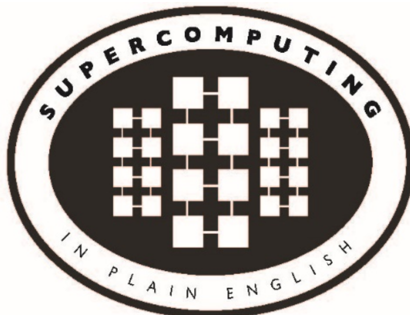
Supercomputing in Plain English

Parallel Programming for Beginners

Henry Neeman, University of Oklahoma
Assistant Vice President, Information Technology - Research Strategy Advisory
Director, OU Supercomputing Center for Education & Research (OSCER)
Associate Professor, College of Engineering
Adjunct Associate Professor, School of Computer Science
RMACC HPC Symposium 2015, Tuesday August 11 2015



Outline

- The Desert Islands Analogy
- Distributed Parallelism
- MPI

Distributed Parallelism: 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 `topdawg`

In 2006, a benchmark of the Infiniband interconnect on a large Linux cluster at the University of Oklahoma revealed:

- **Latency** – the time for the first bit to show up at the destination – is about 3 microseconds;
- **Bandwidth** – the speed of the subsequent bits – is about 5 Gigabits per second.

Thus, on this cluster's Infiniband:

- the 1st bit of a message shows up in 3 microsec;
- the 2nd bit shows up in 0.2 nanosec.

So latency is **15,000 times worse** than bandwidth!

Latency vs Bandwidth on `topdawg`

In 2006, a benchmark of the Infiniband interconnect on a large Linux cluster at the University of Oklahoma revealed:

- **Latency** – the time for the first bit to show up at the destination – is about 3 microseconds;
- **Bandwidth** – the speed of the subsequent bits – is about 5 Gigabits per second.

Latency is **15,000 times worse** than bandwidth!

That's like having a long distance service that charges

- $150 to make a call;
- 1¢ per minute – after the **first 10 days** of the call.

MPI:
The Message-Passing Interface

Most of this discussion is from [2] and [3].

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++.

MPI Calls

In **C**, MPI calls look like:

```
mpi_error_code = MPI_Funcname(…);
```

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.

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.

WARNING!

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

- C
- C++
- Fortran 77
- Fortran 90

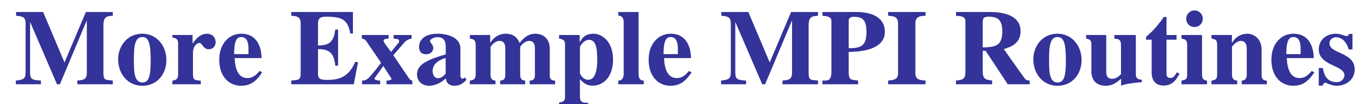
In practice, you should do this:

- 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: Greetings

1. Start the MPI system.
2. Get the rank and 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

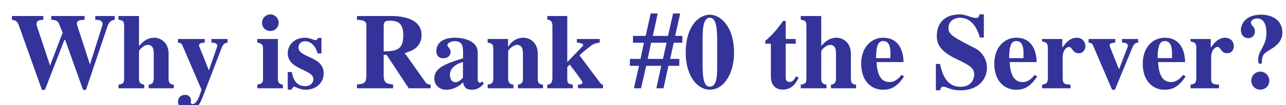
- Every process gets a copy of the executable: ***Single Program, Multiple Data*** (SPMD).
- They all start executing it.
- Each looks at its own rank to determine which part of the problem to work on.
- Each process works **completely independently** of the other processes, except when communicating.

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 Boomer.



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, $N_p \geq 1$.

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 the files and then broadcast the 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 **non-deterministic** order, regardless of the order in which the greeting messages are received.

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 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`	`MPI_CHAR`	`CHARACTER`	`MPI_CHARACTER`
`int`	`MPI_INT`	`INTEGER`	`MPI_INTEGER`
`float`	`MPI_FLOAT`	`REAL`	`MPI_REAL`
`double`	`MPI_DOUBLE`	`DOUBLE PRECISION`	`MPI_DOUBLE_PRECISION`

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
- 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. 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?

```
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 */
```

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***.

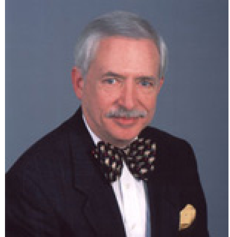
Rule of Thumb for Hiding

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).

OK Supercomputing Symposium 2015



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

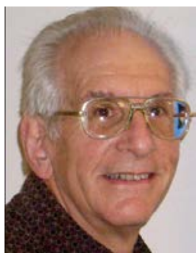
2006 Keynote:
Dan Atkins
Head of NSF's
Office of
Cyberinfrastructure

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

2012 Keynote:
Thom Dunning
Director
National Center for
Supercomputing
Applications

2013 Keynote:
John Shalf
Dept Head CS
Lawrence
Berkeley Lab
CTO, NERSC

2014 Keynote:
Irene Qualters
Division Director
Advanced
Cyberinfarstructure
Division, NSF

2015 Keynote:
Jim Kurose
NSF
Computer & Information
Science & Engineering
Assistant Director

FREE!
Wed Sep 23 2015 @ OU

Reception/Poster Session
Tue Sep 22 2015 @ OU
Symposium
Wed Sep 23 2015 @ OU

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.