

MPI

Overview:

- Generalities
- Point to point communication
- Collective communication

What is MPI

Library interface for distributed/parallel computing

Major goal of MPI : Portability

Allows coupled computations using heterogeneous platforms

IEEE MPI Standard; started on 1992

No restriction to a specific compiler (language bindings for Fortran, C, C++)

No specific product MPICH, MPI Info and standard:

www.mcs.anl.gov/mpi

List of very useful software that runs on MPI:

FFTW, SuperLU (dense parallel linear algebra), PETSc(www.mcs.anl.gov/petsc),

ParMetis

Why MPI

Standardization, Portability, Performance (Vendor implementations), Availability, Functionality

MPI has added credibility to parallel computing!

Main Ideas/Features

can run simultaneously on different architectures

independent and concurrent execution of programs

Each processor has private memory and address space

Communication takes place cooperatively

MPI most commonly runs same binary but can have different programs (support for both SIMD and MIMD)

Ideally each node connected to every other node but very expensive

Deadlocks can occur

Fundamentals. Processes (PEs) - ID identified by group and rank

Messages - ID identified by context and tag

context: group, receiving ranks, sending ranks

- Groups/Communicators - Define independent sets of processors for collective communications

- Point-to-point routines - communication between 2 processors
- Collective communication/Reductions - communication between all processors in a group (data movement and computations like SUM,MIN,MAX)

Point to point - blocking, nonblocking
 Modes - Synchronized, buffered, Ready, standard

MPI-1 has 128 routines MPI-2 has 287 routines but with the following routines you can do most of the work:

```

Management
MPI_Init()      - initialize MPI (no MPI function calls before that)
MPI_Comm_size() - get total number of processors
MPI_Comm_rank() - get process ID
MPI_Finalize()  - terminate MPI (no MPI function calls after that),

Collective
MPI_Reduce()    - collective computation (sum,min,max)
MPI_Exscan()    - collective computation Scan
MPI_Alltoall()  - global data movement
MPI_Alltoallv() - global data movement
MPI_Gather()    - all to one
MPI_Scatter()   - one to all
MPI_Barrier()  - synchronize PEs (waits till all PEs reach this point)

Pont to point
MPI_Send()      - send a message
MPI_Recv()      - receive a message
MPI_Sendrecv()  - order independent send receive messages concurrently
MPI_Isend()     - non blocking send
MPI_Irecv()     - non blocking receive
MPI_Waitall()   - wait for completion of all non blocking messages

```

Basic routines for MPI. All MPI routines return integer indicating error code.
 Dead locks: (synchronous) send(message,0) send(message,1) receive receive

General info on MPI

Examples of MPI - Hello world

```

#include <mpi.h> // Use "mpi.h" do access MPI routines
#include <stdio.h>

/* ***** */
/*
 * Every processor will print 'hello world' and its rank
 */
int main( int  argc, char **argv)
{
    int rank;
    MPI_Init( &argc, &argv); // Start MPI service
    MPI_Comm(MPI_COMM_WORLD, rank); // Get rank

```

```

// ALL Non MPI Routines are local
// Thus printf runs on each processor
printf("[%d] Hello world!\n", rank);
// In general output will not be in order

// However: often not every processor can do I/O
// MPI2 defines I/O API

MPI_Finalize(); // Terminate MPI service
return 0;
}

```

Building an MPI program:
standard compiler, (add appropriate include paths)
link with MPI library

```

>${MPICH_DIR}/bin/cc mpi1.c -o mpi1 -lmpi
or you can use
>${MPICH_DIR}/bin/mpicc mpi1.c -o mpi1
>${MPICH_DIR}/mpirun -np 4 mpi1

```

For solaris MPICH_DIR should be something like
/usr/local/lib/mpich/ch_shmem/bin/mpicc

Ask your administrator.
MPI PE are ordered and identified by their integer rank

Message information. Message envelope: everything you need to identify
from where to where.

Necessary info for a message specification
address: memory location to copy data from
count: number of objects
datatype: type of objects (for Byte specification)
source/destination: where to receive from/where to send to
tag:identifier for message or kind of message
communicator:context of message.

Some MPI Semantics for interface

(from the MPI standard)

nonblocking A nonblocking routine may return before the operation completes; user is NOT allowed to re-use input resources (such as buffers) specified in the call. The routine *starts* an operation, but upon return the operation most likely has not been *completed*. **Notice difference between call-completion and operation-completion.** Typically in the list of return arguments there is a *request* object, which can be used to probe whether a nonblocking operation has been completed.

blocking Upon the return from a call to a blocking routine the user is allowed to re-use resources specified in the call.

local The completion of the routine depends only on the local executing process. Such an operation does not require communication with another user process.

non-local If completion of the operation may require the execution of some MPI procedure on another process. Such an operation may require communication with another process.

collective During a call to a collective routine all processes in a process group need to invoke this routine.

P2P communication

All other MPI routines are built around this operation: Transmittal of data between a pair of PEs, one side sending, the other, receiving.

MPI provides a set of send and receive functions that allow the communication of *typed* data with an associated *tag*.

Standard communication mode

```
#include <stdio.h>
#include "mpi.h"
main( int argc, char **argv )
{
    char message[20];
    int myrank;
    MPI_Status status;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &myrank );
    if (myrank == 0) /* code for process zero */
    {
        strcpy(message, "Hello, there");
        MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
    }
    else /* code for process one */
    {
        MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
        printf("received :%s:\n", message);
    }
    MPI_Finalize();
}
```

standard Send operation (Blocking, non-local)

```
MPI_Send(buf, count, datatype, dest, tag, comm)
[ IN buf] initial address of send buffer (choice)
[ IN count] number of elements in send buffer (nonnegative integer)
[ IN datatype] datatype of each send buffer element (handle)
(MPI_INT, MPI_DOUBLE,
custom, contiguous or non contiguous data etc)
[ IN dest] rank of destination (integer)
```

[IN tag] message tag (integer)
[IN comm] communicator (handle)

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

standard Receive operation (Blocking)

```
MPI_RECV (buf, count, datatype, source, tag, comm, status)  
[ OUT buf] initial address of receive buffer (choice)  
[ IN count] upper bound number of elements in receive buffer (integer)  
[ IN datatype] datatype of each receive buffer element (handle) -  
[ IN source] rank of source (integer)  
[ IN tag] message tag (integer) or wildcards  
[ IN comm] communicator (handle)  
[ OUT status] status object (Status) (must be allocated and use by  
user) or MPI_STATUS_IGNORE
```

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

Receive could complete before its send. **But Recv completes only if the matching send is posted. Upon return receive buffer contains desired data. Send does not return until the message data and envelope have been safely stored away so that the sender is free to access overwrite the data. That means that without buffering send is synchronous doesn't return unless a matching receive is posted.** With buffering Send is asynchronous.

In general send info can be copied to system buffer, or directly to the receive buffer. Implementation, and resource availability dependent. These restrictions can lead to deadlock

```
always OK  
PE 0  
send(m1,1)  
recv(m2,1)  
PE 1  
recv(m1,0)  
send(m2,0)
```

```
will always Deadlock  
PE 0  
recv(m1,1)  
send(m2,1)  
PE 1  
recv(m2,0)  
send(m1,0)
```

```
depends on buffering  
PE 0  
send( m1, 1)  
recv( m2, 1)  
PE 1  
send( m2, 0)  
recv( m1, 0)
```

`status` contains info like length of message (`count` is an upper bound) the exact tag is wildcard was used, and error codes

- Asymmetry between send and receive calls. Must send to specific PE Can receive from any PE - Source equal to destination is allowed. But requires enough buffering. In general this approach is unsafe not portable so avoid it.

Deadlock: all PEs are blocked, no progress possible. Some programs maybe completed given sufficient buffering but once buffering is out they deadlock.

SAFE Programs should not rely on message buffering. They complete with 0 buffer size

Simultaneous Send Recv (blocking, semantically equivalent to two threads followed by a join)

The send-receive operations combine in one call the sending of a message to one destination and the receiving of another message, from another process. The two (source and destination) are possibly the same. A send-receive operation is very useful for executing a shift operation across a chain of processes. If blocking sends and receives are used for such a shift, then one needs to order the sends and receives correctly (for example, even processes send, then receive, odd processes receive first, then send) so as to prevent cyclic dependencies that may lead to deadlock. When a send-receive operation is used, the communication subsystem takes care of these issues.

A message sent by a send-receive operation can be received by a regular receive operation or probed by a probe operation; a send-receive operation can receive a message sent by a regular send operation.

```
MPI_SENDRFCV(sendbuf, sendcount, sendtype, dest, sendtag, recvbuf,
              recvcount, recvtype, source, recvtage, comm, status)
[ IN sendbuf] initial address of send buffer (choice)
[ IN sendcount] number of elements in send buffer (integer)
[ IN sendtype] type of elements in send buffer (handle)
[ IN dest] rank of destination (integer)
[ IN sendtag] send tag (integer)
[ OUT recvbuf] initial address of receive buffer (choice)
[ IN recvcount] number of elements in receive buffer (integer)
[ IN recvtype] type of elements in receive buffer (handle)
[ IN source] rank of source (integer)
[ IN recvtage] receive tag (integer)
[ IN comm] communicator (handle)
[ OUT status] status object (Status)
```

```
int MPI_Sendrecv(void *sendbuf, int sendcount, MPI_Datatype sendtype,
                int dest, int sendtag, void *recvbuf, int recvcount,
                MPI_Datatype recvtype, int source, MPI_Datatype
                recvtage, MPI_Comm comm, MPI_Status *status)
```

General send/Receive 8 different Sends, 2 Receives

If an `MPI_Sent()` has returned what does this tell us about the receiving process? Can we now that the receive has finished, or even that it has begun? For minimum memory: "rendez vous" For minimum time : buffering

Problem: Not all sends/receive best for all applications, all architectures Therefore we need:

Communication modes. Buffered, Synchronous, Ready, Standard. Designed to allow overlapping of computation with communication, or overlap multiple communication calls. But synchronization responsibility is returned to the user. Buffering allows more flexibility, no need for synchronization and allows less costly communication patterns, Portability in standard mode can create large overheads due to conservative estimates of communication patterns.

Non-blocking communication

- post-send, post-receive, test for send completion (complete-send), test for receive completion (complete-receive).

- post-send is local

- complete-send is non-local and may be blocking if no buffering is used

- nonblocking sends can be matched by blocking receives and vice versa

- buffers are not allowed to be written or *read* will in message hasn't completed.

The read restriction is important and must not be overlooked.

- queries on status of a communication request are done with `MPI_Wait()` (blocking,non-local) or `_Test()` (nonblocking,local)

In all "sends" suffix "I" gives a non-blocking version

Sends posted asap. Recvs posted asap. sends completed just before send buffer is to be reused, receives completed just before buffer is to be reused. Ordering is again non-overtaking for posts. But not for completions! First receive matches first send etc.

Synchronous Send (non-local)

`MPI_Ssend()`

Synchronized send (always waits for receive to start executing)

Not good for efficiency, but good for debugging.

Buffered send (local)

`MPI_Bsend()`

Specify the buffer, so no reason to wait. Have to ensure buffer is big enough.

For the nonblocking version what is different is the blocking of completion.

Ready send

`MPI_Rsend()`

Can be posted only if matching receive already posted. Otherwise erroneous behavior) (Used to avoid hand-shake and speed up communication)

buffering: requires memory set by user. Can be only done for the send mode. (`MPI_Buffer_attach()`, `MPI_Buffer_detach()`)

`MPI_Isend()`, `MPI_Ibsend()`, `MPI_Irsend()`, `MPI_Issend()`

Buffered, non-blocking asynchronous

Powerful, but requires large system resources.

Use `MPI_Wait()`, `MPI_Test()`, `MPI_Waitall()` to make sure nonblocking sends are completed.

- Standard Send/Receive are blocking.
- `MPI_Recv()` returns only after receive buffer contains requested message
- `MPI_Send()` initiated before or after matching `MPI_Recv` is posted
- For same **source**, **tag**, and **comm** messages are received in order that they are send
- MPI standard does not specify whether buffering is used or not
- Standard mode is the most portable

Deadlock fixes

- Reverse the order of send and receive in some PEs (not all)
- Use a blocking send in buffered mode
- Use a nonblocking send/or receive
- Use `MPI_Sendrecv()`

Collective communications

- Use whenever possible - More stable
- More efficient

Ordering Messages are non-overtaking PE 0 sends to PE 1 ordered in sequence. PE 1 receives are posted in sequence Messages *from same process* are received in the order which they were send. If single thread PE, with no `MPI_CANCEL` wildcards and `MPI_WAITANY` then a safe program has deterministic behavior

```

MPI_Bcast()      - broadcast send (single) buffer to everybody (itself included)
MPI_Reduce()     - all to one with operation min, max, sum
MPI_Allreduce()  - all to all with operation
MPI_Alltoall()   - all to all
MPI_Alltoallv()  - variable length
MPI_Scatter()    - one (many buffers) to all
MPI_Gather()     - all to one
MPI_Scan()       - prefix reduction
MPI_Barrier()    - synchrhonization

```

Broadcast

```

MPI_BCAST( buffer, count, datatype, root, comm )
[ INOUT buffer] starting address of buffer (choice)
[ IN count] number of entries in buffer (integer)
[ IN datatype] data type of buffer (handle)
[ IN root] rank of broadcast root (integer)
[ IN comm] communicator (handle)

```

```

int MPI_Bcast(void* buffer, int count,
              MPI_Datatype datatype, int root, MPI_Comm comm )

```

`MPI_Bcast` broadcasts a message from the process with rank `root` to all processes of the group, itself included. It is called by all members of group using the same arguments for `comm`, `root`. On return, the contents of `root`'s communication buffer has been copied to all processes.

Alltoall

```

MPI_ALLTOALL(sendbuf, sendcount, sendtype, recvbuf, recvcnt, recvtype, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ IN sendcount] number of elements sent to each process (integer)
[ IN sendtype] data type of send buffer elements (handle)
[ OUT recvbuf] address of receive buffer (choice)
[ IN recvcnt] number of elements received from any process (integer)
[ IN recvtype] data type of receive buffer elements (handle)
[ IN comm] communicator (handle)

```

```

int MPI_Alltoall(void* sendbuf, int sendcount,
MPI_Datatype sendtype, void* recvbuf, int recvcnt,
MPI_Datatype recvtype, MPI_Comm comm)

```

MPI_ALLTOALL is an extension of MPI_ALLGATHER to the case where each process sends distinct data to each of the receivers. The *j*th block sent from process *i* is received by process *j* and is placed in the *i*th block of *recvbuf*. The type signature associated with *sendcount*, *sendtype*, at a process must be equal to the type signature associated with *recvcnt*, *recvtype* at any other process. This implies that the amount of data sent must be equal to the amount of data received, pairwise between every pair of processes. As usual, however, the type maps may be different.

Predefined operations for Reduce and scan: maximum,minimum,sum, product, logical operations, max and location, min and location.

Allreduce

includes variants of each of the reduce operations where the result is returned to all processes in the group. MPI requires that all processes participating in these operations receive identical results.

```

MPI_ALLREDUCE( sendbuf, recvbuf, count, datatype, op, comm)
[ IN sendbuf] starting address of send buffer (choice)
[ OUT recvbuf] starting address of receive buffer (choice)
[ IN count] number of elements in send buffer (integer)
[ IN datatype] data type of elements of send buffer (handle)
[ IN op] operation (handle)
[ IN comm] communicator (handle)

```

```

int MPI_Allreduce(void* sendbuf, void* recvbuf, int count,
MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)

```

example:

```

sum[j] = 0.0;
for(j= 0; j<n; j++) sum(j) = sum(j) + a(i)*b(i)
double c; //result
ierr= MPI_Allreduce(&sum, &c, 1, MPI_REAL, MPI_SUM, 0, comm);

```

Example of scan (Segmented scan)

compute:

```

typedef struct {
    double val;
    int log;
} SegScanPair;

```

```

/* the user-defined function
*/
void segScan( SegScanPair *in, SegScanPair *inout, int *len,
             MPI_Datatype *dptr )
{
    int i;
    SegScanPair c;

    for (i=0; i< *len; ++i) {
        if ( in->log == inout->log )
            c.val = in->val + inout->val;
        else
            c.val = inout->val;
        c.log = inout->log;
        *inout = c;
        in++; inout++;
    }
}

```

Note that the `inout` argument to the user-defined function corresponds to the right-hand operand of the operator. When using this operator, we must be careful to specify that it is non-commutative, as in the following.

```

int i,base;
SeqScanPair a, answer;
MPI_Op      myOp;
MPI_Datatype type[2] = {MPI_DOUBLE, MPI_INT};
MPI_Aint    disp[2];
int         blocklen[2] = { 1, 1};
MPI_Datatype sspair;

/* explain to MPI how type SegScanPair is defined
*/
MPI_Address( a, disp);
MPI_Address( a.log, disp+1);
base = disp[0];
for (i=0; i<2; ++i) disp[i] -= base;
MPI_Type_struct( 2, blocklen, disp, type, &sspair );
MPI_Type_commit( &sspair );
/* create the segmented-scan user-op
*/
MPI_Op_create( segScan, False, &myOp );
...
MPI_Scan( a, answer, 1, sspair, myOp, root, comm );

```

Deadlocks in collective communications The following examples illustrate dangerous use of collective routines.
Example 1

```

switch(rank) {
    case 0:
        MPI_Bcast(buf1, count, type, 0, comm);
        MPI_Bcast(buf2, count, type, 1, comm);
        break;

```

```

case 1:
    MPI_Bcast(buf2, count, type, 1, comm);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;

```

Example 2 The following is erroneous.

```

switch(rank) {
case 0:
    MPI_Bcast(buf1, count, type, 0, comm);
    MPI_Send(buf2, count, type, 1, tag, comm);
    break;
case 1:
    MPI_Recv(buf2, count, type, 0, tag, comm, status);
    MPI_Bcast(buf1, count, type, 0, comm);
    break;
}

```

Process zero executes a broadcast, followed by a blocking send operation. Process one first executes a blocking receive that matches the send, followed by broadcast call that matches the broadcast of process zero. This program may deadlock. The broadcast call on process zero may block until process one executes the matching broadcast call, so that the send is not executed. Process one will definitely block on the receive and so, in this case, never executes the broadcast.

Rules

- Collective operations must be executed in the same order at all members of the communication group.
- The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.
- A correct, portable program must invoke collective communications so that deadlock will not occur, whether collective communications are synchronizing or not.
- The relative order of execution of collective operations and point-to-point operations should be such, so that even if the collective operations and the point-to-point operations are synchronizing, no deadlock will occur.

Timings

`MPI_Wtime()` - returns elapsed wall clock time.
call twice and compute difference

Good only for a single process. Clocks are not necessarily the same, not synchronized and may not be comparable across PEs