Object-oriented message-passing with TPO++[*]

Tobias Grundmann, Marcus Ritt, and 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.

1 Motivation and design goals

With MPI, a widely accepted standard for message-passing has been established. At the same time, object-oriented programming concepts gain increased acceptance in scientific computing (see, for example [1]).
The MPI-2 standard [6, 7] defines C++ language bindings for MPI-1 and MPI-2, but these bindings are no significant improvement compared to the C bindings. The interface is not type-safe, does not simplify the MPI calls and defines no way for transmitting objects. Other approaches such as the mpi++ system [4, 5], OOMPI [8] and Para++ [2] improve certain aspects of message-passing in C++, but, besides some other topics, none of them integrates the STL (Standard Template Library), an important part of the current C++ standard [3]. Further they often don't support user-defined types very well or there is a significant difference to MPI conventions, even if not necessary to introduce object-oriented concepts. In the design of TPO++, we try to address these problems:
A major goal in providing a class library for C++ is a tight integration of object-oriented concepts, in particular the capability of transmitting objects in a simple, efficient, and type-safe manner. Also, an implementation in C++ should take into account all recent C++ features, like the usage of exceptions for error handling and the integration of the Standard Template Library by supporting the STL containers as well as adopting the STL interface conventions. The interface design should conform to the MPI conventions and semantics as closely as possible without violating object-oriented concepts. This helps migrating from C bindings and eases porting of existing C or C++ code. Further, the implementation should not differ much from MPI in terms of communication and memory

[*] This project is funded by the DFG within SFB 382.
[1] Tübingen Parallel Objects

efficiency. Another goal is to guarantee thread-safety to provide maximum flexibility for application software and parallel runtime systems. 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 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, are implemented as separate classes. After initialization, the user can use the global Communicator object `CommWorld`.

Transmission of predefined C++ types In the case of sending a C++ basic type, the send call reduces to:

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

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(), dest_rank, message_tag);
```

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. Asynchronous communication methods return an object of class `Request` which can be used for the application to test or wait for the completion. On the receiver side, a receive-call is done as follows (basic type):

```
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.
To receive a container, a single argument, specifying the insertion point is sufficient. Conforming to the STL, the data can be received into a container which is large enough by means of an iterator, or into any container by means of an *inserter*. The example shows both approaches:

```
vector<double> vd1(x);  // must provide enough space
vector<double> vd2;
CommWorld.recv(vd1.begin());
CommWorld.recv(tpo_back_insert_iterator(vd2)); //allocates space
```

Transmission of user-defined types To enable a class for transmission, the user has to declare its *marshalling category.* The library distinguishes user-defined objects having a trivial copy constructor and complex user-defined objects. Enabling a class with a trivial copy constructor for transmission reduces to the statement `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 declaration of `TPO_MARSHALL(User_type)`. They are supplied with a serializer object. In a serialize method, `insert()` is called repeatedly for every member to prepare the object for transmission. The serializer 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.
Usually, user-defined objects do not have to inherit from any special "message" class. Also note, that the code given in the marshalling methods can be reused in derived classes.
For applications relying on virtual methods and generic interfaces an abstract base class `Message` is also provided.

3 Comparison with MPI

The tests have been done on Sun Ultra 5 machines (Solaris 2.7) using MPICH 1.1.2 and on a Cray T3E a Cray T3E (512 nodes at 450 MHz) using the native MPI implementation. We measured the efficiency of our library using a ping test and compared the achieved latencies and bandwidths of MPI and TPO.

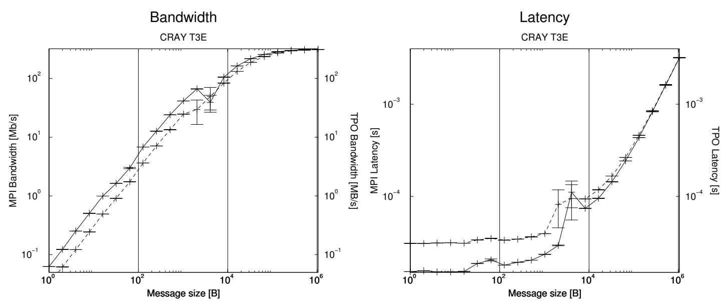

Fig. 1. Comparison of MPI (solid curves) and TPO (dashed curves) on a Cray T3E.

As shown in Figure 1 communication using TPO achieves the same bandwidth as MPI for messages larger than approximately 16 KB. The loss in bandwidth below this size is mainly due to the increased latency. Latencies of MPI and TPO converge as messages are getting larger. For small messages TPO shows a constant latency overhead of $15\mu s$ compared to MPI.

4 Conclusions

We have presented our implementation of an object-oriented message-passing system. It 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 and type-safety to message-passing while preserving MPI semantics and naming conventions as far as possible. This simplifies the transition from existing code. 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. An object-oriented interface can be implemented with almost identical performance compared to MPI. The library is designed as a base for parallelizing scientific applications in an object-oriented environment.

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. International Standards Organization. Programming languages – C++. ISO/IEC publication 14882:1998, 1998.
4. D. Kafure and L. Huang. mpi++: A C++ language binding for MPI. In *Proceedings MPI developers conference*, Notre Dame, IN, June 1995.
5. D. Kafure and L. Huang. Collective communication and communicators in mpi++. Technical report, Department of Computer Science Virginia Tech, 1996.
6. 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.
7. Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997.
8. 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.