating network objects (regions) there. Managing network objects in mpC is similar to managing data objects in C.
- Subnetworks. Subnetwork is another key feature of mpC language. It allows a user to specify sub regions of already allocated regions of the computing space.
- Distribution of data. In mpC there is introduced the notion of distributed data objects. 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.
- Distribution of computations. In mpC any expression can be evaluated not only on one processor but also on a region of computing space or on entire computing space.
- Network functions. To support modular parallel programming as well as the writing of libraries of parallel programs, so-called network functions are introduced in mpC. It is a function that is called and executed on some region of the computing space.
- Pointer to function. As mpC function call can be distributed then the distributed function call shall include the distributed pointer to the corresponding functional components.

Computing space and network objects

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

In mpC, the notion of *computing space* is defined as a set of typed virtual processors connected with links of different 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 with them. A processor of the vector type can perform vector operations more efficiently. Finally, most common processors are of the scalar type.

Besides, 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 destination processor. There can exist no more than one directed link from source to destination. A link has an

attribute called length that characterizes its bandwidth. A pair of opposite directed links between two processors could be considered as a single undirected link.

The basic notion of the mpC language is a so-called *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 that 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 a 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 the 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 and 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 ints 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 Distribution of data, Explicit declaration of distributed data objects, Explicit declaration of undistributed data objects, Implicit declaration of data objects and Declaration of replicated data objects).

Besides the network type, a parameterized family of network types called topology or *generic network type* can be declared. 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 can be positive integer. Any operand in the expression can consist only of coordinate variables, constants and generic parameters (if any). If the value of the expression is equal to 1, the power specifier can be omitted. It is meant that in the framework of the same network-type declaration any performance specifier with the keyword *fast* specifies more powerful processor than a performance specifier with the keyword *slow*. It is also supposed that the greater the value of the expression in a power specifier is the more performance is ascribed to the processor. Note, that in this case the following simplified form of line 4

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

can be used (see also Node declaration).

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 derived not only statically, but also dynamically. 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, 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 Distribution of computations, Network functions) being in scope of its identifier (for global static networks). Once created, the static network exists until the entire program terminates.

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 terminates.

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 identifiers *r3* and *r5* of static network objects have 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 has block scope. (Note, that the header of the definition of the function *f* includes the construct [*] that specifies the kind of the function (see sections Distribution of computations, Network functions)).

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 objects 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, while the network *r7* has the processor of the network *r6* 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 incorrect, while the equivalent declaration

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

is correct.

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. This can be achieved by explicit or implicit definition of a *subnetwork* of the existing network object.

Unlike an implicitly defined subnetwork, an explicitly defined subnetwork has a name introduced by the subnetwork declaration. A subnetwork declaration that causes computing space to be reserved for the subnetwork named by an identifier is just an explicit subnetwork definition. Computing space duration of explicitly defined subnetwork, scope and linkage of its identifier are specified in the same way as 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 the 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 the corresponding coordinates. The equivalent interpretation is that processor of the void type has no memory and can execute no operations.

The topology *Web* corresponds to the web structure networks with n radial threads, each of them stringed with m-1 normal speed scalar processors. In the center of the web there is placed a fast scalar processor. That means that the computation load of this processor will be more intensive than those of the other processors. Radial links between processors are of normal length (the attribute length is equal to 0), while the 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 cannot be specified explicitly, while 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 the *web10x20* that constitute the subnetwork. Namely, a processor 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 the so-called *natural numeration* of processors from 0 to n-1, where n is the number of processors, can be defined. This numeration is determined by lexicographic ordering of 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. It can be specified with a subnetwork declaration, similar to the declaration in line 22 that specifies that *seastar5x5* is a subnetwork of *seastar10x10*.

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 compared with the creation of hard networks.

Distribution of data

The concept 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 region 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) 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 ints 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 concept 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.

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 parallel with the evaluation of the overall expression.

A special type of the distributed expression called an *asynchronous expression* is introduced. In fact, 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 consists of such operators only, and all of them 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 concept 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.

Basic function is a function that is called to by all the virtual processors in the computing space. 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 is a function that can be called to by any virtual processor and is executed by the calling virtual processor only. Call to a *nodal function* is asynchronous. 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 types 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.

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

In order to support modular parallel programming as well as writing libraries of parallel programs, in addition to basic and nodal functions, the 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 parameterized. 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 parameterized 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, access to an element via its indices). The number of indices of the latter and their ranges (can be defined dynamically) are detected by the compiler from the declaration of the corresponding topology.

When calling 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.

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 can be treated as a distributed call to undistributed functions called *functional components* of the distributed call. Therefore, the distributed function call includes the distributed pointer to the corresponding functional components.

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

Managing the computing space

In mpC the notion of the *computing space* is defined as a set of virtual processors of different performances connected with links of different communication speeds accessible to the user for management by means of allocating networks and subnetworks. Every network defined in mpC has a type. Network type specifies number and performances of virtual processors, links between these processors and communication speeds of links.

The following declarations are used for computing space management:

- Network type declaration
- Network declarations
- Subnetwork declaration

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 can 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 parameterized (generic). A declaration of generic network type called also a topology should 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 generic parameter declarator specifies the number of indices and their ranges. 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 can include the specifier extern or static.

A network-type declaration that also causes compiler to generate components of the target program, that provide 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.

EXAMPLE

```
extern nettype _3Dim_net (P,Q,l,k,grid[P*Q])
{
  coord I=P,J=Q;
  node
  {
    I>=0&&J>=0: grid[I*Q+J];
  };
  parent [l,k];
};
```

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

Semantics

A coordinate declaration declares a coordinate system, to which processor nodes of the declared network are related. 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, to which a network containing up to 100*10*N nodes may be related.

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

Either the performance specifier or the node type should 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 must be integer. The operands in the expression should consist only of coordinate variables, constants and generic parameters.

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

Semantics

A node declaration relates 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 keyword *fast* specifies more powerful processor than a performance specifier with the keyword *slow*. 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.

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

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

The expression in the link-length specifier must be integer. The operands in the expression can 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 permitted 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 of a 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];
    }
   ...
};
```

Parent node declaration

Syntax

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

Constraints

An expression in the expression list must be integer. The operands in the expression must consist only of constants and generic parameters of the generic network type (if any). The number of expressions in the expression list should 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 (remember, 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.

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 can be used as computing-space-class specifiers in a network declaration.

Semantics

A network declaration introduces a set of identifiers, which 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 in 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 in one of the source files constituting the source code of the entire program there exists an external definition for the given network identifier. Such a network declaration cannot 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.

EXAMPLE

```
net HeteroGrid(p,q) w;
```

Network type specifier

Syntax

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

Constraints

All expressions in the argument expression list must be integer. If the network type specifier is a part of an external network declaration, the expressions must be constant. Otherwise, they must either be replicated over the entire computing space or belong to (or replicated over) the parent of the network declared.

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 parameterized (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 consists 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.

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 must not appear in a declaration with the typedef specifier.

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

The expression in the subnetwork specifier must 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.

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

Syntax

```
<subnetworks_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>
```

Constraints

Only extern, static, auto or flex may be used as computing-space-class specifiers in a 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 is 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 cannot 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 network functions cannot be called on such subnetworks.

Subnetwork declarations specify 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. The partial order is defined as reflexive and transitive closure of this relation.

Regions and subregions

A set of nodes allocated as a network or specified as a subnetwork is called a region. The notion of a subregion is defined as follows:

1. Any region is a subregion of the whole computing space.
2. The region A is a subregion of region B if A is declared as a subnetwork of B with subnet clauses.
3. The region declared as [host] is a subregion of any network, declared with host as a parent.

The reflexive and transitive closure of rules 1-3 defines a relation "to be a subregion".

There is a special kind of regions, so-called single-node region designed especially for accessing regions consisting of exactly one node. The notion of a single-node predicate is introduced for these purposes:

```
< single_node_predicate>: <coordinate_predicate>
          < single_node_predicate> &&
                        <coordinate_predicate>

 <coordinate_predicate>:
          <identifier> == <coordinate_expr>
          <coordinate_expr> == <identifier>
```

Here <identifier> is a name of a coordinate variable and <coordinate_expr> should not contain identifiers of the coordinate variables.

1. Regions are called single-node in the following three cases:
2. The region is declared as [host].
3. The region is declared as [<network_name>:parent].
4. The region is a subnetwork whose all supernetworks are declared with single-node predicate. Each coordinate variable shall appear exactly one time in the set comprising all predicates of those supernetworks.

EXAMPLE. Consider the following declaration of network type NT an a network mynet of this type:

```
nettype NT
{
  coord I=3,J=3;
  parent [0,0];
};


net NT mynet;
subnet [mynet : I == 0] w;
```

The following subnet declarations define single-node regions:

```
subnet [mynet : parent] w1;
subnet [mynet : I == 1, J == 1] w2;
subnet [w : J == 0] w3;
```

The following declaration introduces the region w4, which consists of one node but is not a single-node region:

```
subnet [mynet : I < 1, J < 1] w4;
```

Declarations of data objects

A “declaration” specifies the interpretation and attributes of a set of identifiers. A declaration that also causes storage to be reserved for the object or function named by the identifier is called a “definition.”

In mpC the notion of *distributed data object* is introduced. Data object distributed over a region of the computing space is comprised of a set of components of the same type i.e. each virtual processor of the region holds one component of the object distributed over this region. Replicated data object is such a distributed data object that values of all components of this object equal to each other.

Constraints

All the expressions in an initializer for an object that has static storage duration or in initializer list for an object of aggregate type or in initializer for a distributed object shall be constant expressions or string literals.

In this chapter there are described how data objects can be declared in mpC:

- Explicit declaration of distributed data objects
- Explicit declaration of undistributed data objects
- Implicit declaration of data objects
- Declaration of replicated data objects

Explicit declaration of distributed data objects

Syntax

```
<distribution_specifier>:
    [*]
    [host]
    [<identifier>]
    [<network_identifier>:parent]
```

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, and identifier in brackets specifies a network or explicitly defined subnetwork. If *parent* is specified the corresponding identifier should belong to the network.

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 *ints* 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.

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.

Implicit declaration of data objects

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 can 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, external definition of which 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.

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 data object is called *replicated*. The compiler shall warn about all changes of the value of a replicated data object that could 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, which are used as topological arguments in lines 4-5, are replicated over the entire computing space also.

Expressions

This chapter describes mpC expressions. Expressions are sequences of operators and operands that are used for one or more of these purposes:

- Computing a value from the operands.
- Designating objects or functions.
- Generating "side effects." (Side effects are any actions other than the evaluation of the expression — for example, modifying the value of an object.)

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

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

The following topics are covered in this chapter:

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.

Asynchronous unary operators

Unary ++, --, &, *, +, -, ~, !, sizeof, [], [*], [/], [%], [?<], [?>], [+], [&], [^], [|] operators and scalar cast operators of the C[] language can have operand with the value distributed over some region of the computing space. In this case, the 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.

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)*.

Asynchronous binary operators

Both operands of binary *, /, %, ?<, ?>, +, -, <<, >>, <, >, <=, >=, ==, !=, &, ^, |, &&, ||, *=, /=, %=, ?<=, ?>=, +=, -=, <<=, >>=, &=, ^=, |=, [] operators of the C[] language can be expressions with the values distributed over any region of the computing space. In this case, an operator is applied asynchronously to 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.

The language permits the value of one of the operands to be distributed over a subnetwork of the region over which the value of 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 subnetwork, and its result is also distributed over the subnetwork.

The operators . and -> may have the left operand, value of which is distributed over a region of the computing space. In this case, an operator is applied asynchronously to all the 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.

Asynchronous ternary operators

All operands of the ternary ?: and [:] operators of the C[] language can be expressions, values of which are distributed over any 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.

Cutting operator

Syntax

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

Constraints

The expressions (if any) in the distribution specifier must be asynchronous (in relation to the corresponding supernetwork) expressions without side effects. The distribution specifier shall either have the form "[identifier]" or specify the single node subnetwork.

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.

Simple assignment

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

Let the left and right operands of an assignment to be distributed over regions R1

and R2 respectively. The operator is performed on the region R where R, R1, R2

satisfy the following conditions:

1. R is a network or a hard subnetwork.
2. R1 is a subnetwork declared with a subnetwork specifier of the form [R:expression] or R is a network with a parent R1 or R and R1 are the same regions.
3. R2 is a subnetwork declared with a subnetwork specifier of the form [R:expression] or R is a network with a parent R2 or R and R2 are the same regions.

If there is no region R satisfying these conditions mentioned above the assignment is invalid.

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

- Asynchronous assignment
- Broadcast/scatter assignment

- Parallel-send assignment
- Gather assignment

Operands for broadcast/scatter, parallel-send and gather assignment must be distributed over subnetworks of one network.

Asynchronous assignment

The values of both operands are distributed over the same region of the computing space (see Asynchronous binary operators). 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.

Broadcast/scatter assignment

Distribution of the left operand is some region R and distribution of the right operand is a single-node region.

If the component of the object designated by left operand may be assigned to the value of right operand without type conversion, then the execution of the operator consists in storing the value of the right operand in each component of the object designated by the left operand.

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 In this case, the execution of the operator consists in storing the value of i-th element of the vector in the i-th component of the object designated by the left operand for all i from 0 to N-1.

EXAMPLE 1 illustrates Broadcast assignment

```
/*Line 1*/net SimpleNet(N) cube;
/*Line 2*/int [cube]a, [cube]b;
            …
/*Line 3*/{
/*Line 4*/flex subnet [cube:I == ri] cur_pr;
            …
/*Line 5*/a = [cur_pr]b;
/*Line 6*/}
            …
```

Line 1 contains definition of network cube. Line 2 contains definition of two variables a and b distributed over cube network. Line 4 contains definition of flexible subnetwork cur_pr. This network consists of one node. Line 5 contains broadcast itself. In fact the expression at Line 5 means that the value of the distributed variable b, which belongs to the node [cube:I == ri], is broadcasted through the whole network cube and the value of b is assigned to a on each node of the network cube.

EXAMPLE 2 illustrates scatter assignment

```
/*Line 1*/net SimpleNet(4) cube;
/*Line 2*/int [cube]a[5], [cube:parent]b[4][5];
            …
/*Line 3*/a [] = b [];
```

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) consisting of the same number of nodes, say N. Types of components of the operands shall be compatible in relation to assignment and not cause type conversion. For all i from 0 to N-1 the i-th (in the natural numeration) component of the value of the right operand is is stored in the object designated by the i-th component of the distributed object designated by the right operand.

EXAMPLE 1

```
/*Line 1*/net SimpleNet(4) cube;
/*Line 2*/int [cube]a, [cube]b;
            …
/*Line 3*/{
/*Line 4*/flex subnet [cube:I == ri] cur_pr;
/*Line 5*/flex subnet [cube:I == cur_i] partner;
/*Line 6*/[cur_pr]a = [partner]b;
/*Line 7*/}
```

Line 1 contains network cube definition. At line 2 two distributed over network cube variables are defined. Lines 4-5 contain definitions of flexible subnets. Line 6 contains the assignment itself. The component of b, which belongs to virtual processor called partner is sent to the virtual processor called cur_pr and value of b is assigned to a on the virtual processor called cur_pr.

EXAMPLE 2

```
/*Line 1*/nettype _2Dim_Net (N) { coord I = N, J = N; };
            …
/*Line 2*/net _2Dim_Net (2) _2dn;
/*Line 3*/int [_2dn]a, [_2dn]b;
/*Line 4*/{
/*Line 5*/subnet [_2dn:I == 0] sbn1;
/*Line 6*/subnet [_2dn:I == 1] sbn2;
            …
/*Line 7*/ [sbn1]a = [sbn2]b;
            …
/*Line 8*/}
```

At line 1 _2Dim_Net network type is declared. Line 2 contains declaration of network _2dn. At line 3 two distributed over _2dn network variables – a and b – are declared. At lines 5-6 there are declared two hard subnets – sbn1 and sbn2. Line 7 contains parallel-send assignment. The component of distributed variable a, which belongs to the virtual processor [_2dn:I==0&&J==0], will be assigned to the value of component of distributed variable b, which belongs to the virtual processor [_2dn:I==1&&J==0]. Similarly, component of distributed variable a, which belongs to the virtual processor [_2dn:I==0&&J==1], will be assigned to the value of component of distributed variable b, which belongs to the virtual processor [_2dn:I==1&&J==1].

Gather assignment

The value of the right operand is distributed over some region *R* of the computing space, and the left operand is distributed over a single-node region. 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 elements 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 storing the i-th component of the value of the right operand to the i-th element of the vector designated by the left operand.

EXAMPLE

```
/*Line 1*/net SimpleNet(4) sn;
/*Line 2*/int [sn:parent]a [4], [sn]b;
            …
/*Line 3*/a [] = b;
            …
```

Line 1 contains network sn definition. At line 2 two distributed over network sn variables are defined. Line 3 contains gather assignment. i-th member of a will be assigned to the i-th component of distributed value of b.

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.

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. Each component of the resulting value equals to the value of the applying of reduction operator with the same operator's sign to the vector consisting of components of the operand.

EXAMPLE

```
/*Line 1*/net SimpleNet (5) sn;
/*Line 2*/int [sn]a [4], [sn]b [4];
            …
/*Line 3*/a [] = b [][+];
            …
```

This example illustrates postfix reduction operator [+]. For all i from 0 to 3 i-th elements of all components of distributed variable a are assigned to the sum of i-th

elements of all components of distributed value b. In other words, on each virtual processor vector a is assigned to the sum of components of distributed vector b.

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 to the same virtual 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 will 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) will be distributed over *R*. The value of a scalar topological argument (if any) will 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*.

EXAMPLE 1 illustrates call to network function.

```
/*Line 1*/net SimpleNet (nnodes) sn;
/*Line 2*/repl int [sn]if_ok;
            …
/*Line 3*/ [sn]: {if_ok = ([(nnodes)sn])Watermp (nnodes);}
            …
```

EXAMPLE 2 illustrates call to basic function.

```
/*Line 1*/void [*]MakeProcGrid (repl int *nnodes, repl int **powe)
/*Line 2*/{
            …
/*Line 3*/}

/*Line 4*/int [*]main ()
/*Line 5*/{
/*Line 6*/  repl int nnodes, *powe;

/*Line 7*/  MakeProcGrid (&nnodes, &powe);
            …
/*Line 8*/}
```

Statements

Statements are the program elements that control how and in what order objects are manipulated.

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 *S1* begins. 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).

The following categories of statements are covered in this chapter:

- Labeled Statements
- Compound statements. These statements are groups of statements enclosed in curly braces ({ }). They can be used wherever the grammar calls for a single statement.
- Expression statements. These statements evaluate an expression for its side effects or for its return value.
- Selection statements. These statements perform a test; they then execute one section of code if the test evaluates to true (nonzero). They may execute another section of code if the test evaluates to false.
- Iteration statements. These statements provide for repeated execution of a block of code until a specified termination criterion is met.
- Jump statements. These statements either transfer control immediately to another location in the function or return control from the function.

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 [*].

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 (except for the cases considered below) and labeling the substatements by the distribution specifier. Obtained expressions shall not violate semantic constraints for cutting operator. The result of recursive application of the described procedure should be a correct mpC statement not having to be asynchronous.

If a substatemnet of the labeled statement is, in its turn, labeled by a distribution specifier, then the above procedure does not apply to the substatement.

The cutting operator determined by the distribution specifier does not apply to an identifier appearing in an expression if:

- the identifier is distributed over a subregion of the region determined by the distribution specifier;
- the identifier appears in the right operand of a coordof operator;
- the identifier is the right operand of a . or -> operator.

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.

A null statement labeled by a distribution specifier is distributed over the corresponding region of the computing space.

Compound statement (block)

Statements that are grouped into a block can 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.

Expression statement

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

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 asynchronous statements distributed over the same region. 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.

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 asynchronous in relation to this region, then the controlling expression shall be replicated. Otherwise, the behavior is undefined.

Iteration statements

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the *break* statement or the *continue* statement is encountered. (For a description of these statements, see The break Statement and The continue Statement.)

Syntax

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

```
for ( <expression>(opt);
      <expression>(opt);
      <expression>(opt) ) <statement>
```

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 an asynchronous statement, distributed over the same region. 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.

Finally, if the value of the controlling expression and the loop body are distributed over the same region, but the loop body is asynchronous in relation to this region, then the controlling expression shall be replicated. Otherwise, the behavior is undefined.

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.

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, then it is distributed over the region defined 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 an overall statement (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.

Syntax

```
<jump_statement>:
     break;
     continue;
     return <expression>(opt) ;
     goto <identifier>;
```

The goto statement

Constraints

An undistributed goto statement and the label used in it can 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.

The continue statement

Constraints

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

A distributed continue statement can 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.

The break statement

Constraints

An undistributed break statement must 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 must 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.

The return statement

Constraints

A return statement cannot be labeled explicitly by a distribution specifier.

A return statement cannot 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 cannot 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.

Library functions

The mpC library functions are developed to make mpC programming and debugging easier and more convenient. There are three categories of library functions:

- Nodal library functions
- Basic library functions
- Network library functions

All the library functions are declared in the header file <mpc.h>.

Nodal library functions

Nodal function is a function that can be called to by any virtual processor and is executed by the calling virtual processor only. Call to a *nodal function* is asynchronous. All pure C functions are nodal from the point of view of mpC.

In this chapter there described the following nodal library functions:

- The MPC_Printf function
- The MPC_Wtime function
- The MPC_Total_nodes function
- The MPC_Processors_static_info function
- The MPC_Abort function
- The MPC_Get_processor_name function

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.

Function MPC_Wtime

Synopsis

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

Description

MPC_Wtime returns a 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 during the life of the process, but can be different on different virtual processors of the computing space. The user is responsible for converting large numbers of seconds to other units if they are preferred.

This function is portable (it returns seconds, not "ticks"), it allows high-resolution, and carries no unnecessary baggage.

EXAMPLE:

```
{
    int starttime, endtime;
    starttime = int MPC_Wtime();
    ...stuff to be timed ...
    endtime = int MPC_Wtime();
    printf("That took %f seconds\n",endtime-starttime);
}
```

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.

Function MPC_Get_processor_name

Synopsis

```
#include <mpc.h>
char * MPC_Get_processor_name(void);
```

Description

MPC_Get_processor_name returns the hostname of the physical processor, to which the calling virtual processor is mapped.

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.

Function MPC_Abort

Synopsis

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

Description

MPC_Abort tries to abort all processes in the computer space. The value of error code will be returned to a command shell.

Return value

Ignored

Basic function is a function that is called to by all the virtual processors in the computing space. 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 return value (if any) shall be distributed over the entire computing space.

In this chapter there described the following basic library functions:

- The MPC_Exit function
- The MPC_Global_barrier function

Function MPC_Exit

Synopsis

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

Description

MPC_Exit terminates execution of an 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 exit code will be returned into the command shell.

Return value

Ignored

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.

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.

In this chapter there described the following network library functions:

- The MPC_Barrier function
- The MPC_Assign function
- The MPC_Bcast function
- The MPC_Scatter function
- The MPC_Gather function

Nettype SimpleNet

Synopsis

```
#include <mpc.h>
nettype SimpleNet(n) { coord I=n; };
```

Description

One-dimensional network type.

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.

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 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 receiver and specify initial the address of the receive buffer and the step between elements in the buffer, respectively. For every element to send 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 the 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.

Function MPC_Bcast

Synopsis

```
#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 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 the 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.

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,
    repl 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 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 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 this situation as erroneous.

The value of the 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.

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,
    repl const count,
    <s_type> *s_buffer);
```

Description

MPC_Gather gathers on a virtual processor of 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 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 <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 the 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.

Implementation restrictions

The current implementation does not support full implementation of parallel statements fan, par, and pipe, so their descriptions were omitted. In addition, following features are not supported in 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 a 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;

C[]

Introduction

The C[] (pronounced "see brackets") programming language is a Fortran90-like C extension. While preserving all ANSI C syntax and semantics, new powerful facilities for array processing are introduced.

The C[] programming language is aimed at producing portable, tunable and efficient code for a variety of modern platforms. In particular, systems with multilevel memory hierarchy and instruction level parallelism are supported.

Support of array-based computations is provided. The language permits to manipulate arrays as single objects. The C[] syntax offers natural form to express array-based computations which also allows compiler to fully utilize the performance potential of a target platform.

The key C[] features are:

- Access to an array as a whole as well as access to both regular and irregular segments of an array
- Variable-size (dynamic) arrays
- Variety of elementwise and reduction operators

C[] is a subset of the mpC programming language. While C[] addresses instruction level parallelism and memory hierarchy of a single-chip platform, mpC is aimed at exploiting parallelism of distributed memory architectures. Thus, mpC provides a way for comprehensive utilization of the performance potential of a target platform (for example, a network of UNIX workstations) at all levels.

Types

Vectors

The basic new notion of the C[] language is a notion of *vector*. A *vector* is defined as an ordered sequence of elements of the same *valid vector element type*. The number of vector elements is called *vector size*. A valid vector element type is any type excepting function type, void type, or any incomplete C type (recall that an incomplete type is an array of unknown size or structure or union of unknown content).

The C[] language introduces a new kind of derived types - *vector type*. A vector type describes a set of objects or values with a particular member object type, called the element type. The element type must be a valid vector element type. A vector type is characterized by its element type *T* and by the number *N* of elements in the vector. A vector type is said to be derived from its element type, and if its element type is *T*, the vector type is called *vector of N elements of type T* or simply *vector of T*. Unlike any other type, a vector type can not be explicitly specified and hence can not appear in declarations. But C[] expressions may have a vector type.

The simplest way to construct an expression of a vector type is to apply a special *blocking* postfix operator *[]* to an expression of an array type. If the expression *e* designates an array of *N* elements of a non-array valid vector element type, then the expression *e[]* designates a vector of *N* elements and the i-th element of that vector is just the i-th element of the array *e*, namely, *e[i]*. If the expression e designates an array of *N* elements of an array type, then the expression *e[]* designates a vector of *N* elements and the i-th element of that vector is the result of applying the blocking operator to the array designated by *e[i]*.

In this document we often use the term "vector a[]" instead of "vector designated by a[]".

EXAMPLE:

Let the array *a* be defined and initialized by the declaration

```
int a[3][2];
```

Then the expression *a[]* has the type "vector consisting of three vectors, each of which consists of two integers" with elements *{{a[0][0], a[0][1]}, {a[1][0], a[1][1]}, {a[2][0], a[2][1]}}* .

Vectors and arrays are similar in many features, but there is one principal difference, namely, in expressions, arrays are converted to pointers meanwhile vectors do not. For example, if array **a** is declared as *int a[8]*, then the expression *a+1* has type "pointer to int" and points to the first element of the array *a*. At the same time, the expression *a[]+1* has type "vector of 8 ints", and the **i**-th element of that vector is equal to *a[i]+1*. See the section Vector Operators for a comprehensive explanation of vector operators.

The blocking operator *[]* is also applicable to pointers. If *e* is an expression of type "pointer to a non-array valid vector element type", then the expression *e[]* designates a vector of *N* elements (*N* is determined from the context according to rules

explained later in the document), and the i-th element of that vector is *e[i]*. If *e* is a pointer to array type *T*, and *T* is also a valid vector element type, then the expression *e[]* designates a vector of N elements (*N* is determined from the context), and the i-th element of that vector is the result of applying the blocking operator to the array designated by *e[i]*.

The blocking operator *[]* is applied to an expression designating an array of unspecified length in the same manner as it is applied to a pointer.

Here are more intricate examples of use of the blocking operator.

EXAMPLE:

```
int a[10];
int b[10];
int *p;
a[]=b[]+p[];
```

Here the expression *p[]* is a vector of *10* elements. The vector length is determined from the context.

EXAMPLE:

Let pointers *p1* and *p2* be declared as follows:

```
int (*p1)[10];
int **p2;
```

Then the expression *p1[]* has the type "vector of *N* of vectors of 10 elements of type *int*", where *N* depends on the context. The expression *p2[]* has the type "vector of *M* pointers to *int*", where *M* depends on the context.

Arrays

In the C language an array comprises "a contiguously allocated set of elements of any one type of object". In the C[] language an array comprises a sequentially allocated elements (with a positive 'step') of any one type of object. Thus, in the C[] language an array has at least three attributes, namely, the type of its elements, the number of elements and the allocation step. In the C[] language, the array declarator syntax differs from the standard in following way. The rule

```
<direct-declarator>:
<direct-declarator> [ <expression>(opt)]
```

is replaced with the rules

```
<direct-declarator>:
    <direct-declarator> [ <expression>(opt) <step>(opt)]
<step>: ':' <expression>
```

If step is not specified, then it is equal to 1. The step should be a positive integral value.

Unlike the C language, C[] allows arrays to have *variable length*, i.e. an array length may be any expression of an integral type. Similarly, a variable array step is also permitted. Arrays with non-constant steps or lengths must have an automatic storage

durations, i.e. must be declared either with *auto* storage class specifier or must be declared within function body without storage class specifiers.

Constraints

Objects of array type with step exceeding 1 shall not be initialized. Objects of array type with non-constant length shall not be initialized.

EXAMPLES**:**

The declarations *int a[3:1]* and *int a[3]* both define an array of the form

The size of the slot between elements of the array is equal to zero.

The declaration *int a[3:3];* defines an array of the form

The size of the slot between array elements is equal to *2*sizeof(int)* bytes.

In the following example the array *a* of *N+M* elements is declared:

```
int N,M;
...
f(){
  int A[N+M];
  ...
}
```

Pointers

In the C language a pointer has only one attribute, namely, the type of object it points to. This attribute is necessary for the correct interpretation of values of objects it points to as well as the address operators + and - . These operators are correct only if the pointer's operands and the pointer's results point to elements of the same array object.

The same rule is valid for the C[] language. Therefore, to support the correct interpretation of the address operators, one additional attribute of pointer is introduced, namely, *step*.

In the C language Standard, "when an expression that has integral type is added to or subtracted from a pointer, the integral value is first multiplied by the size of the object pointed to". In the C[] language , the multiplier is equal to the product of the pointer step and the size of the object pointed to. In the C language, "when two pointers to elements of the same array object are subtracted, the difference is divided by the size of a element". In the C[] language, the divisor is equal to the product of the pointer step and the size of an element.

In the C[] language, the pointer declarator is defined as follows:

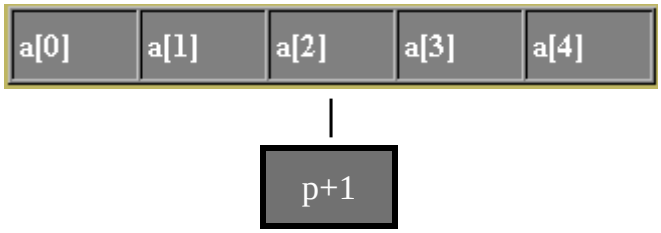
```
<pointer>:
    * <step>(opt) <type-specifier-list>(opt)
    * <step>(opt) <type-specifier-list>(opt) <pointer>
```

If step is not specified, it is equal to 1. The step may be any expression of an integral type. Pointers with non-constant steps must have an automatic storage durations, i.e. must have either *auto* or *register* storage class specifiers or must be declared within function body without storage class specifiers.

EXAMPLE:

The declaration *int a[]={0,1,2,3,4}* defines an array of the form

The pointer declaration *int *:2 p=(void*)a* forms the following structure of storage



and address expressions *(p+1)* points to the *a[2]* element of the array *a*.

Dynamic types

The C[] language allows to use non-constant expressions as specifier of array dimension, array step or pointer. Such arrays and pointers are called *dynamic*. Only arrays with automatic storage class can be dynamic. By analogy with the draft standard of the C language performing of the side effect in the expression for the array dimension is not secure. The same rules are used in the C[] also for pointers.

EXAMPLE

Access to the diagonal elements with the help of pointer to array with step.

```
typedef (* tDiag)[N+1];

int A[N][N];
tDiag p;

...
p=(tDiag)A;
...
(*p)[i]=1;
```

Arrays and pointers with step are convenient tools for access to different sets of elements of the array. Thus, in this example the pointer *p* to the array allows to access diagonal elements of the matrix stored in the array *A*.

The type *tDiag* is a type of pointer to array with step N+1. As arrays in the C language are stored in the memory by rows, the diagonal elements are stored with step N+1, and the expression *(*p)[i]* designates the i-th element of the diagonal.

Expressions

Lvectors

A vector comprising modifiable lvalues is called lvector. In C[], the left operand of a simple or compound assignment operator or the operand of postfix/prefix increment/decrement operators must be either modifiable lvalue or lvector.

EXAMPLE:

Expressions *a[]* and *b[]* are lvectors, but the expression *a[]+b[]* is not.

Note, that the result of applying the blocking operator is always an lvector. Lvectors may also be the result of applying operators other then the blocking one. For example, if array *p* is declared as *int* p[3]*, then the expression **(p[])* is an lvector, and, hence, the expression **(p[])=1* is correct.

Access to the Elements of an Array

In the C[] language, *e2*-th element of an array object *e1* is accessed with the help of one of the expressions *e1[e2]* and *(e2)[e1]* . Both are identical to *(*(e1+(e2)))* . Here, *e2* is an integral expression, *e1* is an lvalue that has the type "array of type". This lvalue is converted to an expression that has the type "pointer to type" and that points to the initial element of the array object (the attribute step of this pointer is identical to the attribute step of the array object).

EXAMPLE 1

In this example to all the elements of the i-th row of the array A there is assigned the value of the i-th element of the array b:

```
int A[M][N];
int b[M];
...
...
A[:][:]=b[:];
```

EXAMPLE 2

In this example to the elements of the array A with the indices 0, 1, 3 there is assigned the value 1:

```
int A[M];
int ind[3]={0,1,3};
...
...
A[ind[:]]=1;
```

Access to Subarrays

In this section it is described, how the blocking operator can be used to access subarrays.

By definition, a (data) object *belongs* to an array, if it is an element of the array or it belongs to an element of the array. Any set of objects belonging to the same array is called a subarray, iff this set can be described as an array (using bounds and step

attributes as defined above). In addition, any subarray can be referred to as an object belonging to its superarray.

In principle, the facilities introduced are sufficient to access subarrays. For example, if the array object **a** is defined by the declaration

```
int a[5][5];
```

then the expression

```
(*(int(*)[5:6])a)[]    (1)
```

designates a vector of five *ints* that contains the main diagonal of the matrix **a** , and the expressions

```
(*(int(*)[4:6])(a[0]+1))[]   (2)
```

and

```
(*(int(*)[4:6])(&a[0][1]))[] (3)
```

designate a vector of four *ints* that contains the diagonal of the matrix **a** which is placed above the main diagonal.

The more compact notation results, if variables of type "pointer to array" are used. So, if the pointer objects *p1* and *p2* are defined by declarations

```
int (*p1)[5:6]=(void*)a;
int (*p2)[4:6]=(void*)(a[0]+1);
```

then the expression *(*p1)[]* can be used instead of (1) and the expression *(*p2)[]* can be used instead of (2) and (3).

Array Segments

Not every regular set of objects belonging to an array is a subarray. For example, the rectangular segment of the array *a* represented in Fig.1 is not a subarray.

Figure 1: Rectangular 2x3 segment of array *A*

Access to such array segments is provided by so-called grid operator *[:]*, the only quaternary operator in C[]. The general notation for the grid operator is:

```
e [ l : r : s ],
```

where expression *e* either is of a pointer type or designates an array, and expressions *l, r, s* are of any integral type. *l, r , s* denotes the left bound, the right bound and the step correspondingly, and *e [r : l : s]* designates a vector of *(r-l)/s + 1* elements whose *i*-th element is *e[l+i*s]* .

EXAMPLE:

If array *a* is declared as *int a[5]*, then the expression *a[2:4:2]* designates a two-element vector comprising *a[2]* and *a[4]* .

The step operand may be omitted, and in that case the second semicolon in the grid notation is optional. One or both bounds are also may be omitted. The omitted left bound is replaced by 0, while the omitted right bound is replaced by *N-1* , where *N* is the number of array elements, if the fist operand designates an array, or determined from the context, if the operand is a pointer. Fig. 2 gives some examples of grid expressions with various combinations of omitted operands.

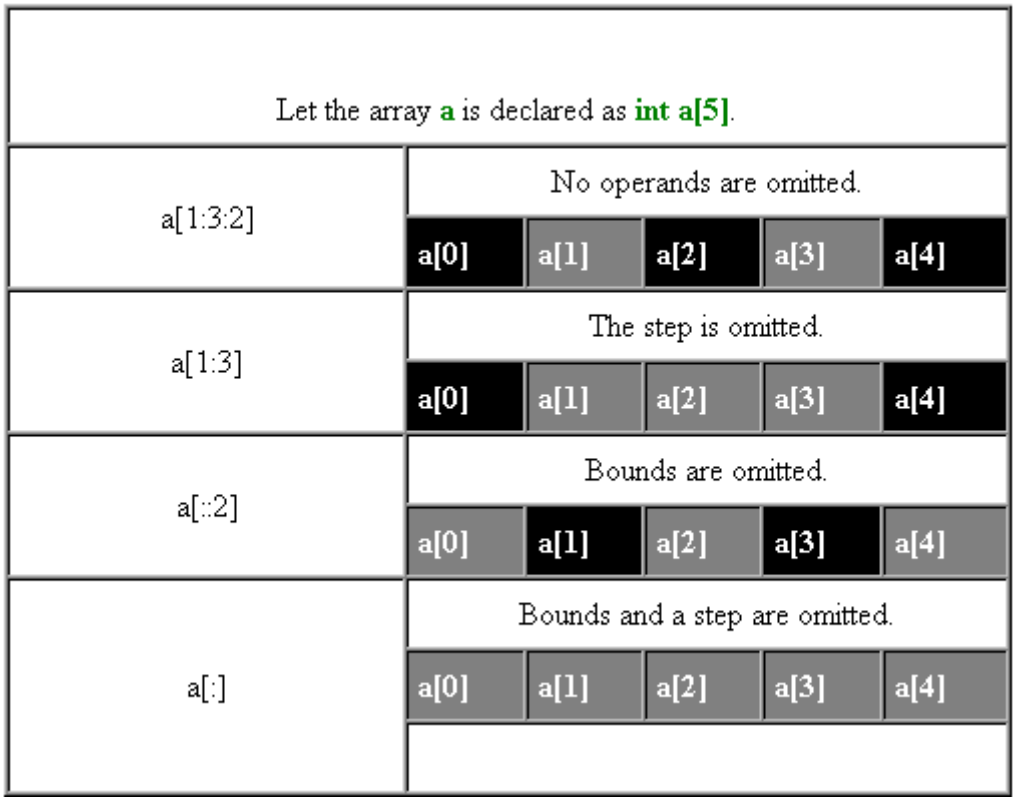


Figure 2. Various combinations of omitted values in grid expressions.

The first operand of the grid operator may have a vector type. In that case the operator is applied elementally. Consider the array *A* which is declared as *int A[4][5]*. The expression *A[1:2]* is a vector of 2 arrays corresponding to second and third rows of *A*. In the expression *A[1:2][1:3]*, the second grid operator (*[1:3]*) is applied to each of the arrays selecting their second, third and fourth elements (Fig. 1). Successive grid operators are very convenient to access segments of a multi-dimensional array.

The operand of the blocking operator *[]* also may have vector type. In that case, the operator is applied elementally. If the array *A* is declared as *int A[5][5][5]*, then the expression *A[1:3]* has type "vector of 3 arrays". In the expression *A[1:3][]*, the blocking operator is applied to each of the arrays. Thus, the expression *A[1:3][]* designates the 3x5x5 array segment.

One can see that the expression *A[1:3][]* has an equivalent representation, *A[1:3][:][:]*. Thus, the blocking operator can be considered as more compact notation to express successive grid operators with omitted steps and bounds.

Irregular Segments

As we have mentioned, operands of the subscript operator may have vector types. Any subset of array elements can be accessed by means of some subscript expression, whose right operand is of vector type (so-called *vector subscripting*). In other words, access to the vector consisting of *i1*-th, *i2*-th, ..., *in*-th elements of array *a* is provided by the expression *a[i[]]*, where *i[]* is the *n*-element vector whose elements are *i1*, *i2*, ..., *in*.

EXAMPLE:

If the array *a* is declared as *int a[5]* and the array *i* is declared as *int i[4]={0,1,3,4}*, the expression *a[i[]]* designates a vector of *a[0], a[1], a[3], a[4]*.

EXAMPLE:

In the following C[] code portion

```
int A[5][5];
int i[4]={0,1,3,4};

A[ i[] ][ i[] ]=1;
```

the value **1** is assigned to all elements of the irregular array *A* region depicted in Fig. 3.

Figure 3: Irregular segment of the array *A*

In fact, the grid operator provides more convenient but less flexible way to access subsets of array elements then vector subscripting. If access to regular segments is required, the grid operator is preferable. Of course any grid operator may be replaced by some equivalent subscript operator. Indeed, consider the array *a* declared as *int a[5]*. Expressions *a[1:3:2]* and *a[i[]]*, where vector *i[]* is a two-element integral vector whose elements are **1** and **3** correspondingly, designate the same vector. Similarly, any combination of successive grid operators can be expressed by means of appropriate vector subscripting.

Unary vector operators

The operand of unary **&**, *****, **+**, **-**, **~**, **?**, **%**, **!**, **++** (postfix and prefix form), and **--** (postfix and prefix form) operators and scalar cast operators may have a vector type. In that case, the operator is applied to the elements of vector; if they are also vectors, the operator is applied to their elements and so on.

Note that the operand of the decrement and increment operation (both postfix and prefix) must be Lvector.

The type name in cast operator can be any scalar type of the C[] language.

EXAMPLE:

```
int a[3];
int b[3];
a[]=-b[];
```

In this example, the i-th element of the vector *a[]* is set to *-b[i]* for *i=0, 1, 2*.

Binary vector operators

One or both operands of binary =, *****, /, **%**, **?<**, **?>**, +, -, <<, >>, <, >, <=, >=, ==, **!=**, **&**, ^, |, ***=**, **/=**, **%=**, **+=**, **-=**, **<<=**, **>>=**, **&=**, **^=**, and **|=** operators may have vector type. In this case both operands must be vectors of the same size, otherwise the behavior is undefined.

In the following C[] code portion

```
int a[10], b[10], c;
a[]=b[]*c;
```

*b[i]*c* is assigned to the i-th element of array *a* for all i (0 <= i <10).

If elements of the operands are vectors of the same size, the operator is applied to their elements and so on. If the elements of one operand are vectors of the size N, and elements of the second one are scalars then the scalars are converted to vectors of the size N, all elements of which are equal to the corresponding scalar value.

However, a binary operator is applicable to vector operands of different number of dimensions. In the following example

```
double a[10], B[10][20];
B[] *= a[];
```

each element of the i-th row of *B* is multiplied by *a[i]*.

Conditional operator may also have vectors as its operands. If the first operand of conditional operator is a scalar and the second or third operand or both are of vector type then the result of the operator has the same vector type as for binary operators discussed above. The first operand of a conditional operator may have vector type. In that case the second or the third operand but not both of them may be omitted. If none of the operands is omitted then unlike the C language all three operands are evaluated. If all three operands are vectors of the same length then the result is produced by elementwise application of the operator. If vector operands of a conditional operator have different lengths then behavior is undefined. If the second or the third operand is non-vector then the length of that operand is converted to the length of the vector operands. If the first and the second (or the third) operands are vectors, the third (the second) operand is omitted, and the elements of the first operand have scalar type, then the result will be the vector of the same type as the second (the third) operand; the i-th element of the result is equal to the k(i)-th element of the second (the third) operand where k(i) is the index of the i-th non-zero (zero) element of the first operand. The other elements have indefinite values.

For example, execution of

```
int a[5]={1,2,3,4,5};
int b[5]={3,3,3,2,6};
int c[5];

c[]=a[] < b[] ? A[]:;
```

results in the vector *c[]* equal to *{1,2,5,w,w}*, where *w* denotes an undefined value.

If the first and the second (or the third) operands are vectors, the third (the second) operand is omitted, and the elements of the first operand have vector type, then the result is achieved by elementwise application of the operator.

Subscript operator also allows vector operands. Remember that the C language expression *e[f]* is defined as **(e+f)*. In C[] the subscript operator is treated exactly in the same way.

EXAMPLE:

In the following C[] program the last column of the array *A* is set to *1*:

```
int A[2][3];
int* p[2];

p[0]=&A[0][0];
p[1]=&A[1][0];
p[][2]=1;
```

Indeed, the expression *p[][2]* has an equivalent form, **(p[]+2)*, that is clearly a vector consisting of pointers to the last matrix *A* column elements.

The blocking operator is applicable to vectors in elementwise fashion. See section Array Segments for details.

Access to structure and union members

The first operand of the "." operator may have vector type. In this case the second operand is identifier – union or structure member name. The result of the expression lexpr.rexpr is vector of the same size as lexpr. Its elements have values of the named structure member of the corresponding vector elements.

The expression lexpr->rexpr is equivalent to the expression (*lexpr).rexpr.

Determining Undefined Vector Size from Context

If one of the operands of a binary operator is a vector of undefined size, and another is a vector of a definite size, *N*, then the size of the operand of undefined size is assumed to be equal to *N*. Similarly, if one of the operands of a ternary operator is a vector of definite size *N*, then the size of a vector operand of undefined size is assumed to be equal to *N*.

EXAMPLE:

```
int a[3];
int b[3];
int *p

a[]=b[]+p[];
```

Here, the size of vector *p[]* is 3, because the size of the left operand of the binary + operator (*b[]*) in the expression *b[]+p[]* is 3.

EXAMPLE:

If the pointer *p* is declared as *int **p* and the array *A* is declared as *int A[]*, then the element size of the vector *p[][]* in the expression *a[]+p[][]* can not be determined from the context regardless the element size of array *a* is definite.

Reduction operations

The unary reduction **[*]**, **[?<]**, **[?>]**, **[+]**, **[&]**, **[^]**, and **[|]** operators correspond to binary *****, **?<**, **?>**, **+**, **&**, ^, and | operators. These operators are applicable only to vector operands. Let *v[0], v[1],...,v[N]* denote the elements of vector operand *v*. Then the expression *[op] v[]* has the same semantics as the expression of *(...((v[0] op v[1]) op v[2]) op ... op v[N])* kind.

EXAMPLE:

In the following code portion

```
int a[]={0,1,2,3,4}, sum;

sum=[+]a[];
```

the value of *sum* is equal to the value of the expression

```
((((a[0]+a[1])+a[2])+a[3])+a[4])
```

which is equal to 10.

EXAMPLE:

```
double A[2][3];
double s[3];
double sum;

s[]=[+]A[];
sum=[+][+]A[];
```

Here the sum of the rows of the array *A* is assigned to the vector *s[]*, and the sum of all elements of the array *A* is assigned to *sum*.

In the C[] language there are defined maximum *?>* and minimum *?<* binary operators. The corresponding *[?>]* and *[?<]* reduction operators are used to evaluate the maximum and minimum values among array elements.

EXAMPLE:

The following C[] code portion is aimed at evaluating the maximum among elements of matrix *A*:

```
int A[2][3];
int max;
max=[?>][?>]A[];
```

Index