The Generic Message Passing Framework

Lie-Quan Lee and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
llee@osl.iu.edu lums@osl.iu.edu

Abstract

This paper presents the design and implementation of the Generic Message Passing framework (GMP). The design of the GMP is guided by two goals: programmer productivity and run-time performance—particularly for parallel systems having both shared and distributed memory (e.g., clusters of SMPs). To achieve these goals simultaneously, the GMP uses a generic programming paradigm, allowing it to integrate seamlessly with modern generic libraries such as the C++ Standard Library. In addition, the GMP presents a single parallel programming model, namely message passing. The implementation of the GMP fully exploits the architectural characteristics of its execution target for maximum run-time performance. Performance results are presented that demonstrate the effectiveness of the Generic Message Passing framework.

1 Introduction

Clusters of SMPs have been widely deployed for cost-effective high-performance parallel computing. Studies of parallel programming on such clusters are of importance and being carried out [2–6, 14–16, 22, 25, 27, 32, 37]. A cluster of SMPs is characterized by (relatively) fast hardware memory for intra-node communication between CPUs, while inter-node communication is performed by commodity network devices. In terms of effectively using such clusters, there are two issues to address. The first is how to most efficiently run a job on a cluster of SMPs. In other words, what is the best execution model. The second one is how to most effectively program for such clusters, or, what is the best programming model.

Because communication can occur through shared data in the same memory address space, it is typically most efficient to execute applications with multiple threads on a single SMP. Correspondingly, it is typical to use threads, OpenMP, or parallelizing compilers as shared-memory programming models for an SMP. Between nodes in a cluster, on the other hand, it is typically most efficient to use a message passing approach between separate processes. Application developers use explicit message passing to communicate and to provide synchronization among those processes. Thus, an ideal execution model for a cluster of SMPs would involve multiple threads on the same SMP and separate processes on the different SMPs.

For programming an SMP cluster, there are three basic programming models to consider: pure shared-memory, pure message-passing, or a combination of the two.

Using a shared memory programming model with distributed memory hardware requires the implementation of a single memory address space via software distributed shared-memory. However, results from previous studies indicate that software distributed shared-memory is inefficient and often suffers from scalability problems when compared to message passing counterparts [25–27]. One important reason that performance is lost with a software shared-memory system is that application programmers are not able to fully control (and exploit) data locality.

A second possibility is to incorporate both message-passing and shared-memory in the same programming model for clusters of SMPs. This approach is able to exploit both intra-node and inter-node parallelism and would seem to have good potential for high levels of performance. Recently, researchers have combined the Message Passing Interface (MPI) [34] together with OpenMP in the same applications. However, the actual run-time performance gain of using this mixed programming model is questionable as discussed in [4, 5, 15, 20, 33]. In many cases, the pure MPI versions run faster than equivalent mixed versions. Moreover, mixing programming approaches in the same application is undesirable from a software engineering standpoint.

We believe that it is unnecessary to have both shared-memory and message-passing co-existing in the programming model. In contrast to previous approaches based on software distributed-shared memory to provide a single programming interface, we advocate a message-passing based...
programming model. With such a model, we are still able to exploit both intra-SMP and inter-SMP communication characteristics for execution on a cluster of SMPs. At the same time this approach simplifies the complexity of the mixed-mode of both message-passing and shared-memory synchronization in application development. Note that we are not advocating a pure MPI-based approach when we speak of a uniform message-passing approach. As we will see, there are some inherent performance limitations to MPI in shared-memory hardware settings—limitations that our system addresses.

Another shortcoming of existing parallel programming systems is the lack of support for modern data structures. Although MPI includes a C++ binding, its one-to-one syntactic mapping from C (equivalently, Fortran), use of void* buffers, and explicitly specified data types provides little substantial support for data abstraction. Object-oriented approaches of various kinds have been developed to raise the level of abstraction for parallel programming [42]. In addition, our previously-developed Object-Oriented MPI (OOMPI) library provides a direct high-level object-oriented encapsulation of MPI [31]. However, there are well-known drawbacks to object-oriented programming, particularly with respect to performance.

More recently, generic programming has emerged as a programming paradigm that can provide high levels of abstraction and high levels of performance [1]. The effectiveness of the generic programming approach has been demonstrated with libraries such as the C++ Standard Library [18, 35] and the Matrix Template Library (for high performance computing) [30].

In this paper, we present the design and implementation of the Generic Message Passing (GMP) framework. The GMP provides a uniform message passing interface for both inter-process communication and intra-process communication while using threads for execution entities. It has the flexibility of running either on a SMP or a cluster of unprocessors or a cluster of SMPs with maximal or near maximal performance. It also provides the ability to efficiently communicate C++ objects with complex data structures.

2 The Interface of Generic Message-Passing

A GMP application consists of a group of execution entities. Each entity is identified with unique ID from zero to size-1, where size is the total number of execution entities in the application. Communication calls made by each execution entity are properly dispatched to intra-process or inter-process operations at run time. GMP provides point-to-point communication, zero-copy communication, and collective communication operations. The current version of GMP supports a single communication space.

2.1 Point-to-Point Communication

The basic point-to-point communication operations are send () and receive (). The syntax for these operations is given below.

```cpp
template<typename T>
void send(rank_type d, T& data, tag_type tag);

template<typename T>
void recv(rank_type s, T& data, tag_type tag);

template<typename Iterator>
void send(rank_type d, Iterator f, Iterator l, tag_type tag);

template<typename Iterator>
void recv(rank_type s, Iterator f, Iterator l, tag_type tag);
```

Here, type T and the value type of Iterator must conform to the concepts of Serializable and GMPCopyable, which will be discussed in detail later.\(^1\) The iterator versions of send () and receive () operate on data in the range [f, l]. The use of iterators allows us to have a generic mechanism for expressing different ranges of data. For example, we can send and receive ranges of strided or indexed data—or we can send and receive data in completely general data structures such as linked lists.

The semantics of the previous operations are blocking; after the communication operation returns, the data structure can either be re-used (in the case of send () ), or the data has been received (in the case of receive () ). GMP also provides nonblocking and persistent point-to-point communication operations.

2.2 Zero-Copy Communication (Ownership Transfer)

The semantics of MPI communication requires that every point to point operation perform least one data copy. That is, between a send operation and its corresponding receive operation, the data must be copied between the send and the receive buffers—even between entities sharing the same memory space. As discussed in [21], complicated zero-copy techniques, such as page remapping with copy-on-write [7, 8, 40], can be used to eliminate the copy. However, such an approach requires substantial support from the underlying OS, whereas we seek a portable means at the library layer of eliminating unnecessary copies.

GMP provides zero-copy communication (ownership transfer) to specialize communication for pointer-based data. The semantics of zero-copy communication is similar to that of the C++ facility std::auto_ptr [12, 36]. In a zero-copy communication, the sender yields its ownership of the pointer data to the receiver. After ownership is transferred, the pointer data cannot be accessed inside the sender and is available only in the receiver. This approach provides

\(^1\)In the parlance of generic programming, a concept is a set of interface requirements for types.
a performance optimization for intra-process communication. Note also that we can overload `send` and `receive` to perform zero copy communication, so that the calling interface is the same as for regular point-to-point communication. The syntax of ownership transfer is given below.

```cpp
template <typename T>
void send(rank_type dst, auto_ptr<T> ptr, tag_type tag);
```

```cpp
template <typename T>
void recv(rank_type src, auto_ptr<T>& ptr, tag_type tag);
```

```cpp
template <typename T>
void send(rank_type dst, array<T> array, tag_type tag);
```

```cpp
template <typename T>
void recv(rank_type src, array<T>& array, tag_type tag);
```

Here `std::auto_ptr` is a C++ standard facility which owns a dynamically allocated pointer, transfers the ownership of the pointer in copy and assignment, and automatically deallocates the data when it is no longer needed. The facility `auto_array` in GMP is similar to `std::auto_ptr` except that it owns a dynamically allocated pointer to array.

### 2.3 Collective Communication

The GMP provides barrier, broadcast, scatter, gather, all gather, all-to-all personalized communication, reduce, and all reduce collective operations. In the collective communication interface, it is convenient to use `Range` and `NRange` concepts. A range can be represented by an iterator pair `[first, last)`. `NRange` is a generalization of the `Range` concept. That is, an `NRange` includes N ranges, namely, a range of ranges. For example, a scatter operation from `root` sends range `i in root` to execution entity `i`. An all-to-all operation exchanges data of range `i in entity j with range j in entity i` for all the execution entities. The ranges in an `NRange` object are not required to have the same length. Thus, the vector versions of collective operations have the same interface as those of the normal versions.

The following are the collective communication operations in GMP. Note that vector versions of operations (e.g., the equivalent of `MPI_Gather` ) are implicitly included by virtue of the `NRange` concept in the interface.

```cpp
void barrier();
```

```cpp
template <typename T>
void broadcast(rank_type root, T& data);
```

```cpp
template <typename Iterator>
void broadcast(rank_type root, Iterator first, Iterator last);
```

```cpp
template <typename T, typename CommutableOp>
void reduce(rank_type root, const T& input, T& result, CommutableOp op);
```

```cpp
template <typename Iterator, typename CommutableOp>
void reduce(rank_type root, Iterator first, Iterator last, Iterator result, CommutableOp op);
```

### 2.4 Object Serialization

One of the technical difficulties in a C++ message-passing system is how to send objects with complicated data structures. Often data in objects are not linear and may encapsulate other objects of complicated types.

The GMP serialization strategy is based on providing usable default serialization functionality while allowing extensibility to allow users to exploit particular characteristics of their data types. The `serialize` and `deserialize` functions are generic functions that operate on data types modeling the `Serializable` concept. That is, if type `data_type` models `Serializable`, the following expressions must be valid:

```cpp
serialize(msg, data); //data_type data
deserialize(msg, data);
```

Here, `msg` is a GMP implementation dependent object.

We note that this approach does not require users to supply `serialize` and `deserialize` functions. GMP provides implementations for all built-in primitive types, iterator range, containers and container adaptors in the C++ Standard Library, as well as for those types with trivial copy constructors. For example, `std::vector<T>` has already conformed to `Serializable` concept as long as its element type `T` has. We will discuss how to supply those two functions for user-defined types in the next section.

As mentioned in the beginning of this section, each data type must also model `GMPCopyable` in order to utilize the GMP interface. The requirement of `GMPCopyable` is to have `fast_copy` function defined. For example, the following expression must be valid if `data_type` models `GMPCopyable`:

```cpp
fast_copy(src, dest); //data_type src, dest
```

All types with accessible copy assignment operator model `GMPCopyable` concept since GMP defines a function template `fast_copy` which is implemented by assignment. The rationale behind the `GMPCopyable` is to allow
users to provide the \texttt{fast\_copy()} function if that can perform a copy faster than default assignment.

3 The Implementation of GMP

3.1 Message-Passing System

Intra-process Point-to-Point Message-Passing GMP uses two-dimensional channels for intra-process message-passing. A message from thread \(i\) to thread \(j\) uses channel \((i,j)\); channel \((i,j)\) is distinct from channel \((j,i)\). Each channel has a fixed-length internal buffer used for the short message protocol. That is, if the message size is less than a certain compile-time constant, the message is copied into the channel buffer on sending and copied from the channel on receiving. Otherwise, the channel only contains the minimal information of source data (for example, address of data) and the receiver will directly copy from source to destination.

\texttt{gmp::fast\_copy()} is used to copy data in intra-communication. By default, \texttt{gmp::fast\_copy()} for all built-in types and their array data are implemented by \texttt{memcpy()} for performance optimization. In addition, users can supply \texttt{gmp::fast\_copy()} for user-defined data types. If the user does not supply a custom \texttt{gmp::fast\_copy()} function, the generic \texttt{gmp::fast\_copy()} template is provided, in which the assignment operator of the type used.

Ownership Transfer Intra-process ownership transfer utilizes the intra-process message-passing channels. The sender puts the address of the pointer in \texttt{auto\_ptr} or \texttt{auto\_array} data (and length if necessary) into the corresponding channel and resets \texttt{auto\_ptr} or \texttt{auto\_array}. The receiver will get the pointer and assign it into the \texttt{auto\_ptr} or \texttt{auto\_array} data. Therefore, there is no memory copy involved and the communication time of ownership transfer is the latency of normal intra-process communication. Inter-process ownership transfer is implemented by normal inter-process communication.

Ownership transfer is distinct from the zero-copy schemes discussed in [7]. In our situation, the intra-SMP communication occurs between threads in the same process. Although complicated schemes such as page remapping with copy-on-write can be used for zero-copy transfers, in order to provide this capability portably and at the library layer, GMP uses pointer passing. Ownership transfer is similar to the semantics of C++ facility \texttt{auto\_ptr}. The pointer data are invalid in the sender after ownership transfer is complete—just as the \texttt{auto\_ptr} object is invalid after it is assigned to another \texttt{auto\_ptr} object. As with \texttt{auto\_ptr}, it is the user’s responsibility to avoid aliases.

Inter-process Point-to-Point Message-Passing If a sender and receiver are not in the same process, communication crosses process boundaries. The sender serializes objects and locks the connection to protect potential harmful interlacing of two messages and sends the serialized data. The send operation is asynchronous. That is, there is no need of a matching receive operation in order to start or complete a send operation.

Each process has a dedicated internal thread that polls for data from a set of existing point-to-point connections. The thread does the following in the loop. It tests whether there are data to read from a set of connections. It blocks on these connections for certain period (for lazy connection) or without timeout (for eager connection) if there is no incoming data. Otherwise, it reads data and puts the data into the corresponding message queue.

Each process has an array of message queues, one for each process. The receiving operation is to look through the message queue with response to sender and message tag, de-queue the message if found, and deserialize the message.

Collective Communication A collective communication in GMP will, in general, mix inter-process and intra-process operations. Inter-process parts of collective operations use different connections than point-to-point connections. All processes are connected in a tree topology for the collective connections. Inter-process parts of collective operations utilize the tree connection structure. On the other hand, intra-process parts of collective operations make use of the same two-dimensional point-to-point channels.

For example, to broadcast a message, each process delegates a representative thread which will receive the data from its parent process in the tree and send the data to its directly-connected processes. The representative thread then sends the data to all other threads in the same process. Similarly, to gather data, a representative thread in each process will receive data from other processes, gather data from local threads, and send them to proper collective channel in the tree which will reach the root eventually. In this way, the hierarchical communication structure of an SMP cluster is exploited [19].

3.2 Object Serialization and Variable-Length Messages

GMP provides serialization and deserialization for common objects such as built-in types, iterator range, containers and container adaptors in the C++ Standard Library, as well as for those types with trivial copy constructors. Users can build serialization and deserialization for their own C++ data structures very easily based on existing ones.

A traits mechanism [23] is used for object serialization of built-in types and common types in the C++ Standard Library. Based on the actual object category, serializing or deserializing of type \(T\) will be dispatched to different function calls. Explicitly defining \texttt{serialize()} and \texttt{deserialize()} for a data type will bypass the traits mechanism.
This strategy also helps us to efficiently communicate variable-size messages. In MPI, a variable-size message is often handled by using two actual messages: one for the size of the message and the other for the message itself. Our serialization strategy allows us to have one message only. For example, suppose at run time one execution entity wants to send a pointer to an array of double to another. One can define a helper struct `variable_size_helper` and define `serialize()` and `deserialize()` for `variable_size_helper`. Then one execution entity sends a `variable_size_helper` object to another. The receiver will not need to worry about the buffer size since it is allocated inside `deserialize()` or `fast_copy()` with the correct size.

```c
struct variable_size_helper {
    double c;
    int len;
};

//inter-process communication
template <class Message>
void serialize(Message& msg, variable_size_helper& s) {
    gmp::serialize(msg, s.c, s.c + s.c_len);
    gmp::serialize(msg, s.c, s.c + s.c_len);
}

template <class Message>
void deserialize(Message& msg, variable_size_helper& s) {
    gmp::deserialize(msg, s.c, s.c + s.c_len);
}

//intra-process communication
void fast_copy(const variable_size_helper& src, variable_size_helper& dest) {
    dest.c = new double[src.clen];
    memcpy(dest.c, src.c, sizeof(double)*src.clen);
}
```

4 Experiments and Results

4.1 Point-to-Point Communication of Irregular Data Structures

We first compare GMP point-to-point communication with that of MPI by performing ping-pong experiments with a user-defined type.

The message type communicated in the ping-pong experiments is a structure defined as follows:

```c
struct Partstruct {
    int particle_class; // particle class
double d[6]; // coordinates
char b[7]; // additional info
};
```

With MPI, it can be quite complicated and error-prone to create a derived MPI data type. Although pre-processing tools have been developed to automate some of this process [10, 11], these tools still require structured comments in the source code.

The following example taken from the MPI standard creates the MPI data type for `Partstruct` objects.

```c
/* build datatype describing structure */
struct Partstruct particle[1000];
MPI_Datatype Particletype;
MPI_Datatype type[3] = {MPI_INT, MPI_DOUBLE, MPI_CHAR};
int blocklen[3] = {1, 6, 7};
MPI_Aint disp[3];
int base;

/* compute displacements */
MPI_Admess(particle, disp);
MPI_Admess(particle[0].d, disp+1);
MPI_Admess(particle[0].b, disp+2);
base = disp[0];
for (int i = 0; i < 3; i++)
    disp[i] += base;
MPI_Type_struct(3, blocklen, disp, type, &Particletype);
MPI_Type_commit(&Particletype);
```

In addition, we will not be able to create a derived data type at all if the structure contains dynamically allocated data (typically accessed through pointers).

The GMP provides a convenient traits mechanism to make a plain-old-data (POD) type conforming to `Serializable` and `GMPCopyable` as described in the previous section. The following specializes the `object_traits` class for `Partstruct`. Thus, we can send and receive `Partstruct` objects as we do for built-in types.

```c
template <>
struct object_traits < Partstruct >
{
    typedef trivial_copy_constructor object::category;
};
```

This approach also insulates the user program from the actual communication mechanism. That is, specifying an alternative means for communications `Partstruct` objects does not require any changes to the user code; only the `object_traits` specialization need be changed.

Figure 1 compares the point-to-point communication results of GMP inside one SMP node with those of MPI. We also plot the memory copy time in the same figures as a reference. As shown in the figures, the GMP curve closely follows that for memory copy. The sub-figures (a) and (c) show that the GMP version was about one order of magnitude faster than the MPI version for communicating irregular data structures. This performance difference is due to the fact that the design of GMP enables users to communicate with GMP by a simple traits mechanism, in conjunction with `memcpy()` , for intra-process communication. On the other hand, the MPI interface does not allow such optimization. We also include the figures for communicating an array of characters in Figure 1.
Figure 1. Performance comparison of point-to-point GMP and MPI. The graphs show communication speed as a function of data size. The sub-figures (a) and (b) show the results on a dual Sun UltraSPARC-III 750 MHz CPU machine with Solaris 2.8. The MPI implementation is LAM-6.5.6 with usys shared-memory optimization. The sub-figures (c) and (d) show the results on a 64-400MHz SMP Sun Enterprise 10000 system where the MPI implementation is SunMPI 4.1.
4.2 Collective Communication

This experiment was performed on a 32-node cluster of dual UltraSPARC CPUs, each having a clock of 400 MHz. Each node is on 100M-baseT switched Ethernet. Sixteen cluster nodes are allocated for this experiment. Each cluster node has two execution entities. Broadcast time is measured as follows: The timer starts at the root before invoking broadcast. The timer stops after a barrier returns at the root. The GMP broadcast results are compared with those of MPI_Beast(). The MPI implementation is LAM-6.5.6 with usysv shared memory optimization.

Figure 2 shows the communication speed of GMP broadcast() and MPI_Beast(). GMP broadcast bandwidth is more than one and half fold of that of LAM MPI_Beast() for large arrays of int. The difference is even bigger for broadcasting irregular data structures. Although both GMP and LAM used a tree-based algorithm for the broadcast operation, the GMP broadcast() sends data through the inter-process tree consisting of one representative thread in each process. Next, the representative thread sends data to local threads in the same process. Thus, the GMP broadcast() is aware of execution entity locality. On the other hand, all processes in the LAM MPI broadcast are in the tree and two processes in the same physical SMP node may not be topological neighbors. This behavior is not a property of MPI per se, however MPI implementations such as LAM are only lately exploiting network topology characteristics for performance.

4.3 Parallel Video Processing Simulation

A parallel video processing simulation was used to compare GMP and MPI. A manager/worker model with two managers and a number of workers is used. One execution entity (the distributor) gets the image frames and distributes slices of each frame to a number of workers. Each worker applies a filter to the frame slice (a 3x3 average filter is used) and sends the processed slices to another execution entity (the collector) which is collecting and assembling slices to frames. The GMP version of the simulation used ownership transfer communication to exchange data between a manager and a worker while the MPI version used standard point-to-point communication. The simulation was run on a 64-CPU SMP Sun Enterprise 10000 machine. In total, 100 frames of 3200 x 3200 images were generated in the distributor. The slice size was fixed at 100.

As shown in Figure 3, the MPI version runs more slowly than GMP with ownership transfer because MPI point-to-point communication requires at least one memory copy as data is transferred from the sender to the receiver (even on an SMP). However, with ownership transfer in GMP, only the data pointer is passed from the sender to the receiver and there is no actual data copy. Figure 3 also indicates that the GMP version scales poorly as the number of workers increases while the GMP version scales well even with 32 workers.

Using a simplified performance model, we are able to explain the performance behaviors of the simulation. Suppose that the time a worker gets a slice of image from the distributor is fixed and is denoted as $T_1$. Similarly, the time to apply the filter for each slice is $T_2$, and the communication time a work sends the processed slice to the collector is $T_3$. When the communication time $T_1$ and $T_3$ are large comparing to processing time $T_2$, the distributor cannot keep up with all 8 workers and each worker has idle time before it processes its next slice as shown in part (1) of Figure 3(b). On the other hand, when the communication time $T_1$ and $T_3$ is small, the distributor may be able to send slices of the image fast enough so that workers are always processing image slices. The condition to keep all workers busy is:

$$(N - 1) \times T_1 < T_2 + T_3$$

where $N$ is the number of workers. The MPI version has a longer communication time because of the memory copy and because workers become idle when the number of workers is larger than 4. On the other hand, GMP ownership transfer consumes very little time (about 5 microseconds on the Sun Enterprise 10000 system).

4.4 Matrix-Matrix Multiplication

The Simple Universal Matrix-Matrix Algorithm (SUMMA), is a parallel algorithm for performing matrix-matrix multiplication [41]. SUMMA decomposes matrices in a two dimensional $m \times n$ mesh. Each portion of data is in one execution entity; there are $m \times n$ execution entities. Each execution entity loops to broadcast a small fixed number of columns of local data of matrix $A$ in the same row mesh, to broadcast a small fixed number (the blocking size) of rows of local data of matrix $B$ in the same column mesh, and then to perform a local matrix-matrix multiplication for updating portion of matrix $C$. The loop ends when all local data are broadcast and $C$ is updated. The broadcast operation in the row or in the column is implemented as a receive operation from its left (or upper) neighbor and a send operation to its right (or lower) neighbor in the execution entity mesh.

We implemented a GMP version of SUMMA and compared its performance to the publicly available MPI version of SUMMA using the cluster described in Section 4.2. The local matrix-matrix multiplication is realized with the Sun performance library function dgemm(). The results are shown in Table 1, where the matrix size was fixed as $6400 \times 6400$ while different decomposition meshes were considered using 16 execution entities. As shown in the table, the GMP version of SUMMA always yields higher aggregate performance than the original MPI code. The ad-
Figure 2. Performance comparison of GMP broadcast() and MPI MPI_Bcast() on a Sun cluster where each node has dual 360MHz UltraSPARC CPUs and is connected on a fast Ethernet switch. Sixteen SMP nodes are used in this experiment. The vertical axis shows aggregate communication speed.

Figure 3. The parallel video processing simulation on a 64-400MHz CPU SMP Sun Enterprise 10000 system. (a) The total execution time versus different number of workers. (b) The performance models of the parallel video processing simulation.
ditional performance comes from better exploitation of the characteristics of intra-process communication in GMP.

### Table 1. Performance comparison of SUMMA implementations.

<table>
<thead>
<tr>
<th>Decomp.</th>
<th>Aggregate MFLOPS</th>
<th>MFLOPS / node</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 x 1</td>
<td>2688.6 (2192.9)</td>
<td>168.0 (137.1)</td>
</tr>
<tr>
<td>8 x 2</td>
<td>2839.2 (2679.5)</td>
<td>177.5 (167.5)</td>
</tr>
<tr>
<td>4 x 4</td>
<td>3009.4 (2673.3)</td>
<td>188.1 (167.1)</td>
</tr>
<tr>
<td>2 x 8</td>
<td>3140.3 (2771.1)</td>
<td>196.3 (173.2)</td>
</tr>
<tr>
<td>1 x 16</td>
<td>2904.4 (2290.3)</td>
<td>181.5 (143.1)</td>
</tr>
</tbody>
</table>

5 Related Work

MPI as a standardized message-passing facility includes a C++ binding. However, the C++ binding is essentially a one-to-one syntactical mapping from the C and Fortran functions that does not accommodate any modern programming features. The type safety of C++ is ignored in the design of the interface of C++ binding. The static memory layout of a user-defined data structure must be known completely at compile time in order to define a MPI derived data type.

OOMPI uses C++ subtype polymorphism to provide support for user-defined data types [31]. Its requirement of inheriting from a single base class as the result of subtype polymorphism is not practical for existing data structures such as containers.

The Amelia Vector Template Library (AVTL) [28], the HPC++ parallel Standard Template Library (PSTL) [17], and the Standard Templates Adaptive Parallel Library (STAPL) [24] extend the Standard Template Library (STL) to have distributed containers or parallel algorithms in the data-parallel manner. The parallel iterator (or pRange in STAPL) provides global access to all elements in the distributed containers by encapsulating communication and facilitates generic parallel programming.

TPO++ [13] provides a way to support data structures in C++ Standard Libraries in point-to-point communication. There is not support for collective communication. In addition, for data structures other than containers, TPO++ requires specified member functions to be defined in order to be supported for transmission. That prevented from re-using existing well-designed and extensively-tested data structures in the 3rd party packages.

TOMPI [9] is an MPI implementation designed for a single workstation. It uses a single Unix process and multiple threads, one for each MPI node. It is designed for the development phase of parallel applications such as testing, debugging, and performance tuning. Its design goal is to have a development tool so that an MPI application can be tested, debugged, or tuned for performance on a single processor.

TOMPI [29, 38, 39] uses programming transformation for MPI applications written in C and maps MPI nodes to threads in run time. It uses node-specific data which is derived from the Thread-Specific Data in POSIX threads. A compiler (or a preprocessor) transforms global and static variables in the original application source code in C programming language to node-specific data which are private to each MPI-node. However, the simplified access model of lock-free queues (one writer and multiple-readers) used in the run-time system limits their system to level-3 MPI thread support. In other words, the user program has to guarantee that MPI functions calls in the same node are serialized. Furthermore, TOMPI does not support communicating user-defined data.

Acknowledgments

This work was supported by NSF grant ACI-9982205 and by a grant from the Lilly Endowment. Some experimental results were obtained using computational facilities at the College of William and Mary which were enabled by grants from Sun Microsystems, the NSF, and the Virginia Commonwealth Technology Research Fund. The authors would like to thank the anonymous referees for helpful suggestions on improvements to this paper.

References

L.-Q. Lee.

S. S. Lumetta, A. M. Mainwaring, and D. E. Culler. Multi-

N. C. Myers. Traits: a new and useful template technique.

T. Grundmann, M. Ritt, and W. Rosenstiel. TPO++: An

D. S. Henty. Performance of hybrid message-passing and

P. Lanucara and S. Rovida. Conjugate-gradients algo-

E. Johnson and D. Gannon. Programming with the HPC++


I. O. for Standardization (ISO). ISO/IEC Final Draft Inter-

N. M. Josuttis. The C++ Standard Library: A Tutorial and

A. Gursoy and I. Cengiz. Mechanisms for programming

A. Skjellum, D. G. Wooley, A. Lumsdaine, and J. M.


J. E. Devaney, M. Michel, J. Peeters, and E. Baland. Au-

J. G. Siek and A. Lumsdaine. The Matrix Template Library:

D. J. Scales, K. Gharachorloo, and A. Aggarwal. Fine-grain

T. J. Sheffler. A portable mpi-based parallel vector tem-

K. Shen, H. Tang, and T. Yang. Optimizing threaded MPI execution on shared memory ma-

R. A. von de Geijn and J. Watts. SUMMA: Scalable universal

S. Satoh, K. Kusano, Y. Tanaka, and M. Matsuda. Paral-

D. S. Henty. Performance of hybrid message-passing and


I. O. for Standardization (ISO). ISO/IEC Final Draft Inter-

N. M. Josuttis. The C++ Standard Library: A Tutorial and

A. Skjellum, D. G. Wooley, A. Lumsdaine, and J. M.

K. Shen, H. Tang, and T. Yang. Optimizing threaded MPI execution on shared memory ma-

R. A. von de Geijn and J. Watts. SUMMA: Scalable universal