Parallel Programming & Cluster Computing

Distributed Multiprocessing

David Joiner, Kean University
Tom Murphy, Contra Costa College
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College
Kay Wanous, Earlham College

SC09 Education Program, Louisiana State University, July 5-11 2009
Message = Envelope+Contents

MPI_Send(message, strlen(message) + 1,
    MPI_CHAR, destination, tag,
    MPI_COMM_WORLD);

When MPI sends a message, it doesn’t just send the contents; it also sends an “envelope” describing the contents:

- **Size** (number of elements of data type)
- **Data type**
- **Source**: rank of sending process
- **Destination**: rank of process to receive
- **Tag** (message ID)
- **Communicator** (for example, MPI_COMM_WORLD)
### MPI Data Types

<table>
<thead>
<tr>
<th>C</th>
<th>Fortran</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>MPI_CHAR</td>
</tr>
<tr>
<td></td>
<td>CHARACTER</td>
</tr>
<tr>
<td>int</td>
<td>MPI_INT</td>
</tr>
<tr>
<td></td>
<td>INTEGER</td>
</tr>
<tr>
<td>float</td>
<td>MPI_FLOAT</td>
</tr>
<tr>
<td></td>
<td>REAL</td>
</tr>
<tr>
<td>double</td>
<td>MPI_DOUBLE</td>
</tr>
<tr>
<td></td>
<td>DOUBLE PRECISION</td>
</tr>
</tbody>
</table>

MPI supports several other data types, but most are variations of these, and probably these are all you’ll use.
Message Tags

My daughter was born in mid-December.

So, if I give her a present in December, how does she know which of these it’s for?

- Her birthday
- Christmas
- Hanukah

She knows because of the tag on the present:

- A little cake and candles means birthday
- A little tree or a Santa means Christmas
- A little menorah means Hanukah
Message Tags

```c
for (source = 0; source < num_procs; source++) {
    if (source != server_rank) {
        mpi_error_code =
            MPI_Recv(message, maximum_message_length + 1,
                    MPI_CHAR, source, tag,
                    MPI_COMM_WORLD, &status);
        fprintf(stderr, "%s\n", message);
    } /* if (source != server_rank) */
} /* for source */
```

The greetings are **printed** in **deterministic** order not because messages are sent and received in order, but because each has a **tag** (message identifier), and **MPI_Recv** asks for a specific message (by tag) from a specific source (by rank).
Parallelism is Nondeterministic

```c
for (source = 0; source < num_procs; source++) {
    if (source != server_rank) {
        mpi_error_code =
            MPI_Recv(message, maximum_message_length + 1,
            MPI_CHAR, MPI_ANY_SOURCE, tag,
            MPI_COMM_WORLD, &status);
        fprintf(stderr, "%s\n", message);
    } /* if (source != server_rank) */
} /* for source */
```

But here the greetings are **printed** in **non-deterministic** order.
Communicators

An MPI communicator is a collection of processes that can send messages to each other.

MPI_COMM_WORLD is the default communicator; it contains all of the processes. It’s probably the only one you’ll need. Some libraries create special library-only communicators, which can simplify keeping track of message tags.
Broadcasting

What happens if one process has data that everyone else needs to know?

For example, what if the server process needs to send an input value to the others?

\[ \texttt{MPI\_Bcast(length, 1, MPI\_INTEGER, source, MPI\_COMM\_WORLD);} \]

Note that \texttt{MPI\_Bcast} doesn’t use a tag, and that the call is the same for both the sender and all of the receivers.

All processes have to call \texttt{MPI\_Bcast} at the same time; everyone waits until everyone is done.
Broadcast Example: Setup

```fortran
PROGRAM broadcast
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER, PARAMETER :: server = 0
  INTEGER, PARAMETER :: source = server
  INTEGER, DIMENSION(:), ALLOCATABLE :: array
  INTEGER :: length, memory_status
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank, &
                    mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs, &
                     mpi_error_code)

  [input]
  [broadcast]
  CALL MPI_Finalize(mpi_error_code)

END PROGRAM broadcast
```
Broadcast Example: Input

```fortran
PROGRAM broadcast
    IMPLICIT NONE
    INCLUDE "mpif.h"
    INTEGER,PARAMETER :: server = 0
    INTEGER,PARAMETER :: source = server
    INTEGER,DIMENSION(:),ALLOCATABLE :: array
    INTEGER :: length, memory_status
    INTEGER :: num_procs, my_rank, mpi_error_code

    [MPI startup]
    IF (my_rank == server) THEN
        OPEN (UNIT=99,FILE="broadcast_in.txt")
        READ (99,*) length
        CLOSE (UNIT=99)
        ALLOCATE(array(length), STAT=memory_status)
        array(1:length) = 0
    END IF !! (my_rank == server)...ELSE

    [broadcast]
    CALL MPI_Finalize(mpi_error_code)

END PROGRAM broadcast
```
Broadcast Example: Broadcast

```fortran
PROGRAM broadcast
    IMPLICIT NONE
    INCLUDE "mpif.h"
    INTEGER, PARAMETER :: server = 0
    INTEGER, PARAMETER :: source = server
    [other declarations]
    [MPI startup and input]
    IF (num_procs > 1) THEN
        CALL MPI_Bcast(length, 1, MPI_INTEGER, source, &
                       MPI_COMM_WORLD, mpi_error_code)
        IF (my_rank /= server) THEN
            ALLOCATE(array(length), STAT=memory_status)
            END IF !! (my_rank /= server)
        CALL MPI_Bcast(array, length, MPI_INTEGER, source, &
                       MPI_COMM_WORLD, mpi_error_code)
        WRITE (0,*) my_rank, "": broadcast length = ", length
    END IF !! (num_procs > 1)
    CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```
Broadcast Compile & Run

% mpif90 -o broadcast broadcast.f90
% mpirun -np 4 broadcast

  0 : broadcast length =  16777216
  1 : broadcast length =  16777216
  2 : broadcast length =  16777216
  3 : broadcast length =  16777216
A **reduction** converts an array to a scalar: for example, sum, product, minimum value, maximum value, Boolean AND, Boolean OR, etc.

Reductions are so common, and so important, that MPI has two routines to handle them:

**MPI_Reduce**: sends result to a single specified process

**MPI_Allreduce**: sends result to all processes (and therefore takes longer)
Reduction Example

```fortran
PROGRAM reduce
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER :: server = 0
  INTEGER :: value, value_sum
  INTEGER :: num_procs, my_rank, mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, my_rank, mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, num_procs, mpi_error_code)
  value_sum = 0
  value     = my_rank * num_procs
  CALL MPI_Reduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
                   server, MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*), my_rank, ': reduce  value_sum = ', value_sum
  CALL MPI_Allreduce(value, value_sum, 1, MPI_INT, MPI_SUM, &
                     MPI_COMM_WORLD, mpi_error_code)
  WRITE (0,*), my_rank, ': allreduce value_sum = ', value_sum
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM reduce
```
Compiling and Running

% mpif90 -o reduce reduce.f90
%
% mpirun -np 4 reduce

3 : reduce value_sum = 0
1 : reduce value_sum = 0
2 : reduce value_sum = 0
0 : reduce value_sum = 24
0 : allreduce value_sum = 24
1 : allreduce value_sum = 24
2 : allreduce value_sum = 24
3 : allreduce value_sum = 24
Why Two Reduction Routines?

MPI has two reduction routines because of the high cost of each communication.

If only one process needs the result, then it doesn’t make sense to pay the cost of sending the result to all processes. But if all processes need the result, then it may be cheaper to reduce to all processes than to reduce to a single process and then broadcast to all.
Non-blocking Communication

MPI allows a process to start a send, then go on and do work while the message is in transit.

This is called *non-blocking* or *immediate* communication. Here, “immediate” refers to the fact that the call to the MPI routine returns immediately rather than waiting for the communication to complete.
Immediate Send

```c
mpi_error_code =
    MPI_Isend(array, size, MPI_FLOAT,
              destination, tag, communicator, request);

Likewise:
mpi_error_code =
    MPI_Irecv(array, size, MPI_FLOAT,
              source, tag, communicator, request);

This call starts the send/receive, but the send/receive won’t be complete until:
MPI_Wait(request, status);

What’s the advantage of this?
Communication Hiding

In between the call to `MPI_Isend/Irecv` and the call to `MPI_Wait`, both processes can do work!

If that work takes at least as much time as the communication, then the cost of the communication is effectively zero, since the communication won’t affect how much work gets done.

This is called **communication hiding**.
Rule of Thumb for Hiding

When you want to hide communication:

- as soon as you calculate the data, send it;
- don’t receive it until you need it.

That way, the communication has the maximal amount of time to happen in background (behind the scenes).
SC09 Summer Workshops

1. May 17-23: Oklahoma State U: Computational Chemistry
2. May 25-30: Calvin Coll (MI): Intro to Computational Thinking
3. June 7-13: U Cal Merced: Computational Biology
4. June 7-13: Kean U (NJ): Parallel Progrmg & Cluster Comp
5. July 5-11: Atlanta U Ctr: Intro to Computational Thinking
8. July 12-18: Ohio Supercomp Ctr: Computational Engineering
9. Aug 2-8: U Arkansas: Intro to Computational Thinking
Thanks for your attention!

Questions?
References
