TR-96-012

Multiple Threads Template Library

– MPC++ Version 2.0 Level 0 Document –

Document Revision 0.13

Yutaka Ishikawa*
ishikawa@rwcp.or.jp

Revised on 17 Septempber 1997
Revised on 30 July 1997

Received 26 September 1996

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

Abstract This document describes a C++ template library for multi-threaded programming in MPC++, called MPC++ multiple threads template library. It contains i) `invoke` and `ainvoke` function templates for synchronous and asynchronous local/remote thread invocation, ii) `Sync` class template for synchronization and communication among threads, iii) `GlobalPtr` class template for pointer to remote memory, iv) `Reduction` class template for reduction, v) `Barrier` class for barrier synchronization, and vi) `yield` function to suspend the thread execution and yield another thread execution.

*Massively Parallel Software Laboratory

Contents

1 Introduction

MPC++ version 1.0, an extension of C++, has been designed for parallel/distributed programming[2]. Instead of setting several language extensions, we have designed low-level parallel description primitives and the MPC++ metalevel architecture[3] to realize an extendable/modifiable programming language system. Higher parallel/distributed constructs, such as data parallel statements and distributed active objects, are implemented using the extendable feature.

The parallel description primitives were designed to form a system programming language on the RWC-1 massively parallel machine that supports multi-threads and message driven execution. These primitives are i) a "function instance" which is an abstraction of thread invocation by message passing and ii) "message entry" and "token" which are abstractions of communication among threads. Primitives are realized as language extensions using the metalevel architecture.

MPC++ Version 2.0 is designed in two levels, level 0 and level 1. The level 0 specifies parallel description primitives realized by the C++ template feature without any language extensions, that define the MPC++ basic parallel execution model. The level 1 specifies the MPC++ metalevel architecture and application specific extensions.

In this document, we describe the level 0 parallel description primitives called the MPC++ multiple threads template library (MTTL). It contains i) `invoke` and `ainvoke` function templates for synchronous and asynchronous local/remote thread invocation, ii) `Sync` class template for synchronization and communication among threads, iii) `GlobalPtr` class template for pointer to remote memory, and iv) `yield` function to suspend the thread execution and yield another thread execution.

2 Programming Model

Program, Processes, and Threads: The MPC++ program is assumed to run on a distributed memory based parallel system. Program code is distributed to all physical processors and a process for the program runs on each processor. Each process has several threads of control which are not preempted. That is, thread execution continues unless it waits for synchronization or exits. The detailed timing of thread execution suspension will be described later. A program may locally or remotely invoke a *function instance* which has a thread of control. Invoking a function instance will involve creation of a new thread and the execution of the function. The program invoking the function instance may block until the end of the invoked function instance execution.

Variables: All variables are processor local. The storage defined in the file scope is allocated to each processor. When such a variable is referred to by a processor, the local memory space is accessed. Variables `myPE` and `peNum` are predefined and initialized to represent processor number and number of processors, respectively. The processor number starts from 0 to `peNum` − 1.

Initialization: The `mpc_main` routine is the main routine in MPC++. This routine is only invoked at processor 0 after initializing the file scope variables. Other processors initialize the file scope variables and then waits for messages.

Synchronization and Communication: The synchronization structure called *Sync* is supported to realize the multiple readers/writers communication model. It acts as a FIFO communication buffer. If a reader tries to read data from a *Sync* object but no data has been available, it blocks until a writer writes data to the *Sync* object. If the reader reads data, the data is removed. The reader may also read data without removing the data. If writers write several data, those data are enqueued to the *Sync* object.

Global Pointer: A program may access any remote data via a global pointer as well as invoke a member function instance of any remote object. Several function instances of an object may be invoked. The concurrency control must be programmers' responsibility. We have not defined any parallel object model such as the Actor model. Such a programming construct may be defined by MPC++ level 1, the language extension part.

Execution Order: Order in remote function instance invocation operations on a processor is preserved at processing those operations on the remote processor. Order in remote memory operations using global pointers on a processor is preserved at processing those operations on the remote processor. However, order in a remote function instance invocation and a remote memory operation is not preserved. The programmer may assume that the order in a remote memory operation followed by a remote function instance invocation is guaranteed, but may not assume the opposite order. In other words, a remote memory operation may be handled prior to executing a remote function instance invocation on the remote machine.

2.1 Timing of Thread Execution Suspension

Thread execution may be suspended in the following cases, but the programmers should not write a program assuming those cases:

1. reading a value of `Sync` object
2. invoking a remote function instance synchronously
3. reading a data via a global pointer

If infinite loop is programmed, any other threads nor remote memory operations requested by other processes never get to be proceeded. Function `yield()` is provided to receive incoming message, suspend current thread execution, and switch to another thread.

3 Function Instance

3.1 Synchronous Function Instance Invocation

The `invoke` function template allows us to invoke a function instance locally or remotely and block the thread execution until the return message is received. This function instance invocation involves creation of a new thread and thread context switch. Programmers may invoke a function instance locally in order to yield another thread execution.

The `invoke` function template has two formats, for a function returning a value and for a `void` function. As shown below, the former `invoke` format takes i) a variable where the return value is stored, ii) processor number on which a function instance is invoked, iii) a function name, and iv) its arguments. The later `invoke` takes i) processor number on which a function instance is invoked, ii) a function name, and iii) its arguments.

function-instance-invocation:
  `invoke(` *variable, processor-number, function-name, arguments* `)`
  `invoke(` *processor-number, function-name, arguments* `)`
processor-name:
  integral-value
  intergral-variable
function-name:
  identifier
arguments:
  c++-function-arguments

The following example shows that a `foo` function instance is invoked in the processor 1. The execution of the `mpc_main` thread blocks until `foo` function execution is terminated. After the end of `foo` function execution, the return value is stored in variable `i` and then `mpc_main` thread execution is resumed. A `void` function instance is invoked in the processor 2 in line 9. After the execution of the `bar` function is finished, `mpc_main` thread execution is resumed.

```
 1 #include <mpcxx.h>
 2 int     foo(int, int);
 3 void    bar(int, int);
 4 mpc_main()
 5 {
 6     int         i;
 7
 8     invoke(i, 1, foo, 1, 2);
 9     invoke(2, bar, 10, 20);
10 }
```

3.2 Asynchronous Function Instance Invocation

Asynchronous function instance invocation means that a thread invokes a function instance and executes the subsequent program without blocking. The thread may get the return value later using the synchronization structure. The `ainvoke` function template is provided to program asynchronous function instance invocation.

The `ainvoke` function template has two formats, for a function returning a value and for a `void` function. As shown below, the former `ainvoke` format takes i) a synchronization variable which will be used to get a return value from a function, ii) processor number on which a function instance is invoked, iii) a function name, and iv) its arguments.

The later `ainvoke` format takes i) processor number, ii) a `void` function name, and iii) its arguments. Because the `void` function does not return any value, no synchronization variable is specified.

function-instance-invocation:
`ainvoke(` *sync-var, processor-number, function-name, arguments* `)`
`ainvoke(` *processor-number, function-name, arguments* `)`

In the following example, a `foo` function instance is asynchronously invoked. The return value from the `foo` will be stored in the `ss` synchronization structure. In line 12, the value stored in the `ss` synchronization structure is read and stored into variable `i`. The syntax and semantics of synchronization structure `Sync` is explained in section 4.

In line 10, `void` function `bar` is asynchronously invoked. The invoker and a `bar` function instance are never joined.

```
 1 #include <mpcxx.h>
 2 int     foo(int, int);
 3 void    bar(int, int);
 4 mpc_main()
 5 {
 6     Sync<int>   ss;
 7     int         i;
 8
 9     ainvoke(ss, 1, foo, 1, 2);
10     ainvoke(2, bar, 10, 20);
11
12     i = *ss;
13 }
```

3.3 Restriction

In the current implementation, the number of invoked function's arguments is restricted to eight.

4 Synchronization Structure

The *synchronization structure* described in section 2 is implemented by the `Sync` class template. It takes one type parameter that represents the data type of objects kept in the structure.

sync-var-definition:
`Sync<` *type-name* `>` *variable*

Figure 1 shows that synchronization structure `l1` which keeps integer values is declared in line 13. In line 15, the `count` function with the `l1` is invoked on processor 1. Since line 15 is an asynchronous remote invocation, execution continues. As shown in line 17, the value on `l1` is extracted as if it were a pointer. When execution reaches line 17, the value is extracted from `l1` if `l1` has received a data. Otherwise, execution is suspended until the synchronization structure `l1` receives value.

The `count` function instance receives the synchronization structure `l1` as the parameter `t1`. In lines 5 and 7, processor local variable `times` is incremented and that value is written into the `l1` via `t1` as if `l1` were a pointer.

```
 1 #include <mpcxx.h>
 2 int    times;
 3 void count(Sync<int> t1)
 4 {
 5     *t1 = ++times;
 6     // ...
 7     *t1 = ++times;
 8     // ...
 9 }
10 mpc_main()
11 {
12     int                 i;
13     Sync<int>           l1;
14     // ...
15     ainvoke(1, count, l1);
16     // ...
17     i = *l1;
18     // ...
19     i = *l1;
20 }
```

Figure 1: Sync Example

When the C++ programmers look at Figure 1, they might think that the synchronization object denoted by `t1` of function `count` on the remote machine differs than the `l1` object because the object is copied to the remote machine. This is true in the sense of copying, however, it works because the synchronization object is a sort of a global pointer object that keeps the processor number, where the object is created, and its local address.

4.1 Other Operations

We have shown read/write operations of the synchronization structure. Those are the same as read/write operations of a pointer. Here, we describe other operations: `read`, `write`, and `queueLength`.

4.1.1 Read Operation

The `Sync` object has the `read` method which takes a variable where the value is stored. Line 17 in Figure 1 may be replaced with the following line.

```
l1.read(i);  // equivalent to i = *l1;
```

Since the read pointer operation involves an extra copy operation, it is better to use the `read` method if the value size is large.

4.1.2 Peek Operation

The `peek` method is available to read data without removing it from the `Sync` object. If data is not available on the object, the execution of the caller is blocked until it is available. The `peek` method takes a variable where the value is stored. Program based on a single writer/multiple readers model will use the `peek` method. An example below shows that reader function instances are invoked over processors with passing the `l1` synchronization structure, and then the value is stored in `l1` so that all readers can read the same value.

```
#include <mpcxx.h>
void reader(Sync<int> t1)
{
    int        i;
    t1.peek(i);
    // ...
}
mpc_main()
{
    Sync<int>   l1;
    ainvoke(1, reader, l1);
    ainvoke(2, reader, l1);
    ainvoke(3, reader, l1);
    *l1 = 123;
}
```

4.1.3 Write Operation

The `Sync` object has the `write` method which takes a variable whose value is written in the object. Lines 5 and 7 in Figure 1 may be replaced with the following line.

```
t1.write(++times);  // equivalent to *t1 = ++times;
```

Since the write pointer operation involves an extra copy operation, it is better to use the `write` method if the data size is large.

4.1.4 Queue Length Operation

The `queueLength` method of `Sync` object allows us to know the number of waiting read operations or data received in the object. The positive number specifies the number of received data. The negative number means that the number of threads has been waited to read data. An example is shown below.

```
#include <mpcxx.h>
void worker(Sync<int> done)
{
    // ...
    *done = 1;
}

mpc_main()
{
    Sync<int>   done;
    ainvoke(1, worker, done);
    ainvoke(2, worker, done);
    ainvoke(3, worker, done);
    while (done.queueLength() != 3) {
        // do something
        yield(); // The yield primitive must be issued
                 // to receive incoming messages. See section 7.1.
    }
}
```

5 Global Pointer

Any local object can be referred to using a global pointer which is realized by the `GlobalPtr` class template. The `GlobalPtr` class template takes one type parameter whose storage is pointed to by the global pointer. The operations on an `GlobalPtr` object are almost the same as a regular pointer object except that a global pointer of a global pointer is not allowed.

global-pointer-definition:
    GlobalPtr< *type-name* > *variable*

A simple example is shown below. The `foo` function takes a global pointer as the parameter and save the value `10` into the storage pointed to by the global pointer. The `foo` function instance is invoked in line 14. A local variable `g1` is converted to a global pointer using the cast operation in this line.

```
 1 #include <mpcxx.h>
 2 void foo(GlobalPtr<int> gp)
 3 {
 4     *gp = 10;
 5 }
 6 void bar(GlobalPtr<int> gp)
 7 {
 8     printf("[Processor %d] *gp = %d\n", myPE, *gp);
 9 }
10 mpc_main(int, char**)
11 {
12     int                g1;
13     GlobalPtr<int>     gp;
14     invoke(1, foo, (GlobalPtr<int>) &g1);
15     printf("[Processor %d] g1 is %d\n", myPE, g1);
16     gp = &g1;
17     invoke(2, bar, gp);
18 }
```

When the example is executed, you can see the following message:

```
[Processor 0] g1 is 10
[Processor 1] *gp = 10
```

5.1 Global Pointer to Array

A program below shows an example of other global pointer operations. You will find the most pointer operations.

```
#include <mpcxx.h>
void foo(GlobalPtr<int> gp)
{
    GlobalPtr<int>     t1;
    gp[0] = myPE;
    gp[1] = myPE + 1;
    *(gp + 2) = myPE + 2;
    t1 = gp + 3;
    *t1++ = myPE + 3;
    *t1++ = myPE + 4;
}
mpc_main(int, char**)
{
    int                ga[128];
    invoke(1, foo, (GlobalPtr<int>) ga);
}
```

```
#include <mpcxx.h>
class Mutex {
    Sync<int>   ss;
public:
    Mutex()      { *ss = 1; }
    void enter() { int tmp = *ss; }
    void leave() { *ss = 1; }
};
class Stack {
  Mutex         mm;
  int           sp;
  int           size;
  int           *buf;
  void error()  { printf("out of range %d\n", sp); }
public:
  Stack()       { buf = new int[BUF_SIZE]; size = BUF_SIZE; sp = 0; }
  Stack(int sz) { buf = new int[sz]; size = sz; sp = 0; }
  void push(int v){
      mm.enter();
      if (sp == size) error();
      else buf[sp++] = v;
      mm.leave(); }
  int pop()     {
      int       val;
      mm.enter();
      if (sp == 0) error();
      else val = buf[--sp];
      mm.leave();
      return val; }
};
```

Figure 2: Classes Mutex and Stack

5.2 Global Pointer to Global Pointer

Unlike regular pointer operations, a global pointer is not allowed to refer to a global pointer. That is, line 5 of the following example is an invalid expression. If the users want to refer to a global pointer of a global pointer, the user must write two steps as shown in lines 6 and 7.

```
 1 #include <mpcxx.h>
 2 void foo(GlobalPtr<GlobalPtr<int> > ggp)
 3 {
 4     GlobalPtr<int>      gp;
 5     // **ggp = myPE;    /* error */
 6     gp = *ggp;
 7     *gp = myPE;
 8 }
 9
10 mpc_main(int, char**)
11 {
12     int                          i1;
13     GlobalPtr<int>               gp;
14     GlobalPtr<GlobalPtr<int> >   ggp;
15
16     gp = &i1;
17     ggp = &gp;
18     invoke(1, foo, ggp);
19
20     printf("i1 = %d\n", i1);
21 }
```

5.3 Global Object Invocation

The programmers may write a remote method invocation of the object using a global pointer to an object. The `invoke` and `ainvoke` function templates are provided.

remote-object-method-synchronous-invocation:
  `invoke(` *variable, global-pointer, class-name* `::` *member-function-name, arguments* `)`
  `invoke(` *global-pinter, class-name* `::` *member-function-name, arguments* `)`
remote-object-method-asynchronous-invocation:
  `ainvoke(` *sync-var, global-pointer, class-name* `::` *member-function-name, arguments* `)`
  `ainvoke(` *global-pointer, class-name* `::` *member-function-name, arguments* `)`

The arguments of the `invoke` template for a member function which returns a value are i) a variable name, where the return value is stored, ii) a global pointer, iii) member function name, and iv) arguments for the function. The arguments of the `invoke` template for a void member function are i) a global pointer, ii) member function name, and iii) arguments of the function. The `invoke` function waits for finishing the execution of the member function.

The arguments of the `ainvoke` template for a member function which returns a value are i) a synchronization structure variable name, where the return value is stored, ii) a global pointer, iii) member function name, and iv) arguments for the function. Using `ainvoke`, a thread invokes a function instance and executes the subsequent program without blocking. The thread may get the return value later using the synchronization structure.

The arguments of `ainvoke` template for a void member function are i) a global pointer, ii) member function name, and iii) arguments of the function. Because the `void` function does not return any value, no synchronization variable is specified.

An example is shown below. Function `allocateStack` is defined to create an object in a remote processor. `mpc_main`, running processor 0, invokes the `allocateStack` routine in processor 1 in line 14. The global pointer `gsp` points to the remote Stack object created in processor 1.

Using the `gsp` global pointer, the member function `push` of the `Stack` object, which is a void function, is synchronously invoked in line 15. In line 16, the member function `pop` of the `Stack` object, which returns an integer value, is synchronously invoked.

Lines 18 and 19 are an example of asynchronous void member function and integer value returned member function invocations. Without waiting for finishing the member function execution, the subsequent program is executed. It should be raised again that order in synchronous/asynchronous invocation requests to a remote processor from the same processor are preserved at processing in the remote processor.

```
#include <mpcxx.h>
GlobalPtr<Stack>
allocateStack()
{
    return (GlobalPtr<Stack>) new Stack();
}

mpc_main(int, char**)
{
    GlobalPtr<Stack>    gsp;
    int                 i;
    Sync<int>           si;

    invoke(gsp, 1, allocateStack);
    invoke(gsp, Stack::push, 1);
    invoke(i, gsp, Stack::pop);
    printf("i = %d\n", i);
    ainvoke(gsp, Stack::push, 2);
    ainvoke(si, gsp, Stack::pop);
    i = *si;
    printf("i = %d\n", i);
}
```

We have programmed the remote object creation function here. In fact, however, the users do not program such a function. As described in section 7.2, Function templates `gallocate` and `gfree` are available to create and free a remote object.

5.4 Remote Memory Address

As described in section 2, the storage defined in the file scope is allocated to each processor. That is, the local address is the same over the processors. The `set` method of the `GlobalPtr` object is available to specify the remote memory address defined in the file scope using the storage name.

The following example shows that the `mpc_main` routine sets 3.1415 to the `dt` variable on each processor.

```
#include <mpcxx.h>
double  dt;
mpc_main(int, char**)
{
    GlobalPtr<double>   gdp;
    int                 i;

    for (int i = 0; i < peNum; i++) {
        gdp.set(&dt, i);
        *gdp = 3.1415;
    }
    // ...
}
```

5.5 Variable Length Remote Memory Operations

The `GlobalPtr` object has the `nread` and `nwrite` methods to read and write a remote memory, respectively. The `nread` method takes i) a local memory address, where the remote memory pointed to by the `GlobalPtr` object is stored, ii) size, and iii) a synchronization structure which is used to wait for operation done. The `nwrite` method writes data given by the first parameter into the remote memory pointed to by the `GlobalPtr` object. The second parameter specifies the memory size.

An example is shown as follows:

```
#include <mpcxx.h>
double  p[256];
double  q[256];

mpc_main(int, char**)
{
    GlobalPtr<double>   gdp;
    Sync<int>           done;
    int                 i;

    /* q[256] on PE#0 <== p[256] on PE#1 */
    gdp.set(p, 1);
    gdp.nread(q, 256, done);
    done.read(i);
    /* p[256] on PE#0 ==> q[256] on PE#1 */
    gdp.set(q, 1);
    gdp.nwrite(p, 256);
}
```

Method `mnwrite` of a `GlobalPtr` object is a multicast remote memory write function to distribute a data to other processors. In line 13 of the example below, The value of the double array `p` in processor 0 is copied to the double array `q` on processors 1 through `peNum - 1`. The second argument specifies the data size, number of elements. The third argument must be an integer array whose element specifies a processor number, and the last argument is its array size.

```
 1 #include <mpcxx.h>
 2 double  q[256];
 3 double  p[256];
 4 mpc_main(int, char**)
 5 {
 6     GlobalPtr<double>   gdp;
 7     int                 dest[64];
 8
 9     for (int i = 1; i < peNum; i++) {
10         dest[i] = i;
11     }
12     gdp.set(q, 0);
13     gdp.mnwrite(p, 256, dest, peNum - 1);
14 }
```

6 Global Synchronization

Facilities barrier synchronization and reduction are presented in this section.

6.1 Barrier

Class `Barrier` realizes a barrier synchronization mechanism among threads on processors. A `Barrier` object must be a file scope object and is initialized by the `setall` method which takes two arguments, processor number and number of processors. The number of processors must be a power of 2.

Threads on processors starting from the specified processor number are synchronized by issuing the `exec` method of the `Barrier` object. That method must be only invoked by single thread on each processor during the barrier synchronization. An example is shown below.

Table 1: Reduction Methods on `Reduction` and `ReductionArray` Class Template

Method	Meaning
`sum`	Sum of values
`and`	Bitwise *and* of values
`xor`	Bitwise *xor* of values
`or`	Bitwise *or* of values
`max`	Maximum of values
`min`	Minimum of values

```
#include <mpcxx.h>
#include <stdio.h>
#include <Barrier.h>
Barrier         barrier;
void
pe_main(Sync<int> done)
{
    // ...
    barrier.exec();
    // ...
    if (myPE == 1) *done = 1;
}
mpc_main(int, char*)
{
    int         i;
    Sync<int>   done;
    barrier.setall(1, peNum - 1);
    for (i = 1; i < peNum; i++) {
        ainvoke(i, pe_main, done);
    }
    done.read(i);
}
```

6.2 Reduction

Class templates `Reduction` and `ReductionArray` realize reduction operations among threads on processors. `Reduction` is used to reduce values to a single value while `ReductionArray` is used to reduce values of an array to a single value of an array. A reduction operator listed in Table 1 is applied at the reduction.

An object of `Reduction` and `ReductionArray` must be a file scope object. As shown below, type of the value is specified at an object declaration. The `Reduction` object `red1` is declared to reduce integer values and `ReductionArray` object `red2` is declared to reduce integer array values whose size is `A_SIZE` in lines 4 and 5, respectively.

An object of `Reduction` and `ReductionArray` must be initialized by the `setall` method which takes two arguments, processor number and number of processors. The number of processors must be a power of 2. Lines 23 and 24 are examples. Expression `peNum - 1` must be a power of 2.

Function `pe_main` is invoked on processors 1 through `peNum - 1`. Each function instance calls the `sum` method of object `red1` in line 12 to add all values of `myPE` on processors and the result is stored in `val`. After calling `sum` method of object `red2`, values of integer array `dt` are results of adding all values of the `dt` on processors.

```
#include <mpcxx.h>
#include <stdio.h>
#include <Reduction.h>
Reduction<int>                  red1;
ReductionArray<int, A_SIZE>     red2;
int     dt[A_SIZE];
void
pe_main(Sync<int> done)
{
    int         val;
    // ...
    val = red1.sum(myPE);
    // ...
    red2.sum(dt);
    // ...
    if (myPE == 1) *done = 1;
}
mpc_main(int, char**)
{
    Sync<int>           done;
    int                 i;

    red1.setall(1, peNum - 1);
    red2.setall(1, peNum - 1);
    for (i = 1; i < peNum; i++) {
        ainvoke(i, pe_main, done);
    }
    done.read(i);
}
```

7 Other Functions

7.1 Thread Execution Control

Function `yield()` is provided to suspend the current thread execution and switch to another thread. If no threads are ready to run, the current thread continues to run.

Function `yield()` will deal with receiving remote thread invocations and remote memory operations requested by other processes.

7.2 Remote Object Allocation/Free

Template Functions `gallocate` and `gfree` are available to allocate and free a remote object. The arguments of `gallocate` are a global pointer, where the remote object address is stored, processor number, and arguments for the constructor. `gfree` takes a global pointer as an argument.

The global object invocation example in section 5.3 has defined the `allocateStack` function to allocate a Stack object in a remote processor. Using `gallocate`, we do not need to implement such a function. The following example shows that a remote Stack object is created by invoking constructor `Stack()` and its global pointer is stored in `gsp1` in line 2. Line 3 shows that a remote Stack object is created by invoking constructor `Stack(int)` and its global pointer is stored int `gsp2`.

```
1 GlobalPtr<Stack>        gsp1, gsp2;
2 gallocate(gsp1, 1);     // will invoke Stack::Stack()
3 gallocate(gsp2, 1, 128);// will invoke Stack::Stack(int)
```

8 Current Implementation Status

MPC++ Version 2.0 level 0 has been operational on our workstation cluster consisting of 36 Sun SS20's using Myricom Myrinet[1]. According to our experience on NAS Parallel Benchmark kernel CG, MPC++ Version 2.0 on the workstation cluster achieves almost the same performance of Cray T3-D.

Readers, who are interested in our current software environment, can visit our WEB home page for more information: "`http://www.rwcp.or.jp/lab/mpslab/`"

References

[1] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W. Su. Myrinet - A Gigabit-per-Second Local-Area Network. *IEEE Micro*, Vol. 15, No. 1, pp. 29–36, February 1995.

[2] Yutaka Ishikawa. The MPC++ Programming Language V1.0 Specification with Commentary Document Version 0.1. Technical Report TR–94014, RWC, June 1994.

[3] Yutaka Ishikawa and et.al. MPC++. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming Using C++*, pp. 427–466. MIT Press, 1996. To be published in 1996 Spring.

A Library Summary

A.1 `invoke` Function Template

A.1.1 Function returning a value

The following template specifications tell you how to write a function instance synchronous invocation which returns a value of type `F`.

```
template <class F>
            int invoke(F &, int pe, F (*f)());
template <class F, class A1>
            int invoke(F &, int pe, F (*f)(A1));
template <class F, class A1, class A2>
            int invoke(F &, int pe, F (*f)(A1, A2));
template <class F, class A1, class A2, class A3>
            int invoke(F &, int pe, F (*f)(A1, A2, A3));
template <class F, class A1, class A2, class A3, class A4>
            int invoke(F &, int pe, F (*f)(A1, A2, A3, A4));
template <class F, class A1, class A2, class A3, class A4, class A5>
            int invoke(F &, int pe, F (*f)(A1, A2, A3, A4, A5));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(F &, int pe, F (*f)(A1, A2, A3, A4, A5, A6));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(F &, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(F &, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7, A8));
```

A.1.2 Void Function

The following template functions allow the programmers to invoke a void function instance synchronously.

```
int invoke(int pe, void (*f)());
template <class A1>
            int invoke(int pe, void (*f)(A1));
template <class A1, class A2>
            int invoke(int pe, void (*f)(A1, A2));
template <class A1, class A2, class A3>
            int invoke(int pe, void (*f)(A1, A2, A3));
template <class A1, class A2, class A3, class A4>
            int invoke(int pe, void (*f)(A1, A2, A3, A4));
template <class A1, class A2, class A3, class A4, class A5>
            int invoke(int pe, void (*f)(A1, A2, A3, A4, A5));
template <class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6));
template <class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6, A7));
template <class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6, A7, A8));
```

A.1.3 Global Object Member Function returning a value

The following template specifications tell you how to write a member function instance synchronous invocation which returns a value of type `F` via a global pointer `GlobalPtr<T>`.

```
template <class T, class F>
            int invoke(F &, GlobalPtr<T>, F (T::*f)());
template <class T, class F, class A1>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1));
template <class T, class F, class A1, class A2>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2));
template <class T, class F, class A1, class A2, class A3>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3));
template <class T, class F, class A1, class A2, class A3, class A4>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4));
template <class T, class F, class A1, class A2, class A3, class A4, class A5>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5, A6));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5, A6, A7));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(F &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5, A6, A7,
                                                    A8));
```

A.1.4 Global Object Void Member Function

The following template functions allow the programmers to invoke a void member function instance synchronously via a global pointer `Global<T>`.

```
template <class T>
            int invoke(GlobalPtr<T>, void (T::*f)());
template <class T, class A1>
            int invoke(GlobalPtr<T>, void (T::*f)(A1));
template <class T, class A1, class A2>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2));
template <class T, class A1, class A2, class A3>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3));
template <class T, class A1, class A2, class A3, class A4>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4));
template <class T, class A1, class A2, class A3, class A4, class A5>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6, A7));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6, A7,
                                                  A8));
```

A.2 `ainvoke` Function Template

A.2.1 Function returning a value

The following template specifications tell you the usage of a asynchronous function instance invocation which returns a value of type `F`.

```
template <class F>
            int ainvoke(Sync<F> &, int pe, F (*f)());
template <class F, class A1>
            int invoke(Sync<F> &, int pe, F (*f)(A1));
template <class F, class A1, class A2>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2));
template <class F, class A1, class A2, class A3>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3));
template <class F, class A1, class A2, class A3, class A4>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3, A4));
template <class F, class A1, class A2, class A3, class A4, class A5>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3, A4, A5));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3, A4, A5, A6));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(Sync<F> &, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7,
                                                 A8));
```

If the return value from a function instance, which has been invoked asynchronously, is not needed, the following function template should be used. That is, instead of specifying a `Sync` object, a `NullSync` object is specified.

```
template <class F>
            int ainvoke(NullSync, int pe, F (*f)());
template <class F, class A1>
            int invoke(NullSync, int pe, F (*f)(A1));
template <class F, class A1, class A2>
            int invoke(NullSync, int pe, F (*f)(A1, A2));
template <class F, class A1, class A2, class A3>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3));
template <class F, class A1, class A2, class A3, class A4>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3, A4));
template <class F, class A1, class A2, class A3, class A4, class A5>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3, A4, A5));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3, A4, A5, A6));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7));
template <class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(NullSync, int pe, F (*f)(A1, A2, A3, A4, A5, A6, A7,
                                                A8));
```

A.2.2 Void Function

The following template functions allow the programmers to invoke a void function instance asynchronously.

```
int ainvoke(int pe, void (*f)());
template <class A1>
            int ainvoke(int pe, void (*f)(A1));
template <class A1, class A2>
            int ainvoke(int pe, void (*f)(A1, A2));
template <class A1, class A2, class A3>
            int ainvoke(int pe, void (*f)(A1, A2, A3));
template <class A1, class A2, class A3, class A4>
            int ainvoke(int pe, void (*f)(A1, A2, A3, A4));
template <class A1, class A2, class A3, class A4, class A5>
            int ainvoke(int pe, void (*f)(A1, A2, A3, A4, A5));
template <class A1, class A2, class A3, class A4, class A5,
          class A6>
            int ainvoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6));
template <class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int ainvoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6, A7));
template <class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int ainvoke(int pe, void (*f)(A1, A2, A3, A4, A5, A6, A7, A8));
```

A.2.3 Global Object Member Function returning a value

The following template specifications tell you how to write a member function instance asynchronous invocation which returns a value of type `F` via a global pointer `GlobalPtr<T>`.

```
template <class T, class F>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)());
template <class T, class F, class A1>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1));
template <class T, class F, class A1, class A2>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2));
template <class T, class F, class A1, class A2, class A3>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3));
template <class T, class F, class A1, class A2, class A3, class A4>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4));
template <class T, class F, class A1, class A2, class A3, class A4, class A5>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                            A6));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int ainvoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                            A6, A7));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int invoke(Sync<F> &, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                           A6, A7, A8));
```

If the return value from a member function instance, which has been invoked asynchronously, is not needed, the following function template should be used. That is, instead of specifying a `Sync` object, a `NullSync` object is specified.

```
template <class T, class F>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)());
template <class T, class F, class A1>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1));
template <class T, class F, class A1, class A2>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2));
template <class T, class F, class A1, class A2, class A3>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3));
template <class T, class F, class A1, class A2, class A3, class A4>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4));
template <class T, class F, class A1, class A2, class A3, class A4, class A5>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                              A6));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                              A6, A7));
template <class T, class F, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int ainvoke(NullSync, GlobalPtr<T>, F (T::*f)(A1, A2, A3, A4, A5,
                                                              A6, A7, A8));
```

A.2.4 Global Object Void Member Function

The following template functions allow the programmers to invoke a void member function instance asynchronously via a global pointer `Global<T>`.

```
template <class T>
            int ainvoke(GlobalPtr<T>, void (T::*f)());
template <class T, class A1>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1));
template <class T, class A1, class A2>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2));
template <class T, class A1, class A2, class A3>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3));
template <class T, class A1, class A2, class A3, class A4>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4));
template <class T, class A1, class A2, class A3, class A4, class A5>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6, A7));
template <class T, class A1, class A2, class A3, class A4, class A5,
          class A6, class A7, class A8>
            int ainvoke(GlobalPtr<T>, void (T::*f)(A1, A2, A3, A4, A5, A6, A7, A8));
```

A.3 `Sync` Class Template

In addition to the pointer read/write operation on a `Sync` object, the following methods are available:

```
template<class T> class Sync {
public:
        operator        T();    /* casting */
        void            read(T&);
        void            peek(T&);
        void            write(T&);
        int             queueLength();
};
```

`read(T&)` A data is extracted from the `Sync` object and stored into the variable specified in the argument.

`peek(T&)` A data is read from the `Sync` object and stored into the variable specified in the argument. The data still remains in the `Sync` object after the operation.

`write(T&)` The value specified in the argument is enqueued into the `Sync` object.

`queueLength()` The number of current available values are returned.

A.4 `GlobalPtr` Class Template

In addition to the pointer read/write and increment/decrement operations of a `GlobalPtr` object, the following methods are available:

```
template<class T> class GlobalPtr {
public:
        void            set(void *laddr, int pe);
        void            set(void *laddr);
        int             getPe();
        void            *getLaddr();
        void            nwrite(T *laddr, int nitem);
        void            nread(T *laddr, int nitem, Sync<int> sync);
        void            mnwrite(T *laddr, int nitem, int dest[], int dsize);
};
```

`void set(void *laddr, int pe)` After the execution of the method, the `GlobalPtr` object points to an object whose memory address is pointed to by `laddr` on a processor `pe`.

`void set(void *laddr)` It changes the local memory address of the `GlobalPtr` object.

`int getPe()` It returns the processor number of an object pointed to by the `GlobalPtr` object.

`void *getLaddr()` It returns the local address of an object pointed to by the `GlobalPtr` object.

`void nwrite(T *laddr, int nitem)` It writes data pointed to by `laddr` into the remote memory represented by the `GlobalPtr` object. The data size, number of items, is specified by the second argument.

`void nread(T *laddr, int nitem, Sync<int> sync)` It reads data represented by the `GlobalPtr` object and stores the data into memory pointed to by `laddr`. The data size, number of items, is specified by the second argument.

`void mnwrite(T *laddr, int nitem, int dest[], int dsize)` It writes data pointed to by `laddr` into all remote memory, whose local address is represented by the `GlobalPtr` object, of processors specified in the third argument, `dest` array each element of which represents processor number. The data size, number of items, is specified by the second argument, `nitem`. The array size of the third argument is specified by the last argument, `dsize`.

A.5 Barrier and Reduction Classes

A.6 Others

Function `yield()` is to suspend the current thread execution and switch to another thread.

```
void yield();
```

The `gallocate` function template is to create an object on a processor specified by the second argument. The object class is resolved using the first argument, an `GlobalPtr` object specifying the class.

```
template<class T>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe);
template<class T, class A1>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1);
template<class T, class A1, class A2>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2);
template<class T, class A1, class A2, class A3>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3);
template<class T, class A1, class A2, class A3, class A4>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3, A4);
template<class T, class A1, class A2, class A3, class A4, class A5>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3, A4, A5);
template<class T, class A1, class A2, class A3, class A4, class A5,
         class A6>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3, A4, A5, A6);
template<class T, class A1, class A2, class A3, class A4, class A5,
         class A6, class A7>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3, A4, A5, A6,
                               A7);
template<class T, class A1, class A2, class A3, class A4, class A5,
         class A6, class A7, class A8>
        GlobalPtr<T> gallocate(GlobalPtr<T>&, int pe, A1, A2, A3, A4, A5, A6,
                               A7, A8);
```

The `gfree` function template is to free an object pointed to by a `GlobalPtr` object:

```
template<class T>
        void gfree(GlobalPtr<T> &);
```