TR-94014

The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1

Yutaka Ishikawa*
ishikawa@rwcp.or.jp

Received 29 June 1994

Tsukuba Research Center, Real World Computing Partnership
Tsukuba Mitsui Building 16F, 1-6-1 Takezono
Tsukuba-shi, Ibaraki 305, JAPAN

Abstract This document is the reference manual of the MPC++ programming language version 1.0 designed and developed at Tsukuba Research Center. In MPC++, C++ is extended for describing an operating system kernel and control/data parallel computation on multi-threaded parallel machine architectures.

*TRC Massively Parallel Software Laboratory

1 Introduction

This document describes the MPC++ programming language Version 1.0 with comments on the design. MPC++ introduces a computational model provided by a message driven multi-threaded architecture such as RWC-1[9]. MPC++ offers users three layers of hardware abstractions: i) the physical layer on which users may implement an operating system kernel, an efficient automatic parallelized compiler, or provide a new data/control parallel mechanism, ii) the virtual layer on which users will write data/control parallel programs, and iii) the C and C++ layer on which users do not consider parallelism.

The MPC++ programming language Version 1.0 specifies the first two levels. On the physical layer, MPC++ provides i) a variable allocation mechanism over the physical processors, ii) a *function instance* which is the abstraction of a thread of control by message passing, iii) a *message entry and token* which are the abstraction of communication among threads, iv) some parallel constructs, and v) rich macro processing. Because machine-dependent programs will appear in this layer, we recommend that end users do not use this layer but instead use the virtual layer.

The key features related with the virtual layer are to provide i) the notion of *topology* and *processors group* which gives us the abstraction of a logical memory space and processors, ii) data parallel operations over a topology, and iii) function instance invocation on a virtual processor.

The new keywords introduced in MPC++ are listed as follows:

physical **virtual** **layer**
entry **token** **topology** **remote**

In this document, we assume that readers are familiar with C++ version 2.0 which is the draft of ANSI C++. If they are not familiar with the C++, we recommend that they read [3, 11] first.

1.1 The Notation

The notation of the syntax follows the C++ syntax described in [3]. That is, an *italic* letter represents a nonterminal symbol while a `typewriter type` letter represents a terminal symbol. For examples, *mpc-parallel-access-specifier* and *access-specifier* are nonterminal symbols while `parallel`is a terminal symbol in the following example:

mpc-parallel-access-specifier:

`parallel` *access-specifier*

Alternatives are listed on separate lines. An optional terminal or nonterminal symbol is indicated by the subscript "opt". For example, the following member-specification$_{\text{opt}}$ is an optional nonterminal symbol:

member-declaration member-specification$_{opt}$

1.2 Organization

In each section, after the features of MPC++ with the BNF syntax are described, then comments on the design and implementation are stated if needed. Complete MPC++ syntax is given in appendix A. In section 2, two layers, virtual and physical processor layers, are introduced. Variables on the virtual processor layer are described in section 3. In section 4, after the notion of processors group is introduced, dynamic data distribution mechanisms are explained. Function instance, which is the abstraction of the thread of control, and message entry and token, which are the abstraction of communication, are introduced in sections 5 and 6. In section 7, the parallel class is introduced. In section 8, parallel statements are described. In section 9, after introducing the new notation for writing a program on the physical processor layer, the implementation of the virtual processor layer is given.

In section 10, the rich macro statements introduced in MPC++ are shown. Section 11 describes the linkage specification. The real-time extension is described in section 12. The MPC++ realtime extension is based on RTC++[7]. This extension has not been fixed in MPC++ version 1.0. Section 13 describes support for programming the operating system. In section 14, class programming examples, `Istructure`, `Qstructure`, and `ByteStream` are shown.

2 Abstraction Layers

The MPC++ programming language provides three programming abstraction layers, i.e., physical, virtual, and C and C++ layers. The layer declaration must appear in the file scope. The syntax of the layer declaration is specified as follows:

mpc-layer-declaration:
`physical layer`
`virtual processors (` *constant-expression* `) layer`

When no layer declaration is specified, the C and C++ layer is assumed by the compiler. The rest of this section will describe an overview of those abstraction layers.

2.1 Physical Processor Layer

The MPC++ programming model assumes a distributed memory based parallel machine. Users may access the local memory on any remote physical processor via a communication primitive on the physical processor layer. Following the SPMD (Single Program Multiple Data) model, program code is distributed to all physical processors and run on them.

2.1.1 Variables

All variables are processor local on the physical processor layer. The storage defined in the file scope is allocated on each processor. That is, when such a variable is referred to on a processor, the local memory space is accessed. The local address of the variable is the same address on all processors.

The key mechanism provided in this layer is an encapsulation mechanism of a physical variable to represent scalar and parallel variables on the virtual processor layer. By a parallel variable, we mean that the variable is a set of storage locations allocated on processors. Operations over a parallel variable may be executed in the data parallel way. The user may program the control of distribution and communication on scalar and parallel variables.

2.1.2 Parallel Flow Control and Communication

To describe control parallel computation in the physical and virtual processor layers, the notion of function instance is introduced. A program may locally or remotely invoke a function instance which has a thread of control. The sender program may block until the end of the invoked function instance execution or may execute the subsequent statements and synchronize with the receiver later.

The communication channels are specified by message entries each of which is an entry point where a message is received. To send a message to an entry point, the sender must have the token of that entry point. The token is capable of sending a message to the entry point specified by that token. The token is obtained by referring to the message entry variable. This mechanism is also open to use in the virtual processor layer.

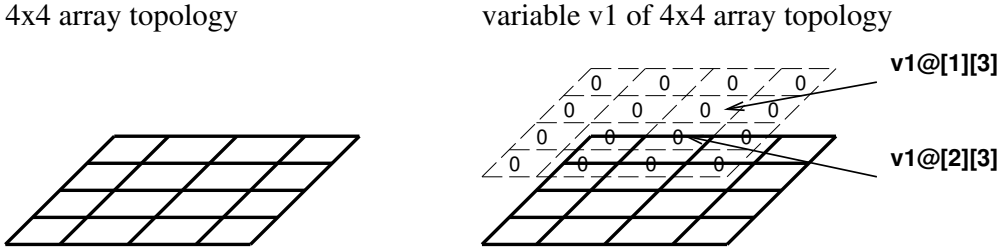


Figure 1: Example of Array Topology and Parallel Variable

2.2 Virtual Processor Layer

The virtual processor layer provides the notion of topology which is the abstraction of a logical memory space on virtual processors. Operations over variables declared on a topology may be handled in the data parallel way. C++ class is extended so that parallel functions are defined on a topology.

2.2.1 Variables

An ordinal C++ file scope variable is a virtual processor local. That is, such a variable is allocated in each virtual processor. A topology is the abstraction of a logical virtual memory space over processors. In MPC++ Version 1.0, topologies, `scalar`, `array`, and `binary tree` are predefined. A variable of `scalar` is called a scalar variable. A scalar variable is allocated as a global variable of virtual processors. That is, the storage denoted by a scalar variable name is allocated on some processor. When a scalar variable is referred to on a processor where that storage does not exist, remote memory access will be issued.

Programmers may specify the dimension of an array topology as shown in Figure 1. A variable may be allocated on such a topology. An element of the variable of an array topology is referred to by indexing. Such a variable is called a parallel variable in the sense that operations over the topology include data parallel operations. The storage of such a variable may be distributed to physical processors. The distribution strategy is defined in the physical processor layer.

Figure 2 illustrates how variables are allocated on virtual processors and mapped to physical processors.

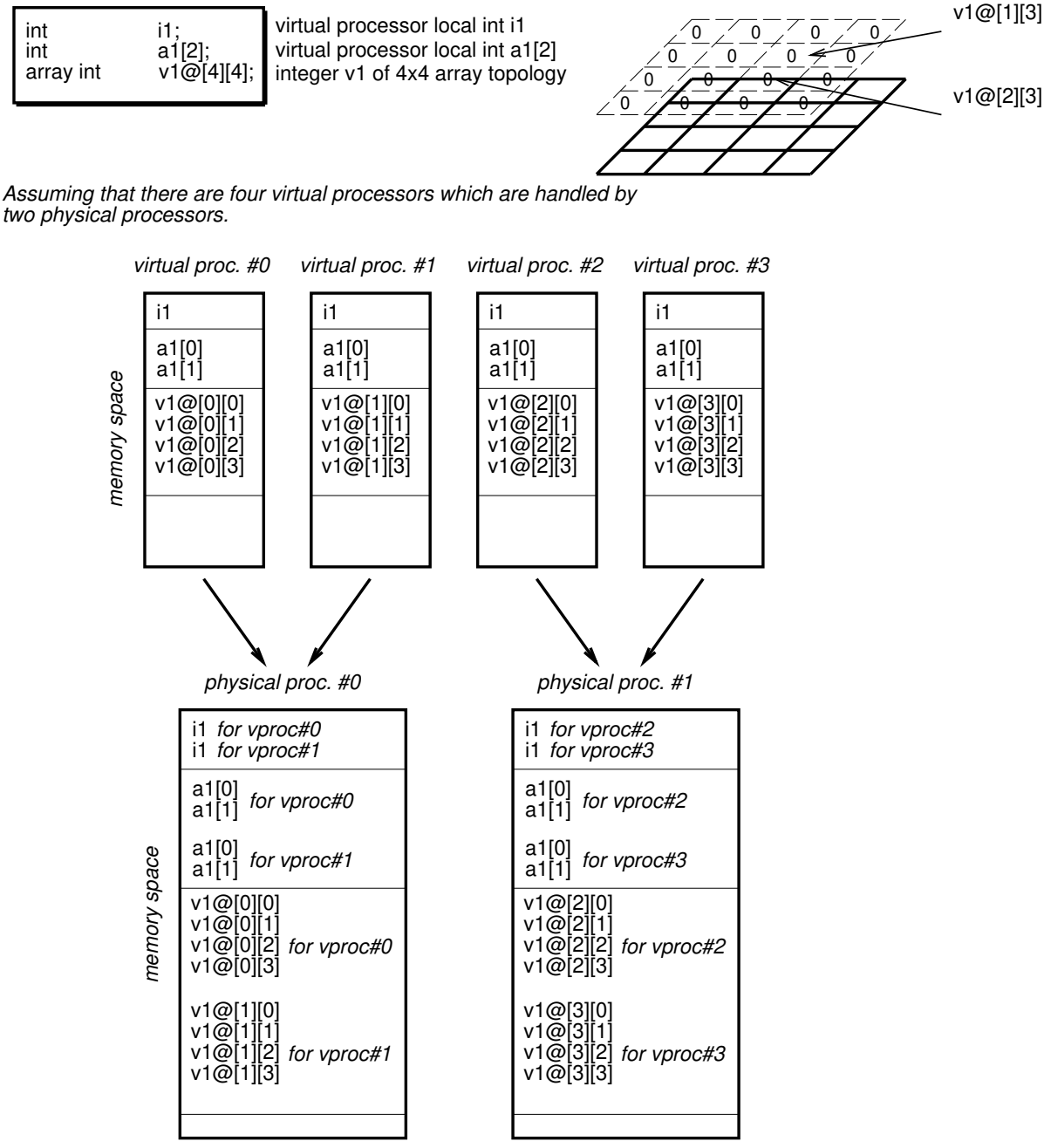


Figure 2: Variables on Virtual Processors on Physical Processors

2.2.2 Parallel Flow Control and Communication

Data parallel computation is realized by a special statement and operations over variables of topology except for `scalar`. Users may program data parallel computation using a special statement `with` for a topology. Users may also specify parallel member functions of a class whose instance must be on a topology. Such a class is called a *parallel class*.

The function instance and communication primitives introduced in the physical layer are also applicable in the virtual processor layer. In the invocation of a remote function instance, the remote PE number is a virtual number on the virtual processors.

2.3 C and C++ layer

The sequential programs written in C and C++ must be compiled to parallel programs in this layer. That is, the automatic parallelized compiler is assumed. To assist the compiler, some compiler directives like in HPF[4] will be introduced. The specification of this layer is out of the scope of this document.

2.4 Commentary

The reasons why we provide three layers are the following:

1. We assume that MPC++ will be used by a very wide range of users. Some users do not like to consider parallelization while others want to program the low level control such as the data distribution and communication timing.

2. The MPC++ language system is an experimental product, and research is ongoing. Instead of designing a language which requires a mature parallelized compiler, we would like to provide some mechanisms to program parallelization by users.

3. The MPC++ language system is firstly used to implement a message passing operating system kernel called SCore. Thus, we need to provide low level mechanisms.

3 Variables

In this section and the following sections through section 8, only the virtual processor layer is considered. Please see section 9 for the description of the physical processor layer.

3.1 File Scope Variable

A file scope variable is a virtual processor local variable in the virtual processor layer. That is, the storage of such a variable is allocated on each virtual processor. A function may access the local storage of the variable where the function runs.

```
1 virtual processors(100) layer;
2 int     a1;
3 double  d1;
```

By default, the MPC++ compiler may generate efficient code to implement a virtual processor local variable for the target machine. The specification of MPC++ reserves room for users to program their own implementation of such a variable. This mechanism is described in section 9.

3.2 Scalar and Parallel Variables

To access a unique variable over processors, a scalar variable is introduced. A scalar variable may be accessed by all processors. A scalar variable is defined with the keyword `scalar` which is used as the type specifier and pointer qualifier, as in `const` and `volatile`.

In MPC++ Version 1.0, a variable may be defined on some topology such as `array` or `btree` which is the binary tree. Such a variable is called a *parallel variable*. Values of a parallel variable are called parallel values. The type of a parallel variable is a type of those values. Such a type must be the ordinal C++ type. The syntax below defines the scalar and parallel variables declaration. The *decl-specifiers* specifies the type of variables specified by the *declarator-list*. The *decl-specifiers* is the same as in C++. The `global` type specifier is reserved for EMC.

mpc-variable-on-topology:
    mpc-type-specifier decl-specifier-seq declarator-list ;
mpc-type-specifier:
    `scalar`
    `global`
    `array`
    `btree`

The *declarator-list* is a list of the *init-declarator* that consists of the *declarator* and optionally the *initializer*. The *declarator* is extended as following:

declarator:
    direct-declarator
    ptr-operator declarator

direct-declarator:
declarator-id
declarator-id `@` *mpc-topology-dimen* $mpc\text{-}distribution\text{-}hint_{opt}$ |
direct-declarator `(` *parameter-declaration-clause* `)` $cv\text{-}qualifier\text{-}seq_{opt}$ $exception\text{-}specification_{opt}$
direct-declarator `[]` $constant\text{-}expression_{opt}$ `]`
`(` *declarator* `)`
mpc-topology-dimen: |
mpc-topology-dimen `[` *constant-expression* `]` |
ptr-operator:
`*` $mpc\text{-}pointer\text{-}qualifier_{opt}$ $cv\text{-}qualifier\text{-}seq_{opt}$ |
`&` $mpc\text{-}pointer\text{-}qualifier_{opt}$ $cv\text{-}qualifier\text{-}seq_{opt}$ |
qualified-class-specifier `::` `*` $mpc\text{-}pointer\text{-}qualifier_{opt}$ $cv\text{-}qualifier\text{-}seq_{opt}$ |
mpc-pointer-qualifier: |
`scalar` |
`global` |
`array` |
`btree` |

The *mpc-distribution-hint* specifies the initial data distribution strategy which is a hint for the compiler.

mpc-distribution-hint: |
`(` $mpc\text{-}proc\text{-}group_{opt}$ *mpc-distribution-hint-elem* `)` |
mpc-proc-group: |
identifier `,` |
mpc-distribution-hint-elem:
mpc-distribution-hint-macro |
mpc-distribution-hint-elem `,` *mpc-distribution-hint-macro* |

Examples of scalar variables are shown below:

```
1 scalar int              sa;
2 scalar int              *psa1;
3 int     * scalar        psa2;
```

The `sa` variable is a scalar integer variable. Variable `psa1` is a pointer variable which points to a scalar integer storage. Variable `psa2` is a scalar pointer variable which points to a local integer variable.

The following examples and Figure 3 show the usage of the array topology.

```
1 virtual processors(4) layer;
2 array int       a0@[10](BLOCK);
3 array int       a1@[10][10](BLOCK(4), ALL);
4 array int       a2@[10][10](ALL, BLOCK);
5 array int       a3@[10][10](CYCLE(2), ALL);
6 array int       a4@[10][10](ALL, CYCLE(1));
```

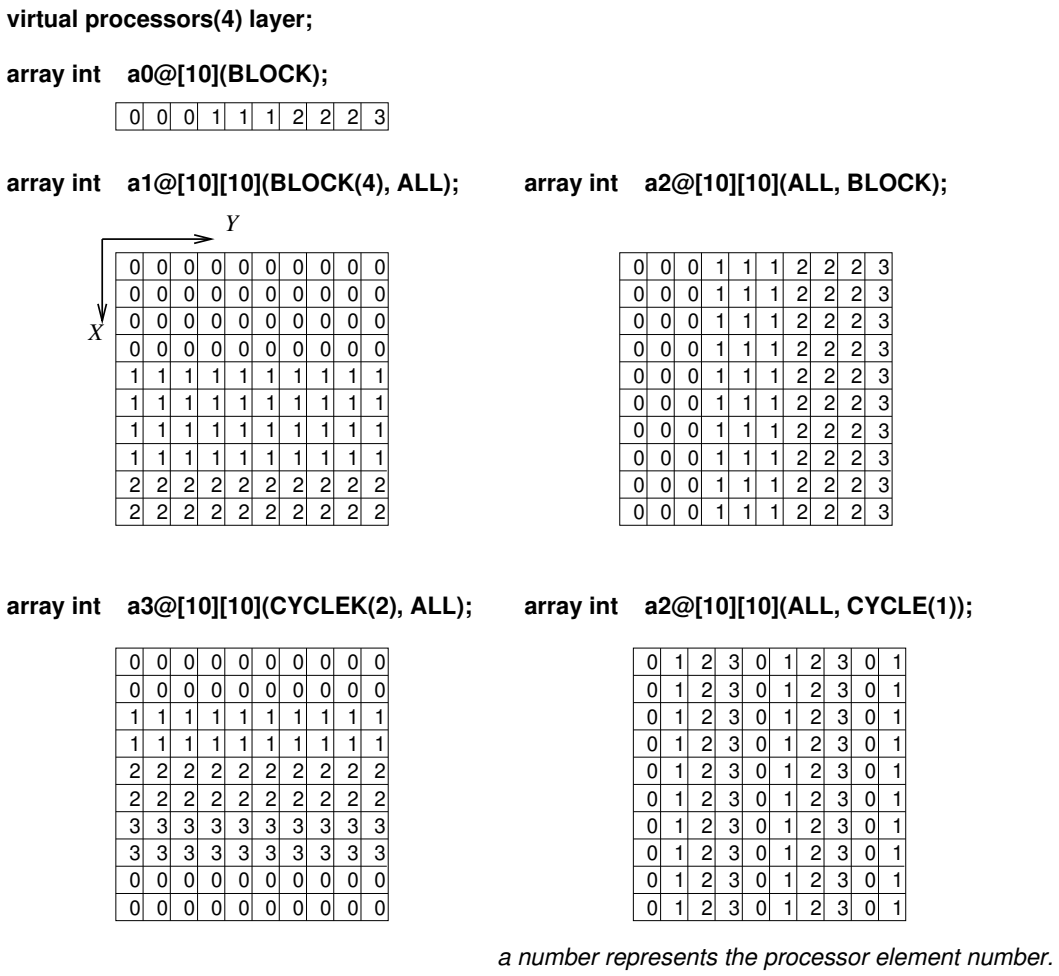


Figure 3: Parallel Variables Distribution

The a0 is an integer variable on the one dimensional array topology whose dimension is 10. The manner of distributing the a0 elements to virtual processors is specified by "(BLOCK)" which means that one block will contain three elements and blocks are distributed to virtual processors. That is, the elements of variable a0 are divided into four elements each and those are distributed to four virtual processors.

The a1 is an integer variable on the two dimensional array topology whose dimension is 10 x 10. The manner of distributing the a1 elements to virtual processors is specified by "(BLOCK(4), ALL)" which means that one block will contain four elements of X axis with all related Y axis's elements. The storage is distributed to four virtual processors in the BLOCK distribution manner.

Variables a2, a3, and a4 are integer variables on the two dimensional array topology whose

dimension is 10 x 10. Their distribution manners is different as shown in Figure 3. The distribution manner of the a2, specified by "(ALL, BLOCK)", means that one block will contain three elements of the Y axis with all related X axis's elements because the number of virtual processors is four.

The a3 distribution manner, specified by "(CYCLE(2), ALL)", means that one block will contain two elements of the X axis with all related Y axis's elements. Such blocks are cyclically allocated in the four virtual processors. The a4 distribution manner, specified by "(ALL, CYCLE(1))", means that one block will contain one element of the Y axis with all related X axis's elements. Such blocks are cyclically allocated in the four virtual processors.

You may specify an array of any type on a topology. The following example declares an integer one dimension array variable s2[4] on two dimension array topology 10 x 10:

```
1 virtual processors[100] layer;
2 array int       s2[4]@[10][10](CYCLE(1), CYCLE(1));
```

It should be noted that the MPC++ feature allows the users to specify their own new topology as well as define a new distribution manner. This feature is described in section 9.

3.3 Accessing Variables on Topology

Users may access a variable on a dimensional topology by describing the variable name followed by @ and index expressions. The syntax is shown below:

mpc-topology-expression:
    postfix-expression @ *mpc-topology-index*
mpc-topology-index:
    mpc-topology-index [*expression*]

An example is shown below.

```
1 array int       a0@[100];
2 array int       a1@[10][10];
3 array int       a2[10]@[10][10];
4 sub()
5 {
6     a0@[10] = 2;
7     a1@[4][2] = 3;
8     a2[2]@[3][2] = 4;
9 }
```

3.4 Pointer and Topology

Following the const and volatile features in C++, a topology name is used as the type specifier and pointer qualifier names. The declaration of a pointer involved with a topology may be a

little difficult to understand if the reader is not familiar with the `const` and `volatile` features in C++.

In the example below, the `gp1` variable is a virtual processor local pointer variable which points to a scalar integer variable. Variable `gp2` is a scalar pointer variable which points to a virtual processor local integer variable. Variable `gp3` is a scalar pointer variable which points to a scalar integer variable. Variable `gp4` is a virtual processor local pointer variable which points to a scalar pointer variable pointing to a scalar integer variable.

```
 1 scalar int      *gp1;
 2 int             * scalar gp2;
 3 scalar int      * scalar gp3;
 4 scalar int      * scalar * gp4;
 5
 6 int             la1;
 7 scalar int      sa1;
 8
 9 sub1()
10 {
11     gp1 = &la1; // ERROR
12     gp1 = &sa1; // OK
13     gp2 = &la1; // OK
14     gp2 = &sa1; // ERROR
15     gp3 = &la1; // ERROR
16     gp3 = &sa1; // OK
17     gp4 = &la1; // ERROR
18     gp4 = &sa1; // ERROR
19     gp4 = &gp1; // ERROR
20     gp4 = &gp2; // ERROR
21     gp4 = &gp3; // OK
22 }
```

3.5 Commentary

We have been considering several notations for declaring scalar and parallel variables.

3.5.1 Type Specifier

In the first design of MPC++, we considered that the type of a variable on a topology is the combination of the ordinal C++ type and a topology. If a user wants to specify a new type of a variable on a topology, she/he has to specify the new type with the topology. In the following example, class `CL1` is declared on the one dimensional array topology.

```
1 class CL1 @ array[100] {
2     // class definition
3 };
4
5 CL1     aa;
```

The drawback of this approach is that it is difficult to support the inheritance mechanism which enables users to modify the class definition of CL1 as well as the array topology definition. Moreover, we cannot declare a variable whose dimension size of the topology is different than that in the `CL1` class since the dimension size is defined in the type specification.

3.5.2 Extended Template Feature

Another consideration was that the template feature is extended to define variables on a topology instead of introducing the new notation to declare them. In the following example, accessing of scalar variable `sa` is defined in the `scalar` template while accessing of two dimensional array variable `aa` is defined in the `array[M][N]` template.

```
1 scalar<int>              sa;
2 array[10][10]<int>       aa;
```

It seems to work fine if we do not apply it to a class. A question arises in the example below where variable `ac` has a `CL1` type on a one-dimensional array. The question is how the member methods of `CL1` are handled in the parallel way because the array template may just define the read/write accessing methods but can not define the member methods of the parameterized class (`CL1` in this case).

```
1 class           CL1;
2 array[10]<CL1>  ac;
```

3.5.3 Storage Class

The other consideration is that the specification of a topology is a sort of storage class such as `auto` and `register`. In C and C++, storage class specifies the storage lifetime or storage area and scope rule. It seems that the specification of a topology is a sort of storage area, which is fit for the C and C++ as shown below.

```
1 scalar int      a1;     // a1 is a scalar integer variable
2 scalar int      *a2;    // a2 is a scalar variable pointing to a local variable
```

However, we can not declare a variable pointing to a scalar variable since no pointer qualifier is introduced. If we introduced such a qualifier, then it would be better to follow the `const` and `volatile` features.

Concluding these considerations, we decided to treat a topology as like `const` and `volatile`.

4 Processors Group

Users may declare the group of a virtual processors to align parallel variables. In the following example, parallel variables `a1` and `a2` are allocated on processors group `p1` while variables `a3` and `a4` are allocated on processors group `p2`.

```
1 processors      p1(10);
2 processors      p2(10);
3 array int       a1@[100](p1, BLOCK);
4 array int       a2@[100](p1, BLOCK);
5 array int       a3@[100](p2, BLOCK);
6 array int       a4@[100](p2, BLOCK);
```

4.1 Dynamic Data Distribution

The data distribution manner and processors group involved with a parallel variable may be dynamically changed by the program. Those functions cannot be defined by the type of a parallel variable but are defined by the topology. To invoke a function defined by a topology, the following expressions are introduced.

mpc-topology-postfix-expression:
postfix-expression `..` *name*
postfix-expression `->>` *name*

The `postfix-expression` followed by `..` followed by *name* must be a parallel class object. The `name` must be a member function of the topology of the parallel object. The `postfix-expression` followed by `..` followed by *name* must be a pointer to a parallel class object. The `name` must be a member function of the topology of the parallel object.

An example of dynamic data distribution is shown below. The `array` topology defines dynamic data distribution functions `cycle` and `block`. Those functions are the same functionalities of the static data distribution functions.

```
 1 processors      p1(10);
 2 processors      p2(10);
 3 array int       a1@[100](p1, BLOCK);
 4 array int       a2@[100](p2, BLOCK);
 5 array int       *pa1 = a1;
 6 foo()
 7 {
 8         a1..cycle(2);
 9         pa1->>cycle(4);
10 }
```

An example of dynamic processors group change is shown below. The `array` topology defines the function of dynamic processors group change `on`.

```
1 bar()
2 {
3         a1..on(p2);
4         pa1->>on(p1);
5 }
```

5 Function Instance

A function instance is the abstraction of the thread of control in which the function is executed. The syntax of the function instance invocation statement is shown below.

mpc-function-instance-invocation:

mpc-simple-function-instance-invocation |
mpc-function-instance-invocation-with-continuation |
mpc-simple-function-instnace-invocation: |
postfix-expression (*expression-list*$_{opt}$) @ [*expression*$_{opt}$] |
mpc-function-instance-invocation-with-continuation: |
postfix-expression (*expression-list*$_{opt}$) @ (*expression*$_{opt}$) [*expression*$_{opt}$] |

```
1 void    hoo(int, int);
2 int     goo(int);
3 sub1()
4 {
5     int          i;
6     hoo(10, 20) @ [];
7     i = goo(10) @ [3];
8 }
```

Figure 4: Message Entry and Token

A simple example of the invocation of a function instance is shown in Figure 4. In line 6, the `hoo` function instance is invoked on a processor on which `sub1` is running. The `goo` function instance is created on processor #3 in line 7. The return value of the instance is assigned to variable `i`. The number enclosed in `[]` after `@` specifies the processor number where the function instance is created. In the case of a virtual layer, the processor number is a virtual processor number which must not exceed the number specified in the virtual processor layer statement, while the processor number is a physical processor number in the case of the physical layer.

5.1 Parameter on Function Instance

A pointer to any type in local memory may not be passed to a function instance invoked on a remote processor. Also, a structure containing a pointer to any type in local memory may not be passed to a function instance invoked on a remote processor.

In the following example, line 9 is an illegal statement because the `swap` function is invoked with a pointer variable to local storage. Line 10 is a legal statement because the function instance is locally invoked.

```
 1 swap(int *ip)
 2 {
 3     int          i = *ip;
 4     *ip = *(ip + 1); *(ip + 1) = i;
 5 }
 6 main()
 7 {
 8     int          ia[20];
 9     swap(ia) @ [10];    // error
10     swap(ia) @ [];      // ok
11     swap(ia);           // ok
12 }
```

A pointer to any type of scalar or a topology may be passed to a function instance invoked on a remote processor. In the following example, all statements are legal because the parameter of the swap function is a pointer to the scalar integer.

```
 1 swap(scalar int *ip)
 2 {
 3     int          i = *ip;
 4     *ip = *(ip + 1); *(ip + 1) = i;
 5 }
 6 main()
 7 {
 8     int          ia[20];
 9     swap(ia) @ [10];    // ok
10     swap(ia) @ [];      // ok
11     swap(ia);           // ok
12 }
```

5.2 Commentary

Since the MPC++ memory model is based on distributed memory machines, an argument of remote invocation is restricted. If arguments of a remote invocation might contain the local memory address, shared virtual memory or something like the same functionality had to be required to the runtime system. Because we do not assume such functionality in the runtime system, we restrict the usage of the arguments.

We believe that the argument restriction of MPC++ is not a serious disadvantage because a pointer to a scalar variable may be passed to a remote machine, though a pointer to local memory cannot be passed to a remote machine.

6 Message Entry and Token

6.1 Message Entry and Token Declaration

Communication between function instances is realized by two built-in data types, *message entry* and *token*, and operations performed on these types. A token is capable of sending one message to a specific message entry where the message is received. The syntax of message entry and token declarations is shown as follows:

mpc-message-passing-decl-specifier:
    mpc-mp-key (*argument-declaration-list*)
mpc-mp-key:
    entry
    token

In the following example, variables `t1` and `t2` are token variables whose token values are capable of sending a message containing an integer value and sending a message containing integer and character values, respectively. Entry labels `e1` and `e2` are entry points where a message containing an integer value and a message containing integer and character values are received, respectively.

```
1 token(int)              t1;
2 token(int, char)        t2;
3 entry(int)              e1;
4 entry(int, char)        e2;
```

6.2 Message Passing using Message Entry and Token

The syntax of message sending and receiving is shown below.

mpc-message-send-statement:
    mpc-tokenvar `<-` `[` *expression-list*$_{opt}$ `]`
    mpc-entryvar `<-` `[` *expression-list*$_{opt}$ `]`
mpc-message-receive-statement:
    mpc-entryvar `(` *argument-list*$_{opt}$ `)` `:`
mpc-tokenvar:
    identifier
mpc-entryvar:
    identifier
argument-list:
    name
    argument-list `,` *name*

The message entry has one token in the first instance. The token may be passed to another function instance so that the instance may send a message to the message entry. When the token for a message entry has been passed, no more message entry tokens are allowed to pass. Once a message is sent using a token, no more messages can be sent using that token. These properties guarantee the single message passing semantics on the *message entry* and *token* types.

An example of message entry and token variable declaration is shown below.

```
 1 void
 2 p1(token(int) t1)
 3 {
 4     t1 <- 10;
 5 }
 6 sub2()
 7 {
 8     entry(int)  l1;
 9     for (i = 0; i < 10; i++) {
10         p1(l1) @ ()[];  // the semantics of "@()[]" is described
11         ...;            //  in the following subsection.
12       l1(int k):/* receive */
13         ...;
14     }
15 }
```

Line 8 declares message entry `l1` in which an arrival message has an `int` value. Token variable `t1`, for a message whose argument is an integer value, is declared in line 2. The token of a message entry is extracted when a message entry is referenced. In other words, the R-value of a message entry is the token of that entry. For example, the `p1` function instance is invoked with the token of the `l1` message entry in line 10. After the execution of line 10, the R-value of the `l1` becomes `UNDEF`. The execution thread of `sub2` waits for a message at the `l1` message entry in line 12. After receiving a message at the entry, execution is resumed and the R-value of the `l1` message entry again becomes the token of the entry.

When the `p1` function instance is invoked by the `sub2` function instance, the token of the `l1` message entry is bound to the `t1` token variable. Operator `<-`, called the *send operator*, enables us to send a message to a message entry associated with the token. In line 4, `int` value 10 is sent to the `sub2` function instance.

The features of message entry and token types are summarized as follows:

- Single Reference of Message Entry Variable

  The R-value of a message entry variable is a token. After accessing the R-value of a message entry variable, the R-value becomes UNDEF. See Figure 5.

- Reusability of Token

  When a message is retrieved from a message entry variable, the R-value of the entry becomes the token of the entry (Figure 5).

- Single Reference of Token

  The R-value of a token variable is a token. After accessing the R-value of a token variable, the R-value becomes UNDEF (Figure 5).

- Communication Completion Guarantee

  When a function instance keeping a message entry variable on which no messages have

arrived exits, an exception may occur.

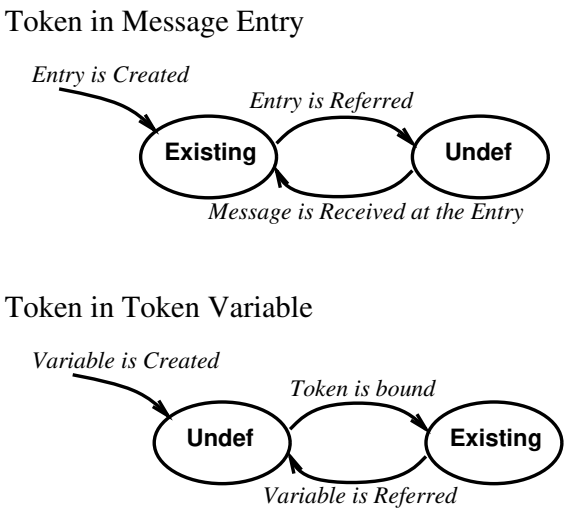


Figure 5: Message Entry and Token

6.3 Message Entry as Continuation

The value returned from a function instance is passed to the caller by the token. The program shown below is equivalent to that in Figure 4. In line 8, the `hoo` function instance is created. Message entry `l1` enclosed in `()` after `@`, specifies that the caller receives the return value at message entry. In line 10, the `goo` function instance is created on processor #3 and the return message is received at `l2` message entry in the following line. The return value is bound to variable `i`.

```
 1 void    hoo(int, int);
 2 int     goo(int);
 3 sub1()
 4 {
 5     int          i;
 6     entry(void) l1;
 7     entry(int)  l2;
 8     hoo(10, 20) @(l1)[];
 9   l1: // entry point
10     goo(10) @(l2)[3];
11   l2(i): // entry point
12 }
```

A function instance may access a token that is used to send a reply message to its caller. Because a function instance may transfer such a token to another instance, the token may be thought of as being a *continuation*. We call such a token a *continuation token*.

```
1 int goo(int a1, int a2)@(cont)
2 {
3     cont <- a1 + a2;
4 }
```

In the above programming example, variable `cont` is the continuation token variable of the `goo` function. The type of variable `cont` is `token(int)` because the function return type is `int`. When the `goo` function instance is invoked, variable `cont` is bound to a token associated with the message entry where the caller waits for the reply message. Since `cont` is an ordinal variable with type `token`, the variable may be passed to another function instance.

Figure 6 shows an example of *token continuation* programming. The `main` routine invokes function instance `hoo` with message entry `l1` in line 7 and awaits the arrival of the return value at `l1`. The `hoo` function instance obtains the continuation by variable `cont` in line 19. The type of variable `cont` is `token(int)` because the function return type is `int`. The continuation is passed to the `goo` function instance in line 22. When the `foo` function returns a value in line 29, that return value is sent to message entry `l1`. Then, the `main` routine receives the return value at `l1`.

```
 1 extern int hoo(int, int);    18 extern int goo(int);         26 int goo(int i)
 2 main()                       19 int hoo(int i, int j)@(cont) 27 {
 3 {                            20 {                            28   ...;
 4   int           i;           21   ...;                       29   return i + 3;
 5   entry(int)    l1;          22   goo(i) @ (cont) [5];       30 }
 6                              23   ...;
 7   hoo(10, 20) @ (l1) [3];    24 }
 8   // main and hoo runs
 9   // in parallel here
10  l1(i): // waiting for
11   ...;  // arriving a msg
12 }
```

Figure 6: Continuation Programming

In order to send a null message along with a token, expression `[]` is used as follows:

```
1 void
2 hoo(int a1, int a2)@(cont)
3 {
4     cont <- [];
5 }
```

6.4 Multiple Wait

To wait for multiple message entries, the `wait` statement is introduced as follows:

mpc-multiple-wait-statement:
`wait` *compound-statement*

An example is shown below. In line 9, messages arriving at entries `l1` and l2 are waited. When one of them arrives, the corresponding statements are executed. For example, when the message for `l1` arrives, statements denoted by `S1` in the example are executed. Since the function instance has a single thread of control, when the other message arrives during the execution, the message is enqueued. When execution reaches the `continue` statement, it is ready to handle the other message.

```
 1 extern int      p1();
 2 extern int      p2();
 3 bar()
 4 {
 5     entry(int)  l1;
 6     entry(int)  l2;
 7     p1() @ (l1)[10];
 8     p2() @ (l2)[20];
 9     wait {
10     l1(int i):
11         ...;// S1
12         continue;
13     l2(int i):
14         ...;// S2
15         continue;
16     }
17 }
```

6.5 Commentary

The usage of message entry and token is restricted as follows: a message must be received on the thread that declares the message entry point. In other words, the message entry point can not be passed to another thread. If users want to pass a message entry point, it is easy to define a class such as *I structure*[8] as shown in section 14.

Several questions might arise in terms of the design of message entry and token:

1. Why was a synchronization structure such as *I structure* not provided as a class instead of introducing the new concept ?
   Readers might think that providing a synchronization structure as a class is better than the MPC++ approach because it does not incur any language extensions. However, it is difficult to program multiple waits for arriving messages without introducing some language construct. Moreover, it is difficult to optimize expressions involving such a synchronization class because of the difficulty of tracing operations over the class. For example, is an instance of the class referred to by a single function instance or multiple function instances ? In contrast with the class approach, we know the optimal code generation for message entry and token because of its restriction described in [5]. A synchronization structure such as *I structure* is easily defined in a MPC++ class.

2. Why was a synchronization structure such as *I structure* not introduced as a basic type instead of introducing the new concept ?
   We believe that the message entry and token is more primitive than a synchronization structure such as *I structure*. Especially, the message entry and token is directly translated to the RWC-1[9] machine code in some cases.

3. Why was a sophisticated synchronization structure such as *Q structure*[10] not introduced ?
   We believe that such a sophisticated structure will be efficiently implemented by a MPC++ class. The introduction of message entry and token is small enough to implement such a structure.

4. Why does the declaration of token and entry not follow the syntax of `template` ?
   Because we can not define token and entry template classes whose template elements are of variable length.

7 Parallel Class

The parallel access specifier is introduced to declare a member function which is executed in the data parallel way. Such a member function is called a *parallel member function* while a class including a parallel member function is called a *parallel class*. The syntax is shown below.

mpc-parallel-access-specifier:
`parallel` *access-specifier*

A parallel class must be used with a topology as shown below:

```
1 class c0 { parallel public: ...; };
2 array c0        ic0@[100];      // OK
3 c0              ic1;            // ERROR
```

If a formal parameter type of a parallel member function is a simple C++ type, the caller must put a parallel variable or a simple variable of that type as its actual parameter. In the latter case, the actual parameter is distributed to the threads of the parallel member function that are involved with this computation.

The return value of a parallel member function is a parallel value of that return type except that the reduction function of that member function is declared. If the reduction function denoted by adding the `!` character to the function name is specified, the return value of the parallel member function is the return value of the reduction function. The syntax of a reduction function is shown as follows:

mpc-reduction-function-definition:
decl-specifiers `!` *declarator fct-body*

The example shown in Figure 7 defines a parallel class `c1` where a parallel operator function `+` and parallel function `aReduction` are defined.

In this example, the add operator `+` in line 23 is defined in line 7 which is a parallel function. Line 24 is an example of invoking a reduction function `aReduction` which is defined in lines 10 and 11. A function in line 10 is executed in each element of the topology. A function which has the `!` character is called a reduction function which reduces one value by two values returned from two elements of the topology.

```
 1 class c1 {
 2 private:
 3   int    b;
 4   int    c;
 5 parallel public:
 6   c1(int a1, int a2) { b = a1; c = a2; }
 7   c1 &operator+(c1 &a) {
 8     return c1(b + a.b, c + a.c);
 9   };
10   int aReduction() { return b + c; }
11   int !aReduction(int lv, int rv)
12         { return lv + rv; }
13 parallel private:
14   c1 &operator-(c1 &a) {
15     return self->b - a.b;
16   };
17 };
18
19 array c1        o1@[100], o2@[100], o3@[100];
20 cex()
21 {
22    int   i;
23    o1 = o2 + o3;
24    i = o1.aReduction();
25 }
```

Figure 7: An Example of a Parallel Class

7.1 Function invocation by a parallel function

The following question might arise: when a parallel function invokes a function which refers to a local variable, which virtual processor is involved with this computation, since accessing such a local variable depends on the virtual processor. Since the representation of a parallel variable is distributed to virtual processors, we can know which virtual processor is involved with such computation. The programmer must take into account the connection between a parallel variable and virtual processors when a parallel function invokes a function which will access a local variable.

7.2 Commentary

Besides the control parallel features, the readers will wonder what are differences than features in the pC++[2].

The pC++ introduces the special class called the `collection`. A collection is a set of elements which are C++ objects. A collection is defined by an extended class feature that has the `MethodOfElement` member data and functions. A collection object is a parallel object in the sense that functions of the element's objects and functions of `MethodOfElement` in the collection are executed in the parallel way.

As long as we understand, a collection provides the mechanism of data parallel computa-

tion. So programming in a collection is rather machine independent than programming in a MPC++ topology which specifies the implementation of parallel computation and distribution. In MPC++, the programmers may program the machine dependent implementation of distribution and data parallel operations if they want. Programming in MPC++ topology is shown in section 9.

8 Parallel Statements

We have so far described language primitives for describing a program in the MIMD and data parallel way, i.e., function instance, message entry and token, and parallel class. In this section, parallel statements to control the execution threads and a data parallel statement are described.

The MPC++ language specification does not specify the priority of a thread. However, a thread may be suspended and another thread will be resumed in the multiple threaded environment. For example, when a thread hits a memory space which has been paged out, the thread must be suspend and another thread will be resumed. Thus, the mutual exclusion of multiple threads is the responsibility of the programmers. MPC++ provides `atomic` and `mutex` statements to realize the mutual exclusion.

8.1 Atomic

The `atomic` statement is provided so that a thread executes a compound statement led by `atomic` without preemption by other threads. The syntax of the `atomic` statement is shown below.

mpc-atomic-statement:
`atomic` *compound-statement*

Note that the message receiving statement and function call are not allowed in the compound statement in the `atomic` statement.

8.2 Mutex

Another general mutual exclusion facility is the `mutex` statement. The syntax of the `mutex` statement is shown as follows:

mpc-mutex-statement:
`mutex` (*expression*) *compound-statement*

The expression in a `mutex` statement must be an object of class `Mutex` which realizes the mutual exclusion. An example of a `mutex` statement is shown below. `S11` and `S21` are executed exclusively in this example.

```
 1 Mutex   rr;
 2 sub1()
 3 {
 4     S10;
 5     mutex (rr) {
 6         S11;
 7     }
 8     S12;
 9 }
10 sub2()
11 {
12     S20;
13     mutex (rr) {
14         S21;
15     }
16     S22;
17 }
```

8.2.1 With Statement

Following C*[12], statement `with` is provided for parallel primitive data variables.

mpc-with-statement:
`with` (*expression*) *compound-statement*

An example of a `with` statement is shown below. The argument of a `with` statement is a parallel variable in MPC++ while the argument in C* is a *shape* which is a sort of topology.

```
 1 mul(array int a1@[10][10], array int a2@[10][10])
 2 {
 3   with (a1) {
 4     int i, sum = 0;
 5     for (i = 0; i < 10; i++) {
 6       sum +=
 7         a1@[pos(0)][i]*a2@[i][pos(0)];
 8     }
 9     a1 = sum;
10   }
11 }
```

In the above example, macro function `pos` is only available inside the `with` statement, taking an integer value as dimension and returns the index of that dimension. For example, `pos(0)` returns the index of the X axis of the current element of the parallel variable being processed.

As shown in line 7, an element of the parallel variable may be referred to by putting the indexes after `@`.

The semantics of a function invoked within the `with` statement follows the invocation semantics in the parallel member function. That is, the caller's virtual processor executes the invoked function.

8.3 Commentary

8.3.1 Implementation of atomic statement

The implementation of the `atomic` statement depends on the machine and operating system. An example of the implementation is shown. A flag for implementing the `atomic` statement is reserved in the shared memory area between the user and runtime library or kernel spaces. On entering the `atomic` statement, the flag is set to some value while the flag is reset to the initial value when leaving the statement. At leaving time, if the flag value is not equal to the value set before, the runtime library has set the flag because some higher priority thread has been ready to run. In that case, the execution yields to the higher priority thread.

8.3.2 With statement

Since we already have linguistic support for describing a parallel class, providing a `with` statement seems unnecessary. What we can describe in the `with` statement can be also described in a parallel class. We think that the relationship between the `with` statement and a parallel class is a sort of the relationship between a function and a class in the sense that what we can describe in a function can be also described in a class. MPC++ provides the `with` statement for users who prefer it.

The consideration of invoking a function within the `with` statement was that such a function must be defined in the same topology of the variable declared in the `with` statement. An example is shown below. In this example, function `foo` is declared on two dimensional array topology. This means that the `foo` function is defined on each element of the topology. In line 5, a local variable `i` is defined on each parallel variable `a1` element. Each parallel variable element invokes function `foo` with parameters, its local variable `i` and its element, in line 6.

```
1 extern array void foo(int i, int j) @[10][10];
2 sub2(array int a1@[10][10])
3 {
4   with (a1) {
5     int i;
6     foo(i, a1);
7   }
8 }
```

Following accessing a parallel variable, function `foo` may be also called in the following syntax:

```
foo(10, a1) @ [1][2];
```

MPC++ version 1.0 does not allow such a function definition.

9 Physical Processor Layer

9.1 File Scope Variable

A variable declaration in the file scope is dealt with as the processor local variable in the physical processor layer. That is, the storage of such a variable is allocated on each physical processor. A function may access the local storage of the variable where the function runs.

In the following example, storages of `a1` and `d1` are allocated on each physical processor.

```
1 physical layer;
2 int     a1;
3 double  d1;
```

9.2 Physical Modifier

The **physical** modifier of the class declaration is introduced. An instance of a class with the **physical** modifier is allocated as the same address and size on each physical processor. This modifier is also effective on the virtual processors and C and C++ layers.

mpc-physical-calss-head:
`physical` *class-head*

An example below declares the physical class `PARI` and the `pa1` instance of that class in the virtual processor layer. The storage of the `pa1` is allocated as the same address and size on each physical processor.

```
1 virtual processors[10] layer;
2 physical class PARI {
3  public:
4     int         i;
5 };
6 PARI    pa1;
```

The physical class is used to implement several MPC++ data parallel features. The detailed description is described later.

A member function of a physical class is allowed to invoke a remote member function. In the example shown below, a physical class `FOO` has member functions `go`, `l1`, and `l2`.

```
 1 physical class FOO {
 2  public:
 3     int         count;
 4     int         total;
 5     token(void) cont;
 6     processors  p1;
 7     int         go()@(cnt) {
 8                         if (pno(P1) != 0) {
 9                             error ("Must be received by PE #0\n");
10                         } else {
11                             this->l1(pno(p1))@()[left(p1)];
12                             this->l2(pno(p1))@()[right(p1)];
13                             count = 2;
14                             total = 0;
15                             cont = cnt;
16                         }
17                     }
18     void        l1(int t1) {
19                         if (pno(p1) == 0) {
20                             atomic {
21                                 total += t1;
22                                 if (--count == 0) {
23                                     cont <- [ total ];
24                                 }
25                             }
26                         } else {
27                             this->l1(t1 + pno(p1))@()[left(p1)];
28                         }
29                     }
30     void        l2(int t1) {
31                         if (pno(p1) == 0) {
32                             atomic {
33                                 total += t1;
34                                 if (--count == 0) {
35                                     cont <- [ total ];
36                                 }
37                             }
38                         } else {
39                             this->l2(t1 * pno(p1))@()[right(p1)];
40                         }
41                     }
42     Foo(processors pp) { p1 = pp; }
43 };
```

When the `go` member function is invoked on processor 0, it invokes member functions `l1` and `l2` on processors specified by `left(p1)` and `right(p2)`, respectively. Macro expressions `left` and `right` return left and right hand side processors of the current execution processor, respectively. The semantics of left and right hand sides is of course implementation dependent.

```
 1 main()
 2 {
 3     FOO         *fp = new (p1) FOO(p1);
 4     entry(void) cont;
 5
 6     fp->go()@(cont)[0];
 7     ...;
 8   cont():
 9     ...;
10 }
```

In line 3, an instance of class `FOO` is created. In C++, parenthesis after the `new` keyword specifies the object allocation area. Following this semantics, if an instance of class `processors` described later appears in the parenthesis, the instance is created on processors specified by the `processors` instance. If the object allocation is omitted, the physical class object is allocated on all processors.

9.3 Remote Pointer

A `remote` pointer is introduced to declare a pointer to remote memory space. By providing this feature, the compiler may generate special code to refer to remote memory space on a distributed memory parallel machine. This feature is also available in the virtual processor layer.

mpc-remote-pointer:
`remote *`

In the following example, variable `gp1` is a pointer to a remote integer variable and `gp2` is a pointer to a remote variable which is a pointer to a remote integer variable.

```
1 int remote      *gp1;
2 int remote * remote *gp2;
```

The MPC++ version 1.0 provides two builtin macros for a remote pointer variable, `peaddr` and `lmemaddr`. As shown in the example below, macro `peaddr` returns the processor element number while macro `lmemaddr` returns the remote local memory address.

```
1 void foo(int remote*gi)
2 {
3     peaddr(gi);         // return the processor element number
4     lmemaddr(gi);       // return the remote local memory address
5 }
```

9.4 Class processors

The MPC++ programming language specification reserves the special physical class called `processors` to represent a processors group.

```
 1 physical class processors {
 2  private:
 3     const int           physpeno;
 4     const int           maxpe;
 5     // for barrier synchronization
 6     entry(token(PNO))   ent;
 7     int                 syncstart(int phase);
 8     int                 syncsend(token(PNO) pno) { ent <- [ pno ]; }
 9     PNO                 syncrec() { token(PNO) tt; ent(tt): return tt; }
10  public:
11     processors(int nproc);
12     void        sync() { return syncstart(2); }
13 };
```

9.5 Implementation of Virtual Processor Local Variables

The implementation of all virtual processor local variables that have primitive data types such as `int`, `short`, and `char` may be redefined using the physical `VPLOCAL$WORD` class. The declaration of the class is shown below.

```
1 template<class T>
2 physical class VPLOCAL$WORD {
3  public:
4     T           data;
5     T           read();
6     void        write(T);
7 };
```

When the compiler looks at the declaration of the `VPLOCAL$WORD` physical class, acceesing of all scalar variables that have a primitive data is translated to invoke the *read* or *write* member function of that class.

If the programmer specifies the `VPLOCAL$WORD` physical class as follows, it means that all virtual processor local variables of primitive data are the physical processor local variables.

```
1 template<class T>
2 physical class VPLOCAL$WORD {
3  public:
4     T           data;
5     T           read() { return data; }
6     void        write(T) { data = T; }
7 };
```

To control all virtual processor local variables of structures, unions, and classes, the `SCALAR$BLOCK` physical class is introduced as follows:

```
 1 template<class T>
 2 physical class VPLOCAL$BLOCK {
 3     T           data;
 4  public:
 5     $macrofunc  readMember(`pos) { $comcall1("vplocal_block_read", `pos); }
 6     $macrofunc  writeMember(`pos, `expr)
 7                 { $comcall2("vplocal_block_write", `pos, `expr); }
 8     $macrofunc  funcall(`func, `arglist)
 9                 { $comcall2("vplocal_funcall", `func, `arglist); }
10 };
```

The macro expression `$macrofunc` and `$comcall` which appear above are explained in section 10. If you specify the `VPLOCAL$BLOCK` physical class as follows, it means that all virtual

processors local variables of structure, union, and class are physical processor local.

```
1 template<class T>
2 physical class VPLOCAL$BLOCK {
3     T           data;
4  public:
5     $macrofunc  readMember(‘pos) { data.‘pos; }
6     $macrofunc  writeMember(‘pos, ‘expr) { data.‘pos = ‘expr; }
7     $macrofunc  funcall(‘func, ‘arglist) { data.‘func(‘arglist); }
8 };
```

9.6 Definition of Topology

Topologies `scalar` and all others such as `array` may be defined using the *topology* constructor which is an extended template feature. In this subsection, the extended syntax is described.

mpc-topology-constructor:
    `topology` *mpc-topology-dimen-arglist*$_{opt}$ `<` *template-argument-list* `>` *declaration*
mpc-topology-dimen-arglist:
    mpc-topology-dimen-arg mpc-topology-dimen-arglist
mpc-topology-dimen-arg:
    `[` *identifier* `]`

The scalar topology is predefined as follows:

```
 1 topology <class T>
 2 physical class scalar {
 3  public:
 4     T           data;
 5     T           read();
 6     void        write(T);
 7     T&          operator*++();
 8     T&          operator*--();
 9     T&          operator++*();
10     T&          operator--*();
11 };
```

The runtime routine specifies member functions, `read` and `write`, and operators of the pointer. Since those routines might be defined as inline functions, the performance will be almost the same as the implementation of the scalar feature by the compiler. Rather, this approach is more flexible and portable in the sense that the implementation of the scalar feature is programmable by users.

It should be noted that the syntax of operator functions of the pointer shown in the above program are introduced in MPC++. Operator functions post increment, post decrement, pre increment, and pre decrement are declared in lines 7 to 10, respectively.

mpc-operator: one of
    `*++` `*--` `++*` `--*`

Examples below show a skeleton of one dimensional and two dimensional `array` topologies represented by physical classes. Such a class is called a *topology class*. Member data of the topology class includes i) `data` storing a parallel variable on a physical processor, ii) `dist` containing parameters for data distribution, and iii) `pp` representing physical processors group. Member functions of the topology class include i) `pno` calculating the physical processor number from the index, ii) `disp` calculating the local address on the physical processor from the index, iii) pointer operations, and iv) data distribution functions. The detailed description of how to program user defined topology appears later.

```
 1 topology[N]<class T>
 2 physical class array {
 3     T           *data;          // data area on a physical processor
 4     Dist        dist[1];        // description of the data distribution
 5     processors  pp;
 6  public:
 7     virtual PNO pno(int i);     // return the physical processor NO of the
 8                                 // i'th data on the array
 9     virtual int disp(int i);    // return the local address on the physical
10                                 // processor of the i'th data on the array
11     array(Distribution);
12     distcycle();
13     distblock();
14     T&          operator*++();
15     T&          operator*--();
16     T&          operator++*();
17     T&          operator--*();
18 };
```

```
 1 topology[M][N]<class T>
 2 physical class array {
 3     T           *data;          // data area on a physical processor
 4     Dist        dist[2];        // description of the data distribution
 5     processors  pp;
 6  public:
 7     virtual PNO pno(int);       // return the physical processor NO of the
 8                                 // i'th data on the array
 9     virtual int disp(int i);    // return the local address on the physical
10                                 // processor of the i'th data on the array
11     void        array(Distribution);
12     void        distcycle();
13     void        distblock();
14     T&          operator*++();
15     T&          operator*--();
16     T&          operator++*();
17     T&          operator--*();
18 };
```

The users may define their own topology inherited from predefined topology or from scratch. The following example declares the one dimensional `userarray` topology from `array`. The extension is to add the `userdist` function which might be a special data distribution.

```
1 topology[N]<class T>
2 physical class userarray : public array[N]<T> {
3  public:
4     void        userdist();
5 };
```

9.6.1 Dist Structure

```
 1 struct Dist {
 2     int         start;  // start position where the physical processor takes care
 3     int         end;    // end position where the physical processor takes care
 4     int         size;   // end - start
 5     int         intval; // interval between the current position and the next
 6                         // position in the CYCLE distribution manner
 7     int         scale;  // multiplied by the scale to calculate the local address
 8     int         pscale; // multiplied by the psclae to calculate the physical
 9                         // process number on n dimensional physical processors
10     int         limit;  // total size
11 };
```

The declaration of the `Dist` structure is shown above. The structure is kept in a physical class which represents a topology. Members `start` and `end` represent the start and end positions where the physical processor keeps elements of the parallel variable. The `size` member represents the number of elements, i.e., `end` $-$ `start`.

Member `intval` is used if the data distribution manner is `CYCLE`. In that case, each physical processor keeps parallel variable elements specified by the following expression where *a@[i]* stands for an *i*th element of parallel variable:

$$a@[i] \quad \text{for } i \ = \ start \text{ until } i \ < \ end \text{ step } \ intval$$

In the case of `BLOCK`, `intval` is the same value of `limit`. Member `scale` is to calculate the local address of a parallel variable. A parallel variable whose dimension is more than two is allocated in a local memory space whose address is linear. The `scale` parameter is used to map n-dimensional axes to linear addressing. If the physical processors are addressed by n-dimensional axes, the `pscale` parameter is used to map n-dimensional axes to linear addressing of the physical processor number. The next subsection will describe the calculation in detail.

9.6.2 Address Calculation

Member functions `pno` and `disp` in a topology take indexes and return the physical processor number and the local address of the processor, respectively. Those functions may specify the following algorithms by using the `Dist` structure:

Let assume that a parallel variable `p` is defined in a topology whose `dist` member, a structure of `Dist`, is `b`. The physical processor number of `p@[`i_0`][`i_1`]...[`i_n`]` is given by expression (1).

$$\Sigma_{j=0}^{n} \ ((i_j / b.size_j) \% b.intval_j) * b.pscale_j \tag{1}$$

The local memory address of `p@[`i_0`][`i_1`]...[`i_n`]` is given by expression (2).

$$\Sigma_{j=0}^{n} \; (\; i_j \% b.size_j \; + \; (i_j / (b.size_j * b.intval_j)) * b.size_j) * b.scale_j \qquad (2)$$

Note that $p@[i_0][i_1]...[i_n]$ exists in a local memory if the following expression is hold:

$$^{\forall j=0..n}(b.start_j \leq i_j \% b.intval_j < b.end_j)$$

9.6.3 Examples of Data Distribution

Examples of data distribution parameters are shown in Figure 8. For example, the address of `a1@[7][3]` is calculated as follows using the `Dist` structure.

$$\begin{aligned} PE &= ((7/p1.size0)\%p1.intval0) * p1.pscale0 \\ &\quad + ((3/p1.size1)\%p1.intval1) * p1.pscale1 \\ &= ((7/4)\%10) * 1 \; + \; ((3/10)\%10) * 0 = 1 \end{aligned}$$

$$\begin{aligned} LADDR &= (7\%p1.size0 \; + \; (7/(p1.size0 * p1.intval0)) * p1.size0) * p1.scale0 \\ &\quad + (3\%p1.size1 \; + \; (3/(p1.size1 * p1.intval1)) * p1.size1) * p1.scale1 \\ &= (7\%4 \; + \; (7/(4 * 10)) * 4) * 10 \; + \; (3\%10 \; + \; (3/(10 * 10)) * 10) * 1 \\ &= 3 * 10 \; + \; 3 = 33 \end{aligned}$$

9.7 Runtime Support for Virtual Processors

The MPC++ programming language specification does not assume that the number of physical processors is defined at the compiling time. Mapping from virtual processors to physical processors are dynamically performed.

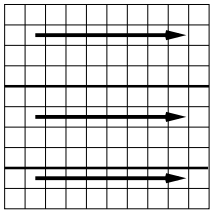
Figure 9 shows an example of the `startup` runtime routine to map virtual processors to physical processors. Function `startup` will be called on each physical processor at the startup time.

Variables `num_vproc` and `num_vpgroup` represent the number of total virtual processors and the number of total virtual processors group required by the user, respectively. The `proclist` data type is to list the virtual processors groups declared by the user. Variable `allvproc`, a pointer to `proclist`, is bound to the top address of the virtual processors group list. Variable `userproc` represents the virtual processors defined in the virtual processor layer statement. Variables `num_vproc`, `num_vpgroup`, `proclist`, `allvproc`, and `userproc` are created by the `mpccollect` tool described later.

The `PPG` processors variable initialized at runtime represents all the processors that are available at the runtime. In line 16, `PPG.processors` stores the number of all processors

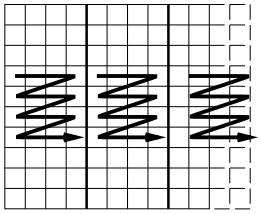
```
processors     vp1(3);
processors     vp2(3, 3);
```

```
array int    a1@[10][10](vp1, BLOCK, ALL);
```



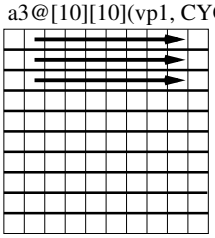
```
p1.start0 = 0;          p1.start0 = 4;          p1.start0 = 8;
p1.end0 = 4;            p1.end0 = 8;            p1.end0 = 10;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 10;
p1.size0 = 4;           p1.size0 = 4;           p1.size0 = 2;
p1.scale0 = 10;         p1.scale0 = 10;         p1.scale0 = 10;
p1.pscale0 = 1;         p1.pscale0 = 1;         p1.pscale0 = 1;
p1.start1= 0;           p1.start1= 0;           p1.start1= 0;
p1.end1 = 10;           p1.end1 = 10;           p1.end1 = 10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 10;
p1.size1 = 10;          p1.size1 = 10;          p1.size1 = 10;
p1.scale1 = 1;          p1.scale1 = 1;          p1.scale1 = 1;
p1.pscale1 = 0;         p1.pscale1 = 0;         p1.pscale1 = 0;
```

```
array int    a2@[10][10](vp1, ALL, BLOCK);
```



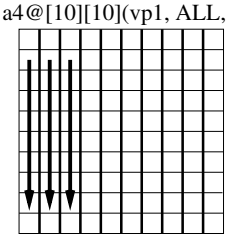
```
p1.start0 = 0;          p1.start0 = 0;          p1.start0 = 0;
p1.end0 = 10;           p1.end0 = 10;           p1.end0 = 10;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 10;
p1.size0 = 10;          p1.size0 = 10;          p1.size0 = 10;
p1.scale0 = 4;          p1.scale0 = 4;          p1.scale0 = 4;
p1.pscale0 = 0;         p1.pscale0 = 0;         p1.pscale0 = 0;
p1.start1= 0;           p1.start1= 4;           p1.start1= 8;
p1.end1 = 4;            p1.end1 =8;             p1.end1 =10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 10;
p1.size1 = 4;           p1.size1 = 4;           p1.size1 = 4;
p1.scale1 = 1;          p1.scale1 = 1;          p1.scale1 = 1;
p1.pscale1 = 1;         p1.pscale1 = 1;         p1.pscale1 = 1;
```

```
array int    a3@[10][10](vp1, CYCLE(1)K, ALL);
```

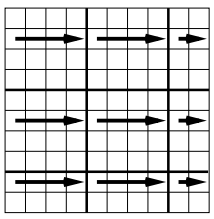


```
p1.start0 = 0;          p1.start0 = 1;          p1.start0 = 2;
p1.end0 = 1;            p1.end0 = 2;            p1.end0 = 3;
p1.interval0 = 3;       p1.interval0 = 3;       p1.interval0 = 3;
p1.size0 = 1;           p1.size0 = 1;           p1.size0 = 1;
p1.scale0 = 10;         p1.scale0 = 10;         p1.scale0 = 10;
p1.pscale0 = 1;         p1.pscale0 = 1;         p1.pscale0 = 1;
p1.start1= 0;           p1.start1= 0;           p1.start1= 0;
p1.end1 = 10;           p1.end1 = 10;           p1.end1 = 10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 10;
p1.size1 = 10;          p1.size1 = 10;          p1.size1 = 10;
p1.scale1 = 1;          p1.scale1 = 1;          p1.scale1 = 1;
p1.pscale1 = 0;         p1.pscale1 = 0;         p1.pscale1 = 0;
```

```
array int    a4@[10][10](vp1, ALL, CYCLE(1));
```



```
p1.start0 = 0;          p1.start0 = 0;          p1.start0 = 0;
p1.end0 = 10;           p1.end0 = 10;           p1.end0 = 10;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 10;
p1.size0 = 10;          p1.size0 = 10;          p1.size0 = 10;
p1.scale0 = 1;          p1.scale0 = 1;          p1.scale0 = 1;
p1.pscale0 = 0;         p1.pscale0 = 0;         p1.pscale0 = 0;
p1.start1= 0;           p1.start1= 1;           p1.start1= 2;
p1.end1 = 1;            p1.end1 =2;             p1.end1 = 3;
p1.interval1 = 3;       p1.interval1 = 3;       p1.interval1 = 3;
p1.size1 = 1;           p1.size1 = 1;           p1.size1 = 1;
p1.scale1 = 1;          p1.scale1 = 1;          p1.scale1 = 1;
p1.pscale1 = 1;         p1.pscale1 = 1;         p1.pscale1 = 1;
```



```
p1.start0 = 0;          p1.start0 = 0;          p1.start0 = 0;
p1.end0 = 4;            p1.end0 = 4;            p1.end0 = 4;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 10;
p1.size0 = 4;           p1.size0 = 4;           p1.size0 = 4;
p1.scale0 = 4;          p1.scale0 = 4;          p1.scale0 = 4;
p1.pscale0 = 3;         p1.pscale0 = 3;         p1.pscale0 = 3;
p1.start1= 0;           p1.start1= 4;           p1.start1= 8;
p1.end1 = 4;            p1.end1 = 8;            p1.end1 = 10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 10;
p1.size1 = 4;           p1.size1 = 4;           p1.size1 = 2;
p1.scale1 = 4;          p1,scale1 = 4;          p1.scale1 = 4;
p1.pscale1 = 3;         p1.pscale1 = 3;         p1.pscale1 = 3;

p1.start0 = 4;          p1.start0 = 4;          p1.start0 = 4;
p1.end0 = 8;            p1.end0 = 8;            p1.end0 = 8;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 10;
p1.size0 = 4;           p1.size0 = 4;           p1.size0 = 4;
p1.scale0 = 4;          p1.scale0 = 4;          p1.scale0 = 4;
p1.pscale0 = 3;         p1.pscale0 = 3;         p1.pscale0 = 3;
p1.start1= 0;           p1.start1= 4;           p1.start1= 8;
p1.end1 = 4;            p1.end1 = 8;            p1.end1 = 10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 10;
p1.size1 = 4;           p1.size1 = 4;           p1.size1 = 2;
p1.scale1 = 4;          p1.scale1 = 4;          p1.scale1 = 4;
p1.pscale1 = 3;         p1.pscale1 = 3;         p1.pscale1 = 3;

p1.start0 = 8;          p1.start0 = 8;          p1.start0 = 8;
p1.end0 = 10;           p1.end0 = 10;           p1.end0 = 10;
p1.interval0 = 10;      p1.interval0 = 10;      p1.interval0 = 104;
p1.size0 = 2;           p1.size0 = 2;           p1.size0 = 2;
p1.scale0 = 4;          p1.scale0 = 4;          p1.scale0 = 4;
p1.pscale0 = 3;         p1.pscale0 = 3;         p1.pscale0 = 3;
p1.start1= 0;           p1.start1= 4;           p1.start1= 8;
p1.end1 = 4;            p1.end1 = 8;            p1.end1 = 10;
p1.interval1 = 10;      p1.interval1 = 10;      p1.interval1 = 104;
p1.size1 = 4;           p1.size1 = 4;           p1.size1 = 2;
p1.scale1 = 4;          p1.scale1 = 4;          p1.scale1 = 4;
p1.pscale1 = 3;         p1.pscale1 = 3;         p1.pscale1 = 3;
```

Figure 8: `Dist` Structures Corresponding to Distribution Strategies

```
 1 physical layer;
 2 extern int              num_vproc;      // # of virtual processors
 3 extern int              num_vpgroup;
 4 extern proclist         *allvproc;
 5 extern processors       userproc;
 6 extern processors       PPG;
 7 void
 8 startup(int nproc)
 9 {
10     subprocessors       *gp;
11     proclist            *plist;
12     int                 power2;
13     int                 np;
14
15     PPG.processors = nproc;
16     if (PPG.processors > num_vproc) {
17         for (plist = allvproc; plist; plist = plist->next) {
18             gp = PPG.asgn_subgroup(plist->pnum);
19             gp->vproc_mapping(plist->vproc);
20         }
21         gp = PPG.rest_subgroup();
22         gp->vproc_mapping(&userproc);
23     } else {
24         power2 = pow(2.0, (double) ((int) log2((double)PPG.processors)));
25         np = powe2/num_vpgroup;
26         for (plist = allvproc; plist; plist = plist->next) {
27             gp = PPG.asgn_subgroup(np);
28             gp->vproc_mapping(plist->vproc);
29         }
30         gp = PPG.rest_subgroup();
31         gp->vproc_mapping(&userproc);
32     }
33 }
```

Figure 9: Dynamic Physical Processors Mapping

that may be involved in this computation. Member function `asgn_subgroup` of `PPG` is to divide physical processors group to sub groups. In line 18, the sub processors group, whose processors number is specified by `plist->pnum`, is reserved. The return value of function `asgn_group` is a `subprocessors` object which represents the sub processors group. Member function `vproc_mapping` of a `subprocessors` object maps virtual processors to a physical processors group.

If the user has not defined the `startup` routine, the MPC++ runtime system will use the default `startup` routine.

9.7.1 Compiler and Linker

```
1 virtual processors(100) layer;
2 processors      p1[10][20];
3 processors      p2[5];
```

Assuming that a user declares virtual processors as shown above, the MPC++ compiler adds the following symbol information to the symbol table of the object code.

```
1 $VPROC$_userproc$100$
2 $VPROC$_p1$10$20$
3 $VPROC$_p2$5$
```

The `mpccollect` routine, which must be invoked between compile and link time, searches strings beginning with `$VPROC$` in the symbol table and generates the following assembler file. Variables `num_vproc`, `num_vpgroup`, and `allvproc`, referred to in the `startup` routine, are defined in this file.

```
 1 num_vproc:      .word   305
 2 num_vpgroup:    .word   3
 3 allvproc:
 4         .word   userproc
 5         .word   vplist_1
 6 vplist_1:
 7         .word   _p1
 8         .word   vplist_2
 9 vplist_2:
10         .word   _p2
11         .word   0
```

9.8 Commentary

To realize the SPMD execution in the physical processor layer, we have considered the `physpar` statement shown below. The purpose of introducing the `physpar` statement was to provide:

1. the same program execution start point on the specified processors group.
2. an efficient implementation of multi-threading in the same context that means local variables are shared in the same context.
3. an efficient implementation of a thread invocation in the remote context.

Instead of introducing the new statement more, we have designed the physical class so that those requirements are satisfied.

The syntax and semantics of the `physpar` statement is defined as follows:

mpc-physical-par-statement:
`physpar` (*name*) *mpc-par-com*$_{opt}$ *compound-statement* *mpc-receive-block*$_{opt}$
mpc-par-com:
`via` (*argument-declaration-list*)
mpc-receive-block:
`=>` *compound-statement*

The `physpar` statement takes i) a `processors` variable, ii) an optional `communication declaration` part beginning with the `via` keyword, iii) a compound statement, and iv) an optional compound statement beginning with `=>`. The first compound statement is called the computation block while the second compound statement is called the receiver block. The receiver block consists of statements starting with the message receive statement and statements

starting with the `init:` label. The statements lead by the `init:` label are called the communication initialization part. The statements may include the `break` and `continue` statement.

The semantics of the `physpar` statement execution is as follows:

- If the communication initialization part is specified, the part is executed in processors specified in the `processors` variable. Execution ends when the `continue` statement is reached. Those processors are synchronized at the end of execution.
- The computation block is executed in processors specified in the `processors` variable.
- When a message is received at an entry variable specified in the `communication declaration` part, the execution of the computation block is suspended and the statements led by the message receive statement are executed. When execution reaches the `break` statement, execution ends and the suspended execution is resumed.
- When the `break` statement is executed in the computation block or the receive block, the `physpar` statement ends.

An example is shown below.

```
 1 processors      p1(100);
 2 main()
 3 {
 4     int t1, t2;
 5
 6     physpar (p1) via(entry(int) l1, entry(int) l2) {
 7         if (pno(p1) == 0) {
 8             l1@[left(p1)] <- [ pno(p1) ];
 9             l1@[right(p1)] <- [ pno(p1) ];
10         }
11     } => {
12     init:
13         continue;
14     l1(t1):
15         if (pno(p1) != 0) {
16             l1@[left(p1)] <- [ t1 + pno(p1) ];
17         }
18         continue;
19     l2(t2):
20         if (pno(p1) != 0) {
21             l1@[right(p1)] <- [ t1 * pno(p1) ];
22         }
23         continue;
24     }
25     printf ("t1 = %d\n", t1);
26     printf ("t2 = %d\n", t2);
27 }
```

10 Macro

In this document, just the syntax of macro statements is given below.

mpc-macro-statement:
`$macro` *identifier* `(` `)` *compound-statement*
`$macro` *identifier* `(` *expression* `)` *compound-statement*
`$macro` *identifier* `(` *expression* `;` *expression* `)` *compound-statement*
`$macro` *identifier* `(` *expression* `;` *expression* `;` *expression* `)` *compound-statement*
`$macro` *identifier* `(` *expression* `;` *expression* `;` *expression* `;` *expression* `)` *compound-statement*
`$macro` *identifier* `(` *expression* `;` *expression* `;` *expression* `;` *expression* `;` *expression* `)` *compound-statement*

mpc-macro-func:
`$macrofunc` *decl-specifier identifier* `(` *mpc-macro-param-list*$_{opt}$ `)` *mpc-macro-func-body*
mpc-macro-addonfunc:
`$macroaux` *identigier* `(` *mpc-macro-param-list*$_{opt}$ `)` *mpc-macro-func-body*
mpc-macro-param-list:
mpc-macro-var
mpc-macro-param-list `,` *mpc-macro-var*
mpc-macro-var:
`‘` *identifier*
mpc-macro-func-body:
mpc-macro-compound-statement

mpc-macro-compound-statement:
`{` *mpc-macro-statement* `}`
mpc-macro-statement:
mpc-macro-if-statement
mpc-macro-expression `;`
mpc-macro-expression:
mpc-macro-function-call
mpc-macro-comcall
mpc-macro-var
mpc-macro-if-statement:
`$if` `(` *mpc-macro-expression* `)` *mpc-macro-compound-statement*
`$if` `(` *mpc-macro-expression* `)` *mpc-macro-compound-statement*
`else` *mpc-macro-compound-statement*
mpc-macro-comcall:
`$comcall1` `(` *string-constant* `,` *mpc-macro-expression* `)`
`$comcall2` `(` *string-constant* `,` *mpc-macro-expression* `,` *mpc-macro-expression* `)`
`$comcall3` `(` *string-constant* `,`
mpc-macro-expression `,` *mpc-macro-expression* `,` *mpc-macro-expression* `)`
`$comcall4` `(` *string-constant* `,`
mpc-macro-expression `,` *mpc-macro-expression* `,` *mpc-macro-expression* `,`
mpc-macro-expression `)`

mpc-macro-new-class:
`$newclass` *identifier*
mpc-macro-hook:
`$hook` *mpc-macro-hook-where identifier mpc-macro-var*
mpc-macro-hook-where:
`classheader`
`memberfunc`
`memberdata`

11 Linkage Specification

Using the linkage specification feature of C++, we define function call sequences. Because the MPC++ programming model is based on multi-threaded execution, the stack implementation for the function invocation might be complicated. If every function invocation is implemented without stack, its overhead would be increased in some machines. In such an implementation, it is better to declare which function should be called with stack by the programmer. The compiler takes this information and may generate good code. In order to realize it, the `stack` linkage specification is introduced.

There are several assembler files existing, especially in the operating system kernel. To link to such assembler routines, the `asm` linkage specification is introduced.

In the following example, function `malloc` may be called using the stack while function `getDS128` may be called based on the assembler linkage specification.

```
1 extern "stack" {
2         void    *malloc(int);
3 };
4 extern "asm" {
5         void    *getDS128();
6 };
```

12 Real-Time Constructs

We are currently designing the real-time extension of MPC++. Many ideas will be derived from RTC++[7]. In this section, facilities we decided to import so far are described.

12.1 Time Constant

MPC++ allows users to specify a time constant. Examples are as follows:

`0t1991Y08M20D12h30m45s50.200`	*means*	1991.8.20 12:30:45 50 milli. 200 micro.
`0t8h20m30s10.20`		8:20:30 10 millisecond 20 microsecond

12.2 Within, At, Before Statements

within-statement:
`within` *(expression) compound-statement exception-handling-block*$_{opt}$
at-statement:
`at` *(expression) compound-statement exception-handling-block*$_{opt}$
before-statement:
`before` *(expression) compound-statement exception-handling-block*$_{opt}$

Statements *within*, *at*, and *before* express statement-level timing constraints. A *within* statement expresses the duration of execution. An *at* statement expresses the constraint of starting time while a *before* statement expresses the constraint of ending time.

An example of a *within* statement is as follows:

```
1 within(time) {
2     ...;
3 } except {
4 case timeout:
5     ...;
6 }
```

The *time* variable keeps an instance of the `Time` class which is supported by the ARTS kernel. We can specify the duration of a time [1]. When the execution of statements surrounded by ``within(time) {'' and ``}'' cannot be finished within the time specified by the `time` value, statements led by ``case timeout:'' are executed.

12.3 Cycle Statement

cycle-statement:
cycle-clause compound-statement exception-handling-block$_{opt}$

A cycle statement specifies a periodic task. As shown below, a cycle statement can be followed by an exception handling block:

```
1 cycle(starttime; endtime; period; deadline) {
2    ...;
3 } except {
4 case timeout:
5    ...;
6 }
```

In this example, `starttime`, `endtime`, and `deadline` specify starting time, ending time, period, and the deadline time of the execution respectively. Those are instances of class `Time`.

[1] For example, 0t1s20.10 means that the duration of a time is second 20 milliseconds 10 microseconds.

13 OS Programming Supports

To implement a multi-threaded operating system kernel, we are ready to support some linguistic facilities. Those facilities will be made clear through writing an operating system kernel called *SCore* which will run on RWC-1.

14 Programming Examples

14.1 I Structure

A synchronization structure *I structure* is implemented as class *Istructure* shown below.

```
 1 class Istructure {
 2 private:
 3     entry(Object remote*)       cont;
 4 public:
 5     void        write(Object remote*v)
 6                 { cont <- v; }
 7     Object      *read()
 8                 { Object    remote*val;
 9                   cont(val):
10                   return val; }
11 };
```

A thread invoking member function `read` will wait for receiving a message at the `cont` message entry in the `read` function. A thread invoking member function `write` sends a message via a token of message entry `cont`. The following example shows the use of the `Istructure` class.

```
1 extern void bar(Istructure remote*);
2 test(int n)
3 {
4     Istructure  *pi1 = new Istructure;
5     bar(pi1)@()[n];
6     pi1->read();
7 }
```

14.2 Q Structure

The Q structure (formerly named *restricted I-structure*[10]) is implemented by the `Istructure` class and its list class called `IstructureList`. The following program list is an example of the Q structure. Handling the `IstructureList` object is a critical region among threads where exclusive execution is guaranteed by the `atomic` statement.

```
 1 class IstructureList {
 2 private:
 3   IstructureList        *next;
 4   Istructure            is;
 5 public:
 6   IstructureList() { next = NIL; }
 7   write(Object *ov)
 8                 { is.write(ov); }
 9   Object        *read()
10                 { return is.read(); }
11 };
12 class Qstructure {
13 private:
14   int                   state;
15   IstructureList        *top;
16   IstructureList        *end;
17 public:
18   Qstructure() { top = end = NIL; }
19   void  write(Object *v);
20   Object        *read();
21 };
```

```
 1 void
 2 Qstructure::write(Object *v)
 3 {
 4   atomic {
 5     if (top == NIL) {
 6       top = end = new IstrucutreList;
 7       top->write(v);
 8       state = WROTE;
 9     } else {
10       if (state == WAITING_WRITE) {
11         top->write(v);
12         /* remove the list */
13         top = top->next;
14         if (top == NIL) end = NIL;
15       } else {
16         end->next
17               = new IstructureList;
18         end = end->next;
19         end->write(v);
20       }
21     }
22   }
23 }
24
25 Object  *
26 Qstructure::read()
27 {
28   Object                *ov;
29   IstructureList        *il;
30
31   atomic {
32     if (top == NIL) {
33       top = new IstrucutureList;
34       il = end = top;
35       state = WAITING_WRITE;
36     } else {
37       if (state == WROTE) {
38         il = top;
39         top = top->next;
40         if (top == NIL) end = NIL;
41       } else {
42         il = new IstructureList;
43         end = end->next = il;
44       }
45     }
46   }
47   ov = il->read();
48   delete il;
49   return ov;
50 }
```

14.3 Stream Class

Byte stream communication is easily realized by the Q structure if the communication network has the FIFOness property. In the following `Stream` class example, we implement it in another way using message entry and tokens.

```
 1 #define BUF_SIZE        8
 2 class WStream;
 3 class Stream {
 4     friend WStream;
 5 private:
 6     entry(Object*)      buffer[BUF_SIZE];
 7     int                 ptr;
 8     token(entry(Object*)[BUF_SIZE])
 9                         ok;
10     WStream             *ws;
11 public:
12     Stream()    {   ptr = 0;
13                     ws = new WStream(buffer);
14                     ok = ws->sem;  }
15     Object      *get() {
16                     Object      *val;
17
18                   buffer[ptr](Object *val):
19                     if (++ptr > BUF_SIZE) {
20                         ok <- [ buffer ];
21                       buffer[0](ok):
22                         ptr = 0;
23                     }
24                     return val;
25                 }
26 };
```

```
 1 class WStream {
 2 private:
 3     token(Object*)      buffer[BUF_SIZE];
 4     entry(entry(Object*)[BUF_SIZE])
 5                         sem;
 6     int                 ptr;
 7 public:
 8     RStream(token(Object*) bb[BUF_SIZE])
 9                 { buffer = bb; ptr = 0; }
10     put(Object *val)
11                 {retry:
12                   if (ptr < BUF_SIZE) {
13                       buffer[ptr] <- [ val ];
14                       ptr++;
15                   } else {
16                      sem(buffer):
17                       buffer[0] <- [ sem ];
18                       ptr = 1;
19                       goto retry;
20                   }
21                 }
22 };
```

In this example, arrays of a message entry and token are used to send and receive byte data. When all message entries and tokens are consumed the receiver sends the sender an array of message tokens.

Acknowledgment

The author would like to thank Professors Akinori Yonezawa, Satoshi Matsuoka, Jiro Tanaka, Kokichi Futatsugi, Takuo Watanabe, Dr. Mitsuhisa Sato, Dr. Satoshi Hirano, and other members of RWC massively parallel programming language working group for their crucial comments on the designing of MPC++ Version 1.0. The basic MPC++ programming model is inspired by the RWC-1 massively parallel machine which has been designed by Dr. Shunichi Sakai, who is the chief architect of the RWC-1, and his group of RWC Tsukuba Research Center. The many intensive discussions on MPC++ have been done with Atsushi Hori, Hiroki Konaka, Munenori Maeda, and Takashi Tomokiyo, members of RWC Tsukuba Research Center. The author would like to thank those Tsukuba Research Center members for their contributions.

The first version of the MPC++ programming environment has been launched by the efforts of Tsutom Horikawa who wrote the RWC-1 assembler, Tadashi Sato who adapted the gnu ld to the RWC-1 architecture, and Hiroshi Harada who wrote the RWC-1 function-level simulator. The author would like to thank them for their efforts.

Discussions with Nobuyuki Ichiyoshi, Toshihiro Nishioka, Daigo Sekita, members of RWC MRI massively parallel distributed laboratory, have been very fruitful. The author would like to thank them.

Finally, the author would like to express his gratitude to Dr. Junichi Shimada, director of the RWC partnership.

References

[1] "Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++," ANSI, Doc No. X3J16/93-0062, WG21/N0269, June 1993.

[2] Dennis Gannon, J. K. Lee, "Object Oriented Parallelism: pC++ Ideas and Experiments," Proceedings of 1991 Joint Symposium of Parallel Processing, pp. 13 - 22, 1991.

[3] Margaret A. Ellis, Bjarne Stroustrup, "The Annotated C++ Reference Manual," Addison-Wesley Publishing Company, ISBN 0-201-51459-1, 1990.

[4] High Performance Fortran Forum, "High Performance Fortran Language Specification," 1993.

[5] Yutaka Ishikawa, "The Implementation Notes of MPC++," RWC Technical Reports, to appear soon.

[6] Yutaka Ishikawa, Hiroki Konaka, Munenori Maeda, Takashi Tomokiyo, Atsushi Hori, "An Overview of Massively Parallel Programming Language:MPC++," IPSJ SIG Notes 93-PRG-13, Information Processing Society of Japan, August 1993. (In Japanese)

[7] Yutaka Ishikawa, Hideyuki Tokuda, Clifford W. Mercer, "Object-Oriented Real-Time Language Design: Constructs for Timing Constraints," ACM OOPSLA'90 Proceedings, 1990. Also CMU-CS-90-111.

[8] R.S., Nikhil, ,K.K. Pingali, "I-Structure: Data Structures for Parallel Computing," ACM Transaction on Programming Languages and Systems Volume 11 number 4, pp. 598 – 639, 1989.

[9] Shuichi Sakai, Andrew Shaw, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, Kazuaki Okamoto, Mitsuhisa Sato, Takashi Yokota, "RICA: Reduced Interprocessor Communication Architecture – Concept and Mechanisms –," Proceedings of the 5th IEEE Symposium on Parallel and Distributed Processing, pp. 122 – 125, Decemeber 1993.

[10] Satoshi Sekiguchi, Toshio Shimada, Kei Hiraki, "Sequential Description and Parallel Execution Language DFC II for Dataflow Supercomputers," ACM 1991 International Conference on Supercomputing, pp. 57 - 66, June 1991.

[11] Bjarne Stroustrup, "The C++ Programming Language Second Edition," Addison-Wesley Publishing Company, ISBN 0-201-53992-6, 1991.

[12] Thinking Machines Corporation, "C* Programming Guide," 1993

A MPC++ Syntax

The syntax described in this section is based on the draft standard of C++[1].

A.1 Keyword

class-name:
    identifier
    template-class-id
enum-name:
    identifier
typedef-name:
    identifier

A.2 Expressions

expression:
    assignment-expression
    expression , *assignment-expression*
assignment-expression:
    conditional-expression
    unary-expression assignment-operator assignment-expression
    throw-expression
assignment-operator: one of
    = *= /= %= += -= >>= <<= &= ^= |=
conditional-expression:
    logical-or-expression
    logical-or-expression ? *expression* : *assignment-expression*
logical-or-expression:
    logical-and-expression
    logical-or-expression || *logical-and-expression*
logical-and-expression:
    inclusive-or-expression
    logical-and-expression && *inclusive-or-expression*
inclusive-or-expression:
    exclusive-or-expression
    inclusive-or-expression | *exclusive-or-expression*
exclusive-or-expression:
    and-expression
    exclusive-or-expression ^ *and-expression*
and-expression:
    equality-expression
    and-expression & *equality-expression*
equality-expression:
    relational-expression

equality-expression `==` *relational-expression*
equality-expression `!=` *relational-expression*
relational-expression:
shift-expression
relational-expression `<` *shift-expression*
relational-expression `>` *shift-expression*
relational-expression `<=` *shift-expression*
relational-expression `>=` *shift-expression*
shift-expression:
additive-expression
shift-expression `<<` *additive-expression*
shift-expression `>>` *additive-expression*
additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*
multiplicative-expression:
pm-expression
multiplicative-expression `*` *pm-expression*
multiplicative-expression `/` *pm-expression*
multiplicative-expression `%` *pm-expression*
pm-expression:
cast-expression
pm-expression `.*` *cast-expression*
pm-expression `->*` *cast-expression*
cast-expression:
unary-expression
`(` *type-id* `)` *cast-expression*
unary-expression:
postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof (` *type-id* `)`
new-expression
delete-expression
unary-operator: one of
`*` `&` `+` `-` `!` `~`
new-expression:
`::`$_{opt}$ `new` *new-placement*$_{opt}$ *new-type-id new-initizlizer*$_{opt}$
`::`$_{opt}$ `new` *new-placement*$_{opt}$ `(` *type-id* `)` *new-initizlizer*$_{opt}$
new-placement:
`(` *expression-list* `)`
new-type-id:
type-specifier-seq new-declarator$_{opt}$
new-declarator:

```
        * cv-qualifier-seq_opt new-declarator_opt
        qualified-class-specifier :: * cv-qualifier-seq_opt new-declarator_opt
        direct-new-declarator
new-initializer:
        direct-new-declarator_opt [ expression ]
new-initializer:
        ( expression-list_opt )
delete-exrepssion:
        ::_opt delete cast-expression
        ::_opt delete [ ] cast-expression
postfix-expression:
        primary-expression
        postfix-expression [ expression ]
        postfix-expression ( expression-list_opt )
        simple-type-specifier ( expression-list_opt )
        postfix-expression . id-expression
        postfix-expression -> id-expression
        postfix-expression ++
        postfix-expression --
        dynamic_cast < type-name > ( expression )
        typeid ( expression )
        typeid ( type-name )
        mpc-topology-expression                                              |
        mpc-topology-postfix-expression                                      |
        mpc-function-instance-invocataion                                    |
mpc-topology-expression:                                                     |
        postfix-expression @ mpc-topology-index                              |
mpc-topology-index:                                                          |
        mpc-topology-index [ expression ]                                    |
mpc-topology-postfix-expression:                                             |
        postfix-expression .. name                                           |
        postfix-expression ->> name                                          |
mpc-function-instance-invocation:                                            |
        mpc-simple-function-instance-invocation                              |
        mpc-function-instance-invocation-with-continuation                   |
mpc-simple-function-instnace-invocation:                                     |
        postfix-expression ( expression-list_opt ) @ [ expression_opt ]      |
mpc-function-instance-invocation-with-continuation:                          |
        postfix-expression ( expression-list_opt ) @ ( expression_opt ) [ expression_opt ]  |
expression-list:
        assignment-expression
        expression-list , assignment-expression
primary-expression:
        literal
        this
        :: identifier
        :: operator-function-id
```

`::` *qualified-id*
`(` *expression* `)`
id-expression

id-expression:
identifier
operator-function-id
conversion-function-id
`~` *class-name*
qualified-id

qualified-id:
nested-class-specifier `::` *id-expression*

A.3 Literal

literal:
integer-literal
character-literal
floating-literal
string-literal

integer-literal:
decimal-literal integer-suffix$_{opt}$
octal-literal integer-suffix$_{opt}$
hexadecimal-literal integer-suffix$_{opt}$

decimal-literal:
nonzero-digit
decimal-literal digit

octal-literal:
0
octal-literal octal-digit

hexadecimal-literal:
`0x` *hexadecimal-digit*
`0x` *hexadecimal-digit*
hexadecimal-literal hexadecimal-digit

nonzero-digit: one of
`1 2 3 4 5 6 7 8 9`

octal-digit: one of
`0 1 2 3 4 5 6 7`

hexadecimal-digit: one of
`0 1 2 3 4 5 6 7 8 9`
`a b c d e f`
`A B C D E F`

integer-suffix:
unsigned-suffix long-suffix$_{opt}$
long-suffix unsigned-suffix$_{opt}$

unsigned-suffix: one of
`u U`

long-suffix: one of
`l` `L`

character-literal:
`'` *c-char-sequence* `'`
`L'` *c-char-sequence* `'`

c-char-sequence:
c-char
c-char-sequence c-char

c-char:
any member of the source character set execpt
the single-quate, backslash, or new-line character
escape-sequence

escape-sequence:
simple-escape-sequence
octal-escape-sequence
hexadecimal-escape-sequence

simple-escape-sequence: one of
`\'` `\*` `\'` `\?` `\\`
`\a` `\f` `\b` `\n` `\r` `\t` `\v`

octal-escape-sequence:
`\` *octal-digit*
`\` *octal-digit octal-digit*
`\` *octal-digit octal-digit octal-digit*

hexadecimal-escape-sequence:
`\x` *hexadecimal-digit*
hexadecimal-escape-sequence hexadecimal-digit

floating-constant:
fractional-constant exponent-part$_{opt}$ *floating-suffix*$_{opt}$
digit-sequence exponent-part floating-suffix$_{opt}$

fractional-constant:
digit-sequence$_{opt}$ `.` *digit-sequence*
digit-sequence `.`

exponent-part:
`e` *sign*$_{opt}$ *digit-sequence*
`E` *sign*$_{opt}$ *digit-sequence*

sign: one of
`+` `-`

digit-sequence:
digit
digit-sequence digit

floating-suffix: one of
`f` `l` `F` `L`

string-literal:
`"` *s-char-sequence*$_{opt}$ `"`
`L` *s-char-sequence*$_{opt}$ `"`

s-char-sequence:
s-char

s-char-sequence s-char

s-char:

any member of the source character set except
the double-quote, backslash, or new-line character

escape-sequence

A.4 Declarations

declaration:

decl-specifier-seq$_{opt}$ *init-declarator-list*$_{opt}$;

mpc-variable-on-topology |

asm-definition

function-definition

template-declaration

linkage-specification

mpc-variable-on-topology: |

mpc-type-specifier decl-specifier-seq declarator-list ; |

mpc-type-specifier: |

`scalar` |

`global` |

`array` |

`btree` |

decl-specifier:

storage-class-specifier

type-specifier

function-specifier

template-specifier

mpc-message-passing-decl-specifier |

`friend`

`typedef`

decl-specifier-seq:

decl-specifier-seq$_{opt}$ *decl-specifier*

mpc-message-passing-decl-specifier: |

mpc-mp-key (*argument-declaration-list*) |

mpc-mp-key: |

`entry` |

`token` |

storage-classs-specifier:

`auto`

`register`

`static`

`extern`

function-specifier:

`inline`

`virtual`

type-specifier:

simple-type-specifier
class-specifier
enum-specifier
elaborated-type-specifier
`::` *typedef-name*
`const`
`volatile`

simple-type-specifier:
qualified-class-specifier
qualified-type-specifier
char
wchar_t
short
int
long
signed
unsigned
float
double
void

elaborated-type-specifier:
class-key identifier
class-key qualified-class-specifier `::` *identifier*
`enum` *identifier*
`enum` *qualified-class-specifier* `::` *identifier*

class-key:
`class`
`struct`
`union`

qualified-type-specifier:
typedef-name
class-name `::` *qualified-type-specifier*

qualified-class-specifier:
nested-class-specifier
`::` *nested-class-specifier*

nested-class-specifier:
class-name
class-name `::` *nested-class-specifier*

enum-specifier:
`enum` $identifier_{opt}$ `{` $enumerator\text{-}list_{opt}$ `}`

enumerator-list:
enumerator
enumerator-list `,` *enumerator*

enumerator:
identifier
identifier `=` *constant-expression*

constant-expression:
conditional-expression

linkage-specification:
`extern` *string-literal* { *declaration-seq*$_{opt}$ }
`extern` *string-literal declaration*
declaration-seq:
declaration
declaration-seq declaration
asm-definition:
`asm` (*string-literal*)

A.5 Declarators

init-declarator-list:
init-declarator
init-declarator-list , *init-declarator*
init-declarator:
declarator initializer$_{opt}$
declarator:
direct-declarator
ptr-operator declarator
direct-declarator:
declarator-id
declarator-id `@` *mpc-topology-dimen mpc-distribution-hint*$_{opt}$ |
direct-declarator (*parameter-declaration-clause*) *cv-qualifier-seq*$_{opt}$
exception-specification$_{opt}$
direct-declarator [*constant-expression*$_{opt}$]
(*declarator*)
mpc-topology-dimen: |
mpc-topology-dimen [*constant-expression*] |
ptr-operator:
`*` *mpc-pointer-qualifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$ |
`&` *mpc-pointer-qualifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$ |
qualified-class-specifier `::` `*` *mpc-pointer-qualifier*$_{opt}$ *cv-qualifier-seq*$_{opt}$ |
mpc-remote-pointer |
mpc-remote-pointer: |
`remote *` |
mpc-pointer-qualifier: |
`scalar` |
`global` |
`array` |
`btree` |
mpc-distribution-hint: |
(*mpc-proc-group*$_{opt}$ *mpc-distribution-hint-elem*) |
mpc-proc-group: |
identifier , |
mpc-distribution-hint-elem:
mpc-distribution-hint-macro |

mpc-distribution-hint-elem , *mpc-distribution-hint-macro* |

cv-qualifier-seq:

cv-qualifier cv-qualifier-seq$_{opt}$

cv-qualifier:

`const`

`volatile`

declarator-id:

id-expression

qualified-type-specifier

type-id:

type-specifier-seq abstract-declarator$_{opt}$

type-specifier-seq:

type-specifier type-specifier$_{opt}$

abstract-declarator:

ptr-operator abstarct-declarator$_{opt}$

direct-abstract-declarator

direct-abstract-declarator:

direct-abstract-declarator$_{opt}$ `(` *parameter-declaration-clause* `)`
cv-qualifier-seq$_{opt}$ *exception-specification*$_{opt}$

direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`

`(` *abstract-declarator* `)`

parameter-declaration-clause:

parameter-declaration-list$_{opt}$ `...`opt

parameter-declaration-list , `...`

parameter-declaration-list:

parameter-declaration

parameter-declaration-list , *parameter-declaration*

parameter-declaration:

decl-specifier-seq declarator

decl-specifier-seq declarator `=` *expression*

decl-specifier-seq abstract-declarator$_{opt}$

decl-specifier-seq abstract-declarator$_{opt}$ `=` *expression*

function-definition:

decl-specifier-seq$_{opt}$ *declarator ctor-initializer*$_{opt}$ *function-body*

function-body:

compound-statement

initializer:

`=` *initializer-clause*

`(` *expression-list* `)`

initializer-clause:

assignment-expression

`(` *initializer-list* ,$_{opt}$ `)`

initializer-list:

initializer-clause

initializer-list , *initializer-clause*

A.6 Class Declarations

class-specifier:
    class-head { *member-specification*$_{opt}$ }
    mpc-physical-class-head { *member-specification*$_{opt}$ } |
class-head:
    class-key identifier$_{opt}$ *base-clause*$_{opt}$
    class-key nested-class-specifier base-clause$_{opt}$
mpc-physical-calss-head: |
    `physical` *class-head* |
member-specification:
    member-declaration member-specification$_{opt}$
    access-specifier : *member-specification*$_{opt}$
    mpc-parallel-access-specifier : *member-specification*$_{opt}$ |
member-declaration:
    decl-specifier-seq$_{opt}$ *member-declarator-list*$_{opt}$;
    function-difinition ;$_{opt}$
    qualified-id ;
    mpc-reduction-function-definition |
mpc-reduction-function-definition: |
    decl-specifiers ! *declarator fct-body* |
member-declarator-list:
    member-declarator
    member-declarator-list , *memeber-declarator*
member-declarator:
    declarator pure-specifier$_{opt}$
    identifier$_{opt}$: *constant-expression*
pure-specifier:
    = 0
base-clause:
    : *base-specifier-list*
base-specifier-list:
    base-specifier
    base-specifier-list , *base-specifier*
base-specifier:
    qualified-class-specifier
    `virtual` *access-specifier*$_{opt}$ *qualified-class-specifier*
    access-specifier `virtual`$_{opt}$ *qualified-class-specifier*
mpc-parallel-access-specifier: |
    `parallel` *access-specifier* |
access-specifier:
    `private`
    `protected`
    `public`
conversion-function-id:
    `operator` *conversion-type-id*

conversion-type-id:
type-specifier-seq ptr-operator$_{opt}$
ctor-initializer:
`:` *mem-initializer-list*
mem-initializer-list:
mem-initializer
mem-initializer `,` *mem-initializer-list*
mem-initalizer:
qualified-class-specifier `(` *expression-list*$_{opt}$ `)`
identifier `(` *expression-list*$_{opt}$ `)`
operator-function-id:
`operator` *operator*
operator: one of
`new` `delete` `new[]` `delete[]`
`+` `-` `*` `/` `%` `^` `&` `|` `=`
`!` `=` `<` `>` `+=` `-=` `*=` `/=` `%=`
`^=` `&=` `|=` `<<` `>>` `>>=` `<<=` `==` `!=`
`<=` `>=` `&&` `||` `++` `--` `,` `->*` `->`
`()` `[]`
mpc-operator |
mpc-operator: one of |
`*++` `*--` `++*` `--*` |

A.7 Statements

statement:
labaled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement
declaration-statement
try-block
mpc-message-send-statement |
mpc-message-receive-statement |
mpc-multiple-wait-statement |
mpc-atomic-statement |
mpc-mutex-statement |
mpc-with-statement |
mpc-where-statement |
labeled-statement:
identifier `:` *statement*
`case` *constant-expression* `:` *statement*
`default` `:` *statement*

expression-statement:
    expression$_{opt}$;
compoud-statement:
    { *statement-seq*$_{opt}$ }
statement-seq:
    statement
    statement-seq statement
selection-statement:
    `if` (*condition*) *statement*
    `if` (*condition*) *statement* `else` *statement*
    `switch` (*condition*) *statement*
condition:
    expression
    type-specifier declarator `=` *expression*
iteration-statement:
    `while` (*condition*) *statement*
    `do` *statement* `while` (*condition*)
    `for` (*for-init-statement condition*$_{opt}$; *expression*$_{opt}$) *statement*
for-init-statement:
    expression-statement
    declaration-statement
jump-statement:
    `break` ;
    `continue` ;
    `return` *expression*$_{opt}$;
    `goto` *identifier* ;
declaration-statement:
    declaration
mpc-message-send-statement:
    mpc-tokenvar `<-` `[` *expression-list*$_{opt}$ `]`
    mpc-entryvar `<-` `[` *expression-list*$_{opt}$ `]`
mpc-message-receive-statement:
    mpc-entryvar (*argument-list*$_{opt}$) :
mpc-tokenvar:
    identifier
mpc-entryvar:
    identifier
argument-list:
    name
    argument-list , *name*
mpc-multiple-wait-statement:
    `wait` *compound-statement*
mpc-atomic-statement:
    `atomic` *compound-statement*
mpc-mutex-statement:
    `mutex` (*expression*) *compound-statement*
mpc-with-statement:

`with` (*expression*) *compound-statement* |
mpc-where-statement: |
`where` (*expression*) *compound-statement* |

A.8 Templates

template-declaration:
`template` < *template-parameter-list* > *declaration*
template-parameter-list:
template-parameter
template-parameter-list , *template-parameter*
template-parameter:
type-parameter
parameter-declaration
type-parameter:
`class` *identifier*
template-class-id:
template-name < *template-argument-list* >
template-name:
identifier
template-argument-list:
template-argument
template-argument-list , *template-argument*
template-argument:
expression
type-id

A.9 Exception Handling

try-block:
`try` *compound-statement handler-seq*
handler-seq:
handler handler-seq$_{opt}$
handler:
`catch` (*exception-declaration*) *compound-statement*
exception-declaration:
type-specifier-seq declarator
type-specifier-seq abstract-declarator
type-specifier-seq
...
throw-expression:
`throw` *assignement-expression*$_{opt}$
exception-specification:

`throw` (*type-id-list*$_{opt}$)

type-id-list:

type-id

type-id-list , *type-id*

A.10 MPC++ Layer Declaration

mpc-layer-declaration:

`physical layer`

`virtual processors` (*constant-expression*) `layer`

A.11 MPC++ Topology Declaration

mpc-topology-constructor:

`topology` *mpc-topology-dimen-arglist*$_{opt}$ `<` *template-argument-list* `>` *declaration*

mpc-topology-dimen-arglist:

mpc-topology-dimen-arg mpc-topology-dimen-arglist

mpc-topology-dimen-arg:

`[` *identifier* `]`

A.12 RTC++ Extension

within-statement:

`within` *(expression) compound-statement exception-handling-block*$_{opt}$

at-statement:

`at` *(expression) compound-statement exception-handling-block*$_{opt}$

before-statement:

`before` *(expression) compound-statement exception-handling-block*$_{opt}$

cycle-statement:

cycle-clause compound-statement exception-handling-block$_{opt}$

A.13 EMC Extension

emc-task-block-statement:

`[` *emc-task-control-prefix* `]` `dowith` *statement* `[` *emc-task-postfix* `]`

`[` *emc-task-control-prefix* `]` `forkwith` *statement*

emc-task-control-prefix:

`where` (*expr*)

`everywhere`

`iterate` (*expr*)

emc-task-post-fix:

```
updatevariables ( variable-list )
reductionsum ( variable-list )
reductionmax ( variable-list )
reductionmin ( variable-list )
reductionor ( variable-list )
reductionand ( variable-list )
```

A.14 Preprocessor

B MPC++ Compiler V1.0 R0.1

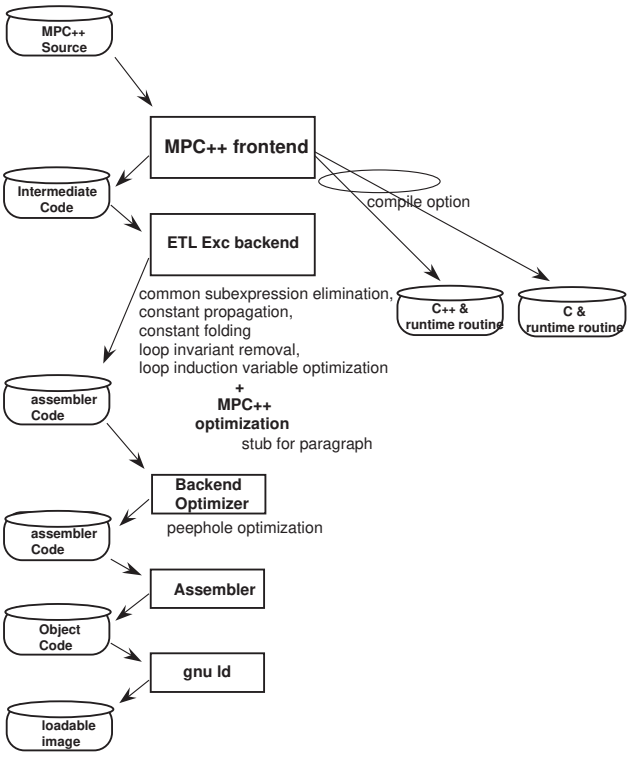


Figure 10: MPC++ Compiler V1.0 R0.1

Contents

1 Introduction **1**

1.1 The Notation 1

1.2 Organization 2

2 Abstraction Layers **2**

2.1 Physical Processor Layer 3

2.1.1 Variables 3

2.1.2 Parallel Flow Control and Communication 3

2.2 Virtual Processor Layer 4

2.2.1 Variables 4

2.2.2 Parallel Flow Control and Communication 6

2.3 C and C++ layer 6

2.4 Commentary 6

3 Variables **6**

3.1 File Scope Variable 7

3.2 Scalar and Parallel Variables 7

3.3 Accessing Variables on Topology 10

3.4 Pointer and Topology 10

3.5 Commentary 11

3.5.1 Type Specifier 11

3.5.2 Extended Template Feature 12

3.5.3 Storage Class 12

4 Processors Group **12**

4.1 Dynamic Data Distribution 13

5 Function Instance **13**

5.1 Parameter on Function Instance 14

5.2 Commentary 15

6 Message Entry and Token **15**

6.1 Message Entry and Token Declaration 15

6.2 Message Passing using Message Entry and Token 16
6.3 Message Entry as Continuation 18
6.4 Multiple Wait 20
6.5 Commentary 21

7 Parallel Class **22**
7.1 Function invocation by a parallel function 23
7.2 Commentary 23

8 Parallel Statements **24**
8.1 Atomic 24
8.2 Mutex 24
8.2.1 With Statement 25
8.3 Commentary 26
8.3.1 Implementation of atomic statement 26
8.3.2 With statement 26

9 Physical Processor Layer **27**
9.1 File Scope Variable 27
9.2 Physical Modifier 27
9.3 Remote Pointer 29
9.4 Class processors 29
9.5 Implementation of Virtual Processor Local Variables 30
9.6 Definition of Topology 31
9.6.1 Dist Structure 33
9.6.2 Address Calculation 33
9.6.3 Examples of Data Distribution 34
9.7 Runtime Support for Virtual Processors 34
9.7.1 Compiler and Linker 36
9.8 Commentary 37

10 Macro **38**

11 Linkage Specification **40**

12 Real-Time Constructs **40**

12.1 Time Constant 40

12.2 Within, At, Before Statements 41

12.3 Cycle Statement 41

13 OS Programming Supports **42**

14 Programming Examples **42**

14.1 I Structure 42

14.2 Q Structure 42

14.3 Stream Class 44

A MPC++ Syntax **47**

A.1 Keyword 47

A.2 Expressions 47

A.3 Literal 50

A.4 Declarations 52

A.5 Declarators 54

A.6 Class Declarations 56

A.7 Statements 57

A.8 Templates 59

A.9 Exception Handling 59

A.10 MPC++ Layer Declaration 60

A.11 MPC++ Topology Declaration 60

A.12 RTC++ Extension 60

A.13 EMC Extension 60

A.14 Preprocessor 61

B MPC++ Compiler V1.0 R0.1 **61**