GOOMPI: A Generic Object Oriented Message Passing Interface

Design and Implementation

Zhen Yao, Qi-long Zheng, and Guo-liang Chen

National High Performance Computing Center at Hefei
Department of Computer Science and Technology
University of Science and Technology of China
Hefei, Anhui 230027, China
yaozhen@ustc.edu qlzheng@ustc.edu.cn glchen@ustc.edu.cn

Abstract. This paper discusses the application of object-oriented and generic programming techniques in high performance parallel computing, then presents a new message-passing interface based on object-oriented and generic programming techniques — GOOMPI, describes its design and implementation issues, shows its values in designing and implementing parallel algorithms or applications based on the message-passing model through typical examples. This paper also analyzes the performance of our GOOMPI implementation.

1 Introduction

One of the most important distinction between parallel computing and normal sequential computing is the complexity and diversity of parallel computing models. From the view of programming, there are three main parallel computing models — the data-parallel model, the shared-variable model and the message-passing model [1].

The data-parallel and shared-variable models are concise and intuitive in programming, however, they often require tight-coupled parallel computers based on shared memory, e.g. parallel vector machines or SMPs, while it is difficult to implement them directly and efficiently on more popular architectures based on distributed memory, such as MPPs, COWs and SMP clusters.

The message-passing model is more intuitive and easy to implement on parallel computing environments based on distributed memory.

Typical message-passing libraries include the Parallel Virtual Machine (PVM) [2] and the Message Passing Interface (MPI) [3]. Both of them can be easily implemented in homogeneous or heterogeneous distributed environments, and so became prevailing.

Parallel programs using message-passing libraries are often able to gain a fairly more excellent performance than that using other models through elaborate handcrafted optimizations on messages sending and receiving between parallel processes.

However, coding a parallel program based on message-passing is more difficult than that based on data-parallel or shared-variable models. It often takes people much time to deal with the part of a program concerning with messages. When handling only messages of primitive data types or their arrays, the effort is still acceptable. However, the effort becomes far more considerable and the complexity of the message-passing part of the program increases significantly when passing complex dynamic data structures such as binary trees, graphs, or large sparse matrices stored in orthogonal lists. It then becomes difficult to guarantee the correctness and efficiency of the program.

Worst of all, the complexity of the message-passing part always obscures the structure of the algorithm and the program itself. These programs tend to fall into implementation details and lose their abstraction and genericity. It also causes great obstacle in reading, maintaining and extending parallel programs.

In one word, it is difficult to map a parallel algorithm into a message-passing based parallel program rapidly and intuitively. It is the problem that the paper tries to solve.

In the last decade, object-oriented (OO) techniques gain great success in program and software system construction. As a complement paradigm to OO, generic programming techniques aided with inlining and template metaprogramming gain genericity and extensibility through compile time *parametric polymorphism*. The C++ Standard Template Library (STL) [5] is a milestone of generic programming techniques.

Many works have been doing on applying OO and generic programming techniques to HPC areas, such as POOMA [6], Janus [7] and HPC++ [8], etc. The GOOMPI presented in this paper also takes advantage of those techniques, trying to provide programmers a unified generic message-passing interface, effectively simplifying the development process of parallel programs and dramatically improving their abstraction, genericity as well as flexibility.

The rest sections of this paper are organized as follows: in Sect. 2, a full discussion on our GOOMPI is presented with some design policy and implementation detail; then in Sect. 3 we outline a well known matrix multiplication example using the GOOMPI; also the performance evaluation of GOOMPI compared to normal MPI is given in Sect. 4; in the last two sections, we show some issues about the related work and provide our conclusions on what we have experienced.

2 The Design and Implementation of GOOMPI

2.1 The Layered Structure of GOOMPI

As an attempt of applying OO and generic programming techniques to high performance parallel computing, we developed a generic object-oriented message-passing library — GOOMPI. It adopts MPI, a currently widely used library in high performance parallel computing areas, as its underlying implementation basis. By using OO and generic programming techniques, it constructs a generic

high performance message-passing framework to effectively support the transfer of user-defined dynamic data structures of arbitrary complexity.

GOOMPI provides a complete framework for message-passing. It includes a set of well-designed interfaces and class hierarchies, and consists of two layers — the serialization layer and the message-passing layer.

The layered communication architecture of GOOMPI is depicted in Fig. 1. Further explanation is presented in the following sections.

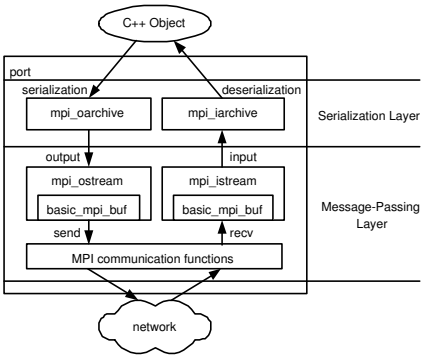


Fig. 1. Layered Communication Architecture of GOOMPI

2.2 Message-Passing Layer

The first layer of GOOMPI, the message-passing layer, uses the Iostream Library to abstract the underlying MPI communication functions and isolate its implementation details.

The iostream library is an important part of the C++ standard library. Its architecture is efficient and excellent for extension.

The iostream library also separates I/O operations into two functional layers — the formatting layer and the transferring layer. In order to effectively perform message-passing using MPI under the architecture of iostream, we write a new transferring layer by deriving a `basic_mpi_buf` class from `std::basic_streambuf`. It supports messages of both the XDR format [10] (for heterogenous environments) and native format (for homogenous environments). It follows the buffered iostream manner, and also provides optional ability to do communication without extra copies to or from buffers to make efficient large bulk of consecutive data transfer possible. Besides improved performance, it also results in more scalability. Unlike some implementations of MPI, our library supports transferring messages of *arbitrary size* due to the extensible architecture of the iostream library, and this requirement is common in scalable high performance scientific computing.

Besides standard-mode send and receive, MPI provides many other communication modes such as buffered mode, synchronous mode and ready mode, as well as nonblocking communications and a variety of collective communications (broadcast, scatter, gather, reduce, all-to-all, etc.). We choose to adopt policy-based design strategy [11] to support these variations without code duplication or loosing efficiency. Actual communication operations are encapsulated uniformly in operation policies as template parameter of `basic_mpi_buf`.

GOOMPI provides several pre-defined communication operation policies: `send_recv`, `isend_recv`, `bcast`, `scatter`, `gather`, `reduce`, `all_to_all`, etc. Each policy has two member functions:

```
void send(const void* addresses[], std::size_t sizes[], int nplayers = 2);
void recv(      void* addresses[], std::size_t sizes[], int nplayers = 2);
```

For example, policy `send_recv` implements these two member functions using `MPI_Send` and `MPI_Recv`, while `bcast` using `MPI_Bcast` correspondingly.

Two classes `mpi_ostream` and `mpi_istream` derived from `std::basic_ostream` and `std::basic_istream` respectively wrap up the `basic_mpi_buf` class to provide a convenient stream interface.

2.3 Serialization Layer

In order to support message-passing based on arbitrary data types, MPI does provide a mechanism to facilitate user-defined types by using data structures like `MPI_Datatype` and functions such as `MPI_Address`, `MPI_Type_struct` and `MPI_Type_commit`. However, it is tedious and cumbersome to use. More unfortunately, it still limited to manipulate simple Plain Old Data (POD) types such as C-style `struct`s.

We introduce necessary serialization and deserialization of any objects. That is, when sending object of any type, the message content includes the object memory layout which is converted into a specific stream with some format; when receiving, the information extracted from the message forms a stream of that format, then a copy of the original object can be easily reconstructed.

We choose the Boost Serialization Library (BSL) [9] as the implementation basis of GOOMPI's serialization part. BSL supports noninvasive serialization. It can easily serialize primitive types as well as STL containers. BSL exploits a layered design approach, taking a stream as a parameter of its archive class to specify the actual storage and transmission of serialized objects. The archive itself only concerns with the serialization-related issues while cares nothing about how to store or transfer the serialized objects. This design makes it convenient for us to combine BSL with MPI's message-passing functionality through our MPI streams.

GOOMPI customized two high performance archive classes, `mpi_oarchive` and `mpi_iarchive` to serialize and deserialize objects of POD types as well as non-POD types (for example, which have nontrivial constructor / destructor) and transfer them through MPI streams. However, under homogenous environments, extra optimizations are provided for commonly used POD types such as

C style `struct`s, arrays of POD types, and even `std::vector`s with POD type elements. Especially, for large arrays and `std::vector`s of POD types, unnecessary copy operations to or from buffers are avoided. These optimizations result in high performance and little abstraction overhead of GOOMPI programs.

2.4 Stream Style User Interface

Finally, GOOMPI provides a `port` class as a *facade* [12] incorporating all the classes above, and exposes a concise iostream style interface for message-passing.

We borrow from OOMPI the notion of port. A port represents a communication channel between parallel processes. Any C++ objects with appropriate serialization support or data of primitive types can be transferred as messages through the channel.

Users of GOOMPI need not to know the internals of a port. However, one may want to customize some of the behaviors of a port before using it.

There are three aspects of port's communication behavior:

1. The communication operation of the port can be blocking or nonblocking, point-to-point or collective. It can be implemented by choosing appropriate communication policy of `basic_mpi_buf`.
2. The port can be an input-only, output-only as well as supports both input and output.
3. The port can transfer any type of data or only messages of specific data types, or in specific sequence.

Accurately, the last two aspects are *restrictions* on the behavior of a port. Restrictions do not always mean limitations. In fact, it is helpful for detecting logical errors of a program at compile time or runtime by imposing restrictions on the `port` class. In specific cases, useless functions can be removed from a port by the programmer from the beginning to eliminate the possibility of making mistakes. For example, performing an input operation on a output-only port will cause compile time error immediately.

Programmers can also choose to apply no restriction on their port classes for more functionality or just for convenience. In the latter case, they also lose the opportunity for the compiler to help detecting program errors earlier at compile time.

We use policy-based method again to design the `port` class. There are three policies correspondingly:

Operation Policy is the same as the operation policy of MPI streams and `basic_mpi_buf`. It specifies blocking or nonblocking, point-to-point or collective communication operations. The default is standard mode point-to-point operation.

Direction Policy specifies communication direction of the port to be `in`, `out` or `inout`. The default is `inout`. Attempts to communicate in invalid direction will be detected at compile time.

Message Type Policy specifies the allowed data types to be transferred by the port. The allowed types can be specified conveniently using a mechanism called *Type Patterns* which is developed by authors of this paper. (For example, pattern `MyClass` means the port can only transfer messages of a type named `MyClass`; pattern `seq_type(A, B, C)` guarantees the port can transfer messages of type `A`, `B` and `C` in sequence; while `set_type(A, derived(B))` means types `A` or all types derived from `B`. Leave it blank means any type is allowed.

The formal definition of `port` is as follows:

```
CommChannel     ::= port<OperationPolicy, DirectionPolicy,
                         MsgTypePolicy>
OperationPolicy ::= P2PComm | CollectiveComm
P2PComm         ::= send_recv | isend_recv | ...
CollectiveComm  ::= bcast | scatter | gather | all_to_all | ...
DirectionPolicy ::= inout | in | inout
MsgTypePolicy   ::= TypePattern
TypePattern     ::= TypeName | any_type | set_type(TypeList)
                  | seq_type(TypeList) | derived(TypeName) | ...
```

For example, programmers can define an output port which broadcasts messages of type `Matrix` like this:

```
goompi::port<bcast, out, Matrix> p(...);
```

At most time, a common port can be defined as follows, if one simply wants to use an ordinary point-to-point communication:

```
goompi::port<> p(...);
```

Such a port uses all its default polices.

2.5 Other Features of GOOMPI

Besides communication, GOOMPI also provides other useful generic components to facilitate parallel programming:

Virtual Topology of Parallel Tasks GOOMPI presents several components to support virtual topologies of parallel tasks. For example, class templates `mesh_view`, `graph_view` and `tree_view` represent Descartes space topologies of any finite dimensions, ordered tree and generic graph structures respectively. They all provide convenient functions to specify neighbors (such as the `left`, `right`, `up` and `down` neighbors, etc.) of each task or particular task groups (for instance a particular `row`, `column` or `block` of tasks). Users can also extend existing topologies or define new topologies when necessary.

Parallel I/O Real world parallel applications always have to deal with a large amount of data, for example, very large matrices or vectors. Storing and retrieving such data efficiently in parallel is a practical requirement. GOOMPI provides C++ iostream style components to support parallel I/O. Large objects can be serialized and deserialized using GOOMPI's parallel I/O streams.

3 Case Study: Implementing Cannon's Algorithm for Matrix Multiplication with GOOMPI

We choose to implement a classic parallel algorithm — the Cannon's algorithm for Matrix Multiplication [13] to illustrate the usage of GOOMPI. To represent matrix data structures, we make use of the Matrix Template Library (MTL) [20], which is an excellent C++ template library that supports a variety of matrix representations as well as many linear algebra functionality. Owe to the extensibility of generic programming, it is convenient to integrate MTL with GOOMPI to effectively divide and transfer various matrices data types.

The GOOMPI program exploits a master / worker paradigm. The master scatters matrices to all worker tasks (including itself), after SPMD style computation, the result is eventually gathered by the master. Suppose there are $P * P$ parallel tasks, `torus_view<2>` is used for representing the 2D-torus topology of these tasks. Source code of the master is as follows:

```
template <typename MatrixA, typename MatrixB, typename MatrixC>
void cannon(const MatrixA& A, const MatrixB& B, MatrixC& C) {
  torus_view<2> self(P, P);          // global 2D-torus view of P*P tasks
  port<scatter> q(self);
  port<gather>  r(self);

  blocked_view<MatrixA> VA(A, P, P); // create blocked views of matrices
  blocked_view<MatrixB> VB(B, P, P);
  blocked_view<MatrixC> VC(C, P, P);

  q << VA << VB;                               // scatter VA and VB

  cannon_worker<MatrixA, MatrixB, MatrixC>(); // act as a worker

  r >> VC;                                     // gather the result
}
```

The following is the source code of workers:

```
template <typename MatrixA, typename MatrixB, typename MatrixC>
void cannon_worker() {
  torus_view<2> self(P, P);        // global 2D-torus view of P*P tasks
  port<scatter> q(self);
  port<gather>  r(self);

  MatrixA a;                       // local submatrices of A and B
  MatrixB b;

  q >> a >> b;                     // receive submatrices from master

  MatrixC c(a.nrows(), a.ncols()); // local result
```

```
  port<isend_recv> s;              // nonblocking send to avoid deadlock

  if (self.i) {                    // initial alignment
    s(self.left(self.i))  << a;
    s(self.right(self.i)) >> a;
  }
  if (self.j) {
    s(self.up(self.j))    << b;
    s(self.down(self.j))  >> b;
  }

  for (int i = 0; i < P; i++) {
    mtl::mult(a, b, c);            // c += a * b
    s(self.left())  << a;          // cyclic left shift a
    s(self.right()) >> a;
    s(self.up())    << b;          // cyclic up shift b
    s(self.down())  >> b;
  }

  r << c;                          // send back result to master
}
```

4 Performance Comparison of MPI and GOOMPI

We tested the performance of LAM MPI [15] and GOOMPI on a 16-node PC Cluster connected with Ethernet. The result is presented in Fig. 2.

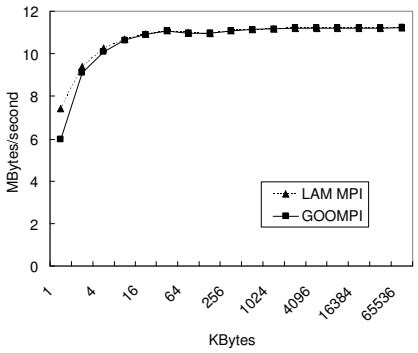
Fig. 2 (a),(b) and (c) show that on trasferring arrays or `std::vector`s of primitive data types or POD `struct`s using both point-to-point and collective communication operations (such as broadcast), there is almost no abstraction penalty. GOOMPI shows a notable performance on par with MPI. Fig. 2 (d) suggests that GOOMPI is much faster when transferring a doubly linked list.

In fact, GOOMPI is especially good at supporting any irregular dynamic data structures. In many situations where complex data structures can not be well fit into primitive types or C arrays, GOOMPI allows more natural and intuitive representations, while corresponding MPI program, is tedious and clumsy. For example, it is boring to reconstruct pointers in dynamic data structures explicitly.

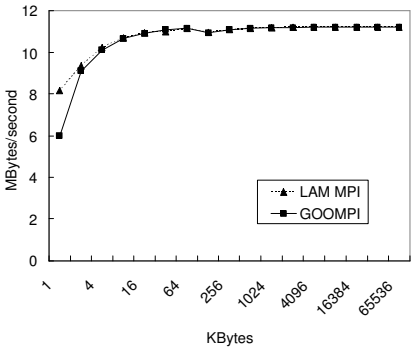
5 Related Work

MPI-2 [4] does have a C++ binding, unfortunately they are just simple class wrappers of MPI C functions. It does not support full object-oriented or generic programming paradigm.

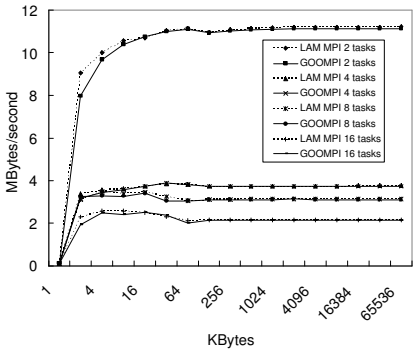
A similar work to GOOMPI is the Object Oriented MPI (OOMPI). It is an object-oriented approach to MPI. It supports messages composed of user-defined



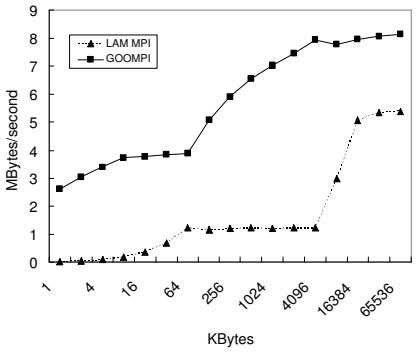
(a)P2P: array/`vector` of `int`



(b)P2P: array/`vector` of a `struct`



(c)Broadcast: array/`vector` of a `struct`



(d)P2P: doubly linked list

Fig. 2. Communication Performance Comparison of LAM MPI and GOOMPI

types which are derived from a common base class called `OOMPI_User_type`, with specific data members and default constructors defined in them. This invasive approach implies that it is impractical to pass messages based on STL containers or classes from other existing libraries using OOMPI.

The Generic Message Passing Framework (GMP) [16] [17] is an attempt which follows generic programming techniques to present a single message-passing programming model for SMP clusters. The GMP provides a brand-new message-passing library which is similar to MPI's message-passing part, with optimizations for communications between threads within a SMP node.

6 Conclusion and Future Work

GOOMPI makes full use of object-oriented and generic programming techniques to support passing messages based on user-defined dynamic data structures in C++. It has many advantages:

Abstraction Programmers using GOOMPI are able to pay more attention to the algorithms and overall structure of parallel programs, instead of running into the boring and error-prone implementation details of message packing / unpacking and sending / receiving.

Extensibility By supporting noninvasive serialization of user-defined types, it can be used in conjunction with C++ STL as well as other libraries (such as MTL and the Boost Graph Library (BGL) [21]) easily.

Efficiency In virtue of generic programming techniques, there is almost no overhead introduced by abstraction. Programs written in GOOMPI have a performance on par with or even better than that of their MPI counterpart written in languages such as C or FORTRAN.

Standard Conforming GOOMPI does not depends on language extensions. Further, it adopts a standard iostream-style interface for message-passing. The implementation of MPI streams and parallel I/O streams all follow the layered design strategies of standard stream classes. Hence GOOMPI is able to support passing messages of arbitrary size.

Type Safety Message type checking policy in GOOMPI enables type checking at both compile time and runtime. This reduces the possibility for programmers to make mistakes on message-passing. GOOMPI programs are likely to be more robust and less error-prone.

With the help of GOOMPI, programmers can map parallel algorithms into high quality parallel programs intuitively, rapidly and effectively. GOOMPI is also of help in the design of parallel algorithms.

Future work includes further enhancements and optimizations of GOOMPI. We are considering about building a generic library and framework based on GOOMPI to support generic parallel and distributed data structures, and facilitate the parallelization of existing generic libraries such as MTL, Boost.uBLAS [18] and Blitz++ [19], etc.

We also prepared to integrate GOOMPI with a thread-level lightweight parallel library called Parallel Multi-Thread Library (PMT) developed by the authors to provide both a unified message-passing model as well as a unified data-parallel model for aiding parallel algorithm design and implementation on different parallel architectures.

References

1. Chen, G.: Parallel Computing — Structure, Algorithm and Programming. High Education Press. (1999) 310-318
2. PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm/pvm_home.html
3. Message Passing Interface Forum: MPI: a message-passing interface standard. International Journal of Supercomputer Applications, 8(3/4), (1994)
4. Message Passing Interface Forum: MPI-2: Extensions to the Message-Passing Interface. (1997) http://www.mpi-forum.org/docs/mpi-20-html/mpi2-report.html
5. Stepanov, A., Lee, M.: The Standard Template Library. HP Technical Report HPL-94-34. (1995)
6. POOMA. http://www.codesourcery.com/pooma/pooma
7. Gerlach, J.: Generic Programming of Parallel Applications with Janus. Parallel Processing Letters, Vol. 12, No. 2 (2002) 175-190
8. Johnson, E., Gannon, D.: HPC++: Experiments with the Parallel Standard Template Library. International Conference on Supercomputing Proceedings of the 11th international conference on Supercomputing. (1997) 124-131
9. Ramey, R.: The Boost Serialization Library. http://www.boost.org/
10. Sun Microsystems, Inc.: XDR: External Data Representation standard. RFC 1014. (1987)
11. Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wisley. (2001) 3-21
12. Gamma, E., Helm. R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. (1994)
13. Cannon, L. E.: A Cellular Computer to Implement the Kalman Filter Algorithm. PH.D. thesis. Montana State Univ. (1969)
14. Object Oriented MPI (OOMPI). http://www.osl.iu.edu/research/oompi/
15. LAM MPI. http://www.lam-mpi.org/
16. Lee, L., Lumsdaine, A.: The Generic Message Passing Framework. Parallel and Distributed Processing Symposium, 2003. Proceedings. International. (2003) 53-62
17. Lee, L.: Generic programming for high-performance scientific computing. Ph.D. thesis. University of Notre Dame. (2003) 1-78
18. Walter, J., Koch, M.: The Boost uBLAS Library. http://www.boost.org/
19. The Blitz++ Library. http://oonumerics.org/blitz/
20. Lumsdaine, A., Siek, J., Lee, L.: The Matrix Template Library. http://www.osl.iu.edu/research/mtl/
21. Siek, J., Lee, L., Lumsdaine, A.: The Boost Graph Library (BGL). http://www.boost.org/