Basics of MPI Programming

Michael Hanke

Summer School on High Performance Computing

Outline

What Is MPI?

- *M*essage *P*assing *I*nterface standard
- The first standard and portable message passing library with good performance
- "Standard" by consensus of MPI Forum participants from over 40 organizations
- Finished and published in May 1994, updated in June 1995
- MPI 2 complete as of July 1997. Extends MPI.
- Documents that define MPI are on the World Wide Web from the MPI Forum.

MPI Versions

MPI version	Calls
Version 1.0 (1994): Support for Fortran77 and C	129
Version 1.1 (1995): Small errata and clarifications	0
Version 1.2 (1997): More errata and clarifications	1
Version 1.3 (2008):	?
Version 2.0 (1997): Essential extensions	193
one-sided communication, MPI-IO,	
support for Fortran90 and C++, dynamic process generation	
Version 2.1 (2008): Errata and clarifications	0
	323

What Does MPI offer?

- Standardization - on many levels
- Portability - to existing and new systems
- Performance - comparable to vendors' proprietary libraries
- Richness - extensive functionality, many quality implementations

MPI at the Parallel Computer Center (PDC)

- There are many implementations of MPI on PDC's machines:

Machine	Implementation
Ferlin	OpenMPI (with Intel or GNU compilers), MVAPICH
Key	MPICH, mpichmx, OpenMPI

- A common implementation is MPICH
- At PDC, you usually use the module command to select your MPI implementation

How to Use MPI

- When possible, start with a debugged serial version
- Design parallel algorithm
- Write code, making calls to MPI library
- Compile and run: covered in PDC's Guided Tours; for example, see:
  - How to run programs on <machine>
  - Run with a few nodes first, increase number gradually
- Hint: For debugging purposes it may be appropriate to develop an MPI program on your workstation first.

A Warning

In many respects, using MPI is low-level and thus comparable to assembly programming.

Format of MPI Routines

- C Bindings
  - `rc = MPI_Xxxxx(parameter, ... )`
  - `rc` is error code, = `MPI_SUCCESS` if successful
- Fortran bindings
  - `call MPI_XXXXX(parameter,..., ierror)`
  - case insensitive
  - `ierror` is error code, = `MPI_SUCCESS` if successful
- Exception: timing functions return double precision reals
- Header file required
  - `#include "mpi.h"` for C programs
  - `include 'mpif.h'` for Fortran programs

MPI Routines

MPI has over 320 functions, but you can do much with just 6:

Initialize for communications

MPI_INIT initializes the MPI environment

MPI_COMM_SIZE returns the number of processes

MPI_COMM_RANK returns this process's number (rank)

Communicate to share data between processes

MPI_SEND sends a message

MPI_RECV receives a message

Exit in a "clean" fashion when done communicating

MPI_FINALIZE

An MPI Sample Program (Fortran) I

Here, we show the 6 basic calls:

```
program hello
include 'mpif.h'
integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)
character(12) message

call MPI_INIT(ierror)
call MPI_COMM_SIZE
  (MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK
  (MPI_COMM_WORLD, rank, ierror)
tag = 100
```

cont ...

An MPI Sample Program (Fortran) II

```
 if (rank .eq. 0) then
   message = 'Hello, world'
   do i=1, size-1
     call MPI_SEND(message, 12, MPI_CHARACTER, i, tag,
&                  MPI_COMM_WORLD, ierror)
   enddo
 else
   call MPI_RECV(message, 12, MPI_CHARACTER, 0, tag,
&                MPI_COMM_WORLD, status, ierror)
 endif

 print*, 'node', rank, ':', message
 call MPI_FINALIZE(ierror)
 end
```

Skip C

An MPI Sample Program (C version) I

```
#include <stdio.h>
#include "mpi.h"
main(int argc, char **argv) {
  int rank, size, tag, rc, i;
  MPI_Status status;
  char message[20];

  rc = MPI_Init(&argc, &argv);
  rc = MPI_Comm_size(MPI_COMM_WORLD, &size);
  rc = MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  tag = 100;
```

cont ...

An MPI Sample Program (C version) II

```
  if(rank == 0) {
    strcpy(message, "Hello, world");
    for (i=1; i<size; i++)
      rc = MPI_Send(message, 13, MPI_CHAR, i,
                    tag, MPI_COMM_WORLD);
  }
  else
    rc = MPI_Recv(message, 13, MPI_CHAR, 0,
                  tag, MPI_COMM_WORLD, &status);
  printf( "node %d : %.13s\n", rank,message);
  rc = MPI_Finalize();
}
```

What Is a Message?

Message = data + envelope

```
call MPI_SEND(startbuf,count,datatype,
                    \      |     /
                     ---DATA---

                   dest,tag,comm,  ierror)
                     \    |    /
                      ENVELOPE
```

Data

- Arguments
  - startbuf (starting location of data)
  - count (number of elements)
    receive >= send
  - datatype (basic or derived)
    receiver = send (unless `MPI_PACKED`)
- MPI Datatypes
  - Basic
  - Derived (mixed, non-contiguous data etc.)

MPI Basic Datatypes (Fortran)

MPI datatype	Fortran datatype
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)
MPI_BYTE	
MPI_PACKED	

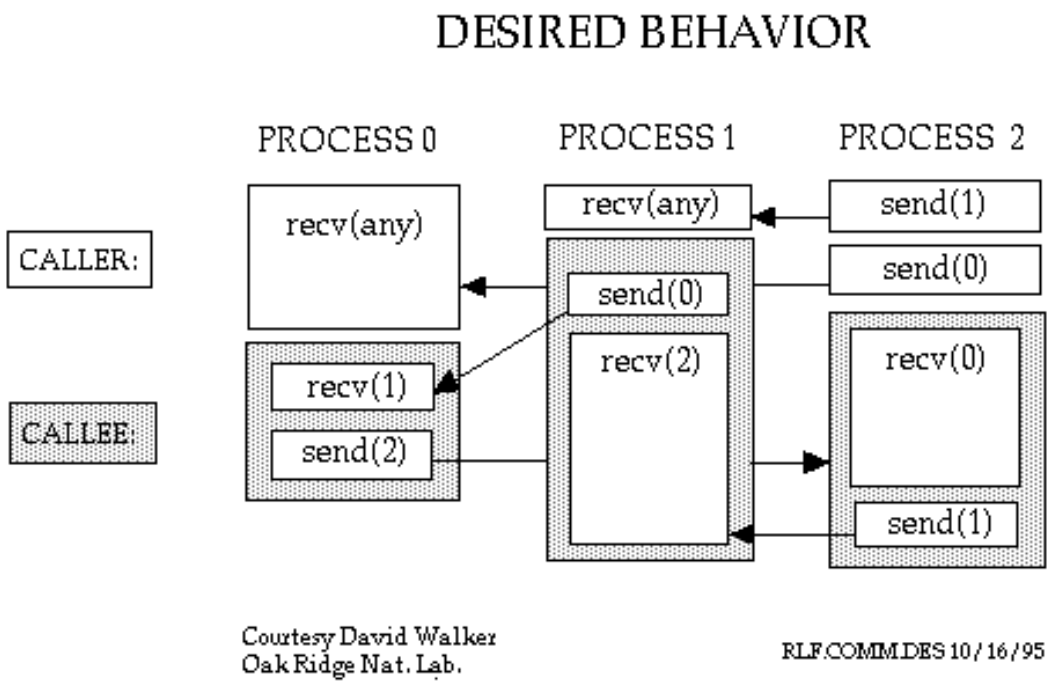
The names of the MPI C datatypes are slightly different!

Envelope

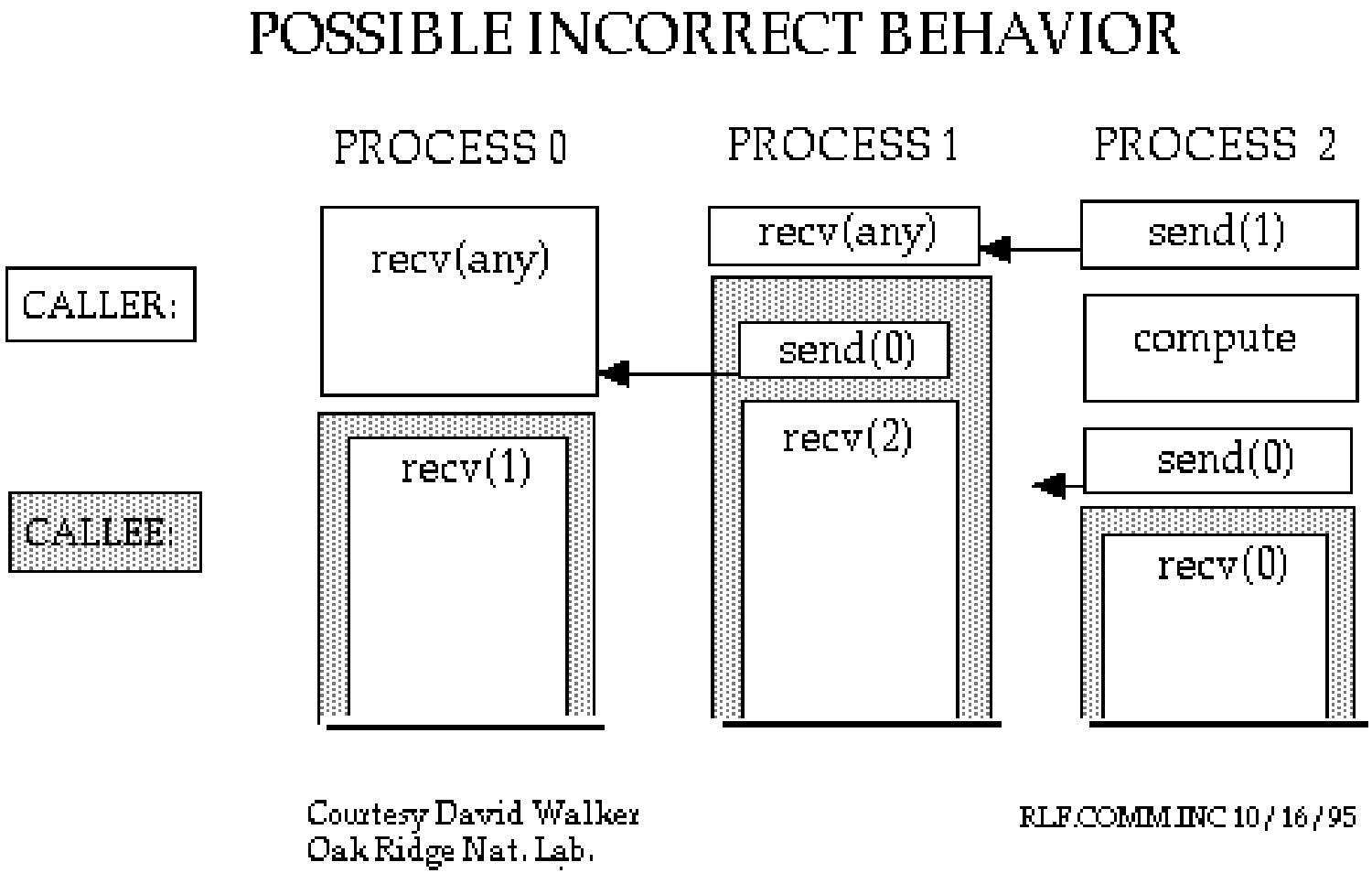
- Destination or source
  - rank in a communicator
  - receiver = sender or `MPI_ANY_SOURCE`
- Tag
  - integer chosen by programmer
  - receiver = sender or `MPI_ANY_TAG`
- Communicator
  - defines communication "space"
  - receiver = sender

Why Have Communicators?

- If you are writing a library, can you choose safe tags?
- Example: variable (and possibly incorrect) behavior if no communicator

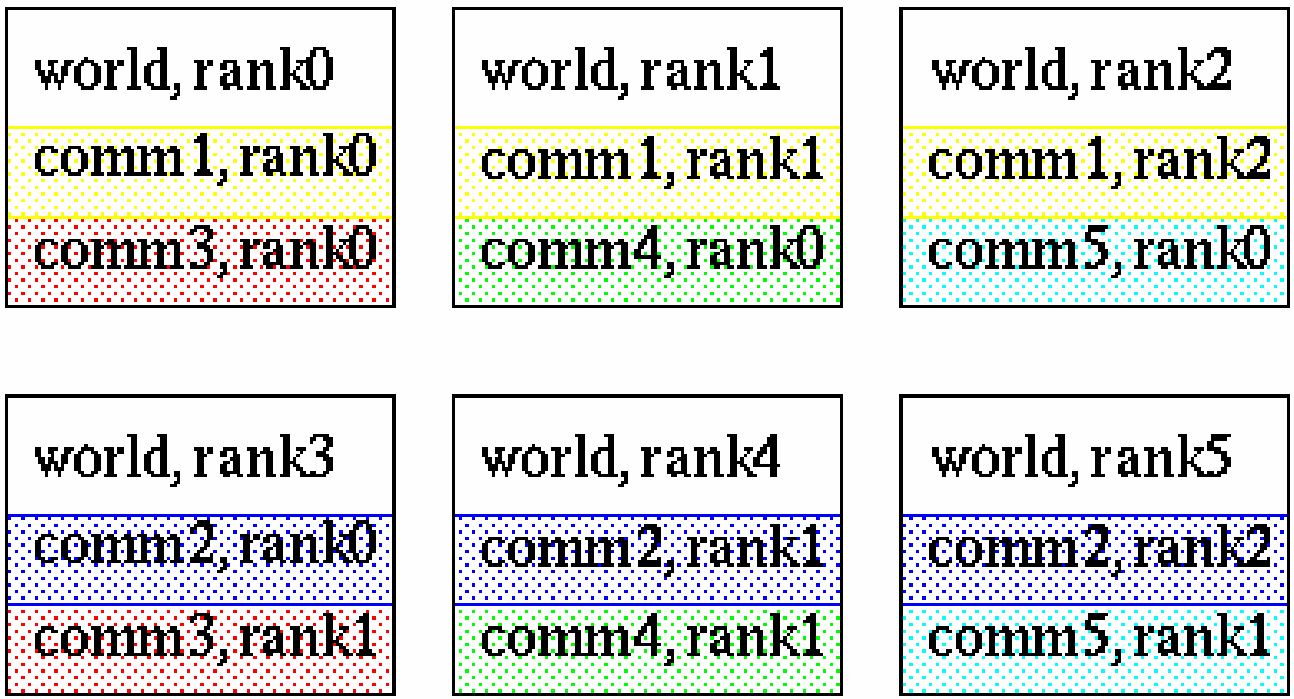


Example, cont.



Communicators And Process Groups

- Programmers can define a process group
- And define new communicator(s) for the process group



Summary

- MPI program: 6 basic calls
  - `MPI_INIT`
  - `MPI_COMM_RANK`
  - `MPI_COMM_SIZE`
  - `MPI_SEND`
  - `MPI_RECV`
  - `MPI_FINALIZE`
- MPI messages
  - data (`startbuf, count, datatype`)
  - envelope (`destination/source, tag, communicator`)
- Communicators
- What comes next?
  - Basics of Point-to-Point communication

MPI Calls: MPI_INIT

- MPI_INIT must be the first MPI routine you call in each process.
- It can only be called once.
- It establishes an environment necessary for MPI to run.
- This environment may be customized for any MPI runtime flags provided by the MPI implementation.
- The command line arguments are passed to the C version.

C

```
int MPI_Init(int *argc, char ***argv)
```

Fortran

```
call MPI_INIT(ierror)
integer ierror
```

MPI calls: MPI_COMM_SIZE

- MPI_COMM_SIZE returns the number of processes within a communicator.
- MPI can determine the number of processes because you specify this when you set the environment variable MP_PROCS.

C

```
int MPI_Comm_size(MPI_Comm comm, int *size)
```

Fortran

```
MPI_COMM_SIZE(comm, size, ierror)
integer comm, size, ierror
```

MPI calls: MPI_COMM_RANK

- Rank is used to specify a particular process.
- It is an integer in the range 0 through size-1.
- MPI_COMM_RANK returns the calling process's rank in the specified communicator.

C

```
int MPI_Comm_rank(MPI_Comm comm, int *rank)
```

Fortran

```
MPI_COMM_RANK(comm, rank, ierror)
integer comm, rank, ierror
```

MPI calls: MPI_SEND

- A message will be sent to another process.
- MPI_SEND is a blocking send.

C

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
```

Fortran

```
MPI_SEND(buf, count, datatype, dest, tag, comm, ierror)
<type> buf(*)
integer count, datatype, dest, tag, comm, ierror
```

MPI calls: MPI_RECV

- A message will be received from another process.
- MPI_RECV is blocking.

C

```
int MPI_Recv(void *buf, int count, MPI_Datatype datatype,
             int source, int tag, MPI_Comm comm,
             MPI_Status *status)
```

Fortran

```
MPI_RECV(buf, count, datatype, source, tag, comm,
         status, ierror)
<type> buf(*)
integer count, datatype, source, tag, comm
integer status(MPI_STATUS_SIZE), ierror
```

MPI calls: MPI_FINALIZE

- The last call to MPI should be MPI_FINALIZE.
- This makes MPI to exit cleanly.
- The code can continue to execute after calling MPI_FINALIZE, but it can longer call MPI routines.

C

```
int MPI_Finalize()
```

Fortran

```
MPI_FINALIZE(ierror)
integer ierror
```

MPI Datatypes (C)

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	
MPI_PACKED