Abstract

This paper describes the current version of mpi++, a C++ language binding for MPI, that includes all of the collective services, and services for contexts, groups and communicators as described in Chapter 4 and 5 of the MPI standard. The code for mpi++ has been tested on a Sun Sparc workstation and an Intel Paragon. Segments of a mpi++ program implementing a parallel algorithm is introduced to illustrate the Collective class hierarchy. The paper also shows how mpi++ deals with other collective operations (e.g., reduction), attribute caching, groups, and communicators. The class hierarchy of mpi++ is presented and briefly explained.

1. Introduction

The Message Passing Interface (MPI) standard provides a common interface for message-based communication and synchronization among processes executing on distributed memory parallel computers and networks of workstations [7]. MPI functionality includes point-to-point and collective communication routines, support for process groups, communication contexts, and application topologies. For application developers to use the MPI services, currently C and FORTRAN are the two available language bindings. Several groups are developing C++ bindings.

The design goal of mpi++ is to provide a language binding that presents the semantic and conceptual model of MPI in a way that is both easily recognizable to those familiar with MPI, though perhaps not with C++, and recognizable as "good" C++ design to those who are familiar with C++, but not necessarily with MPI. Being recognizable to those familiar with MPI means that the concepts and names defined in the MPI standard are presented. Being recognizable as good object-oriented design means that mpi++ will employ as appropriate the full range of object-oriented features including operator overloading, templates, and inheritance. Mpi++ is a layer above an existing C implementation, rather than an implementation of the MPI standard in C++.

There are other proposed C++ interfaces to MPI.

- A "flat" model of MPI classes is presented in Chapter 7 of the MPI standard (mpi-bind@mcs.anl.gov, March 18, 1996). It provides lightweight interface classes, introduces a minimal use of advanced features of C++.
- OOMPI – Object Oriented MPI – provides a C++ class library that is not a simple one-to-one mapping of MPI function to language binding, and may add, lose, or change MPI-1 specified functionality [6].
- "MPI++" [1] is a portable C++ interface to MPI that can be used to build numerical libraries to support object-oriented programming. It provides a parallel programming model without the benefit of compiler support.
- PARA++ [2] is a generic high-level interface for performing message passing in C++ based on an idea of Stream Message Passing. Its library was originally defined for a domain decomposition method to solve the laplace operator with spectral methods.

The version of mpi++ described in this paper includes all of the MPI services for collective communication as described in Chapter 4 of the MPI standard, and also the services for groups, communicators, and attribute caching (Chapter 5). An earlier paper about mpi++ [4] introduced the point to point communication services defined in Chapter 3 of the MPI standard. Subsequent versions of mpi++ will incorporate additional MPI services, such as process topologies, environmental management, and profiling interface. The mpi++ code for the current version of MPI has been tested on a Sun workstation and an Intel Paragon using the Sun CC and Intel icc compiler, respectively.

The remainder of this paper is organized as follows. In Section 2, classes in mpi++ that provide the collective communications are discussed. Section 3 briefly introduces the
classes in mpi++ that capture the group, communicator, and attribute caching services. A brief summary is given in Section 4. Figure 1 shows the mpi++ class hierarchy of Collective class, Communicator, Group, and Attribute classes.

```
Collective<SendType, RecvType>

Reduction<SendType, RecvType>

Communicator

Attribute *operator[](Attribute &a);
void operator=(Attribute &a);
void Delete(Attribute &a);

InterCommunicator

Group

MPIComm

Figure 1. Class Hierarchy of Collective Communication and Communicators.
```

2. Collective Communications

This section introduces the mpi++ classes that capture the collective message passing services of MPI. An example of computation for a queueing system (a G/G/l queue) is described to illustrate the Collective class hierarchy.

2.1. Overview of Collective Communication Classes

Collective communication involves a group of processes [7]. The functions can be categorized as:

- Communication and synchronization operations, e.g., barrier, broadcast, gather, scatter. Such functions provide collection communication services across a group of processes. All processes in the group call the communication routines with matching arguments.

- Global reduction operations, e.g., sum, max, min, or user-defined functions. Such functions provide collection communication services that involve arithmetic or logical computation. The reduction operation can be either one of a predefined list of operations, or a user-defined operation.

- Scan operations, i.e., prefix operations. They perform prefix reductions on data distributed across the group. They return, in the receive buffer of the process with rank i, the reduction of the values in the send buffers of processes with ranks 0, 1, ..., i inclusively.

In mpi++ there are two major class templates for the collective communication: Collective and Reduction. The class template Reduction is derived from the class template Collective, for the reduction operations require collective communication during the reduction computation. All the attributes in Collective classes can be inherited by Reduction classes. Parameters needed by reduction functions have the same pattern as in Collective classes.

The mpi++ Collective communication classes are illustrated throughout this section by showing the GLM algorithm [8] that simulates a G/G/l queueing model [3]. Two parameters, lambda and mu, characterise the mean interarrival and service times, respectively. The following is an outline of the simulation:

1. process generates both the interarrival and service time by using lambda and mu, and computes relative (local) arrival and service times.
2. processes compute the global arrival and departure times by global reduction operations.
3. arrival and departure times are merged into one sequence. Exchanging elements of the time sequences between neighboring processes is necessary during the parallel merge.
4. queue length and its mean are computed by both local computation and global reduction operations.

From the above description we observe that Collective class objects can be used to accomplish some of the steps of this parallel computation, for example, in the first step the root node reads the value of lambda and mu, and broadcasts them to the rest of nodes in the group as follows:

```c
1  Collective<DOUBLE, DOUBLE> coll;
2  coll.Broadcast(&lambda);
3  coll.Broadcast(&mu);
4  coll.barrier();
```

where coll is a Collective class instance with double data type for both send and receive, since lambda and mu are real numbers. In contrast, the following is a segment of the code using MPI routines to accomplish the same function:

```
MPI_Bcast(&lambda, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Bcast(&mu, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
```

In the MPI version, some parameters are repeated twice in the four communication operations: the number of data elements (the constant 1), the MPI data type (MPI_DOUBLE), and the root class (the constant 0). The communicator parameter (MPI_COMM_WORLD) is repeated in all the three communication operations. Such
redundancy is avoided in the mpi++ design. Similar to
the mpi++ design of classes for point-to-point communications, four out of five parameters do not need to be passed to
the broadcast operations of Collective classes.

The parameters of the collective operation are provided
as template arguments, constructor arguments or method
arguments. The send and receive data types are provided
via the two template arguments when a Collective class is
instantiated (line 1). The other two, the root process id.
and the communicator, less likely to be changed at runtime,
are passed as constructor parameters of the class object(line
1). Both the id and communicator arguments of the object
constructor for Collective class have default values. In the
above program segment, there is no parameter list following
class object col1 in its declaration statement (line 1), for
default values are used. This greatly simplifies the param-
eter list in the collective operations. The only information
required is that which is the most frequently changed, in this
case the data to be broadcast (lines 2 and 3).

2.2. Collective Class

The Collective class template listed below provides all
the collective functions for the MPI standard, e.g., broad-
cast, gather, scatter. It has two template arguments, Send-
Type and RecvType, that provide data type information for
both the data to be sent and that to be received, respectively.

Information conveyed by the template arguments, such as
the underlying MPI data types and the number of data el-
ements, are retrieved while an instance of Collective class
is constructed. Information for the id number of the root
process and the communicator of the process group, are ob-
tained through the constructor arguments of a Collective
class instance. Thus, the only information that needs to be
passed when a collective operation executes is a buffer for
the data to be sent. The usage of default values for the mem-
ber function arguments greatly simplifies the parameter list
of member functions of the Collective class template.

The types of template arguments SendType and Recv-
Type are classes of TypeName or its derived classes, e.g.,
class instance of TypeConstruct template. The TypeName
classes and its descendents are designed to keep information
necessary for the Collective classes. The definition of Type-
Name class and its subclasses can be found in [4].

```cpp
1 template<class SendType, class RecvType>
2 class Collective {
3 protected:
4  typedef SendType::Type CsendType;
5  typedef RecvType::Type CrecvType;
6  CrecvType *recvData;
7  int recvCount;
8  Communicator comm;
9  int root, rank, Error;
10 public:
11  Collective(int r=0, Communicator c=
12    Communicator());
13  ~Collective();
14  int barrier();
15  CsendType *Broadcast
16    (CsendType *buf, int count=1);
17  CrecvType *Gather
18    (CsendType *buf, int count=1);
19  CrecvType *GatherV(CsendType *buf);
20  CrecvType *Scatter(CsendType *buf);
21  ...
```

The type definition statements (lines 4 and 5) extract
the original C data types of send and receive data from the
template arguments CsendType and CrecvType. The func-
tion in line 14 encapsulates the MPI_Barrier() operation that
provides barrier synchronization across all group members.
The function in line 15 broadcasts data from the “root” pro-
cess to all group members. Gather() collects data from all
members of a group to the “root”. The following is the func-
tion definition of Gather():

```cpp
8 a CrecvType *Gather(CsendType *buf, int count=1)
9 b (...)
10 c Error=MPI_Gather(buf, count,
11 d SendType::type_of(), recvData, count,
12 e SendType::type_of(), root, comm.MPI_comm());
13 f if (mpi::rank()==root) return recvData;
14 g else return (CrecvType *)0;
```

The parameter, buf, in line c contains the data to be sent.
the buffer contains the number of send data elements, that is
computed in the construction of a Collective class based on the
information of template argument SendType. In this case,
count equals the number of received data elements, so it
is also referenced as the parameter of received data number.
Root refers to the process that receives data from all group
members. Comm.MPIcomm() returns the MPI communic-
ator. The static member function SendType::type.of() of
Collective template argument SendType provides the MPI
data type name, that is referenced both as the MPI send and
receive data type names in Gather(). Finally, the received
data is returned to the “root” process.

2.3. Reduction Class

Reduction is a class template derived from Collective.
The reduction template encapsulates the reduction opera-
tions in MPI standard, for example, Reduce(), ReduceScat-
tter(), Scan(), etc. Each reduction computation involves an
operation that is performed at each step in the reduction.
There are two ways to define the reduction operation, either
by MPI default operators, or by user-defined MPI operators.
In the former case, a Reduction template class can be created
directly; in the latter case, a subclass of Reduction is defined.
The default MPI operators includes MPL_MAX, MPL_MIN,
MPL_SUM, MPL_MAXLOC, MPL_MINLOC, etc. Figure 2
illustrates how to define a Reduction template class with a user-defined operation by subclassing.

There are two different constructors for the Reduction template. When users define their own reduction operations, the first constructor in lines g and h sets the static function User_Func(), that instantiates the user-defined function user_func(), as a MPI operation to create a user defined reduction operator. In this constructor, the commute property of the reduction operation is specified through the parameter commute. If a default reduction operator is used, the second constructor in lines i and j is applied. The default operator is passed through the parameter op. The function Reduce() initiates a global reduction operation, either predefined or user-defined functions, where the result is returned only to the "root" process. Scan() performs a prefix operation across all members of a group.

Figure 2. Subclassing of Reduction Class

```
template <class SendType, class RecvType>
class Reduction : public Collective<SendType,RecvType>
{
    int user_defined;
    MPI-Op Op;
    static Reduction<SendType, RecvType> *mythis;

    public:
    Reduction(int commute=1, int r=0, Communicator c=Communicator());
    Reduction(MPI-Op op, int r=0, Communicator c=Communicator());
    virtual void user_func(CsendType *invec, CrecvType *inoutvec, int *len);
    static void User-Func(CsendType *invec, CrecvType *inoutvec, int *len, MPI-Datatype *DataType);
    CrecvType *Reduce(CsendType *sendbuf, int sendcount=1);
    CrecvType *Scan(CsendType *sendbuf, int sendcount=1);
    ...
};
```

In the queueing system computation, each process performs a cost optimal parallel prefix sum operation on the arrival time of events. This part of the algorithm involves two major steps. At first, each process computes locally the arrival time of the set of events assigned to it. At this point, the time sequence is relative, each process then participates in a global prefix sum (scan) operation contributing the relative arrival time of its last event. Each process then computes the global arrival time sequence using the result from the scan computation. The following code creates an instance prefix Sum of Reduction class template, and computes the prefix sum using a Scan operation:

```
#define num-jobs 50000 // # of jobs
double A[num-jobs]; // Arrival time
double s;

// Compute local prefix sum
// Compute global prefix sum
Reduction<DOUBLE,DOUBLE> prefixSum(MPI-SUM);
s = *(prefixSum.Scan(&A[num-jobs-1])) - A[num-jobs-1];
```

In line 6, a class object prefixSum is declared whose send and receive data types are both double numbers. Because this is a simple prefix sum reduction operation, the default MPI reduction operator MPLSUM is passed to the constructor of prefixSum. Then the reduction operation is performed (lines 7 and 8). Note the default parameter values of root class and communicator are used, since in this case the default values are the ones needed. Only the address of A[num-jobs-1] carrying the send data is passed to the Scan() operation. The result of Scan() operation is subtracted by A[num-jobs-1] and stored in s, yielding the value to compute global arrival time in next step (lines 7 and 8).

A user-defined reduction operator is needed to perform the prefix sum of departure times that is more complicated compared to that for the arrival time A because the computation depends upon the arrival time sequence A. The departure time sequence, D, is a double type of array of num-jobs elements. A data structure Dmatrix of 4 double numbers represents one element of D. The computation of local departure time for each event is an operation between two 2x2 matrices of type Dmatrix [3]. The following algorithm computes the local prefix sum of departure time sequence D. Note it is not exactly the same algorithm as in [3], rather it is a modified version merely for readability, although it returns the same result as the original algorithm.

For proc 1, \[ D(1) \begin{bmatrix} 0 \\ -\infty \end{bmatrix} op \begin{bmatrix} \text{delta}(2) \\ A(2)+\text{delta}(2) \end{bmatrix} \begin{bmatrix} -\infty \\ 0 \end{bmatrix} \text{op...op} \begin{bmatrix} \text{delta}(\text{num-jobs}) \\ A(\text{num-jobs}) \end{bmatrix} \begin{bmatrix} -\infty \\ 0 \end{bmatrix} \] where \(D(1)=A(1)+\text{delta}(1)\); for the rest of processors,

\[ \begin{bmatrix} \text{delta}(1) \\ A(1)+\text{delta}(1) \end{bmatrix} \text{op...op} \begin{bmatrix} \text{delta}(\text{num-jobs}) \\ A(\text{num-jobs}) \end{bmatrix} \begin{bmatrix} -\infty \\ 0 \end{bmatrix} \]

\(\text{delta}\) is the random job service time sequence, and \(op\) is defined as \(\begin{bmatrix} a & b \\ c & d \end{bmatrix} \text{op} \begin{bmatrix} c \\ d \end{bmatrix} = \text{max}(a+c, b+d)\).

In the following code segment, DepartureScanType is a class instance of template Contiguous (line 16), where
the data type for each element is a struct type, defined by
DmatrixType (lines 15), a class instance of template Struct.
There are two elements of DmatrixType in DepartureScanT-
type (lines 15-16). Because the member function user_func() op-
operates on two elements of Dmatrix (line 21), such a data
type definition is necessary for the computation of global
prefix sum for departure time. Lines 17 to 26 declare a user-
defined function user_func() in the definition of a class de-
ferred from the Reduction class template. This function per-
forms the global prefix sum of departure time.

1 class DmatrixType_Parameter {
2 public:
3 static int count;
4 ...
5 static MPI_Datatype typeArray;
6 static void type_array();
7 (typeArray[0]=typeArray[1]=
8 =DOUBLE::type_of()); ...
9 int DmatrixType_Parameter::count=4;
10 ...
11 MPI_Datatype *DmatrixType_Parameter::
12 typeArray= new MPI_Datatype[2];
13 typedef Struct<DmatrixType_Parameter
14 DmatrixType;
15 typedef Contiguous CDmatrixType, 2>
16 DepartureScanType;

17 DepartureTimes depart;
18 Dmatrix msg(2), *glob_sum;
19 scan_msg[1]=D[num_jobs-1];
20 glob_sum=depart.Scan(scan_msg);

In line 27 above, a user-defined MPI reduction operator
is created during the construction of a class instance depart
of Reduction class template, where user_func() is associated
with the operator. In line 28, the send data in Scan operation,
scan_msg is an array of 2 Dmatrix elements. The local
sum of departure time in D[num_jobs-1] is assigned to
scan_msg[1] (line 29), and then the Scan reduction
operation is performed (line 30). During the Scan reduc-
tion operation, user_func() computes the global prefix sum
of departure time (lines 20-26). The Pointer glob_sum of
Dmatrix points to the received data, i.e., the array of two
Dmatrix values containing the global prefix sum (lines 28
and 30). The value in glob_sum[0] has the global prefix sum
for each process to finish the computation of a global depa-
ture time sequence D.

3. Groups, Attributes, and Communicators

This section introduces the classes in mpi++ that capture
the group and communicator services defined in Chapter 5
of the MPI standard.

3.1. Overview

Groups define an ordered collection of processes, each
with a rank [7], a low-level name for inter-process com-
munication. Groups define a scope for process names in
point-to-point communication, and collective operations. A
communicator encapsulates a group, and uses defined attri-
butes [7].

There are two kind of communi-
cators: intra-communicators for operations within a single
group of processes, and inter-communicators, for point-to-
point communication between two groups of processes.

The "caching " facility allows an application to attach ar-
bitrary pieces of information, called attributes, to commu-
icators. The caching interface provides accessor functions
that obtain a key value used to identify an attribute. The user
specifies "callback " functions by which MPI informs the ap-
plication when the communicator is destroyed or copied.
Attributes may be set and queried by the program during ex-
ecution.

The Communicator and Group classes maintain a pointer
to a class object of MPIComm and MPIGroup, respectively.
Each of the latter classes maintains the underlying MPI re-
source, either a group or an attribute, with a reference count.
This is efficient and necessary in that several communication
objects may share the same Communicator or Group object.
While these objects sharing an attribute may be destructed at
different times, the MPIComm or MPIGroup will not be de-
structed until the reference counter is decreased to zero, i.e.,
until the last referencing communication object is destruct-
ed. This can be observed in the assignment operator of the
Communicator class:

1 MPIComm *Comm;
2 void operator=(Communicator &c)
3 { if(!!(this->Comm) & !(Comm->decrement()))
4 delete Comm;
5 Comm = c.comm(); Comm->increment(); }

The assignment operator references to an MPI communi-
cator of the existing Communicator object c, and the refer-
cence counter is incremented by the increment() call associ-
ated with (the pointer to) a MPIComm object Comm of the
new Communicator (line 5). The new Communicator ob-
ject decrements the reference counter when it is destructed,
and destructs the MPIComm object IF AND ONLY if the
counter equals zero:

"Communicator()"
Figure No. 3 illustrates the relationships between the Communicator (Group) class, the MPIComm (MPIGroup) class, and the MPLCommunicator (MPI_Group) objects. Let X and Y be communication objects, both use the same MPI communicator. Both have a Communicator object A and B, respectively. However, B references the MPIComm object of A (A.comm()). Thus X and Y can share the same MPI communicator (A.comm().MPI_comm()) while the reference counter of the MPIComm object equals 2 in this case. The motivation of this approach is to avoid overhead associated with duplication.

Due to the object-oriented nature of functions for groups and communicators in MPI standard, some group functions are mapped directly to the definition of class Group. For example, the function MPLGroup_free() is called by the MPIGroup destructor. However, some other group functions are mapped into several member functions of the Group class. For example, the three member functions operator==(Group &g), similar(Group &g), and operator!=(Group &g) return true values when the results of MPI-Group_compare() operation are MPI_IDENT, MPI_SIMILAR, and MPI_UNEQUAL, respectively.

Group constructor operations are implemented by class constructors, operator overloading, and several named member functions. Some parameters have default values, simplifying application programs when the constructors are used in common case. Operator overloading is also applied for accessor functions, hence the same accessor function in mpi++ would be more symbolic compared to the C binding. For example, the addition operator “+” implements the MPI-Group-union0 operation. If gl and g2 are two Group objects, then gl+g2, returns the union of gl and g2. Pointers passed as parameters are avoided in the Group class, the return values of class functions are references of objects. The use of references reduces potential pointer manipulation errors which appear frequently in application programs. The following program fragments compare mpi++ versus the MPI C binding in the construction of communicators by both the duplication and the creation using group information from the old communicator:

```cpp
1 Communicator comm;
2 Communicator newComm(comm, (Group &)comm);
```

Due to the object-oriented nature of functions for groups and communicators in MPI standard, some group functions are mapped directly to the definition of class Group. For example, the function MPLGroup_free() is called by the MPIGroup destructor. However, some other group functions are mapped into several member functions of the Group class. For example, the three member functions operator==(Group &g), similar(Group &g), and operator!=(Group &g) return true values when the results of MPI-Group_compare() operation are MPI_IDENT, MPI_SIMILAR, and MPI_UNEQUAL, respectively.

Group constructor operations are implemented by class constructors, operator overloading, and several named member functions. Some parameters have default values, simplifying application programs when the constructors are used in common case. Operator overloading is also applied for accessor functions, hence the same accessor function in mpi++ would be more symbolic compared to the C binding. For example, the addition operator “+” implements the MPI-Group-union0 operation. If gl and g2 are two Group objects, then gl+g2, returns the union of gl and g2. Pointers passed as parameters are avoided in the Group class, the return values of class functions are references of objects. The use of references reduces potential pointer manipulation errors which appear frequently in application programs. The following program fragments compare mpi++ versus the MPI C binding in the construction of communicators by both the duplication and the creation using group information from the old communicator:

```cpp
1 Communicator comm;
2 Communicator newComm(comm, (Group &)comm);
```

The statement in line 1 does the same work as line c, since it applies the Communicator class constructor, Communicator(), that duplicates MPI.COMM_WORLD by default. The mpi++ version is concise because the Communicator class constructor handles the duplication detail. The statement in line 2 creates a new communicator duplicating comm and its group information. However, the C version calls two functions to do the same work (lines d and e). Mpi++ provides a
constructor, Communicator(Communicator &c, Group &g),
that extracts the group information from the Communicator
instance, comm, specified as the second parameter in line 2.
This again simplifies application programs.

The example below creates lowGroup, whose group
members are a subset of the group in a Communicator in-
stance comm, each with rank 0, 1, 2, ..., int[world.size/2],
where world.size is the number of members of the group in
comm, called low part of the original group.

```c
1 int n, worldSize, ranges[1][1][3];
2 worldSize=mpi::size();
3 //Duplicate MPI_COMM_WORLD(by default)
4 Communicator comm;
5 n = worldSize/2;
6 ranges[0][0] = 0;
7 ranges[0][1] = (worldSize - n) - 1;
8 ranges[0][2] = 1;
9 //Group include function
10 Group lowGroup=(Group &) comm .
11 range-incl(l,ranges);
12 int rank, worldRank=mpi::rank();
13 Communicator newComm(comm, Group &c).
14 Communicator lowComm(newComm, lowGroup);
15 if (worldRank<(worldSize-n)) {
16 if( (rank=lowComm.rank())==mpi::Undefined)
17 Abort("incorrect lo group rank: ");
18 else lowComm.barrier();
19 } else {
20 if (!lowComm.nullComm())
21 Abort("incorrect lo comm");}

```

3.3. Attribute Class

The Attribute class handles attribute caching for com-
municators. A pointer to the data that carries the attribute
information for the associated communicator is stored as
protected data of the Attribute class. The key value for an
attribute is created during the construction of an Attribute
class instance. The key value is used when an attribute is
retrieved or attached to communicator objects. Copying an
attribute of a communicator when this communicator is du-
plicated, and deleting the attribute when the communicator
is destructed are performed by the static member functions
Copy0 and Del(). The Copy() function is an interface to
function copy(). Function copy() is a virtual function in the
Attribute class definition, it must be substituted by a func-
tion definition when a derived class of Attribute is crearted.
Attribute is an abstract base class; the user must use this to
define classes for attributes in an application.

```c
class Attribute {
    static int Copy(mpi::CommType *c, int *key, void
    "extra_st, void *att, void *att_out, int f);
    static int Del(mpi::CommType *c,
    int *key, void *att, void *extra_st);
    ... protected;
    void *state, *data;
    
    int flag, key;
    virtual Attribute *copy() =0;
    virtual int del() =0;
    public:
    Attribute(void* d=0, void *s=0);
    "Attribute();
    int &keyval(); // get keyval
    void operator=(Attribute &d); // put attribute
    void *attribute(); // get attribute
    ...
}
```

Continued from section 3.2, the segment below between
lines 22 and 32 defines class Rank that is derived from
the Attribute class. Line 35 attaches the object rankAtt of
Rank to the Communicator instance loComm. Retrieving
attribute values is also performed in this example using the
subscript [] operator (line 40).

```c
22 class Rank :public Attribute {
23 public:
24 Rank(void* d) :Attribute(d) {};
25 Attribute *copy() {
26 {return (Attribute *) new Rank(data);}
27 int del() {
28 {Communicator comm;
29 if (((this->attribute()) !=(comm.rank()))
30 //abort "incorrect attribute value 
31 return mpi::Success; }
32 );
33 Rank rankAtt((void *)&worldRank);
34 // Attach attribute
35 lowComm = rankAtt;
36 Communicator dupComm;
37 // Copy() gets called
38 dupComm.dup(lowComm)
39 // Get attribute of rankAtt
40 int value = (int)dupComm[rankAtt];
```

3.4. Communicator Class

The Communicator class is implemented as the Group
class. It has a data member Comm, a pointer to a class ob-
ject of MPIComm. Class MPIComm maintains a reference
counter of the MPIComm object for every Communicator
object that points to it. The MPIComm. Communicator and
InterCommunicator classes are defined as follows:

```c
a class MPIComm {
  int Ref;
  mpi::CommType comm;
  public:
  MPIComm(mpi::CommType c=mpi::commWorld)
  ...
  void dup(mpi::CommType &c);
  ...
  int decrement();
  int decrement();
  ...
  h class Communicator {
  i MPIComm *Comm;
  j MPIComm *comm;
```

85
There are four kinds of member functions in the Communicator class: the constructors, the destructor, the accessors, and the caching functions. The communicator constructor MPI Comm create() is encapsulated as the constructor in line d. The function MPIComm_dup() is abstracted in the member function in line e. There are two other constructors on lines 1 and m, where the latter overloads the "=\" operator. The assignment overloading constructor differs the function in line e in that the former one assigns the MPI communicator of the input Communicator to its pointer Comm, and increments the reference counter, thus both communicators share the same MPI communicator resource; the latter duplicates the MPI communicator of the input Communicator, so each has its own MPI communicator, although the contents are exactly the same. Duplication is useful in some cases such as the duplicated MPI communicator needs to be changed. Among the accessors of class Communicator, MPIComm_compare() is implemented as four functions, one of them is congruent(Communicator &comm). If the current communicator is c, it returns true if c is congruent to comm. The segment between lines 12 to 21 in section 3.2 generates a new communicator lowComm with lowGroup while duplicating newComm, then does a few tests.

The MPI attribute caching functions are captured by these member functions of class Communicator. The subscript operator function in line p returns the attribute that is associated with the key value of a. Function in line q copies the attribute value of Attribute object a. The function MPI.Attr/delete() is captured by the member function in line r. The mpi++ interface of caching functions is simpler than the MPI C binding.

Inter-communication refers to communication between two non-overlapping groups, that is, a point-to-point communication between processes in different groups. This is especially convenient for applications using a client-server computing paradigm. The mpi++ InterCommunicator class inherits properties of the Communicator class, provides methods corresponding to the MPI inter-communicator constructors, destructors, and destructors. The data member Comm is a class instance of MPIComm that maintains a reference counter of the MPI inter-communicator object for every InterCommunicator instance that points to it. The accessor group0() returns a Group object that has the local MPI group object of the inter-communicator. The accessor in line x returns a Group object that has remote group object. The accessor in line w returns the number of processes in the remote group of the inter-communicator. The operator in line y returns a Communicator object that has an intra-communicator created from the union of the two groups that are associated with the inter-communicator. The following is the definition of InterCommunicator class.

4. Conclusions

This paper presents those parts of mpi++ that deal with collection operation and communicator. Class templates for collective communications are illustrated by a brief example, contrasted with the same example using the C binding. The version of mpi++ described in this paper includes the services for collective communication as described in Chapter 4 of the MPI standard, and also the services for groups, communicators, and attribute caching (Chapter 5). The mpi++ code for the current version of MPI has been tested on a Sun workstation and an Intel Paragon using the Sun CC and Intel IC compiler, respectively.

References