TPO++: An object-oriented message-passing library in C++*

Tobias Grundmann, Marcus Ritt, Wolfgang Rosenstiel
Wilhelm-Schickard-Institut für Informatik,
University of Tübingen
Department of Computer Engineering
Sand 13, 72076 Tübingen
{grundman,ritt,rosen}@informatik.uni-tuebingen.de

Abstract

Message-passing is a well known approach for parallelizing programs. The widely used standard MPI (Message passing interface) also defines C++ bindings. Nevertheless, there is a lack of integration of object-oriented concepts. In this paper, we describe our design of TPO++[1], an object-oriented message-passing library written in C++ on top of MPI. Its key features are easy transmission of objects, type-safety, MPI-conformity and integration of the C++ Standard Template Library.

Keywords: Object-orientation, C++, Message-passing, MPI, Scientific computing

1 Introduction

With MPI, a widely accepted standard for parallelizing applications using message-passing has been established. At the same time, object-oriented programming concepts gain increased acceptance in scientific computing (see, for example [1]).
In our project physicists are simulating astrophysical systems, e.g. the behavior of a gaseous disk around a compact star. The problem reduces basically to the simulation of complex particle systems with short-range interactions. To handle the complexity of the software and ease the reuse of components, like particle containers etc. the simulation software is developed in C++ and parallelized using MPI.
The message-passing standard MPI defines C++ bindings for all MPI interface functions. But these bindings provide merely wrappers around MPI constructs and fit not well into object-oriented concepts. MPI provides no means for transmitting objects. The STL (Standard Template Library) – an important part of the current C++ standard [4] – is not considered. In the design of TPO++, we try to address these problems.
In Section 2 we state our design goals for an object-oriented implementation of a message-passing system in C++. Some exsiting message-passing systems are reviewed and discussed in Section 3. We explain the implementation details in Section 4 and introduce our system from a user's viewpoint in Section 5. Section 6 shows a comparison with MPI.

2 Design goals and problems

2.1 Design Goals

In this section we give an overview of the features we consider to be most important in a modern message-passing system designed for use with C++.

Object-orientation and type-safety A major goal in providing an object-oriented class library for message-passing is a tight integration of object-oriented concepts, in particular the capability of transmitting objects in a simple, efficient and type-safe manner. The solution must not break essential features like encapsulation and class inheritance.

C++ integration An implementation in C++ should take into account all recent C++ features, like the usage of exceptions for error handling and, even more important, the integration of the Standard Template Library by supporting the STL containers as well as adopting the STL interface conventions.

MPI conformity The design should conform to the MPI interface, naming conventions and semantics as closely as possible without violating object-oriented concepts. This

*This project is funded by the DFG within SFB 382: Verfahren und Algorithmen zur Simulation physikalischer Prozesse auf Höchstleistungsrechnern (Methods and algorithms to simulate physical processes on supercomputers).

[1] Tübingen Parallel Objects

helps migrating from C bindings and eases porting of existing C or C++ code.

Efficiency The implementation should not differ much from MPI in terms of communication and memory efficiency. Therefore it should be as lightweight as possible, allowing the C++ compiler to statically remove nearly all of the interface overhead. Communication should be done directly via MPI calls and, if feasible, with no additional buffers, which saves memory and enables the underlying MPI implementation to optimize the communication, for example if the network hardware is able to send and receive scattered memory blocks automatically.
For performance reasons we consider an implementation which does not actually transmit type-information in messages and does also not handle different data-layouts on heterogeneous networks of workstations, since our intended applications will run on homogeneous architectures like massively parallel computers.

Thread safety Another goal is to guarantee thread-safety to provide maximum flexibility for application software and parallel runtime systems based on the library. This topic will not be further discussed here since thread-safety is optional in the MPI-Standard and depends mostly on the underlying MPI-Implementation.

2.2 Problems

In an object-oriented message-passing system one would ideally like to have a simple interface providing a single send and a single receive method to which every object could be passed in a type-safe manner and without having the user to give any information about the objects to be transmitted.
A first approach may be to use overloading. This works for built-in types and known library types such as STL containers. Obviously this cannot work for user-defined types, since the user would have to overload methods of our message-passing classes. A solution is to provide an abstract base class from which the user types could inherit marshalling methods. Then, overloading the send and receive methods with this base class and using virtual marshalling methods to get data in and out of the user-defined types would be possible. This is certainly an approach for dynamic user-defined data structures. The drawback is the additional overhead of virtual method calls and the need for the user to specify serialize and deserialize methods for *each* class. Even so, an abstract base class must be provided to support applications, which rely on redefined methods in derived classes. A particular method in such an application will be defined in terms of a base class and may wish to transmit its parameters. To enable the transmission of the actual (derived) objects there is no way but to use virtual methods.
It could be argued that external overloaded operators or functions can achieve quite a lot of functionality without the overhead of virtual methods. This clearly excludes the above mentioned applications. Further, the user still has to define marshalling methods even if the types to be transmitted are trivial.

3 Existing approaches

This section discusses other message-passing systems written in C++. We aim for massively parallel scientific computing, therefore approaches intended for distributed computing like CORBA are neglected. We do not discuss additional functionality, besides message-passing, present in some of these systems. The focus lies on the integration of object-orientation, i.e. the way objects can be transmitted, support of type safe programming, the integration of C++ concepts, in particular the support for STL datatypes, the interface complexity and conformance to MPI semantics. This is discussed in context of point-to-point communication.

3.1 MPI C++ bindings

The current MPI-2 standard [7, 8] defines C++ language bindings for MPI-1 and MPI-2. The MPI library is encapsulated in a `MPI` namespace, which contains all symbolic constants and some simple wrapper classes, representing the MPI objects, most notably the communicator classes. MPI++ [10] was an early implementation of the C++ bindings for MPI.
Unfortunately these bindings are no significant improvement compared to the C bindings. The interface is not type-safe, makes no attempt to simplify the MPI calls and defines no way for transmitting objects.

3.2 Mpi++

The mpi++ system [5, 6] proposes an interface to MPI with messages as central point of view. A message object does not only contain values to be sent but also specifies how to send them. Messages are implemented as templates with two arguments: The type of the actual value to be transmitted and the communication semantics (e.g. blocking or nonblocking). Thereby mpi++ introduces its own types for basic C++ types, e.g. there is a type `FLOAT` to be used as template parameter when a `float` is to be transmitted.
User-defined types are supported similar to plain MPI. The data construction functions of MPI are implemented by separate classes. For the transmission of user-defined objects, the template `Struct` has to be instantiated, which

takes the type of the object as a template parameter. To make this work, the user has to provide the parameters of `MPI_Type_Struct`[2] (i.e. offset, length and basic type of the class data members) as additional static members of the class to be transmitted.
The concept of defining communication semantics only once, when defining the type, corresponds to MPI persistent communication. A possible disadvantage of this approach is that Mpi++ provides no other way to express communication. If the communication parameters are not uniform this creates overhead. Further it reduces code readability, since semantic information is usually located far away (perhaps in a header file) from its use.
The implementation of user-defined types does not substantially simplify the MPI datatypes interface. This approach also prohibits inheritance of type information, since the static description types have to be built again for every derived class.

3.3 OOMPI

The designers of OOMPI [11] concentrated on a "syntactically and semantically consistent interface with MPI". OOMPI provides two major abstractions: *Ports*, which can be seen as communication endpoints encapsulating MPI ranks, and *Messages*, representing data to be transmitted. Messages can be built explicitly but for basic datatypes and arrays they are constructed automatically, when given as an argument to send or receive calls. User-defined datatypes are supported by inheriting from `OOMPI_User_type` and must define a static `OOMPI_Datatype` object representing its type signature. This datatype object usually gets defined in the constructor of the user-defined type.
While OOMPI allows to transmit objects, the approach taken there has some problems: The requirement to inherit from a special user-type class complicates the reuse of existing class-hierarchies, which must be achieved by multiple inheritance. The code for building the type representation in the constructor of an user-defined object cannot be inherited and therefore must be duplicated in all child classes. Representing user-defined types by static type descriptors also makes it impossible to transmit any kind of dynamic data structure. Thus the transmission of STL container classes is not supported.

3.4 Para++

Para++ [2] focuses on providing a common interface for different message-passing libraries (namely PVM [3] and MPI). Its key features are I/O-facilities based on stream objects similar to standard C++ streams `cin` and `cout`. For parallel tasks, these streams provide nearly unified communication mechanisms for point-to-point communication, multicasting and broadcasting. There is no direct support for user-defined objects, instead a user has to implement the stream operators for new types.
Para++ misses the integration of STL datatypes. If collections of objects are to be transmitted, there is no way to avoid iterating over each element in the collection, even if it is just a simple array, which could be packed and sent at once. Besides, while compatible to C++ standard I/O, the interface is considerable different from MPI conventions and semantics.

[2] A MPI type-constructor

4 Implementation of TPO++

Our library was initially developed on a network of workstations running Solaris 2.7 and MPICH 1.1.2. The same environment is used for development and testing of our applications. For production runs, the library has been tested on a Cray T3E (512 nodes at 450 MHz) using the native MPI implementation. It compiles with all recent versions of GNU CC as well as KAI C++.
To solve the problems mentioned in Section 2.2 in a generic way, we have to distinguish at least four different characteristics of types, which are not mutual exclusive: User-defined types versus built-in or predefined (library) types and static versus dynamic sized types. Here we can treat static sized types, which are scattered around in memory, like dynamic ones, since both have no trivial copy constructor and must be handled by marshalling functions. It is worth noting that all these characteristics are in a particular way generic. They are not characteristics of single types but characteristics of whole classes of types. One would like to handle each of these classes with only minimal user assistance. C++ unfortunately does not provide mechanisms to analyze types in a programmed way at compile time, but by using *traits*, this problem can be circumvented in a quite elegant way.
The process of marshalling objects is shown in Figure 1 (simplified). Traits are used to determine the nature of the objects to be serialized at compiletime. This way simple objects are serializable without actually calling any extra functions. This applies recursively to members of objects.

4.1 Distinguishing type characteristics with traits

The traits technique (first developed by Myers [9]), allows to write generalized code, in which one does not generalize over a type as in plain templates, but over particular characteristics of types. Depending on such characteristics the compiler may generate specialized code. The trick is to give information about such characteristics not as another template parameter, which may be annoying to the user who

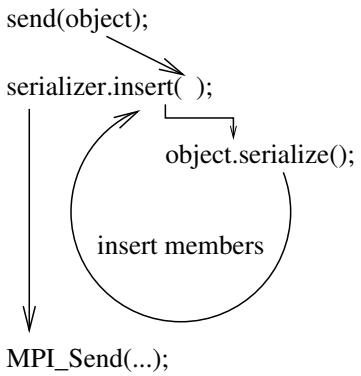

Figure 1. Serialization in TPO++

is normally not interested in implementation details, but instead by another template. This other template – the traits template – can be used internally to gather specific information about the actual template parameter. An example may illustrate this. Suppose you wish to know if a type supplied as a template argument is a container type. If you have specialized traits templates for containers, you can get this information and act accordingly in a type-safe manner as shown in Figure 2. Note that the extra function calls (`_do_it()`) can be easily removed by the compiler. This applies also to the additional (empty) arguments to these functions.

TPO++ is based on exactly one trait for a type-safe mapping of arbitrary C++ types to MPI basic types. In contrast to the simple yes/no differentiation in the example above this trait maps all possible classes of types to appropriate marshalling functions. For user-defined types these are either user-provided or functions which use the trivial copy constructor mechanism, depending on the trait specialization. All basic C++ types are marshalled with the trivial copy constructor mechanism also. For STL containers our library provides already the necessary specializations. Developers can implement their own STL-compatible container types by specializing the `tpo_traits` construct. In this way, containers can be transmitted, if their elements can be transmitted, without additional code. The transmission of trivial containers automatically gets optimized, e.g. for sending a `vector<double>` we do not have to iterate over the complete container.

Utilizing these trait, any type given to the top-level communication methods can be unambiguously broken down to a sequence of basic types. Note that most of the reduction is static, i.e. can be done at compile-time; therefore we can be sure to preserve the efficiency of the underlying MPI implementation. This information is sufficient to pack the data into a communication buffer or to create a MPI datatype for transmission. For performance reasons, our current imple-

```
struct true_type {};
struct false_type {};

//default assumes parame-
ter is not a container
template <class T>
struct container_traits {
     typedef false_type container_type;
};

//specialization for vec-
tor which is a container
template <class T1,class T2>
struct container_traits <vector<T1,T2> > {
     typedef true_type container_type;
};

//specializations for other containers
//...

class X {
public:
     template <class T>
     void do_it (T& x) {
        _do_it(x,
          container_traits<T>::
            container_type());
     }
private:
     template <class T>
     void _do_it (T& x,true_type) {
     //code to handle container
     }

     template <class T>
     void _do_it (T& x,false_type) {
     //code to handle all other types
     }
};
```

Figure 2. Discrimination between a container type and other types

mentation addresses only homogeneous architectures. Further, no type information is actually transmitted. The memory blocks obtained by the reduction mechanism are transmitted directly.
The type information gained from the reduction could be used to build a MPI-datatype on the fly. If the optimizations for trivial copy constructors were omitted, this would also generalize the implementation to heterogeneous architectures.

5 Interface and examples

The basic structure given in the C++ bindings of MPI is similar to our approach. All common MPI objects, i.e. communicators, groups, status, are implemented as separate classes. The improvements can be found in the communication interface, the handling of user-defined classes and error handling.
Startup is implemented by the initialization function `TPO::init()`, which processes the command-line arguments and starts the message-passing environment. After initialization, the user can use the global Communicator `CommWorld`. `CommWorld` is a predefined object of type `TPO::Communicator` and includes all participating hosts. The shutdown is done automatically on destruction of `CommWorld`, that is, after the application terminates. For explicit shutdown, the function `TPO::finalize()` is provided.

```
int main(int argc, char *argv[]) {
  TPO::init(&argc, &argv);
  // main code here
  // implicit MPI_Finalize
}
```

Transmitting of predefined C++ types Predefined C++ types are library and basic types. In the case of sending a C++ basic type, the send call reduces to:

```
double d;
CommWorld.send(d, dest_rank);
```

STL containers can be sent using the same overloaded communicator method. The STL conventions require two iterators specifying begin and end of a range, which also allows to send subranges of containers:

```
vector<double> vd;
CommWorld.send(vd.begin(),
               vd.end(),
               destination_rank);
```

The sender can provide a message tag as an additional argument to the send methods. If omitted, as in the example above, it defaults to 0.

The application can also use the blocking, synchronous and ready send modes defined in MPI by calling the communicator methods `bsend`, `ssend` and `rsend`, respectively.
On the receiver side, the library is equally easy to use. For basic datatypes a receive-call is done as follows:

```
Status status;
status=CommWorld.recv(d);
```

Note that the status object, different from MPI, is a return parameter, because error handling is done via exceptions. This simplifies the receiver code, if no error checking is necessary and makes send and receive calls more symmetric. The receive methods take two optional arguments, the senders rank and a message tag for selecting particular messages. If omitted, they default to any sender and any tag, respectively.
For receiving a container, a single argument, specifying the insertion point is sufficient. Conforming to the STL interface, the data can be received into a full container, where the previous data will be overwritten. Receiving into an empty container or at the end of a full container can be done by means of an *inserter*, which allocates space for the new data automatically, just like STL inserters. The next example shows both approaches:

```
vector<double> vd1(x);
vector<double> vd2;

// vd1 must provide enough space
CommWorld.recv(vd1.begin());

CommWorld.recv(
   tpo_back_insert_iterator(vd2));
```

Persistent and asynchronous communication For asynchronous communication, the communicator class provides asynchronous send methods `isend`, `ibsend`, `issend` and `irsend` for all communication modes and an asynchronous receive method `irecv`. They all return an object of class `Request`. The application can wait or test for the completion of the asynchronous communication using its methods `wait()` and `bool test()`. The status of a completed receive call can be obtained by calling `status`.
MPI also defines persistent communication calls, which encapsulate the communication parameters and allows the communication to be started multiple times using the same arguments. This allows an implementation to optimize communication locally. Persistent communication in MPI is, by default, asynchronous.
In TPO++, persistent send and receive calls are implemented in classes `Persistent_send` and `Persistent_recv`. They are constructed by providing normal send or receive parameters. Unlike in MPI, objects of these

classes can be given as a parameter to all send methods in class `Communicator`, allowing blocking as well as asynchronous persistent communication.

User-defined types To enable a class for network transmission, the user has to specialize the template `tpo_traits` provided by our library. This template specifies its *marshalling category*. As a notational convenience our library provides also short macros to perform the specialization.
For objects having a trivial copy constructor[3], the marshalling category can be set appropriately:

```
namespace TPO {
struct net_traits<User_type> {
    typedef
      has_trivial_copy_constructor
        net_category;
};
}
```

Using a macro, this reduces to `TPO_TRIVIAL(User_type)`. On transmission, the memory block occupied by such an object will be copied directly to the net.
For transmitting complex objects (i.e. without a trivial copy constructor), the user has to define the marshalling methods `serialize` and `deserialize`, as part of the class definition. The presence of these methods must be signaled by a marshalling category of type `has_serializer`, or in macro notation: `TPO_MARSHALL(User_type)`. The only argument of the marshalling methods is a `Message_data` object, provided by the library, used to marshal the objects member data. The `Message_data` class provides insert and extract methods, whose arguments are all kinds of transmittable types. In a serialize method, `insert()` is called repeatedly for every member to prepare the object for transmission. If a member is not inserted it will not be transmitted just like *transient* variables in Java.
The `Message_data` object does not copy the data, but records its memory layout for later transmission. Similarly, a received message can be unpacked to user-provided memory locations by calling extract in the deserialize method.
The code in Figure 3 and 4 implements two user-defined classes, a class with a trivial copy constructor and a little more complex one with marshalling methods. The Circle class could also use a completly different kind of center - i.e. one which has `serialize` and `deserialize` methods itself - without changing its own marshalling methods. Note that in this case both do not have to inherit from any special "message" class. Also note, that the marshalling code given in the `serialize` and `deserialize` methods can be reused in derived classes without modifications. A subclass simply calls the marshalling methods of its base and adds marshalling code for its own members.
For applications relying on virtual methods and generic interfaces an abstract base class `Message` is also provided. This means a user-defined class can also use the benefits of a "real" object-oriented design, paying the usual prize of a loss in performance.

[3] A trivial copy constructor implies a linear memory layout and allows to copy the object with memcpy.

```
class Point {
public:
  Point() : x(0), y(0) {}
private:
  int x, y;
};
TPO_TRIVIAL(Point);
```

Figure 3. User-defined object with trivial copy constructor

```
class Circle {
public:
  Circle() : radius(0.0) {
    center = new Point(0.0);
  }
  ~Circle() { delete center;}
  void
  serialize(Message_data& m) const {
    m.insert(*center);
    m.insert(radius);
  }
  void
  deserialize(Message_data& m) {
    m.extract(*center);
    m.extract(radius);
  }
private:
  Point* center;
  double radius;
};
TPO_MARSHALL(Circle);
```

Figure 4. Dynamic user-defined object

Error handling In TPO++, error handling is completely based on C++ exceptions, conforming to the MPI C++ bindings. In particular, exceptions carry error information, for example the error code and the faulting communicator. To avoid excessive try-catch-blocks, the user can also define global exception handlers. In contrast to the MPI C++ bindings, where global error functions are used, TPO++ uses its own class `ErrorHandler`, which can be specialized by

the application. In case of an error, the `handle` method of the current error handler object is called with an exception. This approach makes the C++ exception mechanism and global error handlers more similar.

6 Measurements

We compared the efficiency of our library on these platforms using a simple ping test and measured the achieved latencies and bandwidths of MPI and TPO. The difference shows the loss in performance due to the abstraction.
Figure 5 shows our measurements on a 100 MBit LAN and a Cray T3E. The measurement on the upper left shows, that communication using TPO on a Cray T3E achieves the same bandwidth as MPI for messages larger than approximately 16 KB. The loss in bandwidth below this size is a constant factor of 2 and mainly due to the increased latency shown on the right upper side. Latencies of MPI and TPO converge as messages are getting larger. For small messages up to approximately 16 KB, TPO shows a constant latency overhead of $30\mu s$ compared to $15\mu s$ using MPI. The measurements on the lower half show, that using a 100 MBit LAN, the overhead introduced by TPO can be neglected. Bandwidth and latencies are identical to MPI.

7 Summary

We have discussed our implementation of an object-oriented message-passing system. Our system exploits object-oriented and generic programming concepts, allows the easy transmission of objects and makes use of advanced C++ techniques and features as well as supporting these features, most notably it supports STL datatypes. The system introduces object-oriented techniques to message-passing while preserving MPI semantics and naming conventions as far as possible. This simplifies the transition from existing code. While providing a convenient user interface its design is still type-safe and efficient. By abstracting over type classes instead of concrete types it enables automatic optimization. In contrast to other implementations the code to marshall an object can be reused in derived classes. Also, our library is able to handle arbitrary complex and dynamic datatypes.
The library is designed as a base for parallelizing scientific applications in an object-oriented environment. We are working on extending the library interface for collective communication.

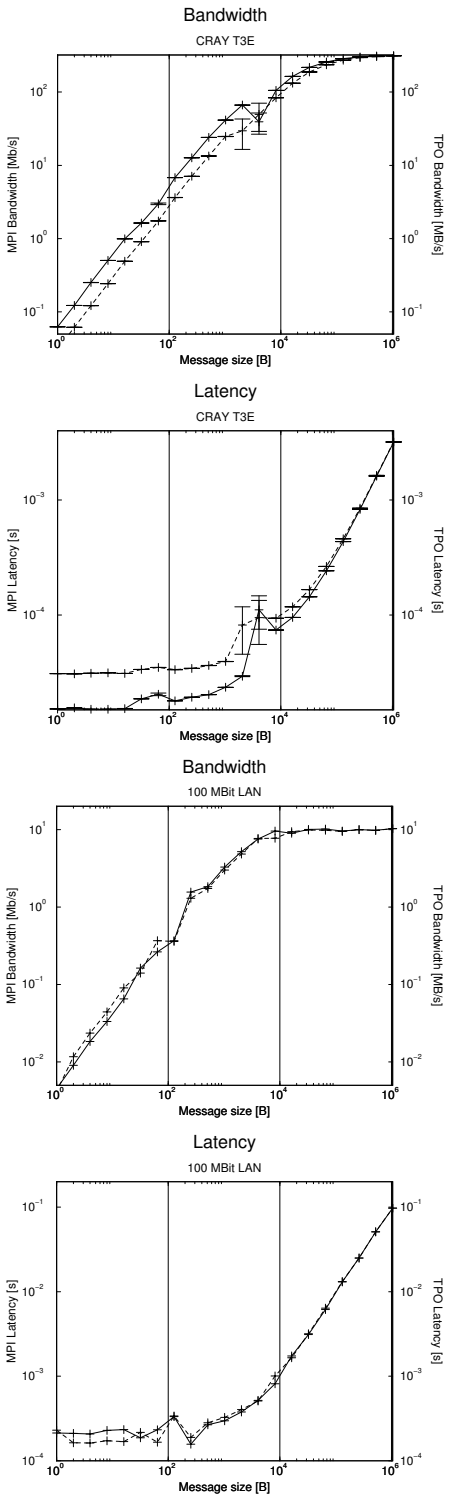

Figure 5. Comparison of MPI (solid curves) and TPO (dashed curves).

References

[1] F. Bassetti, K. Davis, and B. Mohr, editors. *Proceedings of the Workshop on Parallel/High-Performance*

Object-Oriented Scientific Computing (POOSC'99), European Conference on Object-Oriented Programming (ECOOP'99), Technical Report FZJ-ZAM-IB-9906. Forschungszentrum Jülich, Germany, June 1999.

[2] O. Coulaud and E. Dillon. Para++: C++ bindings for message-passing libraries. Users guide. Technical report, INRIA, 1995.

[3] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Mancheck, and V. Sunderam. *PVM 3 User's Guide and Reference Manual*. Oak Ridge National Laboratory, Tennessee, September 1994.

[4] International Standards Organization. Programming languages – C++. ISO/IEC publication 14882:1998, 1998.

[5] D. Kafure and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995.

[6] D. Kafure and L. Huang. Collective communication and communicators in mpi++. Technical report, Department of Computer Science Virginia Tech, 1996.

[7] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical Report CS-94-230, Computer Science Department, University of Tennessee, Knoxville, TN, May 1994.

[8] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.

[9] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[10] A. Skjellum, Z. Lu, P. V. Bangalore, and N. Doss. Mpi++. In G. V. Wilson and P. Lu, editors, *Parallel Programming using C++*, chapter 12, pages 465–506. The MIT Press, Cambridge, 1996.

[11] J. M. Squyres, B. C. McCandless, and A. Lumsdaine. Object Oriented MPI: A Class Library for the Message Passing Interface. In *Proceedings of the POOMA conference*, 1996.