Introduction to Parallel Programming & Cluster Computing

MPI Collective Communications

Josh Alexander, University of Oklahoma
Ivan Babic, Earlham College
Andrew Fitz Gibbon, Shodor Education Foundation Inc.
Henry Neeman, University of Oklahoma
Charlie Peck, Earlham College
Skylar Thompson, University of Washington
Aaron Weeden, Earlham College

Sunday June 26 – Friday July 1 2011

Co-sponsored by SC11

EARLHAM COLLEGE

SHODOR

Co-sponsored by ID,NM,NV EPSCoR

UNIVERSITY of WASHINGTON

Idaho State UNIVERSITY

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

H.323 (Polycom etc)

If you want to use H.323 videoconferencing – for example, Polycom – then:

- If you ARE already registered with the OneNet gatekeeper, dial 2500409.
- If you AREN'T registered with the OneNet gatekeeper (which is probably the case), then:
  - Dial **`164.58.250.47`**
  - When asked for the conference ID, enter:

    `#0409#`

Many thanks to Roger Holder and OneNet for providing this.

H.323 from Internet Explorer

From a Windows PC running Internet Explorer:

1. You **MUST** have the ability to install software on the PC (or have someone install it for you).
2. Download and install the latest Java Runtime Environment (JRE) from here (click on the Java Download icon, because that install package includes both the JRE and other components).
3. Download and install this video decoder.
4. Start Internet Explorer.
5. Copy-and-paste this URL into your IE window:
   http://164.58.250.47/
6. When that webpage loads, in the upper left, click on "Streaming".
7. In the textbox labeled Sign-in Name, type your name.
8. In the textbox labeled Conference ID, type this:
   0409
9. Click on "Stream this conference".
10. When that webpage loads, you may see, at the very top, a bar offering you options. If so, click on it and choose "Install this add-on."

EVO

There's a quick description of how to use EVO on the workshop logistics webpage.

Phone Bridge

If all else fails, you can call into our toll free phone bridge:

1-800-832-0736

* 623 2874 #

Please mute yourself and use the phone to listen.

Don't worry, we'll call out slide numbers as we go.

Please use the phone bridge **ONLY** if you cannot connect any other way: the phone bridge is charged per connection per minute, so our preference is to minimize the number of connections.

Many thanks to OU Information Technology for providing the toll free phone bridge.

Please Mute Yourself

No matter how you connect, please mute yourself, so that we cannot hear you.

At ISU and UW, we will turn off the sound on all conferencing technologies.

That way, we won't have problems with echo cancellation.

Of course, that means we cannot hear questions.

So for questions, you'll need to send some kind of text.

Thanks for helping!

- OSCER operations staff (Brandon George, Dave Akin, Brett Zimmerman, Josh Alexander, Patrick Calhoun)
- Kevin Blake, OU IT (videographer)
- James Deaton and Roger Holder, OneNet
- Keith Weber, Abel Clark and Qifeng Wu, Idaho State U Pocatello
- Nancy Glenn, Idaho State U Boise
- Jeff Gardner and Marya Dominik, U Washington
- Ken Gamradt, South Dakota State U
- Jeff Rufinus, Widener U
- Scott Lathrop, SC11 General Chair
- Donna Cappo, ACM
- Bob Panoff, Jack Parkin and Joyce South, Shodor Education Foundation Inc
- ID, NM, NV EPSCoR (co-sponsors)
- SC11 conference (co-sponsors)

Questions via Text: Piazza

Ask questions via:

http://www.piazza.com/

All questions will be read out loud and then answered out loud.

NOTE: Because of image-and-likeness rules, people attending remotely offsite via videoconferencing **CANNOT** ask questions via voice.

This is an experiment!

It's the nature of these kinds of videoconferences that **FAILURES ARE GUARANTEED TO HAPPEN! NO PROMISES!**

So, please bear with us. Hopefully everything will work out well enough.

If you lose your connection, you can retry the same kind of connection, or try connecting another way.

Remember, if all else fails, you always have the toll free phone bridge to fall back on.

Collective Communications

Point to Point Always Works

- **`MPI_Send`** and **`MPI_Recv`** are known as "point to point" communications: they communicate from one MPI process to another MPI process.
- But, what if you want to communicate like one of these?
  - one to many
  - many to one
  - many to many
- These are known as ***collective communications***.
- **`MPI_Send`** and **`MPI_Recv`** can accomplish any and all of these – but should you use them that way?

Point to Point Isn't Always Good

- We're interested in ***collective communications***:
  - one to many
  - many to one
  - many to many
- In principle, `MPI_Send` and `MPI_Recv` can accomplish any and all of these.
- But that may be:
  - inefficient;
  - inconvenient and cumbersome to code.
- So, the designers of MPI came up with routines that perform these collective communications for you.

Collective Communications

- MPI_Bcast
- MPI_Reduce, MPI_Allreduce
- MPI_Gather, MPI_Gatherv, MPI_Allgather, MPI_Allgatherv
- MPI_Scatter, MPI_Scatterv
- MPI_Alltoall, MPI_Alltoallv

MPI_Bcast (C)

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

For example, what if the server process needs to send a value that it input from standard input to the other processes?

```
mpi_error_code =
    MPI_Bcast(&length, 1, MPI_INTEGER,
        source, MPI_COMM_WORLD);
```

Notice:

- `MPI_Bcast` doesn't use a tag.
- The call is the same for both the sender and all of the receivers (**COUNTERINTUITIVE!**).

All processes have to call `MPI_Bcast` at the same time; everyone waits until everyone is done.

MPI_Bcast (F90)

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

For example, what if the server process needs to send a value that it input from standard input to the other processes?

```
CALL MPI_Bcast(length, 1, MPI_INTEGER, &
&              source, MPI_COMM_WORLD,  &
&              mpi_error_code)
```

Notice:

- `MPI_Bcast` doesn't use a tag.
- The call is the same for both the sender and all of the receivers (**COUNTERINTUITIVE!**).

All processes have to call `MPI_Bcast` at the same time; everyone waits until everyone is done.

Broadcast Example Part 1 (C)

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

int main (int argc, char** argv)
{ /* main */
  const int server = 0;
  const int source = server;
  float* array   = (float*)NULL;
  int    length, index;
  int    number_of_processes, my_rank, mpi_error_code;

  mpi_error_code = MPI_Init(&argc, &argv);
  mpi_error_code = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error_code = MPI_Comm_size(MPI_COMM_WORLD,
                                 &number_of_processes);
```

Broadcast Example Part 2 (C)

```
  if (my_rank == source) {
    scanf("%d", &length);
  } /* if (my_rank == source) */
  fprintf(stderr, "%d: before MPI_Bcast, length = %d\n",
    my_rank, length);
  mpi_error_code =
    MPI_Bcast(&length, 1, MPI_INTEGER, source, MPI_COMM_WORLD);
  fprintf(stderr, "%d: after  MPI_Bcast, length = %d\n",
    my_rank, length);
  array = (float*)malloc(sizeof(float) * length);
  if (my_rank == source) {
    for (index = 0; index < length; index++) {
      array[index] = sqrt(index * 1.0); /* Or whatever you want */
    } /* for index */
  } /* if (my_rank == source) */
  mpi_error_code =
    MPI_Bcast(array, length, MPI_FLOAT, source, MPI_COMM_WORLD);
  mpi_error_code = MPI_Finalize();
} /* main */
```

Broadcast Example Part 1 (F90)

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

  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, number_of_processes,   &
&                    mpi_error_code);
```

Broadcast Example Part 2 (F90)

```
 IF (my_rank == source) THEN
   READ *, length
 END IF !! (my_rank == source)
 WRITE (0,*) my_rank, ": before MPI_Bcast, length = ", length
 CALL MPI_Bcast(length, 1, MPI_INTEGER, source, MPI_COMM_WORLD, &
&               mpi_error_code)
 WRITE (0,*) my_rank, ": after  MPI_Bcast, length = ", length
 ALLOCATE(array(length), STAT=memory_status)
 IF (memory_status /= memory_success) THEN
   WRITE (0,*) "ERROR: cannot allocate array of length ", length
   CALL MPI_Abort(MPI_COMM_WORLD, mpi_error_code, mpi_error_code)
 END IF (memory_status /= memory_success)
 IF (my_rank == source) THEN
   DO index = 1, length
     array(index) = SQRT(index * 1.0); /* Or whatever you want */
   END DO !! index
 END IF !! (my_rank == source)
 CALL MPI_Bcast(array, length, MPI_FLOAT, source,  &
&               MPI_COMM_WORLD, mpi_error_code)
 CALL MPI_Finalize(mpi_error_code)
END PROGRAM broadcast
```

Broadcast Compile & Run

```
% mpicc -o mpibroadcast mpibroadcast.c -lm
% mpirun -np 8 mpibroadcast
4: before MPI_Bcast, length = 0
7: before MPI_Bcast, length = 0
3: before MPI_Bcast, length = 0
5: before MPI_Bcast, length = 0
6: before MPI_Bcast, length = 0
2: before MPI_Bcast, length = 0
0: before MPI_Bcast, length = 1000000
0: after  MPI_Bcast, length = 1000000
2: after  MPI_Bcast, length = 1000000
4: after  MPI_Bcast, length = 1000000
5: after  MPI_Bcast, length = 1000000
7: after  MPI_Bcast, length = 1000000
6: after  MPI_Bcast, length = 1000000
3: after  MPI_Bcast, length = 1000000
1: before MPI_Bcast, length = 0
1: after  MPI_Bcast, length = 1000000
```

Reductions

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 Part 1 (C)

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

int main (int argc, char** argv)
{ /* main */
  const int server      = 0;
  const int destination = server;
  float value, value_sum;
  int   number_of_processes, my_rank, mpi_error_code;

  mpi_error_code = MPI_Init(&argc, &argv);
  mpi_error_code = MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error_code = MPI_Comm_size(MPI_COMM_WORLD,
                                 &number_of_processes);
```

Reduction Example Part 2 (C)

```
  value = 1.5 * my_rank * number_of_processes;
  fprintf(stderr, "%d: reduce    value     = %f\n",
    my_rank, value);

  mpi_error_code =
    MPI_Reduce   (&value, &value_sum, 1, MPI_FLOAT, MPI_SUM,
                  destination, MPI_COMM_WORLD);
  fprintf(stderr, "%d: reduce    value_sum = %f\n",
    my_rank, value_sum);

  mpi_error_code =
    MPI_Allreduce(&value, &value_sum, 1, MPI_FLOAT, MPI_SUM,
                                  MPI_COMM_WORLD);
  fprintf(stderr, "%d: allreduce value_sum = %f\n",
    my_rank, value_sum);

  mpi_error_code = MPI_Finalize();
} /* main */
```

Reduction Example Part 1 (F90)

```
PROGRAM reduction
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER,PARAMETER :: server      = 0
  INTEGER,PARAMETER :: destination = server
  REAL    :: value, value_sum
  INTEGER :: number_of_processes, 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, number_of_processes,  &
 &                   mpi_error_code)
```

Reduction Example Part 2 (F90)

```
  value = 1.5 * my_rank * number_of_processes
  WRITE (0,*) my_rank, ": reduce    value     = ", value

  CALL MPI_Reduce    (value, value_sum, 1,           &
 &                    MPI_FLOAT, MPI_SUM,           &
 &                    destination, MPI_COMM_WORLD,  &
 &                    mpi_error_code
  WRITE (0,*) my_rank, ": reduce    value_sum = ", value_sum

  CALL MPI_Allreduce(value, value_sum, 1,           &
 &                    MPI_FLOAT, 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 reduction
```

Reduce: Compiling and Running

```
% mpicc -o mpireduce mpireduce.c
% mpirun -np 8 mpireduce
0: reduce    value      = 0.0
4: reduce    value      = 48.0
6: reduce    value      = 72.0
7: reduce    value      = 84.0
3: reduce    value      = 36.0
2: reduce    value      = 24.0
5: reduce    value      = 60.0
1: reduce    value      = 12.0
```

```
7: reduce    value_sum = -9120.0
3: reduce    value_sum = -9120.0
2: reduce    value_sum = -9120.0
5: reduce    value_sum = -9120.0
0: reduce    value_sum = 336.0
1: reduce    value_sum = -9120.0
6: reduce    value_sum = -9120.0
4: reduce    value_sum = -9120.0
2: allreduce value_sum = 336.0
7: allreduce value_sum = 336.0
4: allreduce value_sum = 336.0
3: allreduce value_sum = 336.0
1: allreduce value_sum = 336.0
5: allreduce value_sum = 336.0
0: allreduce value_sum = 336.0
6: allreduce value_sum = 336.0
```

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.

You can think of `MPI_Allreduce` as `MPI_Reduce` followed by `MPI_Bcast` (though it doesn't have to be implemented that way).

Reduction on Arrays Part 1 (C)

MPI_Reduce and **MPI_Allreduce** are actually designed to work on arrays, where the corresponding elements of each source array are reduced into the corresponding element of the destination array (all of the same length):

General

```
MPI_Allreduce(source_array,
              destination_array,
              number_of_array_elements,
              MPI_DATATYPE, MPI_OPERATION,
              MPI_COMMUNICATOR);
```

Example

```
MPI_Allreduce(local_force_on_particle,
              global_force_on_particle,
              number_of_particles,
              MPI_FLOAT, MPI_SUM,
              MPI_COMM_WORLD);
```

Reduction on Arrays Part 2 (C)

```
MPI_Allreduce(local_force_on_particle,
              global_force_on_particle,
              number_of_particles,
              MPI_FLOAT, MPI_SUM,
              MPI_COMM_WORLD);
```

```
global_force_on_particle[p] =
  local_force_on_particle[p] on Rank 0 +
  local_force_on_particle[p] on Rank 1 +
  local_force_on_particle[p] on Rank 2 +
  ...
  local_force_on_particle[p] on Rank np-1;
```

Scatter and Gather

- To ***scatter*** is to send data from one place to many places.
- To ***gather*** is to receive data from many places into one place.
- MPI has a variety of scatter and gather routines:
  - `MPI_Scatter`, `MPI_Scatterv`
  - `MPI_Gather`, `MPI_Gatherv`, `MPI_Allgather`, `MPI_Allgatherv`
- The scatter routines split up a single larger array into smaller subarrays, one per MPI process, and send each subarray to an MPI process.
- The gather routines receive many smaller subarrays, one per MPI process, and assemble them into a single larger array.

MPI_Scatter

MPI_Scatter takes an array whose length is divisible by the number of MPI processes, and splits it up into subarrays of equal length, then sends one subarray to each MPI process.

```
MPI_Scatter(large_array,    small_array_length,
            MPI_DATATYPE,
            small_subarray, small_subarray_length,
            MPI_DATATYPE, source, MPI_COMMUNICATOR);
```

So, for a large array of length 100 on 5 MPI processes:

- each smaller subarray has length 20;
- large_array[0] .. large_array[19] go to small_array on Rank 0;
- large_array[20]..large_array[39] go to small_array on Rank 1;
- etc

MPI_Scatterv

MPI_Scatterv is just like MPI_Scatter, except that the subarray lengths don't have to be the same (and therefore the length of the large array doesn't have to be divisible by the number of MPI processes).

```
MPI_Scatterv(large_array,    small_array_length,
             displacements,
             MPI_DATATYPE,
             small_subarray, small_subarray_lengths,
             MPI_DATATYPE, source, MPI_COMMUNICATOR);
```

The **displacements** array says where each small subarray begins within the large array.

MPI_Gather

MPI_Gather receives a small array on each of the MPI processes, all subarrays of equal length, and joins them into a single large array on the destination MPI process.

```
MPI_Gather(small_subarray, small_subarray_length,
           MPI_DATATYPE,
           large_array,   large_array_length,
           MPI_DATATYPE, destination, MPI_COMMUNICATOR);
```

So, for a small subarray of length 20 on each of 5 MPI processes:

- the large array on the destination process has length 100;
- large_array[0] .. large_array[19] come from small_array on Rank 0;
- large_array[20]..large_array[39] come from small_array on Rank 1;
- etc

MPI_Gatherv

MPI_Gatherv is just like **MPI_Gather**, except that the subarray lengths don't have to be the same (and therefore the length of the large array doesn't have to be divisible by the number of MPI processes).

```
MPI_Gatherv(small_subarray, small_subarray_length,
            MPI_DATATYPE,
            large_array,    small_subarray_lengths,
            displacements,
            MPI_DATATYPE, destination, MPI_COMMUNICATOR);
```

The **displacements** array says where each small subarray begins within the large array.

MPI_Allgather & MPI_Allgatherv

MPI_Allgather and **MPI_Allgatherv** are the same as **MPI_Gather** and **MPI_Gatherv**, except that the large array gets filled on every MPI process, so no destination process argument is needed.

Thanks for your attention!

Questions?

Thanks for your attention!

Questions?