Intermediate MPI

M. D. Jones, Ph.D.

Center for Computational Research
University at Buffalo
State University of New York

High Performance Computing I, 2014

The Message Passing Model

- A "parallel" calculation in which each process (out of a specified number of processes) works on a local copy of the data, with local variables. Namely, no process is allowed to *directly* access the memory (available data) of another process.
- The mechanism by which individual processes share information (data) is through explicit sending (and receiving) of data between the processes.
- General assumption - a one-to-one mapping of processes to processors (although this is not necessarily always the case).

Upside of MPI

Advantages:

- Very general model (message passing)
- Applicable to widest variety of hardware platforms (SMPs, NOWs, etc.).
- Allows great control over data location and flow in a program.
- Programs can usually achieve higher performance level (scalability).

Downside of MPI

Disadvantages:

- Programmer has to work hard(er) to implement.
- Best performance gains can involve re-engineering the code.
- The MPI standard does not specify mechanism for launching parallel tasks ("task launcher"). Implementation dependent - it can be a bit of a pain.

The MPI Standard(s)

MPI-1	1.0	released in 1994
	1.1	mostly corrections & clarifications in 1995
	1.2	clarifications (& MPI_GET_VERSION function!) in 1997.
	1.3	clarifications/corrections, 2008.
MPI-2	2.0	1997, significant enhancements to **MPI-1**, including `C++` bindings, replace "deprecated" functions of **MPI-1**.
	2.1	2008, mostly clarifications/corrections.
	2.2	2009, more clarifications/corrections.
MPI-3	3.0	2012 major update, but not yet widely implemented.

MPI-1

Major MPI-1 features:

1. Point-to-point Communications
2. Collective Operations
3. Process Groups
4. Communication Domains
5. Process Topologies
6. Environmental Management & Inquiry
7. Profiling Interface
8. `FORTRAN` and `C` Bindings

MPI-2

MPI-2 Enhancements (mostly implemented, widely available in recent implementations):

1. Dynamic Process Management (pretty available)
2. Input/Output (supporting hardware is hardest to find)
3. One-sided Operations (hardest to find, but generally available)
4. `C++` Bindings (generally available, but deprecated!)

MPI-3

MPI-3 major features (not available in any implementations yet):

1. (deprecated) C++ bindings to be removed
2. Extended nonblocking collective operations
3. Extensions to one-sided operations
4. Fortran 2008 bindings

MPI References

- *Using MPI: Portable Programming With the Message Passing Interface*, second edition, W. Gropp, E. Lusk, and A. Skellum (MIT Press, Cambridge, 1999).
- *MPI–The Complete Reference, Vol. 1, The MPI Core*, M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra (MIT Press, Cambridge, 1998).
- *MPI–The Complete Reference, Vol. 2, The MPI Extensions*, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, M. Snir, and J. Dongarra (MIT Press, Cambridge, 1998).

More MPI References

- *The MPI Forum*, `http://www.mpi-forum.org`.
- `http://www.netlib.org/utk/papers/mpi-book/mpi-book.html`, first edition of the title *MPI – The Complete Reference*, also available as a PostScript file.
- A useful online reference to all of the routines and their bindings:
- `http://www-unix.mcs.anl.gov/mpi/www/www3` Note that this is for MPICH 1.2, but it's quite handy.

MPI: "Large" and "Small"

- MPI is Large
  - MPI 1.2 has 128 functions.
  - MPI 2.0 has 152 functions.
- MPI is Small
  - Many programs need to use only about 6 MPI functions.
- MPI is the "right size".
  - Offers enough flexibility that users don't need to master $>$ 150 functions to use it properly.

Some Available MPI Implementations

Some of the more common MPI implementations, and supported network hardware:

MPICH , from ANL - has generally been considered the reference MPI implementation,

MVAPICH , from Ohio State University, ports MPICH to support Infiniband networks,

OpenMPI , latest from LAM and other (FT-MPI, LA-MPI, PACX-MPI) MPI implementation developers, includes mutli-protocol (tcp/ip,IB,etc.) support

and those are just some of the more common free ones ...

Appropriate Times to Use MPI

- When you need a portable parallel API
- When you are writing a parallel library
- When you have data processing that is not conducive to a data parallel approach
- When you care about parallel performance

Appropriate Times NOT to Use MPI

- When you are can just use a parallel library (which may itself be written in MPI).
- When you need only simple threading on data-parallel tasks.
- When you don't need large (many processor, multi-node) parallel speedup.

Basic Features of Message Passing

Message passing codes run the same (usually serial) code on multiple processors, which communicate with one another via library calls which fall into a few general categories:

- Calls to initialize, manage, and terminate communications
- Calls to communicate between two individual processors (point-to-point)
- Calls to communicate among a group of processors (collective)
- Calls to create custom datatypes

I will briefly cover the first three, and present a few concrete examples.

Outline of a Program Using MPI

General outline of any program using MPI:

```
Include MPI header files
Declare variables & Data Structures
Initialize MPI
.
Main program — message passing enabled
.
Terminate MPI
End program
```

MPI Header Files

All MPI programs need to include the MPI header files to define necessary datatypes.

- In **C/C++**:

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
```

- In **FORTRAN 77**

```
program main
implicit none
include "mpif.h"
```

- **Fortran 90/95**

```
program main
implicit none
use MPI
```


MPI Naming Conventions

MPI functions are designed to be as language independent as possible.

- Routine names all begin with **MPI_**:
  - FORTRAN names are typically upper case:

```
call MPI_XXXXXXX(param1,param2,...,IERR)
```

  - C functions use a mixed case:

```
ierr = MPI_Xxxxxxx(param1,param2, ...)
```

- MPI constants are all upper case in both C and FORTRAN:

```
MPI_COMM_WORLD, MPI_REAL, MPI_DOUBLE, ...
```


MPI Routines & Their Return Values

Generally the MPI routines return an error code, using the exit status in C, which can be tested with a predefined success value:

```
int ierr;
...
ierr = MPI_INIT(&argc,&argv);
if (ierr != MPI_SUCCESS) {
   ... exit with an error ...
}
...
```


and in FORTRAN the error code is passed back as the last argument in the MPI subroutine call:

```
integer :: ierr

call MPI_INIT(ierr)
if (ierr.ne.MPI_SUCCESS) STOP "MPI_INIT failed."
```

MPI Handles

- MPI defines its own data structures, which can be referenced by the use through the use of **handles**.
- handles can be returned by MPI routines, and used as arguments to other MPI routines.
- Some examples:

MPI_SUCCESS - Used to test MPI error codes. An integer in both `C` and `FORTRAN`.

MPI_COMM_WORLD - A (pre-defined) communicator consisting of all of the processes. An integer `FORTRAN`, and a `MPI_Comm` object in `C`.

MPI Datatypes

- MPI defines its own datatypes that correspond to typical datatypes in `C` and `FORTRAN`.
- Allows for automatic translation between different representations in a heterogeneous parallel environment.
- You can build your own datatypes from the basic MPI building blocks.
- Actual representation is implementation dependent.
- *Convention*: program variables are usually declared as normal `C` or `FORTRAN` types, and then calls to MPI routines use MPI type names as needed.

MPI Datatypes in C

In `C`, the basic datatypes (and their `ISO C` equivalents) are:

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

MPI Datatypes in `FORTRAN`

In `FORTRAN`, the basic datatypes (and their `FORTRAN` equivalents) are:

MPI Datatype	C Type
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double complex
MPI_LOGICAL	logical
MPI_CHARACTER	character*1
MPI_BYTE	–
MPI_PACKED	–

Initializing & Terminating MPI

- The first MPI routine called by any MPI program *must be* **MPI_INIT**, called once and only once per program.
- **C:**

```
int ierr;
ierr = MPI_INIT(&argc,&argv);
...
```

- **FORTRAN:**

```
integer ierr
call MPI_INIT(ierr)
...
```


MPI Communicators

Definition (MPI Communicator)

A **communicator** is a group of processors that can communicate with each other.

- There can be many communicators
- A given processor can be a member of multiple communicators.
- Within a communicator, the **rank** of a processor is the number (starting at 0) uniquely identifying it within that communicator.

- A processor's rank is used to specify source and destination in message passing calls.
- A processor's rank can be different in different communicators.
- `MPI_COMM_WORLD` is a pre-defined communicator encompassing all of the processes. Additional communicators can be defined to define subsets of this group.

More on MPI Communicators

Typically a program executes two MPI calls immediately after `MPI_INIT` to determine each processor's rank:

- **C:**

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);
```

- **FORTRAN:**

```
MPI_COMM_RANK(comm, rank, ierr)
MPI_COMM_SIZE(comm, size, ierr)
```

where `rank` and `size` are integers returned with (obviously) the rank and extent (0:number of processors-1).

Simple MPI Program in C

We have already covered enough material to write the simplest of MPI programs: here is one in C:

```
#include <stdio.h>
#include "mpi.h"

int main( int argc, char **argv)
{
    int ierr,myid, numprocs;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf("Hello World, I am Process %d of %d\n", myid, numprocs);
    MPI_Finalize();
}
```


Six Function MPI

Many MPI codes can get away with using only the six most frequently used routines:

- MPI_INIT for intialization
- MPI_COMM_SIZE size of communicator
- MPI_COMM_RANK rank in communicator
- MPI_SEND send message
- MPI_RECV receive message
- MPI_FINALIZE shut down communicator

Basic P2P in MPI

Basic features:

- In MPI 1.2, only “two-sided” communications are allowed, requiring an explicit **send** and **receive**. (2.0 allows for “one-sided” communications, i.e. **get** and **put**).
- Point-to-point (or P2P) communication is explicitly two-sided, and the message will not be sent without the active participation of both processes.
- A *message* generically consists of an envelope (tags indicating source and destination) and a body (data being transferred).
- **Fundamental** - almost all of the MPI comms are built around point-to-point operations.

MPI Message Bodies

MPI uses three points to describe a message body:

1. **buffer**: the starting location in memory where the data is to be found.

   C: actual address of an array element
   FORTRAN: name of the array element

2. **datatype**: the type of data to be sent. Commonly one of the predefined types, e.g. MPI_REAL. Can also be a user defined datatype, allowing great flexibility in defining message content for more advanced applications.
3. **count**: number of items being sent.

MPI standardizes the elementary datatypes, avoiding having the developer have to worry about numerical representation.

MPI Message Envelopes

MPI message wrappers have the following general attributes:

- **communicator** - the group of processes to which the sending and receiving process belong.
- **source** - originating process
- **destination** - receiving process
- **tag** - message identifier, allows program to label classes of messages (e.g. one for name data, another for place data, status, etc.)

Blocking vs. Non-Blocking

- **blocking** routine does not return until operation is complete.
  - blocking sends, for example, ensure that it is safe to overwrite the sent data.
  - blocking receives, the data is here and ready for use.
- **nonblocking** routine returns immediately, with no info about completion. Can test later for success/failure of operation. In the interim, the process is free to go on to other tasks.

Point-to-point Semantics

For MPI sends, there are four available **modes**:

- **standard** - no guarantee that the receive has started.
- **synchronous** - complete when receipt has been acknowledged.
- **buffered** - complete when data has been copied to local buffer. No implication about receipt.
- **ready** - the user asserts that the matching receive has been posted (allows user to gain performance).

MPI receives are easier - they are complete when the data has arrived and is ready for use.

Blocking Send

MPI_SEND

```
MPI_SEND(buff,count,datatype,dest,tag,comm)
```

- **buff** (IN), initial address of message buffer
- **count** (IN), number of entries to send (int)
- **datatype** (IN), datatype of each entry (handle)
- **dest** (IN), rank of destination (int)
- **tag** (IN), message tag (int)
- **comm** (IN), communicator (handle)

Blocking Receive

MPI_RECV

```
MPI_RECV(buff,count,datatype,source,tag,comm,
         status)
```

- **buff** (IN), initial address of message buffer
- **count** (IN), number of entries to send (int)
- **datatype** (IN), datatype of each entry (handle)
- **source** (IN), rank of source (int)
- **tag** (IN), message tag (int)
- **comm** (IN), communicator (handle)
- **status** (OUT), return status (Status)

Blocking Send/Receive Restrictions

- *source, tag,* and *comm* must match those of a pending message for the message to be received. Wildcards can be used for source and tag, but not communicator.
- An error will be returned if the message buffer exceeds that allowed for by the receive.
- It is the user's responsibility to ensure that the send/receive datatypes agree - if they do not, the results are undefined.

Status of a Receive

More information about message reception is available by examining the status returned by the call to `MPI_RECV`. **C:**
status is a structure of type `MPI_STATUS` that contains at minimum the three fields:

1. `MPI_SOURCE`
2. `MPI_TAG`
3. `MPI_ERROR`

FORTRAN:
status is an integer array of length `MPI_STATUS_SIZE`. `MPI_SOURCE`, `MPI_TAG`, and `MPI_ERROR` are indices of entries that store the source, tag, and error fields.

MPI_GET_COUNT

The routine `MPI_GET_COUNT` is an auxiliary routine that allows you to test the amount of data received:

MPI_GET_COUNT

```
MPI_GET_COUNT(status,datatype,count)
```

- **status** (IN), return status of receive (Status)
- **datatype** (IN), datatype of each receive buffer entry (handle)
- **count** (OUT), number of entries received (int)

`MPI_UNDEFINED` will be returned in the event of an error.

A Simple Send/Receive Example

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  int i,ierr,rank,size,dest,source,from,to,count,tag;
  int stat_count,stat_source,stat_tag;
  float data[100];
  MPI_Status status;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  printf("I am process %d of %d\n",rank,size);
  dest=size-1;
  source=0;
  if (rank == source) {  /* Initialize and Send Data */
    to = dest;
    count = 100;
    tag = 11;
    for (i=0;i<=99;i++) data[i]=i;
    ierr = MPI_Send(data,count,MPI_REAL,to,tag,MPI_COMM_WORLD);
  }
```



```c
  else if (rank == dest) { /* Receive & Check Data */
    tag = MPI_ANY_TAG;  /* wildcard */
    count = 100;
    from = MPI_ANY_SOURCE; /* another wildcard */
    ierr = MPI_Recv(data,count,MPI_REAL,from,tag,MPI_COMM_WORLD,&status);
    ierr = MPI_Get_count(&status,MPI_REAL,&stat_count);
    stat_source = status.MPI_SOURCE;
    stat_tag = status.MPI_TAG;
    printf("Status of receive: dest=%d, source=%d, tag=%d,
    count=%d\n",rank,stat_source,stat_tag,stat_count);
  }
  ierr = MPI_Finalize();
  return 0;
}
```


Semantics of Blocking Point-to-point

- For `MPI_RECV` "completion" is easy - the data is here, and can now be used.
- A bit trickier for `MPI_SEND` - completes when the data has been stored away such that the program is free to overwrite the send buffer. It can be **non-local** - the data could be copied directly to the receive buffer, or it could be stored in a local buffer, in which case the send could return before the receive is initiated (thereby allowing even a single threaded send process to continue).

Message Buffering

- Decouples send/receive operations.
- Entails added memory-memory copying (additional overhead)
- Amount of buffering is application and implementation dependent:
  - applications can choose communication **modes** - and gain finer control (with additional hazards) over messaging behavior.
  - the **standard mode** is implementation dependent

More on Message Buffering

- A *properly coded* program will not fail if the buffer throttles back on the sends, thereby causing blocking (imagine the assembly line controlled by the rate at which the final inspector signs off on each item).
- An *improperly coded* program can **deadlock** ...

Deadlock

- safe MPI programs do not rely on system buffering for success.
- Any system will eventually run out of buffer space as message buffer sizes are increased.
- Users are free to take advantage of knowledge of an implementation's buffering policy to increase performance, but they do so by relaxing the margin for safety (as well as decreasing portability, of course).

Deadlock Examples

Safe code (no buffering requirements):

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.eq.0) THEN
   CALL MPI_SEND(sbuff,count,MPI_REAL,1,tag,comm,ierr)
   CALL MPI_RECV(rbuff,count,MPI_REAL,1,tag,comm,status,ierr)
ELSE IF (rank.eq.1) THEN
   CALL MPI_RECV(rbuff,count,MPI_REAL,0,tag,comm,status,ierr)
   CALL MPI_SEND(sbuff,count,MPI_REAL,0,tag,comm,ierr)
END IF
```


Complete & total deadlock (oops!):

```
CALL MPI_COMM_RANK(comm, rank, ierr)
IF (rank.eq.0) THEN
   CALL MPI_RECV(rbuff,count,MPI_REAL,1,tag,comm,status,ierr)
   CALL MPI_SEND(sbuff,count,MPI_REAL,1,tag,comm,ierr)
ELSE IF (rank.eq.1) THEN
   CALL MPI_RECV(rbuff,count,MPI_REAL,0,tag,comm,status,ierr)
   CALL MPI_SEND(sbuff,count,MPI_REAL,0,tag,comm,ierr)
END IF
```

Buffering dependent:

```
CALL MPI_COMM_RANK(comm,rank,ierr)
IF (rank.eq.0) THEN
   CALL MPI_SEND(sbuff,count,MPI_REAL,1,tag,comm,ierr)
   CALL MPI_RECV(rbuff,count,MPI_REAL,1,tag,comm,status,ierr)
ELSE IF (rank.eq.1) THEN
   CALL MPI_SEND(sbuff,count,MPI_REAL,0,tag,comm,ierr)
   CALL MPI_RECV(rbuff,count,MPI_REAL,0,tag,comm,status,ierr)
END IF
```

for this last buffer-dependent example, one of the sends must buffer and return - if the buffer can not hold `count` reals, deadlock occurs. Non-blocking communications can be used to avoid buffering, and possibly increase performance.

Non-blocking Sends & Receives

Advantages:

1. Easier to write code that doesn't deadlock
2. Can mask latency in high latency environments by posting receives early (requires a careful attention to detail).

Disadvantages:

1. Makes code quite a bit more complex.
2. Harder to debug and maintain code.

Non-blocking Send Syntax

MPI_ISEND

```
MPI_ISEND(buff,count,datatype,dest,tag,comm,
          request)
```

- **buff** (IN), initial address of message buffer
- **count** (IN), number of entries to send (int)
- **datatype** (IN), datatype of each entry (handle)
- **dest** (IN), rank of destination (int)
- **tag** (IN), message tag (int)
- **comm** (IN), communicator (handle)
- **request** (OUT), request handle (handle)

Non-blocking Receive Syntax

MPI_IRECV

```
MPI_IRECV(buff,count,datatype,dest,tag,comm,
          request)
```

- **buff** (OUT), initial address of message buffer
- **count** (IN), number of entries to send (int)
- **datatype** (IN), datatype of each entry (handle)
- **dest** (IN), rank of destination (int)
- **tag** (IN), message tag (int)
- **comm** (IN), communicator (handle)
- **request** (OUT), request handle (handle)

Non-blocking Send/Receive Details

- The `request` handle is used to query the status of the communication or to wait for its completion.
- The user must not overwrite the send buffer until the send is complete, nor use elements of the receiving buffer before the receive is complete (intuitively obvious, but worth stating explicitly).

Non-blocking Send/Receive Completion Operations

MPI_WAIT

`MPI_WAIT(request,status)`

- request (INOUT), request handle (handle)
- status (OUT), status object (status)

MPI_TEST

`MPI_TEST(request,flag,status)`

- request (INOUT), request handle (handle)
- flag (OUT), true if operation complete (logical)
- status (OUT), status status object (Status)

Completion Operations Details

- The request handle should identify a previously posted send or receive
- `MPI_WAIT` returns when the operation is complete, and the status is returned for a receive (for a send, may contain a separate error code for the send operation).
- `MPI_TEST` returns immediately, with flag = true if posted operation corresponding to the request handle is complete (and status output similar to `MPI_WAIT`).

A Non-blocking Send/Recv Example

```c
#include <stdio.h>
#include "mpi.h"
int main(int argc, char **argv)
{
  int rank,nprocs,ierr,stat_count;
  MPI_Request request;
  MPI_Status status;
  float a[100],b[100];

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  if (rank == 0) {
    MPI_Irecv(b,100,MPI_REAL,1,19,MPI_COMM_WORLD,&request);
    MPI_Send(a,100,MPI_REAL,1,17,MPI_COMM_WORLD);
    MPI_Wait(&request,&status);
  }
  else if (rank == 1) {
    MPI_Irecv(b,100,MPI_REAL,0,17,MPI_COMM_WORLD,&request);
    MPI_Send(a,100,MPI_REAL,0,19,MPI_COMM_WORLD);
    MPI_Wait(&request,&status);
  }
  MPI_Get_count(&status,MPI_REAL,&stat_count);
  printf("Exchange complete: process %d of %d\n",rank,nprocs);
  printf("  source %d, tag %d, count %d\n",status.MPI_SOURCE,status.MPI_TAG
             ,stat_count);

  MPI_Finalize();
```

More About Send Modes

- 1 receive mode, 4 send modes
  1. **standard** - used thus far, implementation dependent choice of asynchronous buffer transfer, or synchronous direct transfer. (rationale - MPI makes a better low-level choice)
  2. **synchronous** - synchronize sending and receiving process. when a synchronous send is completed, the user can assume that the receive has begun.
  3. **ready** - matching receive has already been posted, else the result is undefined. Can save time and overhead, but requires a very precise knowledge of algorithm and its execution.
  4. **buffered** - force buffering - user is also responsible for maintaining the buffer. Result is undefined if buffer is insufficient. (see `MPI_BUFFER_ATTACH` and `MPI_BUFFER_DETACH`).

Send Routines for Different Modes

Standard	MPI_SEND	MPI_ISEND
Synchronous	MPI_SSEND	MPI_ISSEND
Ready	MPI_RSEND	MPI_IRSEND
Buffered	MPI_BSEND	MPI_IBSEND

Call syntax is the same as for `MPI_SEND` and `MPI_ISEND`.

MPI Collective Communications

Routines that allow groups of processes to communicate (e.g. one-to-many or many-to-one). Although they can usually be built from point-to-point calls, intrinsic collective routines allow for

- simplified code - one routine replacing many point-to-point calls
- optimized forms - implementation can take advantage of faster algorithms

Categories:

barrier synchronization
broadcast
gather
scatter
reduction

Barrier Synchronization

A very simple MPI routine provides the ability to block the calling process until all processes have called it:

MPI_BARRIER

```
MPI_BARRIER( comm )
```

comm (IN), communicator (handle)

returns only when all group members have entered the call.

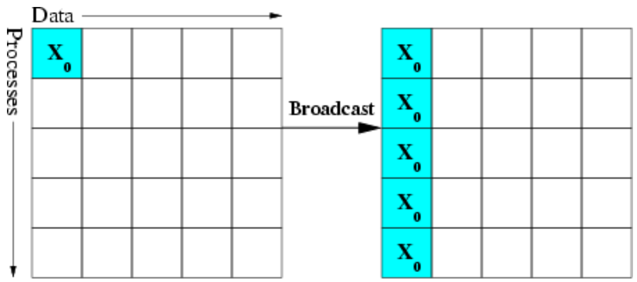

Figure : Broadcast in action - 5 data elements on 5 processes.

Broadcast

MPI_BCAST

`MPI_BCAST(buffer,count,datatype,root,comm)`

- buffer (INOUT), starting address of buffer (choice)
- count (IN), number of entries in buffer (int)
- datatype (IN), data type of buffer (handle)
- root (IN), rank of broadcasting process (int)
- comm (IN), communicator (handle)

Broadcast Details

- broadcast a message from the process to all members of the group (including itself).
- root must have identical value on all processes.
- comm must be the same intra-group domain.

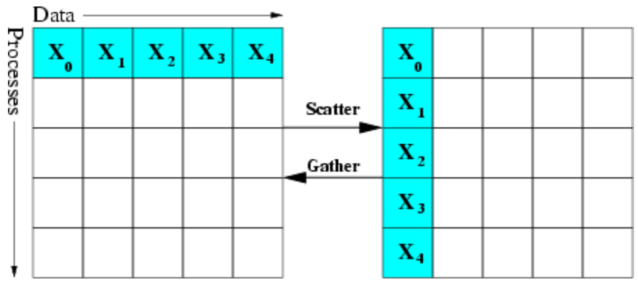

Figure : Scatter/Gather in action - 5 data elements on 5 processes.

Gather

MPI_GATHER

```
MPI_GATHER(sendbuffer, sendcount, sendtype,recvbuffer,recvcount,
           recvtype,root,comm)
```

sendbuffer	(IN), starting address of send buffer (choice)
sendcount	(IN), number of entries in send buffer (int)
sendtype	(IN), data type of send buffer (handle)
recvbuffer	(OUT), starting address of receive buffer (choice)
recvcount	(IN), number of entries any single receive (int)
recvtype	(IN), data type of receive buffer elements (handle)
root	(IN), rank of receiving process (int)
comm	(IN), communicator (handle)

Gather Details

- each process sends contents of send buffer to root.
- root stores receives in rank order (as if there were N posted receives of sends from each process).

Scatter

MPI_SCATTER

```
MPI_SCATTER( sendbuffer, sendcount, sendtype, recvbuffer,
             recvcount,recvtype, root, comm)
```

sendbuffer	(IN), starting address of send buffer (choice)
sendcount	(IN), number of entries sent to each process (int)
sendtype	(IN), data type of send buffer elements (handle)
recvbuffer	(OUT), starting address of receive buffer (choice)
recvcount	(IN), number of entries any single receive (int)
recvtype	(IN), data type of receive buffer elements (handle)
root	(IN), rank of receiving process (int)
comm	(IN), communicator (handle)

Scatter Details

- basically the reverse operation to `MPI_GATHER`.
- a one-to-all operation in which each recipient get a different chunk.

Gather Example

```c
MPI_Comm comm;
int myrank, nprocs, root, iarray[100];
...
MPI_Comm_rank(comm,&myrank);
if (myrank == root) {
   MPI_Comm_size(comm,&nprocs);
   rbuff = (int *)malloc(nprocs*100*sizeof(int));
}
MPI_Gather(iarray,100,MPI_INT,rbuf,100,MPI_INT,root,comm);
...
```


Reduction

MPI_REDUCE

```
MPI_REDUCE( sendbuffer, recvbuffer, count,
        datatype, op, root, comm)
```

- **sendbuffer** (IN), starting address of send buffer (choice)
- **recvbuffer** (OUT), starting address of receive buffer (choice)
- **count** (IN), number of entries in buffer (int)
- **datatype** (IN), data type of buffer (handle)
- **op** (IN), reduce operation (handle)
- **root** (IN), rank of broadcasting process (int)
- **comm** (IN), communicator (handle)

Reduce Details

- combine elements provided in sendbuffer of each process and use op to return combined value in recvbuffer of root process.

Predefined Reduction Operations

MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LAND	logical and
MPI_BAND	bit-wise and
MPI_LOR	logical or
MPI_BOR	bit-wise or
MPI_LXOR	logical xor
MPI_BXOR	bit-wise xor
MPI_MINLOC	min value and location
MPI_MAXLOC	max value and location

More (Advanced) Collective Ops

MPI_ALLGATHER - gather + broadcast

MPI_ALLTOALL - each process sends different subset of data to each receiver

MPI_ALLREDUCE - combine elements of each input buffer, store output in receive buffer of all group members.

User Defined Reduction Ops - you can define your own reduction operations

Gather/Scatter Vector Ops - allows a varying count of data from or to each process in a gather or scatter operation (MPI_GATHERV/MPI_SCATTERV)

MPI_SCAN - prefix reduction on data throughout the comm, returns reduction of values of all processes.

MPI_REDUCE_SCATTER - combination of MPI_REDUCE and MPI_SCATTERV.

Process Startup

- Single most confusing aspect of MPI for most new users
- Implementation dependent! with many implementation specific options, flags, etc.
- Consult the documentation for the MPI implementation that you are using.

Some Examples Using MPI Task Launchers

SGI Origin/Altix (intra-machine):

```
mpirun −np <np> [options] <progname> [progname options]
```

MPICH-1 ch_p4 device:

```
mpirun −machinefile <filename> −np <np> [options] <progname> [args]
```

Sun HPC Tools:

```
mprun −l ``nodename [nproc] [,nodename [nproc],...] [options] <executable> [args]
```

IBM AIX POE:

```
poe ./a.out −nodes [nnodes] −tasks_per_node [ntasks] [options]
```

OSC's PBS/Torque based mpiexec:

```
mpiexec [−pernode] [−kill] [options] <executable> [args]
```


Deprecated form MPICH2/MVAPICH2/Intel MPI (pre-2012)

```
NNODES=`cat $PBS_NODEFILE | uniq | wc −l`
NPROCS=`cat $PBS_NODEFILE | wc −l`
mpdboot −n $NNODES −f $PBS_NODEFILE −v
mpdtrace
mpiexec −np $NPROCS −envall ./my_executable
mpdallexit
```

If at all possible, when you are running using a **resource manager** (Slurm, PBS/Torque, etc.) you will want to use a task launching mechanism integrated with that resource manager (more on that later specifically for UB/CCR).

```
export I_MPI_PMI_LIBRARY=/usr/lib64/libpmi.so # Intel MPI Specific
srun ./mpi−cart−ex
```

Getting Implementation Info from MPI

MPI_GET_VERSION

`MPI_GET_VERSION(version,subversion)`

version	(OUT), version number (int)
subversion	(OUT), subversion number (int)

- Not exactly critical for programming, but a nice function for determining what version of MPI you are using (especially when the documentation for your machine is poor).

Where am I running?

MPI_GET_PROCESSOR_NAME

`MPI_GET_PROCESSOR_NAME(name, resultlen)`

name	(OUT), A unique specifier for the actual node (string)
resultlem	(OUT), Length (in printable chars) of the reslut in name (int)

- returns the name of the processor on which it was called at the moment of the call.
- name should have storage that is at least `MPI_MAX_PROCESSOR_NAME` characters long.

Timing & Synchronization

MPI_WTIME

`MPI_WTIME()`

- double precision value returned representing elapsed wall clock time from some point in the past (origin guaranteed not to change during process execution time).
- A portable timing function (try finding another!) - can be high resolution, provided it has some hardware support.

Testing the resolution of `MPI_WTIME`:

MPI_WTICK

MPI_WTICK()

- double precision value returned which is the resolution of `MPI_WTIME` in seconds.
- hardware dependent, of course - if a high resolution timer is available, it should be accessible through `MPI_WTIME`.

Common `MPI_Wtime` usage:

```
double time0,time1;
...
time0 = MPI_Wtime();
...
/* code to be timed */
...
time1 = MPI_Wtime();
printf("Time interval = %f seconds\n",time1-time0);
```


More About MPI Error Codes

MPI_ERROR_STRING

`MPI_ERROR_STRING(errorcode, string, resultlen)`

- **errorcode** (IN), Error code returned by an MPI routine (int)
- **string** (OUT), Text that corresponds to errorcode (string)
- **resultlen** (OUT), Length (in printable chars) of result returned in string (int)

- Most error codes in MPI are implementation dependent
- `MPI_ERROR_STRING` provides information on the type of MPI exception that occurred.
- argument string must have storage that is at least `MPI_MAX_ERROR_STRING` characters.

MPI Profiling Hooks

- The MPI profiling interface is designed for authors of profiling tools, such that they will not need access to a particular implementation's source code (which a vendor may not wish to release).
- Many profiling tools exist:
  1. **Vampir**, commercial offering (originally developed at Research Center Jülich and Techniche Universität Dresden)
  2. **Intel Trace Analyzer and Visualizer (ITAC)**, part of Intel's cluster tools
  3. **jumpshot** (MPICH)
  4. **Allinea DDT**, primarily a debugger, has a **MAP** product that also does profiling.
- Consult your profiling tools of choice for detailed usage.

More Advanced MPI Topics

Advanced MPI topics not covered thus far:

- User defined data types
- Communicators and Groups
- Process Topologies
- MPI-2 Features
  - MPI-I/O
  - Dynamic process management (`MPI_Spawn`)
  - One-sided communications (get/put)
- MPI-3 Features