# Introduction to Parallel Programming & Cluster Computing

## MPI Collective Communications

Joshua Alexander, U Oklahoma
Ivan Babic, Earlham College
Michial Green, Contra Costa College
Mobeen Ludin, Earlham College
Tom Murphy, Contra Costa College
Kristin Muterspaw, Earlham College
Henry Neeman, U Oklahoma
Charlie Peck, Earlham College

# 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

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_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.

# Broadcast Example Part 1

```
#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

```
  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 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 may take longer)

# Reduction Example Part 1

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

int main (int argc, char** argv)
{ /* main */
  const int server      = 0;
  const int destination = server;
  int 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 1

```
  value = my_rank * number_of_processes;
  fprintf(stderr, "%d: reduce    value     = %d\n",
    my_rank, value);
  mpi_error_code =
    MPI_Reduce   (&value, &value_sum, 1, MPI_INT, MPI_SUM,
                  destination, MPI_COMM_WORLD);
  fprintf(stderr, "%d: reduce    value_sum = %d\n",
    my_rank, value_sum);
  mpi_error_code =
    MPI_Allreduce(&value, &value_sum, 1, MPI_INT, MPI_SUM,
                               MPI_COMM_WORLD);
  fprintf(stderr, "%d: allreduce value_sum = %d\n",
    my_rank, value_sum);
  mpi_error_code = MPI_Finalize();
} /* main */
```

# Reduce: Compiling and Running

```
% mpicc -o mpireduce mpireduce.c
% mpirun -np 8 mpireduce
0: reduce    value     = 0
4: reduce    value     = 32
6: reduce    value     = 48
7: reduce    value     = 56
3: reduce    value     = 24
2: reduce    value     = 16
5: reduce    value     = 40
1: reduce    value     = 8
7: reduce    value_sum = -9120
3: reduce    value_sum = -9120
2: reduce    value_sum = -9120
5: reduce    value_sum = -9120
1: reduce    value_sum = -9120
6: reduce    value_sum = -9120
4: reduce    value_sum = -9120
0: reduce    value_sum = 224
```

```
2: allreduce value_sum = 224
7: allreduce value_sum = 224
4: allreduce value_sum = 224
3: allreduce value_sum = 224
1: allreduce value_sum = 224
5: allreduce value_sum = 224
0: allreduce value_sum = 224
6: allreduce value_sum = 224
```

## 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 #1

`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):

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

For example:

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

## Reduction on Arrays #2

```
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_subarray_lengths,
             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.

# OK Supercomputing Symposium 2012

2003 Keynote:
Peter Freeman
NSF
Computer & Information Science & Engineering
Assistant Director

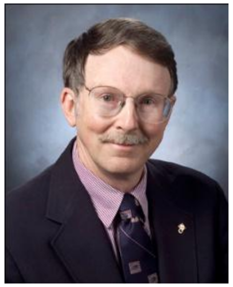
2004 Keynote:
Sangtae Kim
NSF Shared Cyberinfrastructure
Division Director

2005 Keynote:
Walt Brooks
NASA Advanced Supercomputing
Division Director

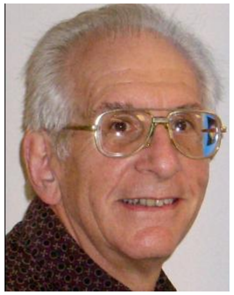
2006 Keynote:
Dan Atkins
Head of NSF's
Office of Cyberinfrastructure

2007 Keynote:
Jay Boisseau
Director
Texas Advanced Computing Center
U. Texas Austin

2008 Keynote:
José Munoz
Deputy Office Director/ Senior Scientific Advisor
NSF Office of Cyberinfrastructure


2009 Keynote:
Douglass Post
Chief Scientist
US Dept of Defense
HPC Modernization Program

2010 Keynote:
Horst Simon
Deputy Director
Lawrence Berkeley National Laboratory


2011 Keynote:
Barry Schneider
Program Manager
National Science Foundation

**Thom Dunning, Director**
**National Center for Supercomputing Applications**

**FREE! Wed Oct 3 2012 @ OU**

http://symposium2012.oscer.ou.edu/

**Reception/Poster Session**
**FREE! Tue Oct 2 2012 @ OU**
**FREE! Symposium Wed Oct 3 2012 @ OU**

# Thanks for your attention!

## Questions?

www.oscer.ou.edu