

Humboldt-Universität zu Berlin

CORBA and the Message Passing Interface

Response to the Aggregated Computing RFI

Document orbos/99-04-12

1 Overview

This response the Aggregated Computing RFI (orbos/99-01-04) gives an introduction into the Message Passing Interface. Questions about this response should be directed to

Martin v. Löwis
Humboldt-Universität zu Berlin
Rudower Chaussee 5
12489 Berlin
Germany
phone: +49 30 2093 3118
fax: +49 30 2093 3112
email: loewis@informatik.hu-berlin.de

Our response addresses the following items in the RFI:

- 4.2.2 Parallel Processing (APIs for parallel programming)
- 4.2.3 Other Issues (Attractive Environments)
- 4.2.4 Other Standardization Activities (MPI Forum)

Since this document gives only an introduction to the MPI, the reader is referred to the following information sources for more information

- [1] MPI Forum Home Page: <http://www.mpi-forum.org/>
- [2] University of Tennessee. MPI: A Message-Passing Interface Standard. MPI Forum, 1995, <http://www.mpi-forum.org/docs/mpi-11.ps.Z>
- [3] University of Tennessee. MPI-2: Extensions to the Message Passing Interface. MPI Forum, 1997, <http://www.mpi-forum.org/docs/mpi-20.ps.Z>.
- [4] Message Passing Interface Implementations. <http://www-unix.mcs.anl.gov/mpi/implementations.html>

2 The Message Passing Interface

The MPI standard document describes the rationale and history of defining a standard for message passing. In [2], the authors write:

Message passing is a paradigm used widely on certain classes of parallel machines, especially those with distributed memory. Although there are many variations, the basic concept of processes communicating through messages is well understood. Over the last ten years, substantial progress has been made in casting significant applications in this paradigm. Each vendor has implemented its own variant. More recently, several systems have demonstrated that a message passing system can be efficiently and portably implemented. It is thus an appropriate time to try to define both the syntax and semantics of a core of library routines that will be useful to a wide range of users and efficiently implementable on a wide range of computers.

In designing MPI we have sought to make use of the most attractive features of a number of existing message passing systems, rather than selecting one of them and adopting it as the standard. Thus, MPI has been strongly influenced by work at the IBM T. J. Watson Research Center, Intel's NX/2, Express, nCUBE's Vertex, p4, and PARMACS. Other important contributions have come from Zipcode, Chimp, PVM, Chameleon, and PICL.

The MPI standardization effort involved about 60 people from 40 organizations mainly from the United States and Europe. Most of the major vendors of concurrent computers were involved in MPI, along with researchers from universities, government laboratories, and industry. The standardization process began with the Workshop on Standards for Message Passing in a Distributed Memory Environment, sponsored by the Center for Research on Parallel Computing, held April 29-30, 1992, in Williamsburg, Virginia [29]. At this workshop the basic features essential to a standard message passing interface were discussed, and a working group established to continue the standardization process.

A preliminary draft proposal, known as MPI1, was put forward by Dongarra, Hempel, Hey, and Walker in November 1992, and a revised version was completed in February 1993 [12]. MPI1 embodied the main features that were identified at the Williamsburg workshop as being necessary in a message passing standard. Since MPI1 was primarily intended to promote discussion and „get the ball rolling," it focused mainly on point-to-point communications. MPI1 brought to the forefront a number of important standardization issues, but did not include any collective communication routines and was not thread-safe.

In November 1992, a meeting of the MPI working group was held in Minneapolis, at which it was decided to place the standardization process on a more formal footing, and to generally adopt the procedures and organization of the High Performance Fortran Forum. Subcommittees were formed for the major component areas of the standard, and an email discussion service established for each. In addition, the goal of producing a draft MPI standard by the Fall of 1993 was set. To achieve this goal the MPI working group met every 6 weeks for two days throughout the first 9 months of 1993, and presented the draft MPI standard at the Supercomputing 93 conference in November 1993. These meetings and the email discus-

sion together constituted the MPI Forum, membership of which has been open to all members of the high performance computing community.

The main advantages of establishing a message-passing standard are portability and ease-of-use. In a distributed memory communication environment in which the higher level routines and/or abstractions are build upon lower level message passing routines the benefits of standardization are particularly apparent. Furthermore, the definition of a message passing standard, such as that proposed here, provides vendors with a clearly defined base set of routines that they can implement efficiently, or in some cases provide hardware support for, thereby enhancing scalability.

The goal of the Message Passing Interface simply stated is to develop a widely used standard for writing message-passing programs. As such the interface should establish a practical, portable, efficient, and flexible standard for message passing.

<end quote, literature references omitted>

The MPI standard defines an abstract library interface for message passing, and bindings of this interface to major programming language. Coming from the high-performance computing community, MPI 1.1 focused on Fortran and C bindings. MPI 2.0 adds a binding to C++.

One major goal of the specification was to allow efficient, high-performance implementations of the interface, which make direct use of inter-node communication mechanisms in a parallel computer. The interface works equally well in a shared-memory architecture, and on distributed memory systems (in particular on the local area network). Therefore, algorithms written for the MPI are scalable across a wide variety of computing platforms.

High-performance computing platforms today often come with customized MPI implementations [4]. In addition, a number of free implementations are available. These implementations typically allow distribution over the local network, thus supporting cluster-of-workstation solutions.

3 MPI Functionality

The MPI specification gives applications access to the following functions [2]:

- Point-to-point communication
- Collective operations
- Process groups
- Communication contexts
- Process topologies
- C and Fortran Bindings
- Environmental Management and inquiry
- Profiling interface

Besides clarifying recognized issues with MPI 1.1, MPI 2.0 adds the following functions:

- Extension of the process model, to support dynamic creation of processes
- One-sided communication
- Extensions to the collective operations
- Generalized requests
- Support for parallel I/O routines
- C++ Binding

MPI Communication Concepts

In MPI, communication occurs between processes, each having a unique identification. Communication in MPI is always assumed to be reliable, so that processes do not need to deal with message loss and retransmission. Communication can involve two or more processes. Several communication modes are available:

1. **Blocking (standard):** The sending call returns as soon as the MPI library has sent the message, i.e. after it is safe for the application to re-use the buffer. Since it may or may not depend on a corresponding receive operation, it is implementation defined whether this is asynchronous or not.
2. **Buffered:** The sending call returns as soon as MPI either delivered the message, or after MPI made a copy of the message. Since this mode involves copying, it may not be as efficient as the normal mode.
3. **Synchronous:** The sender blocks until the receiver has received the message.
4. **Ready:** The sender may post a message only after the receiver is expecting it. If a message is sent without the receiver being ready, the outcome is undefined.

In addition to these communication modes, non-blocking calls are available which require calls to completion routines. In non-blocking communication, computation may continue while a message send is in progress. A number of library routines to test completion status of individual requests is available, as well as routines to wait for completion on individual requests.

In addition to the point-to-point communication, MPI provides a number of collective communication primitives:

- barrier synchronization.
- broadcast communication,
- scatter/gather communication, and
- global reduction, and reduce-scatter.

With MPI 2.0, one-sided communication is added, where processes can read and write to a remote memory location. If data rarely changes, but is frequently accessed in different locations with an unknown access pattern, one-sided communication provides a more efficient communication mechanism.

MPI Data types

MPI supports the transmission of a number of primitive types, and provides a number of type constructors. The primitive types are language dependent, for C, the following types are supported:

- char,
- short,
- int
- unsigned char,
- unsigned short,
- unsigned,
- unsigned long,
- float,
- double,
- BYTE, and
- PACKED.

In addition, the following type constructors are available:

- contiguous copies of a type,
- vectors, obtained by retrieving values at repeated indices in memory (both with byte and object indices),
- indexed types, where the indices may vary from block to block,
- structures, where each element may have a different type.

Constructed types are available by means of constructor functions. A constructor function produces a type handle, which can later be used to refer to a type of a value. In addition to these type constructors, explicit packing and unpacking routines are available to provide “custom marshaling”.

Process Groups

In MPI 1.1, operations for process management allow the definition of process groups to separate processes. Based on process groups, communicators can be defined, which restrict the range of a

communication. Implementations of MPI can use communicators to efficiently implement group communication, for example.

In addition, communicators can be attached to topologies. A topology can be used both to simplify addressing, as well as to map processes onto processor nodes efficiently.

While MPI 1.1 does not specify how processes are started, MPI 2.0 provides explicit operations for management of processes. New processes can be dynamically created, and then join to communicators.

MPI Usage Example: Simulation of distributed systems

While MPI is clearly designed to support homogeneous (SIMD) algorithms, simulation of distributed systems also maps very nicely onto the message passing paradigm.

We are currently researching usage of MPI in simulation of systems specified in SDL, the ITU-T Specification and Description Language. System descriptions consist of a number of communicating (extended) finite state machines. A process has an infinite queue of messages. In each transition, it removes one signal from the queue and processes it. During a transition, the process may send signals to other processes. In addition to reception of a signal, a process may also be triggered by a timer's time-out, or it may act spontaneously.

Traditionally, SDL is used in the telecommunications standardization domain to define telecommunication protocols. Since SDL is a formal description technique, a number of analyzing methods are available. One approach is simulation, which typically requires computational power that grows linear with the number of processes involved. To realistically analyze a telecommunication network, a large number of processes may be necessary.

When performing event-driven simulation, message passing is a convenient approach to perform parallel simulation. Message passing appears in the following ways in an speculative (time-warp) simulation:

- The communicating processes exchange application level messages.
- Distribution of the global virtual time may be based on message passing.
- In case of roll-back, anti-messages need to be generated.

5 MPI and CORBA: An Outlook

While MPI is certainly a convenient interface to implement message-passing-based algorithms, we identified a number of problems when trying to implement a parallel simulation system:

1. It is difficult to integrate external communication into MPI. In the parallel simulator, we have the need to process the simulation result in a variety of ways (interactive debugger, statistical analysis, etc.). To do so, we defined a CORBA interface between the simulator proper and the analysis tools. However, there is no straight-forward way to integrate the CORBA request processing into the processing of MPI send and receive calls.
2. Representing complex data structures is much easier in CORBA. With IDL, complex data structures can be described in a language independent way, and tools generate language bindings to transmit these data automatically.
3. Addressing in MPI is on the process level, not on the object level. Thus, in an object-oriented parallel algorithm, addressing of individual objects must be implemented on top of the existing message-passing functionality.

When looking at both technologies, we think that a combination of both might give application developers additional flexibility. In particular, we think that a harmonization of both in the following areas might be useful:

- Integration of run-time systems: Usage of both MPI and CORBA in a single application is desirable.
- Usage of CORBA type system with MPI: It should be possible to transmit operation invocations for CORBA objects using MPI as an infrastructure. Exposing all of MPI's communication primitives to the CORBA level might be an additional challenge.

Both MPI and CORBA are maintained by independent organizations. It seems clear that members of both organizations should be involved in harmonization efforts.