

Supercomputing in Plain English

Distributed Multiprocessing

Henry Neeman, University of Oklahoma

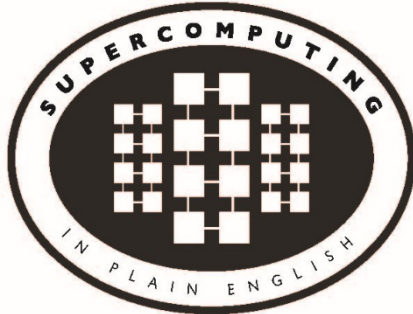
Director, OU Supercomputing Center for Education & Research (OSCER)

Assistant Vice President, Information Technology – Research Strategy Advisor

Associate Professor, Gallogly College of Engineering

Adjunct Associate Professor, School of Computer Science

Tuesday March 6 2018





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 phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





PLEASE MUTE YOURSELF

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

At OU, 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 e-mail:

supercomputinginplainenglish@gmail.com

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.



Download the Slides Beforehand

Before the start of the session, please download the slides from the Supercomputing in Plain English website:

<http://www.oscer.ou.edu/education/>

That way, if anything goes wrong, you can still follow along with just audio.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Zoom

Go to:

<http://zoom.us/j/979158478>

Many thanks Eddie Huebsch, OU CIO, for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.



INFORMATION TECHNOLOGY
UNIVERSITY OF OKLAHOMA

Supercomputing in Plain English: Distributed Par
Tue March 6 2018





YouTube

You can watch from a Windows, MacOS or Linux laptop or an Android or iOS handheld using YouTube.

Go to YouTube via your preferred web browser or app, and then search for:

Supercomputing InPlainEnglish

(**InPlainEnglish** is all one word.)

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Twitch

You can watch from a Windows, MacOS or Linux laptop or an Android or iOS handheld using Twitch.

Go to:

<http://www.twitch.tv/sipe2018>

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Wowza #1

You can watch from a Windows, MacOS or Linux laptop using Wowza from the following URL:

<http://jwplayer.onenet.net/streams/sipe.html>

If that URL fails, then go to:

<http://jwplayer.onenet.net/streams/sipebackup.html>

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Wowza #2

Wowza has been tested on multiple browsers on each of:

- Windows 10: IE, Firefox, Chrome, Opera, Safari
- MacOS: Safari, Firefox
- Linux: Firefox, Opera

We've also successfully tested it via apps on devices with:

- Android
- iOS

Many thanks to Skyler Donahue of OneNet for providing this.

PLEASE MUTE YOURSELF.
PLEASE MUTE YOURSELF.
PLEASE MUTE YOURSELF.





Toll Free Phone Bridge

IF ALL ELSE FAILS, you can use our US TOLL phone bridge:

405-325-6688

684 684 #

NOTE: This is for **US** call-ins **ONLY**.

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 can handle only 100 simultaneous connections, and we have over 1000 participants.

Many thanks to OU CIO Eddie Huebsch for providing the phone bridge..





Please Mute Yourself

No matter how you connect, **PLEASE MUTE YOURSELF**, so that we cannot hear you.

(For YouTube, Twitch and Wowza, you don't need to do that, because the information only goes from us to you, not from you to us.)

At OU, 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 e-mail.

PLEASE MUTE YOURSELF.





Questions via E-mail Only

Ask questions by sending e-mail to:

supercomputinginplainenglish@gmail.com

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

DON'T USE CHAT OR VOICE FOR QUESTIONS!

No one will be monitoring any of the chats, and if we can hear your question, you're creating an **echo cancellation** problem.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Onsite: Talent Release Form

If you're attending onsite, you **MUST** do one of the following:

- complete and sign the Talent Release Form,

OR

- sit behind the cameras (where you can't be seen) and don't talk at all.

If you aren't onsite, then **PLEASE MUTE YOURSELF.**



TENTATIVE Schedule

- Tue Jan 23: Storage: What the Heck is Supercomputing?
- Tue Jan 30: The Tyranny of the Storage Hierarchy Part I
- Tue Feb 6: The Tyranny of the Storage Hierarchy Part II
- Tue Feb 13: Instruction Level Parallelism
- Tue Feb 20: Stupid Compiler Tricks
- Tue Feb 27: Distributed Par Multithreading
- Tue March 6: Distributed Multiprocessing
- Tue March 13: **NO SESSION** (Henry business travel)
- Tue March 20: **NO SESSION** (OU's Spring Break)
- Tue March 27: Applications and Types of Parallelism
- Tue Apr 3: Multicore Madness
- Tue Apr 10: High Throughput Computing
- Tue Apr 17: **NO SESSION** (Henry business travel)
- Tue Apr 24: GPGPU: Number Crunching in Your Graphics Card
- Tue May 1: Grab Bag: Scientific Libraries, I/O Libraries, Visualization





Thanks for helping!

- OU IT
 - OSCER operations staff (Dave Akin, Patrick Calhoun, Kali McLennan, Jason Speckman, Brett Zimmerman)
 - OSCER Research Computing Facilitators (Jim Ferguson, Horst Severini)
 - Debi Gentis, OSCER Coordinator
 - Kyle Dudgeon, OSCER Manager of Operations
 - Ashish Pai, Managing Director for Research IT Services
 - The OU IT network team
 - OU CIO Eddie Huebsch
- OneNet: Skyler Donahue
- Oklahoma State U: Dana Brunson





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 phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Coming in 2018!

- Coalition for Advancing Digital Research & Education (CADRE) Conference:
Apr 17-18 2018 @ Oklahoma State U, Stillwater OK USA
<https://hpcc.okstate.edu/cadre-conference>
- Linux Clusters Institute workshops
<http://www.linuxclustersinstitute.org/workshops/>
 - Introductory HPC Cluster System Administration: May 14-18 2018 @ U Nebraska, Lincoln NE USA
 - Intermediate HPC Cluster System Administration: Aug 13-17 2018 @ Yale U, New Haven CT USA
- Great Plains Network Annual Meeting: details coming soon
- Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency Aug 5-10 2018, U Oklahoma, Norman OK USA
- PEARC 2018, July 22-27, Pittsburgh PA USA
<https://www.pearc18.pearc.org/>
- IEEE Cluster 2018, Sep 10-13, Belfast UK
<https://cluster2018.github.io>
- **OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2018, Sep 25-26 2018 @ OU**
- SC18 supercomputing conference, Nov 11-16 2018, Dallas TX USA
<http://sc18.supercomputing.org/>





Outline

- The Desert Islands Analogy
- Distributed Parallelism
- MPI



The Desert Islands Analogy





An Island Hut

- Imagine you're on an island in a little hut.
- Inside the hut is a desk.
- On the desk is:

- a phone;
- a pencil;
- a calculator;
- a piece of paper with instructions;
- a piece of paper with numbers (data).



Instructions: What to Do

```

...
Add the number in slot 27 to the number in slot 239,
  and put the result in slot 71.
if the number in slot 71 is equal to the number in slot 118 then
  Call 555-0127 and leave a voicemail containing the number in slot 962.
else
  Call your voicemail box and collect a voicemail from 555-0063,
  and put that number in slot 715.
...

```

DATA

1. 27.3
2. -491.41
3. 24
4. -1e-05
5. 141.41
6. 0
7. 4167
8. 94.14
9. -518.481
- ...





Instructions

The instructions are split into two kinds:

- Arithmetic/Logical – for example:
 - Add the number in slot 27 to the number in slot 239, and put the result in slot 71.
 - Compare the number in slot 71 to the number in slot 118, to see whether they are equal.
- Communication – for example:
 - Call 555-0127 and leave a voicemail containing the number in slot 962.
 - Call your voicemail box and collect a voicemail from 555-0063, and put that number in slot 715.





Is There Anybody Out There?

If you're in a hut on an island, you aren't specifically aware of anyone else.

Especially, you don't know whether anyone else is working on the same problem as you are, and you don't know who's at the other end of the phone line.

All you know is what to do with the voicemails you get, and what phone numbers to send voicemails to.



Someone Might Be Out There

Now suppose that Horst is on another island somewhere, in the same kind of hut, with the same kind of equipment.

Suppose that he has the same list of instructions as you, but a different set of numbers (both data and phone numbers).

Like you, he doesn't know whether there's anyone else working on his problem.



Even More People Out There

Now suppose that Bruce and Dee are also in huts on islands.

Suppose that each of the four has the exact same list of instructions, but different lists of numbers.

And suppose that the phone numbers that people call are each others': that is, your instructions have you call Horst, Bruce and Dee, Horst's has him call Bruce, Dee and you, and so on.

Then you might all be working together on the same problem.



All Data Are Private

Notice that you can't see Horst's or Bruce's or Dee's numbers, nor can they see yours or each other's.

Thus, everyone's numbers are private: there's no way for anyone to share numbers, except by leaving them in voicemails.





Long Distance Calls: 2 Costs

When you make a long distance phone call, you typically have to pay two costs:

- **Connection charge**: the **fixed** cost of connecting your phone to someone else's, even if you're only connected for a second
- **Per-minute charge**: the cost per minute of talking, once you're connected

If the connection charge is large, then you want to make as few calls as possible.

See:

<http://www.youtube.com/watch?v=8k1UOEYIQRo>

Distributed Parallelism





Like Desert Islands

Distributed parallelism is very much like the Desert Islands analogy:

- processes are independent of each other.
- All data are private.
- Processes communicate by passing messages (like voicemails).
- The cost of passing a message is split into:
 - latency (connection time)
 - bandwidth (time per byte)





Latency vs Bandwidth on Schooner

In 2018, a benchmark of the Infiniband interconnect on the University of Oklahoma's Linux cluster revealed:

- **Latency** – the time for the first bit to show up at the destination – is ~1.26 microseconds;
- **Bandwidth** – the speed of the subsequent bits – is ~37.2 Gigabits per second (~0.027 nanosec per bit).

Thus, on OU's cluster Infiniband:

- the first bit of a message shows up in ~1260 nanosec;
- the last bit of a message shows up in ~0.027 nanosec.

So latency is ~**47,000 times worse** than bandwidth!





Latency vs Bandwidth on Schooner

In 2018, a benchmark of the Infiniband interconnect on the University of Oklahoma's Linux cluster revealed:

- **Latency** – the time for the first bit to show up at the destination – is ~1.26 microseconds;
- **Bandwidth** – the speed of the subsequent bits – is ~37.2 Gigabits per second (~0.027 nanosec per bit).

Thus, on OU's cluster Infiniband:

- the first bit of a message shows up in ~1260 nanosec;
- the last bit of a message shows up in ~0.027 nanosec.

So latency is ~**47,000 times worse** than bandwidth!

That's like having a long distance service that charges:

- \$470 to make a call at all, regardless of duration;
- 1¢ per minute – after the **first 33 days** on the call.



MPI: The Message-Passing Interface



Most of this discussion is from [1] and [2].



What Is MPI?

The *Message-Passing Interface* (MPI) is a standard for expressing distributed parallelism via message passing.

MPI consists of a *header file*, a *library of routines* and a *runtime environment*.

When you compile a program that has MPI calls in it, your compiler links to a local implementation of MPI, and then you get parallelism; if the MPI library isn't available, then the compile will fail.

MPI can be used in Fortran, C and C++.

There are also **unofficial** bindings for MATLAB, Python, R and a few others, but these aren't part of the official MPI standard.





MPI Calls

In C, MPI calls look like:

```
mpi_error_code = MPI_Funcname (...);
```

In Fortran, MPI calls look like this:

```
CALL MPI_Funcname (... , mpi_error_code)
```

Notice that `mpi_error_code` is returned by the MPI routine `MPI_Funcname`, with a value of `MPI_SUCCESS` indicating that `MPI_Funcname` has worked correctly.

In C++, MPI calls look like:

```
mpi_error_code = MPI::Funcname (...);
```

But, the C++ binding has been deprecated, so **DON'T USE IT.** Instead, use the C binding, above.



MPI is an API

MPI is actually just an *Application Programming Interface* (API).

An API specifies what a call to each routine should look like, and how each routine should behave.

An API does not specify how each routine should be implemented, and sometimes is intentionally vague about certain aspects of a routine's behavior.

Each platform can have its own MPI implementation – or multiple MPI implementations.





Example MPI Implementations

- MPICH2 (<http://www.mpich.org>)
- OpenMPI (<https://www.open-mpi.org>)
- Intel MPI (<https://software.intel.com/en-us/intel-mpi-library>)
- Microsoft MPI ([https://msdn.microsoft.com/en-us/library/bb524831\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/bb524831(v=vs.85).aspx))
- IBM Platform MPI
(https://www.ibm.com/support/knowledgecenter/en/SSF4ZA_9.1.3/pmpi_welcome/pmpi_9.1.3.html)
- IBM Parallel Operating Environment
(https://www.ibm.com/support/knowledgecenter/SSFK3V_2.3.0/com.ibm.cluster.pe.v2r3.pe400.doc/am106_mpibeo.htm)
- Cray Message Passing Toolkit (<https://pubs.cray.com/content/S-2529/17.05/xctm-series-programming-environment-user-guide-1705-s-2529/mpt>)



WARNING!

In principle, the MPI standard provides *bindings* for:

- C
- C++ (deprecated)
- Fortran 77
- Fortran 90

In practice, you should do this:

- To use MPI in a C++ code, use the C binding.
- To use MPI in Fortran 90, use the Fortran 77 binding.

This is because the C++ and Fortran 90 bindings are less popular, and therefore less well tested.



The 6 Most Important MPI Routines

- **MPI_Init** starts up the MPI runtime environment at the beginning of a run.
- **MPI_Finalize** shuts down the MPI runtime environment at the end of a run.
- **MPI_Comm_size** gets the number of processes in a run, N_p (typically called just after **MPI_Init**).
- **MPI_Comm_rank** gets the process ID that the current process uses, which is between 0 and N_p-1 inclusive (typically called just after **MPI_Init**).
- **MPI_Send** sends a message from the current process to some other process (the *destination*).
- **MPI_Recv** receives a message on the current process from some other process (the *source*).





More Example MPI Routines

- **MPI_Bcast** *broadcasts* a message from one process to all of the others.
- **MPI_Reduce** performs a *reduction* (for example, sum, maximum) of a variable on all processes, sending the result to a single process.

NOTE: Here, *reduce* means turn many values into fewer values.

- **MPI_Gather** *gathers* a set of subarrays, one subarray from each process, into a single large array on a single process.
- **MPI_Scatter** *scatters* a single large array on a single process into subarrays, one subarray sent to each process.

Routines that use all processes at once are known as *collective*; routines that involve only a few are known as *point-to-point*.



MPI Program Structure (C)

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
[other includes]

int main (int argc, char* argv[])
{ /* main */
  int my_rank, num_procs, mpi_error_code;
  [other declarations]
  mpi_error_code =
    MPI_Init(&argc, &argv);          /* Start up MPI */
  mpi_error_code =
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  mpi_error_code =
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  [actual work goes here]
  mpi_error_code = MPI_Finalize(); /* Shut down MPI */
} /* main */
```



MPI is SPMD

MPI uses kind of parallelism known as
Single Program, Multiple Data (SPMD).

This means that you have one MPI program –
a single executable – that is executed by all of the processes
in an MPI run.

So, to differentiate the roles of various processes in the MPI
run, you have to have **if** statements:

```
if (my_rank == server_rank) {  
    ...  
}
```





Example: Hello World

1. Start the MPI system.
2. Get this process's rank, and the number of processes.
3. Output "Hello world" along with the rank and number of processes.
4. Shut down the MPI system.





Example: Hello World Code (C)

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

int main (int argc, char** argv)
{ /* main */
    int number_of_processes;
    int my_rank;
    int 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);
    printf("%d of %d: Hello, world!\n", my_rank, number_of_processes);
    mpi_error_code = MPI_Finalize();
} /* main */
```



Example: Hello World Code (F90)

```
PROGRAM hello_world_mpi
  IMPLICIT NONE
  INCLUDE "mpif.h"
  INTEGER :: number_of_processes, my_rank
  INTEGER :: mpi_error_code

  CALL MPI_Init(mpi_error_code)
  CALL MPI_Comm_rank(MPI_COMM_WORLD, number_of_processes, &
&      mpi_error_code)
  CALL MPI_Comm_size(MPI_COMM_WORLD, my_rank, &
&      mpi_error_code)
  PRINT *, my_rank, " of ", number_of_processes, &
&      ": Hello, world!"
  CALL MPI_Finalize(mpi_error_code)
END PROGRAM hello_world_mpi
```



Example: Hello World Output

```
2 of 20: Hello, world!  
4 of 20: Hello, world!  
8 of 20: Hello, world!  
10 of 20: Hello, world!  
14 of 20: Hello, world!  
15 of 20: Hello, world!  
16 of 20: Hello, world!  
17 of 20: Hello, world!  
18 of 20: Hello, world!  
0 of 20: Hello, world!
```

```
1 of 20: Hello, world!  
3 of 20: Hello, world!  
5 of 20: Hello, world!  
6 of 20: Hello, world!  
7 of 20: Hello, world!  
9 of 20: Hello, world!  
11 of 20: Hello, world!  
12 of 20: Hello, world!  
13 of 20: Hello, world!  
19 of 20: Hello, world!
```



Example: Greetings

1. Start the MPI system.
2. Get this process's rank, and the number of processes.
3. If I'm **not** the server process:
 1. Create a greeting string.
 2. Send it to the server process.
4. If I **am** the server process:
 1. For each of the client processes:
 1. Receive its greeting string.
 2. Print its greeting string.
5. Shut down the MPI system.

See [1].





greeting.c

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

int main (int argc, char* argv[])
{ /* main */
    const int    maximum_message_length = 100;
    const int    server_rank            =    0;
    char         message[maximum_message_length+1];
    MPI_Status   status;                /* Info about receive status */
    int          my_rank;                /* This process ID */
    int          num_procs;              /* Number of processes in run */
    int          source;                 /* Process ID to receive from */
    int          destination;            /* Process ID to send to */
    int          tag = 0;                /* Message ID */
    int          mpi_error_code;        /* Error code for MPI calls */

    [work goes here]

} /* main */
```





Greetings Startup/Shutdown

[header file includes]

```
int main (int argc, char* argv[])
{ /* main */
```

[declarations]

```
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, &num_procs);
if (my_rank != server_rank) {
```

[work of each non-server (worker) process]

```
} /* if (my_rank != server_rank) */
else {
```

[work of server process]

```
} /* if (my_rank != server_rank)...else */
mpi_error_code = MPI_Finalize();
} /* main */
```



Greetings Client's Work

[header file includes]

```
int main (int argc, char* argv[])  
{ /* main */
```

[declarations]

[MPI startup (MPI_Init etc)]

```
    if (my_rank != server_rank) {  
        sprintf(message, "Greetings from process %d!",  
                my_rank);  
        destination = server_rank;  
        mpi_error_code =  
            MPI_Send(message, strlen(message) + 1, MPI_CHAR,  
                    destination, tag, MPI_COMM_WORLD);  
    } /* if (my_rank != server_rank) */  
    else {
```

[work of server process]

```
    } /* if (my_rank != server_rank)...else */  
    mpi_error_code = MPI_Finalize();  
} /* main */
```





Greetings Server's Work

[header file includes]

```
int main (int argc, char* argv[])
{ /* main */
```

[declarations, MPI startup]

```
if (my_rank != server_rank) {
```

[work of each client process]

```
} /* if (my_rank != server_rank) */
```

```
else {
```

```
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 */
```

```
} /* if (my_rank != server_rank)...else */
```

```
mpi_error_code = MPI_Finalize();
```

```
} /* main */
```





How an MPI Run Works

- Every process gets a copy of the executable:
Single Program, Multiple Data (SPMD).
- They all start executing it.
- Each looks at its own rank to determine which part of the problem to work on.
- Each process works **completely independently** of the other processes, except when communicating.





Compiling and Running

```
% mpicc -o greeting_mpi greeting.c
% mpirun -np 1 greeting_mpi
% mpirun -np 2 greeting_mpi
Greetings from process #1!
% mpirun -np 3 greeting_mpi
Greetings from process #1!
Greetings from process #2!
% mpirun -np 4 greeting_mpi
Greetings from process #1!
Greetings from process #2!
Greetings from process #3!
```

Note: The compile command and the run command vary from platform to platform.

This **ISN'T** how you run MPI on Schooner.





Why is Rank #0 the Server?

```
const int server_rank = 0;
```

By convention, if an MPI program uses a client-server approach, then the server process has rank (process ID) #0. Why?

A run must use at least one process but can use multiple processes. Process ranks are 0 through $N_p - 1$, for $N_p \geq 1$, where N_p is the number of processes in the run.

Therefore, every MPI run has a process with rank #0.

Note: Every MPI run also has a process with rank $N_p - 1$, so you could use $N_p - 1$ as the server instead of 0 ... but no one does.



Does There Have to be a Server?

There **DOESN'T** have to be a server.

It's perfectly possible to write an MPI code that has no server as such.

For example, weather forecasting and other transport codes typically share most duties equally, and likewise chemistry and astronomy codes.

In practice, though, most codes use rank #0 to do things like small scale I/O, since it's typically more efficient to have one process read small files and then broadcast small input data to the other processes, or to gather the output data and write it to disk.



Why “Rank?”

Why does MPI use the term *rank* to refer to process ID?

In general, a process has an identifier that is assigned by the operating system (for example, Unix), and that is unrelated to MPI:

% **ps**

PID	TTY	TIME	CMD
52170812	ttyq57	0:01	tcsh

Also, each processor has an identifier, but an MPI run that uses fewer than all processors will use an arbitrary subset.

The rank of an MPI process is neither of these.





Compiling and Running

Recall:

```
% mpicc -o greeting_mpi greeting.c
```

```
% mpirun -np 1 greeting_mpi
```

```
% mpirun -np 2 greeting_mpi
```

```
Greetings from process #1!
```

```
% mpirun -np 3 greeting_mpi
```

```
Greetings from process #1!
```

```
Greetings from process #2!
```

```
% mpirun -np 4 greeting_mpi
```

```
Greetings from process #1!
```

```
Greetings from process #2!
```

```
Greetings from process #3!
```





Deterministic Operation?

```
% mpirun -np 4 greeting_mpi  
Greetings from process #1!  
Greetings from process #2!  
Greetings from process #3!
```

The order in which the greetings are output is deterministic.

Why?

```
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 */
```

This loop ignores the order in which messages are received.



Deterministic Parallelism

```
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 */
```

Because of the order in which the loop iterations occur, the greeting messages will be **output** in **deterministic** order, regardless of the order in which the greeting messages are received.

In principle, the run could pause for a long time, waiting for one client process's message to arrive at the server process.





Nondeterministic Parallelism

```
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 */
```

Because of this change, the greeting messages will be output in non-deterministic order, specifically in the order in which they're received.



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 the message's 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

C		Fortran	
char	<code>MPI_CHAR</code>	CHARACTER	<code>MPI_CHARACTER</code>
int	<code>MPI_INT</code>	INTEGER	<code>MPI_INTEGER</code>
float	<code>MPI_FLOAT</code>	REAL	<code>MPI_REAL</code>
double	<code>MPI_DOUBLE</code>	DOUBLE PRECISION	<code>MPI_DOUBLE_PRECISION</code>

MPI supports several other data types, but most are variations on 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
- Hanukkah

She knows because of the tag on the present:

- A little cake with candles means birthday
- A little tree or a Santa means Christmas
- A little menorah means Hanukkah





Message Tags

```
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 **output** 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

```
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 output 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 in the current run. It's probably the only one you'll need in most cases.

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?

```
mpi_error_code =
```

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

Note that **MPI_Bcast** doesn't use a tag, and that the call is the same for both the sender and all of the receivers. This is **COUNTERINTUITIVE!**

All processes have to call **MPI_Bcast** at the same time; everyone waits until everyone is done (synchronization).



Broadcast Example: Setup

```
#include <stdio.h>
#include <stdlib.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;
  int num_procs, 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, &num_procs);
  [input, allocate, initialize on server only]
  [broadcast, output on all processes]
  mpi_error_code = MPI_Finalize();
} /* main */
```





Broadcast Example: Input

```
#include <stdio.h>
#include <stdlib.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;
  int num_procs, my_rank, mpi_error_code;

  [MPI startup]
  if (my_rank == server) {
    scanf("%d", &length);
    array = (float*)malloc(sizeof(float) * length);
    for (index = 0; index < length; index++) {
      array[index] = 0.0;
    } /* for index */
  } /* if (my_rank == server) */
  [broadcast, output on all processes]
  [MPI shutdown]
} /* main */
```





Broadcast Example: Broadcast

```
#include <stdio.h>
#include <stdlib.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;
  int num_procs, my_rank, mpi_error_code;
```

[MPI startup]

[input, allocate, initialize on server only]

```
  if (num_procs > 1) {
    mpi_error_code =
      MPI_Bcast(&length, 1, MPI_INTEGER, source, MPI_COMM_WORLD);
    if (my_rank != server) {
      array = (float*)malloc(sizeof(float) * length);
    } /* if (my_rank != server) */
    mpi_error_code =
      MPI_Bcast(array, length, MPI_INTEGER, source,
        MPI_COMM_WORLD);
    printf("%d: broadcast length = %d\n", my_rank, length);
  } /* if (num_procs > 1) */
  mpi_error_code = MPI_Finalize();
} /* main */
```

Supercomputing in Plain English: Distributed Par

Tue March 6 2018



INFORMATION TECHNOLOGY
UNIVERSITY OF OKLAHOMA





Broadcast Compile & Run

```
% mpicc -o broadcast broadcast.c
```

```
% mpirun -np 4 broadcast
```

```
0 : broadcast length = 16777216
```

```
1 : broadcast length = 16777216
```

```
2 : broadcast length = 16777216
```

```
3 : broadcast length = 16777216
```





Reductions

A ***reduction*** converts an array to a scalar (or, more generally, converts many values to fewer values).

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

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

int main (int argc, char **argv)
{ /* main */
  const int server = 0;
  float value, value_sum;
  int num_procs, 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, &num_procs);
  value_sum = 0.0;
  value      = my_rank * num_procs;
  mpi_error_code =
    MPI_Reduce (&value, &value_sum, 1, MPI_FLOAT, MPI_SUM,
               server, MPI_COMM_WORLD);
  printf("%d: reduce      value_sum = %d\n", my_rank, value_sum);
  mpi_error_code =
    MPI_Allreduce(&value, &value_sum, 1, MPI_FLOAT, MPI_SUM,
                 MPI_COMM_WORLD);
  printf("%d: allreduce value_sum = %d\n", my_rank, value_sum);
  mpi_error_code = MPI_Finalize();
} /* main */
```





Compiling and Running

```
% mpicc -o reduce reduce.c
% mpirun -np 4 reduce
3: reduce      value_sum = 0
1: reduce      value_sum = 0
0: reduce      value_sum = 24
2: reduce      value_sum = 0
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

```
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).





TENTATIVE Schedule

- Tue Jan 23: Storage: What the Heck is Supercomputing?
- Tue Jan 30: The Tyranny of the Storage Hierarchy Part I
- Tue Feb 6: The Tyranny of the Storage Hierarchy Part II
- Tue Feb 13: Instruction Level Parallelism
- Tue Feb 20: Stupid Compiler Tricks
- Tue Feb 27: Distributed Par Multithreading
- Tue March 6: Distributed Multiprocessing
- Tue March 13: **NO SESSION** (Henry business travel)
- Tue March 20: **NO SESSION** (OU's Spring Break)
- Tue March 27: Applications and Types of Parallelism
- Tue Apr 3: Multicore Madness
- Tue Apr 10: High Throughput Computing
- Tue Apr 17: **NO SESSION** (Henry business travel)
- Tue Apr 24: GPGPU: Number Crunching in Your Graphics Card
- Tue May 1: Grab Bag: Scientific Libraries, I/O Libraries, Visualization





Thanks for helping!

- OU IT
 - OSCER operations staff (Dave Akin, Patrick Calhoun, Kali McLennan, Jason Speckman, Brett Zimmerman)
 - OSCER Research Computing Facilitators (Jim Ferguson, Horst Severini)
 - Debi Gentis, OSCER Coordinator
 - Kyle Dudgeon, OSCER Manager of Operations
 - Ashish Pai, Managing Director for Research IT Services
 - The OU IT network team
 - OU CIO Eddie Huebsch
- OneNet: Skyler Donahue
- Oklahoma State U: Dana Brunson





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 phone bridge to fall back on.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.

PLEASE MUTE YOURSELF.





Coming in 2018!

- Coalition for Advancing Digital Research & Education (CADRE) Conference:
Apr 17-18 2018 @ Oklahoma State U, Stillwater OK USA
<https://hpcc.okstate.edu/cadre-conference>
- Linux Clusters Institute workshops
<http://www.linuxclustersinstitute.org/workshops/>
 - Introductory HPC Cluster System Administration: May 14-18 2018 @ U Nebraska, Lincoln NE USA
 - Intermediate HPC Cluster System Administration: Aug 13-17 2018 @ Yale U, New Haven CT USA
- Great Plains Network Annual Meeting: details coming soon
- Advanced Cyberinfrastructure Research & Education Facilitators (ACI-REF) Virtual Residency Aug 5-10 2018, U Oklahoma, Norman OK USA
- PEARC 2018, July 22-27, Pittsburgh PA USA
<https://www.pearc18.pearc.org/>
- IEEE Cluster 2018, Sep 10-13, Belfast UK
<https://cluster2018.github.io>
- **OKLAHOMA SUPERCOMPUTING SYMPOSIUM 2018, Sep 25-26 2018 @ OU**
- SC18 supercomputing conference, Nov 11-16 2018, Dallas TX USA
<http://sc18.supercomputing.org/>



**Thanks for your
attention!**



Questions?

www.oscer.ou.edu



References

- [1] P.S. Pacheco, *Parallel Programming with MPI*, Morgan Kaufmann Publishers, 1997.
- [2] W. Gropp, E. Lusk and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd ed. MIT Press, 1999.

