# An Introduction to the MPI Standard

*Can we program scalable parallel machines in a portable manner?*

This talk based on: *An Introduction to the MPI Standard*, Jack Dongarra, Steve Otto, Marc Snir, David Walker. To appear in *CACM*, and it's on the web(!) ---

- http://www.netlib.org/utk/papers/intro-mpi/intro-mpi.html

MPI defines the syntax and semantics of a core of library routines useful for writing portable message-passing programs in C or Fortran 77.

The intent was to standardize "current practice" and maybe go a bit beyond (we did), and provide a message passing specification suitable for:

- message-passing programmers
- target for compiler writers
- parallel library writers

MPI already exists! Runs on both MPPs and NOWs.

Public domain...

- Argonne / Mississippi State implementation. Runs on top of PVM or P4.
- Edinburgh “CHIMP” implementation. Runs on a variety of machines...workstations, Cray Research T3D.
- Mississippi State “UNIFY” implementation. Provides subset of MPI within PVM.
- Ohio Supercomputer Center LAM implementation. MPI on top of LAM, a Unix cluster environment.

Vendors...

- IBM, Meiko, Intel, Cray Research, Ncube

MPI has its roots in...

- PVM, Express, P4, Zipcode, Parmacs, and in vendor-specific systems such as Intel NX and IBM EUI

MPI was developed by about 80 people from 40 organizations, both academic and industrial. We met regularly for a year...we published a specification...

“MPI: A Message-Passing Interface Standard,” *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4), 1994, and, ftp://www.netlib.org/mpi/mpi-report.ps

## Fast Overview

MPI specifies communication between processes forming a concurrent program.

Message passing attractive because

- wide portability and scalability...easily compatible with both distributed-memory multicomputers and shared-memory multiprocessors...and NOWs.
- it won't be made obsolete by faster networks or fancier architectures...at *some* level, you'll always want to do some message passing.

Much of MPI just standardizes "common practice," but there are places where it goes further:

- user-defined datatypes
- persistent communication ports
- powerful collective communication operations
- scoping mechanisms for communication

## Goals

To *really* understand MPI, one must think about

- goals of the standardization effort
- constraints this implies
- practical constraints

Here's a list of goals:

- timely completion of *something*. Meant that we only did message passing, not process control, file systems, etc.
- portable API, usable by programmers
- allow efficient communications, on many platforms
- allow heterogeneous implementations
- allow convenient C and Fortran 77 bindings
- design an interface compatible with a wide variety of hardware and operating systems

- design an interface that does not require programmer to deal with communication failures
- design an interface not too different from common practice
- semantics must be language independent
- allow implementations with multiple threads of execution within each process

## What MPI Does and Does Not Specify

It Does...

- a powerful and elaborate set of point-to-point communication functions (some would say “overly complex”);
- collective communications
- process groups
- communicators: mechanism for providing separate communication scopes
- process topologies
- bindings for C and Fortran 77
- an open profiling interface for debugging, visualizers, etc

- environment management and inquiry

It Doesn't ...

- shared-memory operations
- interrupt-driven messages, remote execution, active messages
- program construction tools
- debugging support
- thread support
- process management
- input and output

More to do! Time constraint....

## MPI Point-to-Point

This is complex, but it’s good. *Trust me*.

## Blocking Communications

First we consider only the blocking pt-pt, and we will begin our discussion by consider things from the point of view of the sending side.

The *standard-mode*, *blocking* send operation is:

```
MPI_Send( void *buf, int size, MPI_Type type, int dest, int tag, MPI_Comm comm)
```

The function communicates **typed** data with an associated message **tag**.

- typing is necessary for heterogeneous support
- tag allows selectivity of messages at receiving end: one can receive on a particular tag, or one can wild-card this.
- comm is a **communicator**. Among other things, it serves to define the allowed set of processes involved in a commmunication.

- Process ranks are integers, label processes, and are discovered by inquiry to a communicator (the same process may have different labels within different communicators).

This is a *blocking send*: the call blocks until the send buffer can be reclaimed. Similarly, the blocking receive function waits until the receive buffer actually contains the contents of the message.

MPI also provides *non-blocking* send and receive functions that allow the possible overlap of message transmittal with computation, or the overlap of multiple transmittals with each other.

- non-blocking functions always come in two parts: the *posting* functions, which begin the operation; and the *test-for-completion* functions, which allow the application to discover whether the operation has completed.

## Example 1

Process 0 sends a message to process 1.

```
char msg[20];
int myrank, tag = 99;
MPI_Status status;
...
MPI_Comm_rank( MPI_COMM_WORLD, &myrank); /* find my rank */
if (myrank == 0) {
   strcpy( msg, "Hello there" );
   MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD );
}
else {
   MPI_Recv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &status );
}
```

## Example 2

Do same thing using non-blocking function at receive side.

```
...
MPI_Request request;
...
if (myrank == 0) {
   strcpy( msg, "Hello there" );
   MPI_Send( msg, strlen(msg)+1, MPI_CHAR, 1, tag, MPI_COMM_WORLD );
}
else {
                              /* post recv */
   MPI_Irecv( msg, 20, MPI_CHAR, 0, tag, MPI_COMM_WORLD, &request );
   ...                        /* do useful work */
   MPI_Wait( request, &status );     /* now wait for completion */
}
```

## But wait, there's more!

...even more to say about a simple transmittal of data from one process to the other!

- Why? Consider the semantics of the functions, and the underlying protocols.

Suppose a blocking send has completed. Does this tell us anything about the receiving process? Can we know that the receive has finished, or even, begun?

well...it depends...

on the underlying protocol.

- If we wish to implement a protocol minimizing copying and buffering of data, the most natural semantics might be a "rendezvous" version, where completion of the send implies that the receive has started (at least).
- On the other hand, a protocol that attempts to block processes for the least time will, necessarily, end up copying and buffering data.

Trouble is, one choice is not best for all applications, nor is it best for all architectures.

*Because the primary goals of MPI are to standardize operations, yet not sacrifice performance, the decision was made to include all the major choices for point to point semantics in the standard. These choices are called* ***modes****.*

A final complicating factor is that the amount of space available for buffering is always finite. It might even be small. Therefore, MPI is very careful not to mandate a minimal amount of buffering and it is careful about the semantics it requires.

## Communication Modes

Modes are specified at the sending side. There is only one kind of receive.

- in **standard** mode, completion of the send does not necessarily mean that the receive has started, and no assumption should be made about the buffering of the message.
- in **buffered** mode, the user can guarantee that a certain amount of buffering space is available. The catch is that the space must be provided by the application program.
- in **synchronous** mode, a rendezvous semantics is used.

- in a **ready** mode send, the user asserts that the matching receive has already been posted (allows a simplification of protocol...speed!)

## User-defined Datatypes

So far, we have been discussing communications of contiguous buffers containing a sequence of elements of the same, basic type. This is too constraining in two ways:

- we often want to pass messages that contain different types within them
- we often want to send non-contiguous data

A *derived datatype* specifies two things:

- a sequence of basic datatypes
- a sequence of integer (byte) displacements

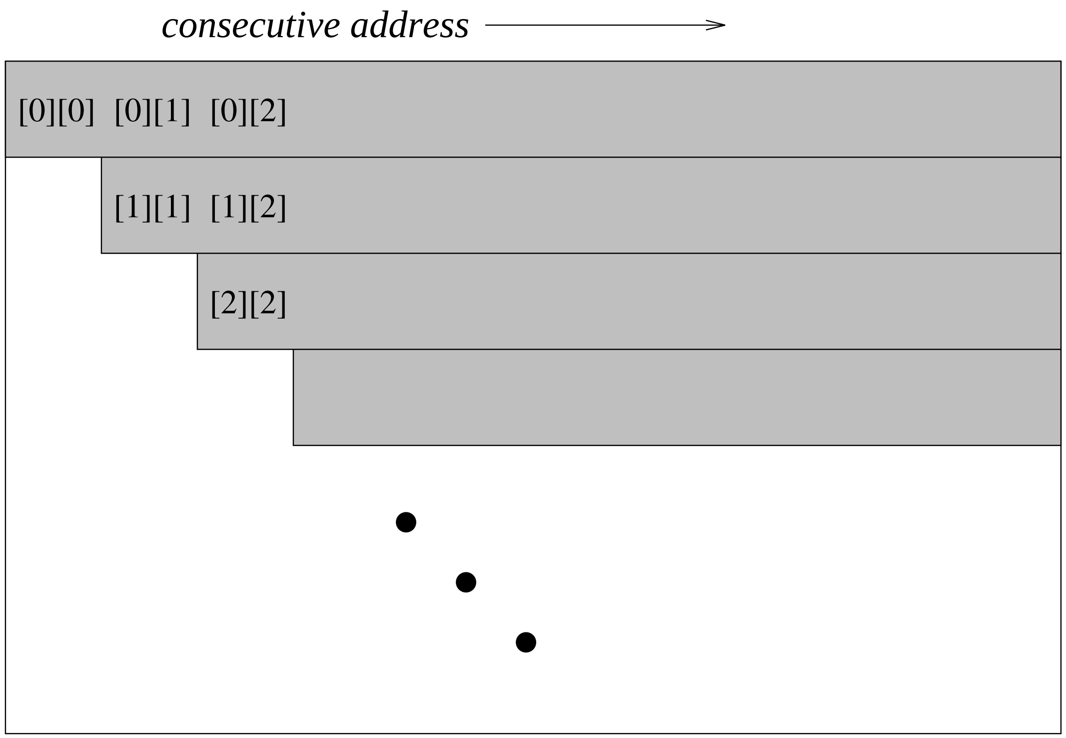
Both sequences, taken together, are called the *type map* of a derived datatype, while the sequence of types is the *type signature* of the derived datatype.

## Example 3

Use user-defined datatype to send upper-triangular part of matrix in one send.

```
double a[100][100];
int disp[100], blocklen[100],i;
MPI_Datatype upper;
...
                    /* compute start and size of each row */
for (i=0; i<100; ++i) {
   disp[i] = 100 * i + i;
   blocklen[i] = 100 - i;
}
                    /* create datatype for upper-triangle */
MPI_Type_indexed( 100, blocklen, disp, MPI_DOUBLE, &upper );
MPI_Type_commit( &upper );
                    /* and send! */
MPI_Send( a, 1, upper, dest, tag, MPI_COMM_WORLD );
```

A picture of the datatype. The cells referred to by the datatype are shaded.

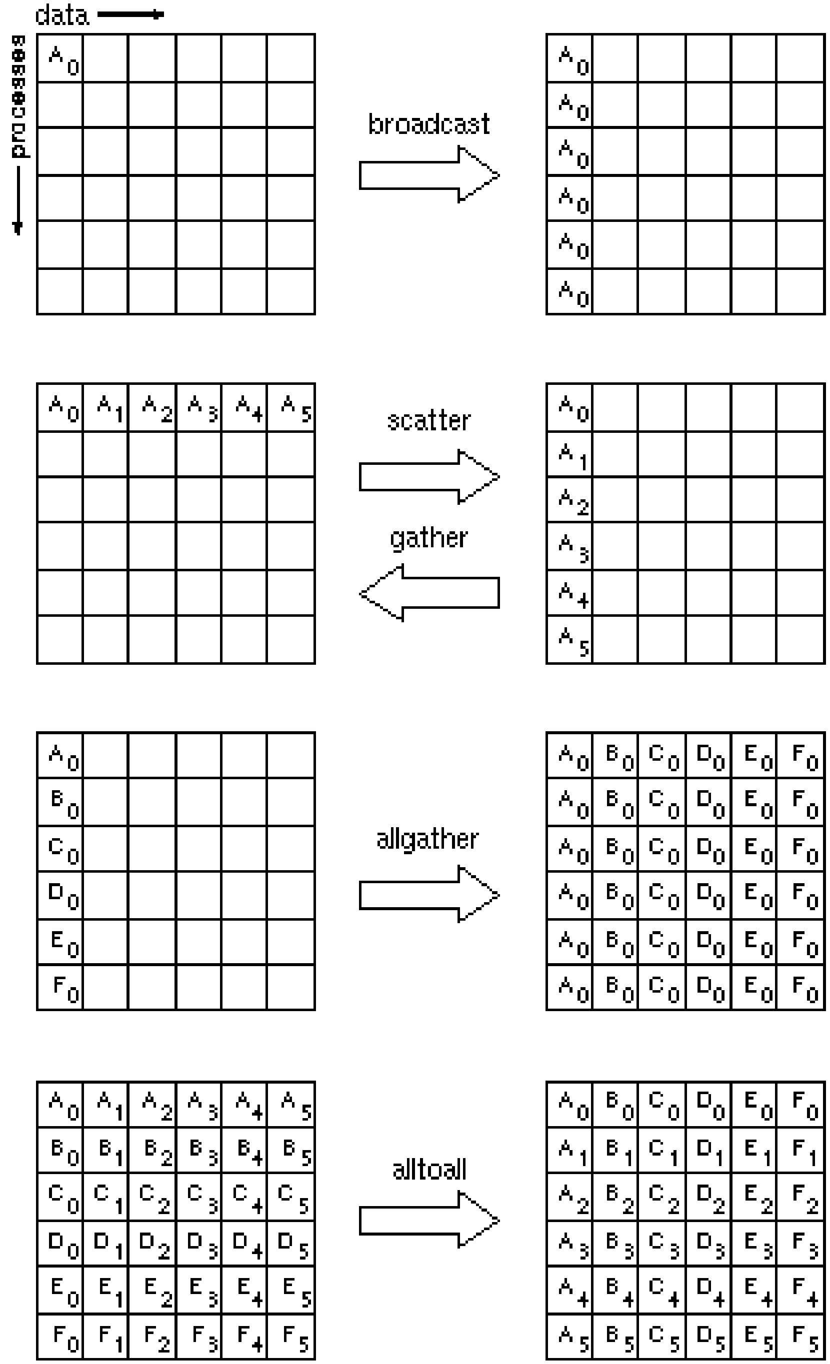


## Collective Communications

Collective communications transmit data among all processes specified by a communicator. (one function, the barrier, serves to synchronize without passing data)

- Barrier synchronization across all group members
- Broadcast from one process to all
- Gather from all to one
- Scatter from one to all
- Allgather: like a gather, followed by a broadcast of the gather output
- Alltoall: like a set of gathers in which each process receives a distinct result
- Global reductions, such as sum, max, min, etc, and user-defined
- Prefix (or scan) across processes

See picture.


data
processes
broadcast
scatter
gather
allgather
alltoall

Many of the collective functions also have “vector” variants, whereby different amounts of data can be sent to or received from different processes.

Syntax and semantics of collective was designed to be consistent with point to point. However, to keep complexity under control, the MPI committee made collective functions more restrictive than point to point.

- amount of data sent must exactly match amount specified by receiver
- collective come in blocking versions only. No “non-blocking barrier” ;^)
- collective functions come in only one mode, and this can be regarded as analogous to the standard mode for point to point. Specifically ...

## Semantics of Collective Functions

A collective function (on a given process) can return as soon as its participation in the overall communication is complete.

- as usual, completion indicates the caller can access and modify the communication buffer
- it does not indicate that other processes have completed, or even started, the operation

- Thus, a collective call may, or may not, have the effect of synchronizing all processes

This was done so as to allow a variety of implementations

User of collective must program in a somewhat defensive manner (if one wishes to be truly portable).

- For example, even though a particular implementation of MPI may provide a broadcast with side-effect of synchronization (MPI allows this)... the standard does not *require* this, and so any program that relies on the synchronization is not portable.
- On the other hand, a correct and portable program must also *allow* for the fact that the broadcast is synchronizing.

*Though one should not rely on synchronization side-effects, one must program so as to allow for it!*

All this might seem unusually obscure (sigh)...but they are a simple consequence of the desire of MPI to:

- allow efficient implementations on a variety of architectures

- be clear about what is, and what is not, guaranteed by the standard

## Groups, Contexts, and Communicators

You are a library writer. You want to guarantee that communication within a library routine does not conflict with communications from the calling, application program.

A **communicator** is a data object that specifies the scope of a communication operation, that is,

- **group** of processes involved
- communication **context**

A **context** partitions the communication space -- a message sent in one context cannot be received in another.

One can create new communicators and doing so is a collective operation.

## Recommendations for Library Authors

One workable design:

- have the application program pass into the library a communicator that specifies the process group and ensures a safe context

Another design:

- have library create a “hidden” (and unique) communicator at library-initialization time.
- this type of design is illustrated in example below.
- as one works with it, one finds that it is convenient to associate these new, hidden communicators with the old communicators (passed in by the application) from which they were derived.
- the **caching** mechanism is a nice way to set up this association -- see example

## Library Caching Example

```
            /* static variable used as ``key'' for library */
            /* Only one per process is necessary, even if multiple */
            /* library invocations can be concurrently active. */
extern int lib_key;
            /* library init. Need to invoke once by each process, */
            /* before library is used. */
void lib_init()
{
            /* allocate a process-unique key */
   MPI_Keyval_create(MPI_NULL_FN, MPI_NULL_FN, &lib_key, (void *)NULL);
}
```

```
void lib_call( MPI_Comm comm, ... )

{   int flag;
                /* private communicator for library-internal communication */
    MPI_Comm *private_comm;
                /* retrieve private communicator */
    MPI_Attr_get( comm, lib_key, &private_comm, &flag );
    if (!flag) {
                /* get failed; this is first call and private_comm */
                /* has not yet been allocated. So, do it. */
                /* Make new communicator, with same process group as comm. */
      private_comm = (MPI_Comm *)malloc(sizeof(MPI_Comm));
      MPI_Comm_Dup( comm, private_comm );
                /* Cache private communicator with public one. */
      MPI_Attr_put( comm, lib_key, (void *)private_comm );
    }
                /* Execute library code, using private_comm for */
                /* internal communication. */
}
```

## Conclusions

Standardizing message passing not so simple...actually kind of interesting

We've concentrated on "hard" things...most of the time, programming with MPI is starightforward

The key is a truly *portable* environment, and to *deliver* it

Future directions for MPI:

- parallel I/O
- non-blocking collective
- remote store/access
- active messages
- process startup and control
- Fortran 90 and C++ bindings
- graphics

- real-time

Stay involved! Pay attention through the MPI newsgroup: comp.parallel.mpi, and the various web pages.